anaboo.ai

BEGINNER GUIDE · GIT & GITHUB BASICS

Git for Beginners: Pull, Commit, Push Across Multiple Computers

How your work moves between your computer and the master copy on GitHub, how to keep a laptop and a desktop in step with each other, and where the rest of a business's files and passwords should actually live. No technical background assumed, written for someone who has never touched Git before.

INSIDE THIS GUIDE

1 What Git and GitHub actually are

2 The three words you will see constantly

3 One-time setup, on every computer you use

4 The everyday routine, on any single computer

5 Working across multiple laptops and desktops

6 When something goes wrong

7 Where different files belong: code, documents, credentials

Version 1.0 · July 2026 · The whole routine on one page is at the end, print it and pin it up · No technical background assumed

1. What Git and GitHub actually are

Git is a small program on your computer that keeps a history of every change you make to a set of files. Every time you save a labelled point in that history, Git remembers exactly what changed, when, and what it looked like before.

GitHub is a website that stores one master copy of that history in the cloud. Your computer holds a copy, a "clone", of that history. Git and GitHub work together: Git tracks changes on your machine, GitHub is the shared place everyone's changes end up.

> **A USEFUL COMPARISON**
>
> It works a little like Google Docs version history, except you choose the moment a "version" gets saved (that is the commit), and you choose the moment it gets shared with everyone else (that is the push).

Why not just use Dropbox or Google Drive instead?

Those tools sync whole files and often struggle when two people edit the same file at once, sometimes overwriting one side quietly or duplicating the file as a "conflicted copy". Git tracks changes line by line, keeps the full history of every version, and tells you plainly when two edits clash instead of guessing.

2. The three words you will see constantly

> **pull**, get the latest version from GitHub onto whichever computer you are sitting at.

> **commit**, save your changes as a labelled point in your computer's local history.

> **push**, send your saved commits up to GitHub, where they become the new master copy.

The loop, every single time you work, is:

```
pull from GitHub  ->  make your changes  ->  commit  ->  push to GitHub
```

If a site is connected to a hosting service such as Vercel, that push is also what triggers the live site to rebuild, usually within a minute or two.

3. One-time setup, on every computer you use

Each computer needs its own copy of the project. This step happens once per machine, not once ever, so do it again on your second laptop or your home desktop.

You need these installed:

- **Git**, the tool that talks to GitHub. On a Mac, type `git --version` in Terminal and it offers to install it if missing. On Windows, install from git-scm.com.
- **A GitHub account**, added to the project by whoever owns it.
- **Claude Code (AIOS)**, if that is how you will be editing, installed and signed in.

Tell Git who you are, once per computer:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

Then make your own copy of the project on that computer:

```
git clone https://github.com/<organisation>/<project>.git
cd <project>
npm install
```

`npm install` is only needed for projects with a website preview. Skip it for plain document or content repositories.

4. The everyday routine, on any single computer

1. **Pull first, every time, before you touch anything.**

   ```
   git pull
   ```

   This brings down anything that changed since you last worked, so you are never editing an old copy without knowing it.

2. **Make your changes.** Edit with AIOS, or edit files directly in your usual editor.

3. **Save your changes as a commit.** The message in quotes is a short note on what changed, for your own record and everyone else's.

```
git add .
git commit -m "Update phone number and opening hours"
```

4. **Push, to publish.**

```
git push
```

That four-step loop, pull, edit, commit, push, is the whole job when you are the only person editing and working on one computer. The next part covers what changes when a second computer joins in.

---

5. Working across multiple laptops and desktops

This is the part that catches most beginners out, because the four-step loop above assumes you are always working from the same computer. As soon as a second laptop or a home desktop enters the picture, GitHub becomes the only place that can be trusted to hold the true, current version. Neither computer's hard drive is that place on its own.

> **THE GOLDEN RULE**
>
> **Pull before you start working on any machine. Push before you walk away from it.**
>
> Every computer holds its own separate copy of the history until you push or pull. If you finish a session on your work laptop and forget to push, that work exists in exactly one place: that laptop, switched off, wherever it happens to be. Sitting down at your home desktop and pulling will not bring it in, because it was never sent to GitHub in the first place.

What this looks like in practice

YOU ARE ABOUT TO…	DO THIS FIRST
Start editing on any computer, including one you used yesterday	`git pull`
Stop editing and walk away, even for a coffee	`git add .` then `git commit -m "..."` then `git push`
Switch to a different computer for the same project	Push from the machine you are leaving, pull on the machine you are arriving at
Come back to a computer you have not used in a while	`git pull`, do not assume it remembered anything on its own

A worked example

Say you edit the website on your laptop on Monday, and remember to push before closing the lid. On Tuesday you sit at your desktop, run `git pull`, and Monday's changes appear exactly as you left them. You add more changes, commit, and push again. On Wednesday you are back on the laptop, you pull, and Tuesday's desktop changes appear. The loop keeps both machines showing the same truth, provided each session starts with a pull and ends with a push.

Now the version that goes wrong: you edit on the laptop Monday and forget to push. Tuesday you sit at the desktop, pull, and get Sunday's version, with no sign that Monday's edits exist. You then start editing the desktop on top of that older version. Both machines now hold changes the other does not know about. This is recoverable, but it takes an extra step to fix (see Part 6), so it is worth avoiding by keeping the habit simple: pull to start, push to stop.

If more than one person edits the same project

The habit above still applies, plus one more layer. When two people might touch the same files, work on a branch (your own safe lane) rather than straight on the main copy, and bring your changes in through a Pull Request so nobody's edits get overwritten by accident.

```
git pull
git checkout -b yourname/short-description-of-work
#   ... make your changes ...
git add .
git commit -m "Describe what changed"
git push -u origin yourname/short-description-of-work
#   ... on GitHub, open a Pull Request and merge it into main ...
git checkout main && git pull        # back to a clean, current main
```

6. When something goes wrong

"YOUR BRANCH IS BEHIND", OR A PULL THAT REFUSES TO APPLY

Someone else, or your other computer, pushed changes you do not have yet. Run `git pull` again, Git usually merges the two sets of changes on its own without you noticing.

GIT WILL NOT LET YOU PULL BECAUSE OF UNCOMMITTED CHANGES

Save your current work first, `git add .` then `git commit -m "..."`, then pull. Git protects you by refusing to overwrite unsaved edits rather than silently discarding them.

A MERGE CONFLICT

Git flags the exact lines where two edits clash on the same spot, in the same file. This is normal and not dangerous, it means Git noticed something a file-sync tool would have silently guessed at. The simplest fix as a beginner: open the file in Claude Code and ask it to "resolve the merge conflict in this file, keep both changes where sensible", then commit the result. Pulling often and keeping each session's changes small is the best way to avoid conflicts arising at all.

YOU ARE NOT SURE WHICH COMPUTER HAS THE NEWEST VERSION

Run `git log -1` on each machine and compare the commit message and date. Whichever machine is missing a commit needs a pull, whichever machine has a commit nobody else has needs a push.

Nothing you do here can destroy the master copy on GitHub. If a change has a mistake in it, GitHub still keeps every earlier version in its history, so nothing is ever truly lost.

7. Where different files belong: code, documents, credentials

Git and GitHub are built for one job: tracking changes to code and text over time, with everyone working from the same shared history. That does not make them the right home for everything a business produces. Most teams end up using three separate places, each suited to a different kind of file.

Three jobs, three homes

- **Code and website source**, lives in Git and GitHub. Every change is tracked, so mistakes can be undone and the whole team works from one shared, current version.
- **Production and live business files**, spreadsheets, contracts, client deliverables, marketing assets, live in Google Drive, SharePoint, OneDrive, or Dropbox.
- **Credentials**, passwords, API keys, login tokens, live only on the local machine that needs them, never in Git, never in a shared drive.

Why code lives in Git, and documents usually do not

Git tracks changes line by line inside text files, which is exactly what source code is. That is also why it is a poor fit for the documents a business actually runs on day to day. A Word document, a spreadsheet, or a set of photos is a large, unreadable file to Git, it cannot show you what changed inside the file the way it can with code, so every save just looks like "the whole file changed again". Repositories full of files like these grow large and slow quickly.

Google Drive, SharePoint, OneDrive, and Dropbox solve a different problem: real-time collaborative editing by people who have never touched Git and never will. Two people can be in the same spreadsheet at once, changes appear instantly, and the built-in version history covers what most teams actually need, "what did this look like yesterday", without anyone needing to learn commit messages or branches.

> **THE RULE OF THUMB**
>
> If a client, a contractor, or a non-technical colleague needs to open and edit it directly, it almost certainly belongs in Drive, SharePoint, OneDrive, or Dropbox, not Git. If it is code, configuration, or written content that only gets changed by someone editing it in a code editor, it belongs in Git.

Credentials never go in either

Passwords, API keys, and login tokens are a special case, and the rule for them is stricter than either of the above: they live only on the individual computer that needs them, and nowhere that is shared with anyone else.

Two reasons this matters:

- A Git repository, even a private one, is not a safe place for a secret. If a key is ever committed, it stays in that repository's history even after the file is deleted. The only real fix is to change (rotate) the key and remove it from history everywhere, on every clone.
- A shared drive is usually visible to more people than should ever see a password, and it has no equivalent of "this folder can never be seen again by anyone who once had access."

Practically, this means each computer gets its own copy of any credentials it needs, entered once on that machine, not synced or copied in from another machine or a shared folder.

The .env file

Most projects that need credentials keep them in a single file called `.env` (short for "environment"), sitting in the project folder on your computer. This is where API keys, database passwords, and login tokens actually live day to day. Every project's `.gitignore` file lists `.env` specifically, so Git skips over it completely, it is never picked up by `git add`, never committed, and never pushed to GitHub, even by accident.

That is also why cloning a project onto a new computer never brings the credentials with it, `git clone` only ever fetches what is in GitHub, and `.env` was deliberately kept out. Each computer needs its own `.env` file created directly on it.

Why credentials never go on a cloud drive either

The same logic that keeps `.env` out of Git keeps it out of Drive, SharePoint, OneDrive, and Dropbox too. Those tools sync automatically to every device signed into the account, plus anyone the folder has ever been shared with. A password or API key uploaded there, even briefly, is effectively out of your control, you cannot be certain which devices already hold a copy or where it was cached along the way. Credentials stay off any cloud sync entirely.

IF YOU GENUINELY NEED TO MOVE CREDENTIALS TO ANOTHER COMPUTER

Sometimes a new laptop or desktop genuinely needs the same `.env` file rather than its own freshly issued keys. When that happens, move the file physically. Do not email it and do not put it through a cloud drive to get there.

Copy the `.env` file onto a USB drive or external disk. Carry that drive to the other computer and copy the file across. Delete the `.env` file from the USB drive immediately afterwards, before it sits in a drawer or goes anywhere else.

Never send it as an email attachment or a chat message. Email and chat systems keep their own copies in sent folders, servers, and backups, well beyond your control, for years after you have forgotten it was ever sent.

Quick reference

WHAT IT IS	WHERE IT LIVES	WHY
Code, website source, configuration	Git and GitHub	Tracks every change, one shared true version
Documents, spreadsheets, client deliverables, marketing assets	Drive, SharePoint, OneDrive, or Dropbox	Built for real-time editing by non-technical people, with simple built-in history
Passwords, API keys, login tokens, the `.env` file	The local machine only, moved by hand if it ever has to move	A leaked secret in Git or a shared drive cannot be safely undone, only rotated

IF A SECRET ENDS UP IN GIT BY ACCIDENT

Stop. Do not just delete the file and commit again, the old value is still sitting in the history. Tell whoever owns the project immediately so the key can be rotated and the history cleaned up. It is the same principle as a lost key: change the lock, do not just hide the old key harder.

The whole routine, one page

Print this page and keep it by your desk. It covers every computer you use for this project.

Every session, on any computer

1. `git pull`, always first, no exceptions
2. Make your changes
3. `git add .`
4. `git commit -m "what changed"`
5. `git push`, always last, before you walk away

Moving between computers

LEAVING A COMPUTER

Commit and push before you close the lid or step away. Unpushed work only exists on that one machine.

ARRIVING AT A COMPUTER

Pull before you change anything, even if you were the last person to use that machine.

THE ONE RULE THAT MATTERS

GitHub is the only copy that counts as the truth. Nothing on a laptop or desktop is safe until it has been pushed.