Setting Up Git for Data Science Workflows

Ronald G. Thomas

2026-02-11

Figure 1: Git logo and branching diagram representing version control workflows

Version control is the foundation of every reproducible research workflow.

Introduction

I did not really know how much damage a missing backup could cause until I accidentally overwrote three days of analysis work on an Alzheimer's disease dataset. The file was gone, the changes were irreversible, and the only copy lived on a USB drive I had left at the office. That experience convinced me that version control was not optional for data science—it was essential.

Git is the most widely adopted version control system in both software engineering and data science. It tracks every change to every file, allows branching and merging of parallel work, and provides a complete history of a project's evolution. Combined with GitHub as a remote hosting service, Git enables collaboration, code review, and reproducibility at a level that manual file management simply cannot match.

We document a path from "I'll just save copies" to a reliable Git-based workflow. The post covers installation, basic commands, branching, collaboration, and the commit message practices that

proved most valuable. This is written as a learner, not as an expert, and corrections or suggestions from readers who have found better approaches are welcome.

More formally, we document the version-control concern that cuts across multiple layers of the Workflow Construct described in post 15. Git is not itself a layer; it is the mechanism by which the file-system layer's snapshots, the dotfiles repository's source of truth, the project compendium's history, and the cloud layer's deployments are all coordinated. This post documents Git's configuration as the baseline that every other layer assumes.

Motivations

- I lost critical analysis files more than once because I relied on manual backups and naming conventions like `analysis_v3_FINAL_FINAL.R`.
- I needed a way to experiment with new modeling approaches without risking the working version of my code.
- Collaborating with team members on shared analysis projects required a system more robust than emailing files back and forth.
- I wanted a clear, timestamped record of every change I made to a project, so that I could trace when and why a result changed.
- I was increasingly working across multiple machines (office workstation, laptop, server) and needed a single source of truth for every project.
- The biostatistics community has been moving toward reproducible research standards, and Git is a core component of that movement (Bryan, 2018).

Objectives

1. Install and configure Git on macOS or Linux, including identity setup and GitHub authentication through the `gh` command-line tool.
2. Learn the core Git workflow: staging, committing, pushing, and pulling changes.
3. Understand branching and merging for safe experimentation and collaboration.
4. Adopt a plain, imperative-mood commit message convention.

I am documenting my learning process here. If you spot errors or have better approaches, please let me know in the Let's Connect section at the end.

Figure 2: A single sheet of carbon transfer paper laid over a blank notepad on a wooden desk, with a fountain pen resting beside it, suggesting an exact, traceable duplicate being made.

GitHub provides the remote hosting layer that makes Git-based collaboration practical for research teams.

Prerequisites and Setup

This tutorial assumes a macOS or Linux environment. The commands shown use a standard terminal (Bash or Zsh). No prior Git experience is required, but familiarity with basic command-line navigation (`cd`, `ls`, `mkdir`) will be helpful.

Required software:

- Git (version 2.30 or later recommended)
- A terminal emulator (Terminal.app, iTerm2, Kitty, or similar)
- A GitHub account (free tier is sufficient)
- The GitHub CLI (`gh`), used for authentication and for the collaboration steps later in this post
- A text editor (Vim, Neovim, VS Code, or RStudio)

Installing Git on macOS:

```
xcode-select --install
```

This installs the Apple Command Line Tools, which include Git. To verify the installation:

```
git --version
```

Installing Git on Linux (Debian/Ubuntu):

```
sudo apt update && sudo apt install git
```

Installing the GitHub CLI:

```
brew install gh
```

On Linux, follow the install instructions at cli.github.com. The `gh` tool handles GitHub authentication and, later in this post, the collaboration workflow (creating repositories, opening pull requests) without leaving the terminal.

What is Git?

Git is a distributed version control system. In plain terms, it is a tool that records every change made to a set of files, stores those changes in a structured history, and allows retrieval of any previous version of any file at any time. Think of it as an unlimited "undo" system that also tracks who made each change and why.

The "distributed" part means that every collaborator has a complete copy of the project history on their own machine. There is no single server that must be online for work to continue. GitHub, GitLab, and Bitbucket serve as convenient central repositories for sharing work, but they are not strictly required for Git itself to function.

For a solo data scientist, Git provides a safety net: every experiment, every model iteration, and every data cleaning step is preserved. For a research team, it provides a coordination mechanism that eliminates the chaos of emailed files and shared drives.

Getting Started: Initial Configuration

Before using Git for the first time, one needs to set an identity. Git attaches a name and email to every commit.

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

I also recommend setting a default branch name and a preferred editor:

```
git config --global init.defaultBranch main
git config --global core.editor "vim"
```

To verify the configuration:

```
git config --list
```

Editing the Git configuration directly:

For advanced users, the repository-level configuration file at `.git/config` can be edited to adjust remote URLs, branch tracking, and other settings.

Authenticating with GitHub:

The simplest way to connect Git to a GitHub account is the `gh` CLI. Run the login command and follow the prompts:

```
gh auth login
```

Choose GitHub.com, choose HTTPS as the preferred protocol, and let `gh` open a browser to complete the one-time sign-in. Once this is done, `git push` and `git pull` against GitHub work without any further setup. This is the right default for most readers, since it avoids generating and registering a key pair.

Check that authentication worked:

```
gh auth status
```

SSH keys are a fine alternative once the Git workflow feels familiar. They avoid a browser step on each new machine but require generating and registering a key pair first:

```
ssh-keygen -t ed25519 -C "you@example.com"
ssh -T git@github.com
```

Creating Your First Repository

There are two common starting points: creating a new repository from scratch, or cloning an existing one from GitHub.

Creating a new local repository:

```
mkdir my-analysis
cd my-analysis
git init
```

This creates a hidden `.git` directory that stores the entire version history. The directory itself is now a Git repository.

Cloning an existing repository:

```
gh repo clone username/repo
```

This downloads the full project, including all history, and sets up the remote connection automatically. Plain `git clone https://github.com/username/repo.git` works the same way once `gh auth login` has been run.

I found that creating a repository on GitHub first (with a README and `.gitignore` for R) and then cloning it locally was the most reliable approach for new projects:

```
gh repo create my-analysis --private --clone
```

The Core Workflow: Add, Commit, Push

The daily Git workflow involves three steps: staging changes, committing them to history, and pushing them to a remote repository.

Step 1: Check the current state

```
git status
```

This shows which files have been modified, which are staged for commit, and which are untracked.

Step 2: Stage files for commit

```
git add analysis.R
git add data/clean_data.csv
```

Or, to stage all changed files:

```
git add .
```

I prefer staging files individually rather than using `git add .` because it forces me to review what I am committing. This prevents accidental inclusion of large data files or sensitive credentials.

Step 3: Commit with a message

```
git commit -m "Add linear regression analysis"
```

Each commit is a snapshot of the project at a specific point in time. The message should explain what changed and why.

Step 4: Push to the remote

```
git push origin main
```

This sends local commits to GitHub, making them available to collaborators and serving as an off-site backup.

Pulling changes from the remote:

```
git pull origin main
```

Always pull before starting new work to ensure the latest version is available.

Figure 3: A hand-bound accordion-fold paper file, partially expanded on a desk to reveal several distinct dated sections, each tabbed with a small colored marker.

Establishing a consistent workflow early prevents most version control headaches downstream.

Branching and Merging

Branches are one of Git's most powerful features. A branch is an independent line of development that diverges from the main project history. One can experiment freely on a branch without affecting the stable version of the code.

Creating and switching to a new branch:

```
git branch experiment
git checkout experiment
```

Or, in a single command:

```
git checkout -b experiment
```

Working on the branch:

Make your changes, stage, and commit as usual. The commits will only exist on the `experiment` branch.

```
git add .
git commit -m "Test random forest approach"
```

Merging back into main:

When the experiment succeeds, merge it into the main branch:

```
git checkout main
git merge experiment
```

Cleaning up:

Once merged, delete the branch to keep the repository tidy:

```
git branch -d experiment
```

I found branching indispensable for data science work. When I wanted to test whether a different variable selection strategy improved model performance, I created a branch, ran the analysis, compared results, and either merged or discarded the branch. The main branch always contained the working version.

Collaborating on GitHub

One of the most common collaboration scenarios involves inviting a team member to contribute to an existing repository. I encountered this when I had been working solo on an analysis of data from the public Alzheimer's Disease Neuroimaging Initiative (ADNI) repository, and the project had grown complex enough that I needed help.

Adding a collaborator (repository owner):

```
gh repo edit rgt47/x24 --add-collaborator rgt4748
```

This sends the invitation directly from the terminal. The collaborator gets a notification on GitHub and accepts it there, or from the terminal with `gh api` if they prefer, though the notification click is simpler for a one-time accept.

Cloning and contributing (collaborator):

```
gh repo clone rgt47/x24
cd x24
```

They create a branch for their work:

```
git checkout -b myedits
```

They make their changes (for example, editing `x24.Rmd`), then stage, commit, and push:

```
git add .
git commit -m "Update analysis title"
git push origin myedits
```

Creating a pull request:

```
gh pr create --title "Update analysis title" \
  --body "Corrects the title in x24.Rmd."
```

`gh` fills in the base branch and repository automatically from the current directory.

Reviewing and merging (repository owner):

```
gh pr list
gh pr diff 12
gh pr merge 12 --squash --delete-branch
```

`gh pr diff` shows the changes without opening a browser. Squash-merging keeps the `main` branch history readable, and `--delete-branch` cleans up the feature branch on the remote once it is merged. This workflow keeps the main branch clean and provides a record of every contribution.

Commit Message Best Practices

I initially treated commit messages as an afterthought, writing things like "fixed stuff" or "updates." Over time, I learned that well-crafted commit messages are one of the most valuable artifacts in a project's history.

Use the Imperative Mood

A valuable practice involves writing commit messages as though the commit, when applied, will perform a specific action. Construct each message so that it logically completes the sentence: "If applied, this commit will…"

Instead of:

```
git commit -m "Fixed the bug on the layout page"
```

Write:

```
git commit -m "Fix the bug on the layout page"
```

If this commit were applied, it would indeed fix the bug on the layout page. The imperative mood keeps messages consistent and action-oriented.

Explain "What" and "Why," Not "How"

Limiting commit messages to "what" and "why" creates concise yet informative explanations. Developers seeking the "how" can refer to the code itself. Instead of describing implementation details, highlight what was changed and the rationale behind it.

Optional: Prefixed Commit Conventions

Some projects prefix each commit message with a category, following the Angular project's convention:

- `feat:`: a new feature
- `fix:`: a bug fix
- `docs:`: documentation changes
- `style:`: formatting, no code change
- `refactor:`: code restructuring
- `test:`: adding or updating tests
- `chore:`: maintenance tasks

```
git commit -m "feat: Add cross-validation to model"
git commit -m "fix: Correct off-by-one in data merge"
```

I tried this for a while and found it added overhead without much payoff for a single-analyst project. My default now is the plain imperative style from the section above, with no prefix:

```
git commit -m "Add cross-validation to model"
git commit -m "Correct off-by-one in data merge"
```

A prefix convention is worth adopting for a team project where commits need to be filtered or grouped by category. For solo analysis work, plain imperative messages are simpler and just as informative.

Viewing History and Recovering Work

Git's history commands are essential for understanding how a project evolved and for recovering from mistakes.

Viewing the commit log:

```
git log
git log --oneline --graph
```

The `--oneline --graph` flags produce a compact, visual representation of the branch and merge history.

Viewing a file from a specific commit:

```
git show main:analysis.Rmd
```

This displays the file contents as they existed on the specified branch or commit, without modifying the working directory.

Restoring a file from another branch:

```
git checkout main -- data/results.csv
```

Recovering an earlier version of the entire project:

```
git checkout abc1234
```

Where `abc1234` is the commit hash. This produces a "detached HEAD" state where one can inspect the old version. To return to the current state:

```
git checkout main
```

Troubleshooting Common Issues

When working with Git, I encountered several recurring problems that required specific solutions.

Unrelated histories error:

When merging repositories that were initialized independently, Git may refuse with an "unrelated histories" error. The fix:

```
git pull --allow-unrelated-histories
```

Large files accidentally committed:

If a large data file is committed by mistake, it becomes part of the repository history even after deletion. Use `.gitignore` to prevent this:

```
*.csv
*.rds
data/raw/
```

Merge conflicts:

When two people edit the same lines of the same file, Git cannot automatically merge the changes. Open the conflicted file, look for the `<<<<<<<`, `=======`, and `>>>>>>>` markers, resolve the conflict manually, then stage and commit.

Authentication failures:

If `git push` fails with a permission error, check the `gh` login status first:

```
gh auth status
gh auth login
```

If using SSH instead, verify that the key is added to the agent:

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519
```

Verification

After installing and configuring Git, confirm that the setup is working correctly.

```
git --version
```

```
git config user.name && git config user.email
```

```
gh auth status
```

```
mkdir /tmp/test-repo && cd /tmp/test-repo && \
  git init && touch README.md && \
  git add . && git commit -m "test" && \
  git log --oneline
```

If `git --version` returns 2.30+, the config shows the correct name and email, `gh auth status` reports a logged-in account, and the test commit appears in the log, the setup is complete.

Daily Workflow

Task	Command
Check status	`git status`
Stage files	`git add analysis.R data/clean.csv`
Commit changes	`git commit -m "Add model"`
Push to remote	`git push origin main`

Task	Command
Pull latest	`git pull origin main`
Create branch	`git checkout -b experiment`
Merge branch	`git checkout main && git merge experiment`

Things to Watch Out For

1. **Never commit credentials or API keys.** Add `.env`, `.Renviron`, and credential files to `.gitignore` before the first commit. Once a secret is in Git history, removing it completely requires rewriting history.
2. **Avoid committing large binary files.** CSV files over 50 MB, RDS objects, and image datasets do not belong in Git. Use Git LFS or external storage for large data.
3. **Pull before pushing.** When collaborating, always `git pull` before starting new work. Pushing without pulling first leads to merge conflicts that could have been avoided.
4. **Do not force-push to shared branches.** Running `git push --force` on `main` rewrites history for all collaborators. Reserve force-push for personal branches only.
5. **Commit frequently, push regularly.** Small, focused commits are easier to review, easier to revert, and create a more useful history than large, infrequent commits.

Uninstall / Rollback

To remove Git configuration without uninstalling Git itself:

```
git config --global --unset user.name
git config --global --unset user.email
git config --global --unset init.defaultBranch
git config --global --unset core.editor
```

To sign out of the `gh` CLI:

```
gh auth logout
```

To remove SSH keys used for GitHub, if any were created:

```
rm ~/.ssh/id_ed25519 ~/.ssh/id_ed25519.pub
```

To remove Git entirely on macOS, uninstall the Command Line Tools. On Ubuntu:

```
sudo apt remove git
```

To remove version control from a single project, delete its `.git` directory:

```
rm -rf .git
```

This is irreversible and destroys all project history.

Figure 4: UCSD Geisel Library, a landmark of research and learning

Disciplined version control practices support the same rigor that characterizes scholarly research.

What Did We Learn?

Lessons Learned

Conceptual Understanding:

- Version control is not merely a backup system; it is a structured record of a project's intellectual history, capturing the reasoning behind every change.
- Branching transforms experimentation from a risky activity into a safe, reversible process, which is particularly valuable in exploratory data analysis.
- The distributed nature of Git means that every collaborator holds a complete copy of the project, eliminating single points of failure.
- Commit messages, when written with care, become a form of documentation that is often more useful than formal project notes.

Technical Skills:

- The core workflow (`add`, `commit`, `push`, `pull`) covers the vast majority of daily version control needs for a solo analyst.

- Authenticating with `gh auth login` over HTTPS eliminates password friction for regular use, and skips the SSH key setup step entirely.
- The `git log --oneline --graph` command provides a rapid overview of project history and branching structure.
- The `.gitignore` file is the first line of defense against accidentally committing sensitive or oversized files.

Gotchas and Pitfalls:

- Forgetting to add a `.gitignore` before the first commit can result in large data files entering the repository history permanently.
- Using `git add .` without reviewing staged files is a common source of accidental credential leaks.
- Merge conflicts in data files (CSV, JSON) are particularly difficult to resolve because the conflict markers corrupt the file format.
- Working directly on `main` instead of a branch removes the safety net that makes Git valuable for experimentation.

Limitations

- This guide covers only the command-line interface. GUI tools like GitKraken, Sourcetree, and the RStudio Git pane provide alternative workflows that some users may prefer.
- The collaboration scenario assumes a small team with direct repository access. Larger projects typically use fork-based workflows, which are not covered here.
- Git tracks text files effectively but is poorly suited for binary files such as images, compiled documents, and large datasets.
- This tutorial does not cover Git internals (the object model, packfiles, or the reflog), which are important for advanced troubleshooting.
- The plain imperative-mood commit style used throughout this post is a default, not a rule. Prefixed conventions like Angular's are common on team projects, and consistency within a team matters more than which specific convention is chosen.
- Windows-specific configuration (line endings, path length limits, credential managers) is not addressed.

Opportunities for Improvement

1. Explore Git hooks (pre-commit, pre-push) to automate code linting, testing, and credential scanning before changes enter the repository.
2. Investigate Git LFS (Large File Storage) for managing datasets that exceed Git's practical size limits.
3. Adopt a branching strategy such as Git Flow or trunk-based development for projects with multiple active contributors.
4. Integrate Git with continuous integration services (GitHub Actions, GitLab CI) to automatically run tests and render reports on every push.
5. Learn interactive rebase (`git rebase -i`) for cleaning up commit history before merging feature branches.

6. Set up signed commits using GPG keys to verify authorship in security-sensitive research contexts.

Wrapping Up

Setting up Git for data science work requires a modest initial investment—perhaps an afternoon of configuration and practice—but the returns are substantial. Every analysis becomes recoverable, every experiment becomes safe to try, and every collaboration becomes structured.

What I learned most clearly from adopting Git is that version control changes the way one approaches work. Once every change is tracked, there is greater willingness to experiment, more disciplined documentation, and greater confidence that nothing important will be lost.

For readers getting started with Git, the advice is straightforward:

- Start with the core workflow: `init`, `add`, `commit`, `push`.
- Use branches for every experiment, no matter how small.
- Write commit messages in the imperative mood.
- Add a `.gitignore` before the first commit.
- Pull before pushing when collaborating.

The commit history built today is the documentation one will rely on later. Make it a habit to create messages that stand as informative, concise, and consistent narratives of a project's evolution.

See Also

Related blog posts:

- Multi-Laptop macOS Bootstrap: Managing configuration files with Git
- Unix Command-Line Workspace Setup for Data Science: Terminal setup that complements a Git workflow, and the place to look for git aliases and diff tooling (`git-delta`, `lazygit`) not covered in this post

Course materials:

- *Git and GitHub for Biostatistics* (the lab's five-day intro course) covers the same ground as this post in more depth, with exercises. The authentication and commit-message conventions in this post now follow that course.
- *Git Intermediate* (the three-day follow-on course) covers selective staging, rewriting history, and recovery, none of which is addressed here.

Key external resources:

- Pro Git Book: The definitive free reference by Scott Chacon and Ben Straub
- Git Official Documentation: Command reference and manual pages
- GitHub Skills: Interactive courses for learning GitHub workflows
- Atlassian Git Tutorials: Clear, well-structured guides for all skill levels
- Learn Git Branching: Interactive visualization tool for understanding branches

- Git Tower Learning Resources: Tutorials and cheat sheets
- GitKraken Git Cheat Sheet: Quick command reference
- Bryan, J. (2018). Excuse Me, Do You Have a Moment to Talk About Version Control? *The American Statistician*, 72(1), 20–27. DOI: 10.1080/00031305.2017.1399928
- Bonfim, G. (2023). Git Basics—All You Need to Know as a New Developer. Medium.

Reproducibility

This post does not involve computational analysis, so there is no analysis pipeline to reproduce. The Git commands shown throughout the post can be executed in any standard terminal environment with Git installed.

Environment used for this post:

```
git --version
```

Minimum requirements:

- Git 2.30 or later
- macOS 12+ or Ubuntu 20.04+
- A GitHub account authenticated with `gh auth login`

All commands were tested on macOS with Zsh and Git 2.43.0.

Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

I would enjoy hearing from you if:

- You spot an error or a better approach to any of the code in this post.
- You have suggestions for topics you would like to see covered.
- You want to discuss R programming, data science, or reproducible research.
- You have questions about anything in this tutorial.
- You just want to say hello and connect.

Related posts in this cluster

This post is part of the *Workflow Construct* series. Recommended reading order:

1. Post 15: A Workflow Construct for the Modern Data Scientist
2. Post 16: Unix Command-Line Workspace Setup for Data Science
3. Post 17: Multi-Laptop macOS Bootstrap
4. **Post 18: Setting Up Git for Data Science Workflows** (this post)
5. Post 19: Setting Up Neovim as a Data Science IDE
6. Post 21: Modern CLI Replacements for the Shell Layer
7. Post 25: Install Linux Mint on a MacBook Air