Version control using Git and GitHub

Git is a source control management tool that tracks changes to files over time and allows users to manage and document the evolution of a project. GitHub is a web-based platform that hosts Git repositories and provides additional tools for collaboration, automation, documentation, and issue tracking.

While this protocol cannot be exhaustive, it introduces the most basic operations such as setting up a repository, recording file updates through commits, working with branches and forks, and merging contributions into the main version via pull requests. It also provides recommendations on commit practices, file embedding, privacy, and project sharing that will benefit research groups who want to adopt Git and GitHub not only for code, but also for notebooks, documentation, protocols, and other project deliverables.

Risk assessment

– Files uploaded to external servers may unintentionally expose sensitive information such as personal data, credentials, or API tokens.

▷ DO NOT track sensitive data; use `.gitignore` to exclude these files

▷ Avoid destructive flags such as `--force`, `--force-with-lease`, and `--hard` unless you understand that they rewrite history or discard changes. These operations should only be used deliberately by the repository owner and should never be used on shared production branches without agreement.

Reviewed: May 12, 2026

Procedures

⊞ **Optional:** *Creating a GitHub account*

(1.) Create a GitHub account at https://github.com/.

> ***Hint:*** Choose a short, professional username and a strong password. This password is used *only* for logging into the GitHub web interface. To work with repositories from your local machine, use fine-grained personal access tokens (PATs) instead. These are scoped credentials, specifying what actions can be performed and which repositories they apply to.

> ***This is why:*** Other remote authentication methods like SSH keys or OAuth are supported, but fine-grained PATs provide better security for collaborative projects.

›› *Working with Git on a local machine*

(1.) Confirm Git is installed by running `git --version` in the terminal. If not, install Git using your system's package manager or download it from https://git-scm.com/downloads.

(2.) *Optional:* Execute the following commands to associate commits with your GitHub identity. ⊞

```
git config --global init.defaultBranch main
git config --global user.name my-gh-name
git config --global user.email id+my-gh-name@users.noreply.github.com
```

> ***This is why:*** The first command sets "main" as the default branch name for new repositories. The remaining commands set username and email address which are included in each commit. If you want privacy, GitHub provides a noreply alias for your account. Look for "Keep my email addresses private" in your GitHub account settings.

(3.) Navigate to the project folder, then initialize a new Git repository with

```
cd /path/to/my-local-repository
git init
```

> ***This is why:*** This creates a hidden `.git` directory within the current folder to track version history. You can also provide a positional argument `git init my-folder` to create and initialize Git on a subfolder in the current directory.

> ***Hint:*** You can name the project folder freely, but using a short, lowercase name with dashes instead of spaces is recommended.

(4.) Create or modify files in the folder.

(5.) Inspect which files have changed.

```
git status
git diff
```

Add ("stage") files or part of a file to include in the next snapshot:

```
git add .
git add my-file
git add -p my-file
```

(6.) *Optional:* To prevent certain files from being tracked, create a `.gitignore` file in the repository root and list patterns for files or directories that should be excluded on individual lines.

> ***Hint:*** Use a command line editor such as `nano .gitignore` to create or edit this file. Typical patterns include, for example, `*.log`, `__pycache__/`, or `my_patient_data.csv`. Once added, matching files should no longer appear on `git status`.

(7.) Create a versioned snapshot of the staged files.

```
git commit -m "Commit message"
```

> ***Hint:*** A good commit message is a short, present-tense summary of its consequences such as "Add scripts for data loading from PubMed", "Change default font size for printing", "Fix typo in README.md".

(8.) Repeat editing, staging, and committing as needed. Use `git log` to view the commit history.

Advanced Git operations

(1.) *Optional:* Use `git stash` to temporarily save all tracked changes (both staged and unstaged) without committing. You can recover the most recent stash later using `git stash pop`.

> ***Hint:*** This is useful if you need to switch branches or pull updates but aren't ready to commit your current changes. Untracked files and ignored files are not stashed by default unless explicitly included. You can use `git stash list` to view all saved stashes, and `git stash drop stash@{n}` to delete one.

(2.) *Optional:* To recover a previous state *after committing*, execute:

- `git reset --soft HEAD~1` to undo the last commit without deleting changes.

  > ***This is why:*** This moves the branch pointer back by one commit and stages the changes again, allowing you to amend them. No changes will be undone and no files will be removed.

- `git reset --mixed HEAD~1` to undo the last commit and unstage the changes.

  > ***This is why:*** The changes will remain in your working directory but not be staged.

- `git reset --hard HEAD~1` to discard the last commit and all its changes completely.

  > ***Critical:*** Any local changes that were part of that commit will be permanently discarded and cannot be restored with Git.

  > ***Hint:*** A safer option can be `git reset --keep HEAD~1` which only resets the files which are different between the current head and the given commit. It aborts the reset if there are one or more uncommitted changes in other files.

(3.) *Optional:* To undo changes in the working directory *before committing*, run

- `git reset my-file` to remove the file from the staging area and add further edits.
- `git checkout -- my-file` to discard all unsaved changes in the file.

  > ***Critical:*** Only use this if you are sure you want to go back to the last committed version of that file.

(4.) *Optional:* At important milestones such as manuscript submissions, dataset releases, or publication, add semantic versioning such as `git tag v1.0.0`.

> ***Resource:*** Semantic Versioning (https://semver.org/) differentiates "major", "minor", and "patch" versions. Major versions may be incompatible, while minor versions add functionality or bug fixes in a backward-compatible manner respectively.

›› *Working with branches and merging changes*

(1.) View available branches with `git branch`. The current branch is marked with an asterisk.

> ***Note:*** The default branch is usually named "main", but it may be called "master" in older Git installations.

(2.) Create a new branch and/or switch to it.

```
git branch my-feature
git checkout my-feature
```

> ***This is why:*** Branches can be used to work on experimental changes or additions without affecting the main history. If pushed to a remote repository, they can also backup your work in progress. You may want the remote repository to be private.

> ***Hint:*** To push the current branch only if present in the remote, set `git config push.default simple`. You may want this setup when working on public remote repositories.

(3.) Edit, stage, and commit changes on the new branch as usual.

(4.) Update your new branch with the latest changes from the main branch.

```
git fetch origin
git merge origin/main
```

(5.) When done, switch back to the main branch. After you merge your feature branch, you can delete it.

```
git checkout main
git merge my-feature
git branch -d my-feature
```

> ***Hint:*** This removes the branch label, not the commits. It's safe to delete a branch after merging and will keep the repository readable. You can always create a new branch later for the next change.

›› *Working with remote repositories on GitHub*

(1.) Create a new repository on GitHub using the web interface.

- Make the repository *private*. You can publish the repository later.
- Initialize the repository with a `README.md` and a `.gitignore` file.

> ***Critical:*** Before first push, confirm that `.gitignore` excludes any files that contain private data, credentials, personal identifiers, or unpublished results. GitHub is public by default and search engines can index it quickly.

- If institutional guidelines do not otherwise specify, choose a permissive license for your work:

License	Attribution	Share alike	Commercial use	Patent clause	Recommended use
CC0-1.0	No	No	Yes	No	Public domain datasets; templates; code snippets
CC-BY-4.0	Yes	No	Yes	No	Protocols, educational content, documentation
MIT	Yes	No	Yes	No	Code for figures, small analysis scripts
Apache 2.0	Yes	No	Yes	Yes	Reusable libraries, substantial academic/industry software
GPLv3	Yes	Yes	Yes	Yes	Full analysis pipelines where openness of all modifications is required

Please do not add a license to the repository if you are unsure. Software and documentation licenses are not revocable once published and may affect collaborators and reuse policies.

> ***Hint:*** For software and code, Creative Commons (CC) licenses are not suitable. Instead, use a software license such as MIT, Apache 2.0, or GPLv3. If your repository includes or modifies code licensed under GPL, your entire repository must adopt a GPL-compatible license such as GPL, MIT, or BSD. Apache 2.0 is *not* compatible with GPL.

(2.) *Critical:* Generate a fine-grained personal access token (PAT) for remote authentication:

- Go to https://github.com/settings/personal-access-tokens/ and select "Generate new token".
- Provide a token name and expiration date.
- Choose "All repositories" to allow access to future repositories, or explicitly select existing ones.

  Hint: To get a token for a collaboration project, the repository owner must have first added you as a collaborator.

- Under "Repository permissions", set "Contents" to "Read and write".
- Click "Generate token" and copy the generated string to your clipboard.

  Hint: After the first clone or push, Git may prompt for your GitHub credentials. Use your username and paste your fine-grained PAT as the password; Git is typically configured to securely store your token using the system password manager via `git config --global credential.helper manager` (Windows) or `osxkeychain` (macOS).

(3.) Create a copy of the remote repository on your local machine, or link a local repository to the remote, by executing one of the following commands:

```
git clone gh-repo-url.git
git remote add origin gh-repo-url.git
```

(4.) To retrieve changes from the remote repository, execute:

```
git fetch origin
git merge origin/main
```

Hint: This two-step method is safer for avoiding unintended overwrites or merge conflicts when local file changes exist. `git pull origin main` is faster but may create automatic merge commits or conflicts if your local branch has diverged.

(5.) Upload your committed local changes to the remote repository with

```
git push origin main
```

›› *Collaborating on remote repositories with GitHub*

(1.) As repository owner, add collaborators to your repository; set privileges as appropriate.

Hint: For public repositories, users can also contribute via pull requests without being added explicitly.

(2.) *Optional:* As repository owner, create a development branch for the project once

```
git checkout -b dev
git push origin dev
```

(3.) As a collaborator, accept the invitation, pull the repository, and create a new branch for your edits to isolate changes and make reviewing easier.

```
git pull origin dev  # or "main" if no development branch exists
git checkout -b my-feature
git push origin my-feature
```

(4.) Edit, stage, commit, and push changes on your branch as usual.

This is why: Pushing will make your branch visible to other collaborators and allows you to create a pull request (PR).

Critical: Pull the latest changes regularly from the remote repository to keep your local repository up to date.

(5.) Once your branch is ready to be merged, create a pull request on GitHub:

- Navigate to the repository on GitHub, click "Compare & pull request".

  Hint: For production repositories, request a merge from the feature branch into the `dev` branch for active development first. Merge the `dev` branch to the `main` branch for stable releases.

- Respond to comments and update the pull request as needed.

  > ***Note:*** You can push additional commits to the same branch to update an open pull request.

- Once approved, the pull request will be merged into the development or the main branch.

[PGW+16]

›› *Linking repositories with submodules*

(1.) Use a submodule when part of one repository (private content, shared utilities, a vendored dependency) needs its own version history but should appear as a subdirectory of the parent. The parent pins a specific commit of the submodule, so updates are explicit and reproducible.

> ***This is why:*** A submodule is one repository nested inside another. The parent stores the submodule's commit SHA, not its files. Pulling the parent never silently updates the submodule.

(2.) Add a submodule to the parent repository.

```
git submodule add gh-other-repo-url.git submodule/path/in/parent
git commit -m "Add submodule/path/in/parent as submodule"
```

(3.) When others clone the parent, initialize submodules in the same step or after the fact:

```
git clone --recurse-submodules gh-parent-repo-url.git
git submodule update --init --recursive
```

> ***Note:*** The second form initializes submodules in an existing clone that was made without `--recurse-submodules`.

(4.) To update the submodule to a newer commit, update the pointer in the parent.

```
cd submodule/path/in/parent
git pull origin main
cd ..
git add submodule/path/in/parent
git commit -m "Bump submodule pointer"
```

> ***Critical:*** Editing files inside the submodule does *not* automatically update the parent's pointer. You must commit twice: once inside the submodule, once in the parent.

(5.) *Optional:* To remove a submodule:

```
git rm path/in/parent
git commit -m "Remove submodule path/in/parent"
```

> ***Hint:*** This removes the entry from `.gitmodules` and the working tree. The submodule's own repository is untouched.

›› *Preserving file history when renaming or splitting repositories*

(1.) For renames *within* a single repository, use `git mv` rather than a plain shell `mv` followed by `git add`.

> ***Hint:*** Git stores content, not paths. `git mv` stages a rename automically; a plain `mv` registers as a delete + add, which Git can still detect as a rename if content similarity is high enough, but the intent is less explicit.

(2.) Confirm Git detected the rename.

```
git status              # "renamed: old -> new"
git log --follow new    # history across the rename
```

> ***This is why:*** By default, `git log` stops at the rename. The `--follow` flag traces history through it.

(3.) To move files *between* repositories while preserving history, `git mv` is not enough — it only operates within one repository. Use `git filter-repo` to extract the relevant history into a portable form, then merge it into the target repository.

> ***Resource:*** Workflows are more invovled, consult https://github.com/newren/git-filter-repo.

T › ***Repository transfer***

(1.) Audit the repository.

- No pending branches, unmerged pull requests, or unpublished changes?
- No dependencies on private data or credentials?

(2.) Finalize `README.md` to reflect the latest project title, description, and status of the project.

> ***Hint:*** Highlight key contributors, funding sources, dependencies on software versions, datasets, and analysis pipelines.

(3.) Create a final release tag so there is a citable, frozen snapshot of the code at time of handoff.

```
git tag v1.0.0-final
git push origin v1.0.0-final
```

(4.) *Optional:* For published work, link the repository to Zenodo.

> ***Resource:*** The repository is then stored safely for the future in CERN's Data Centre (https://zenodo.org/) for as long as CERN exists. Every upload is assigned a digital object identifier (DOI), making it citable and trackable.

(5.) *Critical:* If the repository is associated with a personal GitHub account, transfer the repository to a designated maintainer, the lab's GitHub account or the institutional GitHub organization. Ensure a maintainer is assigned for substantial software projects.

> ***Note:*** The target organization must be configured to accept outside transfers.

(6.) Remove the departing member's write/admin access after transfer is complete.

(7.) Archive the repository if it is no longer maintained. Archived repositories are read-only.

(8.) Add the repository title, archive status, and DOI to the lab's knowledgebase or data inventory.

List of references

Y. Perez-Riverol, L. Gatto, R. Wang, T. Sachsenberg, J. Uszkoreit, F. da Veiga Leprevost, C. Fufezan, T. Ternent, S.J. Eglen, D.S. Katz *et al.*, *PLoS Comput. Biol.* **12**(7), e1004947 (2016).

Change log

2025-07-26	Benjamin C. Buchmuller	Initial commit.
2025-10-25	Benjamin C. Buchmuller	Clarified branch/merge workflow and revised code.
2025-10-31	Benjamin C. Buchmuller	Clarified credential manager setup.
2026-05-12	Benjamin C. Buchmuller	Add procedures for submodules and rename history preservation.