AERO - UFSM

APOSTILAS INTERATIVAS

Git

Edição: 2025

Desenvolvido por: Antônio Kipper e Bernardo Recktenvald

Sumário

1 Introdução Geral

O desenvolvimento de projetos de engenharia, especialmente na área aeroespacial, envolve o trabalho com múltiplos arquivos, modelos, códigos e documentações que são constantemente atualizados e aprimorados. Nesse cenário, manter a organização, a rastreabilidade e a colaboração entre os membros de um time é um grande desafio. Para lidar com essas demandas, o uso de sistemas de **controle de versão** tornou-se indispensável, permitindo registrar alterações, restaurar versões anteriores, integrar contribuições de diferentes colaboradores e garantir a integridade do projeto.

O **Git** é atualmente a ferramenta de controle de versão mais utilizada no mundo acadêmico e industrial. Desenvolvido inicialmente por Linus Torvalds para gerenciar o código-fonte do sistema operacional Linux, o Git tornou-se um padrão na indústria de tecnologia devido à sua eficiência, flexibilidade e suporte à colaboração distribuída. Diferente de sistemas tradicionais de controle de versão, o Git permite que cada desenvolvedor possua uma cópia completa do histórico do projeto, o que aumenta a segurança, a autonomia e a velocidade no desenvolvimento.

Além disso, o Git é amplamente integrado a plataformas como *GitHub*, *GitLab* e *Bitbucket*, que oferecem recursos adicionais para colaboração, revisão de código, gestão de tarefas e integração com ferramentas de automação. Essas plataformas são utilizadas não apenas por empresas de tecnologia, mas também por instituições acadêmicas e equipes de pesquisa, tornando o domínio do Git uma habilidade essencial para qualquer estudante ou profissional que deseja atuar em projetos modernos e colaborativos.

O objetivo central desta apostila é introduzir o Git como uma ferramenta prática e poderosa para organização, versionamento e colaboração no desenvolvimento de projetos. Ao longo do material, os leitores serão guiados por conceitos teóricos fundamentais e exemplos práticos, de forma que, ao final, estejam aptos a criar, gerenciar e compartilhar seus próprios repositórios, além de contribuir para projetos em equipe com segurança e eficiência.

1.1 Objetivo da Apostila

Esta apostila foi desenvolvida pela Escola Piloto de Engenharia Aeroespacial da Universidade Federal de Santa Maria com o objetivo de apresentar, de forma clara e progressiva, os fundamentos teóricos e práticos do **Git**. O material busca capacitar os leitores para:

- Compreender os conceitos essenciais de controle de versão e a importância do Git no gerenciamento de projetos;
- Criar e manipular repositórios locais, registrando o histórico de alterações por meio de commits;
- Trabalhar com repositórios remotos, utilizando plataformas como GitHub e GitLab para compartilhar e integrar projetos;
- Aplicar técnicas de *branching* e *merging* para organizar fluxos de trabalho e gerenciar o desenvolvimento colaborativo;

- Adotar boas práticas para escrever mensagens de commit claras, organizar arquivos e estruturar projetos de forma eficiente;
- Desenvolver autonomia para lidar com conflitos de versão, entender fluxos de trabalho de equipes e colaborar de maneira produtiva.

Mais do que ensinar comandos e procedimentos, esta apostila busca desenvolver uma **mentalidade de versionamento**, na qual os estudantes aprendem a pensar na evolução de um projeto como um processo estruturado, seguro e colaborativo. Ao final, espera-se que os leitores tenham adquirido não apenas conhecimento técnico, mas também uma visão prática de como o Git se integra ao ciclo de desenvolvimento de sistemas e aplicações na engenharia aeroespacial.

1.2 Público-Alvo

Esta apostila foi elaborada especialmente para **estudantes dos primeiros semestres do curso de Engenharia Aeroespacial** da Universidade Federal de Santa Maria, mas também pode ser útil para alunos de outros cursos de engenharia, ciência da computação e áreas afins. O material parte do princípio de que o leitor está em fase inicial de contato com ferramentas de desenvolvimento e colaboração, não exigindo conhecimentos avançados de programação ou experiência prévia com controle de versão.

Para os estudantes de engenharia aeroespacial, o domínio do Git é particularmente relevante, pois a área envolve projetos multidisciplinares que integram diferentes componentes, como simulações, modelagem matemática, controle de sistemas, análise de dados e desenvolvimento de software embarcado. Projetos colaborativos, sejam eles acadêmicos, laboratoriais ou de pesquisa, tornam-se mais organizados, seguros e eficientes com o uso do Git, permitindo gerenciar versões, acompanhar o progresso e integrar contribuições de diferentes integrantes da equipe.

Além disso, aprender Git desde os primeiros semestres proporciona aos estudantes uma base sólida para enfrentar desafios futuros, seja no desenvolvimento de trabalhos acadêmicos, na execução de projetos de pesquisa, na participação em equipes de competição tecnológica ou na preparação para estágios e oportunidades no setor aeroespacial.

1.3 Estrutura da Apostila

A apostila foi organizada de forma progressiva, abordando desde os conceitos fundamentais até práticas mais avançadas de versionamento e colaboração. Sua estrutura é dividida em seções principais:

- **Conceitos Básicos de Git:** Introdução ao controle de versão, instalação e configuração inicial.
- **Primeiros Passos com Git:** Criação de repositórios, rastreamento de arquivos, commits e fluxo de trabalho.
- **Trabalhando com Repositórios Remotos:** Integração com plataformas como GitHub, envio e atualização de projetos.

- **Colaboração e Trabalho em Equipe:** Revisão de código, pull requests e boas práticas de contribuição.
- **Branching e Merging:** Criação e gerenciamento de branches, resolução de conflitos e organização do desenvolvimento.
- **Boas Práticas:** Estratégias para mensagens de commit, organização de repositórios e metodologias como GitFlow.
- **Exercícios Práticos:** Atividades guiadas para consolidar os conhecimentos, incluindo um mini-projeto colaborativo.
- **Recursos e Referências:** Fontes de estudo e materiais de apoio para aprofundar o aprendizado.

Com essa abordagem, espera-se que o leitor desenvolva não apenas o domínio técnico sobre os comandos do Git, mas também a compreensão de como aplicar essas ferramentas para melhorar a organização, a produtividade e a qualidade de projetos individuais e colaborativos.

2 Conceitos Básicos de Git

2.1 Motivação para o Controle de Versão

Durante a graduação em Engenharia Aeroespacial, os estudantes frequentemente trabalham em projetos que envolvem diversos arquivos, como relatórios, códigos de simulação, diagramas, modelos matemáticos e documentação técnica. Esses projetos evoluem constantemente: novos recursos são adicionados, erros são corrigidos e diferentes versões dos arquivos são criadas ao longo do tempo. Em equipes, esse processo torna-se ainda mais complexo, pois várias pessoas modificam os mesmos arquivos simultaneamente.

Em um cenário sem controle de versão, os problemas aparecem rapidamente. Imagine um grupo de estudantes desenvolvendo um modelo de dinâmica orbital no *MATLAB/-Simulink* para uma disciplina de Mecânica Espacial. Cada integrante precisa ajustar parâmetros, modificar funções e atualizar gráficos. Sem uma ferramenta adequada, seria comum surgirem situações como:

- Várias cópias do mesmo arquivo com nomes diferentes, como `simulacao_final.m`, `simulacao_final_corrigida.m` ou `simulacao_nova_versao_FINAL.m`;
- Perda de trabalho por sobrescrever arquivos de colegas sem querer;
- Dificuldade em identificar qual versão contém os resultados corretos;
- Retrabalho ao tentar combinar manualmente alterações feitas por diferentes membros da equipe;
- Falta de histórico, tornando impossível "voltar atrás" para recuperar uma versão anterior que funcionava.

Agora, considere outro exemplo: um projeto interdisciplinar envolvendo a modelagem aerodinâmica de um veículo hipersônico. Os alunos trabalham com diversos tipos de arquivos — simulações de dinâmica de fluidos, códigos de processamento de dados, gráficos de desempenho, relatórios técnicos e apresentações. Sem um sistema organizado, fica praticamente impossível coordenar todas as contribuições, documentar mudanças e garantir que os resultados sejam confiáveis.

É nesse contexto que entra o **Git**, um sistema de *controle de versão distribuído* que permite:

- Registrar todas as alterações realizadas no projeto, criando um histórico completo e detalhado;
- Criar diferentes versões dos arquivos sem precisar duplicá-los;
- Trabalhar de forma colaborativa, permitindo que várias pessoas editem o mesmo projeto simultaneamente;
- Integrar mudanças de forma controlada, evitando conflitos e perdas de dados;

- Retornar facilmente para versões anteriores, caso seja necessário corrigir erros ou recuperar resultados antigos.

O Git funciona como uma **"linha do tempo"** do projeto. Cada modificação feita nos arquivos pode ser registrada em um *commit*, criando um ponto de restauração com informações sobre quem alterou, quando alterou e por quê. Isso é especialmente útil em trabalhos acadêmicos e projetos colaborativos, pois permite que todos acompanhem o histórico e a evolução dos resultados.

Além disso, quando combinado com plataformas como o *GitHub* e o *GitLab*, o Git potencializa a colaboração entre equipes, permitindo que os integrantes compartilhem seus projetos, revisem código, comentem alterações e integrem contribuições de forma organizada. Para estudantes de Engenharia Aeroespacial, isso significa maior eficiência no desenvolvimento de simulações, controle de modelos complexos e integração de dados experimentais.

Em resumo, o Git não é apenas uma ferramenta para desenvolvedores de software: ele é um aliado fundamental no gerenciamento de projetos de engenharia. Ele garante organização, segurança, rastreabilidade e colaboração, preparando os estudantes para lidar com desafios acadêmicos e profissionais, tanto na universidade quanto na indústria aeroespacial.

2.2 O que é Git

O **Git** é um *sistema de controle de versão distribuído* desenvolvido por Linus Torvalds em 2005 para gerenciar o código-fonte do sistema operacional Linux. Em termos simples, o Git permite registrar, organizar e acompanhar todas as alterações realizadas em um conjunto de arquivos, garantindo segurança, rastreabilidade e eficiência no desenvolvimento de projetos.

Em vez de criar várias cópias de um mesmo arquivo com nomes diferentes, como `simulacao_final.m` ou `relatorio_versao2.docx`, o Git mantém um **histórico detalhado** de tudo o que foi modificado, quem fez a alteração, quando e por quê. Cada modificação é registrada por meio de um *commit*, que funciona como um "ponto de controle" na linha do tempo do projeto. Isso permite que você volte para versões anteriores a qualquer momento, sem perder dados.

Para estudantes de Engenharia Aeroespacial, isso é particularmente útil. Imagine um projeto de simulação de voo atmosférico desenvolvido no *MATLAB/Simulink*. Durante o semestre, vários ajustes são realizados: mudanças nos parâmetros de altitude, novas funções para modelar a resistência do ar e gráficos para analisar a estabilidade da trajetória. Sem Git, seria necessário salvar manualmente várias versões dos arquivos para evitar perder o progresso. Com o Git, basta registrar as alterações e, se necessário, restaurar qualquer estado anterior do projeto com um único comando.

Outro benefício do Git é que ele é **distribuído**: cada membro de uma equipe tem uma cópia completa do repositório, incluindo todo o histórico do projeto. Isso significa que o trabalho não depende de um servidor central — você pode continuar desenvolvendo mesmo sem conexão com a internet e sincronizar suas alterações mais tarde. Essa característica é valiosa para projetos colaborativos, como laboratórios, trabalhos em grupo e

competições acadêmicas, onde diferentes integrantes contribuem simultaneamente para o mesmo código, modelo ou relatório.

Em resumo, o Git é mais do que uma ferramenta para programadores: ele é um recurso essencial para organizar, versionar e colaborar em projetos complexos, garantindo que todas as etapas do desenvolvimento sejam registradas e facilmente acessíveis.

2.3 Git vs GitHub vs GitLab

É comum confundir o **Git** com plataformas como **GitHub** e **GitLab**, mas há uma diferença importante entre eles:

- **Git** — É a **ferramenta** de controle de versão em si. Ela roda no seu computador e permite criar repositórios, registrar alterações, criar branches, fazer merges e gerenciar o histórico do projeto. Você pode usar o Git localmente, sem precisar estar conectado à internet.
- **GitHub** — É uma **plataforma online** que hospeda repositórios Git e oferece recursos extras para colaboração, como revisão de código, gerenciamento de tarefas, abertura de *issues* e integração com ferramentas externas. É amplamente utilizada no meio acadêmico e industrial para armazenar projetos abertos e privados. Além disso, o GitHub facilita o compartilhamento de trabalhos com professores, colegas e equipes de pesquisa.
- **GitLab** — Também é uma **plataforma online** para hospedagem de repositórios Git, semelhante ao GitHub, mas com foco maior em integração contínua, automação de testes e gestão de projetos. Muitas empresas e instituições de pesquisa preferem o GitLab por permitir a instalação de servidores privados, garantindo maior controle sobre os dados e a segurança dos projetos.

Um exemplo prático: imagine que um grupo de estudantes de Engenharia Aeroespacial está desenvolvendo um **simulador de trajetória de foguetes** para um projeto interdisciplinar. O **Git** seria utilizado para gerenciar o histórico do código e das simulações localmente. Para colaborar com os colegas, compartilhar resultados e revisar contribuições, o grupo poderia criar um repositório remoto no **GitHub** ou no **GitLab**, permitindo que todos sincronizem suas alterações com facilidade. Caso o projeto fosse confidencial ou envolvesse dados sensíveis, o **GitLab** poderia ser configurado em um servidor interno da instituição, oferecendo mais privacidade e controle.

Em resumo:

- **Git** = ferramenta de controle de versão local.
- **GitHub/GitLab** = plataformas que usam o Git para facilitar colaboração, hospedagem e integração.

Na prática, você usará o Git para versionar e gerenciar seus arquivos, e o GitHub ou GitLab para compartilhar o projeto e trabalhar de forma colaborativa com sua equipe.

2.4 Instalação e Configuração Inicial do Git

Antes de começarmos a utilizar o Git, é necessário instalá-lo e preparar o ambiente de desenvolvimento. O processo de instalação varia de acordo com o sistema operacional, mas o objetivo final é o mesmo: permitir que você crie e gerencie repositórios locais e trabalhe com plataformas remotas, como GitHub e GitLab.

Nesta apostila, recomendamos também o uso do **Visual Studio Code (VS Code)** como editor de código. Ele é gratuito, multiplataforma e possui integração nativa com o Git, permitindo visualizar alterações, realizar commits, gerenciar branches e sincronizar repositórios diretamente.

A seguir, apresentamos os passos para instalação no macOS, Linux e Windows.

Instalação no macOS

1. **Verificar se o Git já está instalado** Abra o *Terminal* e execute:

```
git --version
```

   Se o Git já estiver instalado, o comando exibirá a versão atual. Caso contrário, siga os próximos passos.

2. **Instalar o Git usando o Homebrew (recomendado)** Primeiro, instale o Homebrew:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
   Homebrew/install/HEAD/install.sh)"
```

   Em seguida, instale o Git:

```
brew install git
```

3. **Instalar o Visual Studio Code** Acesse o site oficial: `https://code.visualstudio.com/` Baixe a versão para macOS e siga as instruções de instalação.

Instalação no Linux (Ubuntu/Debian)

1. **Verificar se o Git já está instalado** Abra o *Terminal* e execute:

```
git --version
```

   Caso não esteja instalado, prossiga com os próximos passos.

2. **Instalar o Git via gerenciador de pacotes** Para distribuições baseadas em Debian/Ubuntu:

```
sudo apt update
sudo apt install git
```

   Para distribuições baseadas em Fedora/RHEL:

```
sudo dnf install git
```

3. **Instalar o Visual Studio Code** Baixe o pacote `.deb` (para Ubuntu/Debian) ou `.rpm` (para Fedora) no site oficial: `https://code.visualstudio.com/`

   Alternativamente, no Ubuntu/Debian, você pode instalar via terminal:

```
sudo snap install code --classic
```

Instalação no Windows

1. **Baixar o instalador do Git** Acesse o site oficial: `https://git-scm.com/download/win` Faça o download do instalador mais recente.

2. **Executar o instalador** Durante a instalação, mantenha as opções padrão recomendadas, especialmente:

   - Instalar o **Git Bash** — um terminal otimizado para comandos Git.
   - Configurar o editor padrão como **Visual Studio Code**, caso já esteja instalado.

3. **Instalar o Visual Studio Code** Caso ainda não tenha o VS Code, baixe-o no site oficial: `https://code.visualstudio.com/` Durante a instalação, marque a opção *"Add to PATH"* para facilitar a integração com o Git.

Configuração Inicial do Git (para todos os sistemas)

Após instalar o Git, é necessário configurá-lo pela primeira vez. Essas configurações são feitas uma única vez e serão aplicadas a todos os seus repositórios no computador.

1. **Definir nome e e-mail** Esses dados serão utilizados para identificar quem realizou cada alteração no projeto:

```
git config --global user.name "Seu Nome"
git config --global user.email "seu.email@exemplo.com"
```

2. **Verificar as configurações** Para confirmar se as credenciais foram registradas corretamente:

```
git config --list
```

Após concluir esses passos, o ambiente estará pronto para criar repositórios, versionar arquivos e integrar seus projetos com plataformas como GitHub e GitLab.

2.5 Como Registrar-se no GitHub

Para começar a utilizar o GitHub e aproveitar todos os recursos de colaboração, siga este passo a passo detalhado para criar sua conta:

1. **Acesse o site oficial:** Abra o navegador e digite `https://github.com/`.

2. **Clique em "Sign up":** No canto superior direito da página inicial, clique no botão **Sign up** para iniciar o processo de registro, como mostrado na Figura 1.

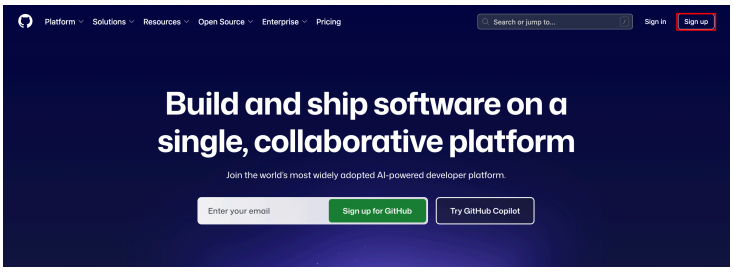

Figura 1 – Botão "Sign up".

3. **Informe seu e-mail, nome de usuário desejado e senha e crie sua conta.** Digite um endereço de e-mail válido. Você receberá um código de verificação neste e-mail. Escolha um nome de usuário único, que será seu identificador público no GitHub. Crie uma senha forte. Clique no botão "Create account"para continuar, como mostrado na Figura 2.

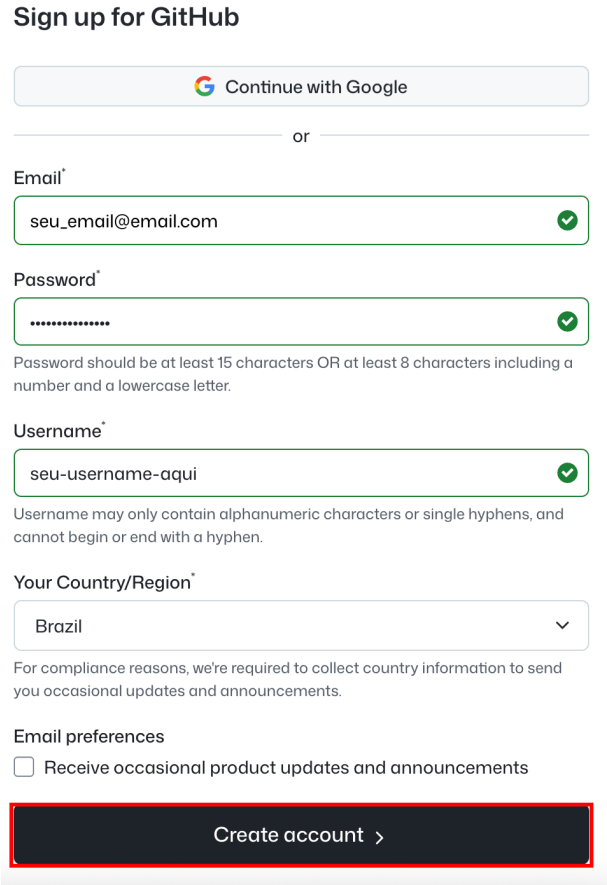

Figura 2 – Informações da conta e botão "Create account".

4. **Informe o código enviado ao e-mail.** Caso necessário, informe o código enviado ao seu e-mail.

5. **Configuração inicial:** Após o cadastro e primeira conexão na sua conta, você pode adicionar uma foto de perfil, preencher sua bio e configurar autenticação em dois fatores para maior segurança. Para isso, primeiro clique na sua foto de perfil, no canto direito superior da página inicial, como mostrado na Figura 3.

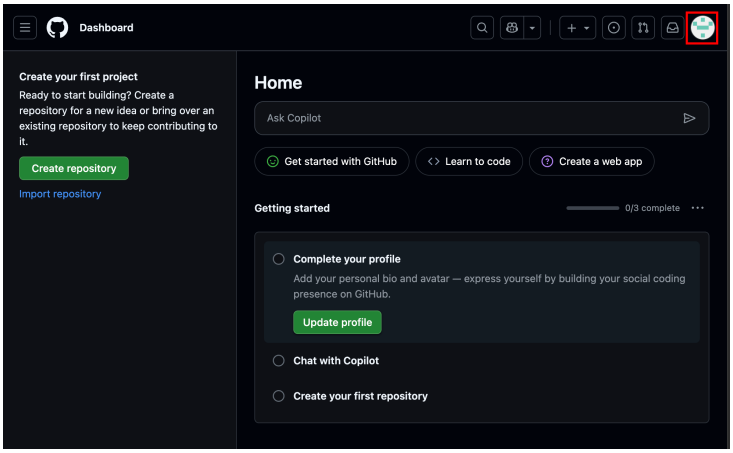

Figura 3 – Foto de perfil no Github.

6. **Navegue até as configurações do perfil:** Clique na aba ”Profile”, como mostrado na Figura 4.

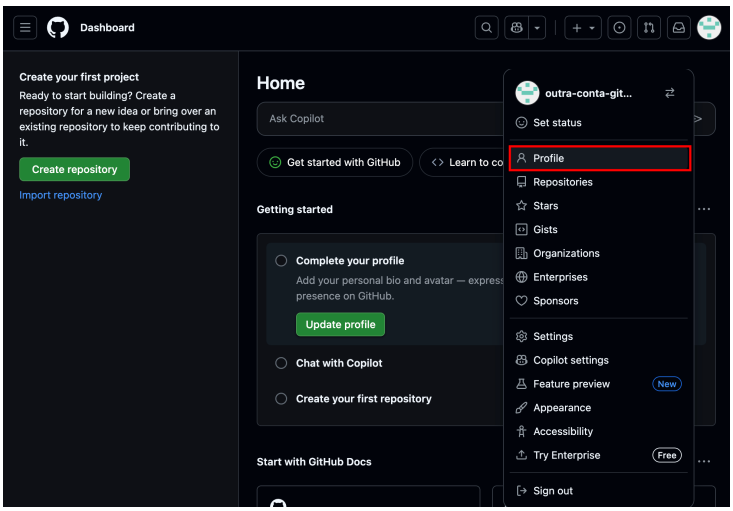

Figura 4 – Aba ”Profile”no Github.

Na sequência, clique em ”Edit profile”, como mostrado na Figura 5.

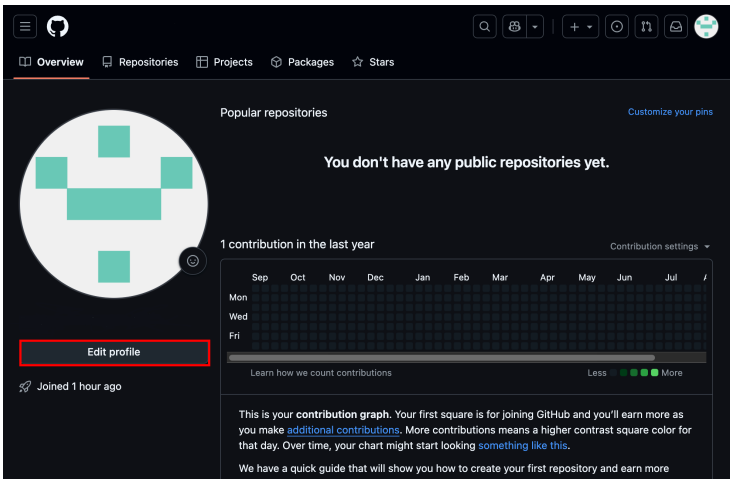

Figura 5 – Botão ”Edit profile”no Github.

Em seguida, edite as informações do seu perfil da maneira como desejar.

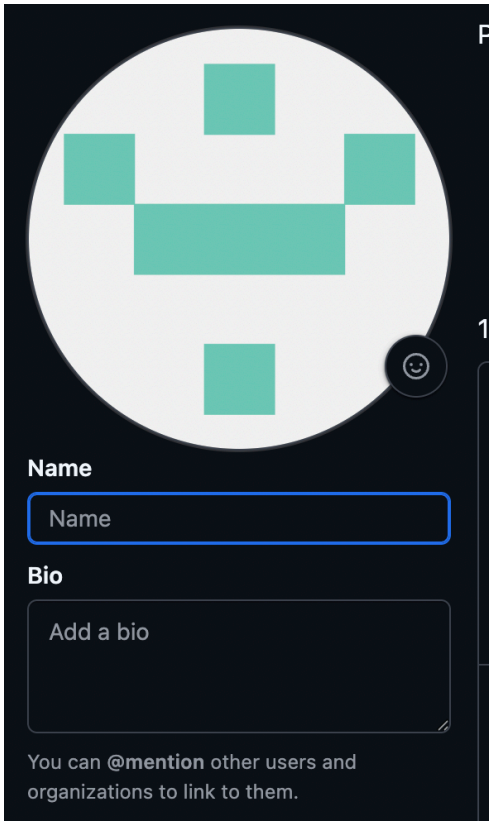

Figura 6 – Configuração do seu perfil no Github.

2.6 Criação de uma chave SSH

Para trabalhar com o GitHub, é possível realizar a comunicação com a plataforma de duas formas principais: HTTPS ou SSH.

A utilização de HTTPS é mais simples, pois não exige configuração inicial. No entanto, toda vez que for comunicar com o GitHub, é necessário informar usuário e senha ou um token de acesso.

Já o uso de SSH permite uma conexão mais prática e segura. Ao gerar uma chave SSH e adicioná-la à sua conta do GitHub, a autenticação passa a ser feita de forma automática, sem a necessidade de informar usuário e senha a cada comunicação. Essa abordagem é recomendada para quem utiliza o Git com frequência, pois agiliza o fluxo de trabalho e aumenta a segurança da comunicação.

Cada computador que se comunica com o GitHub precisa ter sua própria chave SSH. A chave é formada por duas partes: uma chave privada, que fica no computador e nunca deve ser compartilhada, e uma chave pública, que é adicionada à conta do GitHub. Assim, o GitHub reconhece o computador como autorizado a acessar os repositórios, sem precisar digitar usuário e senha a cada comunicação.

Para gerar uma chave SSH, no VS Code, abra o terminal integrado (`Ctrl +`) e siga os passos abaixo para gerar a chave SSH:

1. Digite o comando abaixo, substituindo seu e-mail pelo utilizado no GitHub:

```
ssh-keygen -t ed25519 -C "seu-email@exemplo.com"
```

2. Pressione `Enter` para aceitar o local padrão onde a chave será salva.

3. Opcionalmente, digite uma senha para proteger a chave privada ou pressione `Enter` para deixar em branco.

4. A chave será criada em duas partes: a chave privada (`id_ed25519`) e a chave pública (`id_ed25519.pub`).

A chave privada deve permanecer no computador e nunca ser compartilhada. A chave pública será usada para configurar o acesso ao GitHub.

Para adicionar sua chave pública ao Github, primeiramente, copie os conteúdos do arquivo `id_ed25519.pub`. Para fazer isso, use:

```
cat ~/.ssh/id_ed25519.pub
```

O comando `cat` mostra no terminal os conteúdos de um arquivo. Copie a chave mostrada. Ela deve ter o seguinte formato:

```
ssh-ed25519 <caracteres-aleatórios> seu-email@exemplo.com
```

Na sequência, siga os seguintes passos:

- **Clique novamente na sua foto de perfil no Github:** Desta vez, clique na aba "Settings".

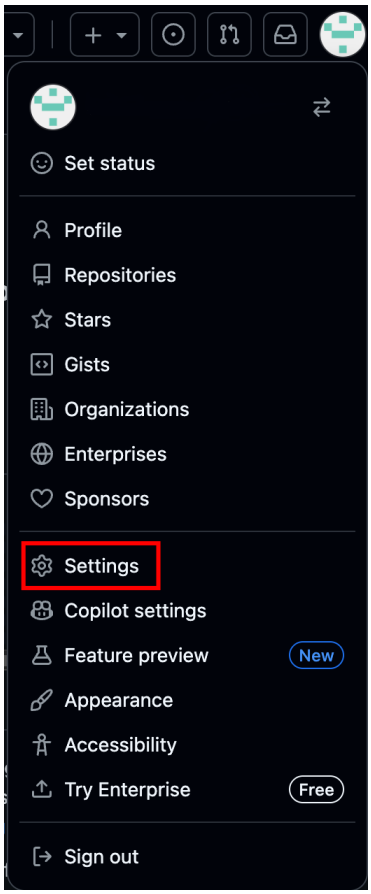

Figura 7 – Aba ”Settings”no Github.

- **Clique na opção ”SSH and GPG keys”:** Essa seção permite de configurar as chaves de acesso adicionados ao seu Github.

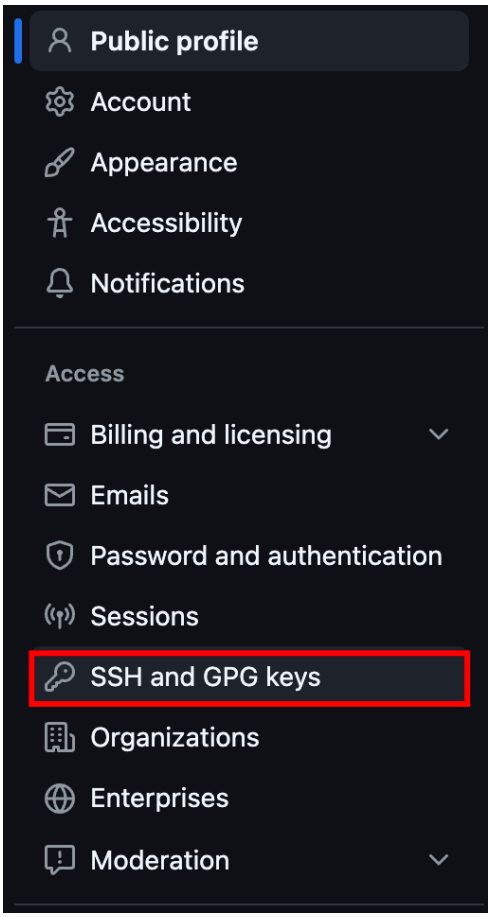

Figura 8 – Opção ”SSH and GPG keys”.

- **Clique em ”New SSH key”:** Clique nesse botão para adicionar uma nova chave SSH.

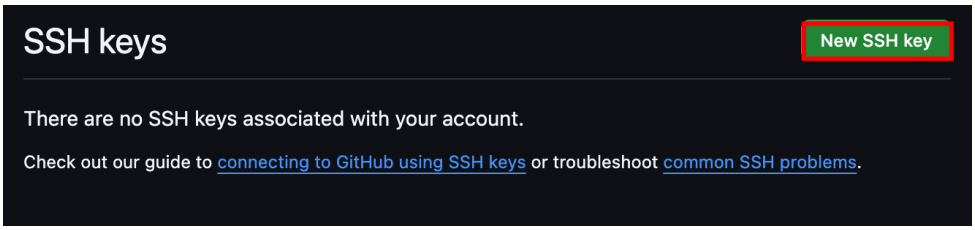

Figura 9 – Botão ”New SSH key”.

- **Adicione as informações:** Coloque um nome que identifique a chave SSH que você está adicionando. Lembre-se que uma chave de acesso por computador é necessária. Nomes descritivos são úteis, como ”Laptop”ou ”Computador de casa”. Cole a sua chave SSH pública. Clique em ”Add SSH key”para adicionar a chave.

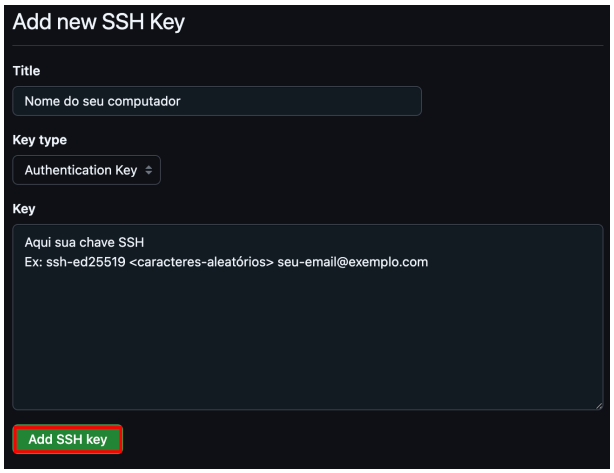

Figura 10 – Botão "Add SSH key".

3 Primeiros Passos com Git

3.1 Criando seu Primeiro Repositório

Um repositório é o local onde o Git armazena todo o histórico de alterações de um projeto. Ele pode ser **local** (no seu computador) ou **remoto** (em plataformas como GitHub ou GitLab). Criar seu primeiro repositório é o passo inicial para começar a versionar arquivos e acompanhar a evolução de seus projetos.

Para estudantes de Engenharia Aeroespacial, um exemplo típico seria um projeto de simulação de voo, com arquivos MATLAB/Simulink, planilhas de dados e relatórios técnicos. Ao criar um repositório, todas essas informações podem ser organizadas, versionadas e recuperadas facilmente.

Passo a Passo: Criando um Repositório Local

1. **Escolha ou crie uma pasta para seu projeto** Por exemplo, crie uma pasta chamada `simulacao_orbital` no seu computador.

2. **Abra o terminal** - No macOS ou Linux, use o *Terminal*. - No Windows, abra o **Git Bash**. - Alternativamente, você pode usar o terminal integrado do **Visual Studio Code**.

3. **Navegue até a pasta do projeto**

```
cd /caminho/para/simulacao_orbital
```

Comentário: O comando `cd` significa "change directory"e serve para trocar o diretório de trabalho atual no terminal. Assim, você garante que os próximos comandos do Git serão aplicados à pasta correta.

4. **Inicialize o repositório Git** Esse comando cria um repositório Git local na pasta atual, gerando a pasta oculta `.git` que armazenará o histórico do projeto:

```
git init
```

5. **Verifique se o repositório foi criado** Execute:

```
git status
```

O Git mostrará que você está em um repositório vazio e pronto para adicionar arquivos.

3.2 Rastreando Arquivos

Após criar seu repositório, o próximo passo é informar ao Git quais arquivos você deseja versionar. Esse processo é chamado de **staging**, ou preparação, e é realizado com o comando `git add`. Ele permite que você escolha exatamente quais alterações serão incluídas no próximo commit.

Adicionando arquivos ao repositório

Para adicionar todos os arquivos da pasta do projeto, execute:

```
git add .
```

O ponto (`.`) indica que todos os arquivos e subpastas do diretório atual serão rastreados.

Se você quiser adicionar apenas arquivos específicos, use:

```
git add simulacao.m relatorio.pdf
```

Ignorando arquivos desnecessários

Em projetos de engenharia aeroespacial, é comum gerar arquivos temporários ou grandes que não precisam ser versionados, como: - Logs de simulação (`*.log`) - Arquivos de saída do Simulink (`*.slx`) - Dados intermediários ou temporários (`*.mat`, `*.tmp`)

Para evitar que esses arquivos sejam rastreados, crie um arquivo chamado `.gitignore` na raiz do projeto e adicione as regras:

```
*.log
*.slx~
*.tmp
*.mat
```

Dessa forma, o Git ignorará automaticamente esses arquivos ao executar `git add`.

—

3.3 Fazendo seu Primeiro Commit

Depois de adicionar os arquivos desejados à área de staging, o próximo passo é registrar essas alterações no repositório através de um **commit**. Um commit funciona como um "ponto de restauração" no histórico do projeto, permitindo recuperar versões anteriores a qualquer momento.

Criando um commit

Para fazer o primeiro commit, execute:

```
git commit -m "Primeiro commit: estrutura inicial do projeto"
```

A opção `-m` permite adicionar uma mensagem descritiva para o commit. Mensagens claras ajudam você e sua equipe a entender rapidamente o que foi alterado em cada commit.

Verificando o commit

Após o commit, você pode conferir o histórico do repositório:

```
git log
```

O Git exibirá informações como:

- Identificador único do commit (hash)
- Autor do commit
- Data e hora do commit
- Mensagem do commit

4 Trabalhando com Repositórios Remotos

4.1 Conectando-se ao GitHub

O GitHub é uma plataforma de hospedagem de repositórios Git remotos que permite armazenar, compartilhar e colaborar em projetos. Para enviar e receber alterações de um repositório remoto, é necessário primeiro estabelecer uma conexão entre o seu repositório local e o repositório no GitHub.

Criando uma conta no GitHub

Se você ainda não possui uma conta, siga estes passos:

1. Acesse o site: `https://github.com/`
2. Clique em **Sign up** e preencha os campos solicitados:
   - Nome de usuário
   - E-mail
   - Senha
3. Siga as instruções para verificar o e-mail e completar o registro.

Criando um repositório remoto no GitHub

Após criar sua conta, é hora de criar o repositório remoto:

1. Faça login no GitHub e clique no botão **New repository** no canto superior direito.
2. Defina o nome do repositório, por exemplo: `simulacao_orbital`.
3. Opcional: adicione uma descrição do projeto.
4. Escolha se o repositório será **público** ou **privado**.
5. Não selecione a opção **Initialize this repository with a README**, pois você já possui um repositório local. Isso evita conflitos.
6. Clique em **Create repository**.

Conectando seu repositório local ao remoto

Depois de criar o repositório remoto, copie a URL HTTPS do repositório, por exemplo:

`https://github.com/seu-usuario/simulacao_orbital.git`

No terminal, dentro da pasta do seu projeto local, execute:

```
1 git remote add origin https://github.com/seu-usuario/
    simulacao_orbital.git
```

O comando acima associa o repositório remoto chamado `origin` ao seu repositório local. O nome `origin` é convencional e será usado nas operações de envio e atualização.

Para verificar se a conexão foi estabelecida corretamente, execute:

```
git remote -v
```

4.2 Enviando seu Trabalho

Depois de criar e commitar alterações no repositório local, você pode enviá-las para o repositório remoto no GitHub. Isso garante que seu projeto fique armazenado na nuvem e possa ser acessado de outros computadores ou compartilhado com colegas.

Passo a passo

1. Certifique-se de que todos os arquivos desejados foram adicionados e commited no repositório local:

```
git add .
git commit -m "Mensagem descrevendo as alterações"
```

2. Envie as alterações para o repositório remoto:

```
git push origin main
```

- `origin` é o nome do repositório remoto. - `main` é o nome do branch principal. - Ao usar HTTPS, o Git solicitará seu nome de usuário e senha ou token pessoal.

—

4.3 Atualizando seu Repositório

Quando outras pessoas contribuem para o mesmo projeto, ou quando você acessa o projeto de outro computador, é importante atualizar seu repositório local para refletir as alterações feitas remotamente.

Passo a passo

1. Baixe as alterações do repositório remoto:

```
git fetch origin
```

2. Integre as alterações ao seu branch local:

```
git pull origin main
```

4.4 Entendendo o Fluxo de Trabalho do Git

O Git possui um fluxo de trabalho que permite gerenciar e versionar projetos de forma organizada. Ele pode ser resumido nos seguintes passos:

1. Modificação e adição de arquivos

Você trabalha no seu projeto criando ou modificando arquivos, por exemplo, códigos MATLAB, modelos Simulink ou relatórios técnicos. Para informar ao Git quais arquivos devem ser incluídos no próximo commit, use:

```
git add nome_arquivo.ext
```

Ou para adicionar todos os arquivos modificados:

```
git add .
```

2. Commit

Após adicionar os arquivos à staging area, você cria um commit para registrar as alterações no histórico do repositório local:

```
git commit -m "Mensagem descrevendo as alterações"
```

3. Envio para o repositório remoto

Para compartilhar seu trabalho com outros colaboradores ou armazenar uma cópia na nuvem, envie as alterações para o repositório remoto no GitHub:

```
git push origin main
```

4. Recuperando alterações do remoto

Quando outras pessoas fazem alterações ou você acessa o projeto de outro computador, é necessário atualizar seu repositório local:

```
git pull origin main
```

O comando `pull` baixa as alterações do repositório remoto e as integra ao seu branch local, garantindo que seu projeto esteja atualizado.

Resumo do fluxo

O ciclo básico do Git pode ser representado assim:

1. Modificar ou criar arquivos (Working Directory)
2. Adicionar arquivos ao staging area (`git add`)
3. Criar commits (`git commit`)
4. Enviar alterações para o remoto (`git push`)
5. Atualizar o repositório local (`git pull`)

Este fluxo permite manter o histórico de alterações organizado, colaborar com colegas e recuperar versões anteriores do projeto sempre que necessário.

4.5 Exemplo Prático: Criando e Enviando um Projeto

Neste exemplo, vamos criar um projeto local simples de simulação de voo e enviá-lo para um repositório remoto no GitHub, seguindo o fluxo de trabalho do Git.

1. Criar a pasta do projeto

Crie uma pasta chamada `simulacao_orbital` e adicione alguns arquivos de exemplo:

- `simulacao.m` – código MATLAB de simulação de trajetória
- `modelo.slx` – modelo Simulink do sistema
- `relatorio.pdf` – relatório técnico inicial

2. Inicializar o repositório local

Abra o terminal na pasta do projeto e execute:

```
git init
```

3. Adicionar arquivos à staging area

```
git add .
```

4. Criar o primeiro commit

```
git commit -m "Primeiro commit: estrutura inicial do projeto"
```

5. Conectar ao repositório remoto no GitHub

Após criar o repositório remoto no GitHub (ex.: `simulacao_orbital`), associe-o ao repositório local:

```
git remote add origin https://github.com/seu-usuario/
    simulacao_orbital.git
```

6. Enviar arquivos para o repositório remoto

```
git push origin main
```

7. Atualizar o repositório local

Se houver alterações no remoto ou se outros colaboradores tiverem feito commits, atualize seu projeto local:

```
git pull origin main
```

Resumo do exemplo

Este exemplo prático mostra todo o ciclo básico de trabalho com Git:

1. Criar e modificar arquivos (Working Directory)
2. Adicionar arquivos ao staging area (`git add`)
3. Criar commits (`git commit`)
4. Conectar o repositório local ao remoto (`git remote add`)
5. Enviar alterações para o remoto (`git push`)
6. Recuperar alterações do remoto (`git pull`)

Seguindo esse fluxo, você mantém seu projeto organizado, versionado e pronto para colaboração com colegas ou acesso remoto.

Esse fluxo de trabalho pode ser visualizado graficamente na Figura 11.

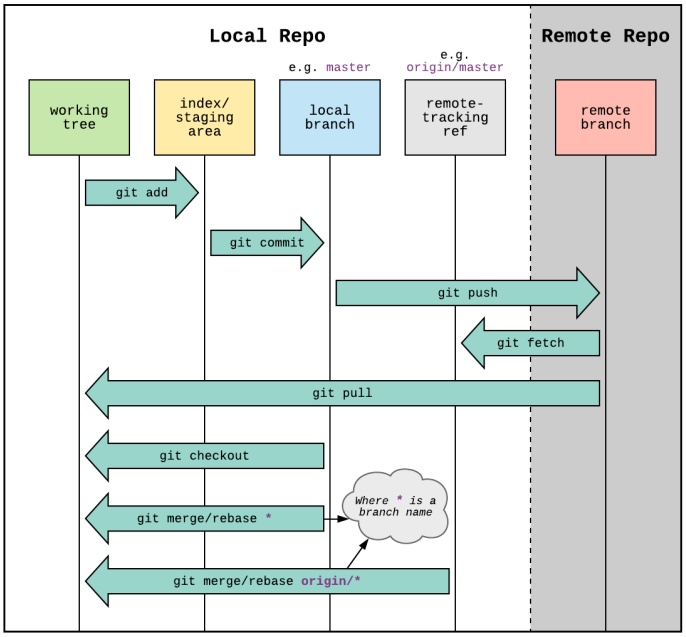

Figura 11 – Fluxo gráfico de trabalho com repositório remoto.

5 Branching e Merging

O uso de **branches** (ramificações) é um dos recursos mais poderosos do Git para organizar o desenvolvimento de projetos colaborativos. Branches permitem que diferentes funcionalidades, correções ou experimentos sejam desenvolvidos de forma independente, sem afetar o código principal do projeto. Após o desenvolvimento e testes, as alterações podem ser integradas ao projeto principal por meio do processo de **merge** (mesclagem).

5.1 Introdução aos Branches

Um **branch** é uma linha de desenvolvimento separada dentro do repositório Git. Por padrão, todo repositório possui um branch principal chamado `main` (ou `master`). Ao criar branches, você pode trabalhar em novas funcionalidades, corrigir bugs ou testar ideias sem interferir no código estável do projeto.

Branches são especialmente úteis em equipes, pois cada membro pode trabalhar em sua própria tarefa e, depois de pronta, integrar as alterações ao branch principal.

5.2 Criando e Mudando de Branch

Para criar um novo branch, utilize o comando:

```
git branch nome-da-branch
```

Para alternar para o branch criado:

```
git checkout nome-da-branch
```

Ou, de forma combinada:

```
git checkout -b nome-da-branch
```

Exemplo: Para desenvolver uma nova funcionalidade de simulação, crie um branch chamado `feature/simulacao`:

```
git checkout -b feature/simulacao
```

5.3 Mesclando Branches

Após finalizar o desenvolvimento em um branch, é necessário integrar as alterações ao branch principal. Isso é feito com o comando `merge`:

```
git checkout main
git merge feature/simulacao
```

O Git irá combinar as alterações do branch `feature/simulacao` ao branch `main`. Se não houver conflitos, o merge será realizado automaticamente.

5.4 Resolvendo Conflitos de Merge

Conflitos de merge ocorrem quando duas ou mais pessoas modificam a mesma linha de um arquivo em branches diferentes. O Git sinaliza o conflito e pede que o usuário escolha qual versão manter.

Ao realizar um merge e encontrar um conflito, o arquivo afetado exibirá marcações como:

```
<<<<<<< HEAD
Conteúdo da branch atual
=======
Conteúdo da branch a ser mesclado
>>>>>>> feature/simulacao
```

Para resolver o conflito, edite o arquivo, escolha o conteúdo correto e remova as marcações. Depois, finalize o merge:

```
git add arquivo-afetado.ext
git commit -m "Resolve conflito de merge"
```

5.5 Rebasing (Opcional)

O **rebase** é uma alternativa ao merge para integrar alterações de um branch ao outro. Ele reorganiza o histórico de commits, tornando-o mais linear. O comando básico é:

```
git checkout feature/simulacao
git rebase main
```

O rebase é útil para manter um histórico limpo, mas deve ser usado com cautela em projetos colaborativos, pois pode reescrever o histórico de commits.

5.6 Exemplo Prático: Fluxo de Trabalho com Branches de Funcionalidade

Imagine que sua equipe está desenvolvendo um projeto de simulação de voo e precisa adicionar uma nova funcionalidade para calcular a altitude máxima.

1. **Criar um branch para a funcionalidade:**

```
git checkout -b feature/altitude-maxima
```

2. **Desenvolver e commitar as alterações:**

```
git add simulacao.m
git commit -m "Adiciona cálculo de altitude máxima"
```

3. **Mesclar o branch ao principal:**

```
git checkout main
git merge feature/altitude-maxima
```

4. **Resolver conflitos, se houver, e finalizar o merge:**

```
git add simulacao.m
git commit -m "Resolve conflito e integra cálculo de altitude
    máxima"
```

Esse fluxo permite que diferentes funcionalidades sejam desenvolvidas em paralelo, testadas e integradas ao projeto principal de forma organizada e segura.

6 Colaboração e Trabalho em Equipe

O Git é uma ferramenta poderosa para facilitar o trabalho colaborativo em projetos, permitindo que equipes trabalhem simultaneamente em diferentes partes de um projeto sem conflitos e com total rastreabilidade. Nesta seção, exploraremos como o Git pode ser utilizado para gerenciar contribuições de diferentes membros da equipe, revisar código e adotar boas práticas para garantir a eficiência e a organização do trabalho em grupo.

6.1 Pull Requests

O **Pull Request** (PR) é uma funcionalidade oferecida por plataformas como GitHub e GitLab que permite que um colaborador proponha alterações para um repositório. Ele é amplamente utilizado em projetos colaborativos para integrar contribuições de forma controlada e revisada.

O fluxo básico de um Pull Request é o seguinte:

1. O colaborador cria uma nova **branch** para desenvolver uma funcionalidade ou corrigir um problema.
2. Após realizar as alterações e fazer os *commits*, o colaborador envia a branch para o repositório remoto.
3. No repositório remoto, o colaborador abre um Pull Request, descrevendo as alterações realizadas e o motivo delas.
4. Outros membros da equipe revisam o Pull Request, sugerem melhorias e aprovam as alterações.
5. Após a aprovação, as alterações são integradas à branch principal (`main` ou `master`) por meio de um *merge*.

Os Pull Requests são uma excelente forma de garantir que todas as alterações sejam revisadas antes de serem integradas ao projeto, promovendo a qualidade do código e a colaboração entre os membros da equipe.

6.2 Revisão de Código

A revisão de código é uma etapa essencial no trabalho colaborativo, pois permite que os membros da equipe avaliem as contribuições uns dos outros, identifiquem erros, sugiram melhorias e garantam a consistência do projeto. Durante a revisão de um Pull Request, os revisores devem:

- Verificar se o código segue os padrões e boas práticas estabelecidos pela equipe.
- Garantir que as alterações não introduzam erros ou quebras no projeto.
- Testar as novas funcionalidades ou correções, quando aplicável.
- Sugerir melhorias para tornar o código mais eficiente, legível ou organizado.

A revisão de código não deve ser vista como uma crítica pessoal, mas como uma oportunidade de aprendizado e melhoria contínua para toda a equipe. Ferramentas como comentários em Pull Requests no GitHub ou GitLab tornam esse processo mais eficiente e colaborativo.

6.3 Boas Práticas em Projetos de Equipe

Para garantir o sucesso de um projeto colaborativo, é importante adotar boas práticas no uso do Git. Algumas recomendações incluem:

- **Criar branches para cada tarefa:** Cada funcionalidade ou correção deve ser desenvolvida em uma branch separada, com um nome descritivo, como `feature/simulacao` ou `bugfix/corrigir-grafico`.
- **Escrever mensagens de commit claras:** Cada commit deve ter uma mensagem que explique de forma objetiva o que foi alterado, como `"Adiciona função para calcular resistência do ar"`.
- **Sincronizar frequentemente:** Antes de iniciar uma nova tarefa, atualize sua branch local com as alterações mais recentes da branch principal para evitar conflitos.
- **Resolver conflitos de forma colaborativa:** Quando conflitos de versão ocorrerem, discuta com os membros da equipe para decidir a melhor forma de resolvê-los.
- **Revisar e testar antes de integrar:** Antes de fazer o *merge* de uma branch, certifique-se de que todas as alterações foram revisadas e testadas.

Essas práticas ajudam a manter o projeto organizado, reduzir erros e promover um ambiente de trabalho colaborativo e produtivo.

6.4 Exemplo Prático: Contribuindo para um Repositório Compartilhado

Vamos considerar um exemplo prático de colaboração em um projeto de simulação de voo atmosférico. Suponha que você e sua equipe estão desenvolvendo um modelo no MATLAB/Simulink e precisam adicionar uma nova funcionalidade para calcular a resistência do ar.

Passo 1: Criar uma branch para a tarefa

No terminal, crie uma nova branch para desenvolver a funcionalidade:

```
git checkout -b feature/resistencia-ar
```

Passo 2: Fazer as alterações e commits

Implemente a funcionalidade no código e registre as alterações com commits claros:

```
git add simulacao.m
git commit -m "Adiciona função para calcular resistência do ar"
```

Passo 3: Enviar a branch para o repositório remoto

Envie a branch para o repositório remoto para compartilhar seu trabalho:

```
git push origin feature/resistencia-ar
```

Passo 4: Abrir um Pull Request

No GitHub ou GitLab, abra um Pull Request para a branch `feature/resistencia-ar`, descrevendo as alterações realizadas.

Passo 5: Revisar e integrar as alterações

Os membros da equipe revisam o Pull Request, sugerem melhorias e, após a aprovação, fazem o *merge* da branch na branch principal:

```
git checkout main
git merge feature/resistencia-ar
```

Passo 6: Atualizar o repositório local

Após o *merge*, atualize seu repositório local para refletir as alterações mais recentes:

```
git pull origin main
```

Com esse fluxo, a equipe pode colaborar de forma eficiente, garantindo que todas as contribuições sejam revisadas, testadas e integradas de maneira

7 Recuperação de Versões e Versionamento

7.1 Visualização do Histórico de Commits

O histórico de commits permite acompanhar todas as alterações feitas no projeto, quem realizou cada mudança e quando. Para visualizar o histórico completo, utilize:

```
git log
```

No GitHub, acesse a aba **Commits** do repositório para ver o histórico online, incluindo autor, data e mensagem de cada commit.

7.2 Recuperação de Arquivos ou Versões Anteriores

Se for necessário restaurar um arquivo (caso ele tenha sido deletado, por exemplo), utilize:

```
git checkout <hash-do-commit> -- <nome-do-arquivo>
```

Caso seja necessário restaurar todo o projeto para um estado anterior, utilize:

```
git checkout <hash-do-commit>
```

Nesse caso, tenha atenção: ao fazer isso, o Git colocará o repositório em um estado chamado **detached HEAD**. Isso significa que você estará apenas **visitando** um commit antigo, e não estará mais no seu branch atual (por exemplo, `main` ou `develop`).

Se o seu objetivo for apenas consultar o estado do projeto ou recuperar arquivos antigos, isso não é um problema. No entanto, se você fizer alterações e criar novos commits nesse estado, eles não estarão vinculados a nenhum branch e poderão ser perdidos ao trocar de branch.

Se você quiser apenas olhar um commit antigo, depois de terminar, basta voltar para o seu branch principal com:

```
git checkout main
```

Por outro lado, se o objetivo for **continuar trabalhando a partir de um commit antigo**, o recomendado é criar um novo branch antes de fazer alterações. Isso garante que seus novos commits fiquem salvos com segurança:

```
git checkout -b meu-novo-branch <hash-do-commit>
```

Dessa forma, você cria um branch começando do commit escolhido e evita problemas com o modo **detached HEAD**.

7.3 Reversão de Commits (Undo/Revert/Reset)

Para desfazer um commit que já foi enviado ao repositório remoto (usando `git push`), utilize:

```
git revert <hash-do-commit>
```

Se o commit ainda não foi enviado ao remoto, pode-se usar:

```
git reset --hard <hash-do-commit>
```

No GitHub, a reversão pode ser feita manualmente criando um novo commit que desfaz as alterações indesejadas.

7.4 Comparação entre Versões (Diff)

Para comparar as diferenças entre dois commits, utilize:

```
git diff <hash1> <hash2>
```

No GitHub, selecione dois commits ou branches e use a aba **Compare** para visualizar as diferenças entre eles.

7.5 Criação de Tags e Releases

Tags marcam pontos importantes no histórico, como versões estáveis. Para criar uma tag:

```
git tag -a v1.0 -m "Versão 1.0"
git push origin v1.0
```

No GitHub, utilize a aba **Releases** para criar e gerenciar versões do projeto.

7.6 Uso de Branches para Versionamento Seguro

Branches permitem desenvolver novas funcionalidades ou corrigir erros sem afetar o código principal. Para criar um branch:

```
git checkout -b nome-do-branch
```

No GitHub, branches são facilmente visualizados e gerenciados na interface do repositório.

7.7 Auditoria e Rastreabilidade (Blame, Log Detalhado)

Para identificar quem modificou cada linha de um arquivo, utilize:

```
git blame <nome-do-arquivo>
```

O comando `git log` também pode ser usado com filtros para auditoria detalhada:

```
git log -p <nome-do-arquivo>
```

No GitHub, use a opção **Blame** na visualização de arquivos para

8 Boas Práticas

Adotar boas práticas no uso do Git é essencial para garantir a organização, a rastreabilidade e a eficiência dos projetos colaborativos. Nesta seção, abordamos recomendações para escrever mensagens de commit claras, organizar repositórios, evitar erros comuns e implementar metodologias de desenvolvimento como o GitFlow.

8.1 Escrevendo Bons Commit Messages

Mensagens de commit bem escritas facilitam o entendimento do histórico do projeto e ajudam toda a equipe a acompanhar as alterações. Recomendações:

- Seja objetivo e claro: descreva o que foi alterado e por quê.
- Evite mensagens genéricas como `"Update"` ou `"Correções"`.
- Se necessário, adicione uma descrição mais detalhada após a primeira linha.

8.2 Organizando Repositórios

Um repositório bem organizado facilita a navegação e o entendimento do projeto. Dicas:

- Mantenha uma estrutura de pastas lógica (ex: `src/`, `docs/`, `data/`).
- Utilize arquivos `README.md` para documentar o objetivo do projeto e instruções de uso.
- Adote o `.gitignore` para evitar versionar arquivos desnecessários ou temporários.
- Padronize nomes de arquivos e pastas.

8.3 Erros Comuns a Evitar

Alguns erros podem comprometer a organização e a colaboração no projeto:

- Versionar arquivos grandes ou gerados automaticamente (ex: `.mat`, `.exe`, `.log`).
- Realizar commits diretamente na branch principal sem revisão.
- Não atualizar o repositório local antes de iniciar novas tarefas.
- Mensagens de commit vagas ou sem contexto.
- Não resolver conflitos de merge corretamente.

8.4 Assinatura de Commits (Opcional)

Assinar commits com uma chave GPG garante autenticidade e segurança, especialmente em projetos públicos ou críticos. Para configurar:

```
git config --global user.signingkey <ID-da-chave>
git commit -S -m "Mensagem assinada"
```

Mais detalhes podem ser encontrados na documentação oficial do Git: `https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work`

8.5 GitFlow - Metodologia de Desenvolvimento Colaborativo

O **GitFlow** é uma metodologia que define um fluxo de trabalho estruturado para equipes que usam Git. Ele organiza o desenvolvimento em diferentes tipos de branches, cada um com uma finalidade específica:

- `main` (ou `master`): branch principal, sempre estável.
- `develop`: branch de desenvolvimento, onde novas funcionalidades são integradas antes de serem lançadas.
- `feature/*`: branches para desenvolvimento de novas funcionalidades.
- `release/*`: branches para preparação de novas versões.
- `hotfix/*`: branches para correção rápida de bugs em produção.

O GitFlow facilita o gerenciamento de múltiplas tarefas simultâneas, garante que o código principal permaneça estável e organiza o processo de lançamento de versões.

Para implementar o GitFlow, existem extensões e ferramentas que automatizam o processo. O documento original do GitFlow pode ser acessado em: `https://nvie.com/posts/a-successful-git-branching-model/`

Adotar o GitFlow em projetos colaborativos aumenta a produtividade, reduz conflitos e torna o ciclo de desenvolvimento mais previsível.

> **Nota:** A leitura do documento oficial do GitFlow é fortemente recomendada. Uma boa compreensão dessa metodologia colaborativa esclarece o funcionamento de um repositório Git bem estruturado e contribui para o sucesso de projetos versionados em equipe.

9 Exercícios Práticos

Esta seção apresenta exercícios para consolidar os conceitos abordados ao longo da apostila. Os exercícios estão divididos em níveis de dificuldade e incluem um mini-projeto colaborativo, permitindo que você pratique desde os comandos básicos até situações reais de trabalho em equipe com Git.

Para todos os exercícios dessa seção, o ambiente integrado de desenvolvimento Visual Studio Code será utilizado.

9.1 Nível Iniciante

9.1.1 Criar um Repositório Local

- Crie uma pasta chamada `meu_projeto_git` em seu computador.

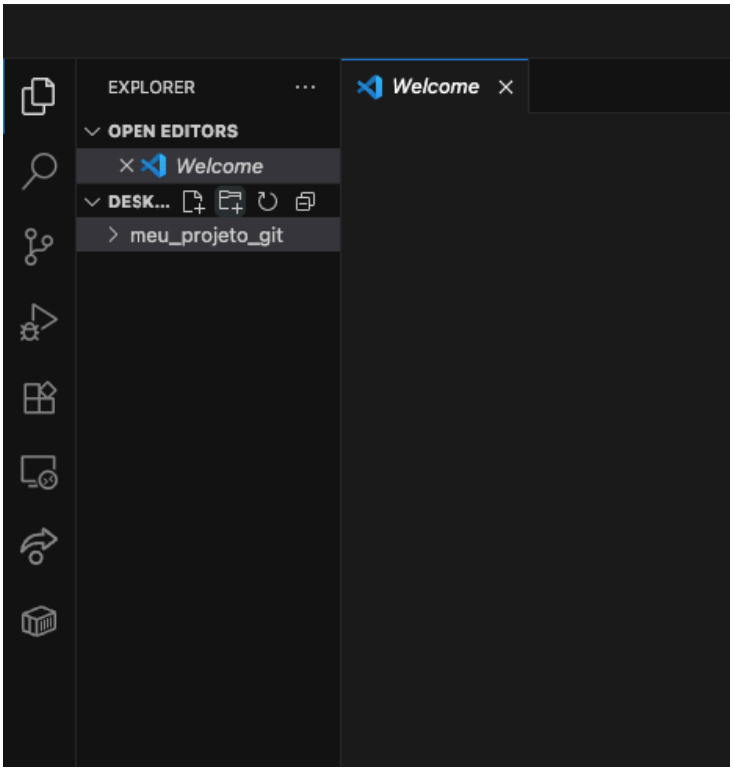

Figura 12 – Solução do exercício 01: Criando a pasta meu_projeto_git no VS Code.

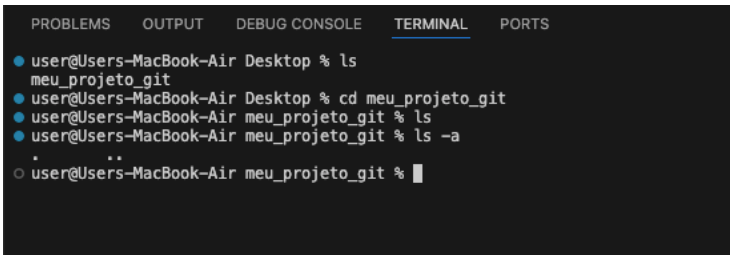

Figura 13 – Solução do exercício 01: Utilizando o terminal do VS Code e alguns comandos para entrar na pasta meu_projeto_git e verificar os arquivos da pasta.

- Inicialize um repositório Git na pasta:

```
git init
```

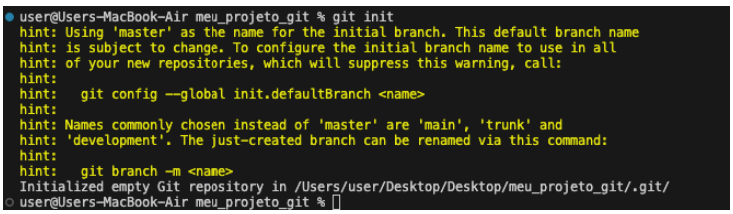

Figura 14 – Solução do exercício 01: Inicializando o repositório git.

- Verifique o status do repositório:

```
git status
```

Figura 15 – Solução do exercício 01: Verificando o status do repositório.

9.1.2 Rastrear Arquivos e Fazer Commits

- Crie um arquivo chamado `README.md` e escreva uma breve descrição do projeto.

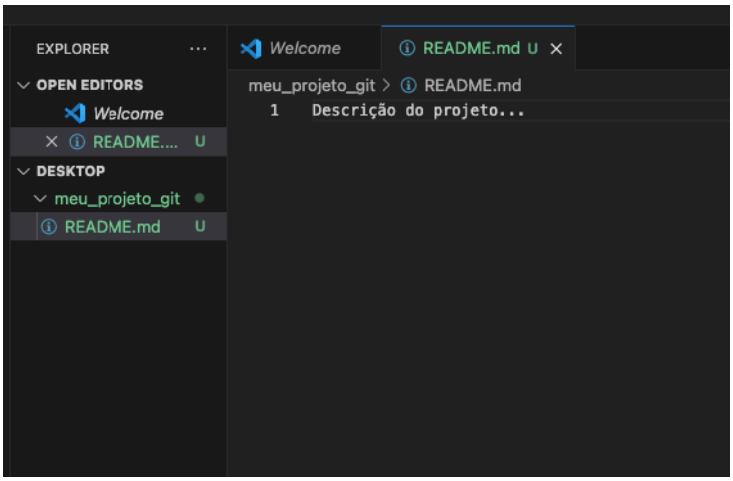

Figura 16 – Solução do exercício 02: Criando o arquivo README.md.

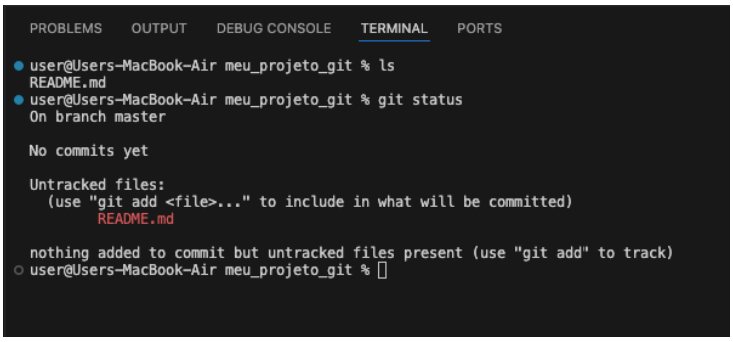

Figura 17 – Solução do exercício 02: Git status retorna que README.md é um novo arquivo que precisa ser adicionado ao repositório git.

- Adicione o arquivo ao staging area:

```
git add README.md
```

- Faça um commit com uma mensagem clara:

```
git commit -m "Adiciona README inicial"
```

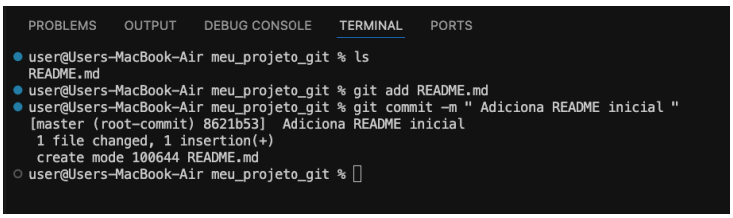

Figura 18 – Solução do exercício 02: Adicionando README.md e fazendo o commit.

- Crie mais um arquivo (ex: `codigo.m`) e repita o processo:

```
git add codigo.m
git commit -m "Adiciona código MATLAB inicial"
```

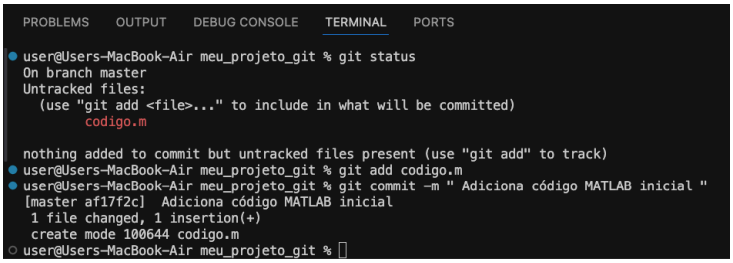

Figura 19 – Solução do exercício 02: Adicionando codigo.m e fazendo o commit.

9.1.3 Enviar para um Repositório Remoto

- Crie um repositório remoto no GitHub ou GitLab.

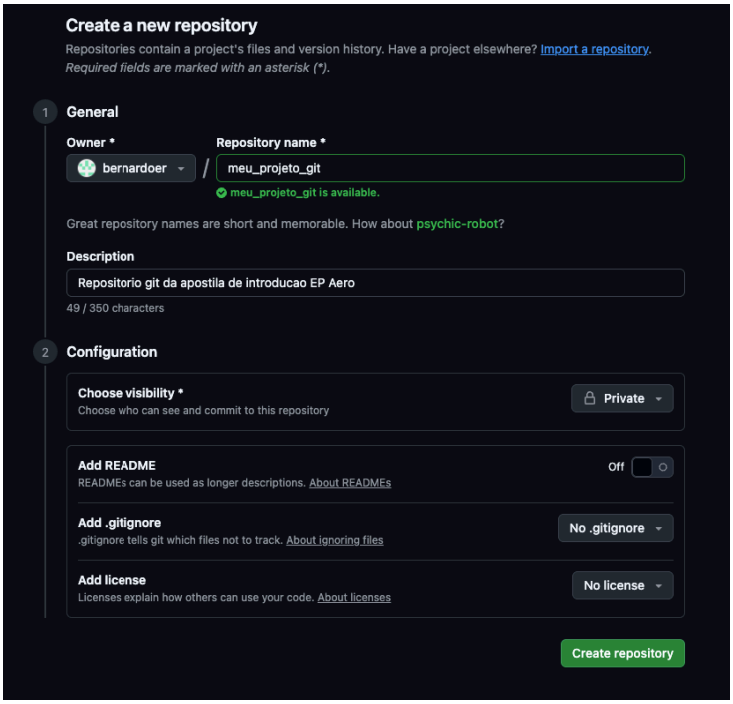

Figura 20 – Solução do exercício 03: Criando o repositório no GitHub.

- Conecte o repositório local ao remoto:

```
git remote add origin <URL-do-repositório>
```

- Envie seus commits para o remoto:

```
git push origin main
```

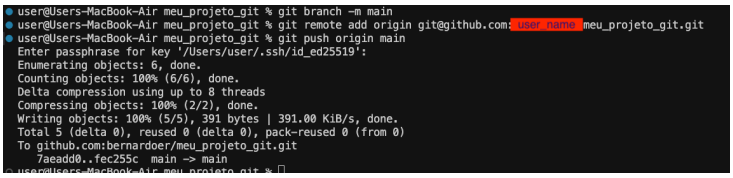

Figura 21 – Solução do exercício 03: Adicionando a branch master no repositório GitHub.

9.2 Nível Intermediário

9.2.1 Criar e Mesclar Branches

- Crie um branch chamado `feature/descricao`:

```
git checkout -b feature/descricao
```

- No novo branch, edite o `README.md` adicionando mais detalhes sobre o projeto.
- Adicione e faça commit das alterações:

```
git add README.md
git commit -m "Adiciona detalhes ao README"
```

- Retorne ao branch principal:

```
git checkout main
```

- Mescle o branch de funcionalidade:

```
git merge feature/descricao
```

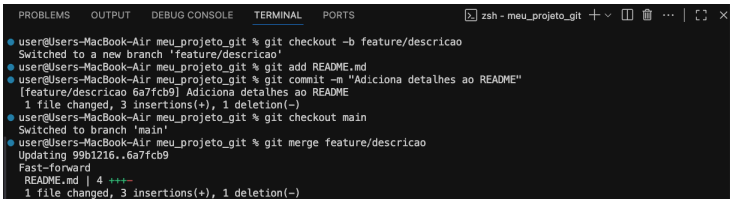

Figura 22 – Solução do exercício 04: Criando feature branch para editar o projeto.

9.2.2 Resolver Conflitos de Merge

- Simule um conflito editando a mesma linha do `README.md` em dois branches diferentes.
- Tente fazer o merge e, ao encontrar um conflito, edite o arquivo manualmente para resolver.
- Após resolver, adicione e faça commit:

```
git add README.md
git commit -m "Resolve conflito de merge no README"
```

9.2.3 Utilizar Efetivamente o .gitignore

- Crie arquivos temporários (`.log`, `.tmp`) na pasta do projeto.
- Crie um arquivo `.gitignore` e adicione regras para ignorar esses arquivos.
- Verifique que os arquivos ignorados não aparecem no status:

```
git status
```

9.3 Mini-Projeto Prático: Gerenciando um Pequeno Projeto de Equipe

Monte uma equipe de 2 a 4 pessoas e siga os passos abaixo:

1. Crie um repositório remoto compartilhado no GitHub ou GitLab.
2. Cada integrante deve clonar o repositório:

```
git clone <URL-do-repositório>
```

3. Divida as tarefas: cada pessoa deve criar um branch para desenvolver uma funcionalidade:

```
git checkout -b feature/modelo
```

4. Realize commits frequentes e mensagens claras:

```
git add <arquivo>
git commit -m "Mensagem clara sobre a alteração"
```

5. Abra Pull Requests na plataforma para integrar as funcionalidades ao branch principal.
6. Realize revisões de código e resolva possíveis conflitos de merge.
7. Ao final, garanta que o projeto esteja organizado, com README atualizado e arquivos desnecessários ignorados pelo `.gitignore`.

Esses exercícios proporcionam experiência prática com os principais comandos e fluxos de trabalho do Git, preparando você para atuar em projetos reais de engenharia

10 Recursos e Referências

Para aprofundar seus conhecimentos em Git e trabalho colaborativo, é importante consultar materiais de referência e recursos oficiais. Abaixo estão algumas recomendações úteis para estudo e consulta durante o desenvolvimento de projetos.

10.1 Documentação Oficial do Git

A documentação oficial do Git é completa e detalhada, abordando desde conceitos básicos até comandos avançados. Recomenda-se a leitura para esclarecer dúvidas e explorar funcionalidades adicionais.

- Site oficial: `https://git-scm.com/doc`
- Livro gratuito: `https://git-scm.com/book/en/v2`

10.2 GitHub Learning Lab

O **GitHub Learning Lab** oferece cursos interativos gratuitos sobre Git, GitHub e colaboração em projetos. Os exercícios são práticos e guiados, ideais para iniciantes e para quem deseja aprimorar suas habilidades.

- Acesse:

10.3 Cheat Sheets Recomendados

Cheat sheets são resumos práticos dos principais comandos e fluxos de trabalho do Git, úteis para consulta rápida durante o desenvolvimento.

- Git Cheat Sheet oficial: `https://education.github.com/git-cheat-sheet-education.pdf`
- GitHub Git Cheat Sheet: `https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf`
- Atlassian Git Cheat Sheet: `https://www.atlassian.com/git/tutorials/atlassian-git-ch`

Esses recursos complementam o conteúdo da apostila e ajudam a resolver dúvidas, explorar novas funcionalidades e aprimorar o uso do Git em projetos acadêmicos