

Minicurso – Git e GitHub

Victor Machado da Silva, MSc
victor.silva@professores.ibmec.edu.br



Apresentação do curso

Aula 1: Introdução

- Apresentação
- O que é versionamento?
- Breve história do Git
- Primeiros conceitos
- Instalação e configuração do Git
- Criação de conta no GitHub
- Criando o primeiro repositório!

Aula 2: Conceitos básicos

- Sobre controle de versão
- Áreas de trabalho: diretório de trabalho, área de preparação, repositório
- Comandos essenciais: ``init``, ``add``, ``commit``
- Criando o primeiro repositório local

Aula 3: Branches

- Introdução sobre branches
- Criando e alternando entre branches
- Comandos ``branch``, ``checkout``, ``merge``
- Resolvendo conflitos de merge de forma básica
- Criando e mesclando branches simples

Aula 4: GitHub

- Sobre repositórios remotos
- Conectando um repositório local a um remoto
- Clonando repositórios remotos
- Comandos ``push`` e ``pull``
- Clonando um repo, alterando e enviando de volta

Aula 5: Colaboração

- Pull Requests
- Configurando regras de proteção de branches
- Criando um PR e solicitando revisão
- Fusão de PRs
- Criando um PR e colaborando com colegas

Aula 6: Conflitos

- Sobre resolução de conflitos
- Causas comuns de conflitos
- Utilização de ferramentas para resolver conflitos
- Atividade prática: simulação de conflitos e suas resoluções

Aula 7: GitHub Issues

- Utilização do Issues para gerenciar tarefas
- Organização de um projeto com o GitHub Projects
- Labels e Milestones
- Criando issues e organizando um projeto

Aula 8: Git Ignore

- O que é e como usar o arquivo ``.gitignore``
- Boas práticas de organização de repositórios
- Como criar um ``.gitignore``
- Como incluir os padrões a serem ignorados

Aula 9: Git Rebase

- Introdução ao ``git rebase``
- Alterando o histórico de commits
- Comando ``git cherry-pick``
- Atividade prática: reorganizando commits com ``rebase`` e ``cherry-pick``

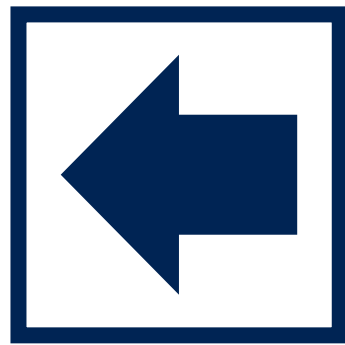
Aula 10: Workflows

- Sobre fluxos de trabalho, uso de branches e trabalho colaborativo
- Git Flow
- GitHub Flow
- Trunk-based development

Clique nas aulas para ir direto para o conteúdo!

Apresentação - Links interessantes

- [Link para download do Git](#)
- [Documentação oficial](#)
- [Jogo “Learn git branching”](#)
- [Curso sobre versionamento no YouTube](#)
- [Vídeo avançado sobre estratégias de branching](#)



Aula 1: Introdução

Introdução

Talvez você reconheça esse nome por causa de sites como *GitHub* ou *GitLab*. **Git** é um sistema *open-source* de controle de versionamento.

Versionar projetos é uma prática essencial no mundo profissional e, em particular, na área de tecnologia, para manter um **histórico de modificações** deles, ou poder reverter alguma modificação que possa ter comprometido o projeto inteiro, dentre outras funcionalidades que serão aprendidas na prática.

Os projetos são normalmente armazenados em **repositórios**.

O que é Controle de Versões?

Controle de Versões é um sistema de grava mudanças aplicadas em um arquivo ou um conjunto de arquivos ao longo do tempo, para que o usuário possa lembrar ou recuperar depois.

Na área de tecnologia o controle de versões já é algo consolidado, uma vez que inúmeras alterações são aplicadas em um software durante a sua implementação e manutenção. No entanto, adotar um sistema de controle de versões (VCS, da sigla em inglês) é algo recomendado para qualquer área.

Quando se quer adotar um VCS, normalmente o primeiro passo é trabalhar com um controle local, fazendo cópias dos arquivos e os renomeando com algum padrão (p.ex., incluindo a data ao final do nome do arquivo).

O Git veio para otimizar esse controle de versões, permitindo inúmeras alterações que seriam muito complicadas se fossem feitas manualmente.

Uma breve história do Git

O Git foi criado por Linus Torvalds em 2005 para auxiliar no desenvolvimento do kernel Linux. É um sistema de controle de versão distribuído, o que significa que cada membro da equipe possui uma cópia completa do repositório, permitindo trabalhar offline e facilitando a colaboração.

O sistema foi projetado de forma a atender aos seguintes requisitos:

- Velocidade
- Design simples
- Suporte forte para desenvolvimento não-linear, com milhares de atividades sendo realizadas em paralelo
- Completamente distribuído, permitindo o acesso de qualquer lugar
- Eficiente no suporte a grandes projetos

Primeiros conceitos

Repositório: É o local onde todas as versões do seu projeto são armazenadas. Pode ser local ou remoto (ou ambos).

Commit: Uma snapshot (imagem instantânea) de todas as alterações feitas no código em um determinado momento. Cada commit tem uma mensagem que descreve as mudanças.

Branch: É uma linha de desenvolvimento separada que permite trabalhar em recursos diferentes sem interferir no código principal.

Merge: É o processo de combinar as alterações de um branch em outro. Quando você deseja incorporar as alterações feitas em um branch em outro, você realiza uma mesclagem.

Configuração inicial

Faça a instalação usual do Git na sua máquina, baixando-o pelo site oficial. Considere todas as opções recomendadas durante a instalação. É recomendado instalar o Git em um diretório fácil de acessar (p.ex., na raiz do diretório C:\).

Em seguida, abra o prompt de comando (ou powershell, gitbash ou outro programa de terminal) e entre com as seguintes opções de configuração:

```
git config --global user.name "Seu Nome"  
git config --global user.email "seu@email.com"  
git config --global user.username "seu_username"  
git config --global core.editor "code --wait"  
git config --global core.autocrlf false
```

Criando o primeiro repo no GitHub

O GitHub é uma plataforma de hospedagem de código-fonte e colaboração. Criar uma conta no GitHub é essencial para compartilhar e colaborar em projetos.

Após criar uma conta em <https://github.com/>, faça o login e siga com as instruções abaixo para criar o primeiro repositório remoto:

- No canto superior direito, clique no ícone "+" e selecione "New Repository".
- Dê um nome ao repositório, adicione uma descrição.
- Escolha a opção "public" e marque a opção "add a README file".
- Clique em "Create repository" para criar o repositório remoto.

Aula 1 - Links interessantes

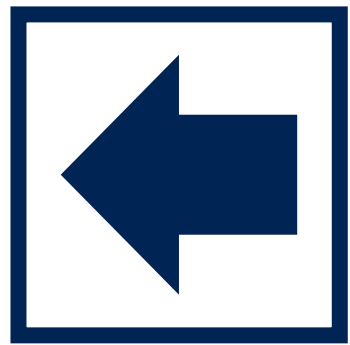
Introdução ao Versionamento de Código: [Artigo](#)

História e Conceitos do Git: [Vídeo](#)

Git e GitHub para Iniciantes: [Artigo](#)

Git e GitHub: Guia Rápido de Configuração: [Vídeo](#)

GitHub: Guia de Criação de Repositórios: [Artigo](#)



Aula 2: Conceitos básicos

O que é Git?

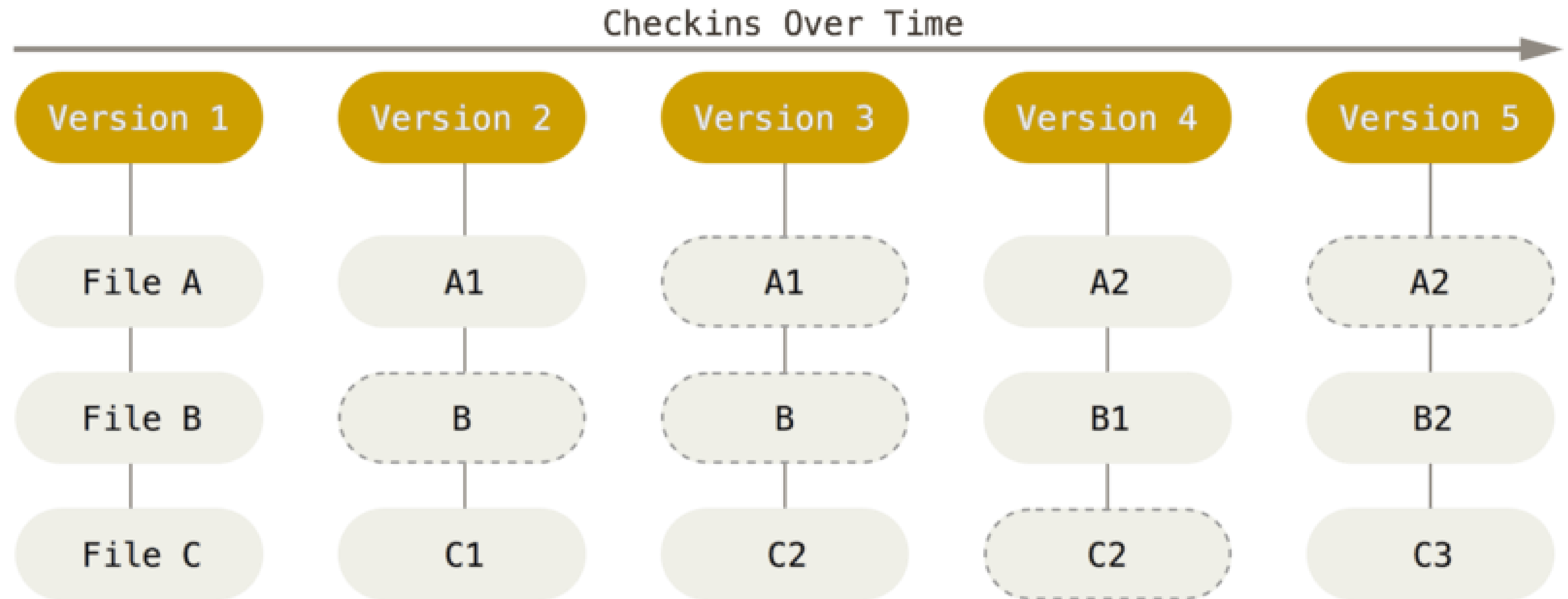
Git funciona pensando que os dados de um repositório compõem uma série de “fotografias” de um sistema de arquivos em miniatura.

No Git, a cada vez que você aplica uma alteração (ou *commit*), ou salva o estado do seu projeto, o Git basicamente tira uma “foto” de como os arquivos do repositório estão naquele momento e então ele armazena uma referência a essa foto.

Para ser eficiente, se os arquivos não foram alterados, o Git não altera os arquivos novamente, apenas um link para a última versão do arquivo armazenada.

Sendo assim, o Git trabalha os dados como um **fluxo de fotografias**.

O que é Git?



O que é Git?

A maior parte das operações no Git precisa apenas de arquivos e recursos locais para operarem, e normalmente nenhuma informação é necessária de outro computador na rede. Isso fornece uma velocidade de operação que outros VCS não possuem. Como cada usuário possui todo o histórico do projeto no computador, a maioria das operações aparenta ser quase instantânea.

Para todos os efeitos, na prática, o Git normalmente só acrescenta informações ao seu banco de dados, nunca removendo informações. É muito difícil, e não recomendado, gerar operações que removam informações, já que essas operações podem afetar o histórico do seu projeto.

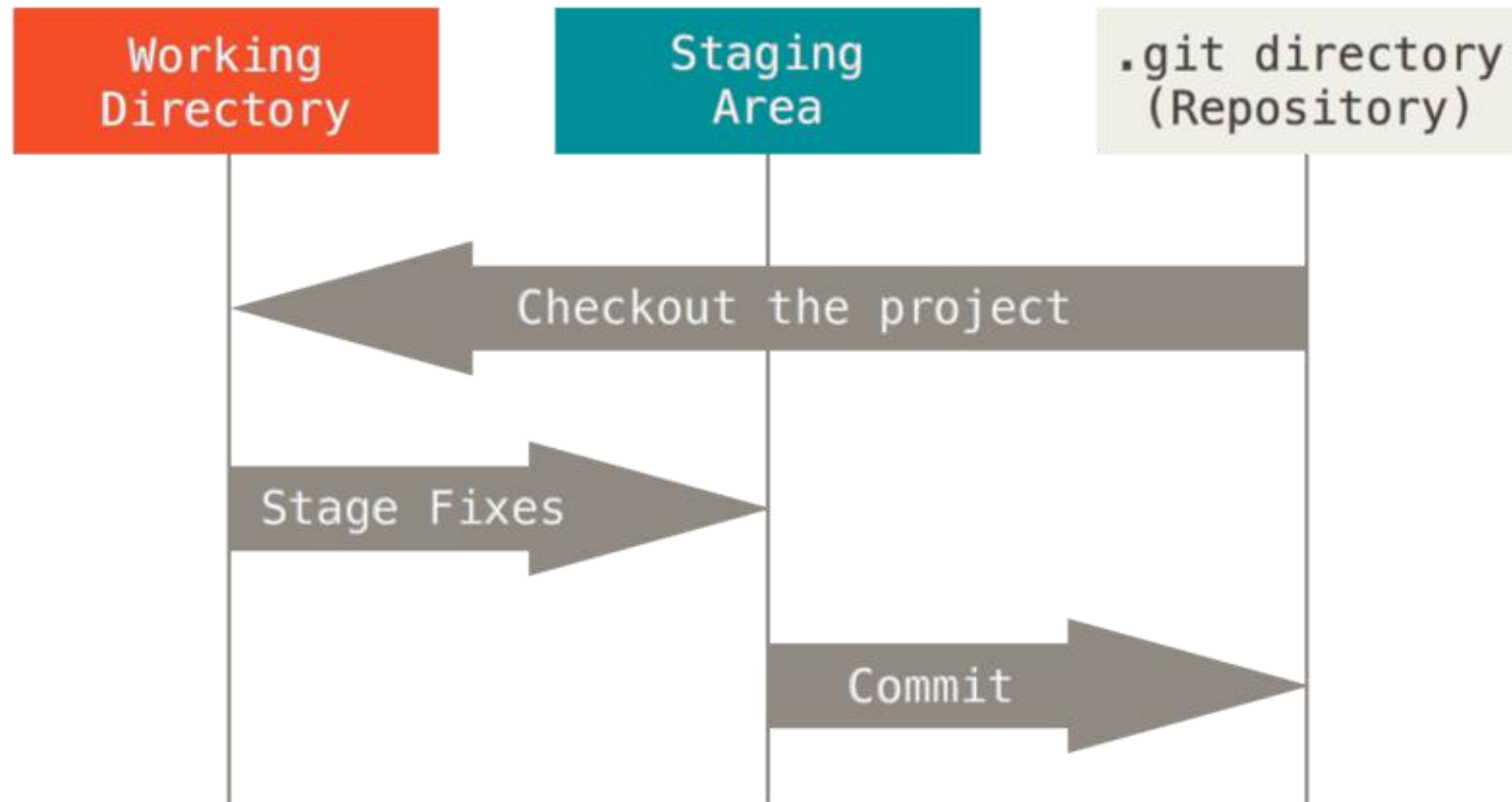
Os três estados

O Git tem três estados principais nos quais os arquivos de um repositório podem se encontrar: **modificado**, **preparado** e **“commitado”**:

- **Commitado** significa que os dados estão armazenados de forma segura em seu banco de dados local;
- **Modificado** significa que você alterou o arquivo, mas ainda não fez o commit no banco de dados;
- **Preparado** significa que você marcou a versão atual de um arquivo modificado para fazer parte do seu próximo commit.

Isso leva a três seções principais de um projeto Git: o diretório Git, o diretório de trabalho e área de preparo.

Os três estados



Comandos principais

Comando	Descrição
git init	Inicializa um repositório git, sem nenhum commit
git status	Indica os status de arquivos modificados, adicionados ou removidos, além de arquivos preparados (staged)
git add <nome_arquivo>	Prepara o arquivo mencionado
git add -u	Prepara todos os arquivos modificados (porém não faz nada com arquivos novos)
git add .	Prepara todos os arquivos (incluindo arquivos novos)
git commit	Faz um commit dos arquivos preparados, sem uma mensagem de commit
git commit -m "mensagem"	Faz um commit dos arquivos preparados, incluindo a mensagem de commit definida
git restore <arquivo>	Desfaz modificações do arquivo que não foi preparado
git restore --staged <arquivo>	Desfaz a preparação do arquivo (porém mantém modificações)

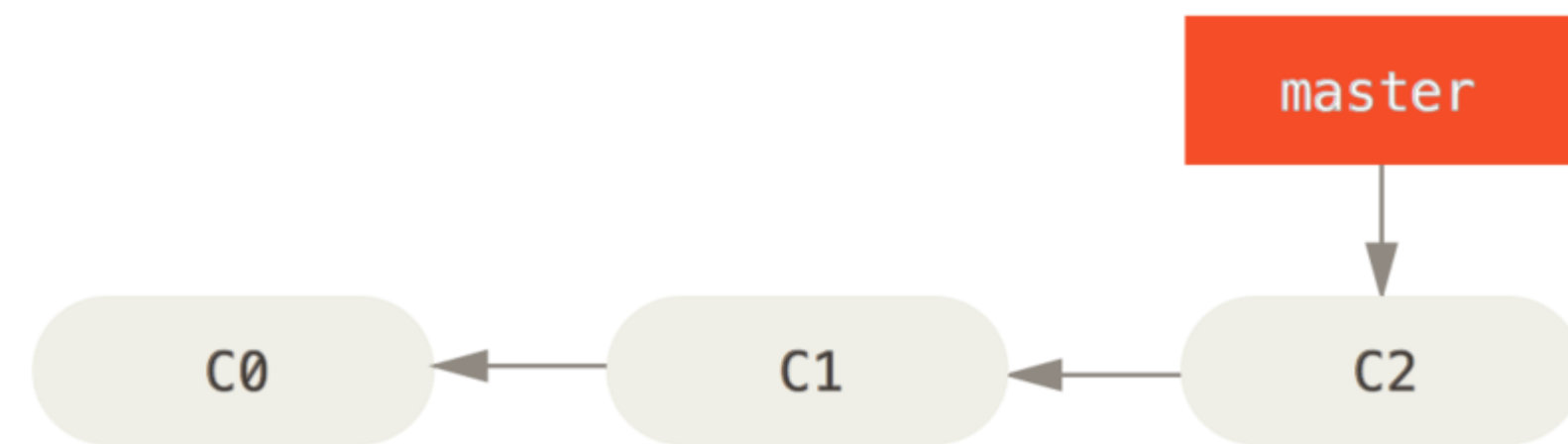


Aula 3: Branches

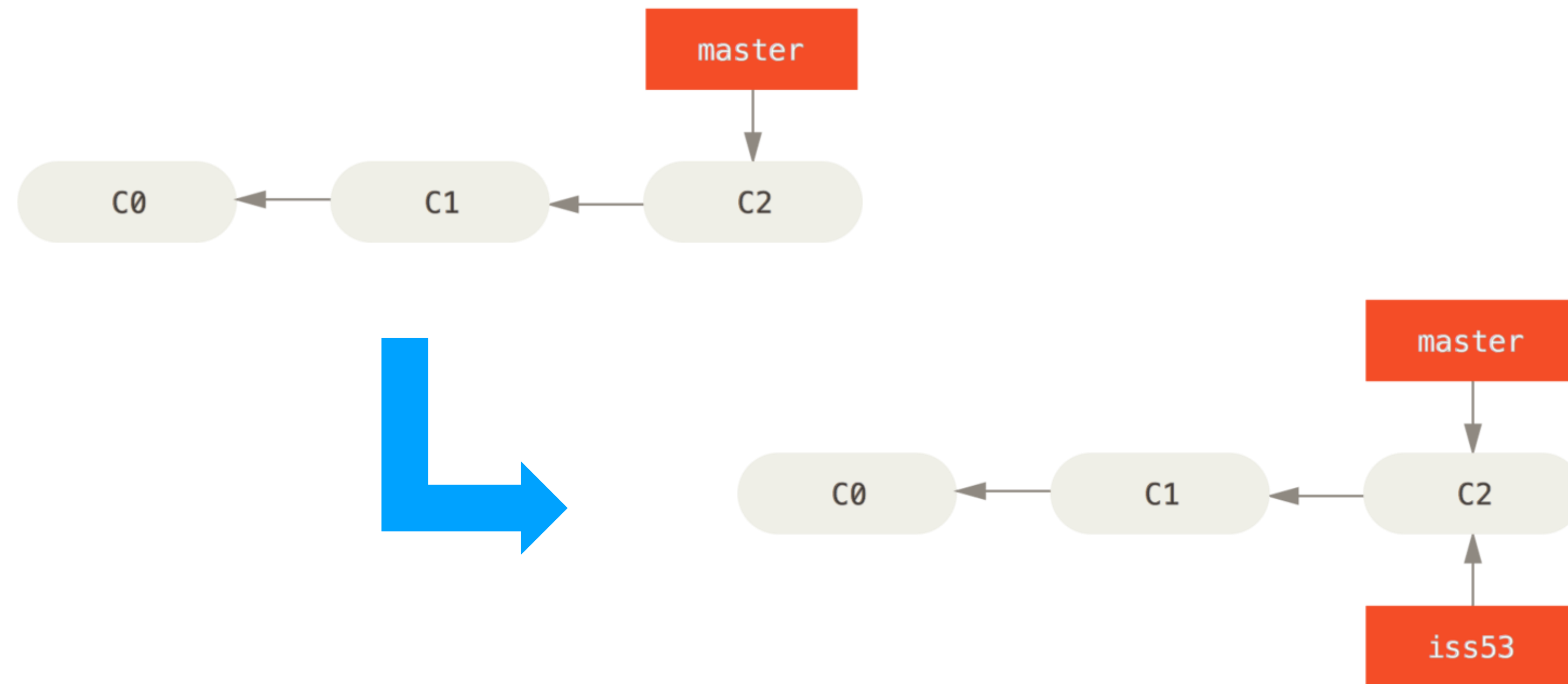
Entendendo os conceitos

Branches (ramificações) são uma característica fundamental no Git que permite criar caminhos de desenvolvimento separados. Cada branch representa uma linha independente de desenvolvimento dentro do repositório, permitindo que você trabalhe em diferentes recursos ou correções de bugs sem interferir no código principal. Isso é especialmente útil em projetos colaborativos, onde várias pessoas podem estar trabalhando em funcionalidades diferentes ao mesmo tempo.

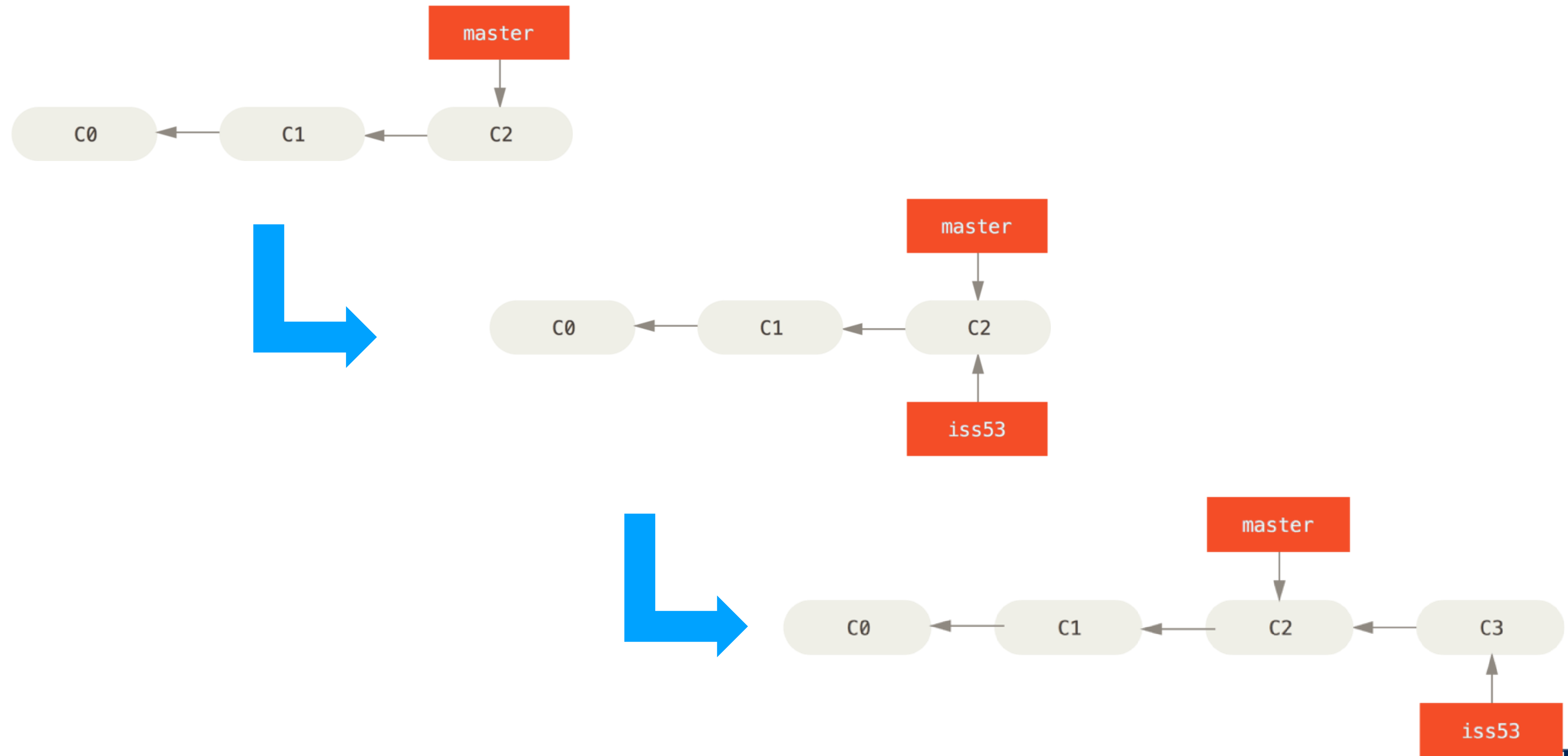
Branches no Git



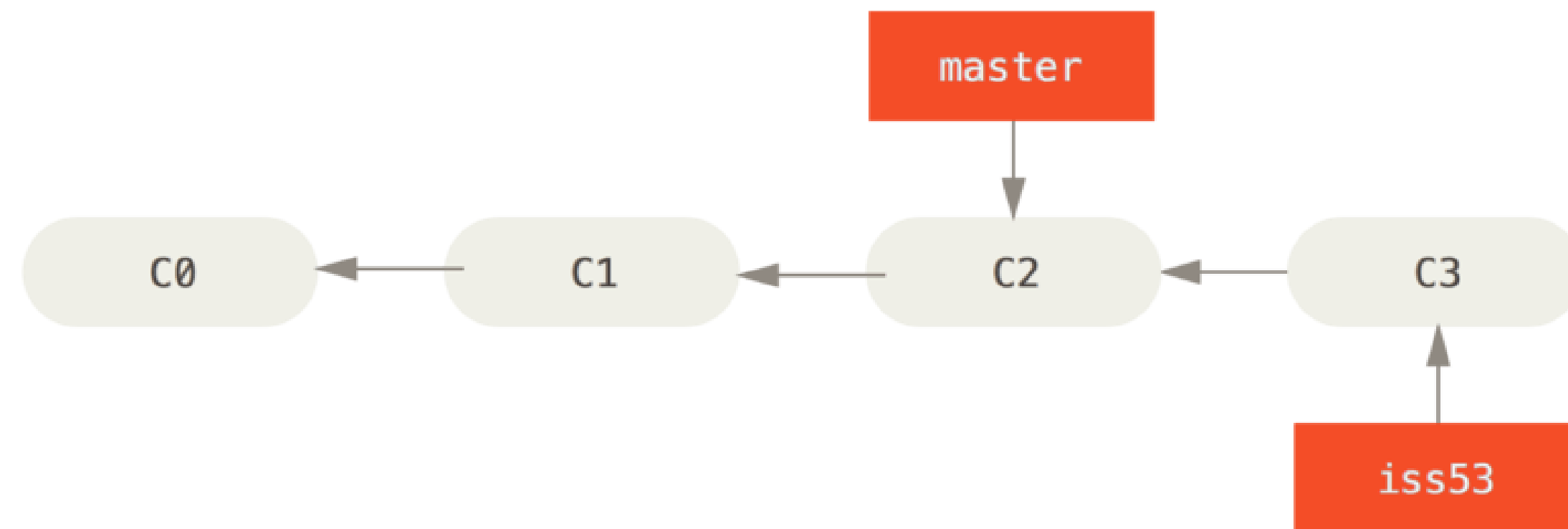
Branches no Git



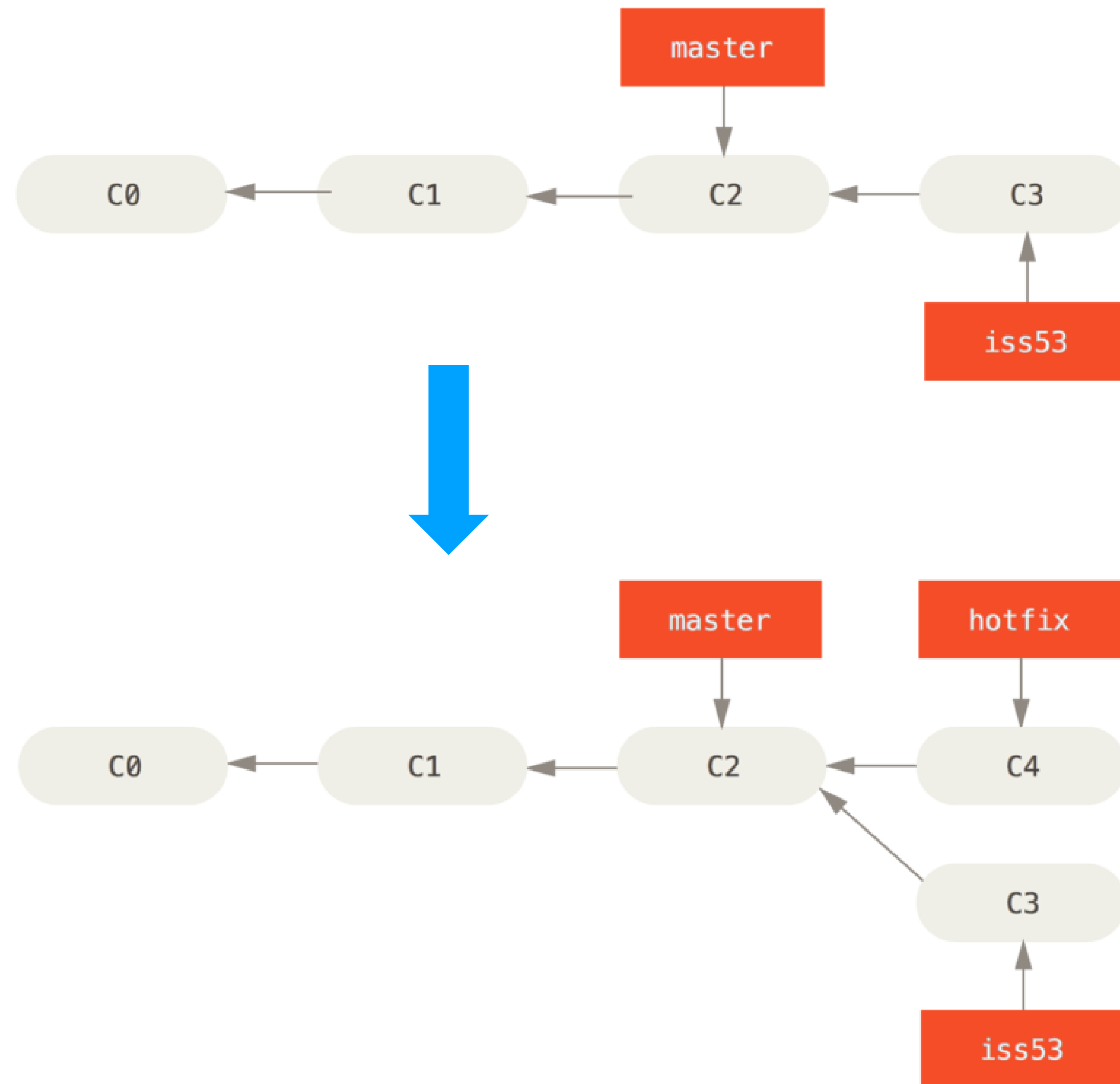
Branches no Git



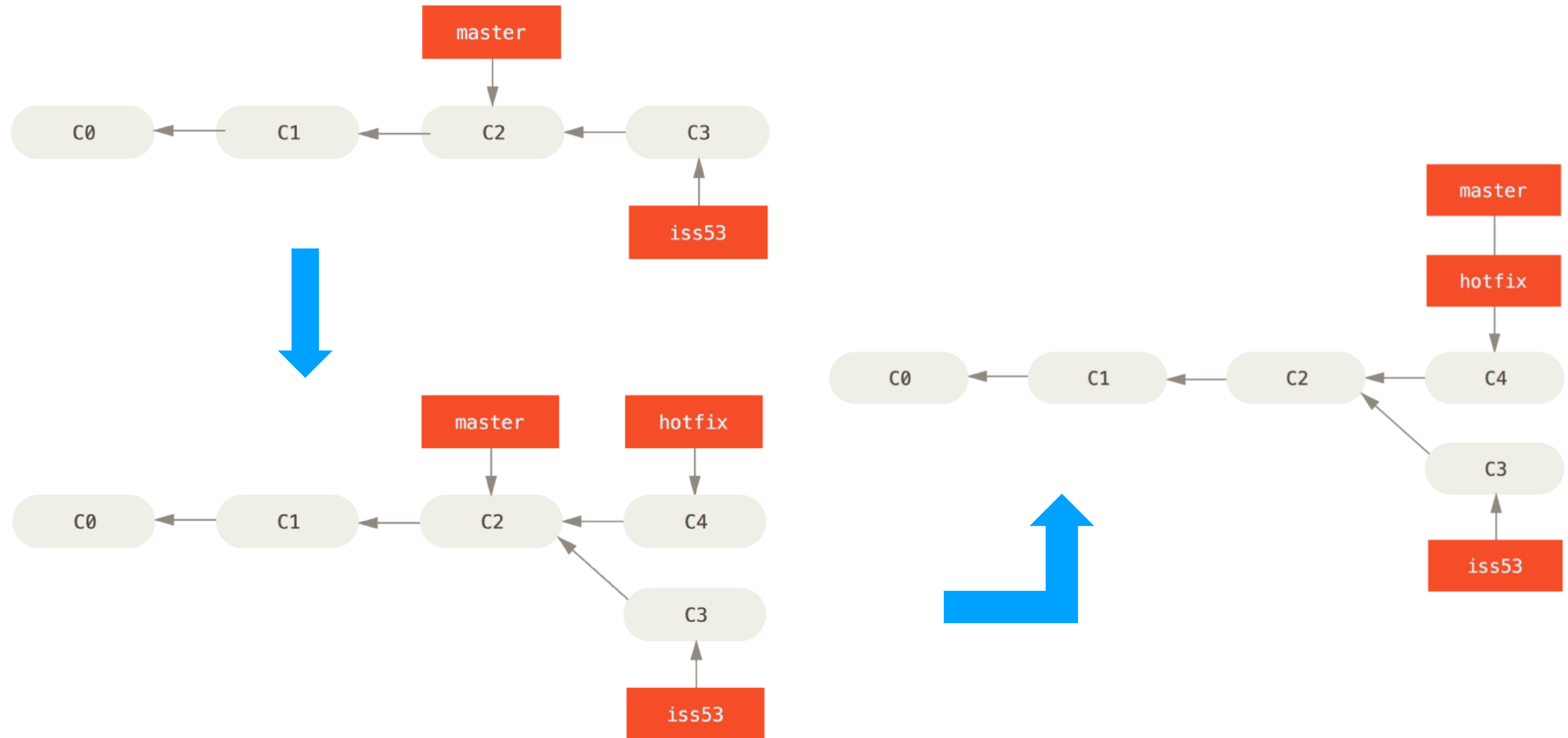
Branches no Git



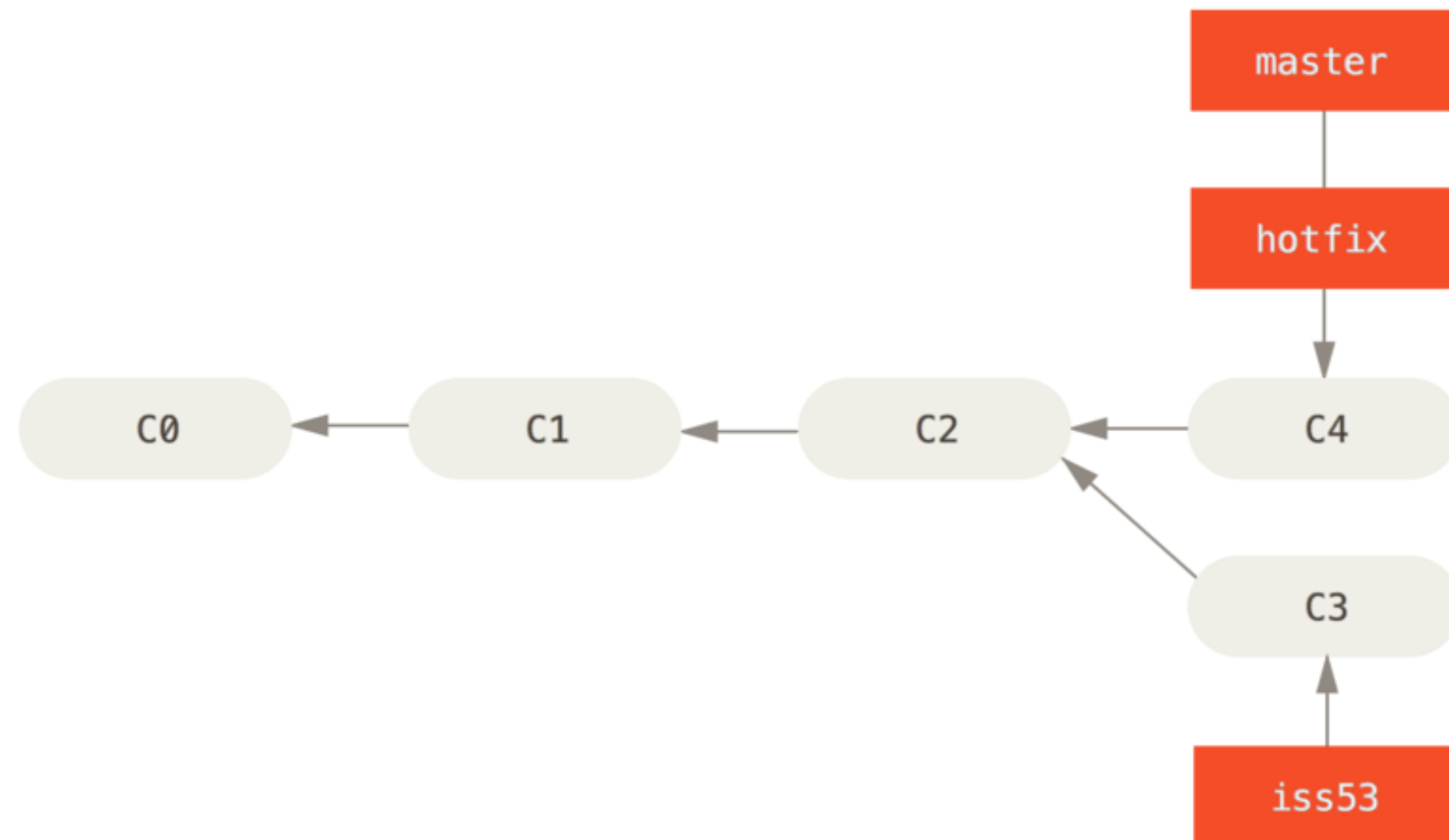
Branches no Git



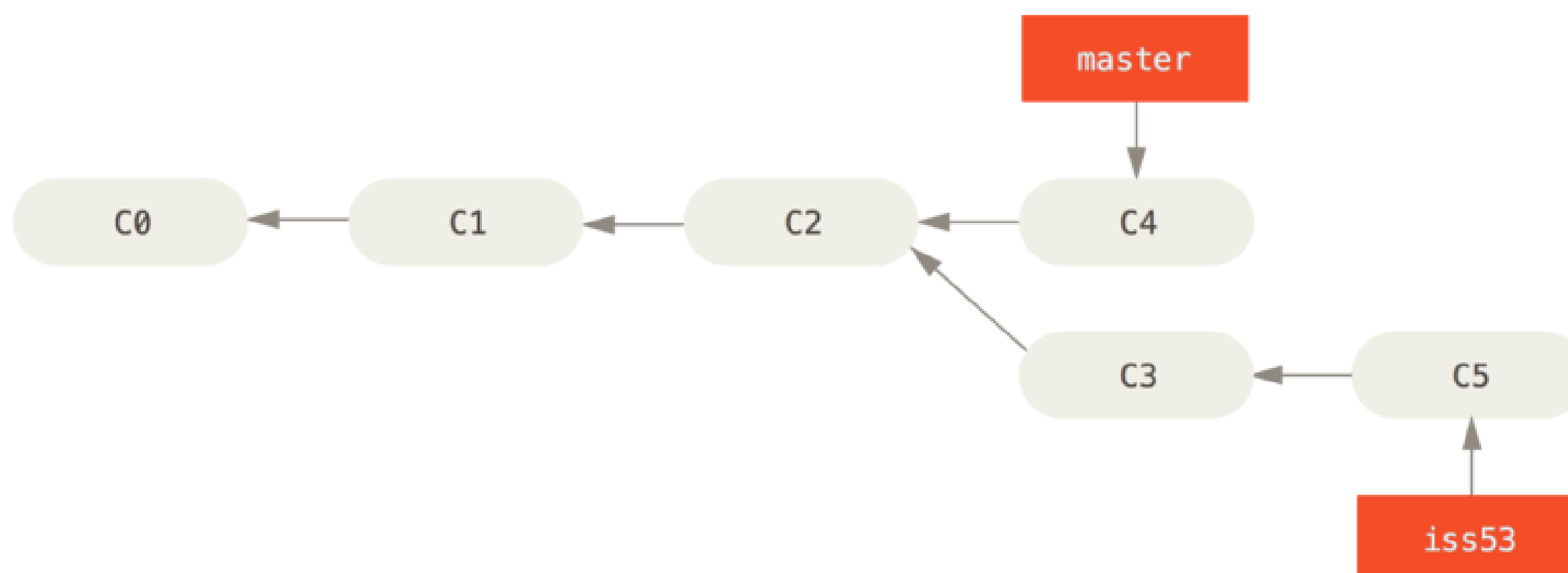
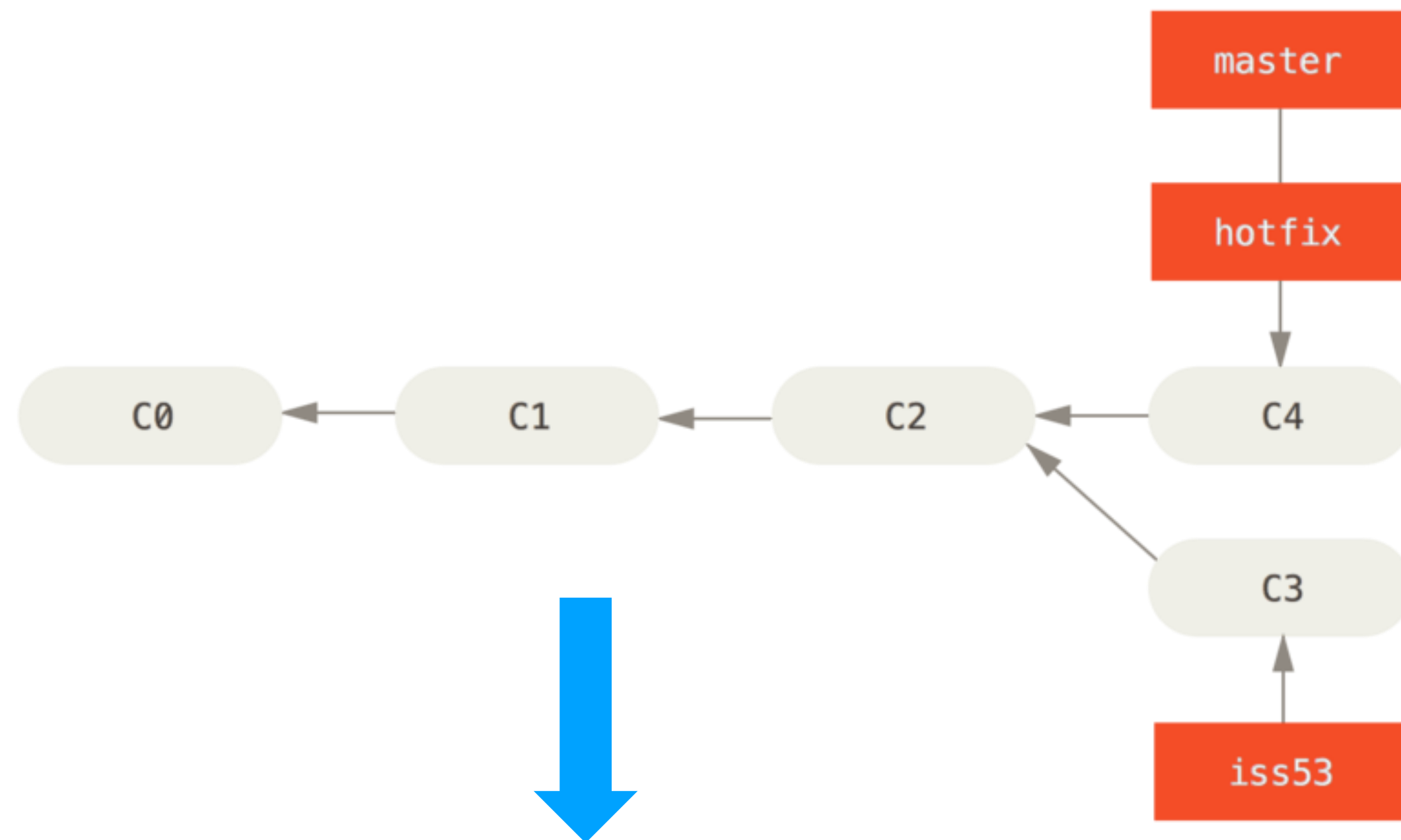
Branches no Git



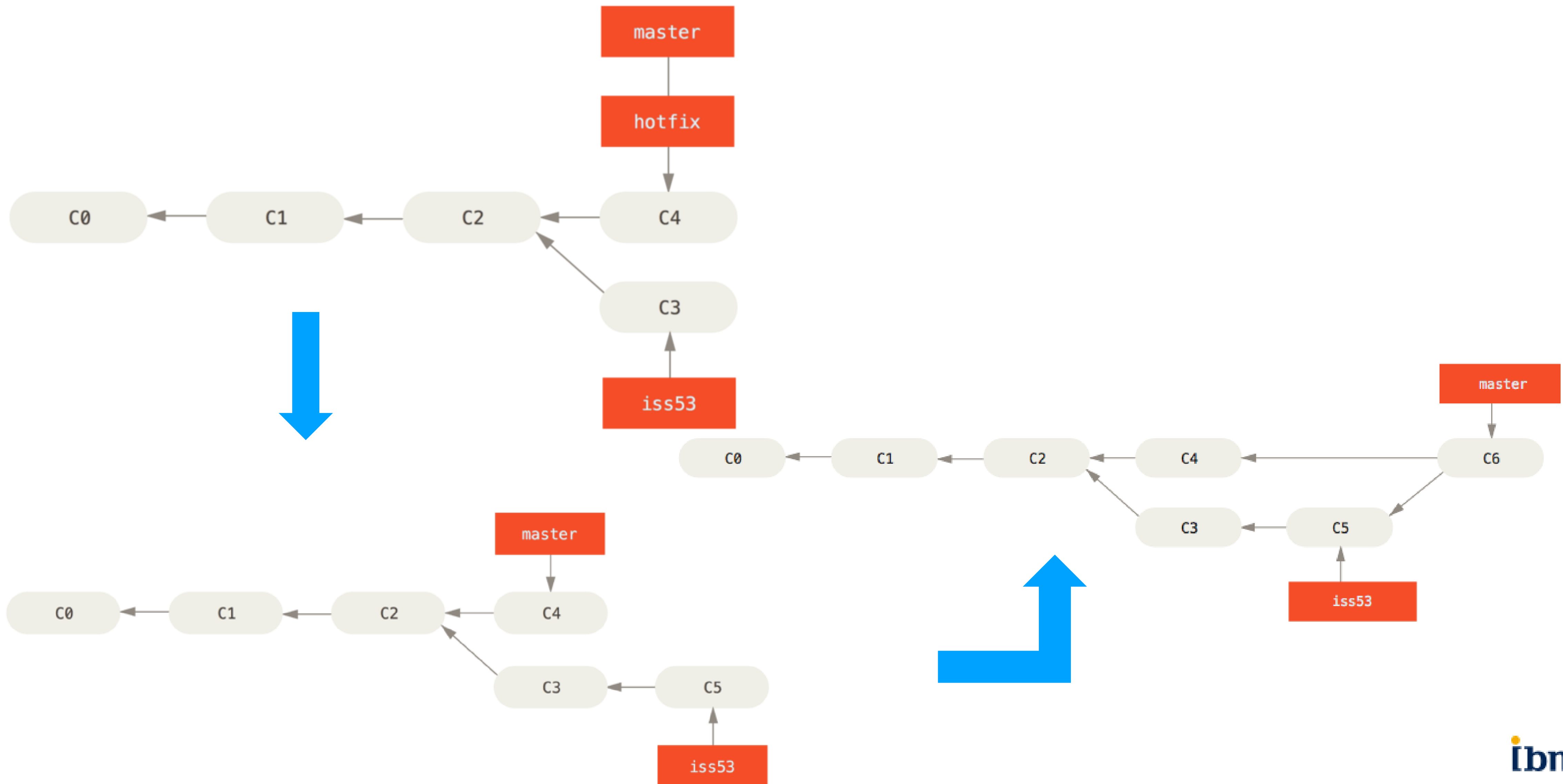
Branches no Git



Branches no Git



Branches no Git



Criando e alternando entre branches

Comando	Descrição
git branch <nome_branch>	Cria um novo branch com o nome indicado
git branch	Lista todos os branches no repositório, destacando o branch atual com um asterisco (*)
git branch -d <nome_branch>	Exclui um branch que já foi mesclado a outro branch
git branch -D <nome_branch>	Força a exclusão de um branch, mesmo que ele tenha alterações não mescladas
git branch -m <antigo> <novo>	Renomeia um branch
git branch -m <novo>	Renomeia o branch atual
git checkout <nome_branch>	Alterna entre branches. Move o HEAD (ponteiro atual) para um branch específico
git checkout -b <nome_branch>	Cria um novo branch (caso ele não exista), e alterna entre branches
git merge <nome_branch>	Combina as alterações do branch listado no branch atual

Resolvendo conflitos de merge

Conflitos de merge ocorrem quando o Git não consegue determinar automaticamente como combinar as alterações de dois branches. Isso pode acontecer quando você e outra pessoa fizeram alterações na mesma parte do código.

Para resolver os conflitos manualmente:

- O Git marcará as áreas com conflito no código-fonte com marcadores especiais.
- Você deve editar o código manualmente, removendo os marcadores e decidindo qual alteração manter.
- Após a resolução do conflito, você deve fazer um novo commit para finalizar o merge.

Fluxo de trabalho básico

- Crie um novo branch com `git branch nome-do-branch`.
- Alterne para o novo branch com `git checkout nome-do-branch`.
- Faça suas alterações no código.
- Faça um commit no novo branch.
- Volte para o branch principal (por exemplo, "master") com `git checkout master`.
- Use `git merge nome-do-branch` para mesclar as alterações do novo branch no branch principal.
- Resolva conflitos, se houver.
- Faça um novo commit após resolver conflitos.
- O novo branch pode ser excluído após a mesclagem, se desejado, usando `git branch -d nome-do-branch`.



Aula 4: GitHub e repositórios remotos

Sobre repositórios remotos

Repositórios remotos desempenham um papel essencial no desenvolvimento de software colaborativo. Eles oferecem às equipes de desenvolvimento a capacidade de compartilhar e colaborar em projetos, independentemente da localização física dos membros da equipe. Esses repositórios são hospedados em servidores acessíveis pela internet, tornando o código acessível a todos os membros autorizados da equipe. O GitHub e o GitLab são exemplos proeminentes de plataformas de hospedagem de repositórios remotos, conhecidos por suas interfaces amigáveis, recursos avançados de colaboração e integração contínua.

Essas plataformas não apenas fornecem um espaço centralizado para armazenar código, mas também oferecem recursos de controle de acesso, acompanhamento de alterações, gerenciamento de problemas e automação de fluxos de trabalho. Elas são amplamente adotadas por desenvolvedores em todo o mundo, permitindo que equipes colaborem eficazmente em projetos complexos, independentemente de onde estejam localizados.

Trabalhando com repositórios remotos

A rotina de trabalho ao associar um repositório local a um repositório remoto é uma prática central no desenvolvimento de projetos colaborativos com Git. Isso se torna necessário porque, em um ambiente de equipe ou em colaborações externas, vários desenvolvedores podem estar trabalhando no mesmo projeto simultaneamente. Ao vincular um repositório local ao repositório remoto, cada membro da equipe pode compartilhar e sincronizar seu trabalho de forma eficaz, garantindo que todos tenham acesso às últimas alterações e possam contribuir de maneira organizada.

Essa associação permite que os membros da equipe trabalhem independentemente em suas próprias cópias locais do código, experimentando e fazendo modificações sem afetar diretamente o código principal. Quando estão prontos para compartilhar suas contribuições, eles podem enviar (push) suas alterações para o repositório remoto, onde outros podem revisá-las antes de mesclá-las ao código principal.

Trabalhando com repositórios remotos

Vamos analisar com alguns cenários:

- **Cenário 1:** você possui um repositório remoto já criado, e um repositório local também criado, e quer associar os dois:

```
git remote add origin <url_do_repo_remoto>
```

- **Cenário 2:** você ainda não possui um repositório local criado, e quer “baixar” e já sincronizar as informações de um repositório remoto:

```
git clone <url_do_repo_remoto>
```

- **Cenário 3:** você já possui repositórios local e remoto sincronizados, porém houve uma atualização no repositório remoto e você deseja atualizar suas referências locais, sem fazer nenhuma mudança:

```
git fetch
```

Trabalhando com repositórios remotos

Vamos analisar com alguns cenários:

- **Cenário 4:** você já possui repositórios local e remoto sincronizados, porém houve uma atualização no repositório remoto e você deseja atualizar suas referências locais, mesclando quaisquer mudanças que houverem ocorrido:

`git pull`

- **Cenário 5:** você está em um branch novo no seu repositório local, com commits realizados e pronto para enviar pela primeira vez para o repositório remoto:

`git push --set-upstream origin <nome_do_branch>`

- **Cenário 6:** você está em um branch que já foi enviado para o repositório remoto, e possui novos commits para serem enviados para o remoto:

`git push`

Atividade prática

Com o conhecimento que tivemos, tente realizar as seguintes ações para praticar:

- Crie um repositório remoto no GitHub, porém não o inicialize com um arquivo readme.md
- Crie um novo repositório local e associe com o repositório remoto
- No branch principal, faça alguns commits e suba para o repositório remoto
- Crie um novo branch local, faça mais alguns commits e suba para o repositório remoto
- Volte para o branch principal do projeto, faça um merge com o branch recém-criado e suba as atualizações para o repositório remoto

Atividade prática

Com o conhecimento que tivemos, tente realizar as seguintes ações para praticar:

- Crie um novo repositório remoto no GitHub, agora inicializando com um arquivo `readme.md`
- Em uma pasta local, faça um clone do repositório remoto
- No GitHub, adicione manualmente um arquivo de teste
- Faça uma atualização do repositório local, mesclando quaisquer mudanças feitas remotamente

Aula 5: Colaboração e pull requests

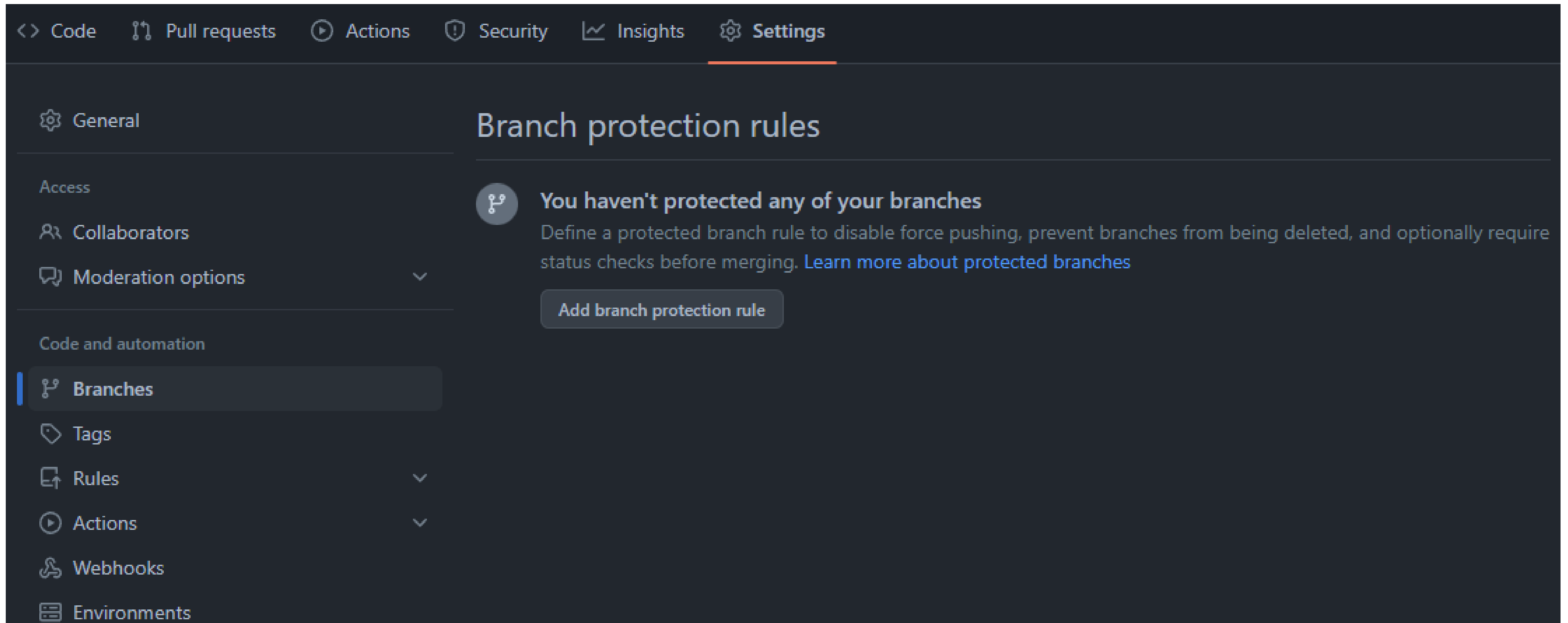
Pull requests (PRs)

Pull Requests (ou PRs) são uma maneira fundamental de colaboração no GitHub. Eles permitem que você proponha alterações em um repositório e solicite que o mantenedor do repositório (ou alguma liderança técnica da sua equipe) as revise e as incorpore. PRs são amplamente usados para discutir, revisar e colaborar no desenvolvimento de código, corrigir erros e adicionar novos recursos.

Trabalhar colaborativamente é a base de qualquer projeto de software. Por padrão, várias equipes de desenvolvimento definem que ninguém pode realizar merges direto para o branch principal (ou branches principais). Há como ajustar essas preferências no menu de configurações do repositório, direto no GitHub.


Mesmo trabalhando de forma individual, é interessante usar o conceito de branches e PRs para você conseguir controlar o que entra no projeto, e permitir fazer uma última análise estática do código, buscando melhorias e identificando defeitos.

Configurando proteções de branches



Configurando proteções de branches

Branch protection rule

**Protect your most important branches**

Branch protection rules define whether collaborators can delete or force push to the branch and set requirements for any pushes to the branch, such as passing status checks or a linear commit history.

Your GitHub Free plan can only enforce rules on its public repositories, like this one.

Branch name pattern *

Protect matching branches

☐ **Require a pull request before merging**

When enabled, all commits must be made to a non-protected branch and submitted via a pull request before they can be merged into a branch that matches this rule.

☐ **Require status checks to pass before merging**

Choose which [status checks](#) must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.

☐ **Require conversation resolution before merging**

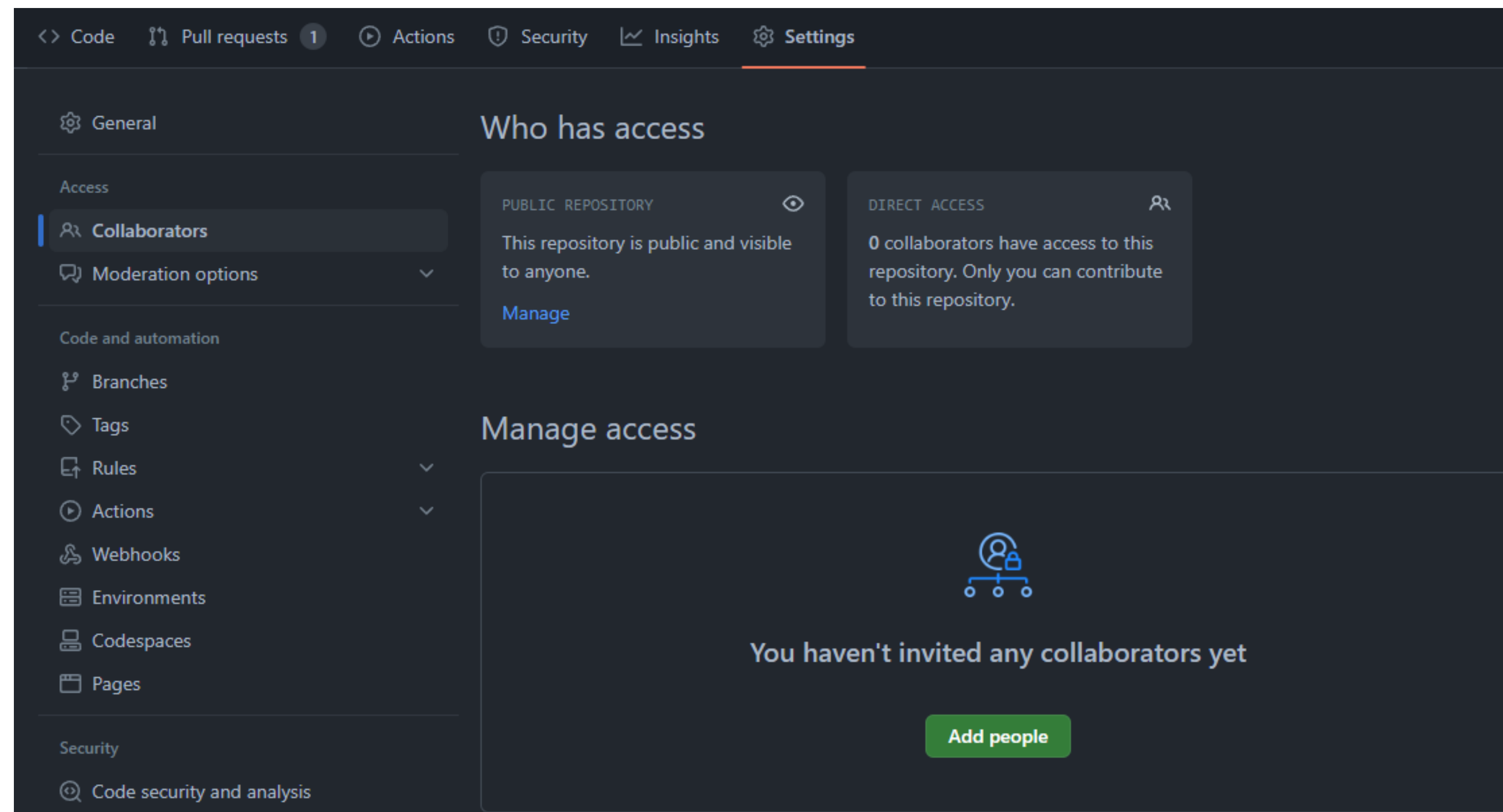
When enabled, all conversations on code must be resolved before a pull request can be merged into a branch that matches this rule. [Learn more.](#)

☐ **Require signed commits**

Commits pushed to matching branches must have verified signatures.

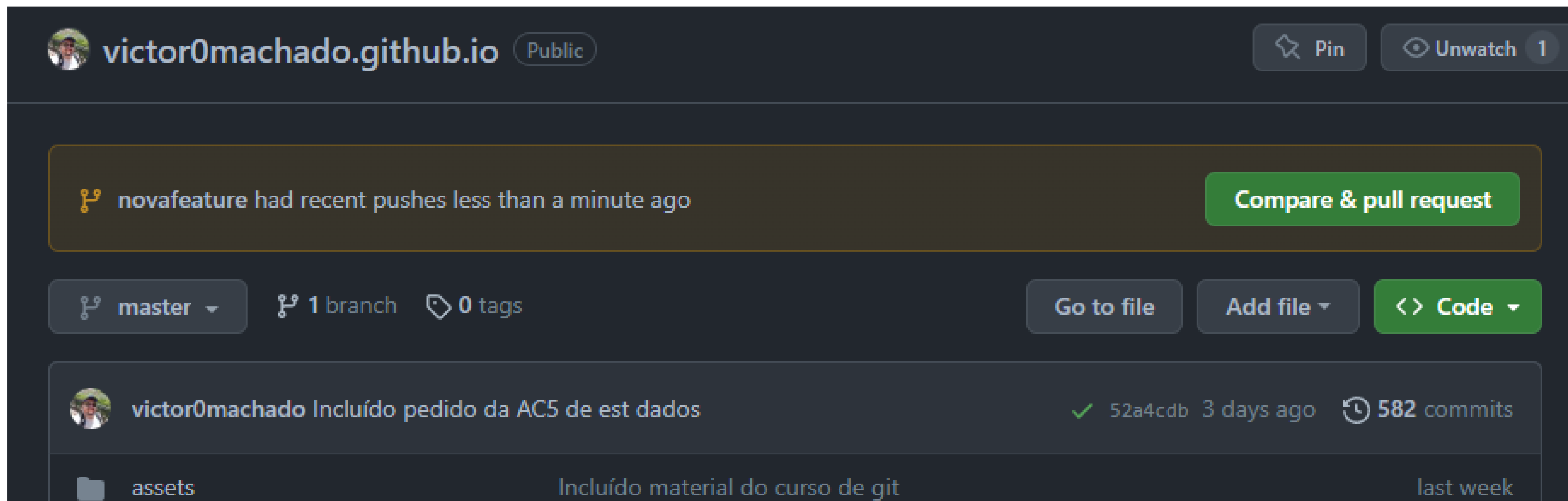
Convidando outras pessoas para o seu repo

Os PRs foram projetados para que mais de uma pessoa possa analisar o código antes deste entrar para os branches principais do repositório. Sendo assim, é preciso convidar outras pessoas para serem colaboradoras no seu repositório. Isso é feito na janela de configurações:



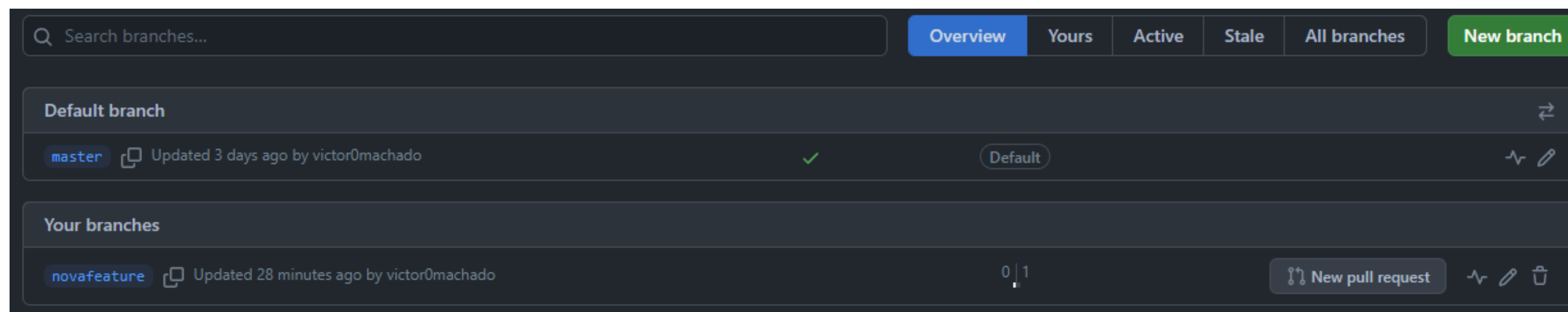
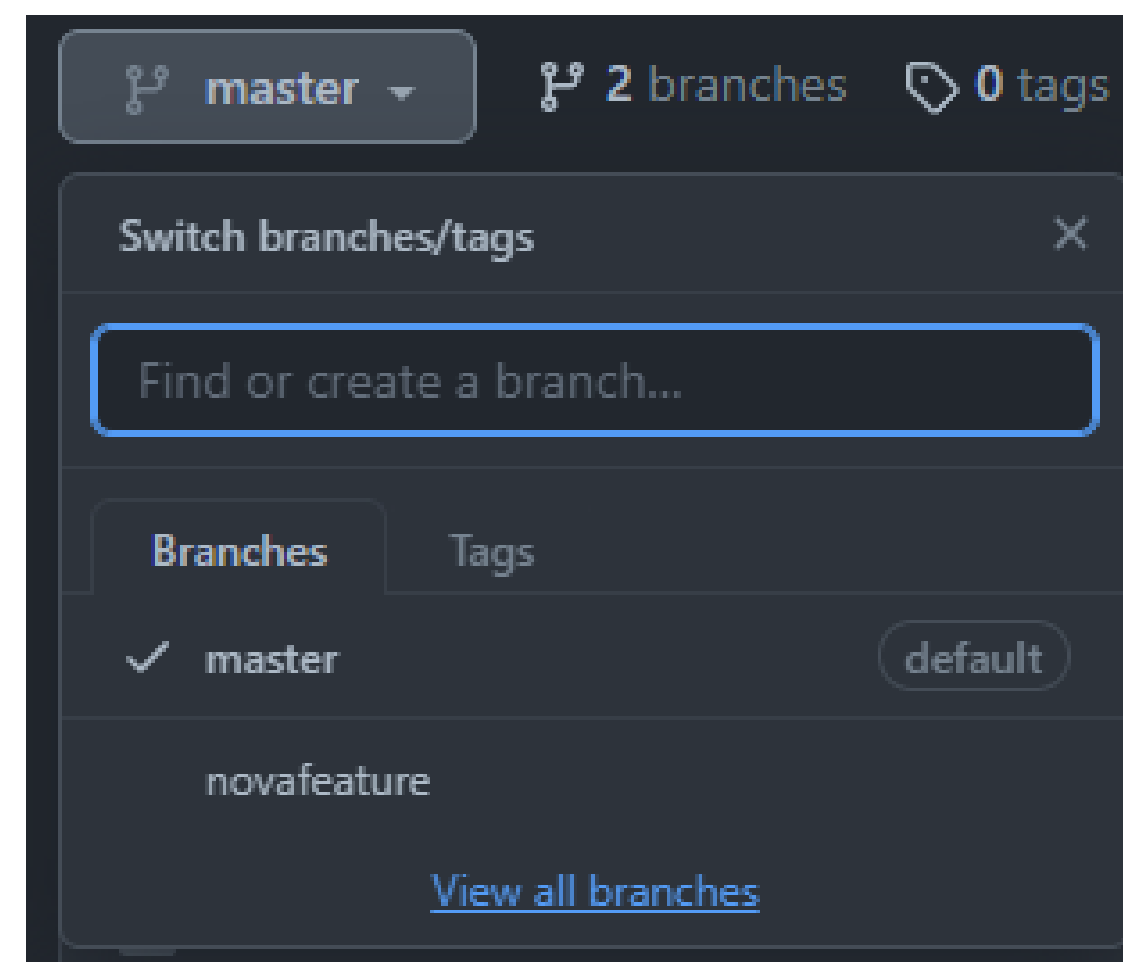
Criando um novo PR

Após fazer alterações no seu branch e subi-lo para o repositório remoto, vá na interface web do GitHub. A plataforma já reconhece que houve um push recente no branch desejado e disponibiliza uma opção para comparar e abrir um PR.



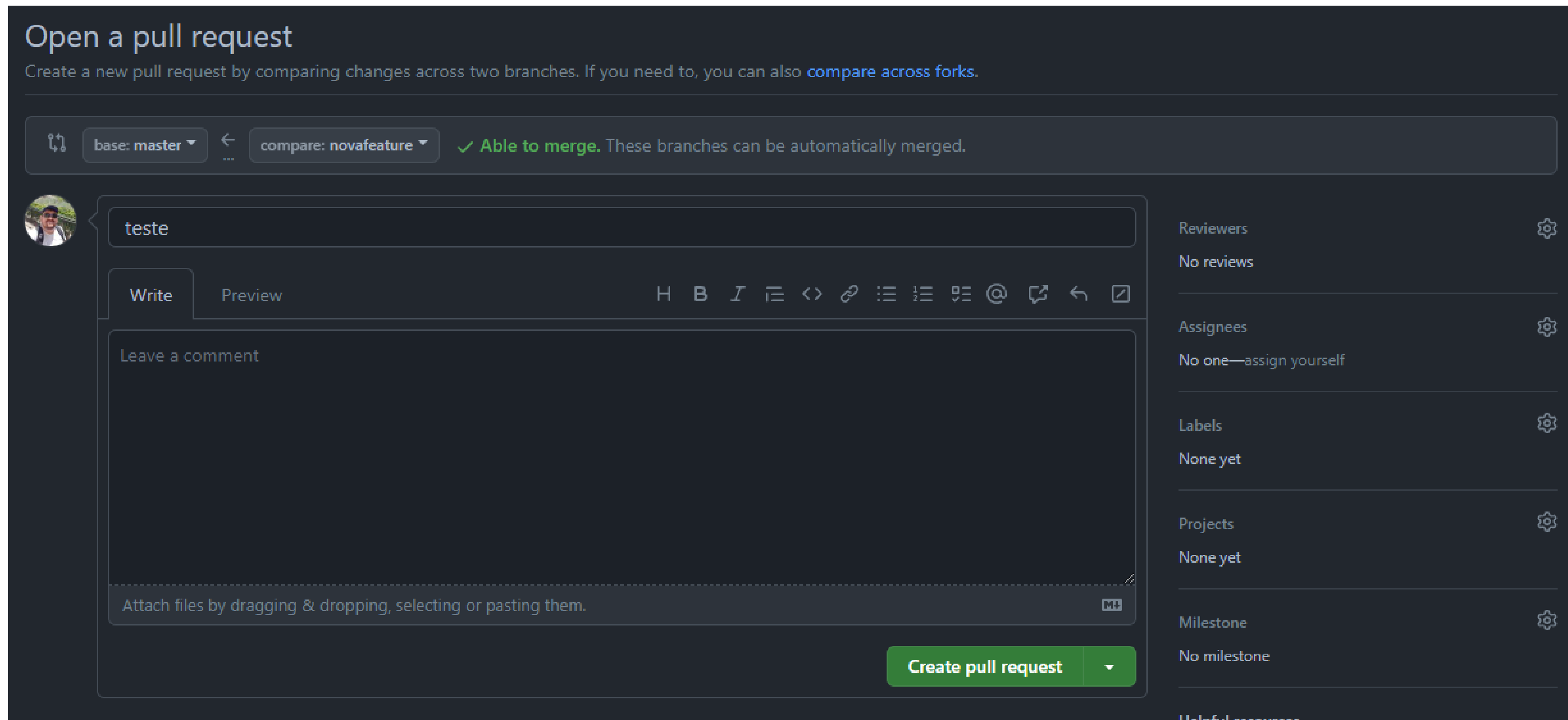
Criando um novo PR

Uma alternativa, caso a mensagem não esteja sendo exibida, é selecionar a opção para visualizar todos os branches e, em seguida, selecionar qual branch deseja abrir um PR:



Criando um novo PR

Na janela de abertura de PR, há diversos campos que podem ser preenchidos. É sempre interessante verificar se não há conflitos antes de abrir o PR.



The screenshot shows the GitHub 'Open a pull request' interface. At the top, it says 'Open a pull request' and 'Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).' Below this, there are two dropdown menus: 'base: master' and 'compare: novafeature'. To the right of these is a green checkmark and the text 'Able to merge. These branches can be automatically merged.' Below the dropdowns is a text input field with the placeholder 'teste'. To the right of the input field are tabs for 'Write' and 'Preview'. Below the tabs is a rich text editor with a toolbar containing icons for bold, italic, code, link, unlink, list, ordered list, table, quote, mention, link preview, and a close button. Below the editor is a large text area with the placeholder 'Leave a comment'. At the bottom of the text area is a button that says 'Attach files by dragging & dropping, selecting or pasting them.' To the right of the text area is a green button that says 'Create pull request'. On the right side of the interface, there are several sections: 'Reviewers' with 'No reviews', 'Assignees' with 'No one—assign yourself', 'Labels' with 'None yet', 'Projects' with 'None yet', and 'Milestone' with 'No milestone'. Each section has a gear icon to its right.

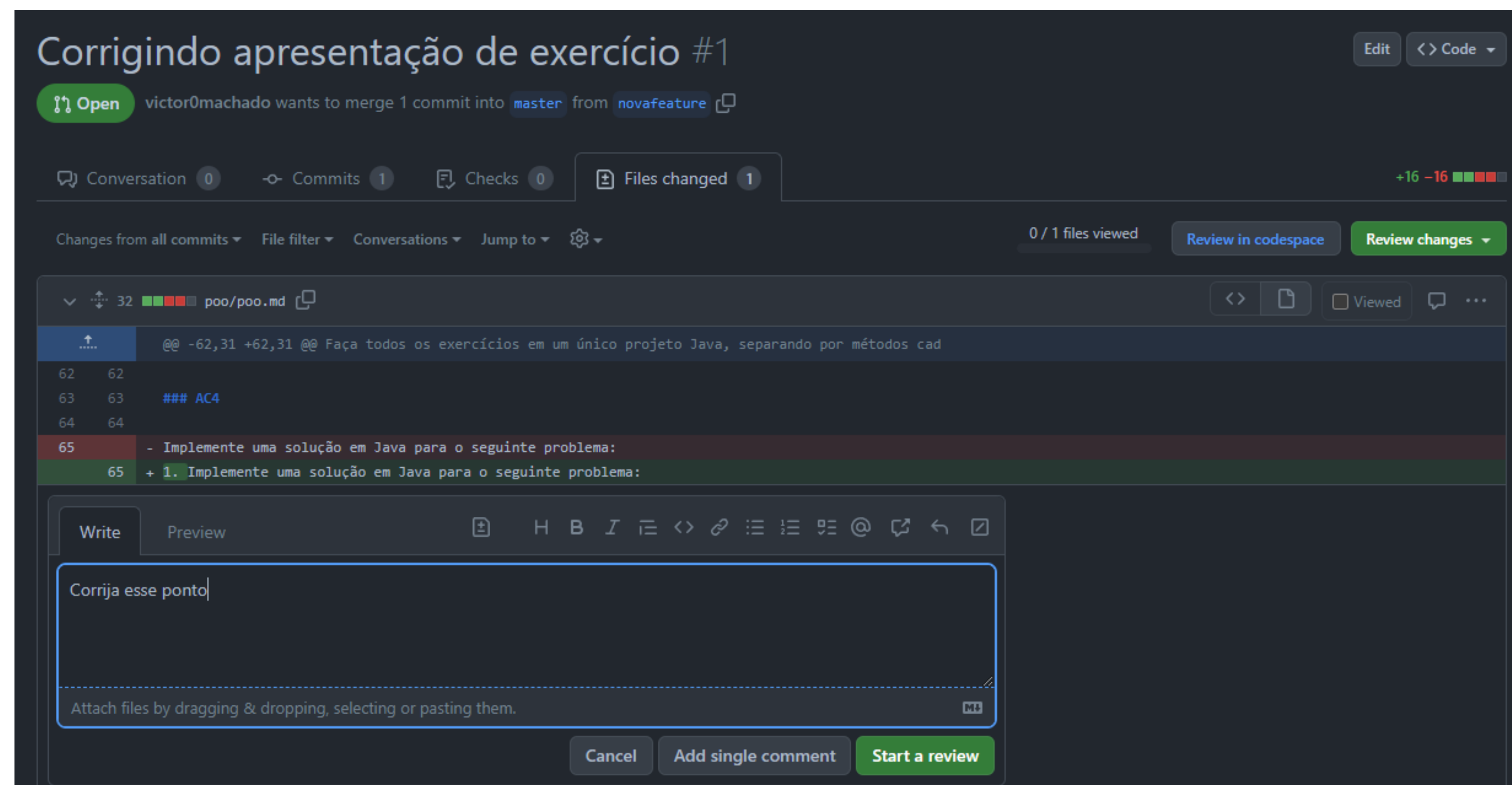
Criando um novo PR

Campos para se preencher em um PR:

- **Nome:** usualmente um nome que descreva o que foi feito, p.ex. “Correção da issue #43” ou “Melhorias na interface da tela xyz”;
- **Descrição:** faça uma descrição ampla do trabalho realizado. Apresente links para documentações, utilize imagens que descrevam mudanças visuais, explique decisões de design e traga outros comentários que auxiliem as pessoas que farão a revisão do PR;
- **Revisores:** inclua pessoas que você acha que poderiam contribuir com a análise do código. Normalmente, quanto mais complexo for o PR, mais pessoas vão precisar se envolver na análise. É interessante incluir pessoas que já trabalharam naquela parte do código;
- **Aprovador (assignee):** é a pessoa que vai aprovar, de fato, o PR. Normalmente é alguma liderança técnica ou gerente da equipe. Varia de empresa para empresa;
- **Rótulos:** palavras-chave que descrevam o PR, p.ex. “bugfix”, “feature”, “refactor”.

Revisando um PR

Com um PR aberto, as pessoas podem realizar análises estáticas do código, ou seja, ler o código e identificar possíveis defeitos ou oportunidades de melhoria. Comentários podem (e devem) ser abertos sempre que for identificada a necessidade de alteração.



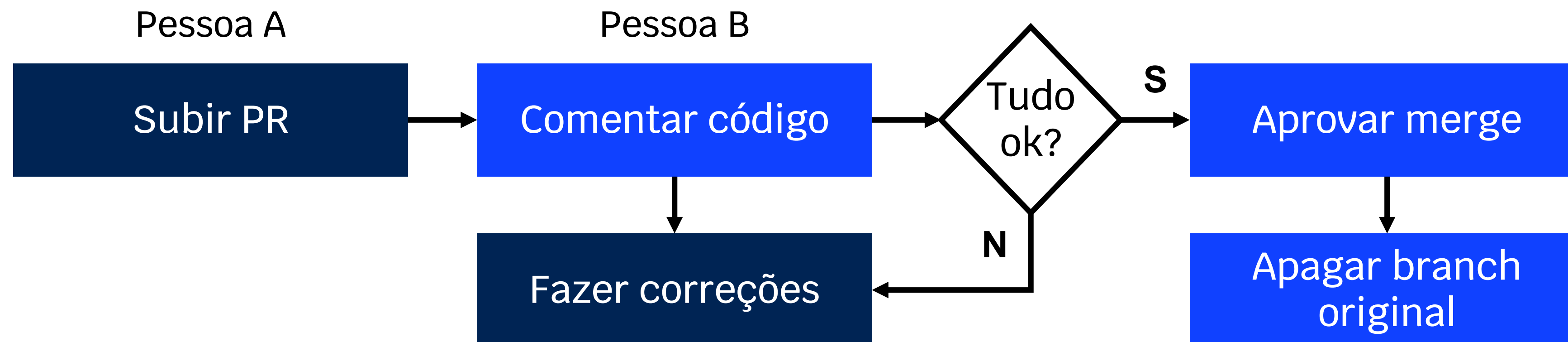
Revisando um PR

Algumas dicas para um bom processo de revisão de PR:

- Evite se estender muito em um comentário. Se você sentir que a alteração precisa de um debate mais aprofundado, chame a pessoa para conversar e discutir a situação;
- Da mesma forma, se foi você quem abriu o PR e percebeu um comentário confuso ou que daria muito trabalho, chame a pessoa para conversar, entenda bem o que foi solicitado e suba um novo comentário com um resumo dessa conversa;
- Fechar PRs não é um problema! Se a pessoa revisora identificou uma falha grande e que precisará de diversos commits para resolver, feche o PR, trabalhe nas mudanças e suba um novo PR após ter tudo corrigido;
- Se você é a pessoa que abriu o PR, não resolva as issues abertas por outras pessoas. Deixe que quem abriu o comentário o resolva, para ele visualizar que as mudanças foram feitas.

Finalizando um PR

Fluxo principal do PR:



Atividade prática

Com o conhecimento que tivemos, tente realizar as seguintes ações para praticar:

- Em um repositório remoto pessoal, convide um colega para colaborar
- Localmente, abra um branch, faça alguma implementação e suba para o remoto
- Abra um PR, lembrando de marcar seu colega como aprovador
- Peça para o colega fazer pelo menos um comentário no material enviado
- Faça as correções necessárias localmente e suba as mudanças
- Peça para o colega revisar novamente e aprovar (ou não!) as mudanças
- Peça para o colega aprovar o PR e apagar o branch original
- Faça um `git pull` localmente, para atualizar o seu branch principal
- Inverta os papéis! Peça para seu colega criar um repositório e te convidar como colaborador

Aula 6: Resolução de conflitos

Sobre conflitos no Git

Conflitos no desenvolvimento colaborativo ocorrem quando duas ou mais pessoas fazem alterações no mesmo arquivo ou trecho de código ao mesmo tempo.

Resolver conflitos é uma parte essencial do trabalho com Git e colaboração em projetos compartilhados.

Nesta aula, exploraremos as razões pelas quais os conflitos ocorrem, e como lidar com eles de maneira eficaz.

Causas comuns de conflitos

Conflitos podem surgir por várias razões, como:

- Alterações simultâneas em um arquivo por diferentes colaboradores;
- Renomeação ou movimentação de arquivos;
- Mesclagens anteriores mal resolvidas.

Apesar de ser importante evitar essas razões de conflitos, inevitavelmente elas vão acontecer. Se estamos trabalhando em projetos com equipes grandes, e o nosso ciclo de *merge* ao branch principal é longo, eventualmente vamos trabalhar em algum código que já foi modificado por alguém e ainda não está no branch principal.

Boas práticas

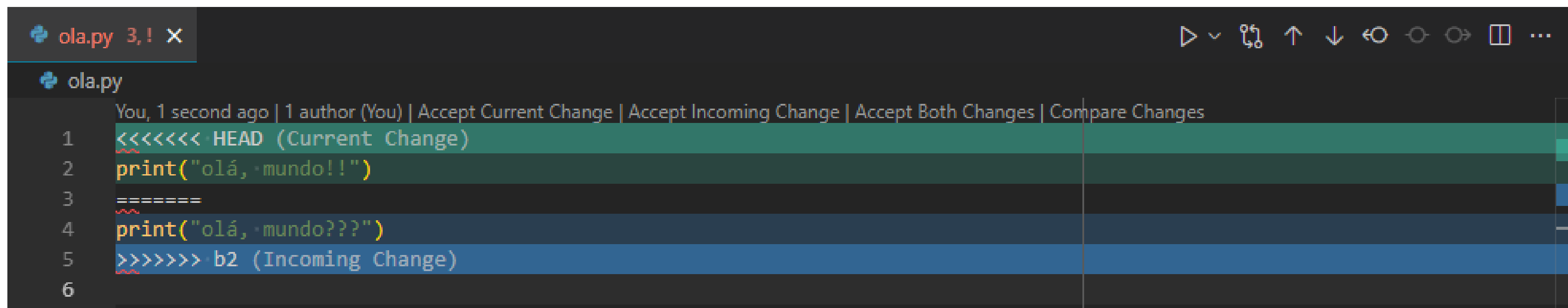
Algumas boas práticas para prevenir as causas comuns de conflitos:

- **Comunicação eficiente:** mantenha um canal de comunicação claro com a equipe para que todos saibam o que os outros estão trabalhando. Discuta grandes alterações e coordenem tarefas para evitar conflitos decorrentes de alterações simultâneas. Práticas ágeis como *daily meetings* e sessões de retrospectiva ajudam muito nesse ponto;
- **Trabalho em branches separados:** encoraje os membros da equipe a trabalhar em branches separados para suas tarefas, e faça o *merge* para o branch principal regularmente. Isso ajuda a reduzir a probabilidade de conflitos;
- **Padrões de nomeação de arquivos e diretórios:** estabeleça padrões de nomeação claros para arquivos e diretórios, para minimizar conflitos decorrentes de renomeações e movimentações;
- **Revisões regulares de código:** realize revisões regulares de código para identificar e resolver conflitos potenciais antes que se tornem um problema real;
- **Atualize seu repo local frequentemente:** antes de iniciar um novo branch local, atualize o seu repo remoto, pois alguém do time pode ter atualizado o branch principal.

Resolvendo conflitos

Conflitos de *merge* ocorrem quando o Git não pode determinar como combinar duas linhas de código que foram alteradas na mesma parte do arquivo em commits diferentes. Nesses casos, você precisa intervir manualmente. Considere as seguintes etapas principais:

1. **Adote uma ferramenta:** uma boa ferramenta para revisão de código ajuda muito. O VSCode possui boas extensões para Git que auxiliam na resolução de conflitos, como o GitLens, mas o IDE já tem vários recursos nativos interessantes;
2. **Identificação de conflitos:** o Git marcará as áreas do arquivo no momento em que ele não conseguir mesclar as alterações. No exemplo abaixo, a linha verde indica o código atual (no branch principal, por exemplo), enquanto que a linha azul indica o código que precisa ser incorporado;



The screenshot shows a code editor window with a file named 'ola.py'. The editor displays a merge conflict. The current change (HEAD) is shown in green, and the incoming change (b2) is shown in blue. The conflict is on line 4, where the current change has 'print("olá, mundo!!")' and the incoming change has 'print("olá, mundo???)'. The editor also shows the file's history and the merge status.

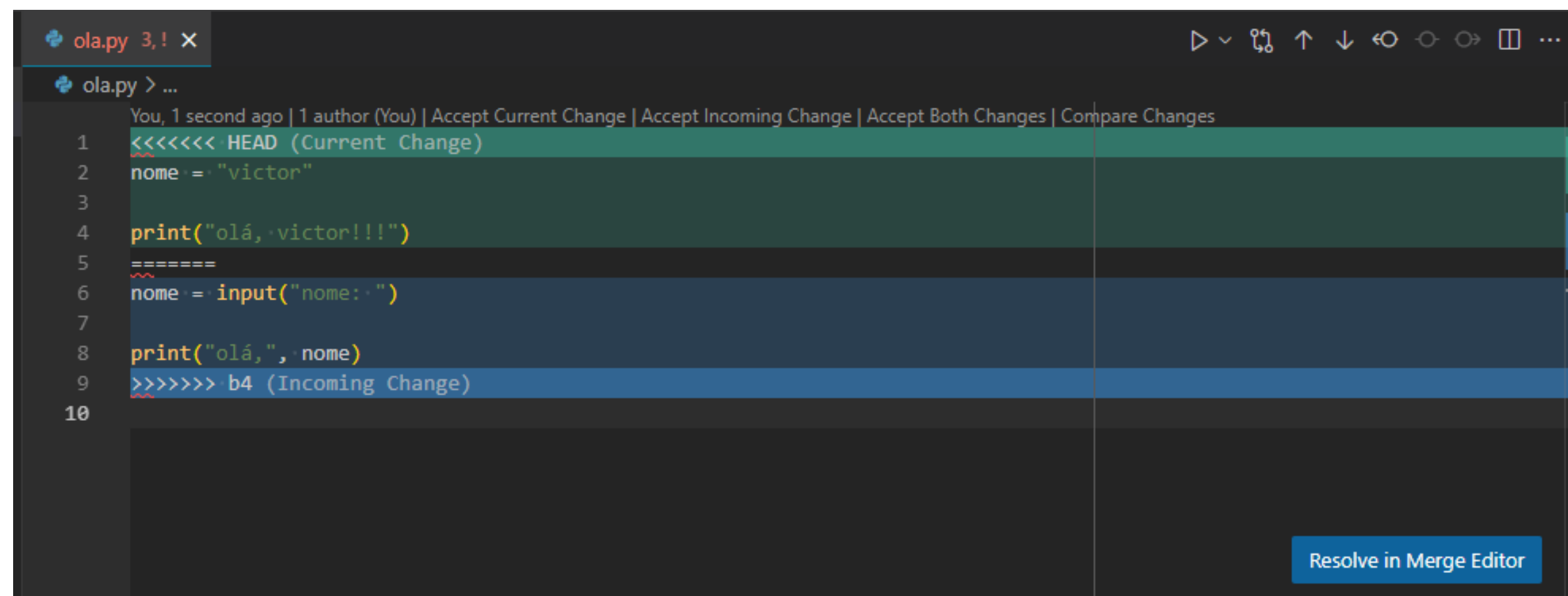
```
ola.py 3, ! X
ola.py
You, 1 second ago | 1 author (You) | Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
1 <<<<<< HEAD (Current Change)
2 print("olá, mundo!!")
3 =====
4 print("olá, mundo???)
5 >>>>>> b2 (Incoming Change)
6
```

Resolvendo conflitos

3. **Resolução manual:** analise as alterações conflitantes e decida qual versão das alterações deve ser mantida. Isso pode envolver a edição do código para combinar as alterações, ou escolher uma das versões;
4. **Remoção dos marcadores:** à medida que você resolve cada conflito, remova os marcadores Git (como “<<<<<< HEAD”) e deixe o código em um estado consistente. Certifique-se de que o arquivo não contenha mais nenhuma marcação de conflito;
5. **Salvar as alterações:** salve o arquivo após resolver todos os conflitos. Certifique-se de que o arquivo esteja no estado desejado;
6. **Commit das alterações:** após resolver todos os conflitos e salvar o arquivo, faça um commit das alterações. O Git reconhecerá que o conflito foi resolvido e registrará a resolução.

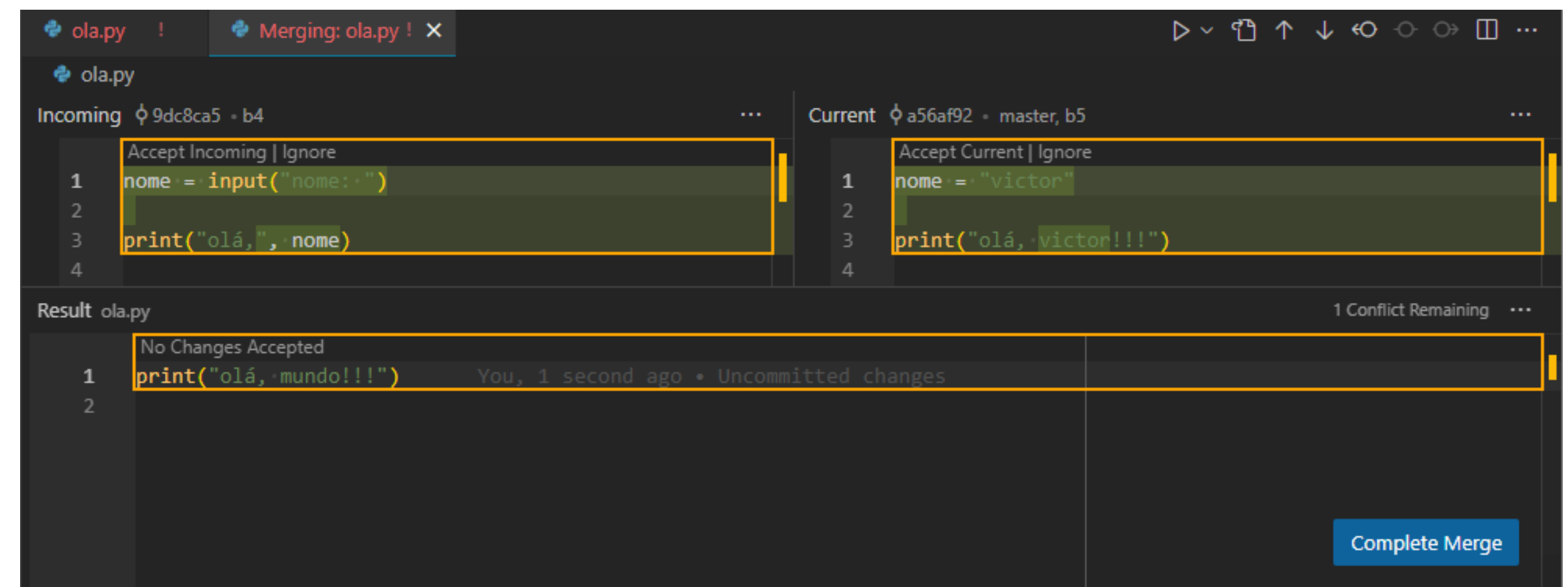
Dicas

- No VSCode, ao abrir um arquivo com conflitos, você verá as seções conflitantes realçadas. Com a extensão GitLens instalada, aparece um novo botão quando o arquivo está em conflito, “Resolve in Merge Editor”. Isso te leva para uma tela em que você pode resolver os conflitos de forma mais interativa.



```
ola.py 3, ! x
ola.py > ...
You, 1 second ago | 1 author (You) | Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
1 <<<<<<< HEAD (Current Change)
2 nome = "victor"
3
4 print("olá, victor!!!")
5 =====
6 nome = input("nome: ")
7
8 print("olá, ", nome)
9 >>>>>> b4 (Incoming Change)
10
```

Resolve in Merge Editor



```
ola.py ! Merging: ola.py ! x
ola.py
Incoming 9dc8ca5 - b4 ... Current a56af92 - master, b5 ...
1 Accept Incoming | Ignore 1 Accept Current | Ignore
2 nome = input("nome: ") 2 nome = "victor"
3 print("olá, ", nome) 3 print("olá, victor!!!")
4 4
Result ola.py 1 Conflict Remaining ...
1 No Changes Accepted 1 print("olá, mundo!!!") You, 1 second ago - Uncommitted changes
2
```

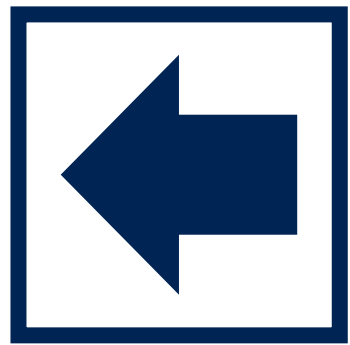
Complete Merge

Dicas

Importante: Se você trabalhou em um branch, subiu para o remoto e abriu um Pull Request para mesclar o seu branch ao branch principal, e só aí identificou o conflito, é necessário resolvê-lo localmente e subir o branch novamente. A solução mais direta é realizar um *merge* do branch principal no seu branch de desenvolvimento, para só então subir novamente.

Subiu para o repo remoto, e lá abriu um PR →
Atualizou o seu repo local →
Fez a tentativa de merge do main no seu branch →
Resolveu o conflito e commitou mudança →
Subiu novamente para o remoto - tudo certo! →

```
git commit -m "Último commit antes do PR"  
git push  
git fetch origin  
git merge origin/main  
git commit -m "Resolvido conflito com o branch main"  
git push
```

Aula 7: GitHub Issues

O que é?

O **GitHub Issues** é uma poderosa ferramenta de rastreamento e gerenciamento de tarefas que desempenha um papel central na colaboração em projetos de desenvolvimento de software.

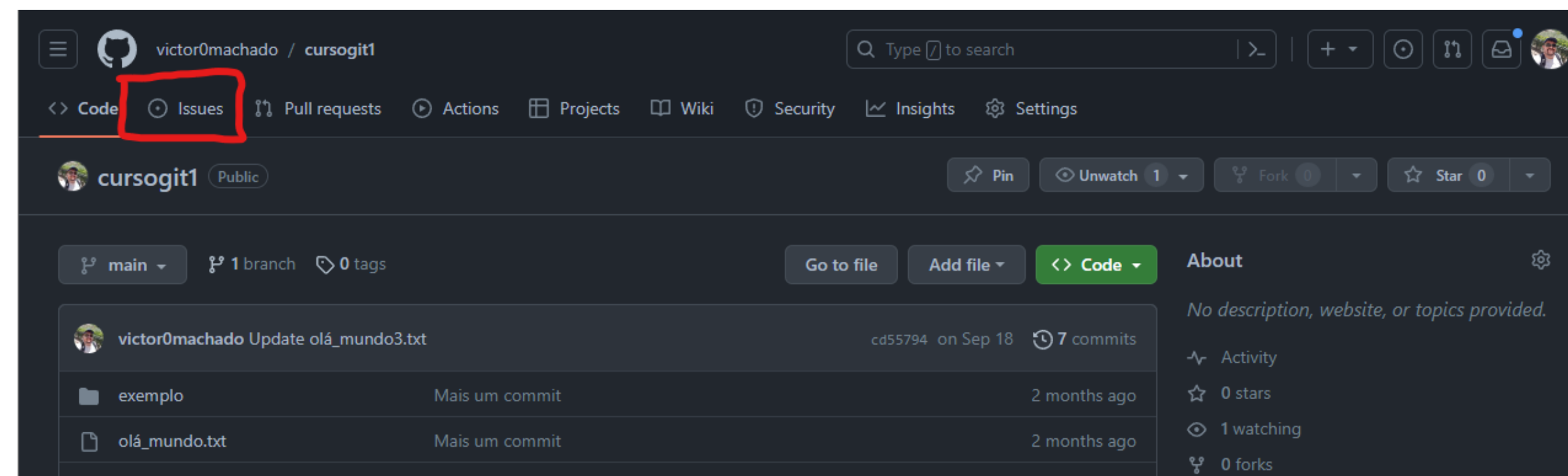
Issues podem representar tarefas, erros, melhorias, discussões e muito mais. Ao criar issues, os membros da equipe podem relatar problemas, atribuir responsabilidades e discutir soluções em um ambiente colaborativo.

Issues oferecem visibilidade e um registro claro do trabalho que precisa ser feito, ajudando a manter a equipe informada sobre o progresso e fornecendo um histórico valioso para futuras referências.

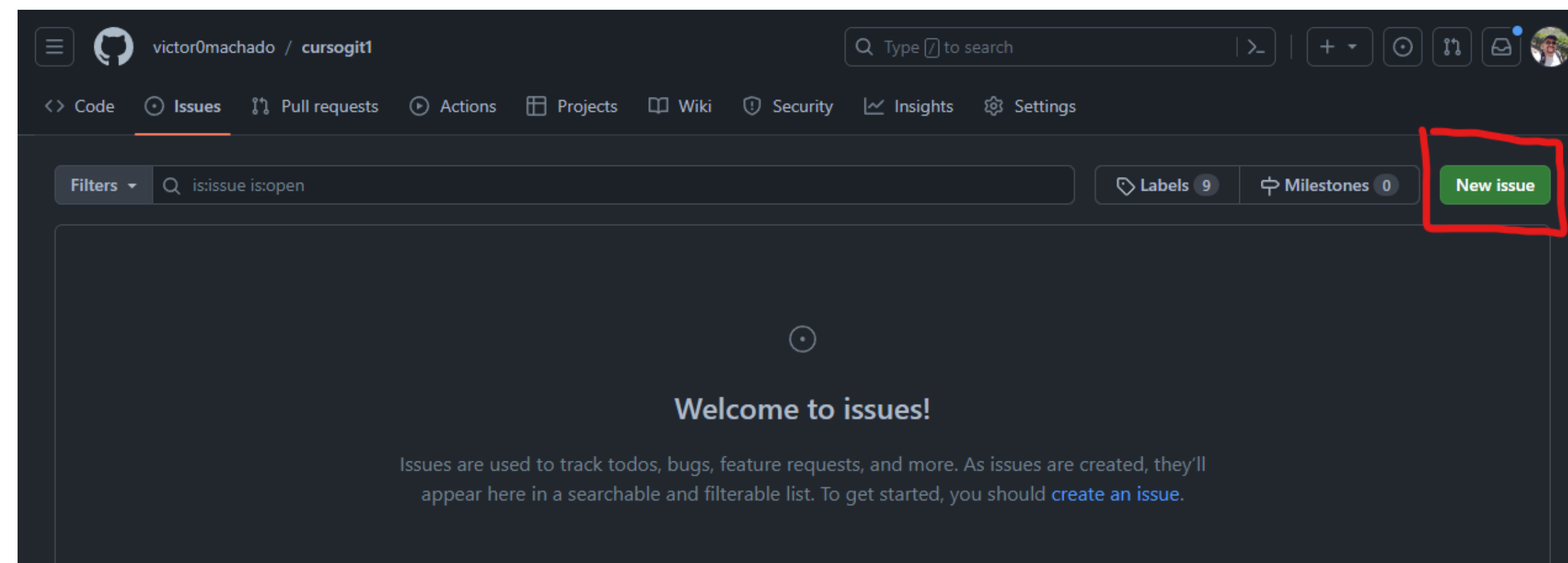
Além disso, issues podem ser vinculadas a PRs, facilitando a rastreabilidade das mudanças de código relacionadas a tarefas específicas.

Criando issues

Para criar uma issue, vá na página do repositório no GitHub e clique na opção “Issues”, na barra superior da tela:



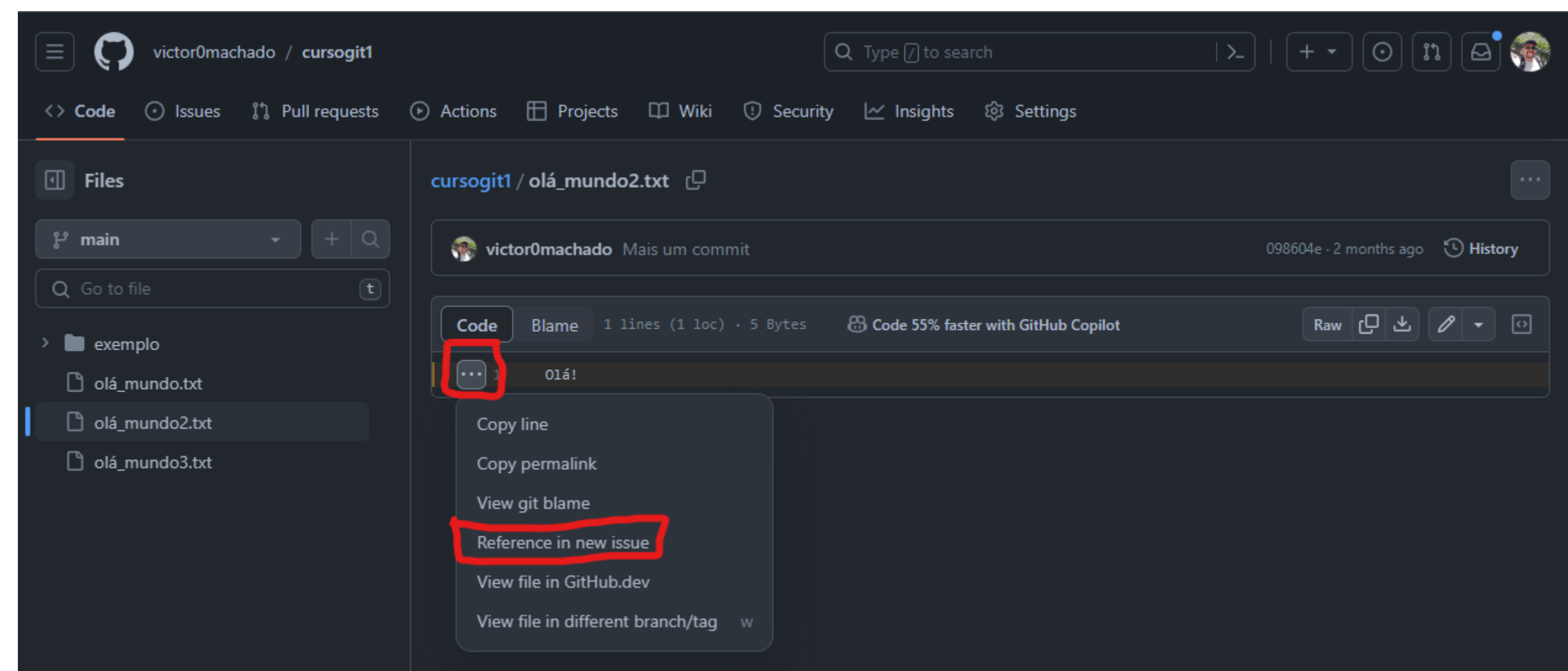
Em seguida, basta clicar em “New Issue”:



Criando issues

Uma outra forma é direto em um arquivo. Isso é interessante quando você quer citar especificamente uma ou mais linhas de código em um arquivo, simplificando o processo de correção, análise e discussão.

Para isso, basta abrir o arquivo no GitHub, selecionar as linhas que você deseja comentar e clicar nos três pontos. Em seguida, clicar em “Reference in new issue”:



Criando issues

O menu de criação de issues permite várias ações:

- Defina um título e insira uma descrição. É possível usar markdown para formatar e organizar o conteúdo! [Consulte a sintaxe básica do markdown aqui](#);
- Defina um “assignee”, ou um responsável pela resolução do problema. Isso é interessante quando você sabe quem está cuidando daquela parte do código;
- Crie labels (rótulos)! É um recurso visual excelente para categorizar as diferentes issues por elementos da arquitetura (interface, dados, pacotes, etc.), por componentes do software (build, automação, integrador, etc.) ou por tipo de issue (bug, nova feature, dívida técnica...);
- Associe com projetos (projects) e marcos (milestones) -- vamos ver em breve!

Criando issues

The screenshot shows a GitHub repository page for 'victor0machado / cursogit1'. The 'Issues' tab is active, showing 'Issue 1 #1'. The issue is marked as 'Open' and was opened by 'victor0machado' 1 minute ago. The issue title is 'Exemplo de issue!'. The issue is assigned to 'victor0machado' and has two labels: 'bug' and 'help wanted'. The issue is also linked to a project and a milestone. The right sidebar shows the 'Assignees' section with 'victor0machado' listed, and the 'Labels' section with 'bug' and 'help wanted' listed. The 'Projects' section shows 'None yet' and the 'Milestone' section shows 'No milestone'. The 'Development' section shows 'Create a branch for this issue or link a pull request.' The bottom section shows the 'Write' and 'Preview' tabs, with a text area for 'Leave a comment' and a 'Comment' button. The 'Close issue' button is also visible.

Issue 1 #1

Open victor0machado opened this issue 1 minute ago · 0 comments

victor0machado commented 1 minute ago

Exemplo de issue!

victor0machado self-assigned this 1 minute ago

victor0machado added the bug label now

victor0machado added the help wanted label now

Assignees

victor0machado

Labels

bug help wanted

Projects

None yet

Milestone

No milestone

Development

Create a branch for this issue or link a pull request.

Notifications

Unsubscribe

You're receiving notifications because you're watching this repository.

1 participant

Write Preview

Leave a comment

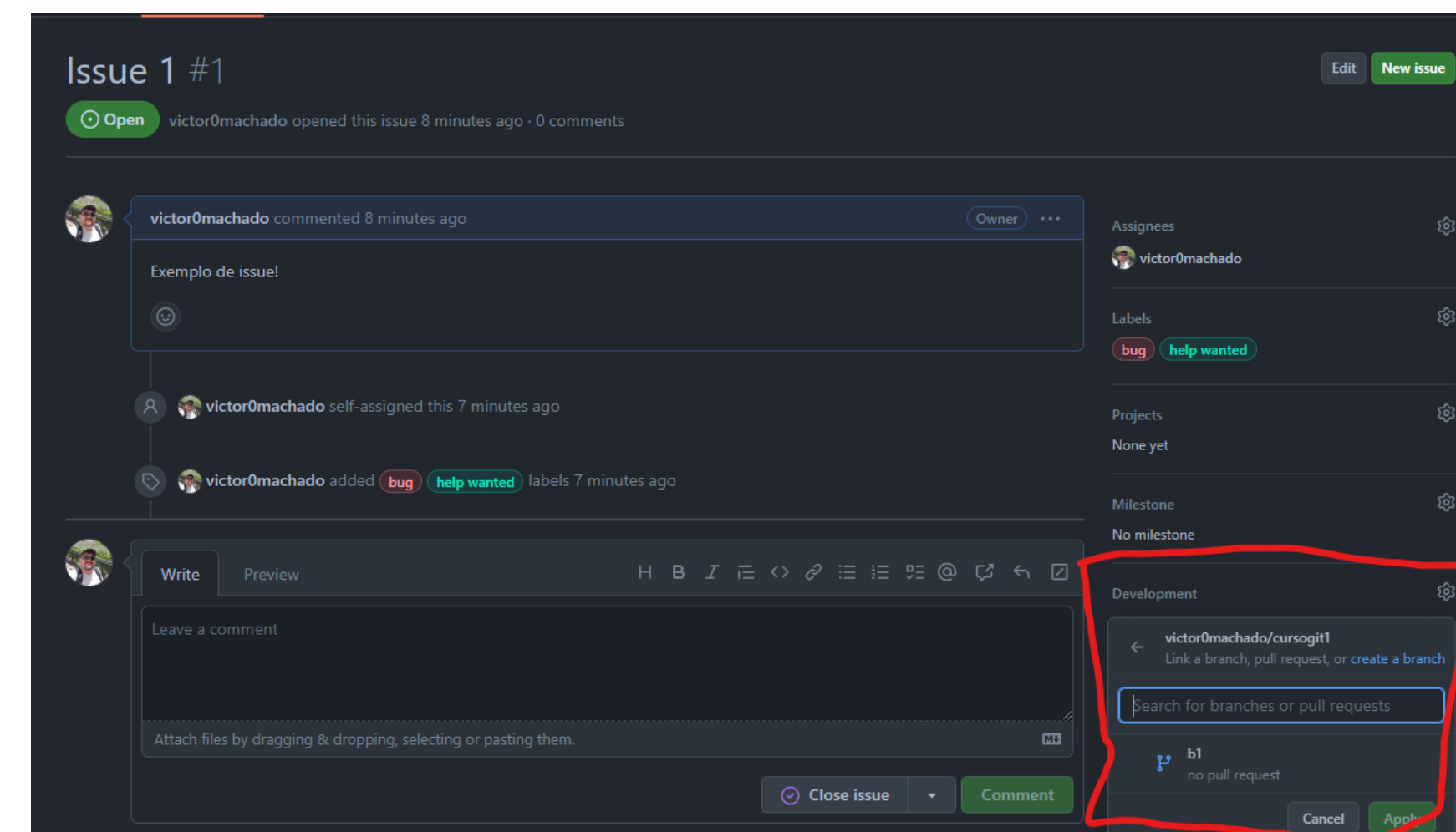
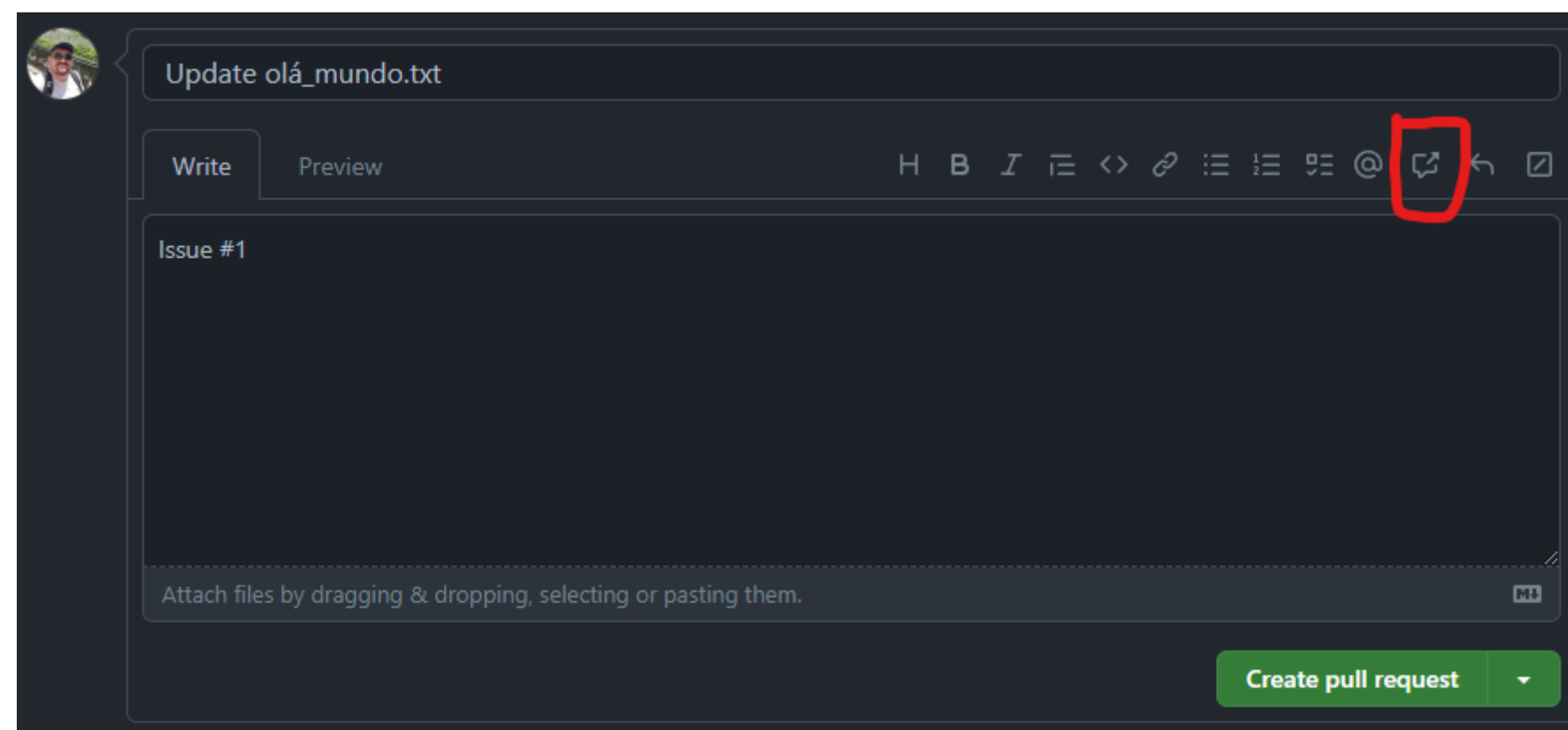
Attach files by dragging & dropping, selecting or pasting them.

Close issue Comment

Usando issues

É possível usar issues de várias formas:

- **Discutindo o problema:** algumas issues em projetos open source rendem páginas e páginas de discussão, em que os grupos debatem em torno da melhor solução para o problema, decidem pela arquitetura e identificam melhorias no projeto. Esse é um recurso que melhora muito a comunicação assíncrona dentro de um projeto (essencial hoje em dia);
- **Associando PRs e branches a issues:** isso é possível tanto dentro do corpo do texto da mensagem de Pull Request, quanto na tela da própria issue. Com isso, é possível manter a rastreabilidade de quando e onde cada issue está sendo trabalhada e resolvida.



GitHub Projects

GitHub Projects é uma ferramenta que capacita equipes a organizar e gerenciar projetos de maneira flexível.

Ele fornece quadros de tarefas estilo Kanban, listas de pendências e tarefas personalizadas para atender às necessidades específicas de cada projeto.

Ao criar projetos, os membros da equipe podem categorizar, priorizar e distribuir tarefas de forma clara. Isso ajuda a manter todos na mesma página, otimizar fluxos de trabalho e garantir que os objetivos do projeto sejam alcançados dentro do prazo.

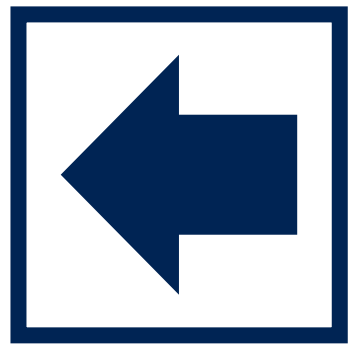
Com o uso do GitHub Projects, você pode ver o panorama geral, entender em que ponto está cada tarefa e planejar a próxima etapa com eficiência.

Rótulos e Marcos

Rótulos (labels) e marcos (milestones) são recurso essenciais para a organização de issues. Labels são etiquetas que você pode aplicar às issues para classificá-las e categorizá-las. Isso facilita a identificação rápida do tipo de tarefa ou problema representado por cada issue.

Milestones, por outro lado, são usados para agrupar issues relacionadas a uma versão específica, um objetivo ou marco importante do projeto. Eles fornecem um contexto mais amplo para as tarefas e ajudam a manter o foco em objetivos de curto e longo prazo.

A combinação eficaz de labels e milestones permite uma organização mais precisa e facilita o acompanhamento do progresso do projeto.



Aula 8: Gitignore

O que é o gitignore?

O arquivo `.gitignore` é uma ferramenta fundamental para manter a integridade e a clareza de um repositório Git. Ele permite que você especifique quais arquivos e diretórios o Git deve ignorar ao rastrear alterações no projeto.

Isso é especialmente útil para excluir arquivos temporários, arquivos de configuração específicos do ambiente, arquivos gerados por ferramentas de compilação e outras informações que não são relevantes para a colaboração ou o histórico de versões.

Além disso, o `.gitignore` é extremamente flexível e suporta uma variedade de padrões de exclusão, o que permite adaptá-lo às necessidades específicas do projeto. Ao dominar o uso do `.gitignore`, você contribui para a limpeza e eficiência do repositório, simplificando a revisão e a colaboração na equipe.

Boas práticas de organização de repositórios

- Organização é essencial: A organização adequada de um repositório é fundamental para facilitar o trabalho em equipe e a manutenção a longo prazo.
- Use modelos de `.gitignore`: O GitHub oferece [modelos](#) de `.gitignore` para linguagens e ambientes comuns, simplificando a configuração inicial do arquivo `.gitignore`.
- Evite arquivos desnecessários: O `.gitignore` ajuda a evitar que arquivos temporários, de compilação e de configuração local sejam rastreados.
- Documente a estrutura: Inclua um arquivo README explicando a estrutura e o propósito do repositório para que outros colaboradores entendam o projeto rapidamente.
- Comunique as práticas: Garanta que a equipe conheça as boas práticas de organização e uso do `.gitignore`.
- Evite a duplicação: Evite incluir arquivos que já estão no `.gitignore` global do sistema. Isso pode economizar tempo e evitar redundância.

Criando o seu gitignore

Arquivo global:

- Vá na sua pasta de usuário (no Windows, usualmente `C:\Usuários\<seu_nome_de_usuario>`)
- Lá, crie o arquivo `.gitignore_global`
- Configure o Git para usar esse arquivo como seu `.gitignore` global. Pode fazer isso usando o seguinte comando no terminal:

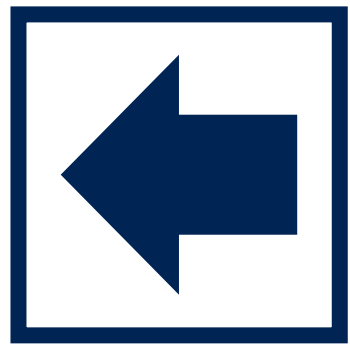
```
git config --global core.excludesfile <caminho_do_arquivo>
```

Arquivo local:

- No seu repositório local, crie o arquivo `.gitignore` na raiz do diretório principal

Como incluir padrões para ignorar?

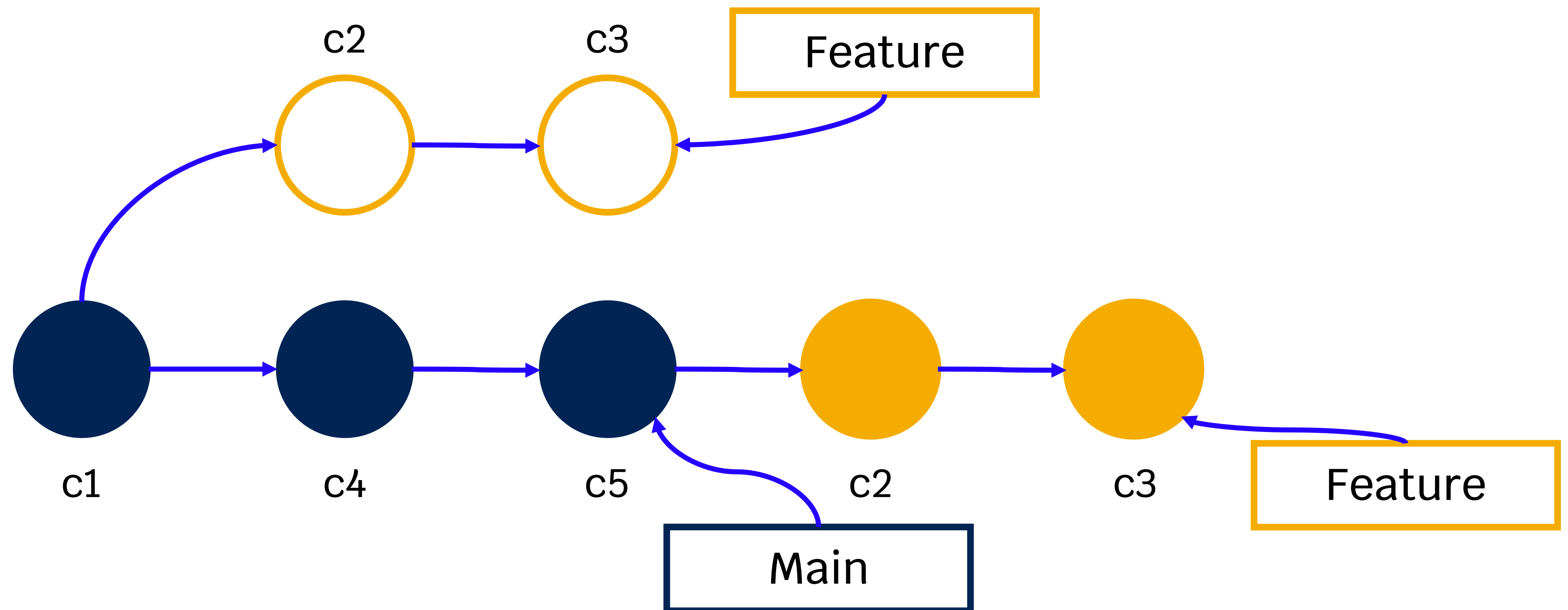
- Todos os arquivos de uma determinada extensão: utilize um asterisco seguido da extensão, como `*.pyc`, `*.log` ou `*.class`
- Excluindo arquivos específicos: insira diretamente o nome do arquivo completo, incluindo a extensão, como em `Thumbs.db` ou `desktop.ini`
- Por diretório ou pasta: inclua uma barra ao final do nome da pasta, como `__pycache__/` ou `config/`
- Excluindo arquivos e pastas recursivamente: inclua dois asteriscos antes ou depois de um determinado padrão a ser ignorado, para ignorar também em subpastas, como `**/__pycache__` (exclui pastas e subpastas com o nome `__pycache__`), `**/*.log` (exclui todos os arquivos `.log` em todas as pastas e subpastas) ou `build/**` (exclui todos os arquivos e pastas dentro da pasta `build`)
- Use `#` para inserir comentários!



Aula 9: Git rebase e cherry-pick

Compreendendo o git rebase

O git rebase permite reescrever o histórico de commits de uma ramificação, aplicando os commits de outra ramificação no topo. Isso é útil para manter um histórico linear e coerente de desenvolvimento.



Compreendendo o git rebase

O Git Rebase é uma operação poderosa e versátil que permite a reescrita do histórico de commits em uma ramificação. Ao contrário do Git Merge, que cria uma nova confirmação de mesclagem, o Git Rebase move ou "rebaseia" seus commits para um novo ponto de base, geralmente em relação a outra ramificação ou commit. Isso é especialmente útil quando você deseja manter um histórico linear e coeso de desenvolvimento, o que pode tornar a leitura e a compreensão do histórico de commits mais simples.

Compreendendo o git rebase

Aqui estão algumas das principais funcionalidades do Git Rebase:

- **Reorganização de Commits:** O Git Rebase permite que você reorganize a ordem dos commits, tornando o histórico de desenvolvimento mais lógico e fácil de seguir. Isso é particularmente útil quando você deseja agrupar commits relacionados ou corrigir erros de organização.
- **Combinando Commits:** Você pode combinar commits diferentes em um único, o que é útil para reduzir a complexidade do histórico ou agrupar mudanças relacionadas.
- **Divisão de Commits:** O Git Rebase também permite dividir commits em commits menores, dividindo alterações em unidades mais granulares e lógicas. Isso pode ajudar a melhorar a modularidade e a rastreabilidade do histórico.
- **Alteração de Mensagens de Commit:** Se você precisa corrigir mensagens de commit com erros ortográficos, informações incorretas ou imprecisas, o Git Rebase permite que você edite as mensagens dos commits durante o processo de rebase.
- **Resolução de Conflitos:** Ao realizar um rebase, é possível que ocorram conflitos entre os commits. O Git Rebase oferece a oportunidade de resolver esses conflitos durante o processo, garantindo que o histórico reorganizado seja livre de problemas.

Como executar um git rebase

1. Atualize o seu branch principal (ou qualquer outro sobre o qual você deseja fazer o rebase)
2. Entre no seu branch de trabalho
3. Execute o comando `git rebase <branch_base>`
4. Resolva os conflitos, se necessário
 - a. Após resolver cada conflito, utilize o comando `git add` para marca-lo como resolvido
 - b. Continue o rebase usando o comando `git rebase --continue`
5. Caso queira, suba a sua ramificação de trabalho usando o `git push`

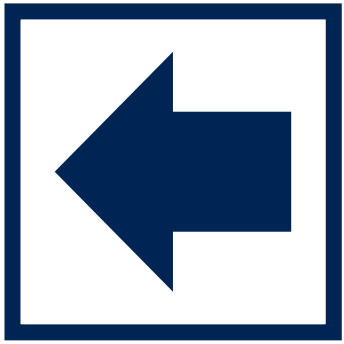
Fazendo cherry-picks

O comando `git cherry-pick` permite copiar commits individuais de uma ramificação para outra. Isso é útil quando você deseja incorporar um commit específico de uma ramificação em outra sem a necessidade de reescrever todo o histórico.

Enquanto o rebase move todo um branch para o topo de um novo branch, com o cherry-pick é possível fazer uma cópia de commits específicos, que tenham mudanças importantes para o desenvolvimento, mas que os demais commits daquele branch não sejam relevantes no momento.

Para fazer um cherry-pick em um branch de desenvolvimento, basta passar o comando `git cherry-pick <hash_do_commit>`. Se houver conflitos, resolva da mesma forma que um merge ou rebase, e continue com `git cherry-pick --continue`. Finalizando os conflitos, basta usar o push.

No cherry-pick, o commit original permanece intacto na ramificação de origem, e você terá uma cópia dessas alterações na ramificação de destino.



Aula 10: Fluxos de trabalho no Git

Fluxos de trabalho?

É corriqueiro pessoas utilizarem apenas um branch para fazer commits em projetos pessoais, o que não é errado, pois quando estamos trabalhando sozinhos é muito tranquilo de se controlar tudo em um branch só.

Entretanto, o cenário se torna totalmente diferente e mais complexo quando estamos trabalhando com mais contribuidores em um projeto.

Em todo projeto real, é importantíssimo que se tenha controle total do que está sendo produzido por uma equipe de pessoas desenvolvedoras, onde, ao mesmo tempo, são feitas muitas coisas, como: implementação de novas funcionalidades, correção de falhas, lançamento de versões, etc. E é justamente aqui, que o Git Flow entra para nos ajudar, facilitando o desenvolvimento compartilhado de código com pessoas desenvolvedoras.

Fluxos de trabalho?

Portanto, ao utilizar o Git para gerenciar versões de um projeto, é de extrema importância definir, entre os desenvolvedores do projeto, qual será o **fluxo de trabalho** (ou *workflow*) para aquele projeto, ou seja, como os branches serão criados e conectados ao longo do processo de desenvolvimento.

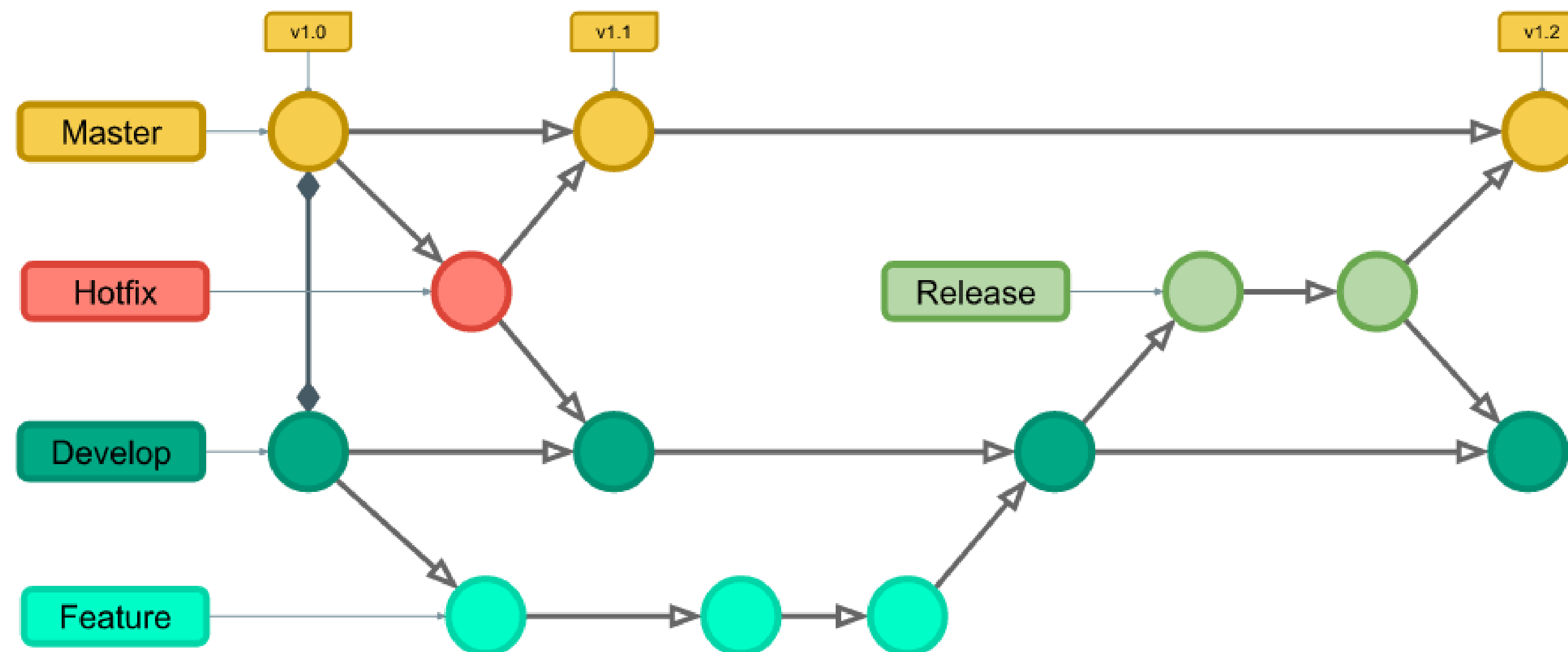
Cada time adota um workflow que melhor se adeque às suas necessidades. Muitas vezes, são fluxos próprios, desenvolvido ao longo do tempo e que atende situações particulares que apenas naquele projeto, naquela equipe e naquela empresa se repetem com frequência e precisam ser consideradas.

No entanto, a literatura apresenta uma série de fluxos “padrão” para se gerenciar o trabalho no Git, cada um com suas vantagens, desvantagens e aplicações. Veremos aqui três desses fluxos, o **Git Flow**, o **GitHub Flow** e o **Trunk-based development**.

Git Flow

O Git Flow foi desenvolvido em 2010 pelo engenheiro de software holandês Vincent Driessen, e é muito aplicável nos casos de projetos que utilizem [versionamento semântico](#) ou que precisam oferecer suporte a várias versões do software.

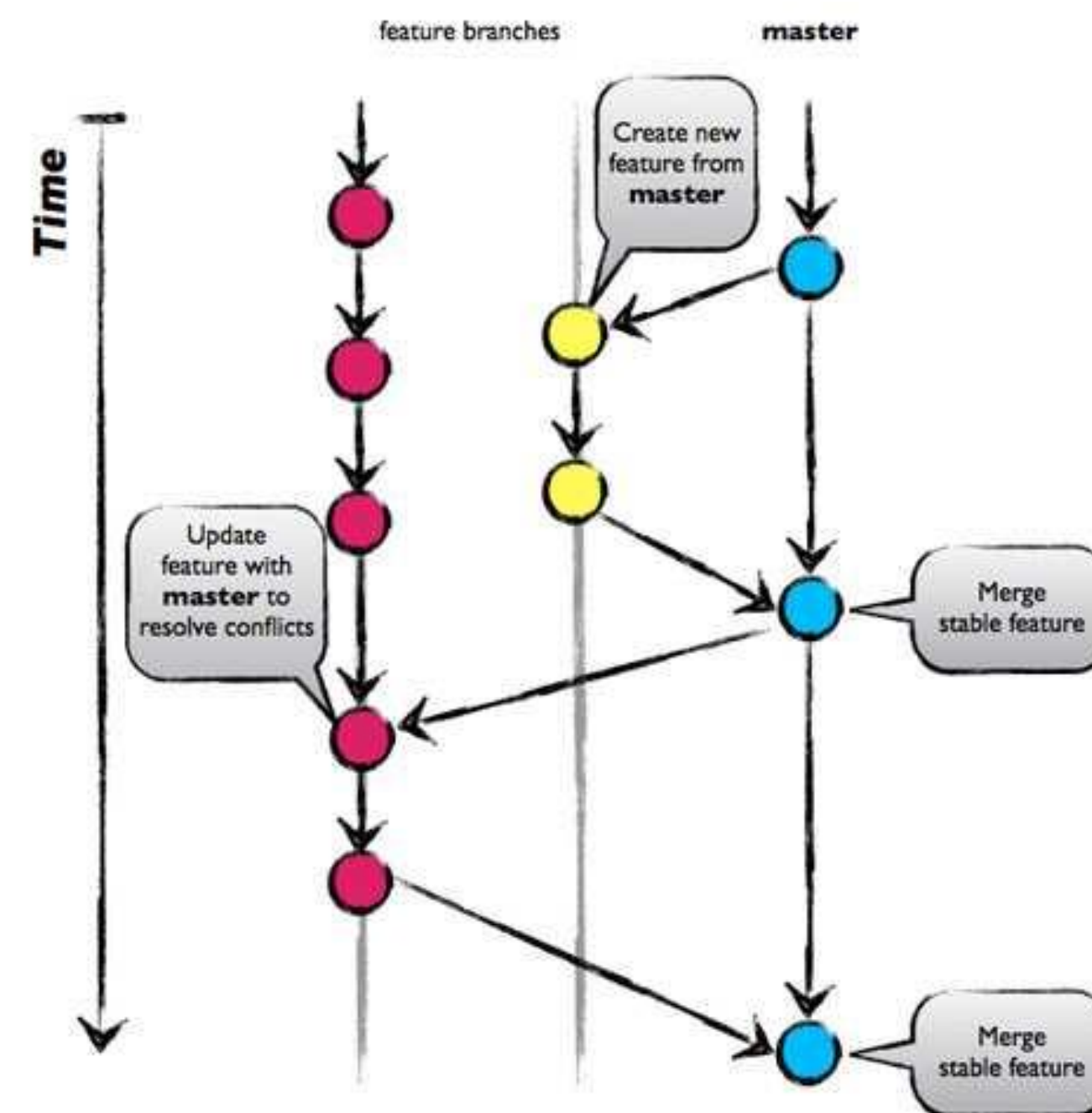
Ele não é recomendado em projetos que aplicam práticas de entrega contínua, uma vez que é uma premissa desse fluxo manter branches ativos durante um longo período de tempo.



GitHub Flow

O GitHub Flow é uma versão simplificada do Git Flow, que elimina o uso dos branches development, release e hotfix, mantendo apenas branches de feature e o principal.

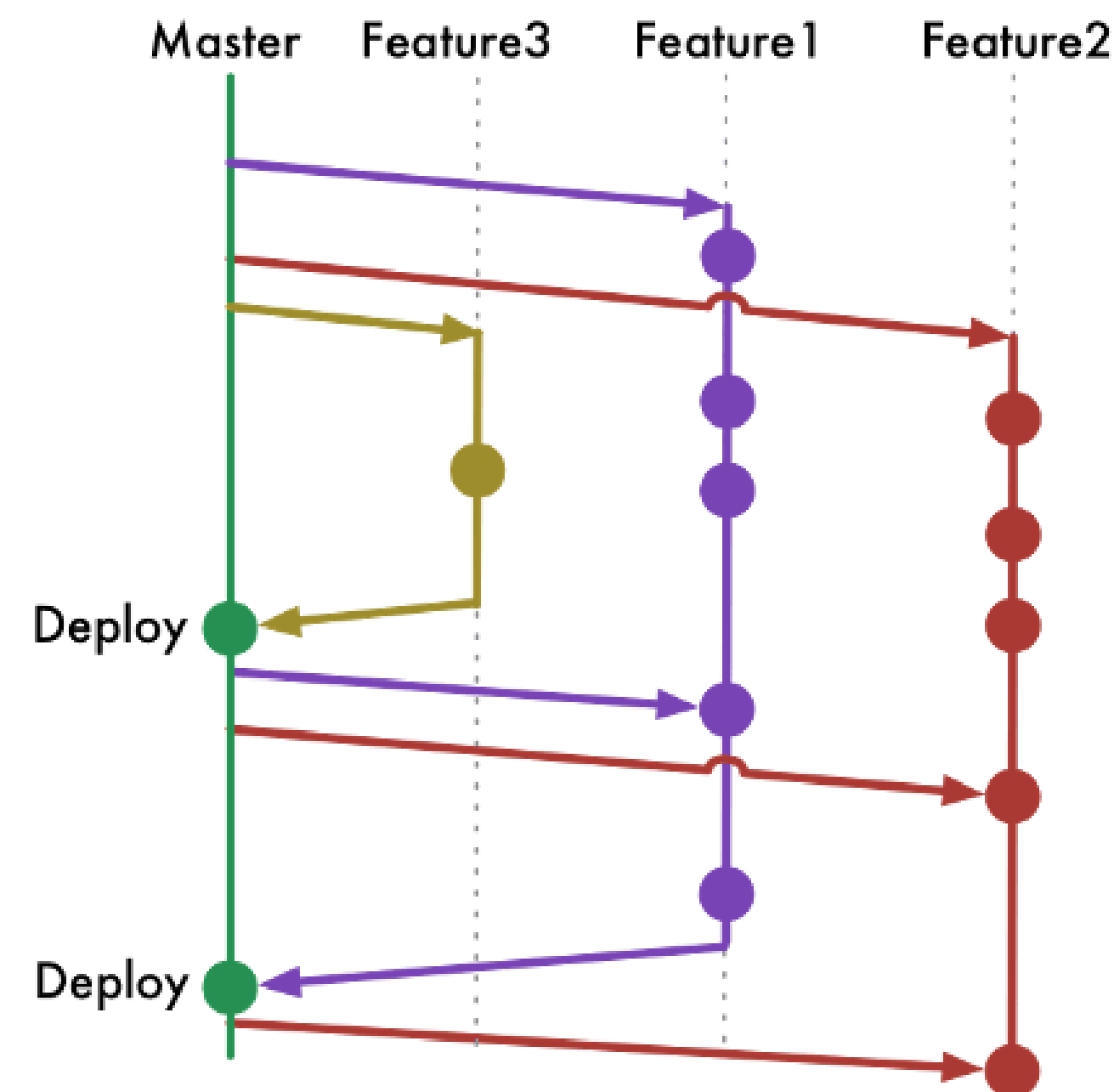
Apesar de parecer tentador sempre utilizar esse fluxo pela sua simplicidade, em casos de projetos complexos é exigido da equipe de desenvolvimento um alto nível de maturidade técnica, uma vez que erros podem ser passados para o branch principal.

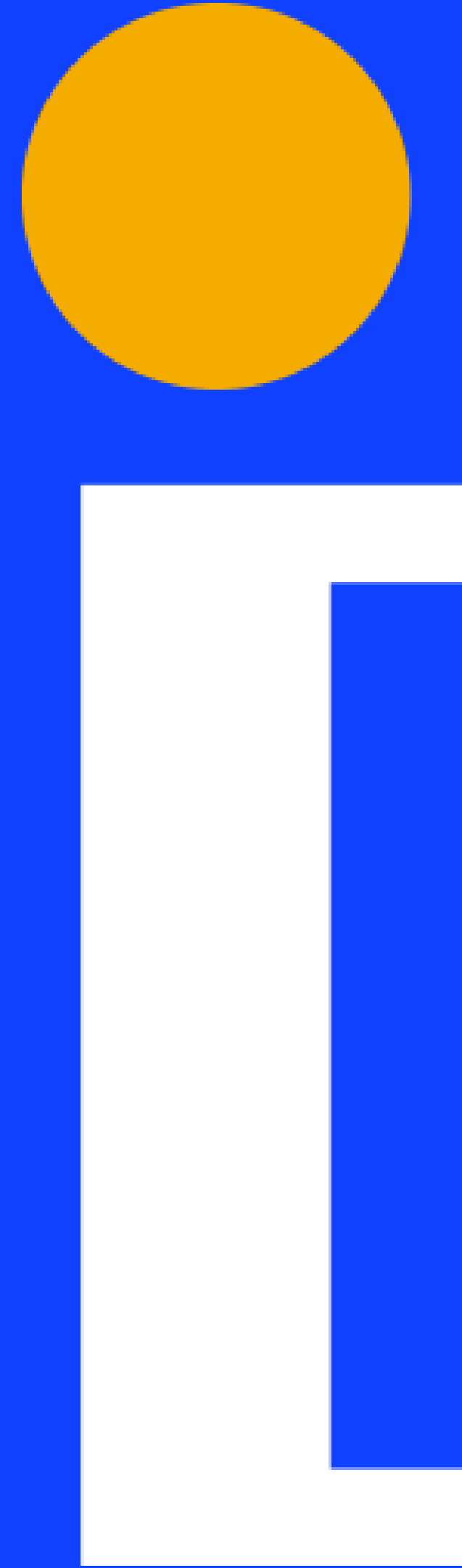


Trunk-based development

O trunk-based development (TBD) é focado na entrega e integração contínuas (CI/CD), e muito utilizado em times de DevOps. É especialmente adequado para projetos em que as atualizações frequentes e a implantação contínua são essenciais.

O TBD exige da equipe de desenvolvimento um alto nível de maturidade técnica, uma vez que erros podem ser passados para o branch principal.





IBMEC.BR

 /IBMEC

 IBMEC

 @IBMEC_OFICIAL

 @IBMEC

 **ibmec**