Git Tutorial for Researchers

1. Introduction to Git

Git is a **version control system (VCS)** that tracks changes in files and facilitates collaborative development. Instead of saving multiple copies of files (e.g., “analysis_final_v2”, “analysis_final_v3”), Git lets you maintain a history of **committed** changes, so you can revert to earlier versions and see what was changed by whom. For individual researchers, Git provides a powerful way to track and compare code versions, retrace errors, and explore new approaches in a structured manner, all while keeping a full history of changes. In collaborative projects, Git allows multiple people to work asynchronously (even offline) and merge their contributions, maintaining an **authorship trail** that shows who contributed what. This makes it easier to reproduce and validate scientific results, since you can precisely identify which version of code or data produced a given result. In short, Git improves research transparency and reproducibility by ensuring that every change to your code or writing is recorded and can be reviewed later.

Using Git might feel like extra overhead at first, but it saves time and prevents loss of work. With Git, you no longer have to manually merge different versions of files emailed between collaborators or worry about breaking things – you can always roll back to a previous state if a change doesn’t work out. Version control is essential for “**reproducible research**” because it captures the provenance of your analyses (which code and data versions were used) and enables others (or your future self) to reliably reuse or extend your work. By sharing your Git-managed repository (for example, on GitHub), you make it easier for colleagues to inspect your methodology, run your code, and contribute improvements. In summary, Git is a valuable tool for researchers to **collaboratively develop code**, **share it openly**, and **ensure reproducibility** in their computational workflows.

2. Setting Up Git

Installing Git on Windows: To get started with Git on Windows, download the official Git for Windows installer from the Git website. Run the installer and follow the prompts (you can accept the default settings in most cases). After installation, you will have an application called **Git Bash** (a terminal for running Git commands) and Git integrated into PowerShell/Command Prompt. Verify the installation by opening **Git Bash** (or any terminal) and typing: `git --version`. You should see Git report a version number if it’s installed correctly. Next, configure your identity so that your commits are attributed to you. In Git Bash or Command Prompt, enter:

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

Replace “Your Name” and email with your actual name and email. This information will be associated with your commits. (The `--global` flag means this is a one-time setup for your user profile on the machine. If you skip this, Git will prompt for your name/email when you make the first commit.)

Installing Git on Linux: Most Linux distributions include Git by default or make it available via package managers. For example, on Debian/Ubuntu, you can install Git by running in the Terminal:

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

This will download and install Git. On Fedora/CentOS, you can use `sudo dnf install git` similarly. After installation, set up your name and email in the Terminal just like on Windows (using the `git config --global user.name` and `user.email` commands as shown above). Finally, you can confirm Git is ready by checking the version (`git --version`). Once Git is installed and configured, you’re ready to start using it on your projects.

Initial Setup Considerations: After installing, you might want to set up a text editor for Git (by default, Git uses Vim on Linux or Notepad on Windows for things like editing commit messages). You can configure a different editor, for example: `git config --global core.editor "code --wait"` to use VS Code, or `"nano"` for Nano editor, etc. This step is optional and can be done later. For now, the critical steps are installing Git and setting your username/email.

3. Git Basics

Before diving into commands, let’s clarify some fundamental Git concepts:

- **Repository (Repo):** A repository is essentially a project folder tracked by Git. When you “initialize” Git in a folder (with `git init`), Git creates a hidden folder called `.git` inside it. This `.git` directory *is* the repository – it contains all the metadata, history, and snapshots of your project. All versions of your files are stored in the repo’s history. If you delete the `.git` folder, you lose the version history of that project. You can create a new repository from scratch (empty folder + `git init`) or by **cloning** an existing repository (more on cloning later). Repositories can be local (on your computer) or remote (on a server like GitHub).

- **Commit:** A commit is a saved **snapshot** of your project at a point in time. In Git, each commit captures the state of all tracked files (those under version control) at that moment

. You can think of a commit as a checkpoint or milestone in your project history. Commits are created with the `git commit` command (after selecting changes to include, which is done by staging files with `git add`). Each commit has a unique ID (a hash) and includes an author, a timestamp, and a commit message describing the changes. Unlike some other systems that store only differences, Git treats each commit as a complete snapshot of the repository's files, but uses compression and linking to make storage efficient (files that didn't change aren't duplicated). This design makes it easy to retrieve any version or compare differences between commits. It's good practice to commit whenever you've made a logical set of changes (for example, added a new analysis function or fixed a bug), along with a descriptive message.

- **Branch:** A branch in Git is like an independent line of development, or a pointer to a series of commits. By default, a new repository has a **default branch** (named `main` in many setups, or `master` in older setups). You can create a new branch to develop a feature or try an experiment without affecting the main line of work. Internally, a branch is just a pointer/reference to a commit, and it moves forward as you add new commits. For example, if you create a branch called `experiment`, initially it points to the same commit as `main`. As you commit on `experiment`, that branch pointer moves along its own sequence of commits, separate from `main`. Branches enable non-linear development – multiple branches can diverge and later be merged. This is especially useful in collaboration: each collaborator can work on their own branch or fork, then integrate changes via pull requests. Even when working solo, branches let you try ideas (say, refactoring code or testing a new analysis method) in isolation. If the experiment works out, you can merge it back; if not, you can simply abandon that branch without disturbing the main code.

- **Remote Repository:** A remote repository is a version of your repository that is hosted on a server or service (like GitHub, GitLab, or Bitbucket) and shared with others. Your local repository (on your computer) is where you do your work, and you can synchronize with a remote repository to share your changes or fetch others' changes. Think of a remote repo as a central **file server** for your project's Git data, which teammates can push to or pull from. Technically, a remote repo contains the same commits, branches, and history as your local repo, but it usually doesn't have a working copy of the files (it's often a "bare" repository, consisting only of the `.git` data). You interact with remotes using commands like `git push` (to upload your commits to the server) and `git pull` (to download and merge commits from the server). If you're the only person working on a project, you don't *need* a remote repository – but having one is useful as an off-site backup or if you plan to work from multiple devices. For collaboration, the remote repository (e.g., on GitHub) is the hub where everyone's contributions come together. It's important to note that almost all version control actions (committing, branching, etc.) happen locally; the remote comes into play only when sharing changes.

In summary, a **repository** is your project storage (with full history), a **commit** is a snapshot in that repository's history, a **branch** is a named series of commits (a line of development), and a **remote** is a copy of the repository on a server for collaboration. Understanding these basics will make the Git workflows (commit, branch, merge, push, pull) much clearer as you start using them.

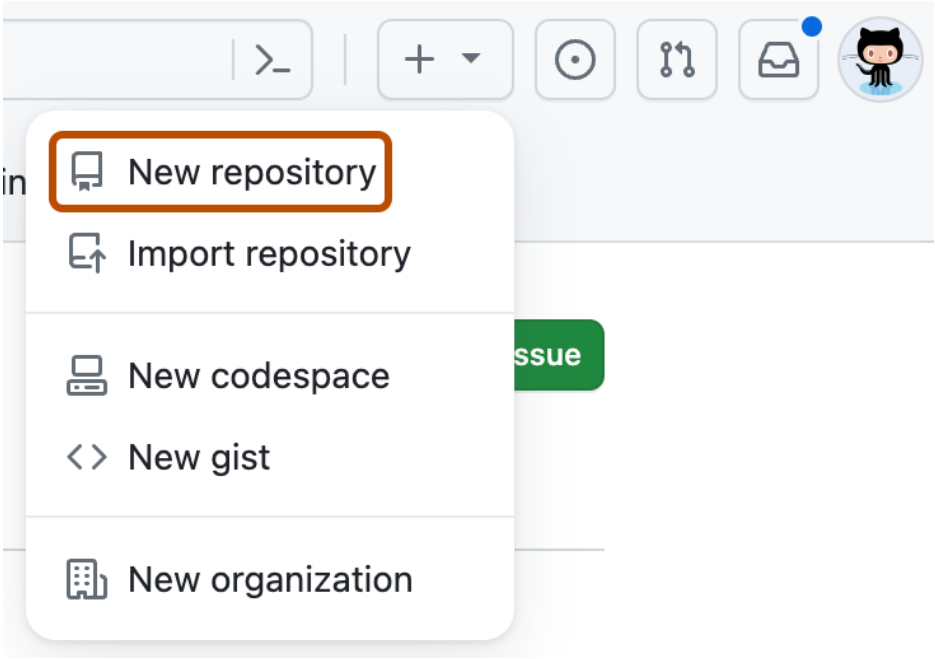
4. Using GitHub

What is GitHub?

GitHub is an online hosting service for Git repositories. It provides a convenient web interface to view your code, issues, documentation, and to manage collaboration (via pull requests, code reviews, etc.). GitHub is free to use for public and private repositories (with certain limitations), which makes it a popular choice for open-source projects and academic code sharing. To use GitHub, first create a free account with your email. Once logged in, you can create a new repository on GitHub through the web interface.

Creating a New Repository on GitHub:

In the top-right corner of GitHub's website, click the **"+"** button and select **"New repository"**.



This will take you to a form where you can initialize your repository:

- **Repository Name:** Pick a short, descriptive name for your project (e.g., `climate-analysis` or `my-experiment`). The name must be unique under your account and can't contain spaces (use hyphens or underscores if needed). You can also add an optional description to explain what the project is about.

- **Public or Private:** Choose the visibility. Public means anyone can see your code (recommended for open-source or if you plan to publish the code with a paper), while private means only you and specific collaborators can access it. (You can change a repo from private to public or vice versa later if needed.)
- **Initialize with a README:** It's often helpful to check this option for a brand-new repository. GitHub will create a README file (you can edit it to add project details) and it ensures the repo is not empty (which makes cloning easier). You may also choose to add a **.gitignore** (a template to ignore certain file types, e.g., Python, R, or TeX build files) and a **license** at this stage (or you can add these later).

After filling out the details, click **"Create repository."** GitHub will set up the repository and take you to its main page. If you initialize with a README, you will see that file listed. On the repository page, there are tabs like **Code**, **Issues**, **Pull Requests**, **Actions**, **Projects**, **Wiki**, **Settings**, etc., which correspond to various features. For now, focus on the **Code** tab – it shows the files in the repository and instructions for how to add more code.

Connecting Your Local Repository to GitHub:

There are two common workflows:

- *Creating on GitHub first:* Since you just made a repo on GitHub, you likely want to "pull" it to your computer. The easiest way is to **clone** it. On the GitHub repo page, find the green **"Code"** button and copy the URL (either the HTTPS link or an SSH link if you set up SSH keys). On your computer, open Git Bash or Terminal and run: `git clone <repo_url>`. This will download the repository to a new folder with the repository's name. Initially, it will just contain the README (and any files you had GitHub create for you). You can then add your files to this folder, use Git to commit changes, and later push them back to GitHub.
- *Existing project, uploading to GitHub:* If you already have a local project folder with Git initialized and some commits, you can add GitHub as a remote. On GitHub, create a new empty repository **without** initializing with a README (since you already have one locally). Then on your computer, in your project folder, run commands provided by GitHub.

  For example:
  `git remote add origin https://github.com/YourUsername/your-repo.git`
  (this adds a remote named "origin"), and then `git push -u origin main` (which uploads your local `main` branch to GitHub and sets it as the upstream for future pushes). After this, your local repo is linked to the GitHub repo.

Managing the Repository on GitHub: Once your code is on GitHub, you can do several things via the web interface:

- Browse the code files and history. GitHub shows each commit and allows you to see diffs (changes) for each commit.

- Edit or create files from the browser (for small changes or documentation).
- Open **Issues** to track bugs or tasks, and use **Pull Requests** to discuss and merge changes (especially if collaborating or if someone forked your repo to contribute).
- Add collaborators under **Settings > Manage access** if it's a private repo (invite team members by their GitHub username or email). Collaborators can push to the repository directly. Note that on GitHub Free, you can have unlimited collaborators even on private repositories.
- Manage settings like repository description, website links (useful if you have a documentation website or project page), branch settings, etc., in the **Settings** tab.

GitHub provides a friendly way to visualize your repository's Git data and to collaborate. However, remember that **GitHub is just one Git hosting service**. The Git commands you use locally (commit, push, pull, etc.) remain the same regardless of using GitHub, GitLab, or others – but GitHub adds extra features around the Git repository to facilitate teamwork.

5. Command-Line & GUI: Working with Git

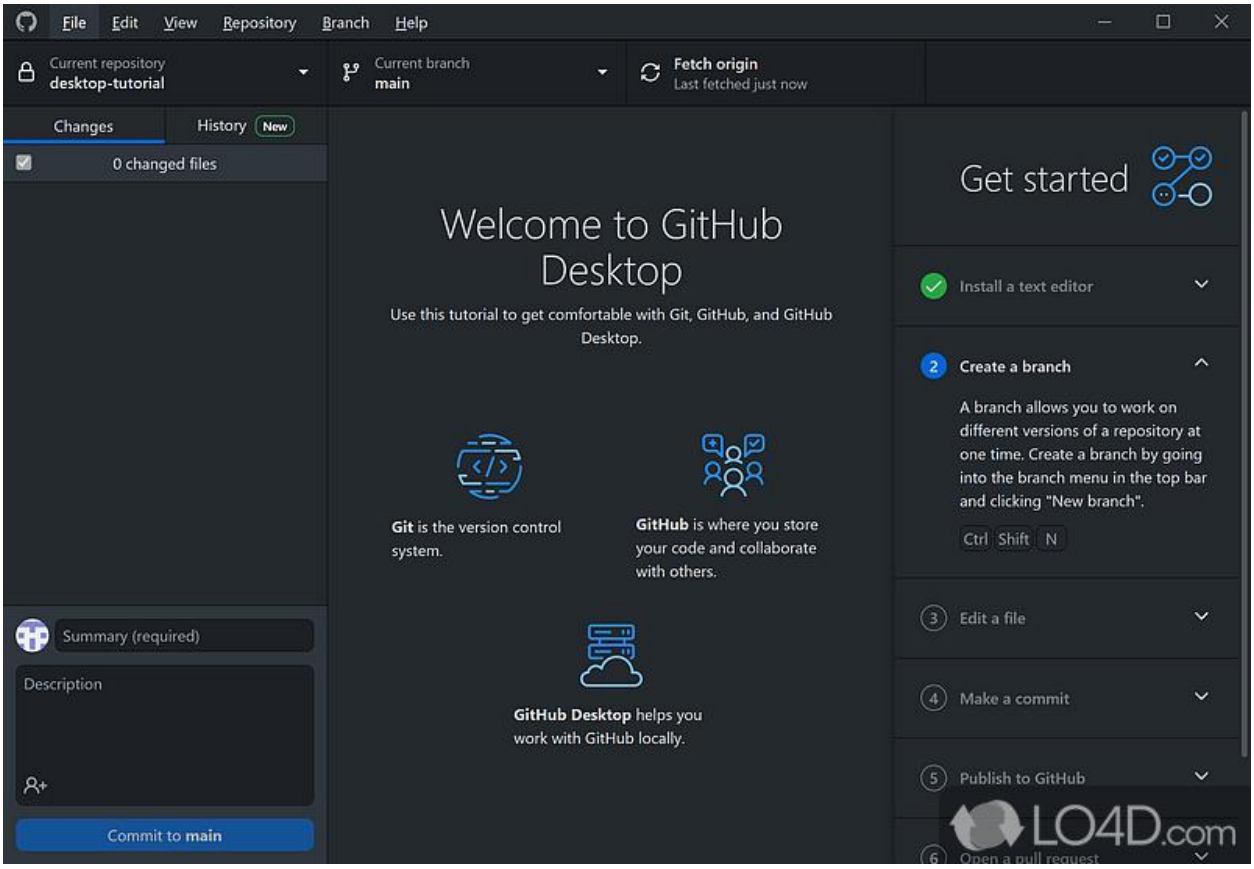
Using Git from the Command Line: The command-line interface (CLI) is the core way to use Git. On Windows, you can use **Git Bash** (installed with Git for Windows) or even PowerShell/CMD. On Linux or macOS, you'd use the Terminal. Using the CLI might seem daunting, but it gives you full control and is the most widely documented approach. Here's a typical workflow using Git commands in a terminal:

1. *Navigate to your project directory:* e.g., `cd ~/projects/my-repo`.
2. *Initialize Git (if not a repo yet):* `git init` – this creates the `.git` directory and starts version control for that folder.
3. *Check status:* `git status` – this shows which files are untracked, modified, or staged. Initially, your files will be untracked.
4. *Stage changes:* `git add filename` to stage a specific file, or `git add .` to stage all changes in the directory. Staging means selecting changes for the next commit (Git uses a staging area called the "index").
5. *Commit:* `git commit -m "Write a short message here"` – this creates a new commit with the staged changes. The message should briefly describe what you changed (e.g., "Add initial data analysis script"). After committing, `git status` will show a clean working tree (no pending changes).
6. *Branching:* To create a new branch, use `git branch new-branch-name`, then `git switch new-branch-name` (or `git checkout -b new-branch-name` in older syntax) to move to that branch. Now any commits you make will be on the new branch. You can switch back to `main` with `git switch main`.
7. *Viewing history:* `git log` – shows the commit history (press `q` to exit the log). You can use `git log --oneline --graph` for a condensed view with branches graph.

8. *Pushing to remote:* `git push origin main` – sends your `main` branch commits to the remote named "origin" (which is typically GitHub). The first time, you might need to set the upstream with `-u` as mentioned earlier.
9. *Pulling updates:* `git pull origin main` – fetches and merges changes from the remote's main branch into your current branch. (If collaborators pushed to GitHub, you'd use this to update your local copy.) Pull is essentially a shortcut for `git fetch` (download commits) followed by `git merge` (merge into your branch).

Common command-line tasks also include `git diff` (to see unstaged changes), `git add -p` (to interactively stage parts of a file), and `git merge` (to combine branches). While it's important to get comfortable with at least the basics of `add`, `commit`, `push`, and `pull`, you don't have to memorize every command up front. You can refer to a Git cheat sheet or the `--help` flag (e.g., `git commit --help`) when needed. Over time, the commands will become second nature.

Using GUI Tools (e.g., GitHub Desktop): If you prefer a visual interface, GitHub Desktop is a user-friendly GUI application that integrates with GitHub. It provides buttons and menus for the same Git operations, which can be easier for beginners or those who dislike the terminal. With GitHub Desktop, you can **clone repositories** with a few clicks, **view changes** in a visual diff, and fill in commit messages in a text box. The tool shows you a side-by-side view of your files and the repository status. For example, the welcome screen of GitHub Desktop even walks you through a tutorial repository with steps like "Create a branch," "Edit a file," "Make a commit," and "Publish to GitHub":



This can help you learn the Git concepts as you perform them in the GUI.

Using GitHub Desktop is straightforward: you sign in with your GitHub account, then either **create a new repository** (locally, with the option to publish it to GitHub) or **clone an existing repository** from GitHub. The interface has a left pane listing your repositories, a middle pane showing changed files, and a right pane showing the diff or the Getting Started steps for new users. To commit changes, you'd type a summary (and optional description) in the bottom-left, then click the **"Commit to main"** button instead of typing `git commit`. To push or pull, you use the **Repository** menu or toolbar buttons (e.g., "Push origin" to upload your commits to GitHub, "Fetch origin" to see if there are new changes on GitHub, which then allows you to pull). GUI apps often label these clearly, so you don't have to remember the exact commands.

Other GUI options include source-code editors like **VS Code**, which has built-in Git GUI features (source control panel), and third-party Git clients like **Sourcetree**, **GitKraken**, etc. Regardless of GUI, it's beneficial to understand what actions the GUI is performing under the hood (staging, committing, merging, etc.), which is why learning the CLI basics is recommended. You can mix both approaches: for example, use CLI for advanced operations and GUI for quick visual inspections of diffs. The choice comes down to preference – the underlying result in the Git repository is the same.

Tip: Even if you primarily use a GUI, try to familiarize yourself with Git terminology and a few core commands. This will help in communicating with collaborators and searching for help online, since

many solutions or discussions reference commands. Both the CLI and GUI are just interfaces to Git; use whichever makes you most productive while ensuring you understand the version control concepts.

6. Making Commits

Commits are the fundamental units of work in Git, so mastering them is key to effective version control. A **commit** represents a set of changes (additions, deletions, modifications) that you have bundled together with a message describing those changes. When you make a commit, you should aim to create a *logical, self-contained chunk* of work – for instance, "added a new function to normalize data" or "fixed typo in introduction section" rather than an unrelated grab-bag of changes. This makes your history easier to follow.

How to Commit Changes: The process is typically:

1. **Edit files** to make your changes.
2. **Stage the changes** with `git add`. This tells Git which changes you intend to include in the next commit. For example, `git add analysis.py` will stage changes in that file. You can stage multiple files or use `git add .` to stage everything (but be careful to only include intended changes).
3. **Commit** with `git commit`. If you use the `-m "message"` option, you provide the commit message inline; otherwise, Git will open an editor for you to type a multi-line message. In GitHub Desktop or other GUIs, staging is often done by check-marking files in a list of changed files, and committing is done by filling out a message field and clicking a Commit button.

When writing commit messages, be descriptive but concise. A common convention (especially in open-source) is to use a short summary in the first line (50 characters or less), followed by an empty line and then more details if needed. For example:

pgSQL

```
Add function to normalize dataset
This function scales each feature to a 0-1 range using min-max normalization.
It will be used before training the ML model. Added unit tests for the function.
```

The first line stands out in `git log` or GitHub, and the body (if present) can explain *why* the change was made or any additional context. Writing good commit messages is important for future you and others to understand the evolution of the project. In research projects, you might include references to experiment numbers or analysis steps in commit messages (e.g., "Re-run simulation with corrected parameter (exp#5)").

Commit Early, Commit Often: Don't be afraid to commit frequently. Each commit is cheap in Git (thanks to its snapshot mechanism) and acts as a safe restore point. If you're working on a long analysis, break it into multiple commits at logical breakpoints (e.g., "import initial dataset", then "implement outlier removal", then "add plot for results"). Fine-grained commits make it easier to track down when a bug was introduced using tools like `git bisect` (an advanced command), and they make code review simpler because each commit addresses a focused change. That said, avoid committing half-written code that doesn't run, unless you're working on a throwaway branch – each commit should ideally leave the project in a working state (this isn't a strict rule, but a good guideline especially on shared branches).

Tracking Changes Effectively: After each commit, you can use `git diff HEAD~1` (HEAD~1 means one commit ago) to review what you just committed, ensuring you didn't include anything unintended. To see a list of recent commits, use `git log --oneline --decorate --graph --all` – this shows a concise history with branches and commit IDs. If you realize you forgot to include a file in the last commit, you can still add it and use `git commit --amend` to add it to the previous commit (this rewrites that commit, so use amend only for local commits not yet pushed). Always double-check `git status` before committing, to see which changes are staged vs unstaged.

In summary, making commits in Git involves selecting changes to commit, writing a meaningful message, and executing the commit. Each commit becomes a permanent part of the project's history (unless you explicitly rewrite history, which is an advanced topic). By committing often with clear messages, you create a narrative of your project that greatly aids in understanding and reproducing your research process.

7. Forking and Cloning

Git's flexibility really shines in open-source and collaborative environments where you may want to contribute to someone else's project or use their code as a starting point. Two important actions in this context are **forking** and **cloning**:

- **Cloning a repository:** Cloning is the act of making a local copy of a remote repository. When you run `git clone <URL>`, Git retrieves all the data (the full history, all branches, commits, etc.) from the remote and creates a new directory on your computer with that content. After a clone, you have a local repository that is connected to the original remote (by default named "origin"). Cloning is common when you want to start working with an existing project – for example, if a colleague has put a research software tool on GitHub, you would clone it to run or modify it on your machine. Cloning does not impact the original repository at all; it's a read-only action as far as the source is concerned. You can clone your own repos or others' public repos freely. (For private repos, you need access permission to clone)

- **Forking a repository:** A fork is a copy of someone else's repository that lives on *your* GitHub account (or GitLab, etc.). Think of it as "cloning on the server side." On GitHub, when you fork a repository, GitHub creates a new repository under your account that starts as an exact clone of the original ("upstream") repository. This fork on your account is now independent – you have full control over it, meaning you can push changes to *your* fork without affecting the upstream. Forking is typically done by clicking the **"Fork"** button on a repository's page on GitHub (you need to be logged in). Once forked, you'll have your own copy on GitHub, e.g., `YourUsername/original-project`. You would then **clone your fork** to your local machine to work on it (so you're cloning from your own GitHub repo).

Forking is most useful when you want to contribute to a project you don't own. The workflow is: fork the repository (creating your copy on GitHub), clone your fork locally, make changes and commits in your local repo, push back to your fork on GitHub, and then create a **pull request** from your fork asking the original repository to pull in your changes. This is how contributions to open-source projects on GitHub are commonly managed. The maintainers of the original project can review your pull request and decide whether to merge it. Meanwhile, your fork remains yours – if the upstream maintainers don't accept your changes, you still have them in your fork.

To clarify the difference: **cloning** is about copying a repository *to your computer*, while **forking** is about copying a repository *to your GitHub account*. You often do both in sequence (fork on GitHub, then clone the fork). If you have write access to a repository (e.g., it's your own or you're a collaborator), you don't need to fork it – you can clone it directly and push to it. Forking is specifically for when you *don't* have permission to push to the original repo (common in open source).

Contributing to Open Source via Forks: Suppose you find a bug in someone else's published analysis code on GitHub. You can fork their repository, clone it, fix the bug in your local copy, commit the fix, push the commit to your fork on GitHub, and then open a pull request back to the original repo describing the fix. The project owners can then merge your contribution if they agree with it. All of this is made easy by Git and GitHub, without you ever emailing patches or worrying about conflicting file versions. Your fork is also a full Git repository on its own – if the original project disappears, your fork still has the complete history. In practice, it's good to keep your fork updated with the original project's changes. You can do this by adding the original repo as a second remote (often named "upstream") in your local repo (`git remote add upstream <original-repo-URL>`) and periodically fetching and merging from it, or by using GitHub's online interface to sync forks.

For researchers, forking is useful not only for contributing to others' projects but also for creating your own variation of a project. For example, you might fork a simulation software to adapt it for your specific experiment. Your fork starts identical to the source, but then you can diverge to tailor it to your needs, all while the original remains unaffected.

In summary, **clone** whenever you need a local copy of any repo, and **fork** on GitHub when you want your own server-side copy of someone else's repo (usually to contribute or modify it independently). Forking and cloning are core to open-source collaboration, enabling distributed development where each contributor works on their own copy and then merges contributions through pull requests.

8. Pushing and Pulling Changes

When working with a remote repository (like one on GitHub), the way to sync changes between your local repo and the remote is via **push** and **pull** operations:

- **Push:** Uploads your new commits from your local repository up to the remote repository. If you have a series of commits on your local `main` branch that are not yet on GitHub, `git push origin main` will send those commits to the `main` branch on the remote (here "origin" refers to the remote name, which by default is the name given to the repository you cloned from). Pushing is how you publish your work for others to see or use. For example, after you've implemented a new feature and committed it, you push to share that feature with your collaborators on GitHub. Only changes that are committed will be pushed (your working directory or staged but uncommitted changes are not affected by push). It's a good habit to push regularly so that your remote is up-to-date with your local work, especially before and after collaborating or switching computers.

- **Pull:** Downloads and integrates commits from the remote repository into your local repository. `git pull` is essentially a shorthand for two steps: first `git fetch` (which downloads new commits from the remote to your local *git database* without altering your working files), and then `git merge` (which merges those fetched commits into your current branch). If someone else has pushed changes to GitHub while you were working, you need to pull to get those changes into your local repo. For instance, if a collaborator added a new data preprocessing function and pushed it, when you do `git pull origin main`, you'll receive that commit and your local files will update to include the new function. If your local work and the pulled changes touched different parts of the code, Git will merge automatically. If there were conflicting changes (e.g., you and someone else edited the same line in different ways), Git will pause the merge and mark the conflict in the files for you to resolve manually, then commit a merge commit.

Think of **push** as "send my changes up" and **pull** as "bring others' changes down." A common workflow at the start of your day might be: *pull* the latest changes from the remote (to ensure you're up to date), then do your work and make commits, and finally *push* your commits to the remote. This ensures a smooth integration with your team. If you try to push but someone has pushed in the meantime such that your local is behind, Git will usually prevent the push (to avoid overwrite). In that case, you'll be prompted to pull first, merge their changes, then push again. This scenario often arises if you and a collaborator both commit to `main`. One solution is to use

separate branches for each person or feature and then merge via pull requests to minimize direct conflicts on the same branch.

Fetch vs Pull: It's worth noting the distinction between `git fetch` and `git pull`. `fetch` only downloads the new commits, but doesn't merge them. This is useful if you want to see what others have pushed before integrating. You could fetch and then examine the changes (with `git log` or GitHub UI) to decide if you're ready to merge. `git pull` does fetch+merge in one step for convenience. In GitHub Desktop, clicking "Fetch origin" will perform a fetch and indicate whether there are new changes on the remote. If yes, it might show a button to "Pull" those changes.

Pushing branches: By default, pushing with `git push origin main` updates the main branch on remote. If you created a new branch (say, `analysis-improvements`), you can push it with `git push origin analysis-improvements`. The first push of a new branch will create that branch on the remote. This is how you publish a feature branch for others to see or for opening a pull request on GitHub. If you're done with a feature branch and have merged it, you (or the repo maintainer) might delete the branch on GitHub to keep things tidy (the commits remain part of history and on main after merge).

Pulling and conflicts: When you pull, if Git encounters conflicting changes, it will notify you with markers in the affected files (lines like `<<<<<< HEAD` and `======` and `>>>>>>> otherbranch`). You will need to edit those files to reconcile the differences (keep the correct parts from each side or modify as needed), then mark the conflict as resolved by staging the file, and finally commit the merged result. GitHub also has a web editor to resolve simple conflicts during pull requests. To avoid complex conflicts, communicate with collaborators about which parts of the code each person will work on, or use separate branches and merge changes in a controlled manner.

Authentication: Note that pushing to a remote typically requires authentication. With GitHub, you may use HTTPS (which requires entering your username/password or a Personal Access Token on push) or SSH (which requires setting up an SSH key pair). If using GitHub CLI or Desktop, authentication is handled for you after login. Modern Git for Windows also can store credentials securely. If you encounter authentication issues, consider using a PAT (personal access token) in place of your password for HTTPS, as GitHub no longer accepts account passwords for Git operations.

In summary, **`git push`** and **`git pull`** are your primary commands for collaborating via a remote repository. Push shares your work; pull brings in others' work. Together, they keep the local and remote repositories in sync. Always remember to pull before starting new work (to avoid building on an outdated base) and pull before pushing if your push is rejected due to remote changes. This "sync often" approach will help prevent big merge surprises and keep everyone on the team up to date with the project's latest state.

9. README and Licensing

Every repository should include a **README file** and a **license** if you plan to share the code. These files are critical for making your project understandable and usable by others (and by you, when you come back to it in the future!).

README File: A README (often named `README.md` for Markdown format) is a text document that introduces and explains your project. It typically appears on the front page of your repository on GitHub. A good README answers questions like *"What does this project do? How do I use it? How do I install or run it? Who wrote it? How can I cite or contact the author?"*. In short, it provides information needed to understand the project's purpose and how to get started. Writing a README might seem optional for your own private code, but it's extremely helpful once you share the code or even for yourself after some time passes.

A common structure for a README includes:

- **Project Title** and a short description (one or two sentences explaining the project or repository).
- **Background/Motivation:** If it's research code, mention the context (e.g., "Code for analyzing genomic data for paper XYZ").
- **Installation or Requirements:** What does someone need to run this code? List dependencies, required Python/R packages or MATLAB toolboxes, etc., and how to install them. If it's a compiled program, provide build instructions.
- **Usage:** Show how to use the code. This could be example command-line calls, or sample code snippets if it's a library, or instructions like "run `analysis.R` to reproduce the figures." Provide examples with expected inputs/outputs.
- **Results:** Optionally, if this is tied to a research output, mention what results or files are generated and perhaps how they relate to the publication (if any).
- **Contributing:** If you welcome contributions, note how others can contribute (pull requests, contacting you, etc.). For personal research code, you might omit this, but for lab projects or collaborative tools, it's nice to include guidelines.
- **License:** You should mention the license in the README (e.g., "Licensed under MIT, see LICENSE file for details") especially if you want to make it obvious how the code can be used.
- **Acknowledgements or References:** Credit any funding sources, collaborators, or references to academic papers or datasets. If the code accompanies a paper, cite the paper or provide a DOI.

Keep the README relatively concise; if you have extensive documentation, you can link to separate docs or a wiki. But the README should have enough for a newcomer to understand the project and try it out. Remember, the README is often the first thing others see – it's like the cover page and abstract for your code.

License: Choosing a license for your code determines how others are permitted to use it. If you want your code to be open source (which is common and encouraged in academia for transparency and reuse), you should include an explicit license file (commonly named `LICENSE`

or `LICENSE.txt`) in your repository. Without a license, the default is “all rights reserved” – meaning no one else can legally use, copy, or modify your code (in many jurisdictions). That’s probably not what you intend if you put it on GitHub. Including a license grants permissions to users under certain conditions.

Common licenses and their characteristics:

- **MIT License:** A short, permissive license. It basically says anyone can use, copy, modify, and distribute your code, as long as they include the license notice. It does *not* require derivative works to open-source their code. This is a good choice if you want maximum reuse and don’t mind your code being used in closed-source projects. Many researchers choose MIT to encourage broad adoption.
- **GNU GPL (General Public License):** A copyleft license. It allows others to use and modify your code, but if they distribute a modified version (even as part of another project), they *must* also release their source code under the GPL. This ensures that improvements remain open-source, but it can discourage use in proprietary software. GPL is a stronger stance to guarantee openness (sometimes preferred if you’re releasing software and want to ensure it stays free for all users).
- **Apache 2.0:** Another permissive license, similar to MIT, but longer and with provisions related to patents. Good for projects that might involve patented algorithms.
- **BSD 2-Clause/3-Clause:** Permissive like MIT, just different wording (MIT and BSD are functionally similar for most purposes).
- **Creative Commons Licenses:** *Not typically used for software*. CC licenses are more for documents, data, or other creative works. For code, stick to software licenses like the above.

For most research code, either MIT (permissive) or GPL (copyleft) are popular choices, but you should choose based on how you want your work to be used. GitHub has a site called *choosealicense.com* that can help you pick a license by answering a few questions. Also, if your code uses other libraries, ensure your license is compatible with them.

To add a license, you can often use GitHub’s interface: on the repository page, click “Add file” > “Create new file”, name it “LICENSE”, and GitHub will even offer a license template picker. Or you can copy the text from choosealicense.com for the license you want. Once added, the license type will show up near the top of your repo page on GitHub (e.g., “MIT License”).

In your README, mention the license to make it clear (“This project is licensed under the MIT License – see the LICENSE file for details.”).

Why License? Beyond the legal reasons, licensing is about giving permission explicitly. If you want your fellow researchers to confidently reuse your code in their own work, a license clarifies that they can. Some journals and conferences also require that code/data be released under an open license when you publish. Even for your own use, if you leave a project and later join a new one, having a clear license on the old code can simplify questions of reuse.

Documentation and Citation: Along with README and license, consider adding citation information if applicable. Many projects include a `CITATION.cff` file or just instructions in the README on how to cite the code or associated paper. This is not about Git, but it's a good practice in research code sharing – it increases the chances you get academic credit when others use your code.

In summary, a README is your project's welcome guide (what it does, how to use it) and a license tells others what they're allowed to do with your code. These files turn your repository from just a code dump into a usable and shareable project. They are especially important when you release your code publicly to comply with open-science best practices and to ensure your work can be built upon legally and easily.

10. Repository Visibility and Access

When creating a repository on platforms like GitHub, you have control over who can see and interact with your code. The two primary visibility settings are **public** and **private**, and each has its uses in research contexts:

- **Public Repository:** This means *anyone* on the internet can see the repository contents. On GitHub, a public repo can be viewed and forked by others (if it has a license allowing reuse). Public repos are great for open-source projects, code accompanying publications, or any scenario where you want to share your work widely. They enable community contributions (via issues or pull requests) and let others learn from or build on your code. If you publish a paper and want to release the code, making the repo public (with an appropriate license and documentation) is how you'd do that. Keep in mind that publishing a repo publicly should be done intentionally – ensure you're not exposing sensitive data or credentials. GitHub provides free unlimited public repositories, so there's no barrier to open-sourcing your research code if appropriate.

- **Private Repository:** This is only accessible to you and people you explicitly give access to. If a repo is private, it won't show up for others and they cannot fork or view the code. This is useful for early-stage research or internal projects where you're not ready (or willing) to share the code yet. For example, if you're working on a paper and the code is rough or contains confidential information (like participant data or proprietary algorithms), keep it private. Later, you might choose to make it public (you can change a repository from private to public in settings). GitHub Free allows unlimited private repos and you can add collaborators to them at no cost.

Changing Visibility: You can toggle a GitHub repository's visibility (from private to public or vice versa) in the repository's **Settings** under **Danger Zone** ("Change visibility"). Making a private repo public is common when you're ready to release. Going public will notify collaborators and watchers, and GitHub will detach any public forks that existed (usually only relevant if it was public then made private). Making a public repo private will revoke general access; if it had forks, those

remain as separate repos (they won't be made private automatically). Be cautious: if your project was public and others have already forked it, those forks stay out there even if you go private later.

Collaborator Access: For both private and public repos, you can invite collaborators (specific people) to have direct access. On a personal GitHub account repository (not in an organization), collaborators you add essentially get push access – they can clone, push, and manage issues/pull requests as if it's their own repo. They cannot change the repo's settings or delete it (those are admin rights reserved for the owner). You add collaborators via **Settings > Manage access**, and you need to know the person's GitHub username or email. Once they accept the invite, they can treat the repo as if it were theirs in terms of contributing. In private repos, *only* collaborators (and the owner) can see or push to the repo. In public repos, anyone can see, but only collaborators can push (others would have to fork and submit pull requests for contributions).

For more granular permissions or larger teams, GitHub Organizations are used, where you can have roles like Read, Triage, Write, Maintain, Admin for members on a repo. But for a personal or small project, just adding collaborators by default gives them write access (there's no read-only collaborator for personal repositories – it's either full access or none).

When to use Private vs Public: It often depends on the project stage:

- During initial development or if the code is a mess or contains hardcoded paths or secrets, you might keep it private.
- The moment you want feedback, help, or to cite the code, consider making it public. Even early in a project, making it public can be fine as long as you're okay with others seeing it. You don't have to advertise it until it's ready, but having it public early means any work (commits, issues) is already accessible if needed.
- Some researchers prefer developing in private and only open-sourcing upon publication to avoid being scooped or judged on incomplete work. This is a personal or lab policy decision.
- If working with human subjects data or other sensitive info, **do not put that data in a public repo** (or even in a private repo unless all collaborators are allowed to see it). Keep data separate and use appropriate data repositories for sharing anonymized datasets if required.

Managing Permissions: Aside from adding collaborators, you might consider:

- **Branch protection rules (in Settings):** For public repos especially, you can protect the `main` branch to require pull requests for changes (so no one can push directly to main without review). This is more relevant in team settings or open-source projects.
- **Teams (if in an organization):** If your research group has a GitHub organization, you can manage teams and give them access to certain repositories collectively.
- **Access expiration:** If you added a collaborator temporarily, remember to remove them if they no longer need access (especially important for private repos).

- In academic collaborations, all members of a project might just share a private repo and that's it – which is perfectly fine.

Visibility and Citations: Note that if you intend to cite your code repository in a paper, it generally needs to be accessible to readers. A private repo can't be accessed by peer reviewers or readers (unless you invite them, which is cumbersome). Often the solution is to make the repo public upon submission or acceptance of the paper, or archive a specific version of the repo on a service like Zenodo which provides a DOI. But that's beyond Git itself – just a consideration that public code is better for transparency and credit.

Security considerations: If your repo is public, be mindful not to commit passwords, API keys, or sensitive data. GitHub has scanners that alert on some secrets, but not all. If a private repo accidentally contains a secret and you plan to open-source it, remove those secrets (and ideally rotate them) before making it public – history can expose them otherwise.

In summary, **public vs private** is about who can see the repo. Use private for restricted access and early development, and public for open collaboration and release. Git and GitHub give you the flexibility to start private and go public later. And with collaborator management, you can tailor who has write access or view access as needed. This way, you maintain control over your research code while still leveraging the collaboration features of Git when you need them.

11. Useful References

- Slideshare: Introduction to Git —overview of Git.

Version Control Basics
- The Turing Way: Version Control — Overview of version control in research.
- NCBI: Version Control Systems in Research — Benefits of VCS in collaborative research.
- Git Reference Basics — Git fundamental concepts.

Installing Git
- Git SCM: Installing Git — Official Git installation guide.
- Atlassian: Install Git — Step-by-step Git installation guide.

Repositories and Branching
- GitHub Docs: Creating Repositories — GitHub repository setup.
- GitKraken: What is a Git Repository — Git repository basics.
- GitKraken: Git Checkout & Branches — Explanation of checkout and branches.
- Git Tower: Remote Repositories (Part 1) — Introduction to remotes.
- Git Tower: Remote Repositories (Part 2) — Working with files remotely.

Forking, Cloning, and Collaboration
- GitHub Docs: Inviting Collaborators — Manage repository collaborators.
- Fluxnet Docs: Forking and Cloning — Guide on forking and cloning.
- Fluxnet Docs: Pull Requests — Learn to contribute back via PR.

Syncing and Pushing
- Atlassian: Git Push — Pushing code to remote repositories.

Project Documentation
- Make a README — How to write good READMEs.

Licensing
- StackOverflow: MIT vs GPL License — Quick comparison of common open-source licenses.