MaCall

Guia de Boas Práticas em Versionamento

Devs & Qas

Documentação e boas práticas em git e github

v1.0.0

INTRODUÇÃO

Este guia de estilo foi elaborado para padronizar o código-fonte dos projetos da MaCall, garantindo que todos os desenvolvedores sigam as melhores práticas recomendadas. Nosso objetivo é promover a legibilidade, consistência e manutenção fácil do código.

OBJETIVO

O objetivo deste guia é padronizar o uso do Git e do GitHub entre todos os membros da equipe, garantindo clareza, organização e previsibilidade no ciclo de desenvolvimento. Dessa forma, evitamos conflitos desnecessários, facilitamos a colaboração entre diferentes times e asseguramos que cada entrega seja controlada, validada e integrada de forma eficiente, independentemente do nível de experiência dos desenvolvedores.

DEFINIÇÕES

Para fins deste guia, aplicam-se as seguintes definições:

- **Branch**: linha de desenvolvimento isolada onde criamos novas funcionalidades, correções ou ajustes sem impactar diretamente o código principal.
- **Main**: branch principal e estável, que reflete o código pronto para produção.
- **Homolog**: branch de testes internos, usada para validar antes da produção.
- **Dev**: branch de desenvolvimento contínuo, onde as equipes integram suas mudanças.
- **Pull Request (PR)**: solicitação para integrar o código de uma branch à outra, permitindo revisão antes do merge.
- **Versionamento**: numeração que identifica cada release (exemplo: `v1.2.0`).

1. Estrutura de Branches

- **main** → sempre estável, código pronto para produção.
- **dev** → ambiente de integração (tudo é testado aqui antes de ir para homolog/main).
- **homolog** → ambiente de staging para validações e testes em conjunto com o cliente.
- **feature/** → novas funcionalidades.
- **fix/** → correções de bugs.
- **hotfix/** → correção urgente em produção.

📌 Regra: branches devem ser **curtas e objetivas** (evitar acumular muitas mudanças).

2. Tempo de vida das Branches

- **Curto prazo**: uma branch deve existir apenas até a conclusão da tarefa.
- **Após integração (merge)**: a branch deve ser **deletada imediatamente**.
- **Longo prazo**: apenas `main` e `dev` permanecem vivas continuamente.
- **Evite branches órfãs** → manter branches antigas confunde e gera risco de bugs ao retomar código obsoleto.

3. Ciclo de Release / Integração

- **Criar branch** a partir de `dev`.
- **Desenvolver, testar localmente e commitar** em inglês (claro e objetivo).
- **Abrir Pull Request (PR)** para `dev`.
- Após aprovação, realizar **merge** → **dev**.
- **Quando for fechar um ciclo (release):**
    - Realizar **merge** **`dev`** → **`homolog`** para testes de validação.
    - Após aprovação em homologação, realizar **merge** **`homolog`** → **`main`**.
    - Criar **tag de versão** (exemplo: `v1.2.0`).

4. Pull Requests (PR)

- Sempre revisar antes do merge.

- Adicionar descrição clara: **o que foi feito e por quê**.
- Mínimo de **1 aprovação** antes de integrar.
- Evitar PRs gigantes (preferir pequenas entregas).

5. Padrão de Nomenclatura

🔹 Branches:

```
feature/nome-curto
fix/erro-login
hotfix/corrige-timeout
```

🔹 Commits:

```
feat: adiciona suporte a múltiplos idiomas
fix: corrige bug no fluxo de transcrição
docs: atualiza README com instruções
refactor: melhora performance da API
```

6. Gitflow Simplificado

- Criar branch → Trabalhar → Commitar → PR → Merge em `dev`.
- Release → Merge `dev` em `main` → Criar versão.

7. Estratégia de Versionamento (SemVer - Versionamento Semântico)

Usamos o padrão **SemVer (X.Y.Z)**:

- **X** (Major): mudanças grandes/incompatíveis.
- **Y** (Minor): novas funcionalidades sem quebrar código.
- **Z** (Patch): correções pequenas/bugs.

Exemplo:

- `v1.0.0` → primeira versão estável.
- `v1.1.0` → nova feature adicionada.
- `v1.1.1` → bug corrigido.

CONCLUSÃO

A aplicação consistente dessas boas práticas garante que o desenvolvimento nos repositórios de BACKEND, FRONTEND, LLM, RAG e SPEECH ocorra de forma organizada, previsível e colaborativa. O uso correto de branches, versionamento e pull requests facilita a integração de novas funcionalidades, reduz conflitos e melhora a qualidade das entregas.

Manter um fluxo simples e padronizado permite que toda a equipe, independentemente do nível de experiência com GitHub, consiga contribuir de maneira eficiente e segura. Assim, garantimos maior estabilidade nos ambientes, agilidade nos ciclos de release, um processo de desenvolvimento mais transparente e confiável para todos.