

Table of Contents

Git Pie: Aprenda sobre VCS	2
Conceitos Básicos de Versionamento	9
Tipos de Sistemas de Controle de Versão	15
Sistemas de Controle de Versão Local	21
Sistemas de Controle de Versão Centralizado	29
Sistemas de Controle de Versão Distribuído	38
Comparando Sistemas de Controle de Versão	50
História do Controle de Versão	56
Controle de Versão Moderno	62
Fluxos de Trabalho em Versionamento	64
Trunk-Based Development	67
Feature Branch Workflow	71
Gitflow Workflow	78
Melhores Práticas em Controle de Versão	86
Terminologia do Controle de Versão	89
Segurança em Controle de Versão	92
História do Git	95
Conceitos Básicos do Git	98
Fluxo de Trabalho do Git	100
Comandos Essenciais do Git	102
Links e Referências	104

Git Pie: Aprenda sobre VCS



American pie

Nota do Autor

Olá pessoas, nesse texto irei falar sobre VCS (Sistema de Versionamento de Código, sigla em inglês) ou melhor, como o tema é mais conhecido - falarei sobre Git.

O que você vai aprender aqui?



Stifler teaching

"Deixa que o Stifler te explica essa parada!"

Nesse guia você vai aprender:

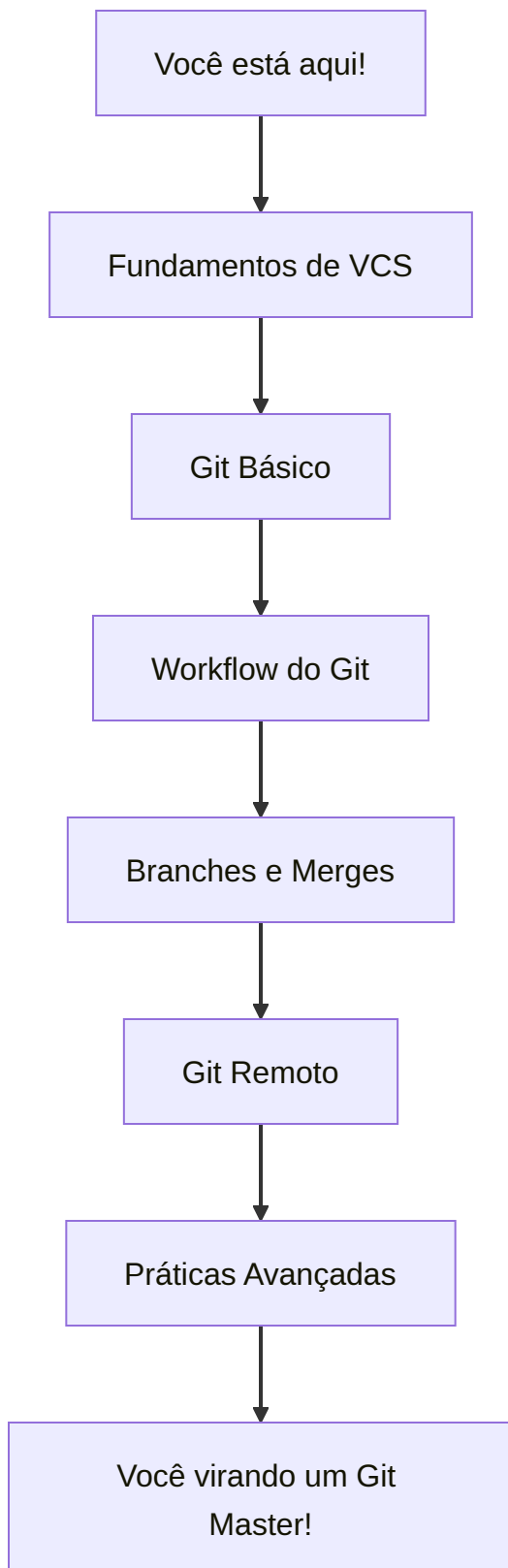
- Como não perder código igual perdeu aquela crush do ensino médio
- Como trabalhar em equipe sem querer matar seus colegas
- Como versionar código igual um profissional (e não usando projeto-final-v3-agora-vai-mesmo.zip)
- Como usar Git e não passar vergonha nas entrevistas de emprego

Roadmap de Aprendizado



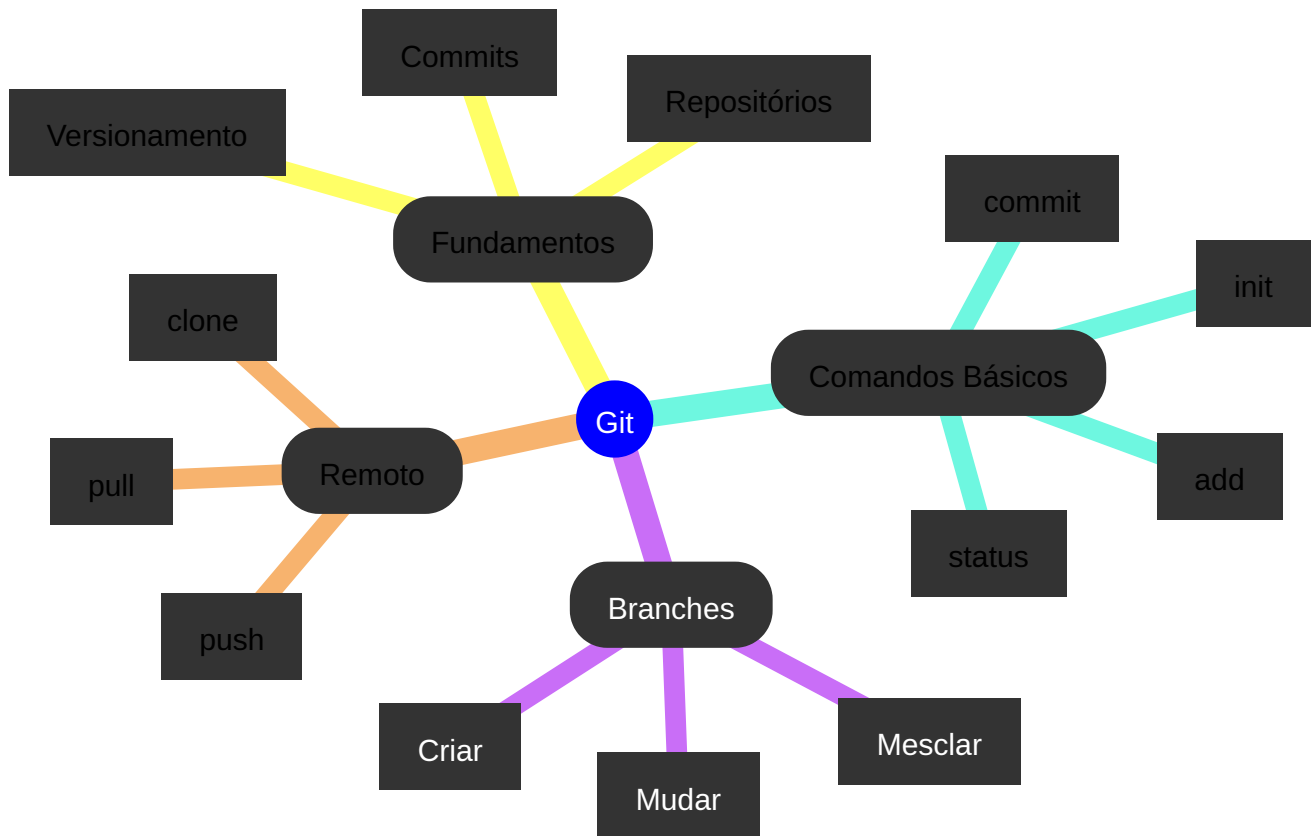
American pie road

A estrada do conhecimento é longa, mas é divertida!



Mapa Mental dos Conceitos

⚠ Para você que gosta de ver o todo antes de se perder nos detalhes (tipo quando você olha o cardápio inteiro antes de pedir)



Por que você deveria aprender Git?



Stifler convinced

"Confia no pai que essa é boa!"

Imagina só:

- Você tá lá, codando tranquilo
- Fez alterações MASSAS no projeto
- Aí seu PC resolve dar aquela travada marota
- E... BOOM! 💥 Perdeu tudo!

Ou pior:

- Você e seu amigo precisam trabalhar no mesmo projeto
- Vocês ficam trocando arquivo por WhatsApp
- projeto_final.zip, projeto_final_v2.zip, projeto_final_v2_agora_vai.zip
- No final ninguém sabe qual é a versão certa 🧑

É aí que entra o Git! Ele é tipo aquele amigo que:

- Guarda todas as versões do seu código
- Deixa você voltar no tempo quando der m*rda
- Permite que você e seus amigos trabalhem juntos sem criar caos
- Te salva de passar vergonha em entrevistas de emprego

Pré-requisitos



Jim thinking

"O que eu preciso saber antes de começar?"

- Saber usar um terminal básico (tipo `cd`, `ls`, essas coisas)
- Ter um editor de código (VSCode, Sublime, ou qualquer outro que você curta)
- Vontade de aprender (e senso de humor para aguentar minhas piadas ruins)

Como usar este guia

Este material está organizado de forma progressiva:

1. Começamos com o básico dos básicos
2. Vamos evoluindo aos poucos
3. No final você estará usando Git igual um profissional



Dica do Stifler: Não pule etapas! É tipo American Pie, você precisa ver o primeiro filme antes de entender as piadas do segundo!

Bora começar?



Lets do this

É hora de botar a mão na massa!

Escolha sua aventura:

- Fundamentos de Versionamento ([Conceitos Básicos de Versionamento](#)) - Para entender o básico
- História do Git ([História do Git](#)) - Para os curiosos
- Git na Prática ([Fluxo de Trabalho do Git](#)) - Para quem quer ir direto ao código



Nota: Se em algum momento você se perder, não se preocupe! É normal, todo mundo já passou por isso. Até o Stifler já perdeu código antes de aprender Git!

Conceitos Básicos de Versionamento

Versionamento de Código

Versionamento é um conceito muito simples e usado no dia a dia de forma que nem percebemos. Por exemplo: Estamos em um projeto onde temos dois desenvolvedores:

- Stifler



Stifler dude no

- Jim



Jim american pie

Esses dois desenvolvedores estão fazendo o "Milfs Go" uma especie revolucionaria e inovadora, além do tempo sendo um *app* para acharem a "milfs".



Aqui está uma *milf* para aqueles não habituados com o termo:



American pie good stuff

Controle de Versão

Versionamento é o ato de manipular versões, agora o Controle de Versão é um sistema que vai registrar as mudanças tanto num arquivo como em um projeto gigante ao longo do tempo.

Tipos de Controle de Versão

1. Local

- Mantém as versões apenas na sua máquina
- Simples mas limitado
- Exemplo: copiar e renomear arquivos

2. Centralizado

- Um servidor central guarda todas as versões
- Todos se conectam a este servidor
- Exemplo: SVN

3. Distribuído

- Cada desenvolvedor tem uma cópia completa
- Trabalho offline possível
- Exemplo: Git

Importância

Talvez agora você levante uma questão de o porque aprender "este trem" - como diria um amigo mineiro. Logo, a resposta é simples: esse tipo de ferramenta é essencial para o desenvolvimento já que nos entrega um poder de não somente trabalhar em conjunto de forma assíncrona e sem medo de acabar perdendo o que já foi feito.

Benefícios do Controle de Versão

1. Histórico Completo

- Rastreamento de todas as mudanças
- Quem fez o quê e quando
- Possibilidade de reverter alterações

2. Trabalho em Equipe

- Múltiplos desenvolvedores
- Desenvolvimento paralelo
- Resolução de conflitos

3. Backup

- Cópia segura do código
- Recuperação de desastres
- Múltiplas cópias distribuídas

Fluxo Básico

1. Modificação

- Alteração nos arquivos
- Criação de novos arquivos
- Exclusão de arquivos

2. Stage

- Preparação das mudanças
- Seleção do que será versionado
- Organização das alterações

3. Commit

- Confirmação das mudanças
- Criação do ponto de versão
- Registro no histórico

Boas Práticas

1. Commits Frequentes

- Mudanças pequenas e focadas
- Mais fácil de entender e reverter
- Melhor rastreabilidade

2. Mensagens Claras

- Descreva o que foi alterado
- Seja conciso mas informativo
- Use tempo verbal consistente

3. Branches Organizados

- Separe features em branches

- Mantenha o main/master estável
- Merge apenas código testado

Próximos Passos

Agora que você entende os conceitos básicos, está pronto para:

- Aprender comandos específicos do Git
- Entender branches e merges
- Trabalhar com repositórios remotos

Próximo Capítulo: Git Básico ([Conceitos Básicos do Git](#))



Dica: Mantenha este capítulo como referência! Os conceitos básicos são fundamentais para entender as operações mais avançadas que virão pela frente.

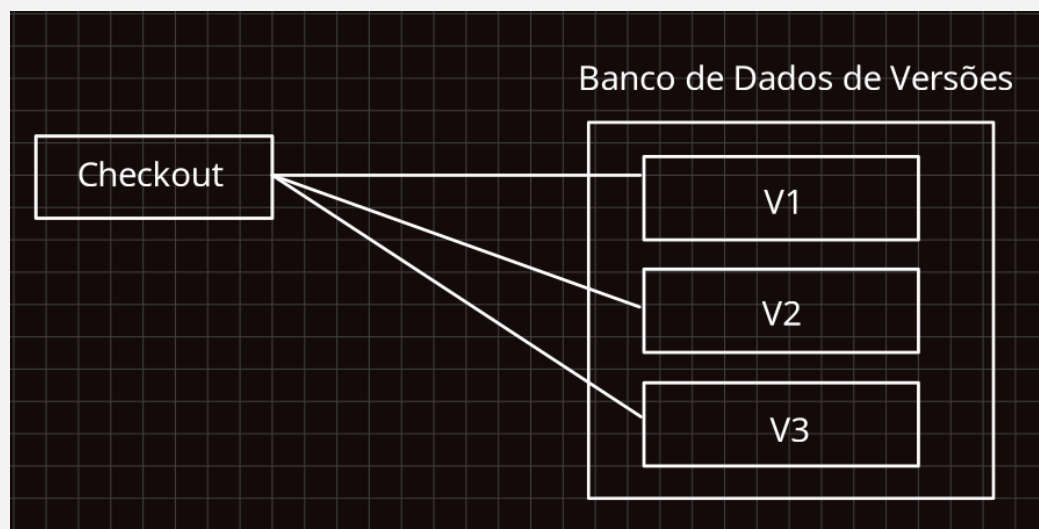
Tipos de Sistemas de Controle de Versão

Sistemas Locais

Imagine que o Stifler está tentando escrever a "bíblia das milfs" em seu computador. Toda vez que ele faz uma alteração importante, cria uma nova pasta chamada "versão_final", "versão_final_2", "versão_final_2_agora_vai"... Isso é basicamente um sistema local de controle de versão!

Características dos Sistemas Locais

- **Simplicidade:** Tão simples quanto renomear arquivos
- **Independência:** Funciona offline, como o Stifler escrevendo sozinho em casa
- **Limitações:** Se o HD queimar, tchau bíblia das milfs
- **Risco:** Um problema no computador e todo o histórico se perde



Version control system sistema local

Diagrama de um sistema local (ou como Stifler organiza seus arquivos)

Analogia da Festa

É como fazer uma festa sozinho. Você tem todo o controle, mas:

- Ninguém mais participa
- Se sua casa pegar fogo, acabou a festa
- Você não pode estar em dois lugares ao mesmo tempo

Sistemas Centralizados

Agora imagine que Jim e Stifler decidem trabalhar juntos no "Milfs Go". Eles precisam de um lugar central para guardar o código - tipo a casa da mãe do Stifler (que ironicamente é uma milf).

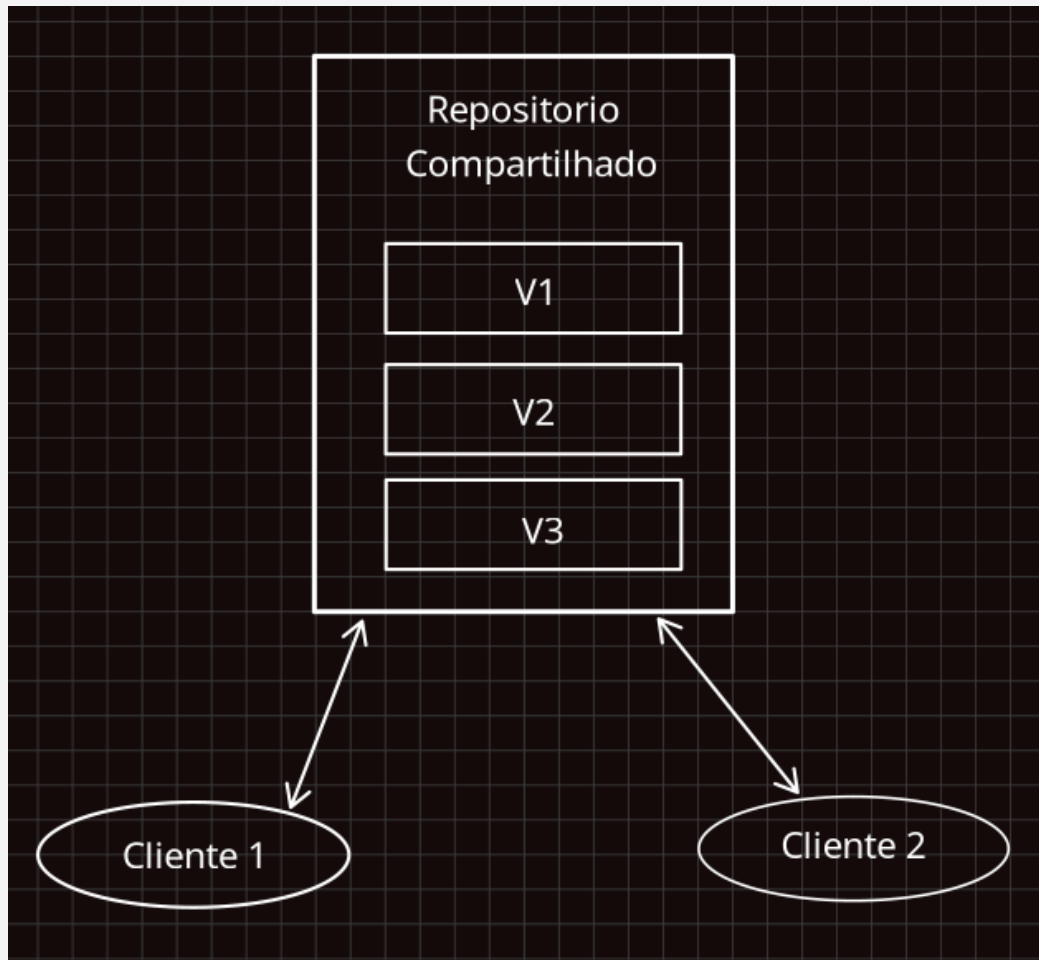
Como Funciona

- Um servidor central (a casa da mãe do Stifler)
- Todos os desenvolvedores se conectam a ele
- Precisa de internet para trabalhar

Desvantagens dos Sistemas Centralizados

- **Ponto único de falha:** Se a mãe do Stifler sair de casa, ninguém trabalha
- **Dependência de rede:** Sem internet, sem código
- **Performance:** Lento como Stifler tentando resolver cálculo
- **Conflitos:** Como Jim e Stifler brigando pelo mesmo arquivo





Version control system sistema compartilhado

Diagrama de um sistema centralizado (ou a casa da mãe do Stifler)

Analogia da Festa Centralizada

É como uma festa na casa da mãe do Stifler:

- Todo mundo precisa ir até lá
- Se a casa fechar, acabou a festa
- Só dá para fazer as coisas se você estiver lá

Sistemas Distribuídos

Finalmente, temos o sistema que é tipo a internet das milfs - todo mundo tem uma cópia completa de tudo!

Por que é Melhor?

- **Trabalho offline:** Como Stifler "estudando" em casa
- **Backup distribuído:** Cada cópia é um backup completo
- **Performance:** Rápido como Stifler correndo atrás de... você sabe
- **Flexibilidade:** Múltiplos fluxos de trabalho possíveis

Analogia da Festa Distribuída

É como ter várias festas simultâneas:

- Cada um pode ter sua própria festa
- As festas podem se sincronizar
- Se uma festa acabar, as outras continuam

Características Avançadas

1. Branches Distribuídos

- Como diferentes capítulos do "Milfs Go"
- Cada um trabalha no seu
- Depois junta tudo

2. Colaboração

- Pull requests (como pedir permissão para a mãe do Stifler)
- Code review (Jim revisando as besteiras do Stifler)
- Forks (fazer sua própria versão do "Milfs Go")

Tabela Comparativa Estilo American Pie

Característica	Local	Centralizado	Distribuído
Backup	Frágil como o ego do Stifler	Médio	Forte como a mãe do Stifler
Colaboração	Solo	Limitada	Total
Offline	Sim	Não	Sim
Complexidade	Fácil	Média	Complexa
Confiabilidade	Baixa	Média	Alta

Exemplos Históricos

Sistemas Locais (Anos 80)

- RCS: O vovô dos sistemas de versão
- SCCS: Ainda mais velho que a mãe do Stifler

Sistemas Centralizados (Anos 90-2000)

- SVN: O pai dos sistemas centralizados
- CVS: O tio que ninguém mais visita
- Perforce: O primo rico

Sistemas Distribuídos (2005+)


- Git: O rei da festa

- Mercurial: O amigo legal que ninguém lembra
- Bazaar: Aquele que tentou mas não vingou

Conclusão


Escolher um sistema de controle de versão é como escolher onde fazer a festa:

- Na sua casa (Local)
- Na casa da mãe do Stifler (Centralizado)
- Em todas as casas ao mesmo tempo (Distribuído)

 Stifler aprovando sistemas distribuídos

Nota Final

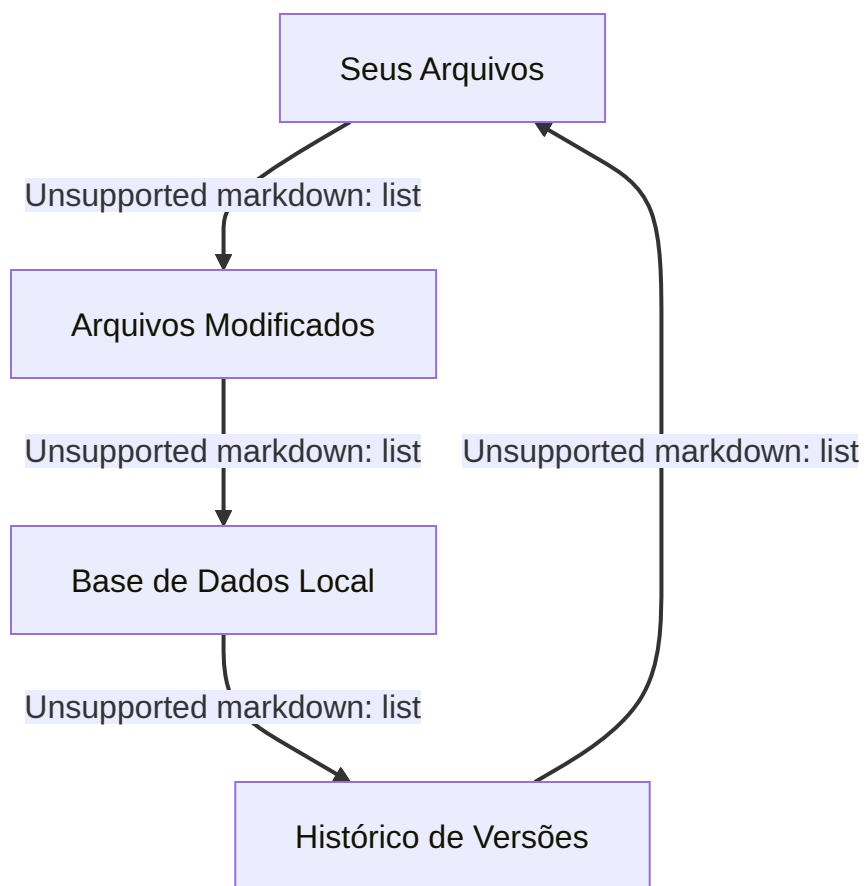
Lembre-se: assim como Stifler aprendeu a respeitar as milfs, você precisa respeitar seu sistema de controle de versão. Escolha sabiamente!

 E viveram felizes para sempre com Git

Sistemas de Controle de Versão Local

Um sistema de controle de versão local é a primeira e mais básica forma de versionamento de código. Imagine como uma máquina do tempo pessoal para seu código, onde todas as mudanças são registradas e armazenadas localmente no seu computador.

Como Funciona na Prática



Analogia com um Álbum de Fotos

```
+-----+
| Seu Projeto |
| +-----+ |
| | Versão Atual | |
```

```

| +-----+ |
| | Versão Anterior | |
| +-----+ |
| | Versões Antigas | |
| +-----+ |
+-----+

```

Componentes Principais

1. Base de Dados Local

- Armazena todas as mudanças
- Mantém metadados (autor, data, descrição)
- Gerencia diferentes versões
- Organiza o histórico completo

2. Sistema de Tracking



3. Mecanismo de Snapshots

Tempo ----->

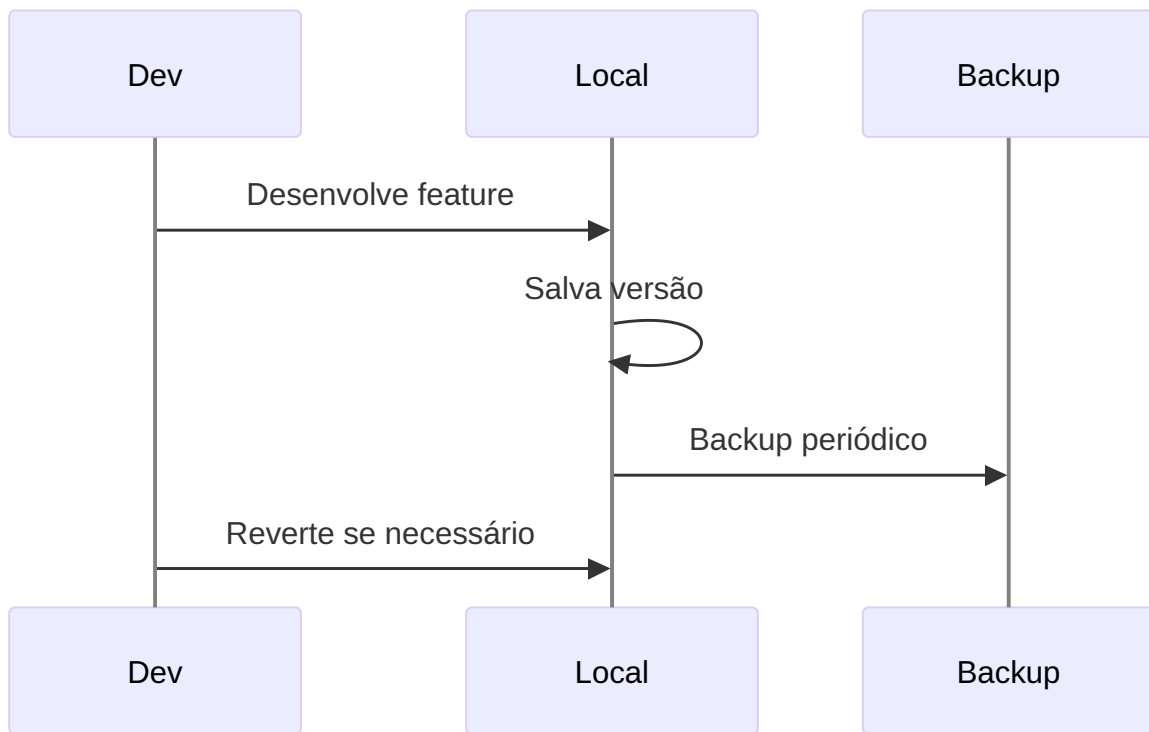
```

V1  [Snapshot 1]
    |
V2  [Snapshot 2]
    |
V3  [Snapshot 3]
    |
V4  [Snapshot 4]

```

Cenários de Uso

1. Desenvolvimento Solo

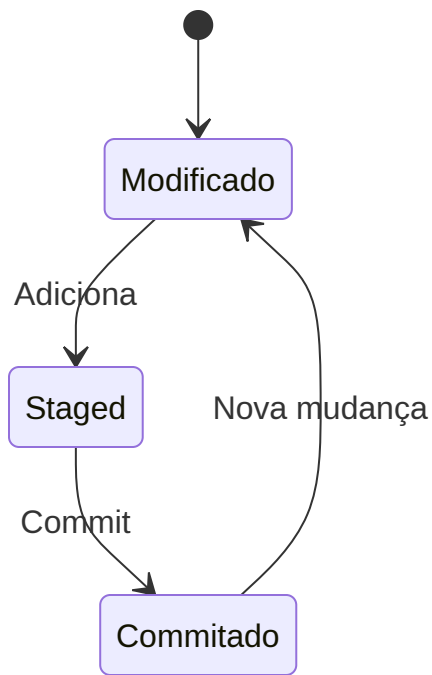


2. Projetos Pessoais

```
+-----+
| Projeto Pessoal |
|               |
| + Código       |
| + Documentação |
| + Recursos     |
| + Configurações|
+-----+
    |
    v
+-----+
| Sistema Local VCS |
+-----+
```

Processo de Versionamento

1. Criação de Versões



2. Recuperação de Versões

```
HEAD (Versão Atual)
|
v
[V3] --> [V2] --> [V1]
^
|
Checkout
```

Vantagens Detalhadas

1. Simplicidade

- Fácil de configurar
- Sem dependências externas
- Interface simples
- Aprendizado rápido

2. Performance

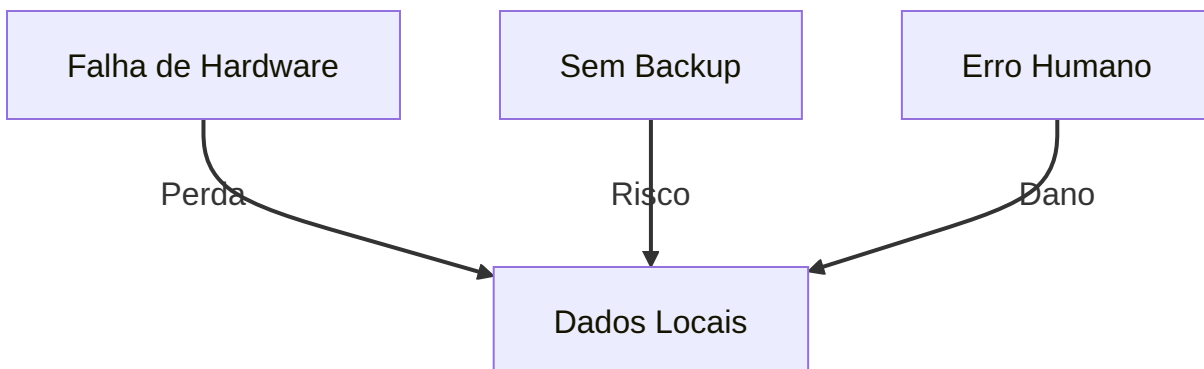


3. Autonomia

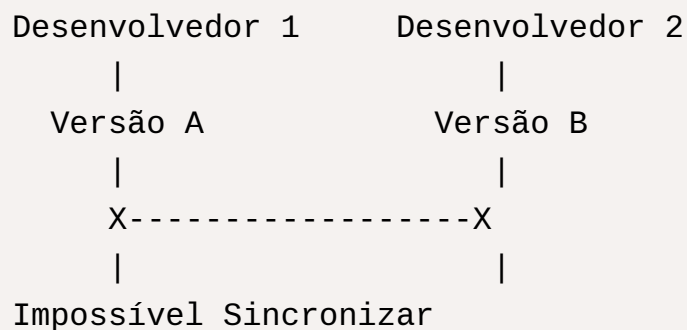
- Trabalho offline
- Controle total
- Independência de rede
- Decisões imediatas

Limitações Detalhadas

1. Riscos de Perda

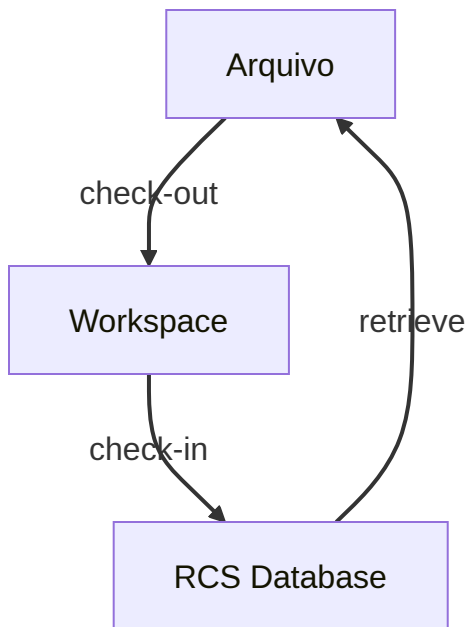


2. Colaboração Limitada



Ferramentas Populares

1. RCS (Revision Control System)

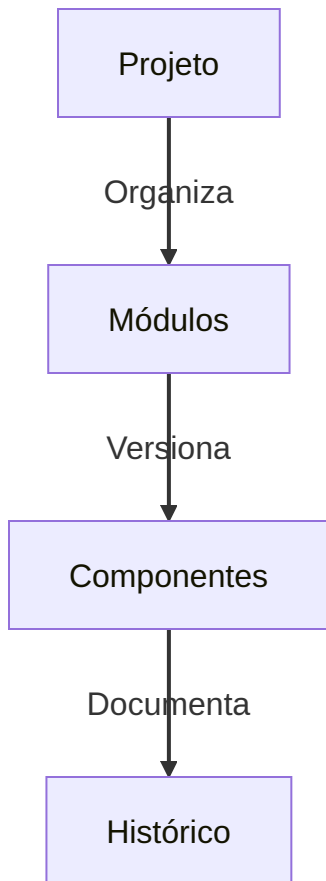


2. SCCS (Source Code Control System)

```
+-----+
| SCCS Structure |
|               |
| s.file1       |
| s.file2       |
| s.file3       |
+-----+
```

Melhores Práticas

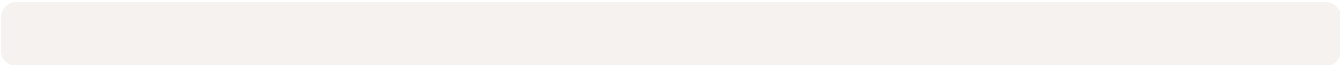
1. Organização



2. Backup Regular

```
+-----+
| Projeto Local |
+-----+
      |
      v
+-----+
| Backup Externo |
+-----+
      |
      v
+-----+
| Cloud Storage  |
+-----+
```

3. Documentação

- Comentários claros
 - Descrições de versão
 - Registro de mudanças
 - Notas de implementação
- 

Sistemas de Controle de Versão Centralizado

Um sistema de controle de versão centralizado (CVCS) é como uma festa na casa da mãe do Stifler - todos precisam ir ao mesmo lugar para participar! Este sistema utiliza um servidor central que armazena todos os arquivos versionados e permite que múltiplos desenvolvedores colaborem no mesmo projeto.

Características Principais

1. Servidor Central

- Repositório único e autoritativo
- Controle de acesso centralizado
- Backup centralizado
- Administração simplificada

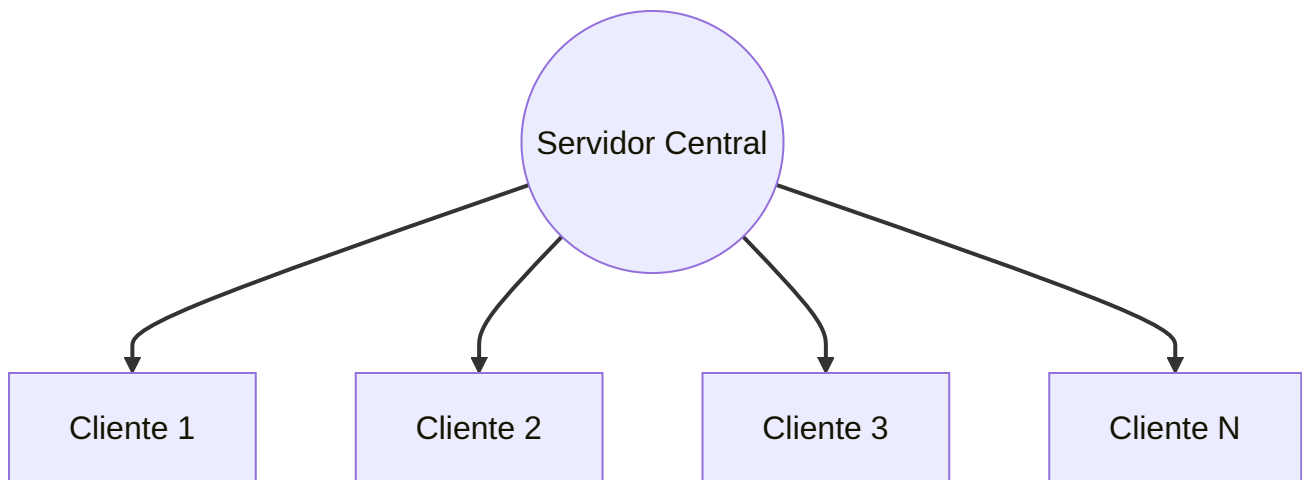
2. Clientes

- Checkout de arquivos específicos
- Histórico parcial
- Dependência de conectividade
- Workspace local limitado

A Casa da Mãe do Stifler

Como uma festa na casa da mãe do Stifler, todos precisam ir ao mesmo lugar para participar!

Arquitetura



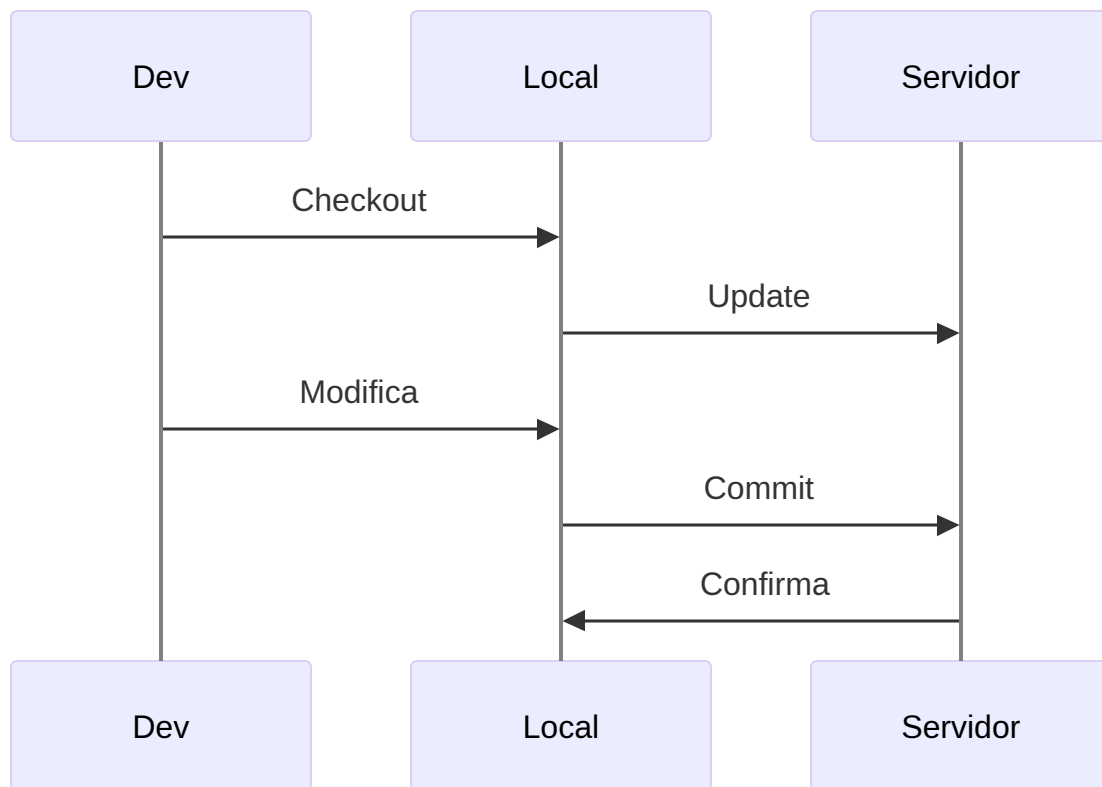
Estrutura do Sistema

```

+-----+
|  Servidor  |
|  Central   |
+-----+
      ||||
+-----++-----+
|           |           |
+-----+ +-----+ +-----+
|Cliente| |Cliente| |Cliente|
|  1    | |  2    | |  3    |
+-----+ +-----+ +-----+
  
```

Esta representação textual utiliza caracteres de texto para simular a estrutura visual do sistema. No topo, uma caixa retangular encerra o "Servidor Central". Quatro linhas de barras verticais ("||||") indicam a conexão entre o servidor e os clientes. Abaixo, três caixas retangulares representam os clientes, rotulados "Cliente 1", "Cliente 2" e "Cliente 3".

Fluxo de Operações



Vantagens e Desvantagens

Vantagens

1. Controle Centralizado

- Governança simplificada
- Políticas uniformes
- Backup único
- Auditoria facilitada

2. Administração Simples

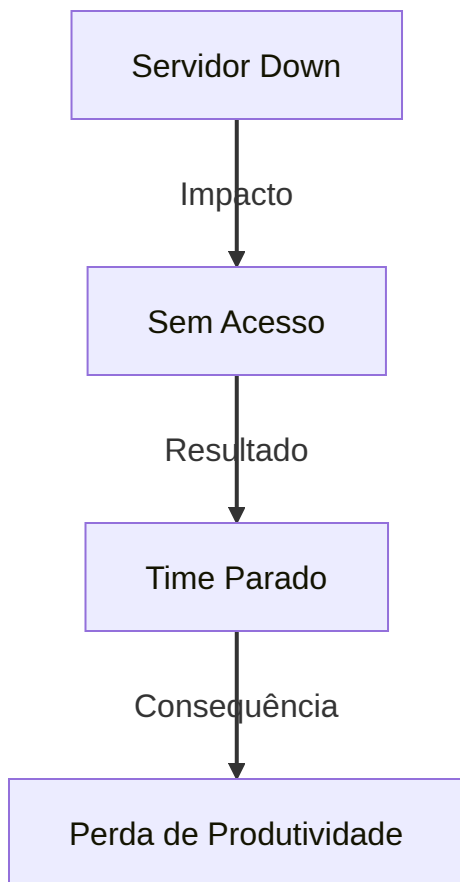
- Gerenciamento de usuários
- Controle de permissões
- Monitoramento de uso
- Manutenção única

3. Visibilidade do Projeto

- Visão única do projeto
- Status em tempo real
- Progresso transparente
- Colaboração sincronizada

Desvantagens

1. Ponto Único de Falha



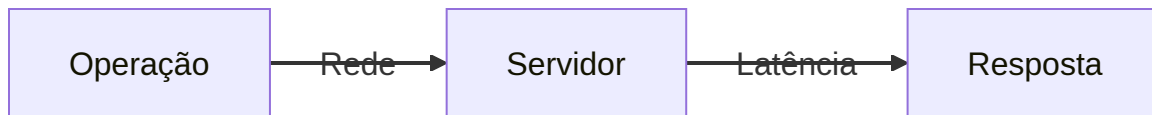
2. Dependência de Rede

Servidor

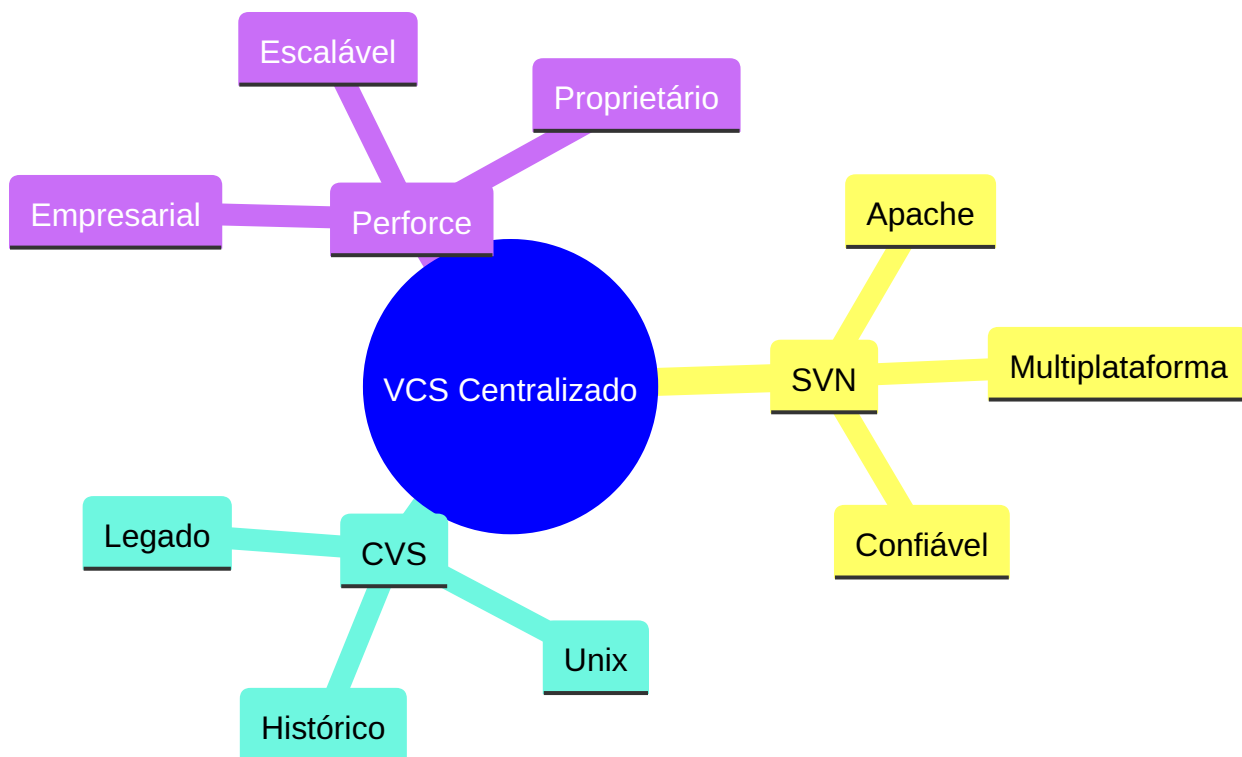
^
|

X (Conexão Perdida)
|
Cliente

3. Performance Limitada



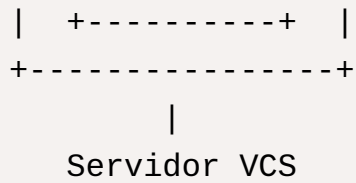
Exemplos Famosos



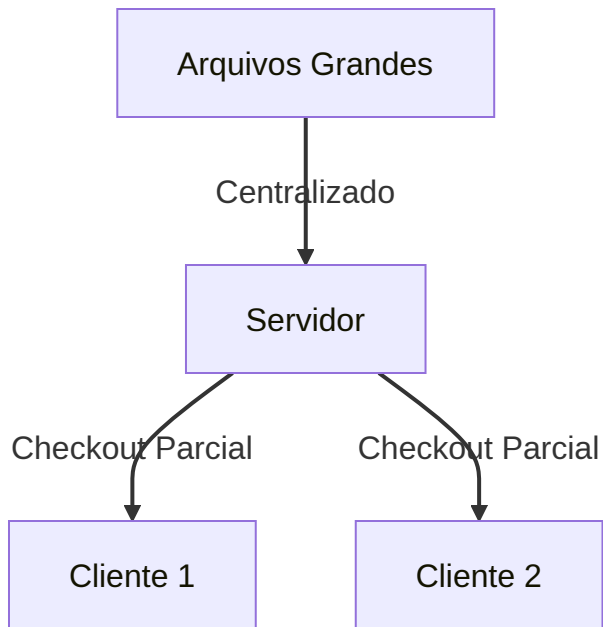
Casos de Uso Ideais

1. Equipes Localizadas

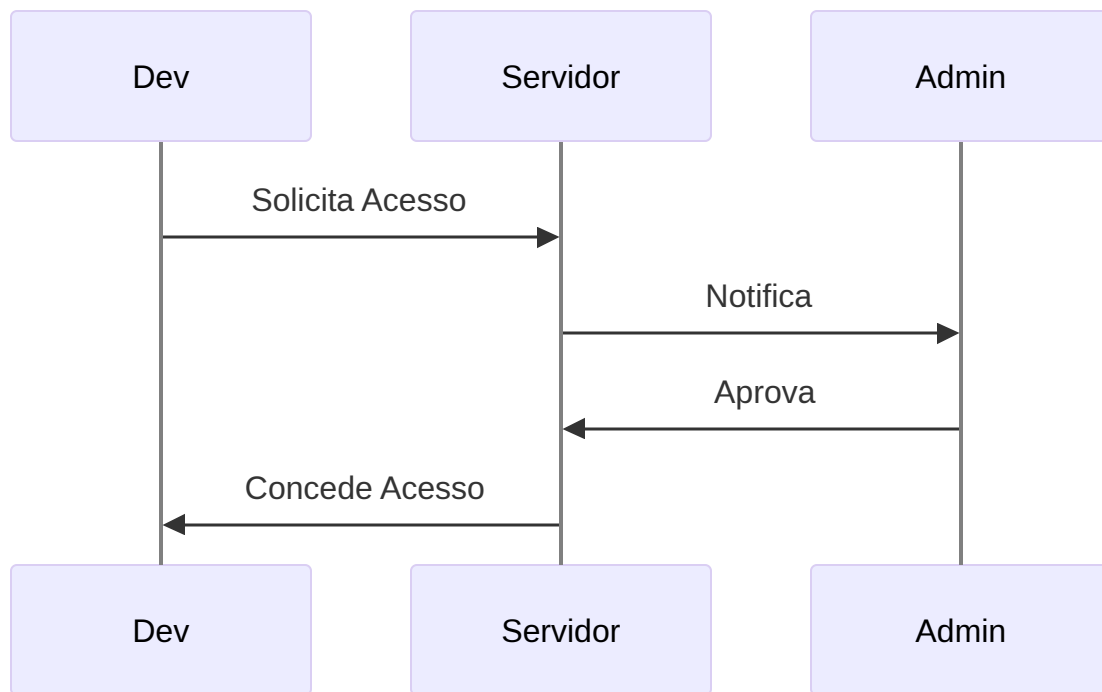
```
+-----+
|  Escritório  |
| +-----+   |
| | Time Dev | |
```



2. Projetos com Ativos Grandes



3. Controle Rigoroso

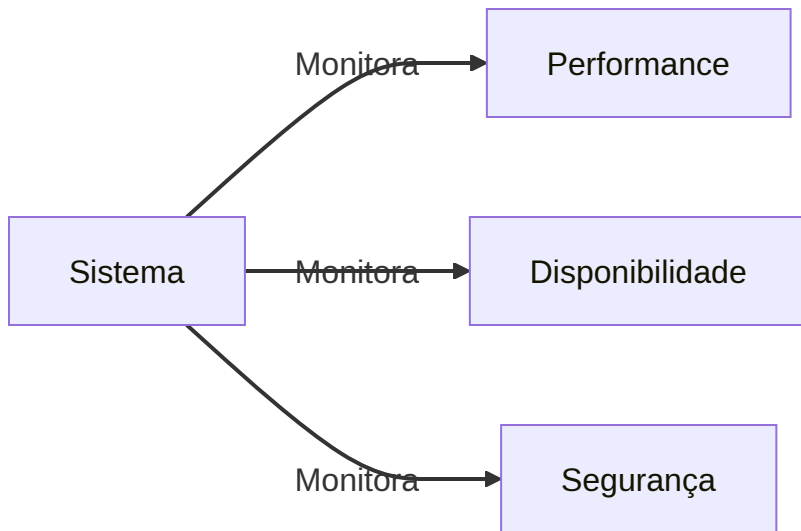


Melhores Práticas

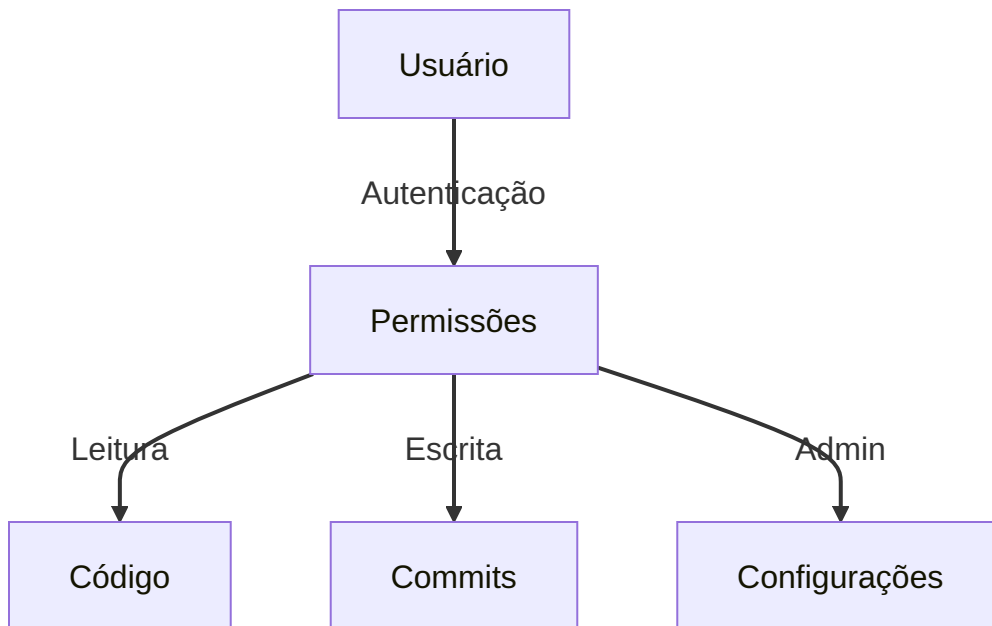
1. Backup Regular

```
Servidor Principal
|
v
Backup Diário
|
v
Backup Offsite
```

2. Monitoramento



3. Políticas de Acesso



Ferramentas de Suporte

1. Integração Contínua

```

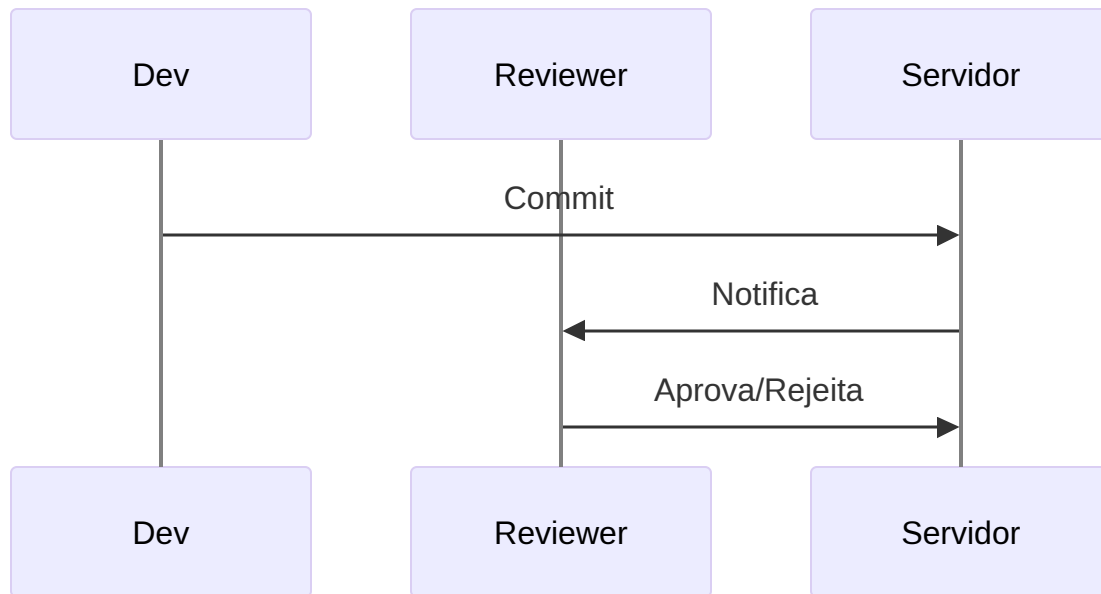
+-----+
| Build Server |
| +-----+ |
| | CI/CD   | |
| |         | |
  
```

```

| +-----+ |
+-----+

```

2. Code Review



3. Rastreamento de Issues

```

+-----+
| Issue Tracker |
| #123 Bug      |
| #124 Feature  |
| #125 Task     |
+-----+

```

Sistemas de Controle de Versão Distribuído

A Rede Social das Milfs

Sabe aquela rede social onde todo mundo tem sua própria cópia das fotos e vídeos? Pois é, um sistema distribuído é exatamente assim! Cada desenvolvedor tem uma cópia completa do projeto, como se cada um tivesse sua própria festa particular.

Por que é tipo uma Rede Social?

Todo Mundo tem Tudo

Imagine que o Stifler, o Jim e o Finch estão trabalhando juntos. Cada um tem uma cópia completa do projeto no seu computador. É como se cada um tivesse baixado todas as fotos e vídeos da festa - ninguém depende do celular dos outros pra ter acesso às memórias da noite.

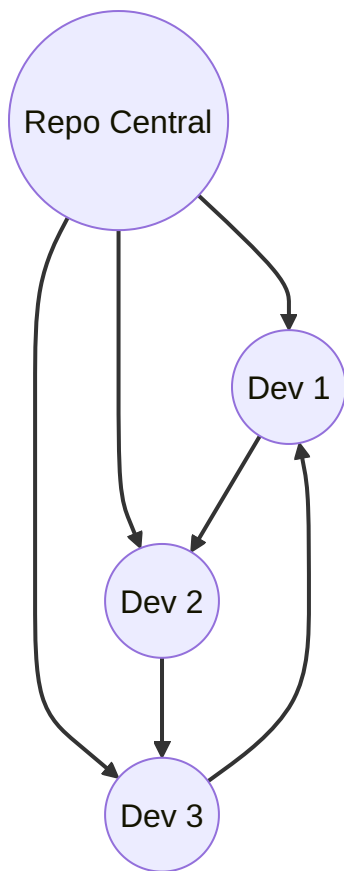
Trabalho Offline? Pode Sim!

Diferente do sistema centralizado (onde todo mundo depende da casa da mãe do Stifler), aqui cada um pode trabalhar no seu canto. O Jim pode codar mesmo quando sua internet cair, o Finch pode fazer alterações no ônibus, e o Stifler... bem, ele pode programar onde ele quiser (provavelmente enquanto procura milfs no Tinder).

Compartilhando as Novidades

Quando alguém quer mostrar seu trabalho, é só dar um "push" (tipo postar na rede social). E quando quer ver o que os outros fizeram? Dá um "pull" (como dar aquela stalkeada básica no feed dos amigos).

Conceito Básico



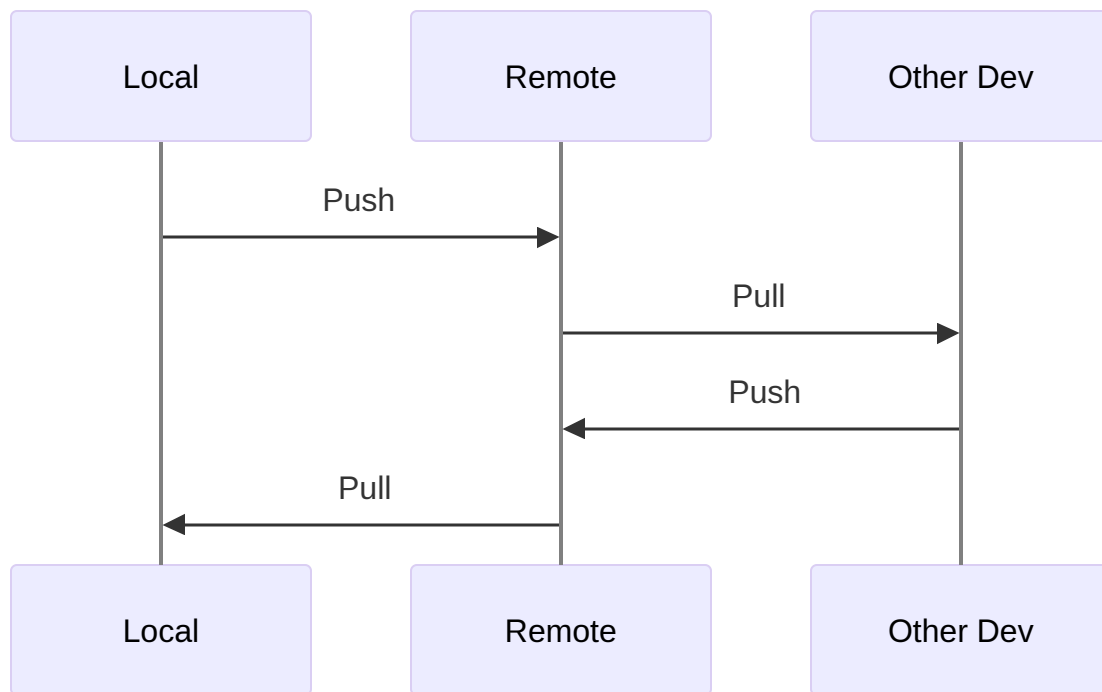
Estrutura Distribuída

```

+-----+
|  Repositório  |
|   Remoto     |
+-----+
/      |      \
+-----+ +-----+ +-----+
| Clone | |Clone| | Clone |
|  1   | |  2   | |  3   |
+-----+ +-----+ +-----+

```

Fluxo de Trabalho

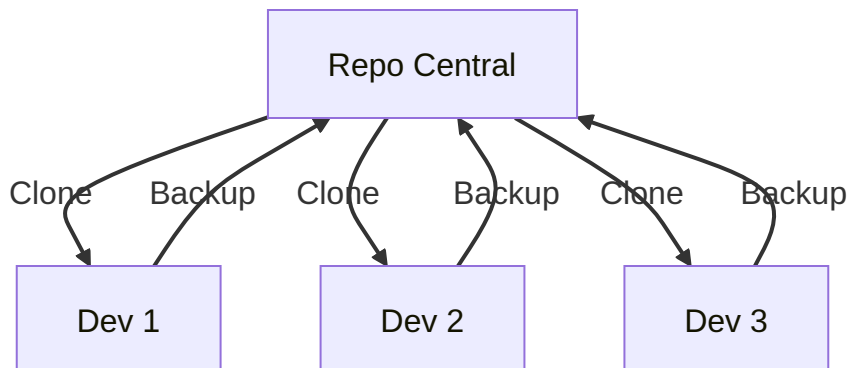


Características Principais

1. Independência Total

- Trabalho offline como um campeão
- Commits locais sem depender de ninguém
- Sua festa, suas regras

2. Backup Distribuído



3. Performance Aprimorada

Local Operations

⚡ SUPER RÁPIDO ⚡

- └─ Commits
- └─ Branches
- └─ History
- └─ Diffs

Vantagens de Ter Sua Própria Festa

1. Independência Total

- Faça commits sem precisar de internet
- Crie branches experimentais sem medo
- Trabalhe no seu ritmo
- Teste coisas malucas sem ninguém saber

2. Backup em Todo Lugar

Lembra quando o Stifler perdeu todas as fotos da festa porque derrubou cerveja no computador? Com DVCS isso não seria um problema! Como todo mundo tem uma cópia completa, é praticamente impossível perder o código. É tipo ter backup até no backup do backup.

3. Performance Insana

Quase tudo é local, então é mais rápido que o Stifler correndo atrás de uma milf. Commits, branches, histórico - tudo acontece na velocidade da luz porque não precisa ficar perguntando pro servidor.

Como Funciona na Prática?

O Dia a Dia

1. **Clone:** Primeiro você clona o repositório - é tipo fazer o download da festa inteira

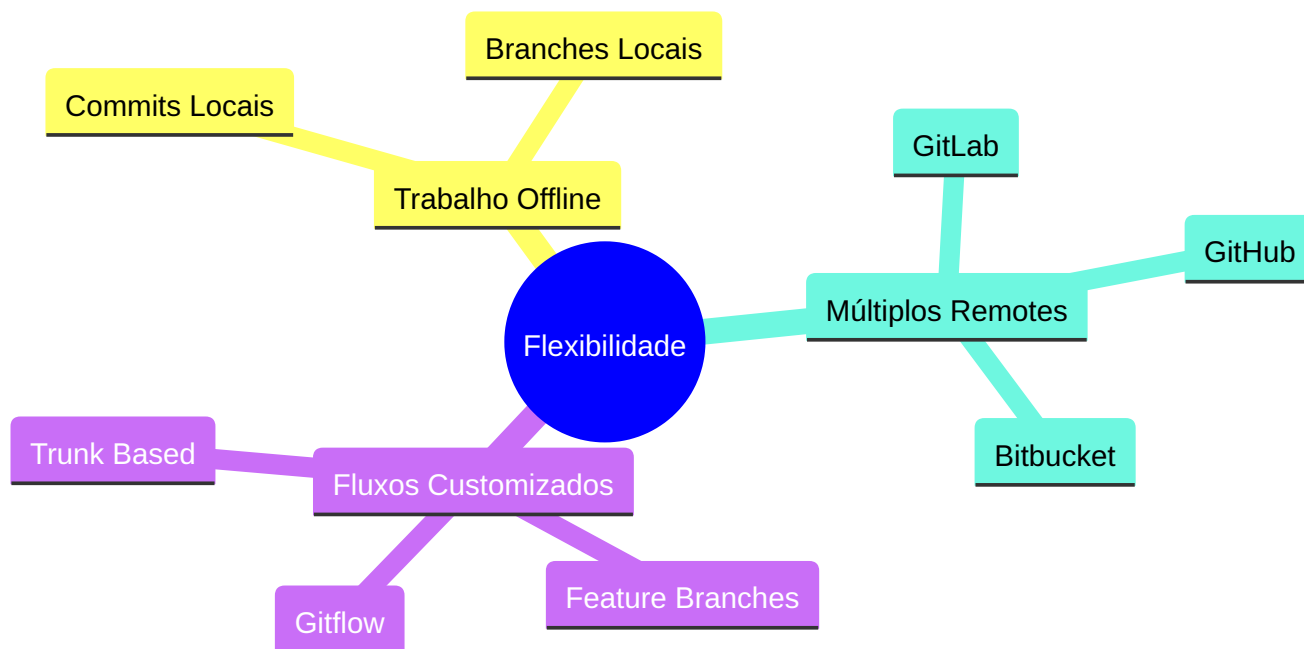
2. **Trabalho Local:** Faz suas alterações na sua cópia - como editar suas fotos antes de postar
3. **Commit:** Salva as alterações localmente - guardando suas edições no rascunho
4. **Push:** Envia para o repositório remoto - finalmente postando na rede social
5. **Pull:** Baixa alterações dos outros - atualizando seu feed

Quando Tem Treta

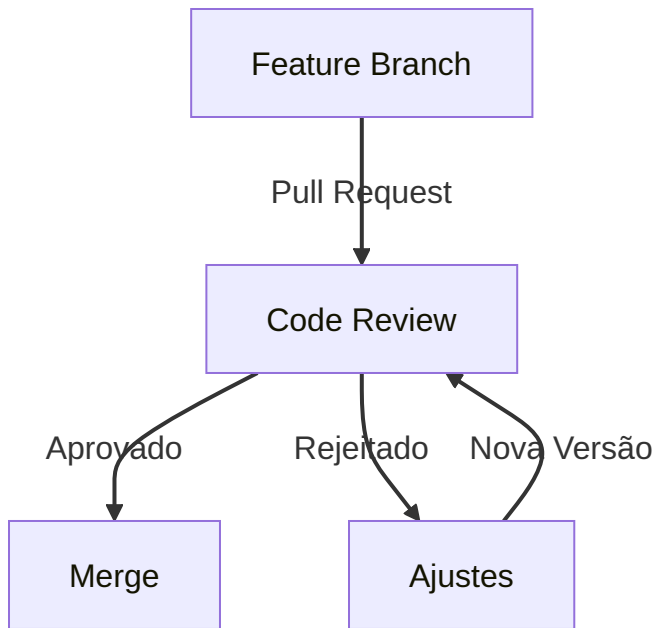
Às vezes duas pessoas mudam a mesma coisa - tipo o Stifler e o Jim editando a mesma foto. Isso gera um conflito, mas não é o fim do mundo:

1. O sistema avisa que tem conflito
2. Você decide qual versão manter (ou combina as duas)
3. Faz um novo commit com a resolução
4. Todo mundo fica feliz!

1. Flexibilidade Máxima



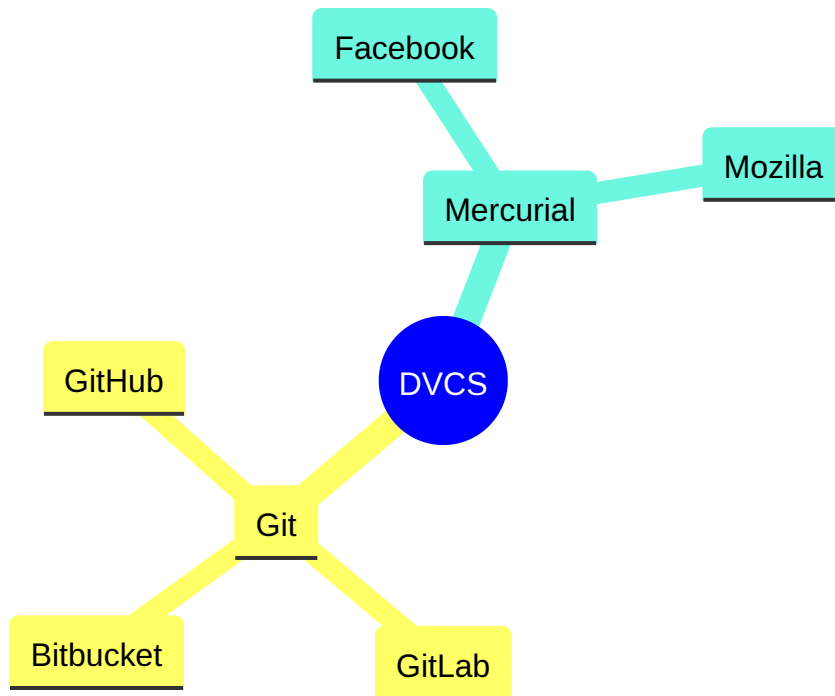
2. Colaboração Avançada



3. Segurança Reforçada

```
+-----+
|  Repo Central  |
+-----+
      |||
      +-----+
      | Clones   |
      +-----+
      | Backups  |
      +-----+
      | História |
      +-----+
```

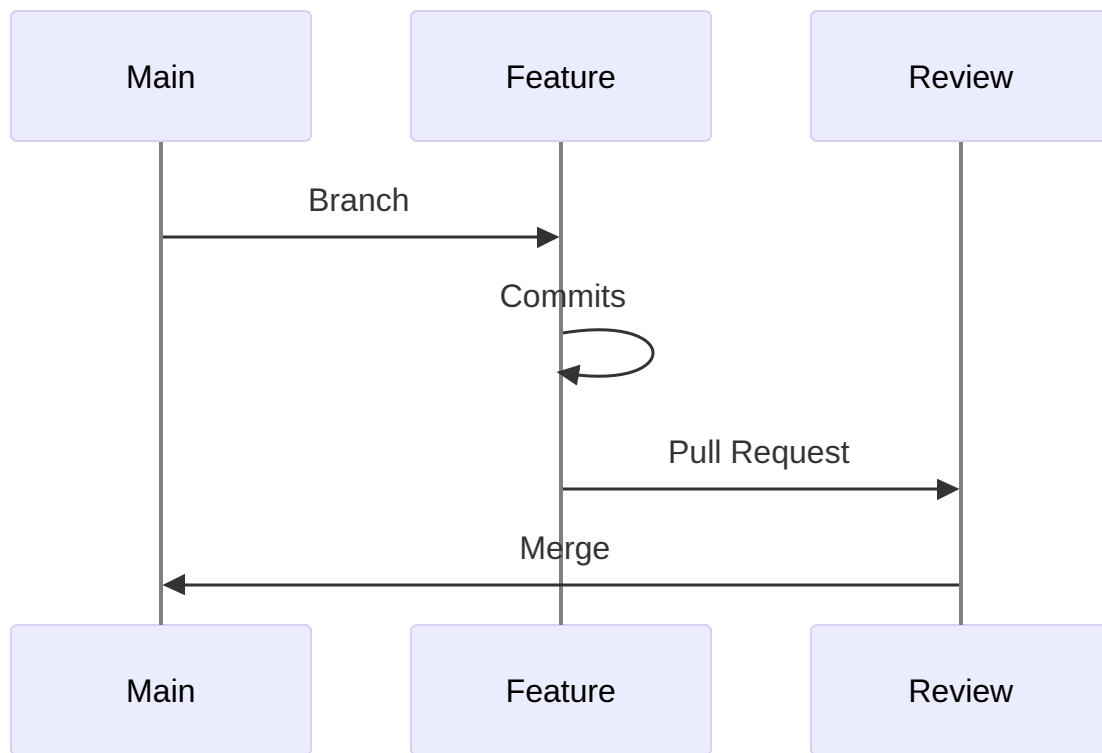
Sistemas Populares



Workflows Populares

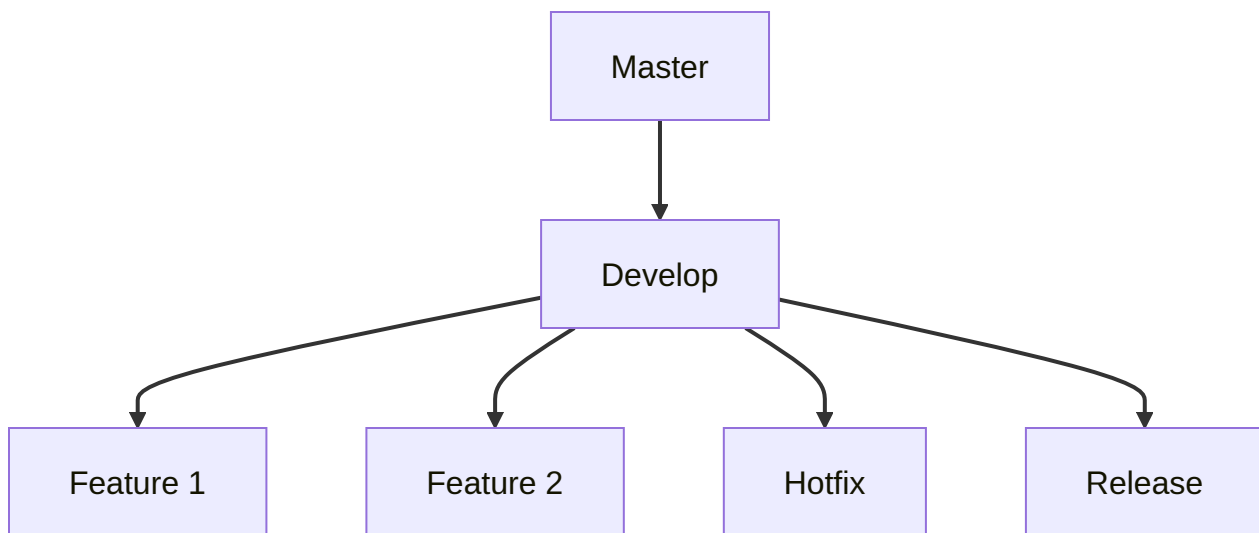
1. Feature Branch

Cada nova funcionalidade ganha sua própria branch. É como se cada nova ideia maluca do Stifler tivesse seu próprio espaço para não bagunçar a festa principal.



2. Gitflow

Um workflow mais estruturado, com branches específicas para desenvolvimento, features, releases e hotfixes. É tipo ter áreas VIP, pista de dança e bar separados na festa.



3. Trunk Based

Desenvolvimento direto na main com branches curtas. É como uma festa mais intimista, onde todo mundo fica no mesmo ambiente.

```

main
|
|— feature/quick
|   └─ merge rápido
|
|— feature/small
|   └─ merge rápido
|
└─ atual

```

Melhores Práticas

1. Commits Atômicos

- Faça commits pequenos e focados
- Escreva mensagens que façam sentido
- Não commita código quebrado
- Imagine que você vai ler isso bêbado depois



2. Branches Organizados

- Crie uma branch pra cada feature nova
- Mantenha a main/master sempre funcionando
- Não tenha medo de experimentar em branches
- Merge só quando tiver certeza

```

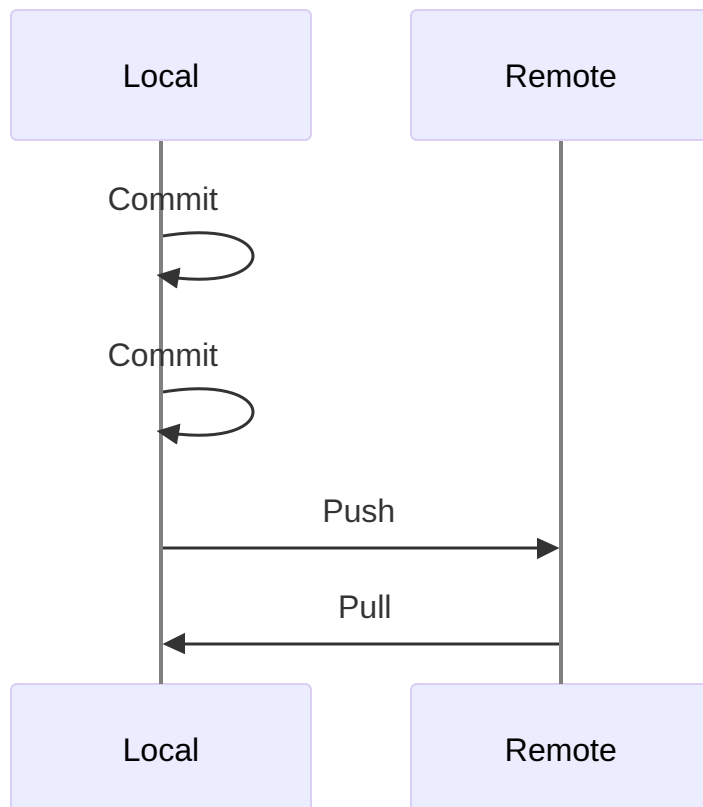
main
|— feature/
|   └─ nova-festa

```

```
|   └─ mais-milfs
|─ hotfix/
|   └─ bug-critico
└─ release/
    └─ v2.0
```

3. Sincronização Regular

- Dê pull antes de começar a trabalhar
- Push quando terminar algo importante
- Mantenha seu código atualizado
- Não deixe commits acumularem



Ferramentas Essenciais

1. Interfaces Gráficas

- GitKraken
- SourceTree
- GitHub Desktop

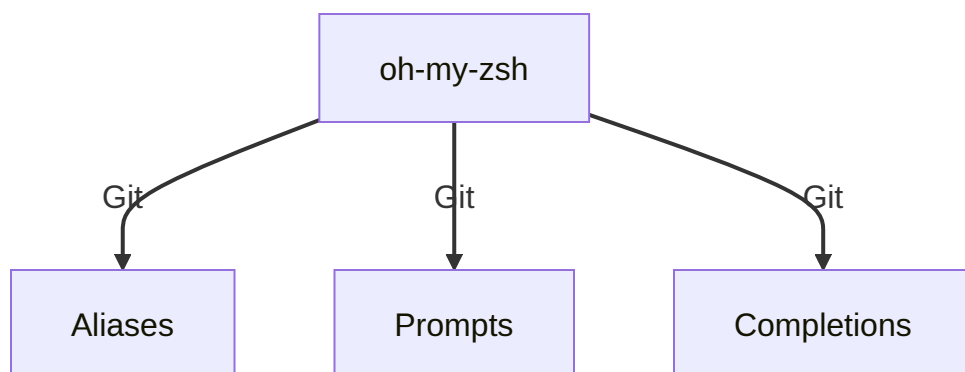
2. Extensões IDE

```
+-----+
| IDE    |
| +-----+ |
| | Git  | |
| | Tools| |
| +-----+ |
+-----+
```

Toda IDE que se preze tem integração com Git. Use e abuse delas!

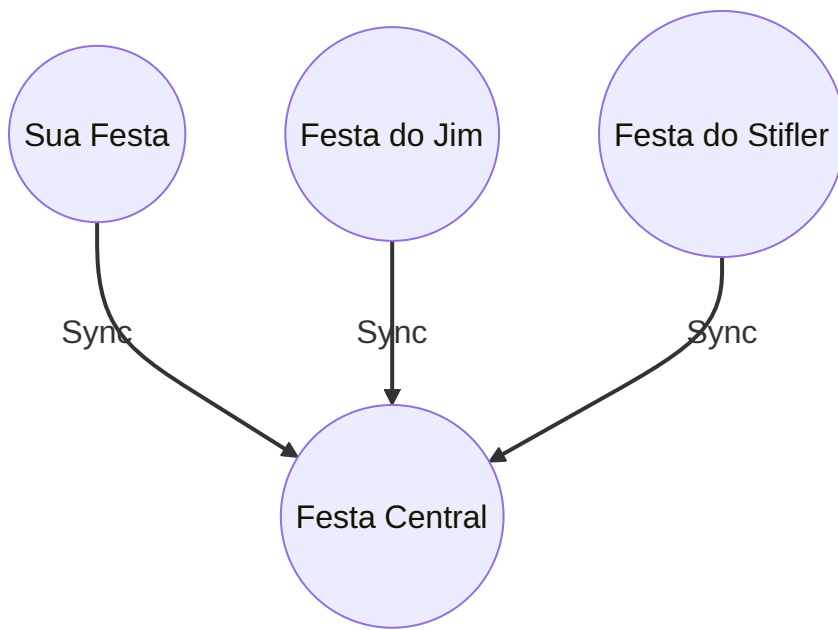
3. CLI Aprimorada

Personalize seu terminal para trabalhar melhor com Git. Aliases e prompts podem salvar seu dia!



Conclusão

DVCS é como ter uma festa particular que pode se conectar com outras festas quando quiser. Cada um tem seu espaço, suas regras, mas todo mundo pode compartilhar quando estiver pronto! É a democracia do código - todo mundo tem poder igual, ninguém depende de um servidor central, e a festa nunca para!



Comparando Sistemas de Controle de Versão

Vamos fazer uma análise profunda dos diferentes sistemas de controle de versão, usando analogias divertidas para entender melhor cada um. É como comparar diferentes tipos de festas - cada uma tem seu propósito e seu público!

Sistemas Locais: A Festa Caseira

```
+-----+
|   Seu PC   |
| +-----+  |
| | Projeto  |  |
| | v1.txt   |  |
| | v2.txt   |  |
| | final.txt|  |
| +-----+  |
+-----+
```

Vantagens

- Rápido como Flash - tudo acontece no seu PC
- Simples de usar - é só copiar e colar
- Funciona offline - não precisa de internet

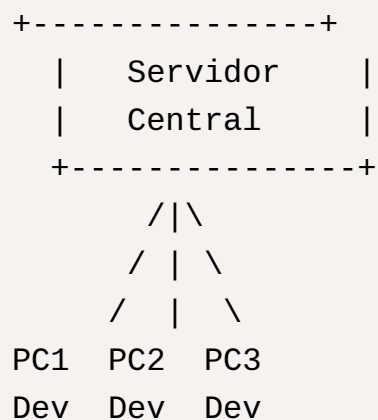
Desvantagens

- Zero colaboração - é festa solo
- Sem backup - se o PC morrer, adeus código
- Organização manual - você precisa gerenciar tudo

Quando Usar

- Projetos pessoais pequenos
- Aprendizado inicial
- Quando você é tipo o Stifler trabalhando sozinho

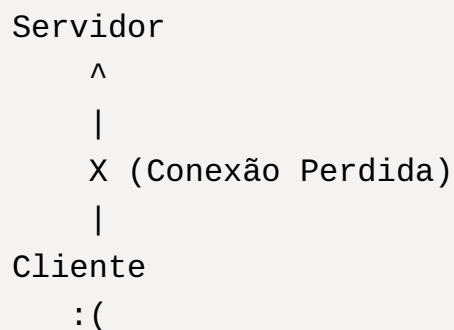
Sistemas Centralizados: A Festa na Casa da Mãe do Stifler



Vantagens

- Controle central - todo mundo sabe onde está o código
- Mais organizado - versões numeradas certinhas
- Permissões claras - você decide quem pode fazer o quê

Desvantagens



- Precisa de internet - sem conexão, sem festa

- Servidor único - se cair, todo mundo chora
- Branches pesados - criar branches é como organizar outra festa

Quando Usar

- Equipes pequenas e médias
- Projetos que precisam de controle rígido
- Quando você quer saber exatamente quem fez o quê

Sistemas Distribuídos: O Festival de Código

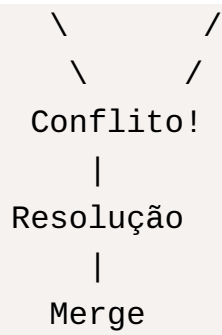
```
+-----+
|  Repositório  |
|   Central    |
+-----+
/           \
+-----+ +-----+ +-----+
| Clone | |Clone| | Clone |
|  1   | |  2   | |  3   |
+-----+ +-----+ +-----+
    Jim      Stifler  Finch
```

Vantagens

- Todo mundo tem uma cópia - a festa está em todo lugar
- Trabalho offline - faça código até no busão
- Branches leves - crie quantas quiser
- Backup natural - cada clone é um backup

Desvantagens

```
Branch      Branch
 \          /
```



- Curva de aprendizado - tem muito comando pra aprender
- Complexidade - às vezes é difícil saber o que está acontecendo
- Conflitos mais frequentes - quando todo mundo mexe em tudo

Quando Usar

- Projetos grandes
- Equipes distribuídas
- Código open source
- Quando você quer a flexibilidade máxima

Tabela Comparativa Completa

Característica	Local	Centralizado	Distribuído
Velocidade	Muito Rápida	Depende da Rede	Rápida
Colaboração	Impossível	Limitada	Ilimitada
Backup	Nenhum	Único	Múltiplos
Complexidade	Simples	Média	Alta
Offline	Sempre	Nunca	Sempre
Aprendizado	Fácil	Médio	Difícil
Conflitos	Nenhum	Comuns	Gerenciáveis

Escolhendo Seu Sistema

```

+-----+
| Sua Escolha |
+-----+
      |
    +---+---+
    |           |
Local   Centralizado
    |           |
    +---+---+
      |
    Distribuído

```

Para Iniciantes

Se você está começando, comece com um sistema local. É como aprender a fazer festa no seu quarto antes de ir pra balada.

Para Times Pequenos

Um sistema centralizado pode ser perfeito. Todo mundo sabe onde é a festa (o servidor) e as regras são claras.

Para Projetos Grandes

Sistema distribuído é o caminho. É como ter várias festas interligadas, cada uma com sua própria dinâmica.

Conclusão

+-----+	+-----+	+-----+
Local	Centralizado	Distribuído
Festa Solo	Festa	Festival
	na Casa	Open
\o/	\o/\o/	\o/\o/\o/
+-----+	+-----+	+-----+

Não existe sistema perfeito - existe o sistema certo para cada situação. É como escolher entre:

- Uma festa íntima em casa (Local)
- Uma festa organizada na casa da mãe do Stifler (Centralizado)
- Um mega festival com várias stages (Distribuído)

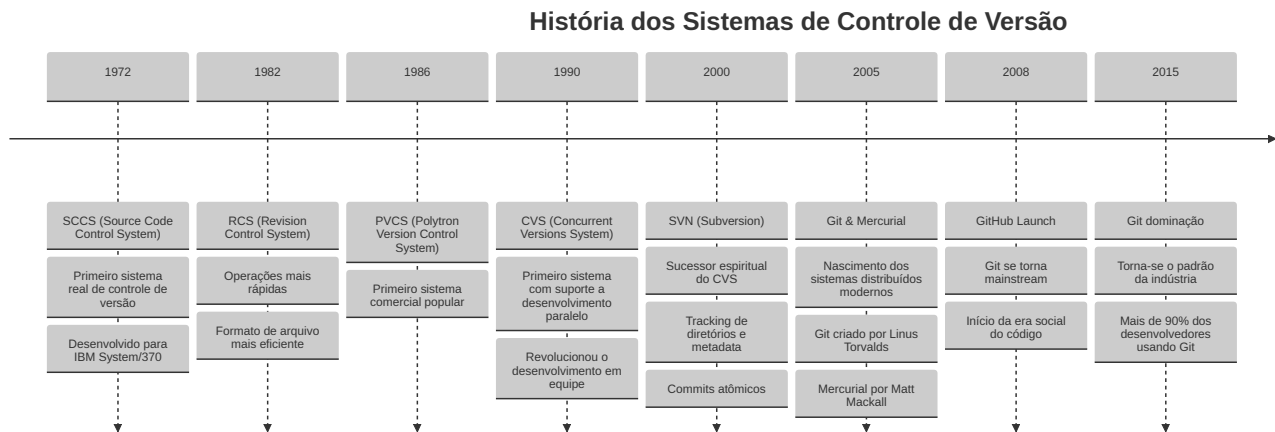
A escolha depende do seu projeto, equipe e necessidades. E lembre-se: o importante é o código (ou a festa) fluir bem!

História do Controle de Versão

A Evolução do Versionamento



Como passamos de backups manuais para sistemas distribuídos modernos



A Linha do Tempo Detalhada

Anos 70-80: A Pré-História do Código

SCCS (1972)

- **Criador:** Marc Rochkind na Bell Labs
- **Inovações:**
 - Primeiro sistema real de controle de versão
 - Introduziu o conceito de deltas reversos
 - Arquivos de histórico com extensão ,v
- **Limitações:**
 - Apenas um arquivo por vez
 - Sem networking

- Unix only

SCCS

```
|-- arquivo,v  
|-- histórico  
`-- locks
```

RCS (1982)

- **Criador:** Walter F. Tichy
- **Melhorias:**
 - Sistema de branching primitivo
 - Melhor performance
 - Formato de arquivo mais eficiente
 - Comandos mais intuitivos
- **Ainda usado para:**
 - Controle de configuração
 - Documentação
 - Projetos simples

Anos 90: A Revolução Centralizada

CVS (1990)

- **Criador:** Dick Grune
- **Revolucionou com:**
 - Desenvolvimento paralelo
 - Operações em rede
 - Repositórios compartilhados

- Tags e branches
- **Problemas famosos:**
 - Commits não atômicos
 - Renomeação de arquivos complicada
 - Bugs de merge

```

CVS Server
  /   |   \
Client Client Client

```

SVN (2000)

- **Criador:** CollabNet
- **Avanços:**
 - Commits verdadeiramente atômicos
 - Melhor handling de binários
 - Renomeação e move de arquivos
 - Metadados versionados
- **Ainda popular em:**
 - Empresas tradicionais
 - Projetos com muitos binários
 - Sistemas legados

Anos 2000+: A Era Distribuída

Git (2005)

- **Criador:** Linus Torvalds
- **Motivação:**

- BitKeeper removeu licença gratuita do kernel Linux
- Necessidade de sistema rápido e distribuído
- **Inovações:**
 - Modelo distribuído
 - Branching super leve
 - Staging area
 - Integridade criptográfica
- **Por que dominou:**
 - Performance excepcional
 - GitHub e social coding
 - Flexibilidade extrema
 - Workflow distribuído

```

Git Flow
main
| \
|  feature
| /
| \
|  hotfix
| /

```

Mercurial (2005)

- **Criador:** Matt Mackall
- **Diferencias:**
 - Interface mais amigável
 - Curva de aprendizado menor

- Extensibilidade via Python
- Usado por:
 - Facebook
 - Mozilla
 - Google (parcialmente)

Anos 2010+: A Era Social

GitHub (2008)

- Transformou Git em plataforma social
- Pull Requests revolucionaram code review
- Actions trouxeram CI/CD integrado
- Copilot iniciou era da IA no código

GitLab (2011)

- Alternativa self-hosted ao GitHub
- CI/CD integrado desde o início
- DevOps como plataforma

Lições da História

O que Aprendemos

1. Evolução Constante

- De single-file para repositórios completos
- De local para distribuído
- De linha de comando para interfaces gráficas

2. Padrões que Permaneceram

- Importância do histórico
- Necessidade de branches
- Valor da colaboração

3. Tendências Futuras

- Integração com IA
- Automação crescente
- Colaboração em tempo real

Conclusão

A história dos sistemas de controle de versão é uma jornada fascinante de evolução tecnológica. De simples backups numerados até sistemas distribuídos com IA, cada era trouxe suas inovações e aprendizados. Como diria a mãe do Stifler: "As festas podem mudar, mas a diversão continua a mesma!"

E lembre-se: conhecer a história nos ajuda a entender melhor as ferramentas que usamos hoje e apreciar como chegamos até aqui. Afinal, se hoje podemos fazer um git push sem pensar duas vezes, é porque muita gente quebrou a cabeça com SCCS e CVS antes!

Controle de Versão Moderno

A Festa Continua!

Tendências Atuais

1. Integração com Cloud

- GitHub/GitLab/Bitbucket
- Como festas online
- Sempre disponível

2. CI/CD Integration

- Automação de testes
- Deploy automático
- Festa sem trabalho manual

3. Ferramentas Gráficas

- GitKraken
- SourceTree
- Interface amigável

O Futuro

1. IA e Machine Learning

- Resolução automática de conflitos
- Sugestões de código
- Como ter um DJ automático

2. Blockchain

- Versionamento descentralizado
- Imutabilidade
- A próxima revolução?

Melhores Práticas Modernas

1. Trunk-Based Development

- Integração contínua
- Deploys frequentes
- Festa sem fim

2. Feature Flags

- Controle de funcionalidades
- Testes em produção
- Como VIP da festa

Fluxos de Trabalho em Versionamento

Modelos de Fluxo de Trabalho

Trunk-Based Development

- Desenvolvimento direto na branch principal
- Integração contínua frequente
- Ideal para equipes pequenas e ágeis

Feature Branch Workflow

- Branch separada para cada feature
- Merge através de pull requests
- Revisão de código facilitada

Gitflow

- Branches específicas para features, releases e hotfixes
- Estrutura mais rigorosa
- Ideal para releases planejadas

Forking Workflow

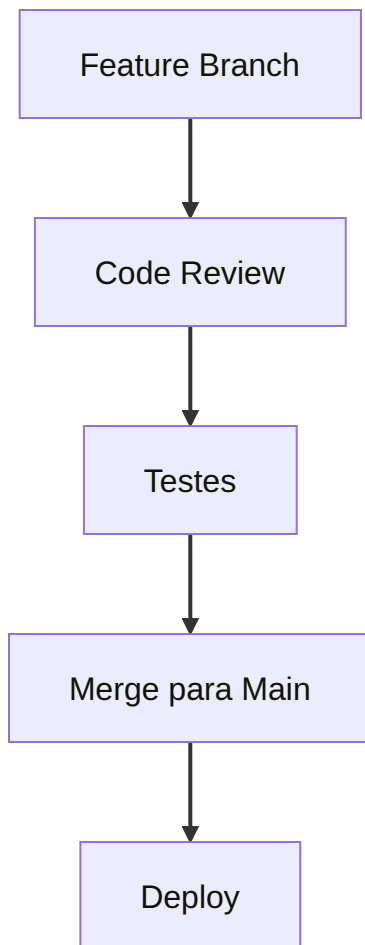
- Fork do repositório principal
- Comum em projetos open source
- Maior isolamento entre contribuições

Escolhendo um Workflow

Fatores a Considerar

- Tamanho da equipe
- Frequência de releases
- Complexidade do projeto
- Necessidades de QA

Exemplos Práticos



Boas Práticas

1. Commits frequentes e pequenos

2. Mensagens de commit claras
3. Code review regular
4. Testes antes do merge
5. Documentação atualizada

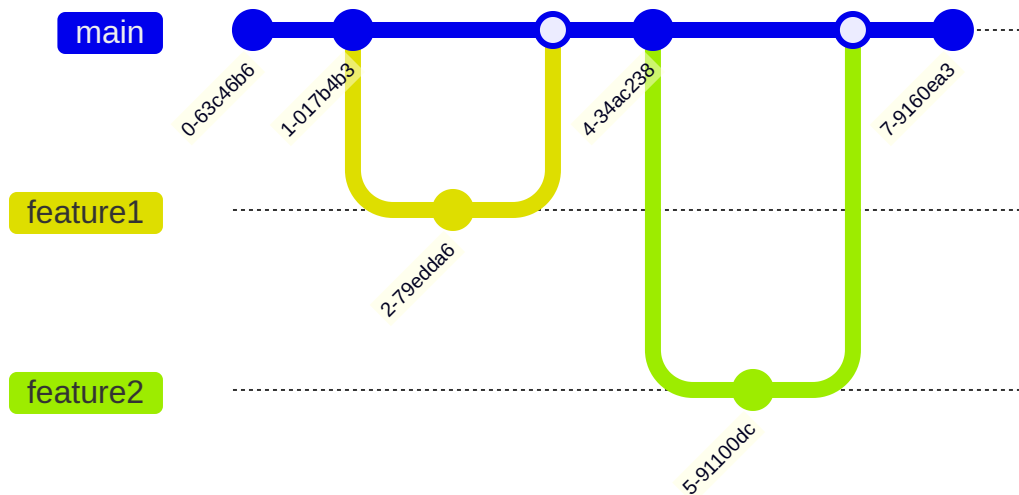
Ferramentas de Suporte

- CI/CD pipelines
- Code review platforms
- Issue trackers
- Automação de testes

Trunk-Based Development

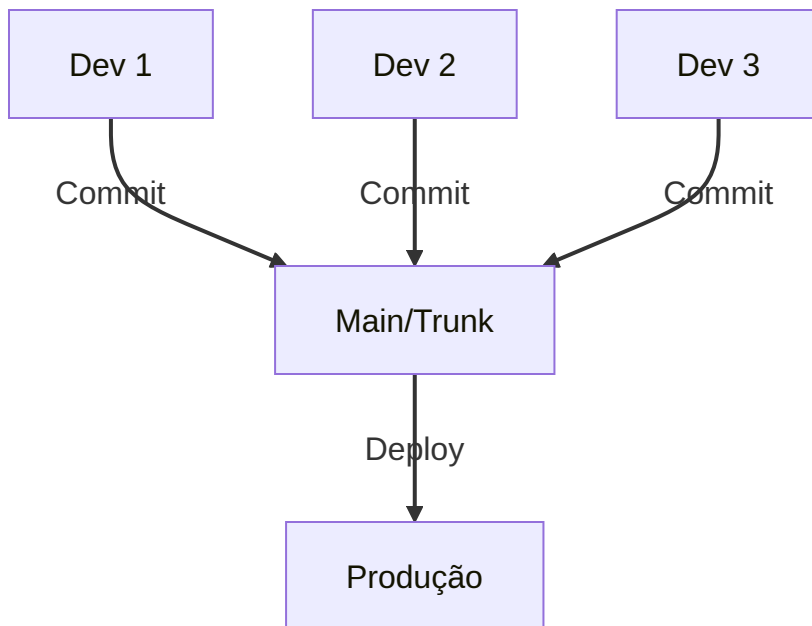
Imagine uma festa onde todo mundo dança na mesma pista. É assim que funciona o Trunk-Based Development (TBD)!

Anatomia do TBD



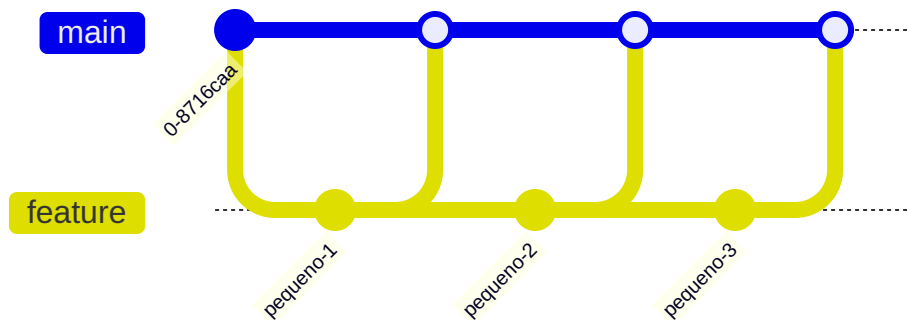
Como Funciona?

Todo mundo trabalha direto na branch principal (trunk/main):

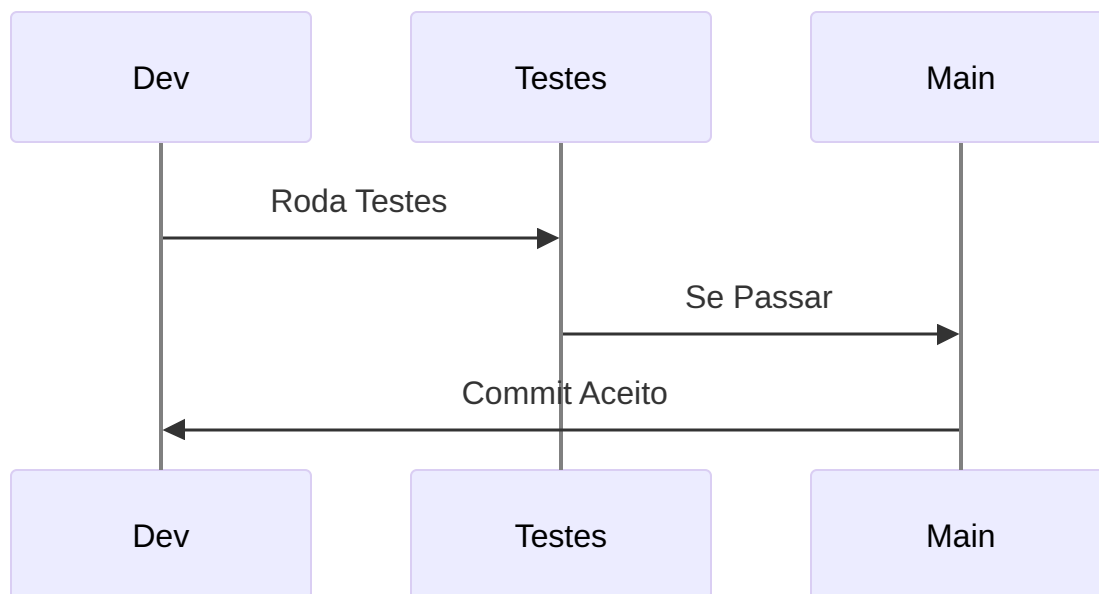


Regras do Jogo

1. Commits Pequenos e Frequentes



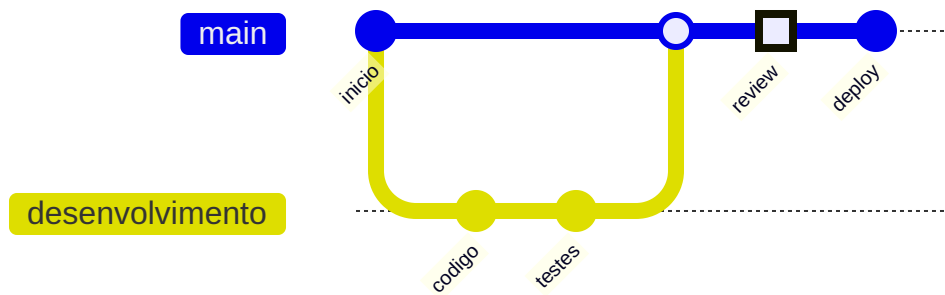
2. Testes Antes de Tudo



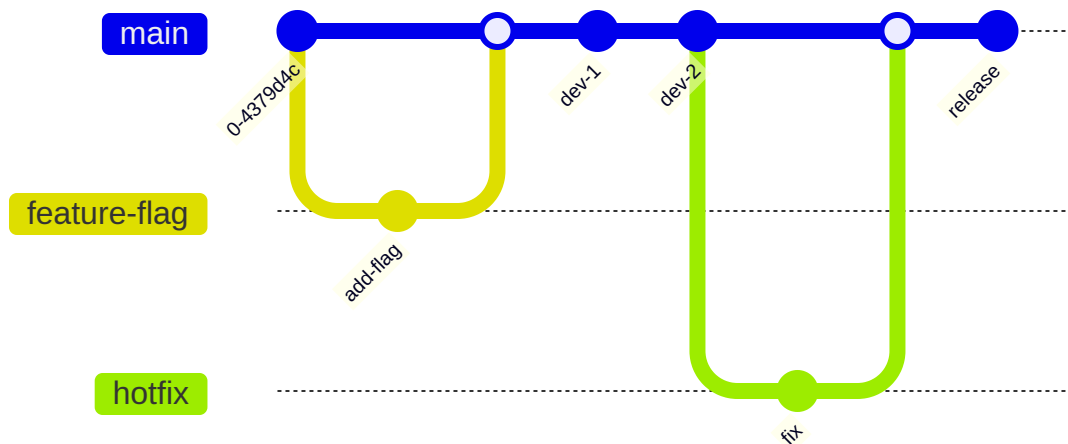
3. Feature Flags

- Código novo entra escondido
- Ativa quando estiver pronto
- Como uma surpresa na festa!

Ciclo de Vida do Código



Fluxo de Trabalho Típico



Por Que Usar?

Vantagens

- Integração contínua real
- Menos conflitos
- Deploy mais rápido
- Todo mundo no mesmo ritmo

Desafios

- Precisa de muita disciplina
- Testes automatizados são obrigatórios
- Feature flags para código incompleto

Na Prática

Fluxo Básico

1. Código novo
2. Testes locais
3. Code review
4. Merge na main
5. Deploy

Dicas de Sobrevivência

- Commits pequenos
- Testes, testes e mais testes
- Feature flags são seus amigos
- Code review rápido

Conclusão

TBD é rápido, moderno e eficiente. Como uma festa bem organizada, todo mundo se diverte junto, mas seguindo algumas regras básicas para manter tudo funcionando!

Feature Branch Workflow

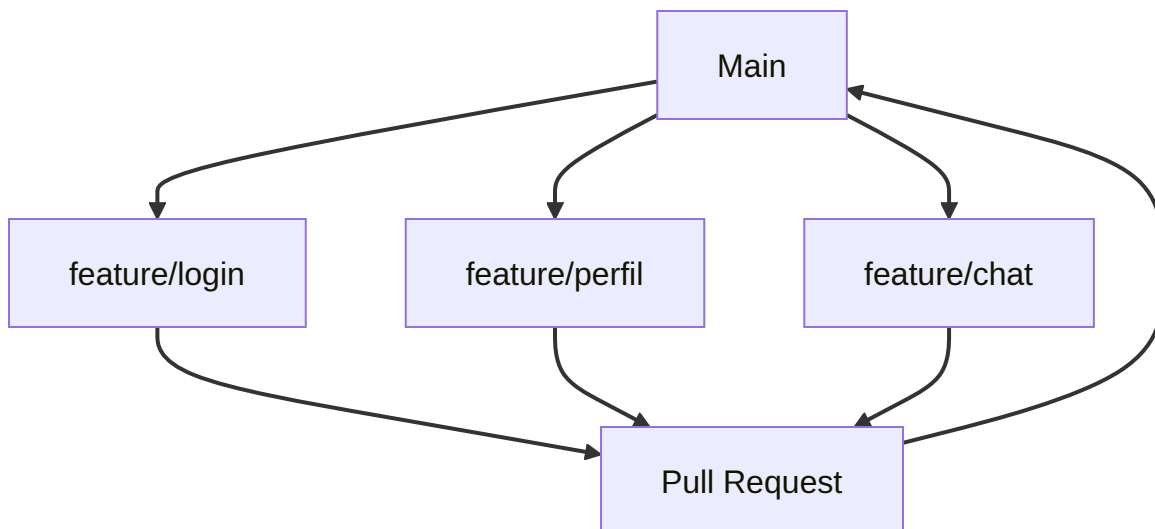
Imagine que cada nova funcionalidade é como uma nova cena do American Pie - precisa ser filmada separadamente antes de entrar no filme final!

Como Funciona?

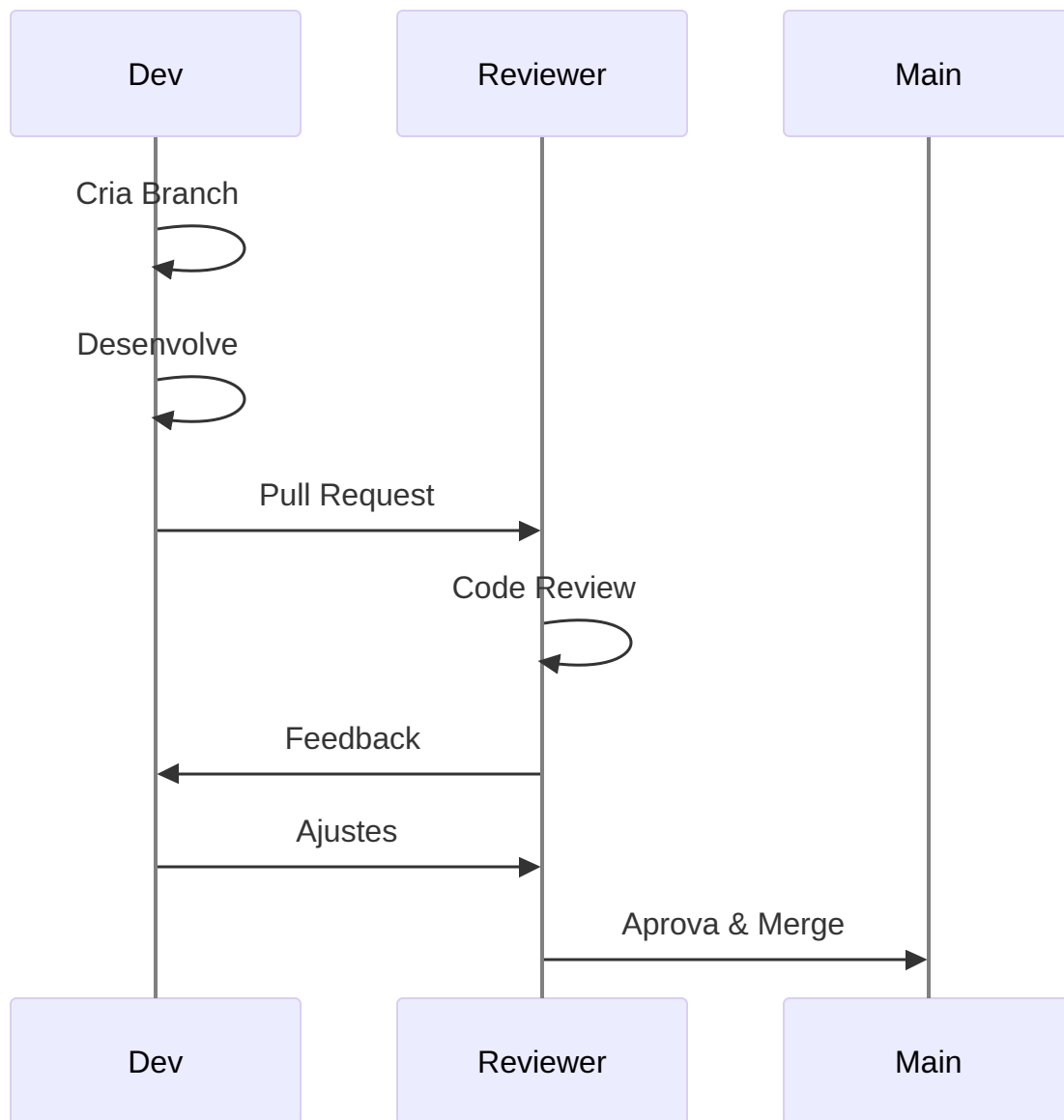


Regras do Jogo

1. Uma Branch por Feature



2. Processo de Review



Anatomia de uma Feature Branch

```
main
|
├─ feature/login
|   ├── commit: "Adiciona form"
|   ├── commit: "Valida campos"
|   └─ commit: "Integra API"
|
├─ feature/perfil
|   └─ commit: "Layout base"
```

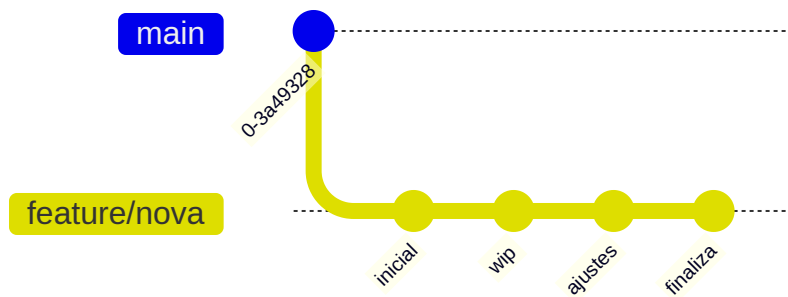
```
|   └─ commit: "Upload foto"
|
|   └─ feature/chat
|       └─ commit: "MVP chat"
```

Fluxo de Trabalho

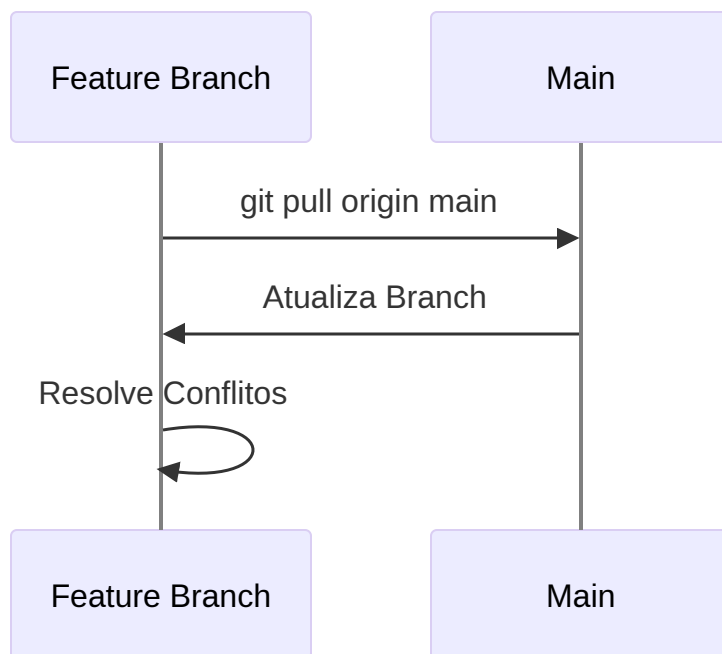
1. Iniciando uma Feature

```
git checkout -b feature/nova-funcionalidade
```

2. Desenvolvimento



3. Mantendo Atualizado

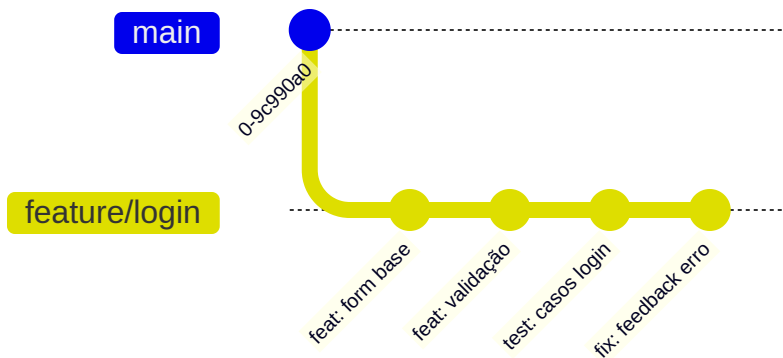


Boas Práticas

1. Nomes de Branches

- ✓ feature/adiciona-login
- ✓ feature/perfil-usuario
- ✓ feature/chat-tempo-real
- ✗ feature/f1
- ✗ nova-coisa
- ✗ mudancas-jim

2. Commits Organizados



Pull Requests

Estrutura Ideal

Pull Request: Adiciona Sistema de Login

✨ O que foi feito:

- Form de login responsivo
- Validação de campos
- Integração com API
- Testes unitários

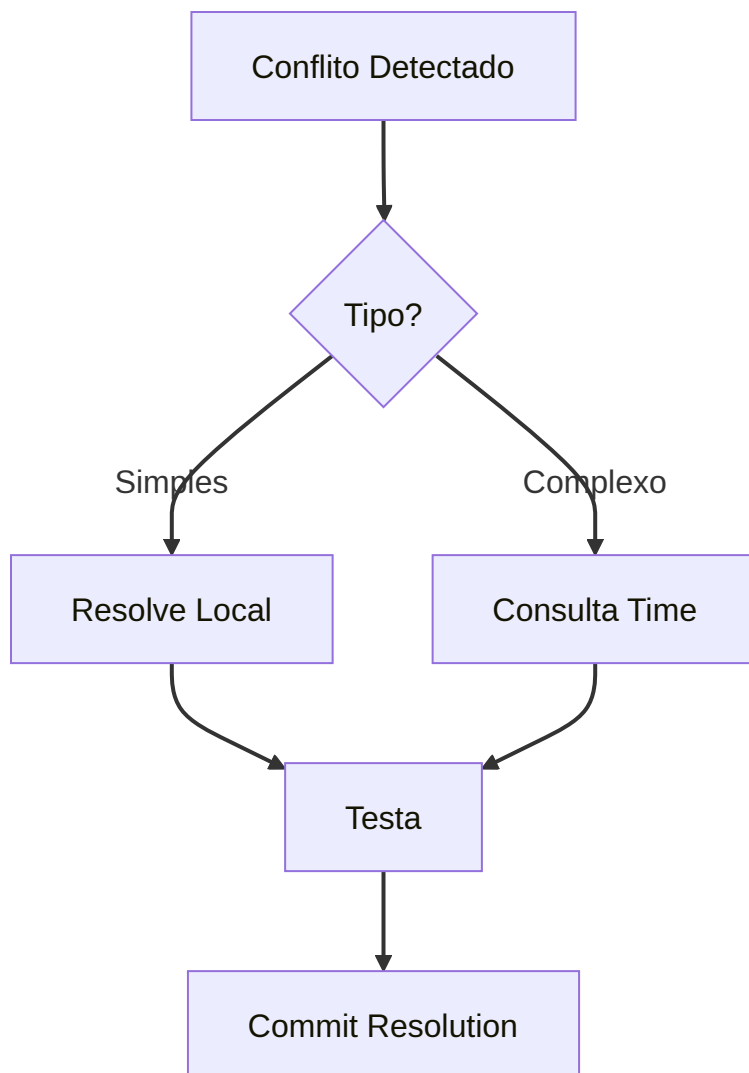
Como testar:

1. Checkout na branch
2. npm install

3. `npm run test`
4. Teste manual do form

📸 Screenshots:
[imagens do antes/depois]

Resolução de Conflitos



Dicas de Sobrevivência

1. Mantenha as Features Pequenas

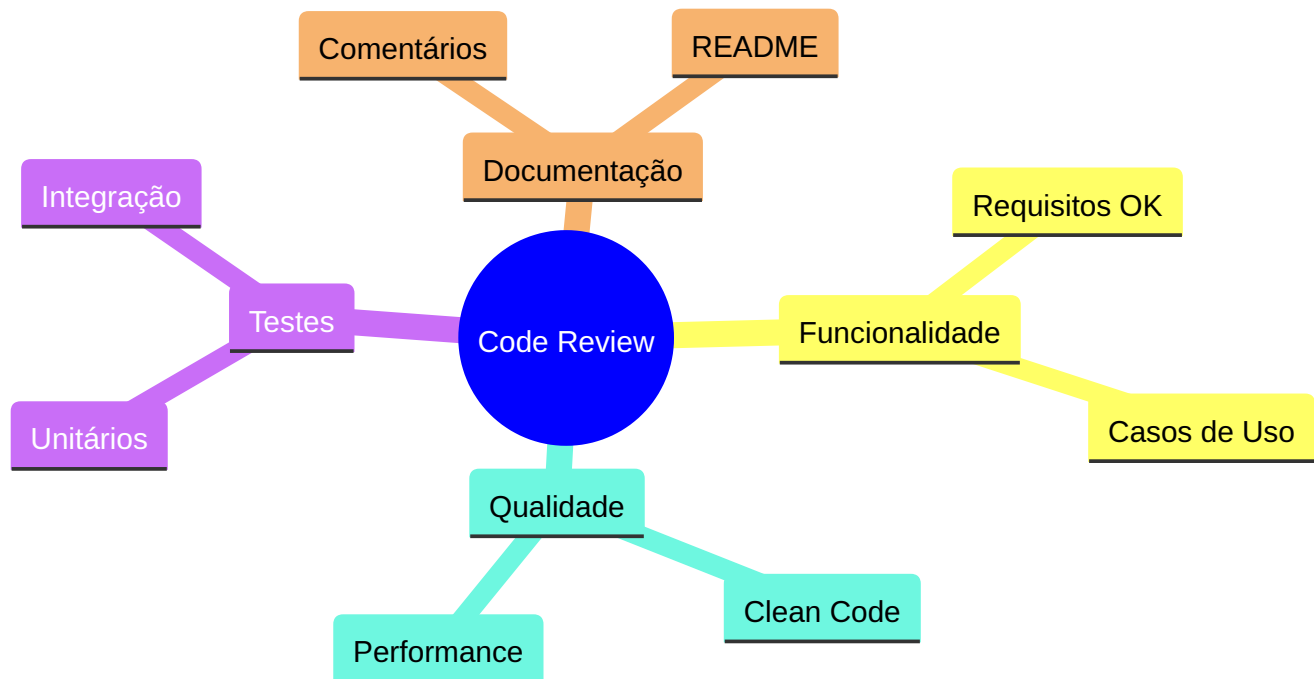
Grande Feature ❌

- | 2 semanas
- | 50 arquivos
- | Difícil review

Features Menores ✅

- | 2-3 dias
- | 5-10 arquivos
- | Review tranquilo

2. Review Checklist



Métricas de Sucesso

 Indicadores Saudáveis

Tempo de Branch
2-3 dias



Tamanho do PR
200-400 linhas

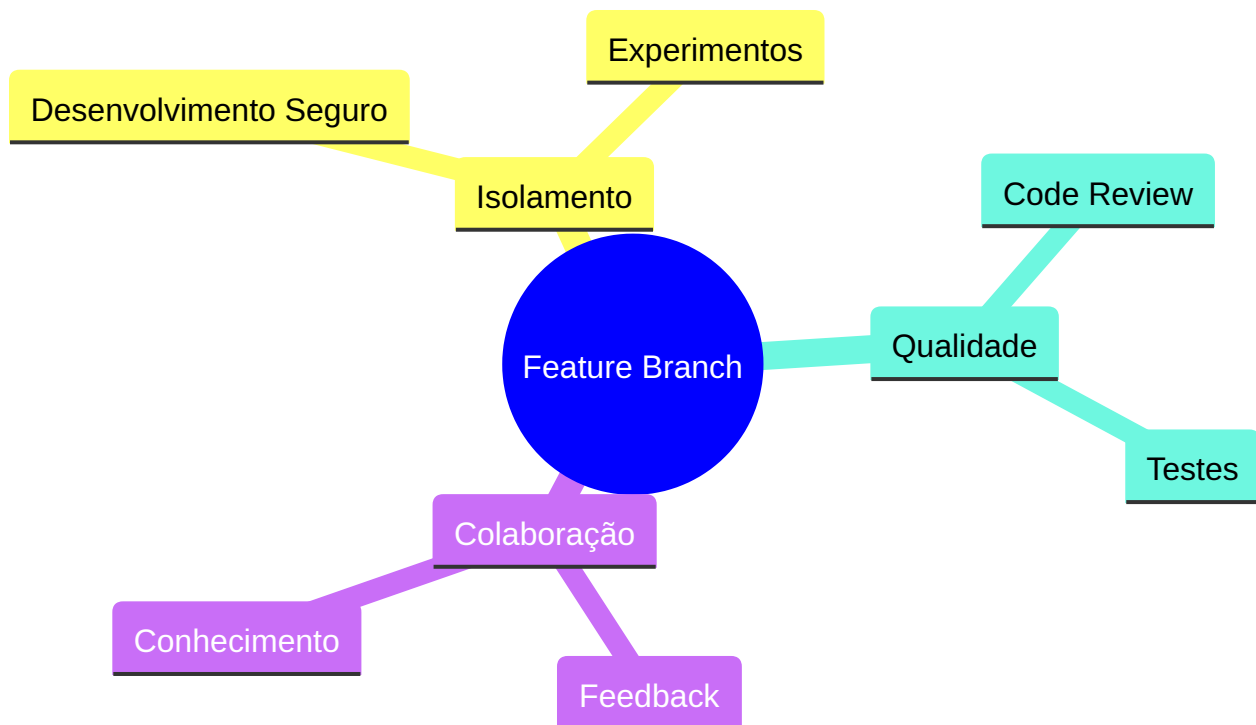


Taxa de Aprovação
Primeira review



Conclusão

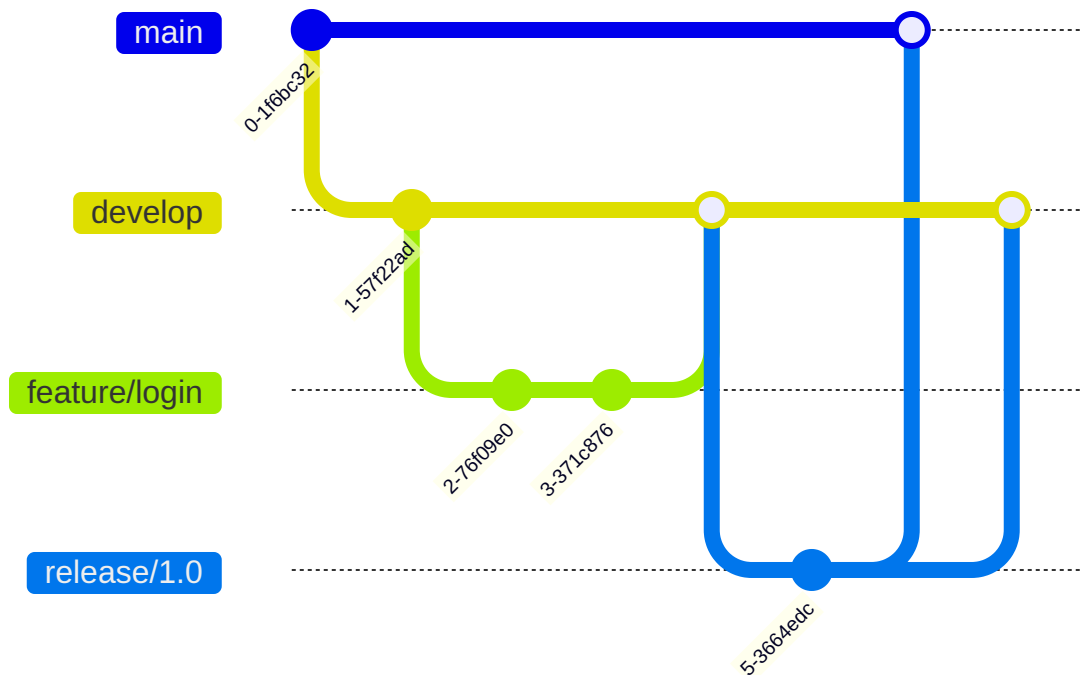
Feature Branch Workflow é como dirigir na sua própria pista: você tem liberdade para desenvolver no seu ritmo, mas sempre seguindo as regras de trânsito para chegar seguro ao destino!



Gitflow Workflow

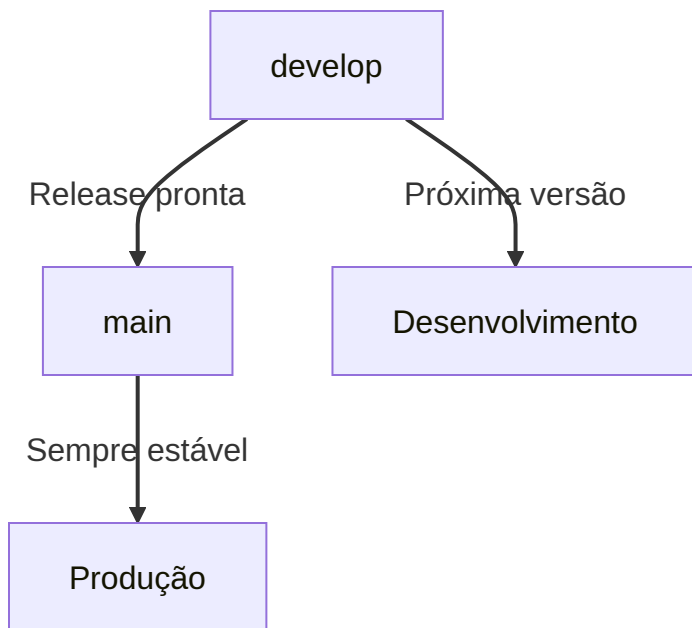
Se o Feature Branch é uma festa na casa do Stifler, o Gitflow é o baile de formatura – tem regras, tem estrutura, mas ainda é divertido!

Estrutura Principal

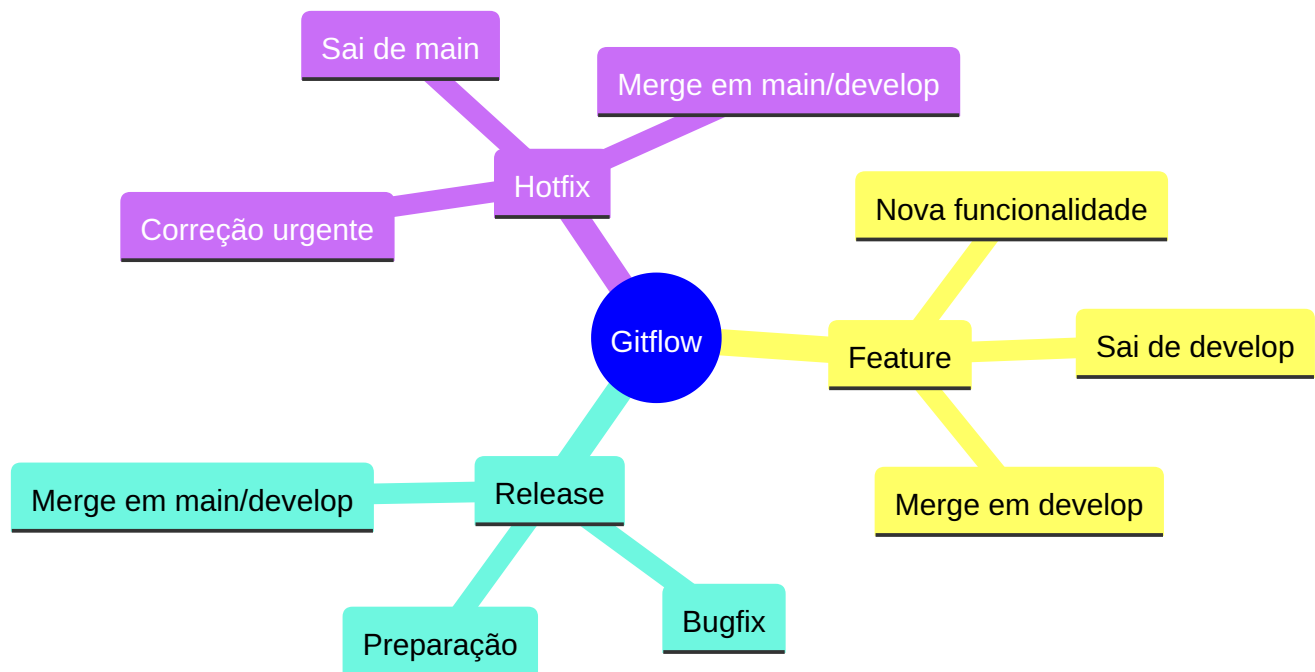


Branches Principais

1. Main e Develop

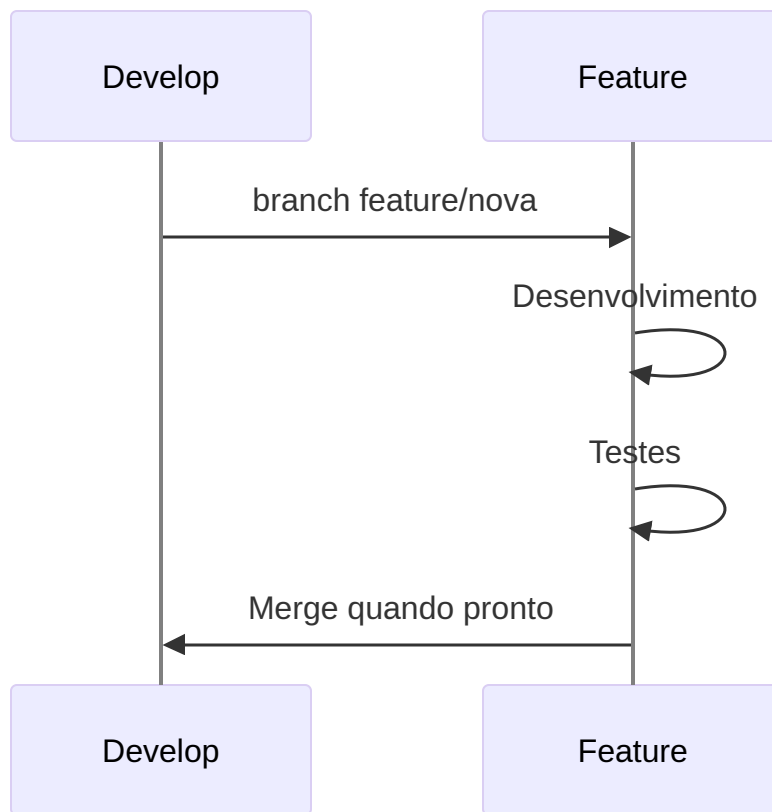


2. Branches de Suporte

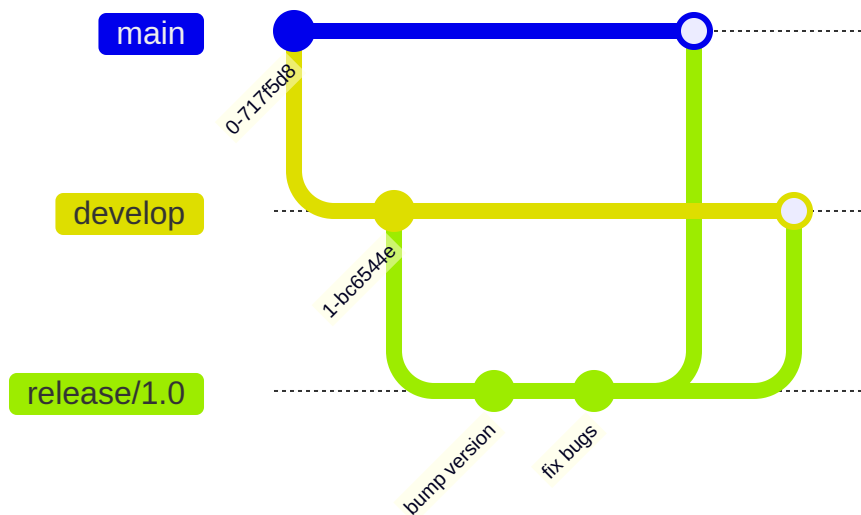


Ciclo de Vida

1. Feature Development



2. Preparação de Release



3. Hotfix em Produção

Comandos Essenciais

1. Iniciando Gitflow

```
git flow init
```

2. Features

```
# Iniciar feature  
git flow feature start login  
  
# Finalizar feature  
git flow feature finish login
```

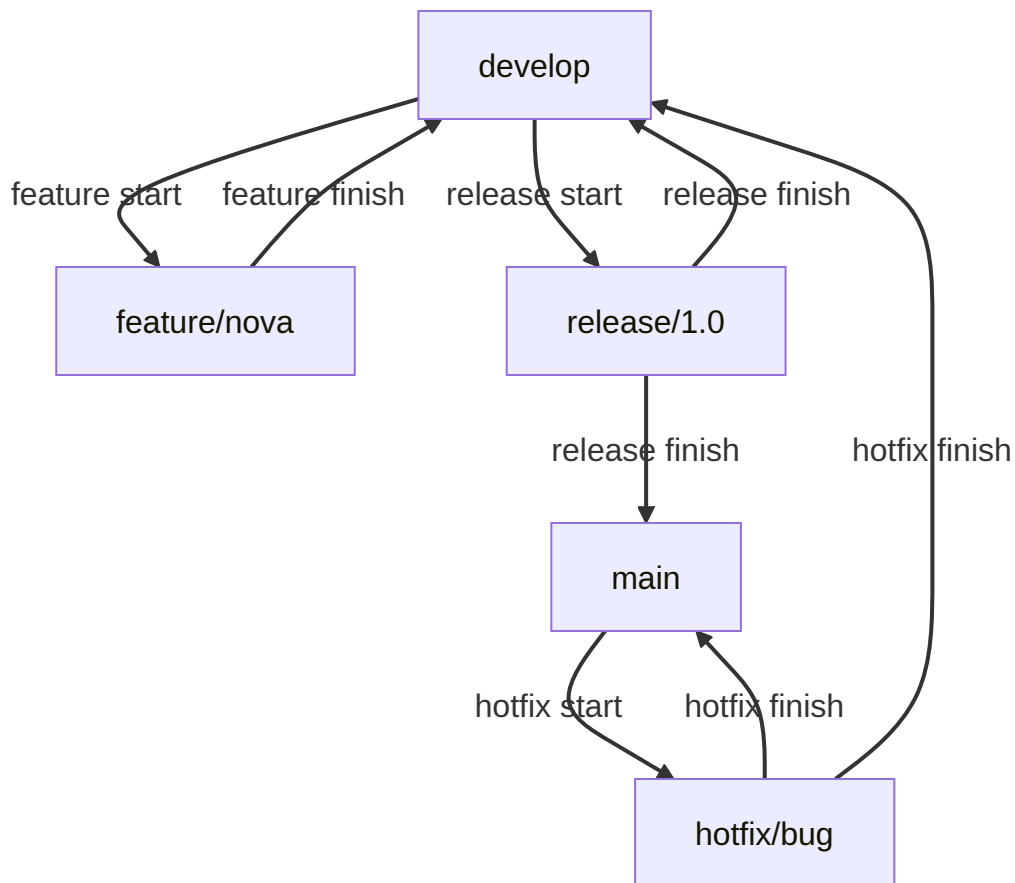
3. Releases

```
# Criar release  
git flow release start 1.0.0  
  
# Finalizar release  
git flow release finish 1.0.0
```

4. Hotfixes

```
# Criar hotfix  
git flow hotfix start bug-critical  
  
# Finalizar hotfix  
git flow hotfix finish bug-critical
```

Fluxo de Trabalho Completo



Boas Práticas

1. Nomenclatura

Features:

feature/login

feature/user-profile

Releases:

release/1.0.0

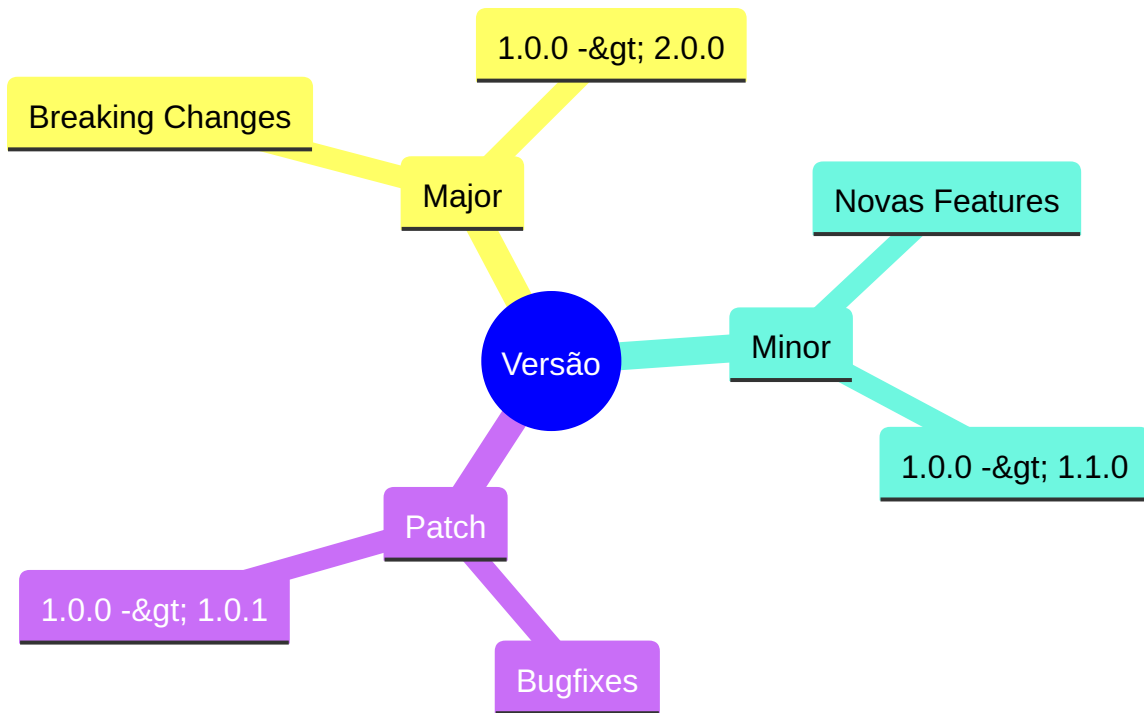
release/2.1.0

Hotfixes:

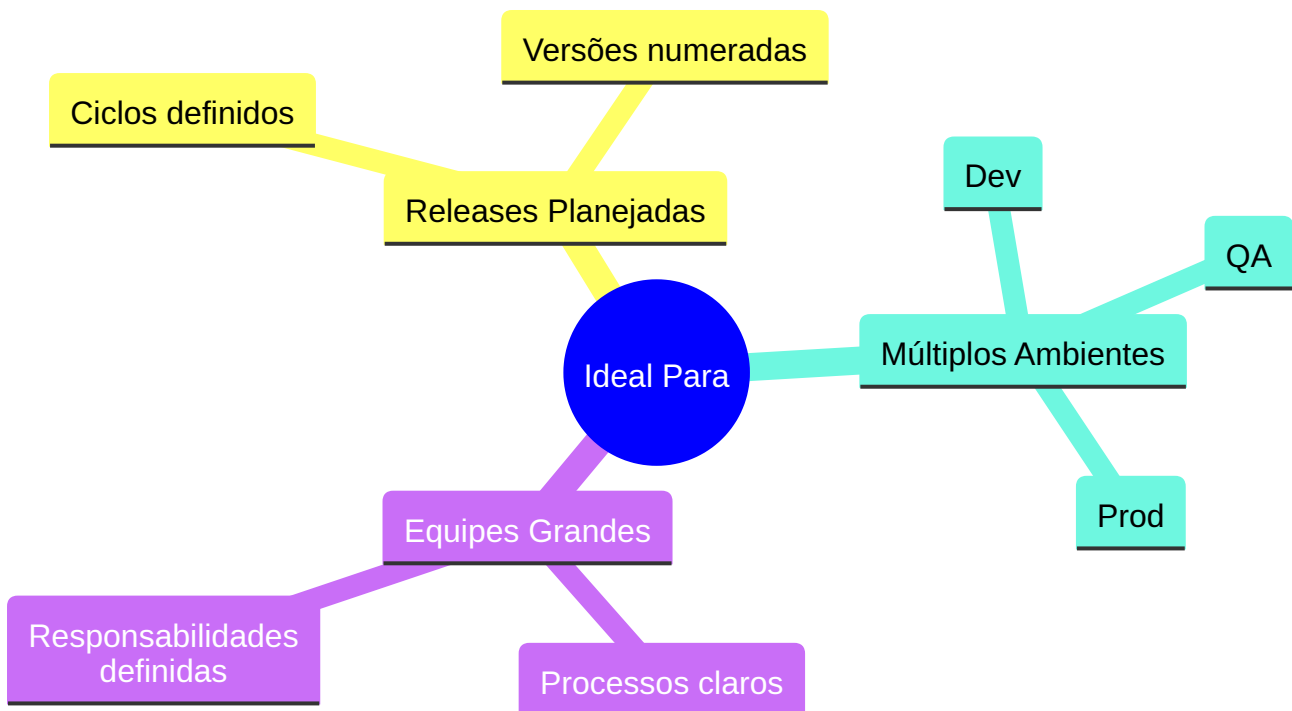
hotfix/security-fix

hotfix/crash-bug

2. Versionamento



Quando Usar Gitflow?



Prós e Contras

Vantagens

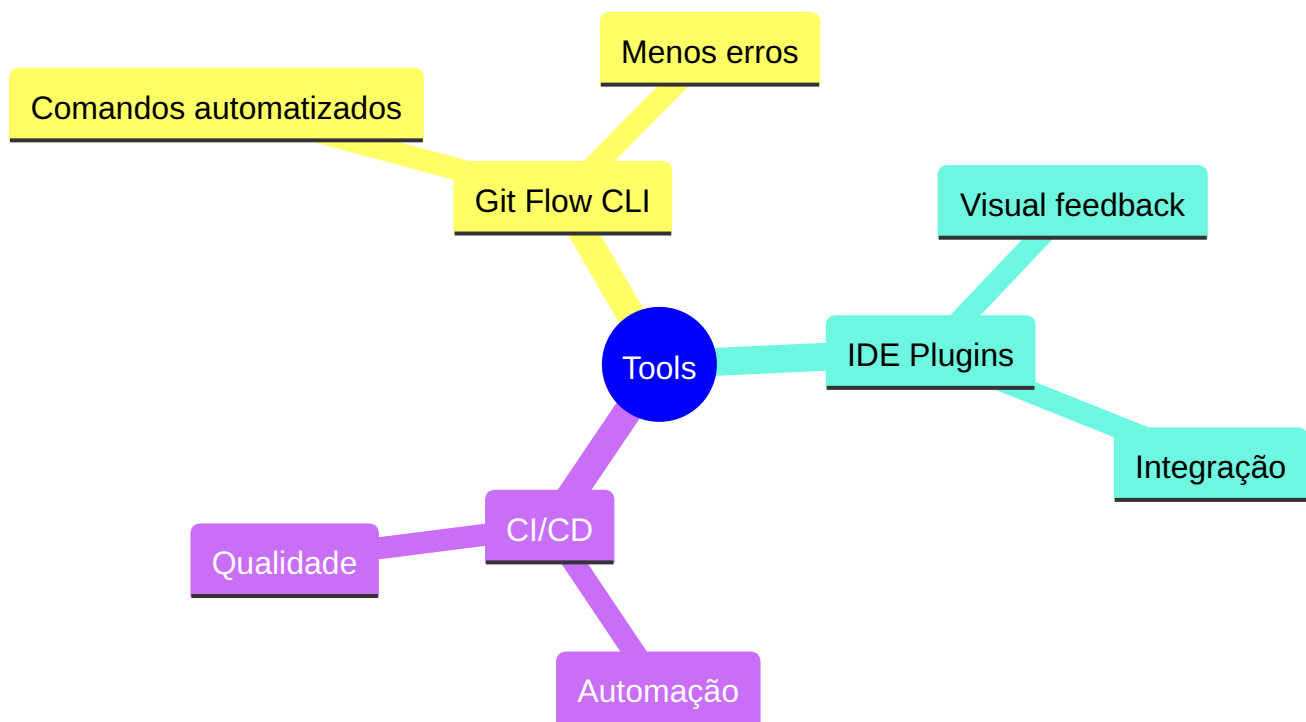
- ✓ Estrutura clara e definida
- ✓ Ideal para releases planejadas
- ✓ Suporte a hotfixes
- ✓ Processos bem documentados

Desvantagens

- ✗ Mais complexo que feature branch
- ✗ Overhead para projetos pequenos
- ✗ Curva de aprendizado maior
- ✗ Pode ser "pesado" demais

Dicas de Implementação

1. Ferramentas de Suporte



2. Checklist de Release

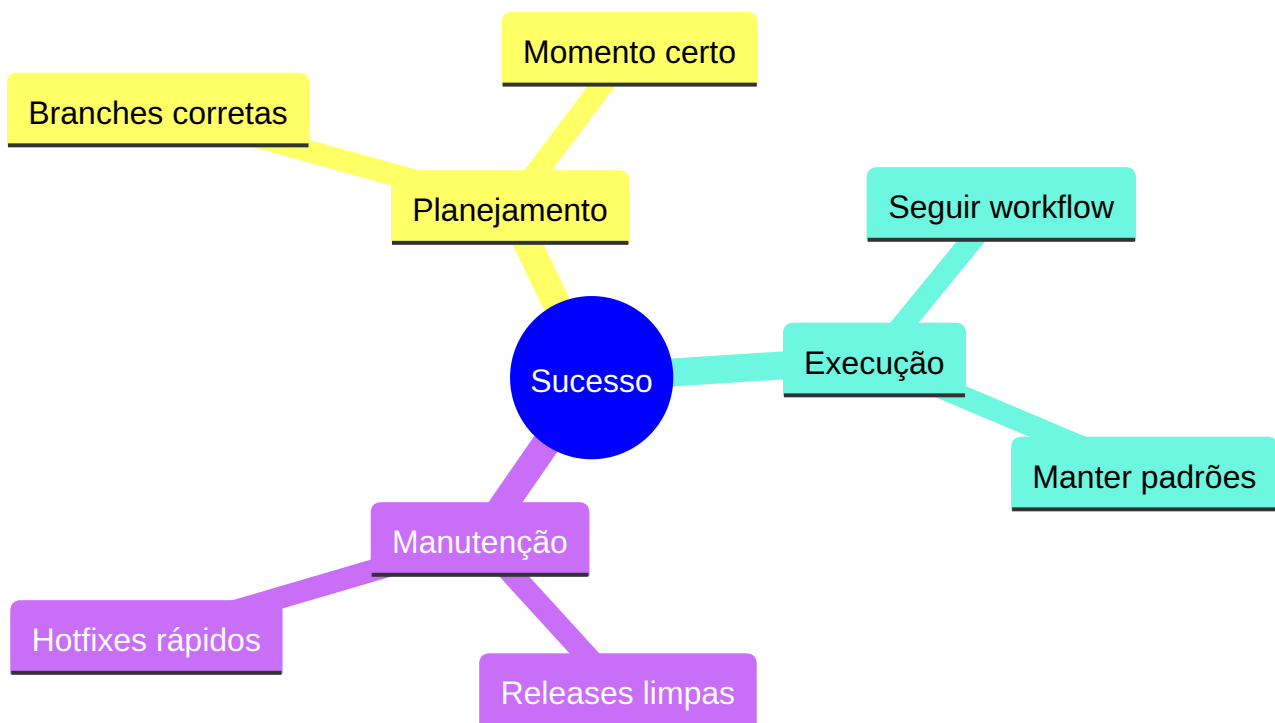
Release Checklist

1. ☐ Feature freeze

2. [] Criar branch release
3. [] Bump version
4. [] Testes de regressão
5. [] Documentação
6. [] Code freeze
7. [] Deploy staging
8. [] Merge em main
9. [] Tag version
10. [] Deploy prod

Conclusão

Gitflow é como um roteiro de filme bem planejado - tem pré-produção (develop), filmagem (features), edição (release) e até correções de última hora (hotfix). Quando bem executado, o resultado é um blockbuster!



Melhores Práticas em Controle de Versão

Organização de Repositório

Estrutura de Diretórios

```
projeto/  
├─ src/  
├─ tests/  
├─ docs/  
├─ .gitignore  
└─ README.md
```

Arquivos Essenciais

- README.md
- .gitignore
- CONTRIBUTING.md
- LICENSE

Commits

Anatomia de um Bom Commit

- Título claro e conciso
- Descrição detalhada quando necessário
- Referência a issues/tickets

Convenções de Commit

```
feat: adiciona novo recurso
fix: corrige bug
docs: atualiza documentação
style: formatação de código
refactor: refatoração de código
test: adiciona/modifica testes
```

Branches

Nomenclatura

- feature/nome-da-feature
- bugfix/descricao-do-bug
- hotfix/correcao-urgente
- release/versao

Estratégias de Merge

- Merge commit
- Squash and merge
- Rebase and merge

Code Review

Checklist

- [] Código segue padrões
- [] Testes adicionados/atualizados
- [] Documentação atualizada
- [] Performance considerada

- [] Segurança verificada

Feedback Construtivo

- Foco no código, não no desenvolvedor
- Sugestões específicas
- Explicações claras
- Reconhecimento de boas práticas

Terminologia do Controle de Versão

Conceitos Básicos

Repository (Repositório)



- Local onde o código é armazenado
- Contém todo o histórico do projeto
- Pode ser local ou remoto

Branch (Ramo)

- Linha independente de desenvolvimento
- Permite trabalho paralelo
- Isola mudanças em desenvolvimento

Commit (Confirmação)

- Snapshot do código em um momento
- Inclui mensagem descritiva
- Possui identificador único (hash)

Operações Comuns

Merge (Mesclagem)

- Combina mudanças de diferentes branches

- Pode gerar conflitos
- Mantém histórico de ambas as branches

Rebase (Rebase)

- Reaplica commits sobre outra base
- Mantém histórico linear
- Útil para manter branches atualizadas

Cherry-pick

- Aplica commits específicos
- Seletivo e preciso
- Útil para hotfixes

Estados de Arquivos

Tracked (Rastreado)

- Modified (Modificado)
- Staged (Preparado)
- Committed (Confirmado)

Untracked (Não Rastreado)

- Arquivos novos
- Não incluídos no controle de versão
- Precisam ser adicionados explicitamente

Glossário Expandido

Termo	Definição
Clone	Cópia completa do repositório
Fork	Cópia independente do repositório
Pull Request	Solicitação para integrar mudanças
Tag	Marco específico no histórico
Hook	Script automatizado em eventos

Segurança em Controle de Versão

Boas Práticas de Segurança

Credenciais e Dados Sensíveis

- Nunca commitar senhas
- Usar variáveis de ambiente
- Implementar .gitignore adequado

Exemplo de .gitignore

```
# Arquivos de configuração
.env
config.json
secrets.yaml

# Diretórios sensíveis
private/
credentials/

# Logs e temporários
*.log
tmp/
```

Controle de Acesso

Níveis de Permissão

1. Read (Leitura)
2. Write (Escrita)
3. Admin (Administração)

Autenticação

- Chaves SSH
- Tokens de acesso
- Autenticação de dois fatores

Vulnerabilidades Comuns

Exposição de Dados

- Commits com dados sensíveis
- Histórico exposto
- Metadados reveladores

Mitigação

1. Git-secrets
2. Pre-commit hooks
3. Análise de segurança automatizada

Auditoria

Logs e Monitoramento

- Registro de acessos
- Histórico de alterações
- Alertas de segurança

Ferramentas de Análise

- Git forensics

- Security scanners
- Dependency checkers

Recuperação

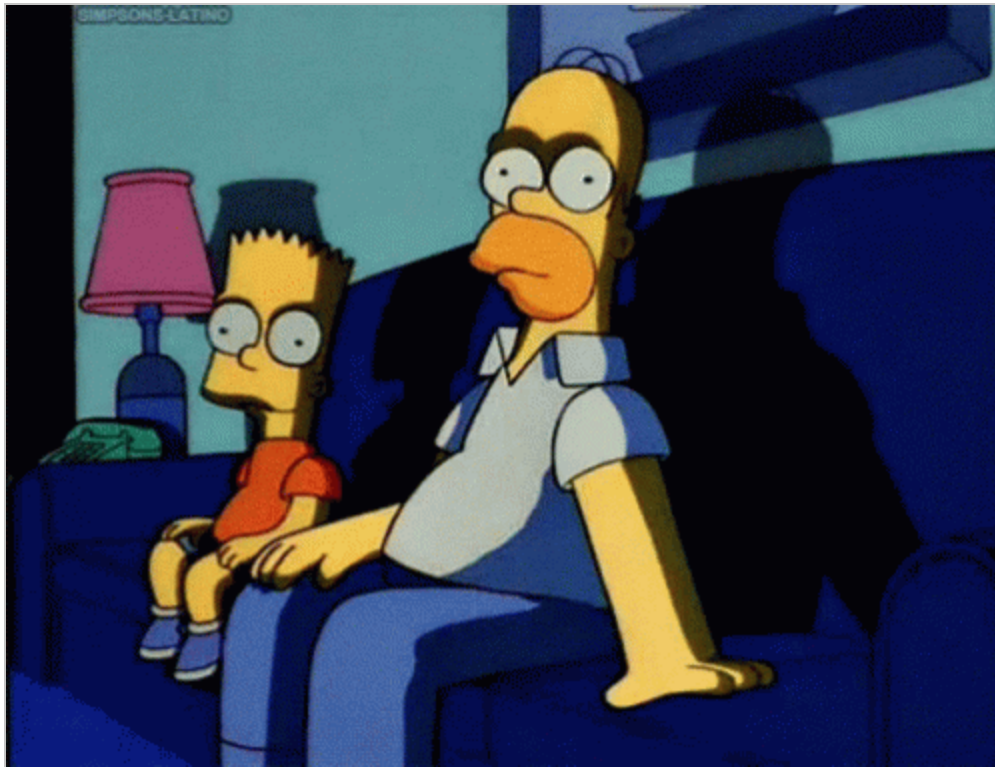
Backup e Restauração

- Estratégias de backup
- Procedimentos de recuperação
- Testes regulares

Incidentes de Segurança

1. Identificação
2. Contenção
3. Remediação
4. Documentação

História do Git



The simpsons homer

Para começar a história do Git é até bem curta e direta. A comunidade do Linux usava um VCS distribuído chamado **BitKeeper** só que ele é proprietário.

Sim, um sistema open source usando um proprietário. Claramente isso era algo que causava um estranhamento na comunidade.



Stifler kiss

Que por sua vez chegou ao ápice quando o BitKeeper se tornou pago, logo a comunidade do Linux ficou alerta já que eles teriam que fazer o versionamento do núcleo do Linux em outro sistema.

Assim então a comunidade começou a criar seu próprio VCS que fosse:

- Simples
- Veloz
- Não linear, ou seja, que aceite vários ramos (***branches***) de modificação
- Capaz de lidar com grandes projetos, afinal, Linux é gigante

E assim nasceu o Git, exatamente em 2005 e até hoje está em evolução sendo um dos VCS mais utilizados em todo o mundo de desenvolvimento de gambiarras (softwares).



Ou seja, tudo nasceu de uma revolta popular



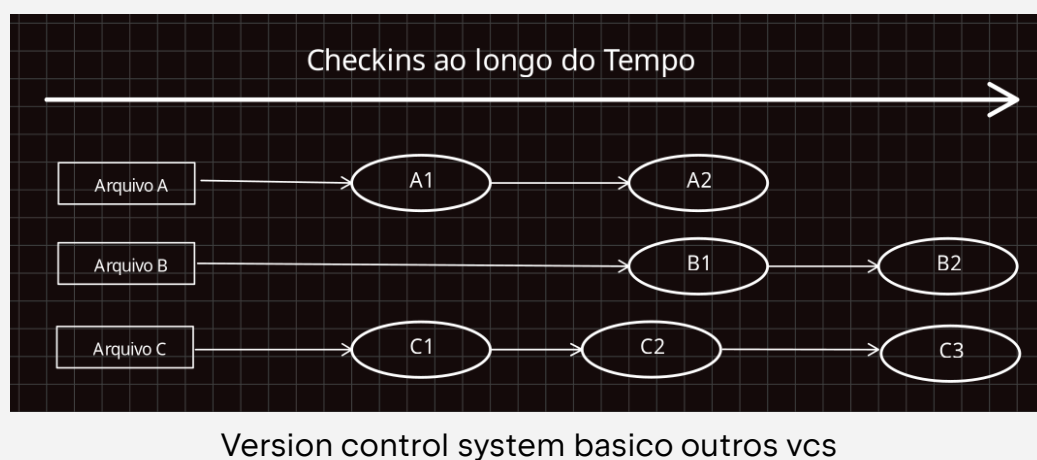
Cachorro comuna

Conceitos Básicos do Git

Como o Git Funciona

O Git funciona de forma diferente de outros VCS. Em um outro VCS ele terá os arquivos e quando houver alteração eles criam uma lista somente das alterações.

Em um outro VCS ele terá os arquivos e quando houver alteração eles criam uma lista somente das alterações:



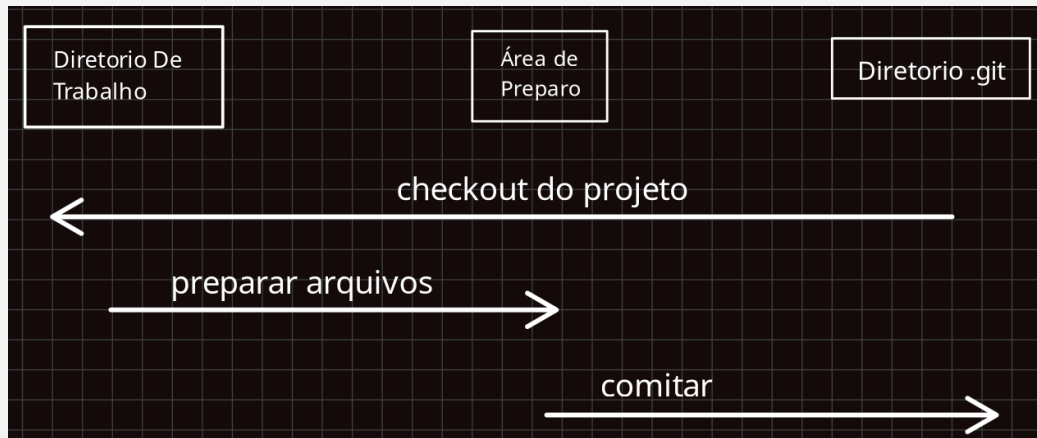
Agora com o Git ele faz diferente, já que vai tirando *snapshots* que são como fotos quando ocorre uma mudança e caso tenha algum arquivo que não foi alterado será guardado uma referencia para ele, assim pode ser recuperado.

Estrutura de Diretórios

Assim temos dois níveis principais:

- Diretório de trabalho
- Área de preparo
- Diretório `.git` que vai ser o repositório ou banco de dados local





Version control system fluxodetrabalho

Diretórios quando se trabalha com Git

Fluxo de Trabalho do Git

Iniciando um Repositório

Devemos usar o comando abaixo para iniciar o repositório para que o Git consiga ver os arquivos.

```
md MilfsGo # Cria a pasta
cd MilfsGo # acessa a pasta
git init
```

Fazendo Alterações

Agora vamos fazer alterações básicas como adicionar um *README* para o projeto.

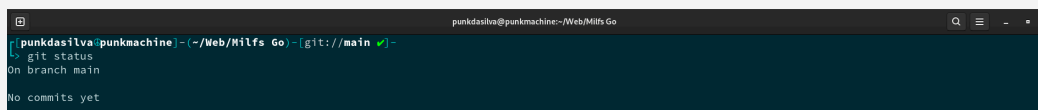


README são arquivos geralmente em markdown (.md) para registrar a documentação do repositório com informações importantes como:

- Nome
- Descrição
- Como usar
- Etc

Verificando Status

```
git status
```



```
punkdasilva@punkmachine ~/Web/Milfs Go
git status
On branch main
No commits yet
```

Version control system gitstatus

Resultado da execução do comando

Comandos Essenciais do Git

Cheat Sheet (Tabela de preguiçoso)



American pie its not what it looks like

Essa tabela fornece uma visão geral dos principais comandos Git e suas funcionalidades básicas.

Comando Git	Descrição
<code>git init</code>	Inicializa um novo repositório Git
<code>git add <arquivo></code>	Adiciona um arquivo modificado à área de stage
<code>git add .</code>	Adiciona todos os arquivos modificados à área de stage
<code>git commit -m "Mensagem do commit"</code>	Cria um novo commit com a mensagem especificada
<code>git mv <arquivo-original> <arquivo-novo></code>	Renomeia ou move um arquivo no repositório

Links e Referências

- GIT-SCM.COM. Git - Documentation. Disponível em: <https://git-scm.com/doc> (<https://git-scm.com/doc>).
- YOUTUBE. YouTube. Disponível em: <https://www.youtube.com/watch?v=un8CDE8qOR8> (<https://www.youtube.com/watch?v=un8CDE8qOR8>).
- GITLAB. GitLab Documentation. Disponível em: <https://docs.gitlab.com/> (<https://docs.gitlab.com/>).
- GITHUB. Git Cheat Sheet. Disponível em: <https://education.github.com/git-cheat-sheet-education.pdf> (<https://education.github.com/git-cheat-sheet-education.pdf>).