

**CICLO FORMATIVO**

# GitHub



# INTRODUÇÃO À PROGRAMAÇÃO, GIT E GITHUB



## Cronograma

1. Linhas de comando;
2. Versionamento, Git e GitHub;
3. Exercícios.

### 1. Linhas de Comando

Quando usamos o computador, frequentemente interagimos por meio de cliques e menus, via interface gráfica. Existe outra maneira poderosa de passar instruções ao computador, por linhas de comando - uma forma direta e eficiente de se comunicar com o sistema, usando apenas texto.

Linhas de comando permitem manipular arquivos e pastas, executar programas e uma série de tarefas que, muitas vezes, são mais rápidas e flexíveis se comparadas ao uso do *mouse*. Ao invés de navegar por janelas e pastas, basta digitar comandos específicos para o computador interpretá-los e executá-los imediatamente.

Algumas interfaces interpretam linhas de comando por textos que manipulam arquivos em nossos computadores, tais como:

- ⇒ **Command Power/Prompt de Comando (CMD)**, um interpretador Windows simples e funcional;
- ⇒ **PowerShell** criado pela Microsoft, mais robusto e com capacidade maior de programação;
- ⇒ **Bash**, criado como *software* livre. Trata-se de um Unix Shell com linguagem de comando.

## Comandos básicos

Agora que você tem uma ideia do que sejam linhas de comando, vamos aprender alguns comandos básicos úteis no seu dia a dia.

Por funcionarem em vários sistemas (especialmente *Bash* e *PowerShell*), tais comandos ajudam a navegar e manipular arquivos e pastas.

São eles:

- ➔ **pwd**: imprime diretório de trabalho ("print working directory") e encontra o caminho para o diretório atual da pasta na qual você estiver;
- ➔ **ls**: lista todos os arquivos da pasta na qual você estiver;
- ➔ **cd nome-da-pasta**: entra em uma pasta dentro da que você estiver;
- ➔ **cd ~**: volta para a pasta raiz na qual você estava;
- ➔ **cd..** : volta uma pasta;
- ➔ **mkdir nome-da-pasta**: cria uma pasta;
- ➔ **rm nome-do-arquivo**: deleta um arquivo;
- ➔ **rm -r nome-da-pasta**: deleta um repositório;
- ➔ **whoami** : identifica usuário logado.



## Exercício 1\*

- a. Abra o *bash*;
- b. Identifique o usuário;
- c. Confirme a pasta na qual você está;
- d. Crie uma pasta;
- e. Entre na pasta;
- f. Crie um arquivo e insira uma frase;
- g. Tire uma *print* e mostre para a gente! \*

### \*Exercício 1 - Resposta

- a. Abra o *bash*;
- b. Identifique o usuário (**whoami**);
- c. Confirma a pasta na qual ele se encontra (**pwd**);
- d. Crie uma pasta (**mkdir nome-da-pasta**);
- e. Entre na pasta (**cd nome-da-pasta**).
- f. Crie um arquivo;
- g. insira uma frase pelo terminal **echo frase > nome.txt linda > nome.txt** excluindo terminal **rm -f nome.txt**;

Tire uma **print de tela** e mostre para a gente!

## Exercício 2\*

- a. Abra o terminal *bash*;
- b. Confirme a pasta na qual estiver;
- c. Entre na pasta criada anteriormente;
- d. Apague o arquivo criado;
- e. Volte uma pasta;
- f. Apague a pasta criada. \*

### Respostas\*

- a. Abra o *bash*
- b. Confirme pasta na qual estiver (*pwd*);
- c. Entre na pasta criada antes (*cd nome-da-pasta*);
- d. Apague o arquivo criado (*rm nome-do-arquivo*);
- e. Volte uma pasta (*cd*);
- f. Apague a pasta criada (*rm -r nome-da-pasta*).

## 2. Versionamento de código

Quando trabalhamos em projetos de programação ou que envolvam arquivos (documentos, planilhas ou código), é importante ter controle sobre as mudanças feitas ao longo do tempo.

Imagine perder horas de trabalho porque você não conseguiu recuperar a versão anterior de um arquivo, ou misturar alterações de diferentes membros da equipe sem saber quem fez o quê. Aqui entra o controle de versão.

### Controle de versão

“Controle de versão” é um sistema que registra as alterações feitas em um arquivo, ou conjunto de arquivos, ao longo do tempo. Ele permite retornar a versões anteriores, comparar diferentes versões entre si e visualizar quem fez alterações; uma espécie de máquina do tempo para seus arquivos.

Essencial em ambientes corporativos e acadêmicos nos quais várias pessoas trabalham em um mesmo projeto simultaneamente, esse tipo de *software* não se limita apenas a arquivos de código; também pode ser usado com qualquer tipo de arquivo, como documentos de texto, planilhas, ou imagens.

Com o controle de versão, é possível gerenciar desde pequenos projetos individuais até grandes projetos em equipe, o que torna a ferramenta indispensável para desenvolvedores e profissionais em diversas áreas.

### Git

O Git é um *software* livre, inicialmente projetado e desenvolvido pelo engenheiro [Linus Torvalds](#) para o desenvolvimento do Kernel Linux. É rápido e eficiente, especialmente em projetos grandes.

Git é um dos sistemas de controle de versões mais usados hoje, principalmente para desenvolvimento de *softwares*. Ele se destaca por ser:

- ⇒ **Livre e de código aberto:** qualquer pessoa pode usar, modificar e distribuir o Git, o que contribui para sua enorme adesão global;
- ⇒ **Rápido e eficiente:** mesmo em projetos grandes com muitos arquivos e alterações, o Git foi projetado para ser rápido e garantir que você trabalhe sem atrasos;

- ➔ **Distribuído:** cada desenvolvedor possui uma cópia completa do histórico do projeto em seu próprio computador, o que oferece grande flexibilidade e resiliência, já que você pode trabalhar *offline* e manter o acesso a todas as funcionalidades do controle de versão.

## Ferramentas de versionamento

### GitHub



O GitHub é uma das plataformas mais populares para hospedagem de código-fonte e controle de versões, por utilizar Git como sistema de controle de versão, além de oferecer interface amigável para desenvolvedores de todo o mundo serem capazes de colaborar em projetos de forma pública ou privada.

Com GitHub, você pode:

- ➔ **Armazenar e gerenciar seu código** hospedando seus projetos e controle de versões e mantendo-os organizados em repositórios;
- ➔ **Colaborar com outros desenvolvedores** contribuindo em projetos de código aberto ou trabalhando em conjunto com seu time em projetos privados, independentemente da localização;
- ➔ **Compartilhar seu trabalho** publicando código e projetos para que outros possam utilizá-los, comentar a respeito e até mesmo contribuir com eles.

O GitHub é mais do que uma plataforma de hospedagem: ele também funciona como rede social para desenvolvedores, ao permitir que você descubra novos projetos, aprenda com o código de terceiros e construa uma reputação na comunidade de *software*.

## Conceitos básicos Git

Antes de mergulharmos nos comandos do Git, é importante que entendamos os seguintes conceitos fundamentais:

- ➡ **Repositório:** diretório ou pasta onde seu projeto é armazenado, onde o Git rastreia todas as mudanças feitas no projeto. Pode ser encontrado em seu computador (local) ou internet (remoto, como no GitHub);
- ➡ **Clone:** ao clonar um repositório, você cria uma cópia total do repositório remoto em seu computador local (por exemplo, do GitHub), lhe permitindo trabalhar no projeto de maneira independente;
- ➡ **Branches (ramificações):** permitem trabalhar em diferentes partes de um projeto sem afetar a linha principal de desenvolvimento. Cada *branch* funciona como “braço” do projeto, pois permite que desenvolvedores façam alterações e experimentos sem interferir no trabalho de terceiros;
- ➡ **Pull:** puxa as últimas atualizações do repositório remoto para o repositório local, para garantir que você esteja sempre trabalhando na versão mais recente do projeto;
- ➡ **Commit:** funciona como um “salvar” no Git. Ao fazer um *commit*, você registra mudanças em seu projeto junto a uma mensagem explicando o que foi alterado;
- ➡ **Push:** envia mudanças feitas no seu repositório local para o remoto, como no GitHub. O projeto remoto será atualizado com as alterações feitas por você;
- ➡ **Merge:** combina mudanças de diferentes *branches* em um único *branch*. Especialmente útil ao integrar o trabalho feito por diferentes desenvolvedores de um mesmo projeto em uma só versão, unificada;
- ➡ **Fork:** criar uma cópia de um projeto preexistente na sua conta do GitHub impedindo que mudanças e melhorias afetem o repositório original;
- ➡ **Pull Request:** solicitação para que suas mudanças feitas em *branch* ou *fork* sejam integradas ao projeto principal. Uma maneira de colaborar com projetos de outros desenvolvedores;
- ➡ **Rebase:** similar ao *merge*, reorganiza o histórico de *commits* apagando ou combinando-os para um histórico mais limpo. É recomendado para a integração de mudanças entre *branches* de desenvolvedores, antes do envio ao *branch* principal.

## Comandos básicos de Git

Alguns comandos fundamentais que você usará com frequência:

- ⇒ **git init** inicia um novo repositório Git em uma pasta local;
- ⇒ **git add** adiciona arquivos modificados à área de *staging* (preparação), onde você decidirá quais mudanças incluir no próximo *commit*;
- ⇒ **git status** mostra o estado atual dos arquivos em seu repositório e indica quais foram modificados, adicionados ou removidos;
- ⇒ **git commit-m ("mensagem")** cria um *commit* salvando suas mudanças no histórico do projeto com uma mensagem descriptiva;
- ⇒ **git pull** atualiza seu repositório local com as mudanças mais recentes do repositório remoto ("remoto → local");
- ⇒ **git push** envia as mudanças do seu repositório local para o remoto ("local → remoto");
- ⇒ **git remote add origin (URL do repositório)** conecta seu repositório local a um remoto no GitHub;
- ⇒ **git checkout (nome-do-arquivo)** desfaz alterações locais em um arquivo específico revertendo-o para a última versão confirmada (*committed*).

### Exercício 3\*

- a. Comece com um Git no terminal;
- b. Crie uma pasta;
- c. Navegue até a pasta criada e inicie o git;
- d. Crie um arquivo e verifique o estado;
- e. Adicione o arquivo ao *stage* do Git;
- f. Faça um *commit*;
- g. Faça um *push*.

### Exercício 4\*

- a. Crie um repositório localmente e inicie o git;
- b. Adicione um arquivo *markdown* chamado "**README**" com seu nome e prato favoritos e, em seguida, faça um *commit*;
- c. Adicione uma curiosidade a seu respeito e faça outro *commit*;
- d. Publique o repositório em seu GitHub.

## APROFUNDANDO NO GIT

Agora que entendemos os conceitos básicos do Git, é hora de configurar seu ambiente para começar a trabalhar em projetos.

Antes de utilizar o Git de forma eficaz, é essencial efetuar algumas configurações iniciais que garantirão a ele saber quem você é e como se comportar em seu sistema.

### Configurações iniciais

Antes de criar repositórios e fazer *commits*, o Git precisa conhecer algumas informações a seu respeito que o ajudem a associar suas alterações ao seu nome e e-mail. Isso é crucial para um histórico claro e organizado de quem fez o quê em um projeto.

As configurações iniciais incluem:

#### **git config - global user.name ("seuUser")**

Este comando define o nome de usuário associado a todos os *commits* feitos por você. O exemplo "seuUser" aparecerá nos históricos de *commits* como autor das alterações;

#### **git config - global user.email ("[nome@email.com](mailto:nome@email.com)")**

Similar ao nome de usuário, define o e-mail associado aos seus *commits*. Este e-mail é importante porque identifica apenas você, especialmente ao contribuir para projetos em plataformas como o GitHub;

#### **git config list**

Após configurar nome e e-mail, você poderá usar o comando para verificar todas as configurações atuais do Git. Ele listará todas as configurações ativas naquele momento e permitirá que você confirme se está tudo OK.

### Modificando configurações

Às vezes, é preciso alterar configurações em um projeto específico, como e-mail ou nome de usuário diferente.

Configurar nome e e-mail de usuário é tarefa essencial para que seu trabalho seja devidamente creditado em projetos dos quais participar. Em um ambiente colaborativo como o

GitHub, tais informações permitem que outras pessoas saibam quem foi responsável por cada alteração.

Essas configurações também ajudam a manter um histórico claro e organizado, que facilita o rastreamento de mudanças e a colaboração entre equipes.

Abaixo, comandos para usar em caso de mudança de configuração:

### **git config --global --unset user.name “seu usuário”**

Remove a configuração atual do nome de usuário e permite configuração de novo nome usando **git config --global user.name**.

### **git config --global --unset user.email “[nome@email.com](mailto:nome@email.com)”**

Remove o e-mail configurado e permite definir um novo e-mail utilizando **git config --global user.email**.

### **Exercício 5 (algoritmos)\***

- a. Faça um *fork* do repositório;
- b. Clone o repositório para a sua máquina;
- c. Crie um novo *branch* com seu nome (exemplo: amanda-silva);
- d. Faça *commits* com a resolução dos exercícios;
- e. Atualize seu repositório remoto.

#### **DESAFIO EXTRA**

Abra um *pull request* para o repositório original.

## Branches no Git

*Branches* (ramificações) representam uma das funcionalidades mais poderosas do Git. Elas permitem a você trabalhar em diferentes partes de um projeto de forma isolada, sem interferir na linha principal de desenvolvimento (usualmente chamada de *main* ou *master*).

Com *branches*, você pode experimentar novas ideias, corrigir *bugs*, ou desenvolver novas funcionalidades, sem medo de “quebrar algo” no projeto principal.

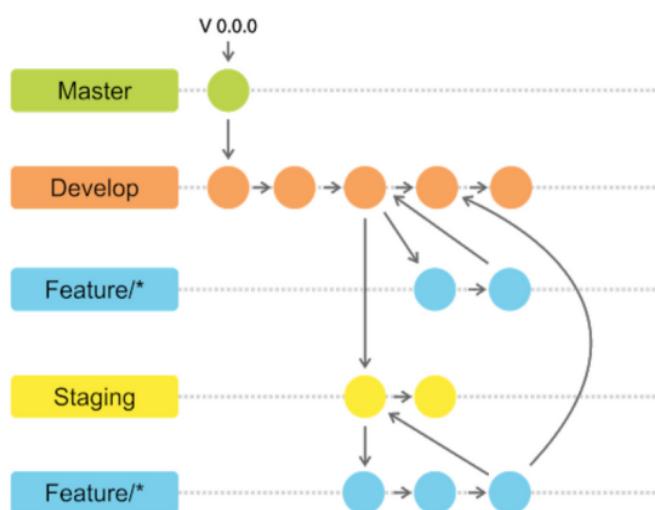
### Por que usar branches?

O uso de *branches* é essencial para um fluxo de trabalho organizado e eficiente, especialmente em equipes.

Com *branches*, você pode:

- ➡ **Isolar novas funcionalidades** e desenvolver novas ideias sem afetar o código já em funcionamento;
- ➡ **Trabalhar em paralelo** junto a várias pessoas em diferentes aspectos de um projeto de forma simultânea, sem conflitos;
- ➡ **Experimentar com segurança** testando novas abordagens ou soluções sem comprometer a estabilidade do projeto.

*Branches* permitem que o desenvolvimento seja mais modular, organizado e seguro. Ao dominar esses comandos, você estará mais bem preparada para colaborar em projetos de qualquer escala, garantindo que suas contribuições sejam integradas de maneira eficiente e sem problemas.



## Comandos essenciais

Abaixo, comandos essenciais para trabalhar com *branches* no Git:

### git branch

Lista todas as *branches* disponíveis no seu repositório. A *branch* na qual estiver trabalhando, será destacada com um asterisco (\*);

### git checkout -b nome-da-branch

Cria uma nova *branch* e muda para ela imediatamente. Se quiser começar a trabalhar em uma nova funcionalidade, pode criar uma *branch* chamada “nova-funcionalidade” e começar a desenvolver nela sem afetar o código na *branch* principal;

### git checkout nome-da-branch

Permite alterar uma *branch* para outra. Caso já tenha criado uma *branch* e deseje voltar a ela, basta usar o comando seguido do nome da *branch* desejada. A ação trocará o contexto de trabalho para a *branch* especificada, para que você possa visualizar e trabalhar nas alterações feitas lá;

### git branch -d nome-da-branch

Ao terminar de trabalhar em uma *branch* integrando alterações ao projeto principal, você poderá deletá-la usando este comando. Ele removerá a *branch* do repositório local e manterá seu ambiente de trabalho organizado.

Só será possível deletar uma *branch* já integrada (mesclada), ou que você tenha certeza de que não precisa mais;

### git push origin --delete nome-da-branch

Se quiser deletar uma *branch* que também tenha sido enviada para o repositório remoto (por exemplo, no GitHub), use este comando para remover a *branch* do repositório remoto e garantir que ela não fique mais disponível *online*;

### git merge nome-da-branch

Integra ou mescla alterações de uma *branch* específica à atual.

Se tiver criado uma *branch* para adicionar nova funcionalidade e quiser incorporar essas alterações à *branch* principal (*main*), mude antes para a *branch* principal (“*git checkout main*”) e, em seguida, execute o comando “*git merge nome-da-branch*”, para unir as mudanças de ambas as *branches*;

## Forks em Git

*Forks* (ramificações) são uma maneira poderosa de colaborar em projetos de código aberto ou contribuir para repositórios que não pertençam a você.

Ao fazer um *fork* de um repositório, você cria uma cópia completa deste repositório em sua conta própria do GitHub que possibilita experimentar, modificar e trabalhar em mudanças sem afetar o repositório original.

Quando suas modificações estiverem prontas, você pode solicitar que sejam integradas ao projeto original através de um *pull request*.

## Quando usar forks?

*Forks* são especialmente úteis quando você quiser:

- ⇒ **Contribuir em projetos de terceiros** sem permissão de escrita no repositório original. O *fork* permite efetuar alterações e propõe melhorias;
- ⇒ **Experimentar novas ideias** modificando o projeto em seu *fork*, sem medo de “quebrar algo” no repositório original;
- ⇒ **Trabalhar em paralelo com a equipe do projeto original** sincronizando seu *fork* com o repositório original, para manter seu código atualizado enquanto desenvolve suas próprias funcionalidades.

## Principais comandos

Abaixo, os principais comandos e passos para trabalhar com *forks*:

- ⇒ **git clone url-do-seu-fork:** depois de fazer um *fork* de repositório no GitHub, você precisará cloná-lo para o seu computador. O comando cria uma cópia local do seu *fork* e permite trabalhar nele diretamente, a partir do seu ambiente local;
- ⇒ **git remote add upstream url-do-repo-original:** conecta seu repositório local ao original, onde você efetuou o *fork*. Usualmente chamado de “*upstream*”, o repositório original permite acompanhar atualizações feitas no projeto original e integrá-las ao seu *fork* sempre que necessário;
- ⇒ **git fetch upstream:** busca mudanças mais recentes no repositório original (“*upstream*”) trazendo-as para seu repositório local. Não aplica essas mudanças diretamente, mas as disponibiliza, para que você possa visualizá-las e decidir o que fazer;

- ⇒ **git merge upstream/main ou git rebase upstream/main:** Para manter seu fork atualizado com as últimas alterações do repositório original, você pode usar o comando *merge* ou *rebase*. Ambos os comandos integram as mudanças do repositório original (*upstream*) à sua *branch* principal (geralmente *main*), mas de maneiras diferentes;
- ⇒ **git merge upstream/main:** Este comando combina as mudanças do repositório original com as suas mudanças atuais, criando novo *commit*, que mescla as duas histórias de desenvolvimento;
- ⇒ **git rebase upstream/main:** Este comando aplica suas mudanças sobre as mudanças mais recentes do repositório original, reorganizando o histórico de *commits* para criar uma linha do tempo mais linear. O *rebase* é útil para manter um histórico de *commits* mais limpo, mas pode ser mais complicado de usar corretamente.

### Exercício 6\*

- a. No repositório/ pasta local, crie um *branch* com seu nome;
- b. Crie um arquivo com seu nome e escreva algo dentro dele;
- c. Adicione o arquivo ao git;
- d. Faça um *commit*;
- e. Faça um *git push* para seu repositório;
- f. Em seu repositório no GitHub, faça um *pull request* entre *forks* enviando sua *branch* para o repositório original.

### Exercício 7\*

- Crie um repositório no GitHub sem o *Readme.md*;
- Crie uma pasta no seu computador;
- Abra o git *bash*;
- Adicione seu repositório remoto como de origem;
- Dê o comando "git init";
- Crie uma *branch* com seu nome;
- Dentro da pasta, crie um arquivo e escreva algo dentro dele;
- Adicione o arquivo ao git;
- Faça um *commit*;
- Envie as alterações para o seu repositório remoto.



## FERRAMENTAS DE IA PARA GERENCIAMENTO DE CÓDIGO E AUTOMAÇÃO

Com a evolução da tecnologia, o desenvolvimento de *software* passou a contar com o auxílio de ferramentas de Inteligência Artificial (IA), que automatizam tarefas repetitivas, sugerem soluções e até escrevem código.

Uma das ferramentas revolucionando o desenvolvimento é o GitHub Copilot.

### O que é?

O GitHub Copilot é uma ferramenta de IA desenvolvida pela GitHub em parceria com a OpenAI. Ela funciona como um assistente de programação integrado ao seu editor de código: sugere linhas inteiras ou blocos de código a medida que você trabalha.

Com base no contexto do código sendo escrito, o Copilot analisa o que você fez e oferece sugestões para economia de tempo e esforço.

### Principais funcionalidades

- ⇒ **Autocompletar código:** Copilot pode completar automaticamente trechos de código a medida que você digita, para sugerir funções, variáveis e estruturas baseadas no que você já tiver escrito;
- ⇒ **Geração de código:** Copilot pode gerar blocos inteiros de código com base nos comentários ou linhas iniciais fornecidas por você. Útil para acelerar o desenvolvimento de funcionalidades comuns ou repetitivas;
- ⇒ **Sugestões inteligentes:** com base em padrões comuns e boas práticas, o Copilot sugere a melhor maneira de implementar uma funcionalidade ou resolver um problema de código;
- ⇒ **Compatibilidade com diversas linguagens:** embora mais otimizado para linguagens populares como Python, JavaScript, e TypeScript, o Copilot também suporta várias outras, o que o torna uma ferramenta versátil para desenvolvedores.

## Como o GitHub Copilot pode ajudar no gerenciamento de código?

- ⇒ **Melhoria na produtividade:** ao automatizar tarefas repetitivas e sugerir códigos, o Copilot permite que você se concentre em resolver problemas mais complexos e criativos. Pode reduzir significativamente o tempo necessário para escrever código, especialmente em tarefas comuns como criar funções ou estruturas de controle;
- ⇒ **Minimização de erros:** ao sugerir códigos que sigam padrões e práticas recomendadas, o Copilot ajuda a reduzir a quantidade de erros e bugs promovendo um código mais limpo e eficiente;
- ⇒ **Facilidade de colaboração:** em equipes nas quais diferentes desenvolvedores possuem diferentes níveis de habilidade, o Copilot serve como um mentor virtual, ajuda a nivelar o conhecimento e garante que todos contribuam de maneira eficaz;
- ⇒ **Automatização de tarefas repetitivas:** com o Copilot, você pode automatizar partes do seu fluxo de trabalho, como criação de testes unitários, documentação automática e até mesmo geração de código *boiler plate* (código padrão).

## Exemplos práticos de uso

- ⇒ **Criação de testes automáticos:** ao fornecer um comentário descrevendo o que um teste deve fazer, o Copilot sugere o código de teste completo e acelera o processo de desenvolvimento, para garantir que seu código seja bem testado;
- ⇒ **Documentação automatizada:** ao escrever código, o Copilot sugere automaticamente comentários e documentação, para manter seu projeto bem documentado sem esforço adicional;
- ⇒ **Refatoração de código:** se precisar refatorar uma função para torná-la mais eficiente ou legível, o Copilot pode sugerir melhorias automaticamente, baseadas em práticas recomendadas.

## Considerações éticas e limitações

Embora o GitHub Copilot seja uma ferramenta poderosa, é importante usá-la com consciência. Como se baseia em grandes quantidades de código aberto para suas sugestões, já existem discussões sobre direitos autorais e originalidade do código gerado. Também é essencial revisar cuidadosamente o código sugerido pelo Copilot para garantir que seja adequado ao seu contexto específico.

GitHub Copilot e outras ferramentas de IA estão transformando a maneira como desenvolvedores trabalham, ao tornar o processo de codificação mais rápido e eficiente. Ao integrar essas ferramentas ao fluxo de trabalho, você não apenas melhora sua produtividade como também aprende novas abordagens e técnicas de programação.

É crucial lembrar que estamos falando de assistentes, não substitutas de avaliação humana; por isso, é sempre importante revisar e entender os códigos gerados antes de utilizá-los.

## Conclusão

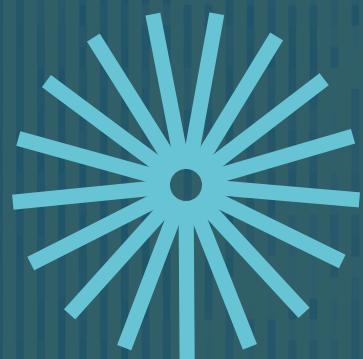
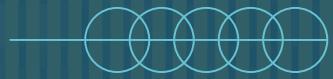
Ao longo desta apostila, exploramos os conceitos fundamentais do Git e GitHub, ferramentas essenciais no desenvolvimento de *software*.

De comandos básicos até funcionalidades mais avançadas como *branches* e *forks*, você agora possui o conhecimento necessário para gerenciar versões de código de maneira eficiente e colaborar em projetos com terceiros.

Também falamos sobre integração a ferramentas de Inteligência Artificial, como GitHub Copilot vem redefinindo o desenvolvimento de *software* ao torná-lo mais ágil e acessível.

Git e GitHub não são apenas ferramentas técnicas; são plataformas que fomentam colaboração, inovação e comunidade. Ao utilizar essas ferramentas, você se conecta com desenvolvedores do mundo todo e contribui para projetos que podem ter um impacto real.

Dominar as ferramentas e conceitos apresentados não só permitirá contribuir com projetos de qualquer escala como também abrirá portas para novas oportunidades na área da Tecnologia.



# CICLO FORMATIVO

---

# GitHub

---

