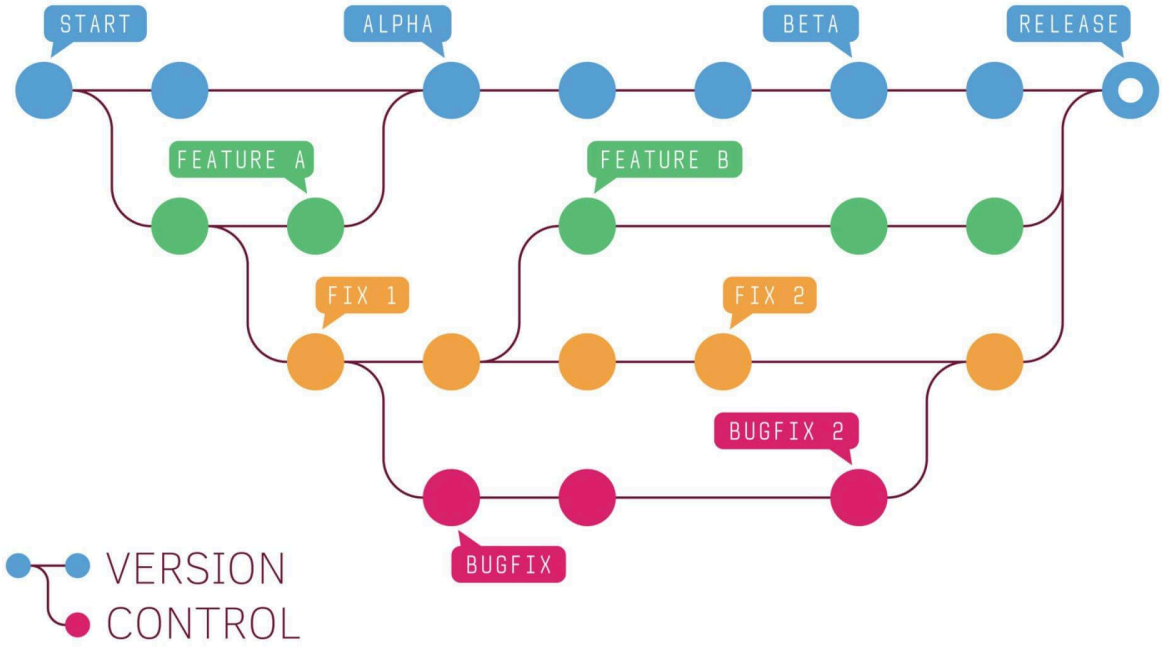
TÓPICO 14 - CONTROLE DE VERSÃO LIMPO

Clean Code - Professor Ramon Venson - SATC 2026.1

O que é Controle de Versão?

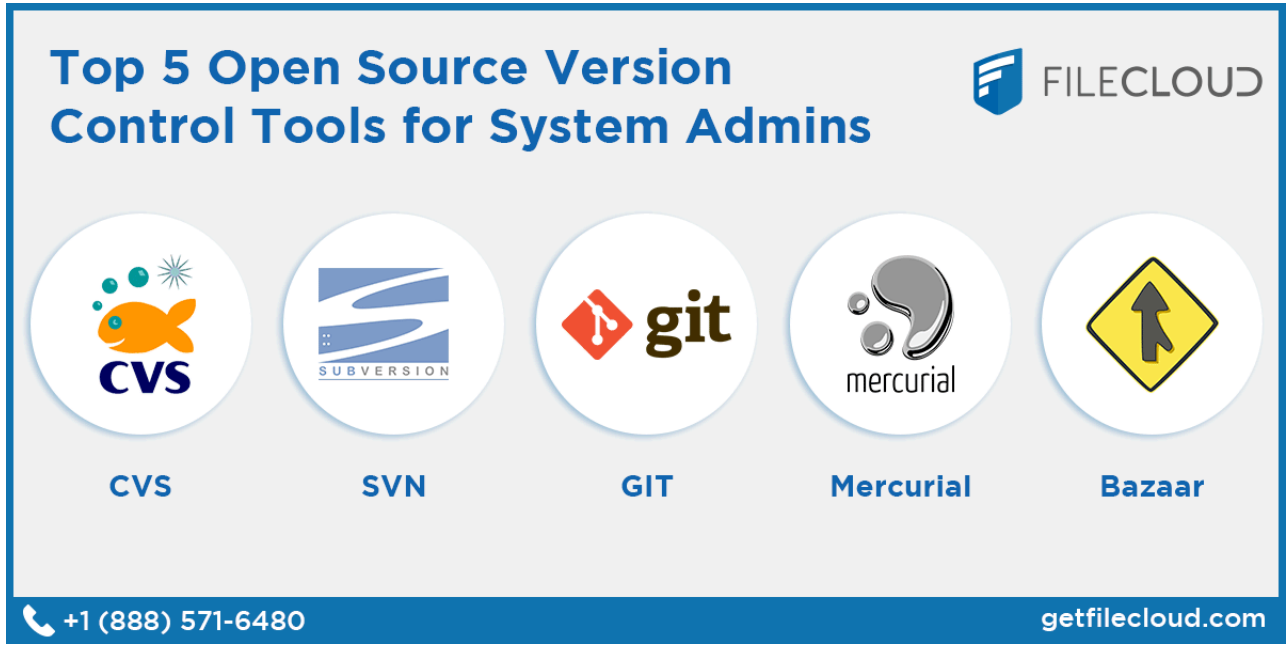
O controle de versão é o processo de rastrear e gerenciar as mudanças em um conjunto de arquivos.



Sistemas de Controle de Versão

Outros sistemas de controle de versão incluem:

- SVN (Subversion)
- Mercurial
- CVS
- Bazaar
- Perforce



Git

O Git é um sistema de controle de versão distribuído, criado por Linus Torvalds para o desenvolvimento do kernel Linux, em 2005.

Serviços de Hospedagem de Código

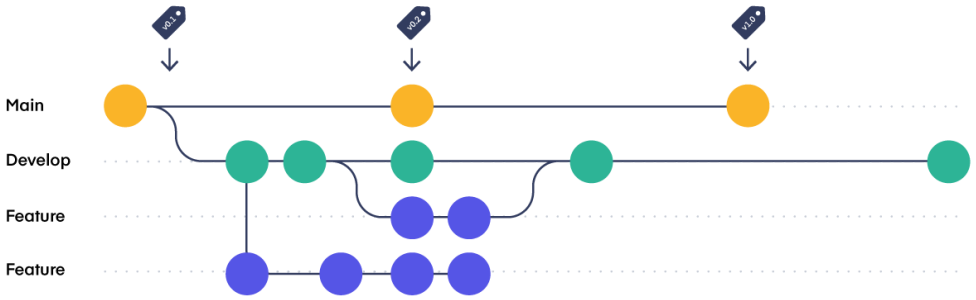
Os serviços de hospedagem são plataformas que permitem que os desenvolvedores armazenem, compartilhem e colaborem em projetos de software.

São frequentemente confundidos com as ferramentas de controle de versão.

Dicas

> Legal. Como nós usamos?

> Nenhuma ideia. Apenas memorize esses comandos de terminal e digite eles pra sincronizar. Se você receber erros, salve seu projeto



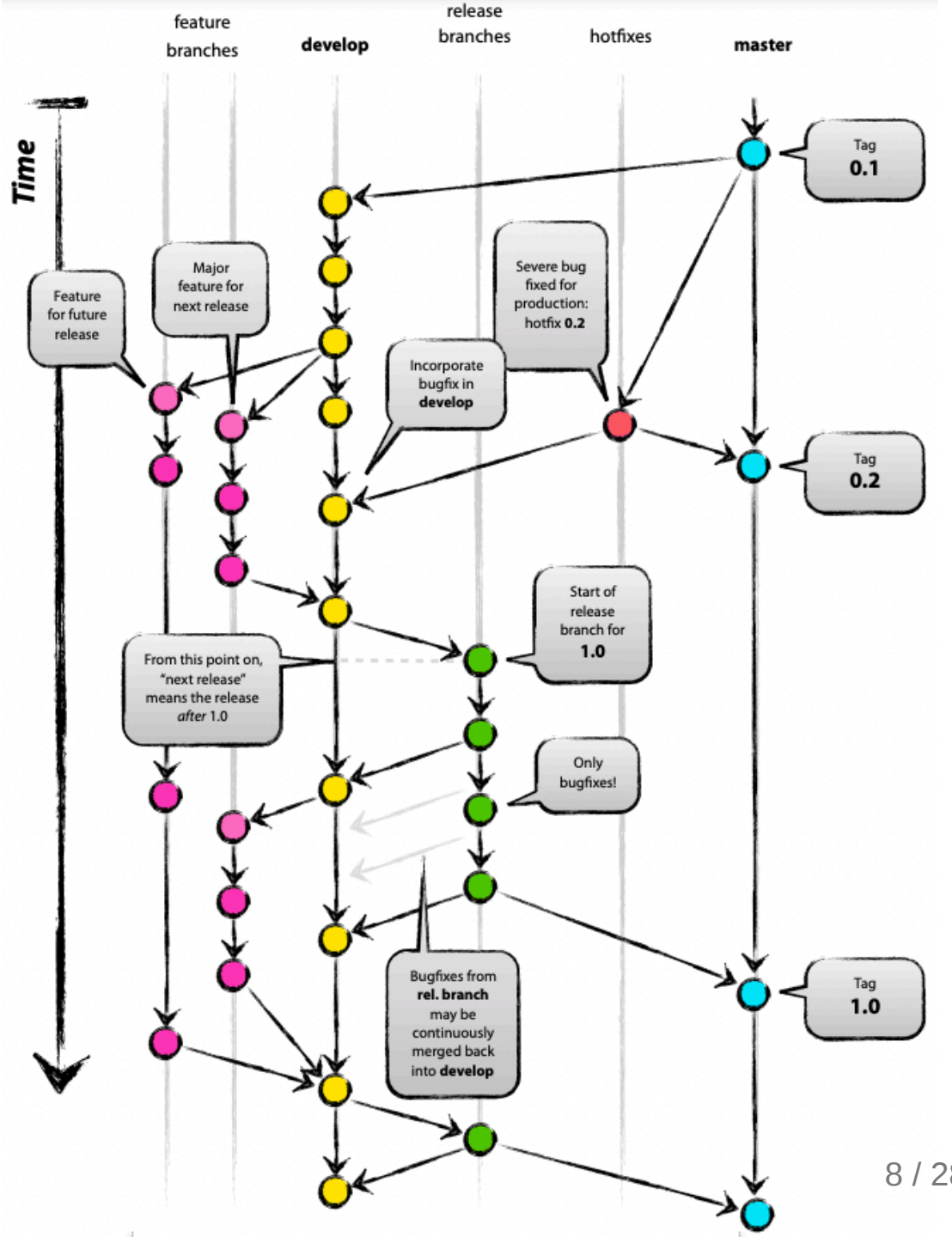
Utilize uma estratégia de branching

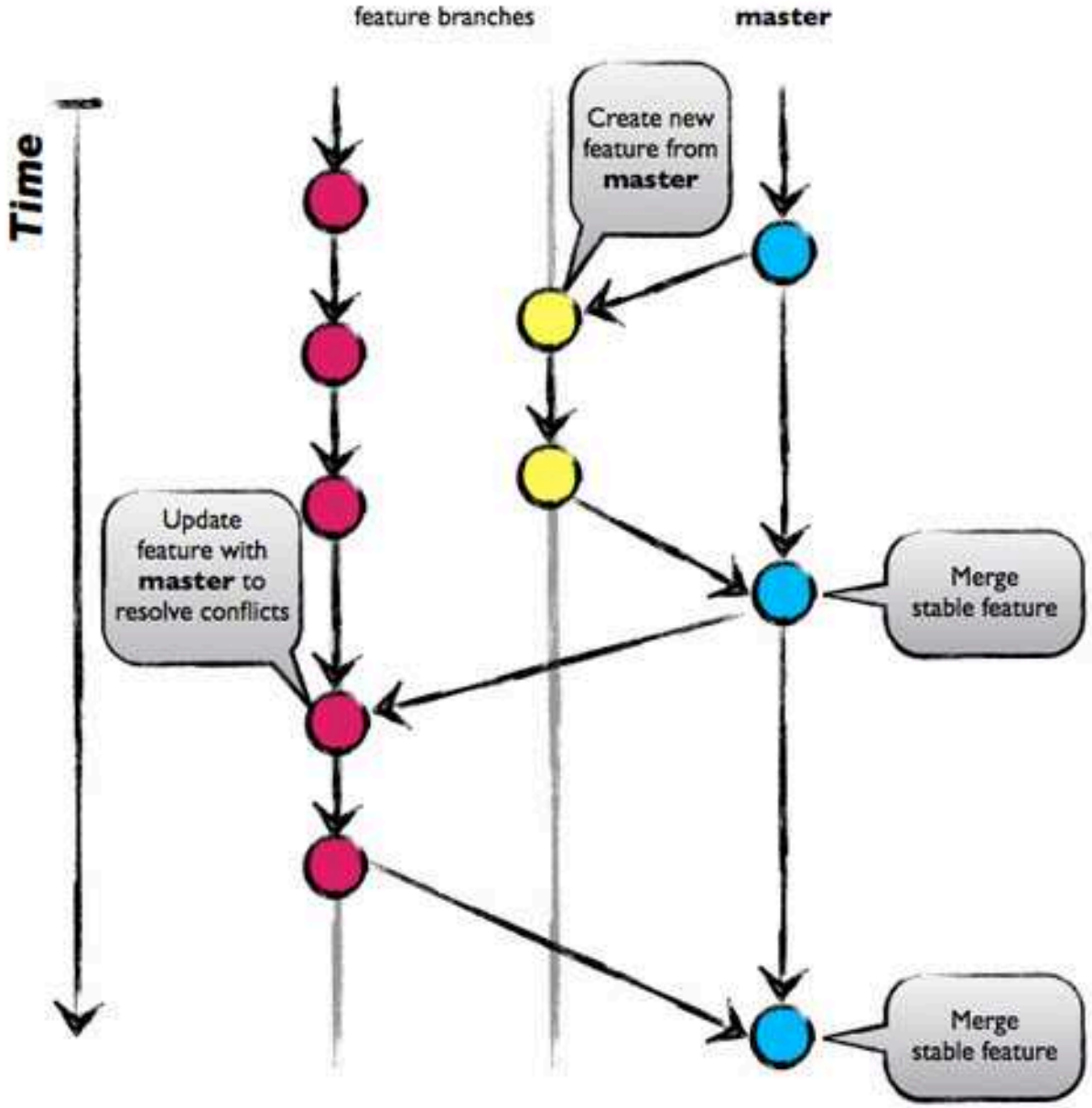
Usar uma estratégia de branching para organizar o fluxo de trabalho.

- `main`: versão estável do software.
- `develop`: versão em desenvolvimento.
- `feature/auth-screen`: nova funcionalidade.
- `hotfix/crash-fix`: correção de bugs.

GitFlow

Os desenvolvedores trabalham em uma ramificação separada, `feature/` ou `hotfix/`, que são integradas à ramificação `develop`. Após os testes, a ramificação `develop` é mesclada à ramificação `main`.





GitHub Flow

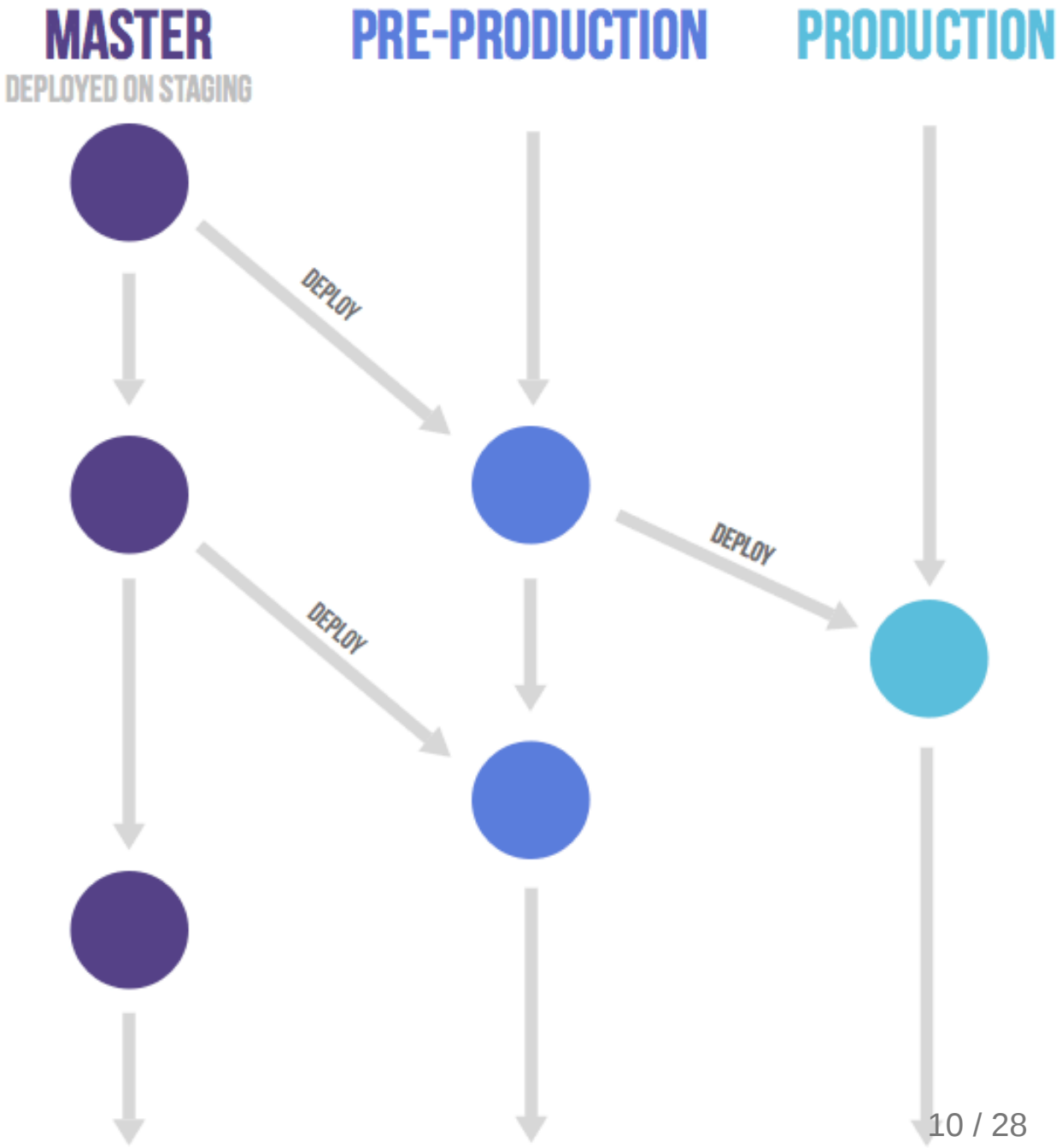
Alternativa simplificada, removendo o branch `develop` e garantindo que o `main` sempre contenha uma versão estável de produção.

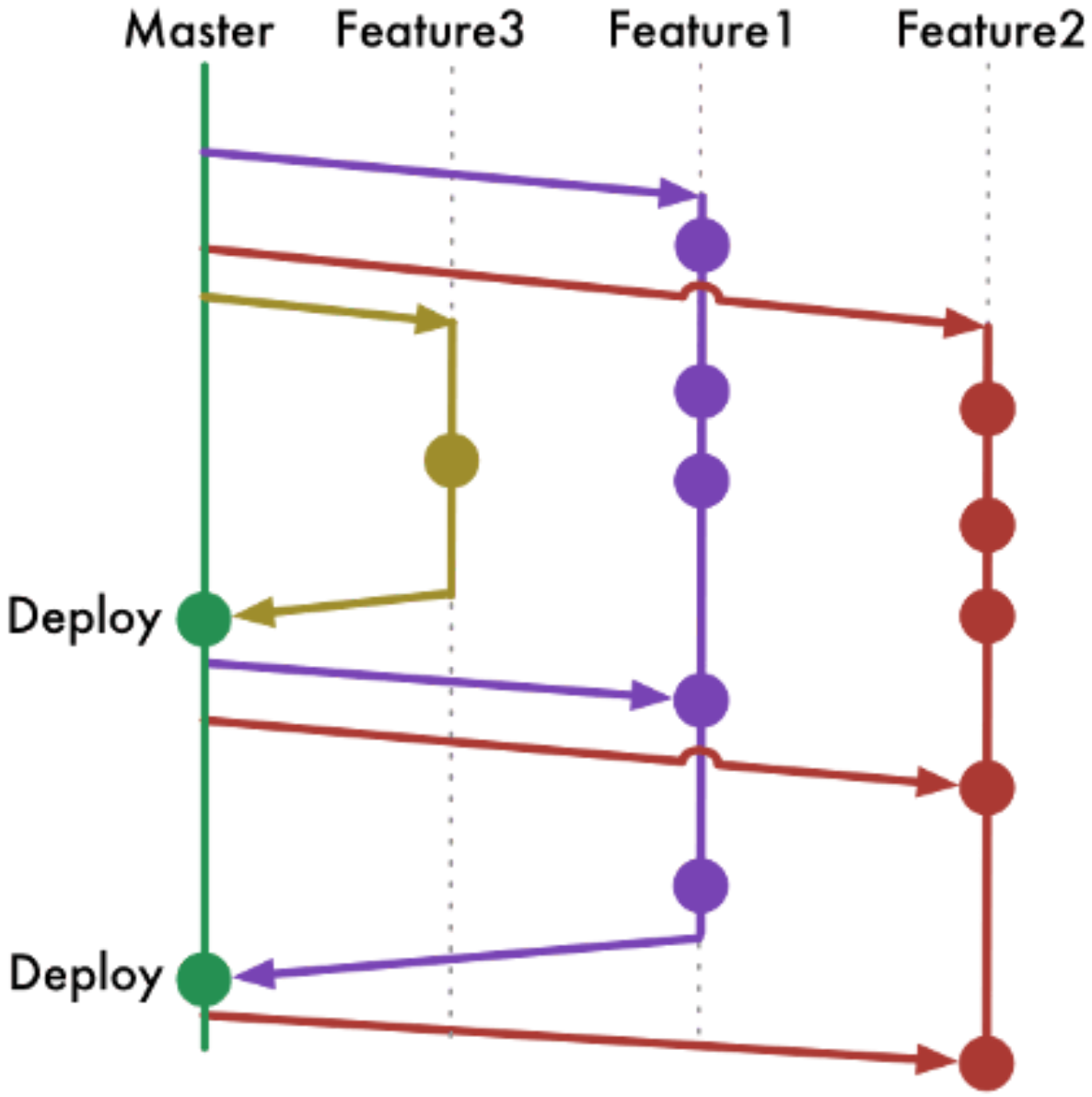
Essa estratégia não permite múltiplas versões de produção.

GitlabFlow

O GitLab Flow é uma estratégia de branching que permite a criação de várias versões de produção.

É a estratégia ideal para projetos que não possuem controle sobre o momento de lançamento de uma versão.





Trunk-based Development

A estratégia de desenvolvimento trunk-based é uma abordagem de desenvolvimento de software que permite que os desenvolvedores trabalhem em uma única ramificação principal, chamada de trunk.

Característica	GitFlow	GitHub Flow	GitLab Flow	Trunk-Based Development
Complexidade	Alta	Baixa	Média	Baixa
Ideal para	Projetos grandes, com lançamentos	Deploys contínuos e simples	Fluxos personalizados com DevOps integrado	Deploy contínuo, times ágeis
Frequência de merge no main	Baixa (após releases)	Alta (a cada PR aprovado)	Alta (em ambientes integrados)	Muito alta (várias vezes ao dia)

Caso de Estudo: Godot's Workflow

- `master`: todas as alterações são mescladas aqui, o que pode incluir atualizações instáveis.
- `3.x`: ramificação de desenvolvimento para a versão `3.x`
- `3.2`: ramificação estável para a versão `3.2`

Todas as contribuições externas são feitas na ramificação master e retro-portadas para as ramificações de desenvolvimento.

Apenas contribuições de manutenção (*hotfixes*) são mescladas diretamente em ramificações estáveis.

Vantagens e Desvantagens

👍 Vantagens	👎 Desvantagens
Suporta versões paralelas de desenvolvimento	Alta complexidade de manutenção
Clara separação entre desenvolvimento e versões estáveis	Requer gerentes experientes para decidir o que pode ser portado entre múltiplas versões
Permite que usuários permaneçam em versões mais antigas	Não é ideal para pequenos times ou projetos rápidos

Não ignore o `.gitignore`

O arquivo `.gitignore` é um arquivo de texto com uma lista de arquivos e pastas desnecessários para o controle de versão.

> Utilize o site gitignore.io para gerar o arquivo.

```
* 7d0fc3e typo
* 8fc509a more changes
* efe5fc5 add test
* 447c5a0 updates
* 1189cf0 update condition
* 68abca0 updates
* 29f73ed more changes
* cefaa18 add file
```

Escreva mensagens de commit significativas

As mensagens de commit são uma forma de documentar as alterações feitas em um repositório Git. Uma boa mensagem de *commit* deve:

- Descrever o que mudou
- Explicar porque mudou
- Usar linguagem clara e direta

Commit Early, Commit Often

Cada commit deve ser entendido como uma modificação atômica, parte de uma funcionalidade. Dessa forma, fica fácil identificar problemas e reverter alterações problemáticas.

✅ Use isso:

```
git commit -m "Implementa a interface gráfica para o login"
git commit -m "Conecta a tela de cadastro à API"
git commit -m "Corrige erros de validação no formulário de cadastro"
```

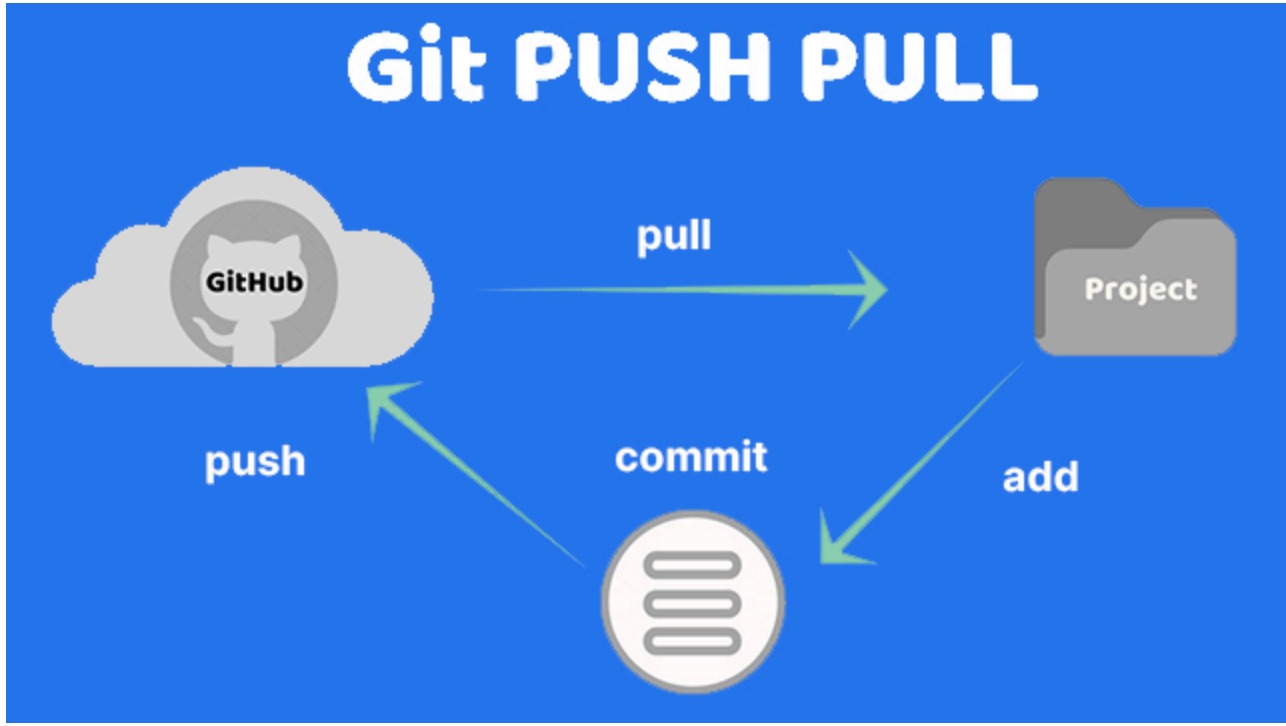
❌ Ao invés de:

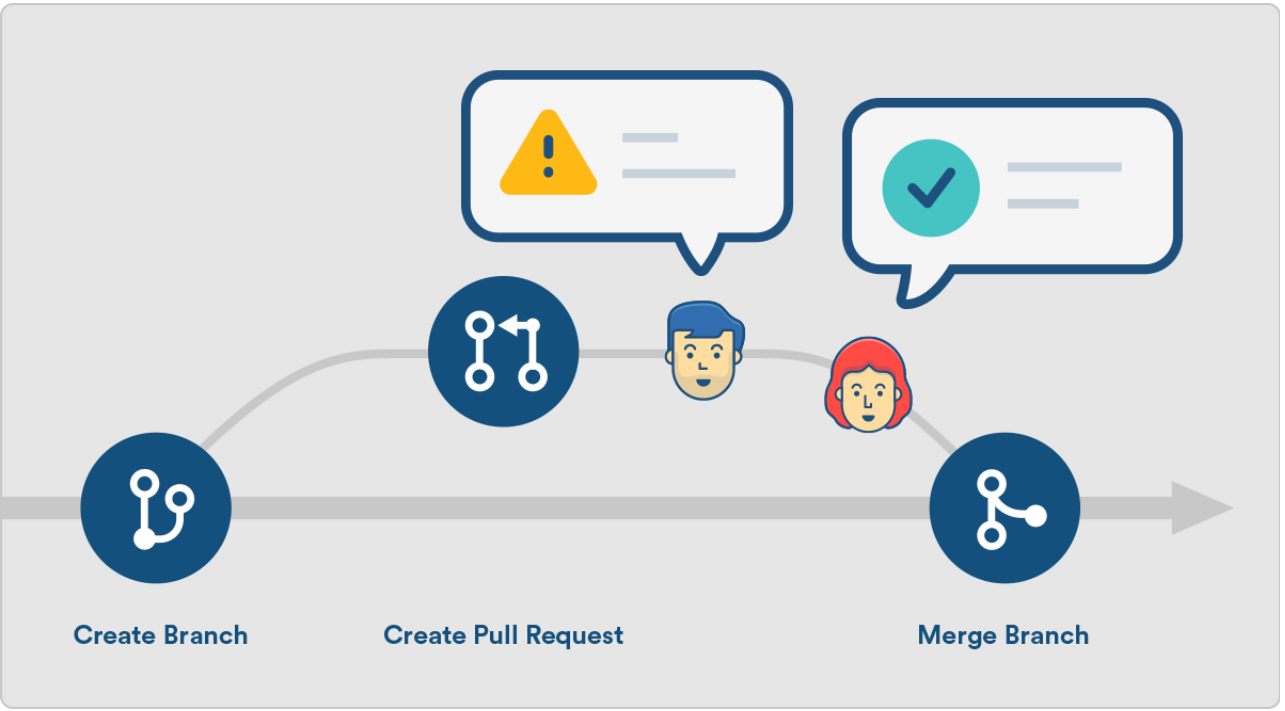
```
git commit -m "Atualiza o aplicativo"
```

`pull` antes de `push`

Antes de enviar suas alterações para o repositório remoto, é importante garantir que seu repositório local esteja atualizado com a versão mais recente.

Usar o comando `git pull` ou `git pull --rebase` permite que você obtenha as alterações mais recentes do repositório remoto e mescle-as com suas alterações locais.





Use *Pull Requests*

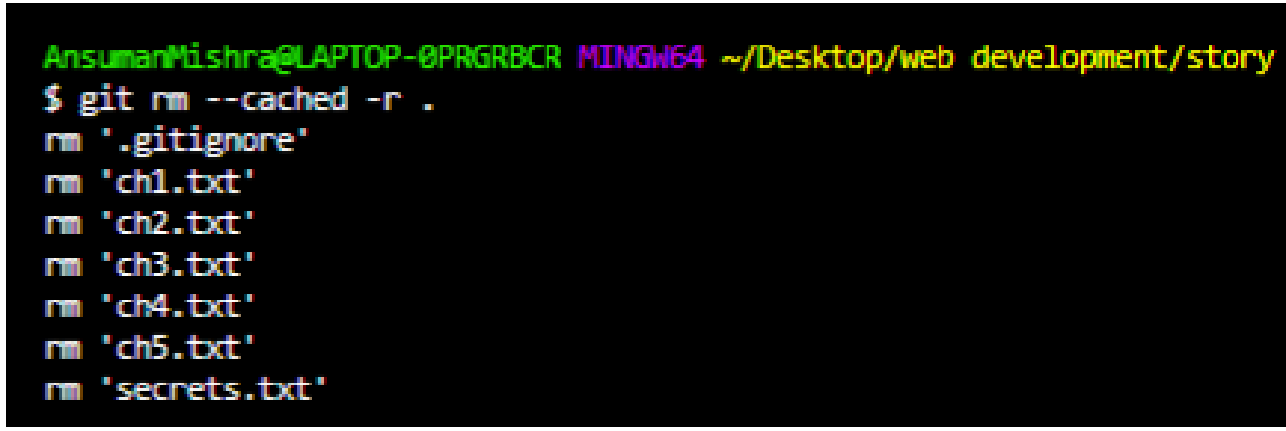
Ao invés de enviar alterações diretamente para o repositório remoto, use *Pull Requests* ou *Merge Requests*. Isso permite:

- Avaliar as alterações antes de mesclar
- Discutir as alterações com outros desenvolvedores
- Evitar conflitos de mesclagem

Versione apenas o necessário

Arquivos desnecessários podem aumentar o tamanho do repositório e tornar a colaboração mais difícil.

Compilações, logs e arquivos temporários devem ser ignorados.



Escrevendo Boas Mensagens de Commit

- Separe a mensagem em um título e uma descrição usando uma linha em branco;
- Limite o título a 50 caracteres;
- Capitalize o título;
- Use o imperativo;
- Não termine a mensagem com um ponto;
- Use o corpo da mensagem para explicar o que e por que você fez as alterações.

Verbos

- Use tempo verbal imperativo;
- Apenas um verbo por commit;
- Use padrões como o Conventional Commits, que permitem a geração automática de CHANGELOGs e automatização de processos;

Verbos comuns em inglês

- `add`: adiciona uma nova capacidade ao sistema;
- `remove` ou `cut`: remove uma capacidade do sistema;
- `fix`: corrige um problema;
- `update`: atualiza uma funcionalidade;
- `refactor`: não altera NENHUMA funcionalidade, apenas a estrutura do código;
- `reformat`: altera a formatação do código;
- `rename`: renomeia um arquivo ou diretório;
- `document`: atualização na documentação do sistema;
- `make`: atualiza processo de building ou infraestrutura;
- `bump`: atualiza uma versão do software ou de uma dependência;

Semantic Versioning

- Versionamento Semântico (SemVer) é um sistema de numeração de versões que reflete mudanças no código de forma estruturada;
- O objetivo é facilitar o controle de versões e a compatibilidade entre diferentes sistemas e bibliotecas;
- As versões seguem o formato **MAJOR.MINOR.PATCH** (ex.: `1.2.3`);

O que cada número significa?

1. **MAJOR**: Mudanças incompatíveis com versões anteriores;
2. **MINOR**: Adiciona novas funcionalidades de forma compatível com versões anteriores;
3. **PATCH**: Correção de bugs de forma compatível com versões anteriores;

Exemplo de versionamento

Versão	Descrição
1.0.1	Correção de pequenos bugs sem novas funcionalidades
1.1.0	Nova funcionalidade adicionada, compatível com 1.x.x
2.0.0	Alterações que quebram compatibilidade com a versão 1.x.x

Regras do Semantic Versioning

1. **A API pública deve ser definida com precisão**: Você deve deixar claro quais partes do código fazem parte da API pública e serão afetadas pelo versionamento;
2. **Mudança de MAJOR**: Sempre que houver uma mudança que quebre a compatibilidade com a versão anterior;
3. **Mudança de MINOR**: Adição de funcionalidades sem quebrar a compatibilidade existente;
4. **Mudança de PATCH**: Correções de bugs ou pequenos ajustes que não impactam a API pública;

Material de Apoio

- What Are the Best Git Branching Strategies
- Version Control - MIT Class
- Git Merge vs GitRebase
- Best Practices for Git and Version Control
- How to use git practices for beginners
- How to write a git commit message
- Conventional Commits
- git commit message
- Semantic Versioning