Capítulo 5

Introdução ao Git e GitHub: Controle de Versão na Prática

Deyvison Samuel Gomes do Nascimento, Maria Vitória da Silva Araújo, Maria Yasmin Oliveira Mélo, M.e Maykol Lívio Sampaio Vieira Santos

Resumo

O minicurso "Introdução ao Git e GitHub"tem como objetivo apresentar, de forma acessível e prática, os fundamentos do controle de versão por meio das ferramentas Git, GitHub e GitHub Desktop. Destinado a estudantes iniciantes na área de tecnologia, o curso visa capacitar os participantes a utilizarem essas ferramentas essenciais tanto no contexto acadêmico quanto no mercado de trabalho. Com carga horária total de três horas, o conteúdo abordará desde a criação de uma conta no GitHub, a utilização dos principais comandos e fluxos de trabalho, além do envio e atualização de projetos em repositórios remotos. A metodologia adotada combina exposições teóricas com atividades práticas em laboratório, permitindo que os participantes acompanhem passo a passo o funcionamento das ferramentas e desenvolvam autonomia na utilização do versionamento de código. Ao final, espera-se que os alunos estejam aptos a criar e gerenciar seus próprios repositórios, colaborar em projetos em equipe e compreender a lógica do controle de versões distribuído, consolidando uma base sólida para práticas modernas de desenvolvimento.

Palavras-chave: *Git, GitHub, Versionamento de Código, GitHub Desktop*

Abstract

The short course "Introduction to Git and GitHub"aims to present, in an accessible and practical way, the fundamentals of version control through the tools Git, GitHub, and GitHub Desktop. Designed for beginner students in the field of technology, the course seeks to enable participants to use these essential tools both in academic settings and in

Deyvison Samuel Gomes do Nascimento (apresentador) é estudante do curso de Tecnologia em Análise e Desenvolvimento de Sistemas pelo Instituto Federal do Piauí (IFPI) campus Piripiri. 1
Maria Vitória da Silva Araújo é estudante do curso de Tecnologia em Análise e Desenvolvimento de Sistemas pelo IFPI campus Piripiri.
Maria Yasmin Oliveira Mélo (apresentadora) é estudante do curso de Tecnologia em Análise e Desenvolvimento de Sistemas pelo IFPI campus Piripiri.
Maykol Lívio Sampaio Vieira Santos (orientador) é professor de Informática no IFPI campus Piripiri e Mestre em Tecnologia e Gestão em EAD pela UFRPE (UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO).

the job market. With a total duration of three hours, the content will cover everything from creating a GitHub account, using the main commands and workflows, to sending and updating projects in remote repositories. The methodology combines theoretical explanations with hands-on lab activities, allowing participants to follow the step-by-step operation of the tools and develop autonomy in using code versioning. By the end of the course, students are expected to be able to create and manage their own repositories, collaborate on team projects, and understand the logic of distributed version control, establishing a solid foundation for modern development practices.
Keywords: *Git, GitHub, Code Versioning, GitHub Desktop*

5.1. Introdução

O Git surgiu em 2005, criado por Linus Torvalds após a ruptura com o BitKeeper como um sistema de controle de versão distribuído, rápido, seguro e adequado a fluxos de trabalho colaborativos. Sua eficiência, confiabilidade e suporte a branches tornaram-no padrão mundial no desenvolvimento de software. Já o GitHub, lançado em 2008, expandiu o uso do Git ao oferecer uma interface web intuitiva e recursos sociais, como *forks*, *pull requests* e *issues*, transformando-se em um espaço central de hospedagem e colaboração de código. Com a aquisição pela Microsoft em 2018, a plataforma ganhou ainda mais infraestrutura e integração com serviços em nuvem, consolidando-se como a maior comunidade de desenvolvedores do mundo. Hoje, Git e GitHub são essenciais para a organização, segurança e produtividade no desenvolvimento de software, além de impulsionarem práticas modernas de colaboração, DevOps e aprendizado em programação [Git-SCM, 2025; Microsoft, 2018].

5.1.1. História do Git

Antes do Git, alguns sistemas já buscavam controlar versões de código, como o *Source Code Control System* (SCCS), criado em 1972, que registrava alterações de forma sequencial, mas tinha limitações e funcionava apenas em Unix. Mais tarde, em 1986, o *Concurrent Versions System (CVS)*, trouxe o conceito de repositório compartilhado e permitiu colaboraço entre vários desenvolvedores, embora apresentasse falhas em operaçes de merge e dificuldades no gerenciamento de arquivos [Rochkind, 1975; Spinellis, 2005; Grune, 1986; Free Software Foundation, 1993].

Apesar dos avanços, esses sistemas eram centralizados, ou seja, dependiam de um servidor único para armazenar o código. Esse modelo gerava problemas como a dependência total do servidor — que, em caso de falha, interrompia o trabalho — e a limitação na colaboração em larga escala, mostrando a necessidade de soluções mais flexíveis e robustas [Loeliger & McCullough, 2012; Tech-in-Japan, 2021].

5.1.2. Linux e BitKeeper

Nos anos 1990, o Linux Kernel já era um dos maiores projetos colaborativos de software, mas até 2002 as contribuições eram integradas manualmente, com *patches* enviados por e-

mail, em um processo lento e sujeito a erros. Para resolver essas limitações, a comunidade passou a utilizar o BitKeeper, um sistema distribuído que permitia a cada desenvolvedor manter uma cópia completa do repositório, facilitando o trabalho offline e a integração em larga escala. Essa adoção representou um grande avanço em relação a modelos centralizados como o Subversion (SVN) [Pro Git — git-scm.com].

No entanto, o BitKeeper era proprietário e, em 2005, o acordo que permitia seu uso gratuito pela comunidade do Linux Kernel foi encerrado. Isso gerou um impasse, já que depender de uma tecnologia fechada colocava em risco a continuidade do projeto. Foi nesse cenário que Linus Torvalds decidiu criar uma nova ferramenta, livre e distribuída, capaz de atender às necessidades do Linux Kernel: o Git [Pro Git, s.d.; LWN.net, s.d.].

5.1.3. Criação do Git

Quando perdeu o acesso ao BitKeeper, Linus Torvalds decidiu criar seu próprio sistema de controle de versão distribuído. Para isso, estabeleceu alguns requisitos fundamentais que orientariam o desenvolvimento da nova ferramenta: precisava ser rápido, superando os sistemas existentes; deveria ser distribuído, garantindo que cada desenvolvedor tivesse uma cópia completa do repositório em sua máquina; tinha que ser seguro, assegurando a integridade dos dados; e, por fim, precisava oferecer bom suporte a fluxos de trabalho não lineares, lidando de forma eficiente com branches e merges [i-Programmer].

Pouco tempo depois, Linus transferiu a manutenção do projeto para Junio C. Hamano, que rapidamente se destacou na comunidade e até hoje atua como mantenedor principal do Git, liderando seu desenvolvimento contínuo e garantindo sua evolução ao longo dos anos [Pro Git, s.d.].

5.1.4. Evolução

Após sua criação em 2005, o Git evoluiu rapidamente, recebendo melhorias constantes e conquistando uma comunidade cada vez maior. Em 2007, ele começou a se popularizar fora do desenvolvimento do kernel Linux, chamando a atenção de outros projetos de software livre. No ano seguinte, em 2008, surgiu o GitHub, uma plataforma que revolucionou a forma de usar o Git ao oferecer uma interface web simples e recursos como *issues*, *pull requests* e colaboração social. Essa combinação foi decisiva para a popularização do Git em escala global [Pro Git, s.d.; GitHub, 2025].

A partir de 2010, grandes empresas de tecnologia, como Google, Microsoft, Facebook e Twitter, passaram a adotar o Git como ferramenta padrão em seus fluxos de desenvolvimento, o que consolidou sua posição como principal sistema de controle de versão distribuído. Em 2018, a Microsoft adquiriu o GitHub, fortalecendo ainda mais o ecossistema em torno do Git e demonstrando a importância estratégica dessa ferramenta para o desenvolvimento de software moderno [Microsoft, 2018; Wired, 2018].

5.1.5. Por que o Git se tornou o padrão?

O Git se consolidou como padrão no desenvolvimento de software por adotar um modelo distribuído, no qual cada cópia do repositório funciona como um *backup* completo e independente, permitindo trabalho mesmo sem conexão a um servidor central. Sua performance também é um destaque, pois operações locais como *commit, branch e merge* são executadas de forma muito rápida em comparação com sistemas anteriores [i-Programmer; Pro Git — git-scm.com].

Outro diferencial é o suporte a *branches* leves, que facilita fluxos paralelos e colaborativos, além da forte ênfase em segurança, com uso de algoritmos de hash como SHA-1 e SHA-256 para proteger a integridade do histórico. O crescimento de plataformas como GitHub, GitLab e Bitbucket criou um ecossistema robusto de colaboração, o que consolidou o Git como o sistema de controle de versão mais utilizado no mundo [GitLab, 2025; Git, 2025; Rewind, 2024].

5.2. História do GitHub

Em 2008, surgiu o GitHub, fundado por Tom Preston-Werner, Chris Wanstrath, PJ Hyett e, posteriormente, Scott Chacon, que entrou para contribuir com a documentação. O objetivo inicial da plataforma era combinar o poder do Git com funcionalidades sociais, criando um espaço que facilitasse o processo de hospedar, revisar e colaborar em projetos, tanto de código aberto quanto privados [Founding; PSL Models; Medium; WIRED].

O grande diferencial do GitHub estava em sua interface web amigável, que permitia navegar pelos repositórios de forma simples e intuitiva. Além disso, introduziu recursos que se tornaram padrão na colaboração de software, como *forks, pull requests e issues*, transformando o fluxo de trabalho dos desenvolvedores. Outro aspecto inovador foi a criação de perfis e métricas para programadores, funcionando como uma espécie de "rede social para desenvolvedores", o que estimulou ainda mais a colaboração e a visibilidade dentro da comunidade de software [Timeline; WIRED].

5.2.1. Crescimento inicial

A partir de 2010, o GitHub deixou de ser apenas um espaço para projetos de código aberto e passou a atrair empresas e grandes corporações, que enxergaram na plataforma uma forma eficiente de organizar e gerenciar fluxos de desenvolvimento. Nesse contexto, a plataforma evoluiu para hospedar não apenas repositórios públicos, mas também projetos privados, o que ampliou significativamente seu alcance no mercado corporativo [Encyclopedia Britannica; GitHub, 2014].

O crescimento foi exponencial, com milhões de desenvolvedores migrando para o GitHub, que se tornou o padrão de facto em hospedagem e colaboração em projetos de software. O modelo baseado em *forks, pull requests e issues* consolidou-se como referência global, influenciando inclusive plataformas concorrentes. Esse movimento marcou a transição do GitHub de uma ferramenta voltada majoritariamente para a comuni-

dade de código aberto para uma infraestrutura essencial no desenvolvimento de software em escala global, utilizada tanto por programadores independentes quanto por grandes empresas de tecnologia [Wired, 2012; Wired, 2013].

Em 2018, o GitHub foi adquirido pela Microsoft por US$ 7,5 bilhões, em uma das maiores negociações do setor de tecnologia daquele período. A compra gerou desconfiança inicial na comunidade de código aberto, que historicamente via a Microsoft com certo receio [Microsoft, 2018; TechCrunch, 2018]. Sob a gestão da Microsoft, a plataforma recebeu novos investimentos, expandiu sua infraestrutura e fortaleceu a integração com serviços como o Azure e outras ferramentas do ecossistema Microsoft. Longe de perder relevância, o GitHub continuou crescendo e consolidou-se ainda mais como o principal espaço de colaboração em software no mundo, reunindo milhões de desenvolvedores e empresas em torno do desenvolvimento aberto e compartilhado [TechCrunch, 2018; Wired, 2018].

5.2.2. Status atual e impacto

Atualmente, o GitHub é a maior plataforma de hospedagem de código do mundo, reunindo mais de 100 milhões de repositórios e cerca de 90 milhões de desenvolvedores registrados. A plataforma atende tanto projetos de código aberto, que impulsionam a inovação coletiva, quanto empresas privadas, que utilizam seus recursos para gerenciar fluxos de desenvolvimento em larga escala [Encyclopedia Britannica].

O GitHub tornou-se peça central na cultura DevOps e nos processos de Integração Contínua e Entrega/Deploy Contínuo(CI/CD), permitindo integração contínua, automação e colaboração global de software. Além disso, consolidou-se também como um hub educacional, oferecendo iniciativas como o GitHub *Student Developer Pack*, cursos e ações voltadas ao incentivo do aprendizado de programação, contribuindo para a formação de novas gerações de desenvolvedores [GitHub, 2025; GitHub, 2025].

5.3. Importância do Git/GitHub hoje

O Git é uma das ferramentas mais importantes no desenvolvimento de software atualmente, pois funciona como um sistema de controle de versão distribuído. Ele registra todo o histórico de modificações feitas em um projeto, permitindo que desenvolvedores acompanhem cada alteração no código, retornem a versões anteriores quando necessário e entendam como o sistema evoluiu ao longo do tempo. Esse recurso evita perdas de trabalho e garante maior segurança no processo de desenvolvimento, além de permitir que cada programador mantenha em sua máquina uma cópia completa do repositório com todo o histórico do projeto [Pro Git Book – Chacon & Straub, Apress, 2014; Atlassian – What is Git?].

Outro ponto central é a colaboração. O Git facilita o trabalho em equipe, permitindo que vários desenvolvedores atuem simultaneamente em um mesmo projeto sem que suas mudanças interfiram umas nas outras. Essa característica torna o fluxo de trabalho

mais ágil, estruturado e produtivo. Em resumo, o Git não é apenas uma ferramenta para salvar versões de código, mas um verdadeiro pilar do desenvolvimento moderno, garantindo organização, segurança e eficiência no ciclo de criação de software [HostRagons, 2025; Pro Git Book – Chacon & Straub, Apress, 2014].

O GitHub é uma das plataformas mais relevantes do ecossistema tecnológico, funcionando como um espaço central para hospedagem, colaboração e compartilhamento de código. Baseado no Git, ele amplia suas funcionalidades ao oferecer uma interface prática na nuvem e ferramentas que apoiam tanto projetos individuais quanto grandes iniciativas globais. Um dos seus principais diferenciais é o incentivo à colaboração: milhões de desenvolvedores e organizações utilizam recursos como *pull requests, issues e discussions* para propor melhorias, revisar alterações e resolver problemas em conjunto. Esse ambiente participativo transformou o GitHub em um ponto de encontro para projetos de impacto mundial, como o Linux, o React e o Visual Studio Code [GitHub Docs – About GitHub; Coursera – What is GitHub?].

Além disso, a plataforma integra práticas modernas de DevOps e CI/CD, permitindo configurar fluxos automáticos de testes, validação e implantação, o que aumenta a eficiência e a qualidade das entregas. O GitHub também tem grande importância na educação, com iniciativas como o *Student Developer Pack*, que oferece ferramentas profissionais gratuitas e incentiva estudantes a aprenderem de forma prática como funciona o desenvolvimento colaborativo. Assim, o GitHub consolidou-se como mais que um repositório de código: é um ecossistema essencial para colaboração, inovação e aprendizado em escala global [DEV Community – How GitHub Improves Security and CI/CD Workflows, 2024; GitHub, 2025].

5.4. Conceitos Iniciais

Os conceitos básicos de Git e GitHub giram em torno do controle de versão e da colaboração em projetos de software. O Git é um sistema que registra todas as alterações feitas em arquivos, permitindo acompanhar o histórico, restaurar versões anteriores e trabalhar com ramificações para testar novas ideias sem comprometer a versão principal. Já o GitHub é uma plataforma online que utiliza o Git como base, mas adiciona ferramentas de colaboração, como revisão de código, gerenciamento de tarefas e integração com automações. No uso prático, o fluxo básico envolve criar ou clonar um repositório, registrar mudanças com *commits*, enviar e receber atualizações de um repositório remoto e gerenciar branches para desenvolvimento paralelo. Assim, Git e GitHub juntos oferecem organização, segurança e eficiência no desenvolvimento individual ou em equipe[Chacon & Straub, 2014; GitHub Docs, 2025; GeeksforGeeks, 2023].

5.4.1. O que é controle de versão?

O controle de versão é um sistema que registra e gerencia todas as alterações feitas em arquivos de um projeto ao longo do tempo, permitindo retornar a versões anteriores,

acompanhar a evolução e desfazer erros. Além de organização, ele facilita o trabalho em equipe, já que vários desenvolvedores podem atuar de forma simultânea sem sobrescrever o trabalho uns dos outros. Com o uso de ramificações (*branches*), cada membro pode testar soluções em separado e depois integrá-las de forma segura ao projeto principal [Earth Data Science – Version Control Introduction; Atlassian – Git branching explained].

O GitHub, por sua vez, leva os benefícios do Git para a nuvem, oferecendo hospedagem de repositórios e ampliando as possibilidades de colaboração. Além de compartilhar código, a plataforma fornece recursos como *pull requests*, gerenciamento de tarefas, permissões de acesso, integração com ferramentas de automação e suporte a projetos de qualquer escala. Dessa forma, Git e GitHub tornaram-se pilares do desenvolvimento moderno, garantindo organização, eficiência e qualidade em equipes de diferentes tamanhos [GitHub Docs – About GitHub; Everhour Blog – Why GitHub is important in modern development (2025)].

5.4.2. Diferença entre Git e GitHub

O Git é um sistema de controle de versão distribuído que funciona localmente, permitindo registrar todas as alterações de um projeto, acompanhar o histórico completo de versões, restaurar estados anteriores e criar ramificações (*branches*) sem comprometer a versão principal. Cada alteração inclui informações sobre autor, data e modificações realizadas, garantindo rastreabilidade e organização do projeto. Por ser distribuído, cada colaborador possui uma cópia completa do repositório, podendo trabalhar offline e sincronizar alterações apenas quando necessário [Stack Overflow – Git is a revision control system; GeeksforGeeks – Differences Between Git and GitHub].

O GitHub é uma plataforma online que hospeda repositórios Git na nuvem, oferecendo armazenamento centralizado e recursos de colaboração, como pull requests, issues, controle de permissões e integração com automação via *GitHub Actions*. Ele reúne uma comunidade global de desenvolvedores, permitindo que equipes de qualquer tamanho trabalhem de forma organizada e colaborativa. Apesar de depender de conexão à internet e de uma conta na plataforma, o GitHub não substitui o Git, que continua responsável pelo controle de versões local e distribuído, podendo ser usado também com outras plataformas como GitLab, Bitbucket ou servidores privados [GitHub Docs – How do Git and GitHub work together?; DataCamp – Git vs GitHub: Key differences explained; HubSpot – Git vs GitHub].

5.5. Desenvolvimento

Quanto ao desenvolvimento, este será realizado de forma isolada em alguns momentos. Inicialmente, no item 1.5.1, serão apresentados os conceitos básicos da instalação da ferramenta GitHub Desktop. Em seguida, no item 1.5.2, será introduzida a criação e a sincronização de repositórios na mesma ferramenta, além da inclusão de outros elementos, como o *.gitignore* e o *README.md*. Posteriormente, haverá ainda uma subseção,

denominada 1.5.3, na qual será apresentada a colaboração por meio de *branches* com o GitHub Desktop [GitHub 2025].

5.5.1. Instalação e Configuração inicial do GitHub Desktop

Quanto à instalação da aplicação GitHub Desktop, esta pode ser realizada atualmente no macOS 11.0 ou posterior e no Windows 10 (64 bits) ou versão posterior [GitHub 2025]. Para efetuar a instalação da aplicação, o usuário deve seguir os seguintes procedimentos:

1. Acesse a página de *download* do GitHub Desktop).
2. Clique em Baixar para Windows, ou baixar para MacOS.

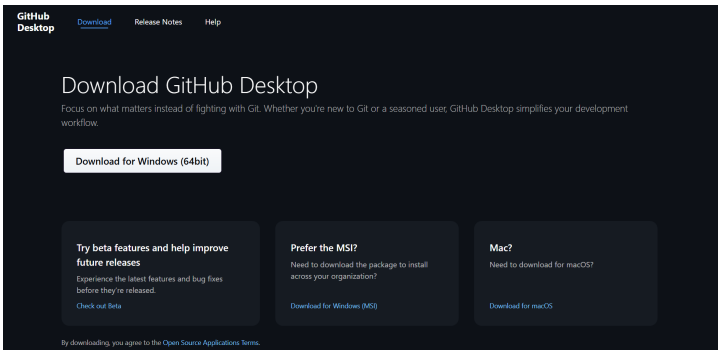

Figura 5.1. Página de descarregamento da ferramenta GitHub Desktop. Fonte: Próprio Autor.

3. Na pasta *Downloads* do computador, o usuário deve clicar duas vezes no arquivo de instalação do GitHub Desktop.

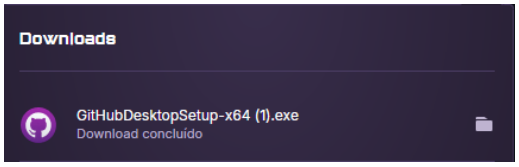

Figura 5.2. Exemplo de criação de *commit* no GitHub Desktop com mensagem clara e objetiva. Fonte: Próprio Autor.

4. GitHub Desktop será executado após a instalação ser concluída.

Após a instalação da aplicação em sua máquina, será necessária a criação de uma conta no GitHub ou no GitHub Enterprise, a fim de que o usuário possa trocar dados entre seus repositórios locais e remotos [GitHub 2025]. Para que o usuário comum se inscreva em uma conta pessoal, devem ser seguidos os seguintes passos:

1. Navegue até a página https://github.com/.

2. Clique em Cadastra-se no GitHub.

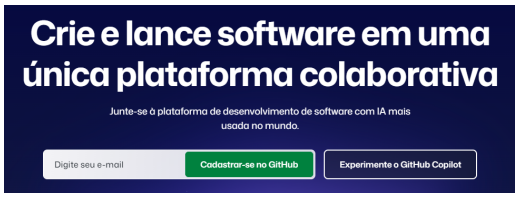

Figura 5.3. Exemplo de criação de commit no GitHub Desktop com mensagem clara e objetiva. Fonte: Próprio Autor.

3. Preencha as informações de cadastro da página, ou como alternativa, clique em *Continue with Google* para se inscrever usando uma conta do Google.

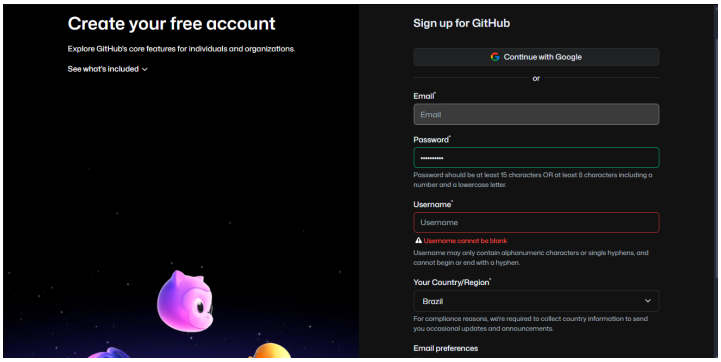

Figura 5.4. Exemplo de criação de commit no GitHub Desktop com mensagem clara e objetiva. Fonte: Próprio Autor.

4. Continue seguindo os prompts indicados pela plataforma para finalizar a criação de sua conta pessoal. Vale ressaltar que durante a inscrição será solicitado ao usuário a verificação de e-mail para fins de segurança.

Com relação à interface da aplicação proposta, que será devidamente apresentada no decorrer do texto, temos a seguinte captura de tela ilustrando a tela inicial dessa ferramenta. Ao visualizá-la, nota-se um evidente minimalismo, com uma interface bastante enxuta, além da ausência de tradução para o português brasileiro, onde toda a interface é exibida em língua inglesa.

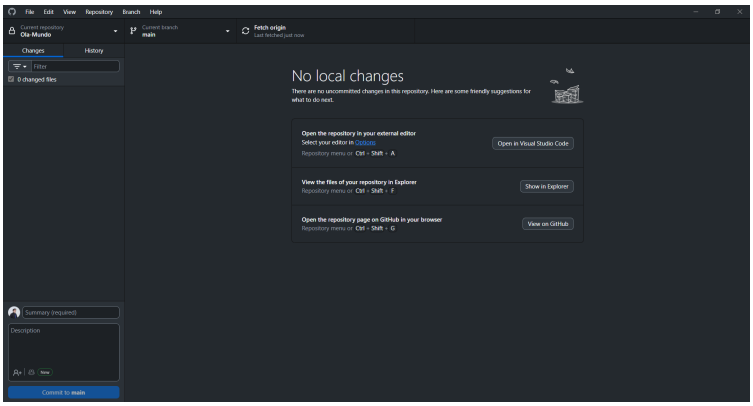

Figura 5.5. Tela inicial do GitHub Desktop. Fonte: Próprio autor.

5.5.2. Criando e Sincronizando Repositórios com GitHub Desktop

Embora o Git possa ser utilizado por meio da linha de comando, muitas equipes e iniciantes preferem ferramentas visuais que simplifiquem sua utilização. O GitHub Desktop é uma dessas soluções, oferecendo uma interface gráfica intuitiva para a criação, sincronização e gerenciamento de repositórios, sem abrir mão das funcionalidades essenciais do versionamento.

5.5.2.1. Criação do repositório local

No GitHub Desktop, o usuário pode criar um novo repositório local diretamente pelo menu *File - New Repository*. Nesse momento, são definidos alguns parâmetros essenciais:

1. **Name:** corresponde ao nome do repositório. É recomendável utilizar nomes significativos e descritivos que facilitem a identificação do projeto.

2. **Description:** campo opcional que permite resumir o objetivo do projeto. Essa descrição auxilia colaboradores a compreenderem rapidamente a finalidade do repositório.

3. **Local path:** define o diretório do computador onde o repositório será armazenado. Manter uma estrutura organizada facilita o gerenciamento de múltiplos projetos.

4. **Initialize this repository with a README:** ao marcar essa opção, o repositório é criado já contendo um arquivo `README.md`. Esse documento funciona como a apresentação inicial do projeto, trazendo informações como objetivos, instruções de instalação e exemplos de uso.

5. **Git ignore:** permite escolher um modelo pré-definido de arquivos a serem ignorados pelo Git, evitando que itens desnecessários sejam versionados. Por exemplo, ao selecionar `Node`, pastas como `node_modules/` não serão incluídas no controle de versão.

6. **License:** possibilita definir a licença do projeto, indicando como o código pode ser utilizado por terceiros. A escolha da *MIT License* é comum em projetos de código aberto por permitir ampla reutilização e modificação com poucas restrições.

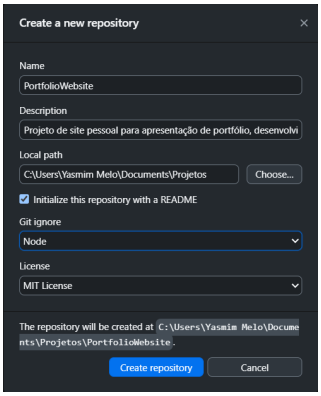

Figura 5.6. Tela criação de um novo repositório. Fonte: Próprio autor.

Após o preenchimento desses campos, basta selecionar **Create repository** para gerar o repositório local já configurado, pronto para receber *commits* e posteriormente ser publicado no GitHub.

5.5.2.2. Adição de arquivos ao repositório

Após a criação do repositório, o GitHub Desktop direciona o usuário para a tela principal do projeto. Nela, inicialmente, não há alterações registradas (*No local changes*), mas o sistema já oferece duas opções fundamentais de interação com o repositório recém-criado:

1. **Abrir o repositório em um editor externo:** O GitHub Desktop possibilita abrir o repositório diretamente em um editor de código, como o Visual Studio Code. Essa funcionalidade agiliza a edição dos arquivos do projeto, dispensando a necessidade de navegar manualmente até a pasta no sistema. O acesso pode ser feito por meio do botão **Open in Visual Studio Code**, disponível na interface principal.

2. **Visualizar os arquivos do repositório no explorador de arquivos:** Caso o usuário deseje acessar a pasta do projeto diretamente no sistema operacional, pode utilizar a

opção **Show in Explorer**. Essa funcionalidade abre a estrutura de diretórios onde o repositório foi criado, possibilitando a inclusão de novos arquivos ou a organização manual do projeto.

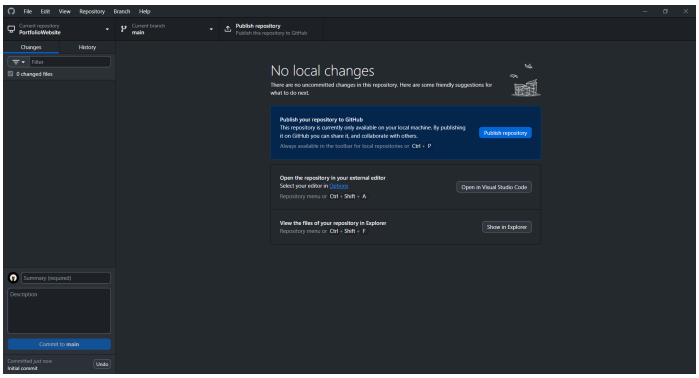

Figura 5.7. Tela inicial após a criação do repositório. Fonte: Próprio autor.

Uma vez que arquivos novos sejam adicionados à pasta do repositório ou que arquivos existentes sejam modificados, o GitHub Desktop detecta automaticamente essas mudanças. O usuário, então, poderá selecionar quais arquivos deseja incluir no próximo *commit*, garantindo que apenas as alterações relevantes sejam registradas no histórico do projeto.

5.5.2.3. Commits com mensagens claras

No GitHub Desktop, cada alteração registrada no repositório precisa ser acompanhada de uma mensagem de *commit*. Esse procedimento é essencial para manter o histórico do projeto organizado e facilitar a colaboração.

1. **Área de mudanças detectadas (Changes):** Sempre que um arquivo é adicionado, modificado ou removido no repositório, o GitHub Desktop exibe essas alterações na aba **Changes**. O usuário pode revisar cada modificação antes de confirmá-la.

2. **Seleção dos arquivos para o commit:** Apenas os arquivos marcados na lista de mudanças serão incluídos no *commit*. Isso permite que o desenvolvedor escolha apenas o que é relevante para registrar no histórico, evitando alterações desnecessárias.

3. **Campo Summary (mensagem obrigatória):** É o título do commit, que deve ser curto e direto. Resume a finalidade da alteração em uma única frase, como por exemplo:

```
feat: adicionar seção de contato
fix: corrigir erro no formulário de login
```

4. **Campo Description (mensagem opcional):** Permite adicionar uma explicação mais detalhada sobre o *commit*. É útil para descrever o contexto da mudança, o motivo da implementação ou observações para outros colaboradores.

5. **Realizar o commit:** Após preencher os campos, o usuário deve clicar no botão **Commit to branch**. A alteração será registrada no histórico local do repositório e ficará disponível para ser enviada ao GitHub posteriormente.

Mensagens claras e objetivas garantem que o histórico do projeto seja compreensível, facilitando futuras consultas, revisões e colaborações entre desenvolvedores.

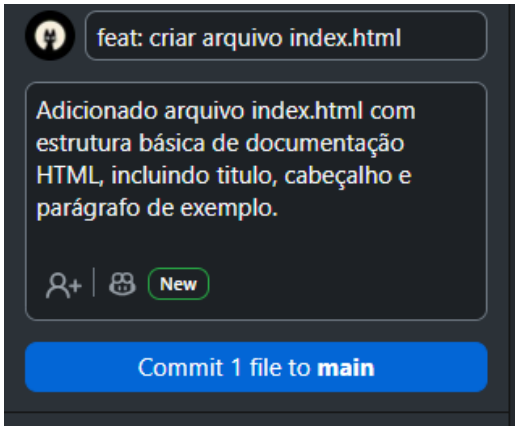

Figura 5.8. Exemplo de criação de commit no GitHub Desktop com mensagem clara e objetiva. Fonte: Próprio Autor.

5.5.3. Sincronização com o GitHub

Após realizar *commits* no repositório local, é necessário garantir que as alterações fiquem disponíveis também no repositório remoto no GitHub. Da mesma forma, ao trabalhar em equipe, é importante manter o repositório local sempre atualizado com as contribuições de outros colaboradores. O GitHub Desktop oferece recursos que facilitam esse processo de envio e recebimento de alterações.

1. **Publicação inicial do repositório:** Clique no botão **Publish repository** para criar automaticamente um repositório remoto no GitHub, vinculado ao seu repositório local.

- Antes de publicar, verifique se já existem *commits* locais, pois, caso contrário, o botão não terá alterações para enviar.
- É possível configurar a visibilidade do repositório como pública (visível a todos) ou privada (restrita).
- Após a publicação, o repositório já estará disponível online e pronto para ser compartilhado.

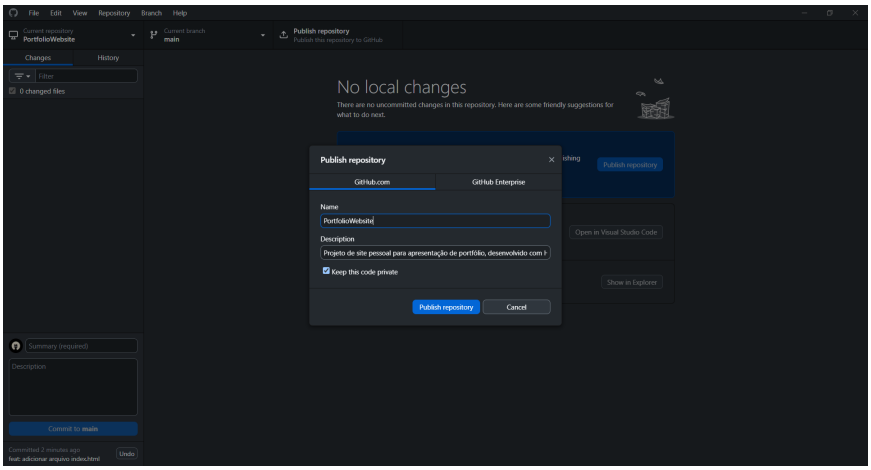

Figura 5.9. Tela para publicação do repositório. Fonte: Próprio autor.

2. **Envio de alterações (Push):** Sempre que novos commits forem criados no repositório local, utilize o botão **Push origin** para enviar essas mudanças ao GitHub, mantendo o repositório remoto atualizado.

3. **Verificação de atualizações (Fetch):** Para checar se há novas alterações feitas por outros colaboradores no repositório remoto, clique em **Fetch origin**.

4. **Baixar alterações remotas (Pull):** Caso sejam encontradas novidades, o botão mudará para **Pull origin**. Clique nele para baixar e aplicar as alterações no repositório local.

5. **Histórico e conflitos:** Após o *pull*, os novos commits aparecerão no histórico do GitHub Desktop. Caso haja conflitos entre alterações locais e remotas, será necessário resolvê-los antes de concluir a sincronização.

5.5.3.1. Clonagem de repositórios existentes

Projetos já hospedados no GitHub podem ser clonados facilmente pelo GitHub Desktop, utilizando a opção **File - Clone Repository**. Esse recurso cria uma cópia completa do

repositório remoto no computador, incluindo todos os arquivos, histórico de *commits e branches*, permitindo que o usuário contribua diretamente no projeto.

1. **Acessar a opção de clonagem:** No menu superior do GitHub Desktop, clique em *File - Clone Repository*. Uma nova janela será exibida para inserir os dados do repositório.

2. **Escolher a origem do repositório:** É possível selecionar um repositório disponível em sua conta do GitHub, em uma organização da qual participa ou inserir manualmente a *Uniform Resource Locator* (URL) de um repositório público.

3. **Definir o diretório local:** Escolha a pasta em seu computador onde deseja armazenar o repositório clonado. Essa será a versão local, totalmente vinculada ao repositório remoto.

4. **Concluir a clonagem:** Após confirmar, o GitHub Desktop fará o download de todos os arquivos e histórico. O projeto será aberto automaticamente na tela principal, pronto para receber *commits*, *pulls* e *pushes*.

5. **Começar a contribuir:** A partir desse momento, o usuário já pode editar arquivos, criar novas *branches* e enviar contribuições, com o GitHub Desktop cuidando da sincronização com o repositório remoto.

5.5.3.2. Inclusão de .gitignore e README.md

O GitHub Desktop permite que arquivos auxiliares, como *.gitignore* e *README.md*, sejam incluídos já no momento da criação do repositório ou adicionados posteriormente. Esses arquivos cumprem funções importantes: o *.gitignore* define quais arquivos e pastas devem ser ignorados pelo versionamento (como *logs*, dependências ou arquivos temporários), enquanto o *README.md* serve como documentação inicial do projeto, apresentando sua finalidade, instruções de uso e informações básicas para colaboradores.

1. **Durante a criação do repositório:** Ao selecionar *File - New Repository*, é possível marcar as opções para gerar automaticamente um arquivo *README.md* e um *.gitignore*. O *.gitignore* pode ser configurado a partir de modelos prontos, de acordo com a linguagem ou tecnologia utilizada no projeto (por exemplo, Node.js, Python, Java).

2. **Adição posterior dos arquivos:** Caso não sejam criados no início, esses arquivos podem ser adicionados manualmente no diretório local do projeto. O GitHub Desktop detectará as novas inclusões, permitindo que sejam versionadas em um *commit*.

3. **Função de cada arquivo:** O *.gitignore* garante que apenas arquivos relevantes sejam controlados pelo Git, evitando poluição do repositório com dados temporários ou específicos do ambiente do desenvolvedor. O *README.md* é exibido automaticamente na página inicial do repositório no GitHub, funcionando como cartão de visita do projeto e facilitando a compreensão por novos colaboradores.

4. **Benefícios:** A inclusão desses arquivos contribui para a organização do projeto, melhora a colaboração entre desenvolvedores e reduz problemas de versionamento.

5.5.3.3. Compreensão do histórico de mudanças

O GitHub Desktop apresenta um histórico visual de todos os *commits* realizados no repositório, permitindo que o usuário acompanhe a evolução do projeto de maneira clara e organizada. Esse recurso mostra, em ordem cronológica, cada alteração registrada, incluindo o autor, a data, a mensagem do *commit* e os arquivos modificados. Dessa forma, torna-se mais fácil realizar auditorias, revisões de código e identificar eventuais regressões que possam ter sido introduzidas.

1. **Acessar a aba de histórico:** Na tela principal do GitHub Desktop, ao lado da lista de alterações pendentes, encontra-se a aba *History*. Ali são exibidos todos os commits já registrados no repositório atual.

2. **Visualizar detalhes de cada commit:** Ao selecionar um *commit* no histórico, a interface mostra os arquivos alterados e, em alguns casos, até o conteúdo exato das modificações realizadas (adições e remoções). Isso facilita a análise de mudanças específicas, sem a necessidade de recorrer à linha de comando.

3. **Identificar a autoria e a data das mudanças:** Cada *commit* apresenta informações sobre quem o realizou e em que momento. Essa funcionalidade é fundamental em projetos colaborativos, pois permite rastrear responsabilidades e compreender o contexto das alterações.

4. **Facilitar auditorias e revisões:** Com a visualização cronológica, é possível auditar o progresso do projeto, revisar decisões tomadas em *commits* passados e até encontrar o ponto em que um problema foi introduzido no código.

5.5.4. Colaboração de Branches com o Github Desktop

Uma *branch* é como uma "linha paralela" do projeto. Ela permite que você trabalhe em algo novo (uma funcionalidade, correção de *bug* ou teste) sem mexer no código principal, que normalmente está na *branch main*. Pense nela como uma cópia do projeto em que você pode fazer alterações à vontade, sem risco de quebrar o que já funciona.

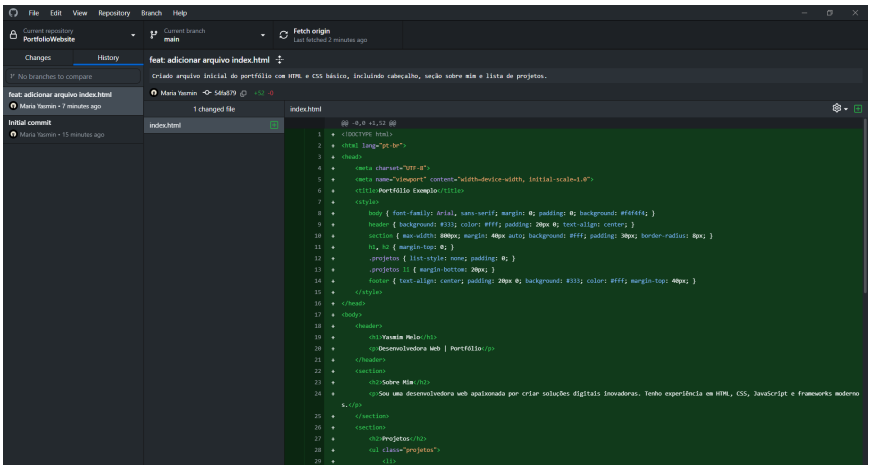

Figura 5.10. Exemplo da tela de históricos. Fonte: Próprio autor.

5.5.4.1. Criando uma branch no GitHub Desktop

1. Abra o GitHub Desktop e selecione o repositório em que quer trabalhar.

2. No topo, clique no botão "*Current Branch*" (ou "Branch atual").

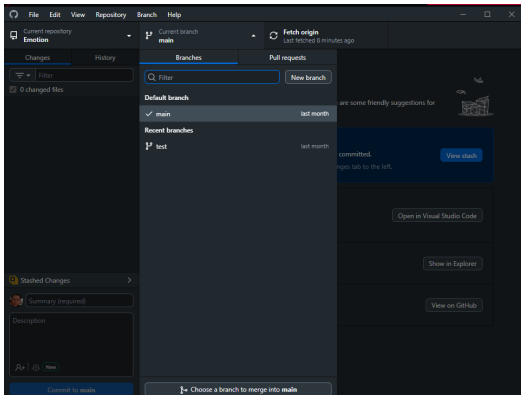

Figura 5.11. Exemplo da tela de Branch atual. Fonte: Próprio autor.

3. Clique em *"New Branch"* (ou "Nova branch").

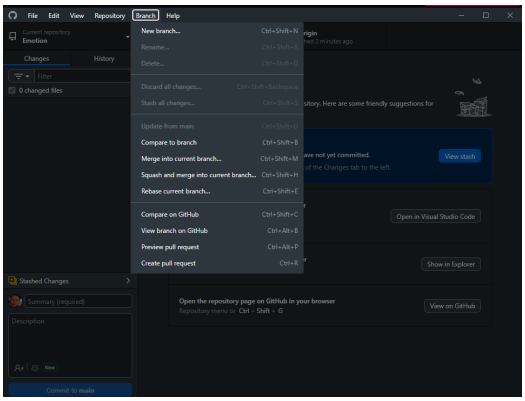

Figura 5.12. Exemplo da tela de nova *Branch*. Fonte: Próprio autor.

4. Dê um nome significativo para a *branch*, como correcao-bug-login ou nova-funcionalidade.

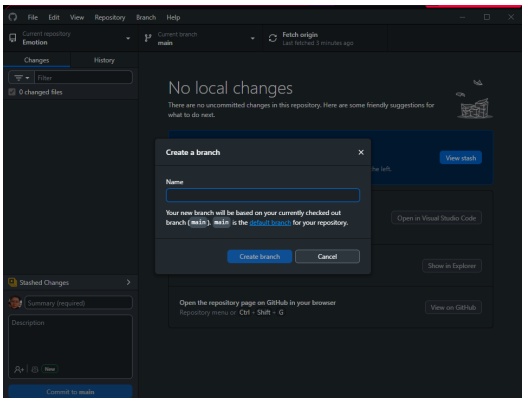

Figura 5.13. Exemplo da tela de colocar nome na *branch*. Fonte: Próprio autor.

5. Clique em *"Create Branch"* (Criar Branch).

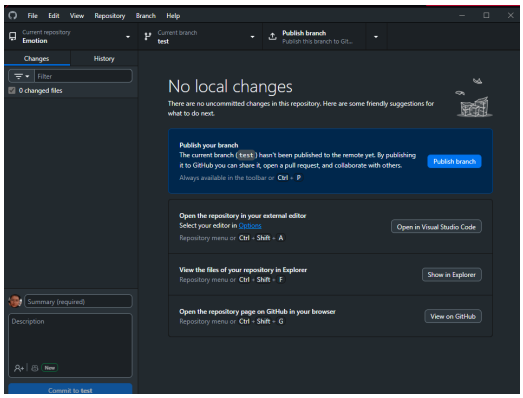

Figura 5.14. Exemplo da tela de *branch* nomeada. Fonte: Próprio autor.

6. O GitHub Desktop troca automaticamente para a nova *branch*. Tudo o que você alterar agora será registrado nela, e não na *branch* principal.

7. Clique em *"Current Branch"*.

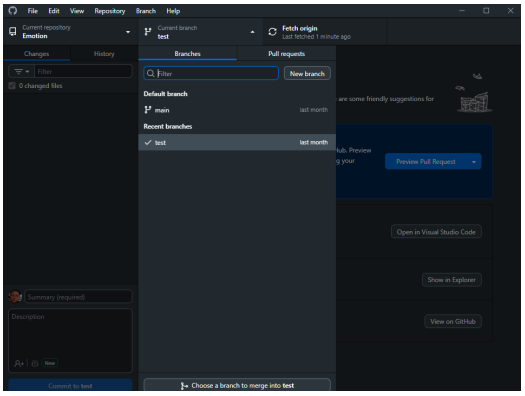

Figura 5.15. Exemplo da tela de lista das atuais *branches*. Fonte: Próprio autor.

8. Você verá a lista de todas as *branches* do repositório.

9. Clique na *branch* que deseja usar e o GitHub Desktop vai automaticamente mudar o foco para ela.

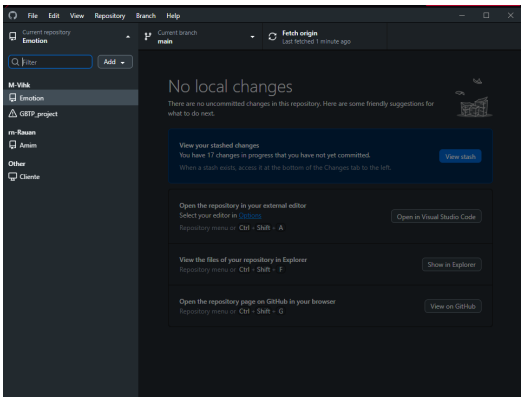

Figura 5.16. Exemplo da tela de seleção de repositório. Fonte: Próprio autor.

10. Agora, qualquer alteração será feita na *branch* selecionada.

5.5.4.2. O que é e como fazer o *merge*?

O *merge* é o processo de unir as alterações feitas em uma *branch* secundária (por exemplo, nova-funcionalidade) de volta para a *branch* principal (*main*). Isso garante que tudo o que você desenvolveu separadamente passe a fazer parte do código principal. Para realizarmos um *merge* no GitHub Desktop, serão desenvolvidos os seguintes passos:

1. No GitHub Desktop, clique em *"Current Branch"* e selecione a *branch* principal (*main*).

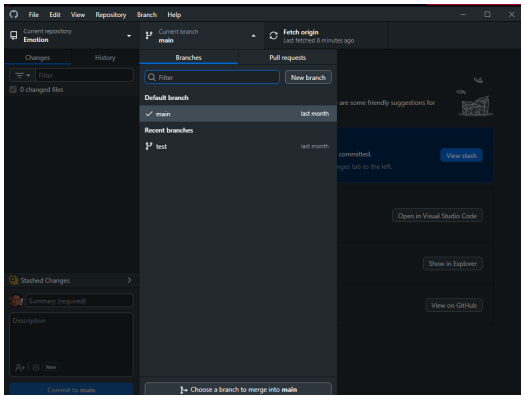

Figura 5.17. Exemplo da tela de *Branch* principal. Fonte: Próprio autor.

2. Vá no menu *Branch* (na parte superior) e clique em "Merge into Current Branch".

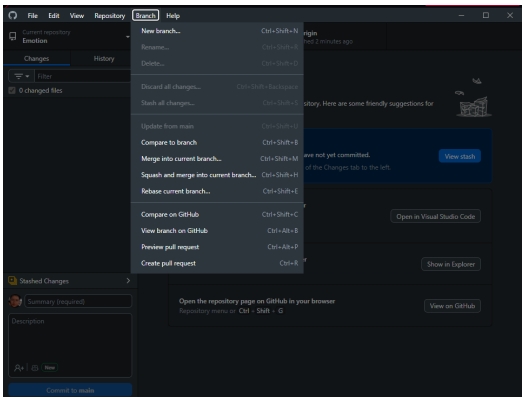

Figura 5.18. Exemplo da tela de *Branch Merge*. Fonte: Próprio autor.

3. Selecione a *branch* que contém as alterações (exemplo: nova-funcionalidade).

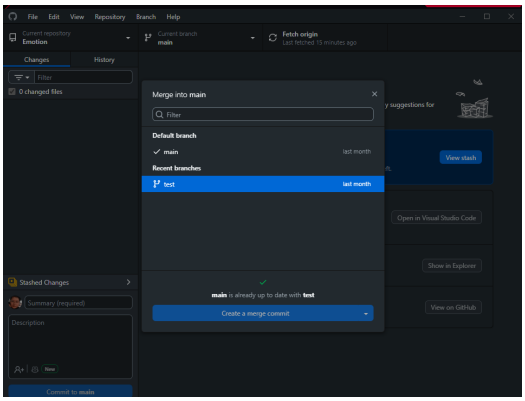

Figura 5.19. Exemplo da tela de *Branch* alterações adicionada a brach principal. Fonte: Próprio autor.

4. O GitHub Desktop vai tentar unir automaticamente as mudanças.

5.5.5. Como resolver conflitos simples e como o GitHub Desktop mostra o conflito

Um conflito de merge acontece quando duas *branches* alteram a mesma linha de um arquivo e o Git não consegue escolher qual versão manter. Nesses casos, o GitHub Desktop mostra uma mensagem de conflito e lista os arquivos afetados para que o usuário resolva manualmente. Já para sincronizar alterações, o processo envolve usar *Fetch origin* para verificar atualizações no repositório remoto, *Push origin* para baixá-las e *Push origin* para enviar seus *commits* locais, garantindo que todos os colaboradores trabalhem na versão mais atualizada [GitHub Docs; Coderefinery, 2025].

Um conflito acontece quando duas *branches* modificam a mesma linha de um arquivo ou mexem em partes que o Git não consegue decidir sozinho qual versão manter. Quando há conflito, o GitHub Desktop exibe uma mensagem como *"This branch has conflicts that must be resolved"*. Ele vai listar os arquivos problemáticos e você precisa abrir esses arquivos para resolver.

5.5.6. Sincronizar alterações com o GitHub

No GitHub Desktop, para manter seu repositório local alinhado com o remoto, você normalmente começa clicando em *"Fetch origin"*: isso verifica se há mudanças no GitHub que ainda não foram baixadas para sua máquina. Se existir algo novo, você pode então usar *"Pull origin"*, que baixa essas alterações e as incorpora ao seu repositório local [GitHub Docs; coderefinery.github.io].

Quando você faz modificações no seu computador, primeiro salva essas mudanças com um *commit*, depois usa *"Push origin"* para enviar aquelas alterações ao repositório no GitHub para que outros colaboradores também possam vê-las. Se existirem *commits* no repositório remoto que você ainda não possui localmente, o GitHub Desktop normalmente pede que você faça um *fetch* antes de permitir o *push*, para evitar conflitos ou divergências de histórico [GitHub Docs].

5.6. Vantagens e Desvantagens

Quanto às vantagens e desvantagens da utilização de ferramentas como o GitHub Desktop, apresentam-se a seguir as formas pelas quais podem ser benéficas ao usuário, bem como alguns pontos negativos observados em seu uso.

5.6.1. Vantagens

No tocante às vantagens em relação à interface de linha de comando, especialmente para o público-alvo menos familiarizado com o Git, destacam-se os seguintes benefícios:

1. Trabalho colaborativo.
2. Controle de conflitos.
3. Plataforma gratuita (até certo nível).

5.6.2. Desvantagens

Quanto às desvantagens encontradas nesta ferramenta, embora poucas, podem ser citadas algumas:

1. Curva de aprendizado inicial.
2. Dependência de internet (para uso do GitHub).
3. Conflitos podem ser complexos de resolver.

5.7. Boas Práticas no Uso do Git e GitHub

O uso de sistemas de controle de versão, como o Git, e de plataformas de hospedagem e colaboração, como o GitHub, é essencial no desenvolvimento de software moderno. Além de permitir o rastreamento das alterações no código, essas ferramentas fomentam a colaboração eficiente entre equipes e garantem maior qualidade no produto final. No entanto, para que se obtenha o máximo de benefícios, é necessário adotar boas práticas que padronizem o fluxo de trabalho e reduzam problemas futuros. A seguir, destacam-se algumas recomendações fundamentais.

5.7.1. Commits pequenos e frequentes

Um erro comum em projetos de software é acumular diversas modificações antes de registrar um *commits*. Essa prática dificulta a rastreabilidade e aumenta a probabilidade de conflitos. O ideal é realizar commits pequenos, que representem uma alteração coesa e independente, como a correção de um bug específico ou a implementação de uma funcionalidade pontual. *Commits* frequentes facilitam a revisão, permitem a reversão de mudanças sem comprometer grandes blocos de código e tornam o histórico do projeto mais legível.

5.7.2. Mensagens de commit claras

A mensagem associada a um commit deve ser descritiva e direta, informando o que foi alterado e, preferencialmente, o motivo. Mensagens vagas como “ajustes” ou “mudanças finais” não auxiliam na compreensão do histórico do projeto. Uma boa prática é utilizar um padrão, como:

- `fix:` para correções de erros.
- `feat:` para implementação de novas funcionalidades.
- `docs:` para alterações na documentação.

Esse tipo de padronização contribui para a manutenção futura do projeto e para a colaboração eficiente em equipe.

5.7.3. Organização do repositório

A estrutura do repositório reflete a maturidade e a profissionalização de um projeto. Pastas mal organizadas ou a ausência de documentação dificultam a contribuição de novos desenvolvedores e prejudicam a escalabilidade do software. Recomenda-se manter um arquivo `README.md` bem escrito, que apresente objetivos, instruções de instalação, dependências e exemplos de uso. Além disso, separar arquivos de código, testes, documentação e recursos auxiliares em diretórios distintos favorece a clareza e a manutenção do projeto.

5.7.4. Nomeação significativa de branches

Branches (ramificações) são fundamentais para o desenvolvimento paralelo de funcionalidades, correções e experimentos. No entanto, nomes genéricos como ”teste”ou ”nova”dificultam a compreensão de seu propósito. Recomenda-se adotar nomenclaturas padronizadas, como:

- `feature/login` para o desenvolvimento de uma nova funcionalidade.
- `bugfix/navbar` para a correção de um problema específico.
- `hotfix/security` para correções críticas e imediatas.

Essa prática organiza o fluxo de trabalho e facilita o gerenciamento do ciclo de vida das alterações.

5.7.5. Revisão de código antes do merge

O processo de *code review* consiste na revisão de código por outros membros da equipe antes da integração na *branch* principal, sendo um dos pilares da qualidade em projetos colaborativos. A revisão permite identificar erros, melhorar a legibilidade e compartilhar conhecimento entre os desenvolvedores. No GitHub, esse processo é facilitado por *pull requests*, que centralizam a discussão sobre uma alteração antes de sua incorporação definitiva. O resultado é um código mais robusto, padronizado e confiável.

5.8. Considerações finais

A adoção dessas boas práticas no uso do Git e do GitHub não apenas melhora a qualidade técnica do código, mas também fortalece a colaboração e a eficiência dentro das equipes de desenvolvimento. Em projetos científicos, acadêmicos ou corporativos, o rigor na aplicação dessas práticas contribui diretamente para a reprodutibilidade, a transparência e a longevidade das soluções desenvolvidas.

Referências

Atlassian. (n.d.). Comparing workflows. *Atlassian Git Tutorials*. https://www.atlassian.com/git/tutorials/comparing-workflows.

Coursera. (2023). What is Git?. *Coursera*. https://www.coursera.org/articles/what-is-git.

Chacon, S., & Straub, B. (2014). *Pro Git* (p. 456). Springer Nature.

Earth Data Science. (n.d.). Basic Git commands. *Earth Data Science Workshops*. https://earthdatascience.org/workshops/intro-version-control-git/basic-git-commands.

Encyclopaedia Britannica. (n.d.). GitHub. *In Britannica*. https://www.britannica.com/technology/GitHub.

Everhour. (n.d.). What is GitHub?. *Everhour Blog*. https://everhour.com/blog/what-is-github.

GitHub, Inc. (s.d.). *Documentação do GitHub Desktop*. Recuperado em 16 de julho de 2025, de https://docs.github.com/pt/desktop.

GitHub. (2025). Instalar o GitHub Desktop. *GitHub Docs*. Recuperado em 8 de setembro de 2025, de https://docs.github.com/pt/desktop/installing-and-authenticating-to-github-desktop/installing-github-desktop.

GitHub. (2025). Criar uma conta no GitHub. *GitHub Docs*. Recuperado em 8 de setembro de 2025, de https://docs.github.com/pt/get-started/start-your-journey/creating-an-account-on-github.

Git SCM. (n.d.). *Getting started: A short history of Git*. Git. https://git-scm.com/book/ms/v2/Getting-Started-A-Short-History-of-Git.

GitLab. (2025). Journey through Git's 20-year history. *GitLab Blog*. https://about.gitlab.com/blog/journey-through-gits-20-year-history.

GitHub. (2025). Git turns 20: A Q&A with Linus Torvalds. *GitLab Blog*. https://github.blog/open-source/git/git-turns-20-a-qa-with-linus-torvalds.

GitHub Docs. (n.d.). About pull requests. *GitHub*. https://docs.github.com/pt/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests.

GitHub Docs. (n.d.). About GitHub and Git. *GitHub*. https://docs.github.com/en/get-started/start-your-journey/about-github-and-git.

I Programmer. (2025). Linus on Git. *I Programmer*. https://www.i-programmer.info/news/82-heritage/17977-linus-on-git.html.

Mats, S. (2020). The story of GitHub: How a weekend hack became the world's code playground. *Medium*. https://stevemats.medium.com/the-story-of-github-how-a-weekend-hack-became-the-worlds-code-playground-928010893bb1.

PSL Models. (n.d.). History of GitHub. *Git Tutorial*. https://pslmodels.github.io/Git-Tutorial/content/background/GitHubHistory.html.