GPS - Gestão de Projetos de Software

Aula 3: Planejamento no GitLab

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

2º Semestre de 2026

Sumário

1. Introdução
2. Estruturando o Ambiente: Controle de Versão e Monorepos
3. Gestão de Tarefas: Issues e Kanban
4. Cronograma com Milestones
5. A Mágica da Automação: CI/CD
6. O Trabalho Prático

A Oficina do Projeto

Conselho do Mentor

Com a viabilidade aprovada e o Charter assinado, é hora de sujar as mãos. Você achou que ia abrir o editor de código e sair digitando? Negativo. Primeiro a gente organiza a oficina. Organização, processos e automação são o que separam a minha tecnologia corporativa das armaduras de sucata feitas numa caverna.

O sucesso de um software não vem apenas do código, mas de **como a equipe trabalha junta**. Nesta aula, entramos no ecossistema do GitLab para estruturar o ambiente do Assistente de IA.

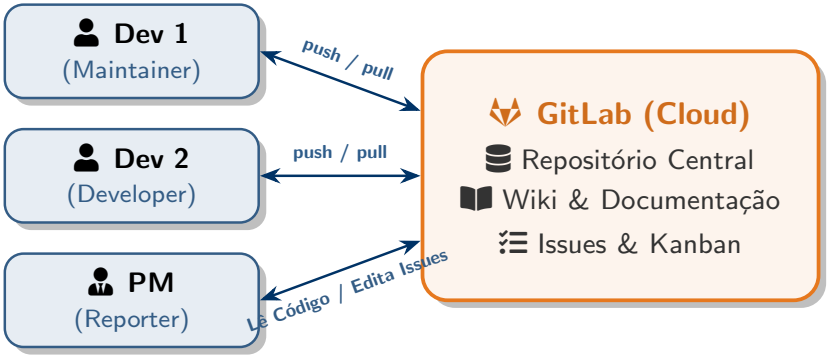
O GitLab: Muito além do Git

Não é mais apenas um "guardador de código":

- Deixou de ser apenas controle de versão.
- É uma plataforma completa de **DevSecOps** e Governança de Projetos.
- Integra repositório, gestão de tarefas (Issues), Wikis, Milestones, Integração e Entrega Contínuas (CI/CD) no mesmo ambiente.

Git local vs GitLab (O "Google Drive" de Código)

Se o **Git** no seu PC é como criar um documento no Word (onde você salva versões no seu HD), o **GitLab** é como o **Google Drive/Docs** (onde a equipe inteira colabora e sincroniza).



Os 3 Estados do Git Local (A Esteira de Versão)

O Git não salva arquivos imediatamente no histórico. Ele utiliza uma **esteira de 3 áreas locais**:

- **Untracked/Modified:** Alterações livres no seu espaço de trabalho.
- **Staged:** Arquivos selecionados para o commit (`git add <arquivo>`).
- **Committed:** Snapshot gravado no repositório (`git commit -m`).
- **.gitignore:** Ignora chaves (`.env`), caches (`__pycache__`) e dependências.

O Ciclo de Vida Local: Verbos e Comandos Essenciais

Comandos do Dia a Dia Local:

- `git status` → O raio-X obrigatório. Mostra quais arquivos foram modificados, preparados ou ignorados.
- `git diff` → Mostra linha a linha o que mudou antes do commit.
- `git add <arq>` → Move arquivos para a *Staging Area* (`git add .` adiciona tudo).
- `git commit -m "msg"` → Grava a foto no histórico local com mensagem semântica.

Comandos de Apoio e Segurança:

- `git stash` → A "gaveta de rascunhos". Salva temporariamente mudanças não commitadas para trocar de branch limpo.
  - `git stash pop` recupera o rascunho.
- `git log --oneline` → Histórico resumido dos últimos commits.
- `git checkout -b <nome>` → Cria e entra em uma nova branch.

Estratégia de Repositório: Polyrepo vs Monorepo

Polyrepo (Múltiplos Repos)

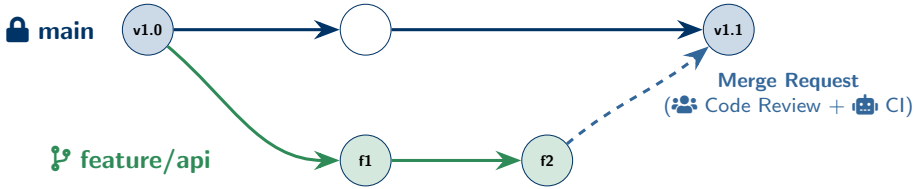
- Cada camada (Front, Back, DB) em um repositório isolado.
- Comum em arquiteturas de *Microserviços*.
- **Problema:** Gera grande sobrecarga de gestão e configuração de CI/CD para equipes enxutas.

Monorepo (Repositório Único)

- Centraliza *tudo* no mesmo lugar (código, interface, testes, documentação, prompts).
- **Vantagem:** Rastreabilidade transversal. Um único commit/MR atualiza a lógica, ajusta o teste e atualiza a Wiki de forma atômica.

O Fluxo Protegido (GitLab Flow)

No Monorepo, aplicamos o **GitLab Flow** para garantir que a versão de produção seja sempre estável e auditável:

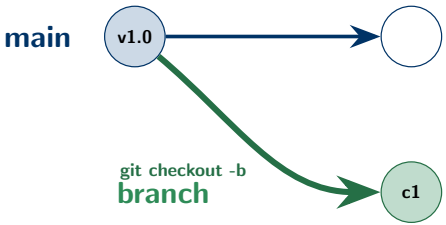


A `main` é blindada. O trabalho ocorre em branches isoladas e só entra na principal via **Merge Request** aprovado.

1. Branch: A Linha do Tempo Isolada

O que é uma Branch?

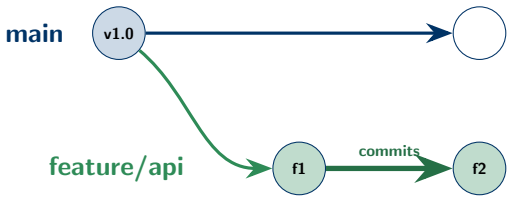
- Uma linha paralela de desenvolvimento no Git.
- Permite criar, alterar e testar código **sem risco** de quebrar a versão principal (`main`).
- **Regra de Ouro:** Nunca faça commits diretos na `main`!
- Boas práticas: Branches curtas (*short-lived*) com nomes semânticos:
  - `feature/nome-da-tarefa`
  - `bugfix/correcao-do-erro`
  - `docs/atualizacao-wiki`



Ramificação
Espaço isolado e seguro para novos commits.

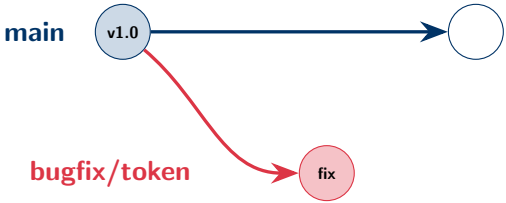
A Branch de Funcionalidade (`feature/`)

- Criada para implementar uma **nova capacidade** de valor para o usuário.
- No Projeto Integrador do Assistente de IA:
  - `feature/conexao-telegram`
  - `feature/prompt-socratico`
  - `feature/interface-web`
- **Conexão Ágil:** Cada Feature nasce diretamente vinculada a uma **Issue** no Board Kanban.
- Vários commits pequenos e atômicos durante o desenvolvimento.



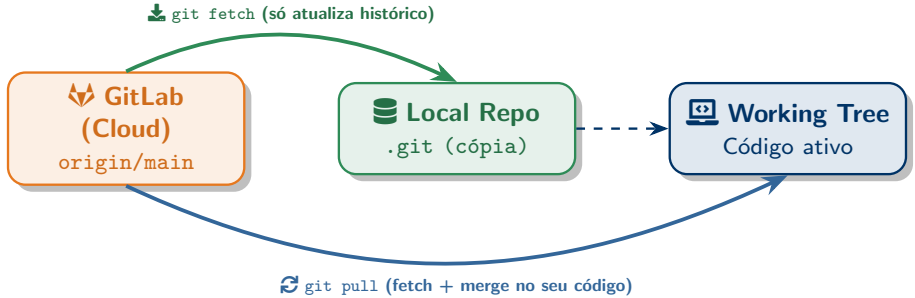
A Branch de Correção (`bugfix/`)

- Criada especificamente para **isolar e corrigir falhas** identificadas em testes ou em produção.
- Exemplos no projeto:
  - `bugfix/erro-limite-tokens-api`
  - `bugfix/timeout-resposta-llm`
  - `hotfix/crash-ao-iniciar-bot`
- **Atenção:** Não tente consertar "direto na main". O bugfix precisa passar pelo mesmo crivo de testes automatizados do CI!



4. Sincronização com a Nuvem: `git fetch` vs `git pull`

Como o seu repositório local conversa com o repositório remoto no GitLab?



`git fetch`:

- Baixa histórico e novas branches da nuvem.
- **Não altera** seus arquivos locais. 100% seguro para espiar novidades.

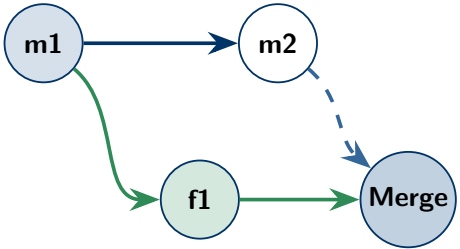
`git pull`:

- Baixa e já funde na sua branch atual.
- Se houver edições conflitantes, exige resolução de **Conflito de Merge**.

5. `git rebase`: Mantendo o Histórico Limpo e Linear

O que fazer quando a `main` avançou enquanto você trabalhava na sua `feature/`?

Opção 1: `git merge main`



Gera commits de merge poluídos que dificultam a leitura ("teia de aranha").

Opção 2: `git rebase main` (✔ **Recomendado**)

Reescreve a base da branch, colocando seu commit no topo da `main`. Histórico linear!

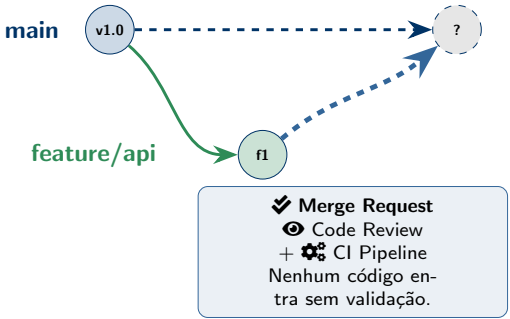
Regra de Ouro do Rebase

Use `git rebase` apenas na sua branch local antes de abrir o Merge Request. **Nunca faça rebase na `main` ou em branches públicas!**

6. Merge Request: O Portão da Governança

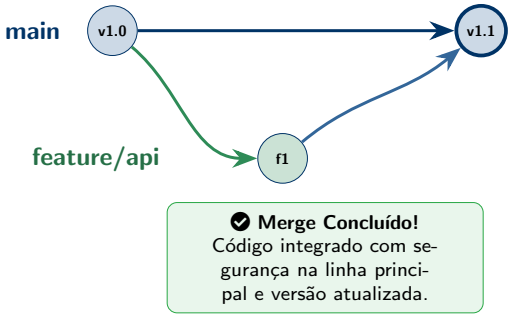
O que acontece no Merge Request (MR)?

- É o **pedido formal** para integrar sua branch (`feature/` ou `bugfix/`) na `main`.
- Os 3 Pilares de Proteção:
  1. **Code Review (👥):** Outro colega inspeciona o código antes do aceite.
  2. **Pipeline de CI (🤖):** Testes rodam sozinhos na nuvem; se quebrar, bloqueia!
  3. **Rastreabilidade (🔗):** Usar `Closes #12` fecha a Issue no Kanban automaticamente.



A Fusão e a Nova Versão (`v1.1`)

- Após a aprovação no MR e sucesso do CI, o código da branch é **fundido** na `main`.
- Gera um commit de merge que representa um novo estado estável do projeto.
- **Disparo Automático (CD):** O deploy contínuo entra em ação e publica a nova versão.
- A branch de trabalho (`feature/`) pode ser deletada com segurança para manter o repositório limpo.



A Issue: Seu Contrato de Trabalho

No mundo ágil, a unidade fundamental de trabalho (equivalente a pacotes da EAP do PMBOK) é a **Issue**.

- Não é uma lista de desejos ou bilhete vago. É um **contrato verificável**.
- Deve descrever o que será feito (Funcionalidade, Bug, Débito Técnico).
- Exige **Assignee** (Responsável) – Se não tem dono, não será feito.
- Exige **Labels** (Prioridade/Tipo) e **Weight** (Estimativa de esforço).

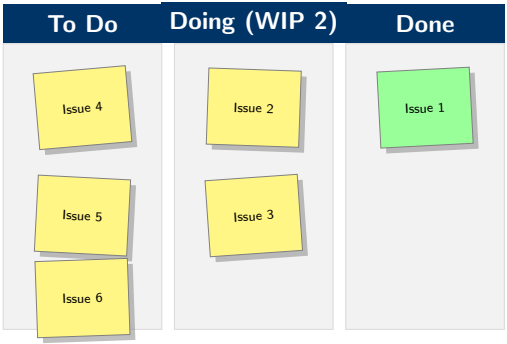
A Armadilha da Issue Épica

Cuidado com o Monstro

Uma Issue chamada "Fazer o Assistente de IA" é uma aberração gerencial. Não tem critério de aceitação claro e demorará semanas. Quebre em frações de valor!

Se uma Issue leva mais de 2 ou 3 dias intensos de código, ela é um **Épico** e precisa ser fatiada. **Certo:** "Criar endpoint de comunicação com Telegram" ou "Tratar erro de limite de tokens da API".

O Kanban: O Fluxo de Valor



Para evitar o caos cognitivo de 50 issues listadas, usamos **Boards Kanban**.

O Kanban **NÃO** é um sistema empurrado (o chefe joga trabalho na mesa). Ele é um **Sistema Puxado (Pull)**: o desenvolvedor puxa uma tarefa para "Em Progresso" apenas quando tem capacidade ociosa.

O Segredo: Limites de WIP

WIP (Work in Progress) = Trabalho em Progresso.

O segredo do Kanban não é ter colunas bonitas, mas limitar quantas coisas podem estar ocorrendo ao mesmo tempo.

- Se a equipe tem 4 pessoas, é insano ter 15 tarefas em "Em Progresso".
- O limite **força a equipe a terminar** o que começou antes de puxar algo novo.
- Reduz a troca de contexto (*context switching*) que destrói a produtividade técnica.

Escopo vs Tempo

- As **Issues** gerenciam e fracionam o **Escopo** (O que será feito).
- Os **Milestones** gerenciam e fracionam o **Tempo** (Quando um pacote de escopo deve estar pronto).

No GitLab, o Milestone funciona como um *Sprint*. Agrupa um pacote de Issues sob um objetivo estratégico temporal fixo (ex: 2 semanas).

Os Milestones do Assistente IA (Sprints)

O cronograma prático do semestre será mapeado no GitLab em 4 Milestones oficiais:

1. **Milestone 1 (Iniciação e Setup):** Repositório, *Project Charter* na Wiki, e a Prova de Conceito (código base operacionais conectando a uma API).
2. **Milestone 2 (Engenharia de Prompt e Core):** Implementação das regras rígidas do comportamento de "Coach" (Socrático) sem dar a resposta.
3. **Milestone 3 (Interface e Experiência):** Conexão do motor lógico com o canal final do usuário (ex: Telegram, Web).
4. **Milestone 4 (Monitoramento e Defesa):** Refinamento, métricas, tratamento de falhas e o Pitch final.

CI/CD: Integração e Entrega Contínuas

Antes: **"Mas na minha máquina funcionava!"** → Hoje: Removemos o fator humano.

- **CI (Integração):** A cada *commit*, um robô baixa o código, compila e roda testes. Se falhar, o *Merge* é bloqueado!
- **CD (Entrega):** Se passar com 100%, o robô empacota e envia direto para produção (Heroku, AWS) em segundos.

Tudo orquestrado pelo `.gitlab-ci.yml`.

Check 01: O Início do Planejamento

Preparando o Terreno (Avaliação 1)

- **Ambiente:** Repositório (Monorepo) inicializado no GitLab; proteção da branch `main` ativada para exigir MR.
- **Documentação (Integração):** *Project Charter* e matriz TELOS detalhados na Wiki.
- **Escopo e Tempo:** Board Kanban montado com Labels; os 4 *Milestones* criados; *Issues* do Sprint 1 populadas, estimadas e atribuídas.

Passo 0: Ideação do Assistente e Papéis na Equipe

Antes de criar o repositório no GitLab, a equipe precisa alinhar **o que vai construir** e **quem faz o quê**:

1. Concepção do Assistente de IA

- **Problema real:** Qual dor ou processo o bot vai resolver? (Ex: atendimento acadêmico, suporte a TI, tutor socrático, assistente financeiro).
- **Público & Canal:** Quem usará e por onde? (Bot Telegram, UI Web, CLI).
- **Comportamento da IA:** Tom de voz, regras de negócio e contexto inicial.

2. Divisão de Papéis e Focos

- **PM / Scrum Master:** Conduz Kanban, Wiki, Milestones e prazos.
- **Backend & IA:** Integração com LLMs (OpenAI/Gemini/Ollama), regras e prompts.
- **Frontend / Bot:** Interface do usuário (Telegram Bot / Web UI).
- **QA & DevOps:** Pipeline CI/CD, testes e governança de MRs.

Momento da Equipe: Reúnam-se agora para definir a ideia central e os responsáveis antes de abrir o GitLab!

Dinâmica em Sala: Hands-On no GitLab

Objetivo: Com a ideia do Assistente e os papéis definidos, realizar o setup no GitLab:

1. **Mestre do Repositório:** Um membro cria o repositório Monorepo (usando o template) e convida os demais membros + Professor.
2. **Charter:** Transferir a proposta e matriz TELOS para a Wiki do projeto.
3. **Configuração Agile:** Criar as colunas do Board Kanban (To Do, Doing, Review, Done).
4. **Sprints:** Cadastrar os 4 Milestones com as datas estimadas do plano de ensino.
5. **Divisão:** Distribuir as primeiras *Issues* de infraestrutura e prototipação.

Dúvidas sobre o Setup Ágil?

Email: alessio@cefetmg.br
Aula: Segunda-Feira, 7h às 8:40h (Sala 39)
WhatsApp: Dúvidas rápidas

Verifiquem as Issues automatizadas no repositório individual!

Próxima aula: Escopo (WBS) e Cronograma Matemático