Setting Up Git for Data Science Workflows

A practical guide to version control for solo analysts and small research teams

Ronald 'Ryy' G. Thomas

2026-02-11

Table of contents

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

Version control is the foundation of every reproducible research workflow.

1 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.

This post documents my own path from "I'll just save copies" to a reliable Git-based workflow. It covers installation, basic commands, branching, collaboration, and the commit message practices that I found most valuable. I am writing as a learner, not as an expert, and I welcome corrections or suggestions from readers who have found better approaches.

More formally, this post documents the version-control concern that cuts across multiple layers of the Workflow Construct described in post 52. 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.

1.1 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).

1.2 Objectives

1. Install and configure Git on macOS or Linux, including identity setup and SSH authentication.
2. Learn the core Git workflow: staging, committing, pushing, and pulling changes.
3. Understand branching and merging for safe experimentation and collaboration.
4. Adopt commit message best practices drawn from the Angular project and the imperative mood 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.

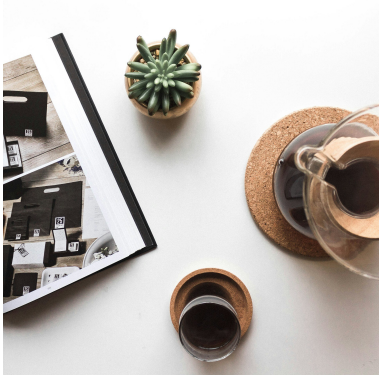

Figure 2: Settling in for version control setup.

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

2 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)
- 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
```

3 What is Git?

Git is a distributed version control system. In plain terms, it is a tool that records every change you make to a set of files, stores those changes in a structured history, and allows you to retrieve

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.

4 Getting Started: Initial Configuration

Before using Git for the first time, you need to set your identity. Git attaches your name and email to every commit you make.

```
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 your configuration:

```
git config --list
```

Setting up SSH authentication with GitHub:

SSH keys allow you to push and pull from GitHub without entering your password each time. Generate a key pair:

```
ssh-keygen -t ed25519 \
  -C "you@example.com"
```

Then add the public key to your GitHub account under Settings > SSH and GPG keys. Test the connection:

```
ssh -T git@github.com
```

A successful response will say "Hi username! You've successfully authenticated."

5 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:

```
git clone git@github.com:username/repo.git
```

This downloads the full project, including all history, and sets up the remote connection automatically.

I found that initializing 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.

6 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 your 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 you have the latest version.

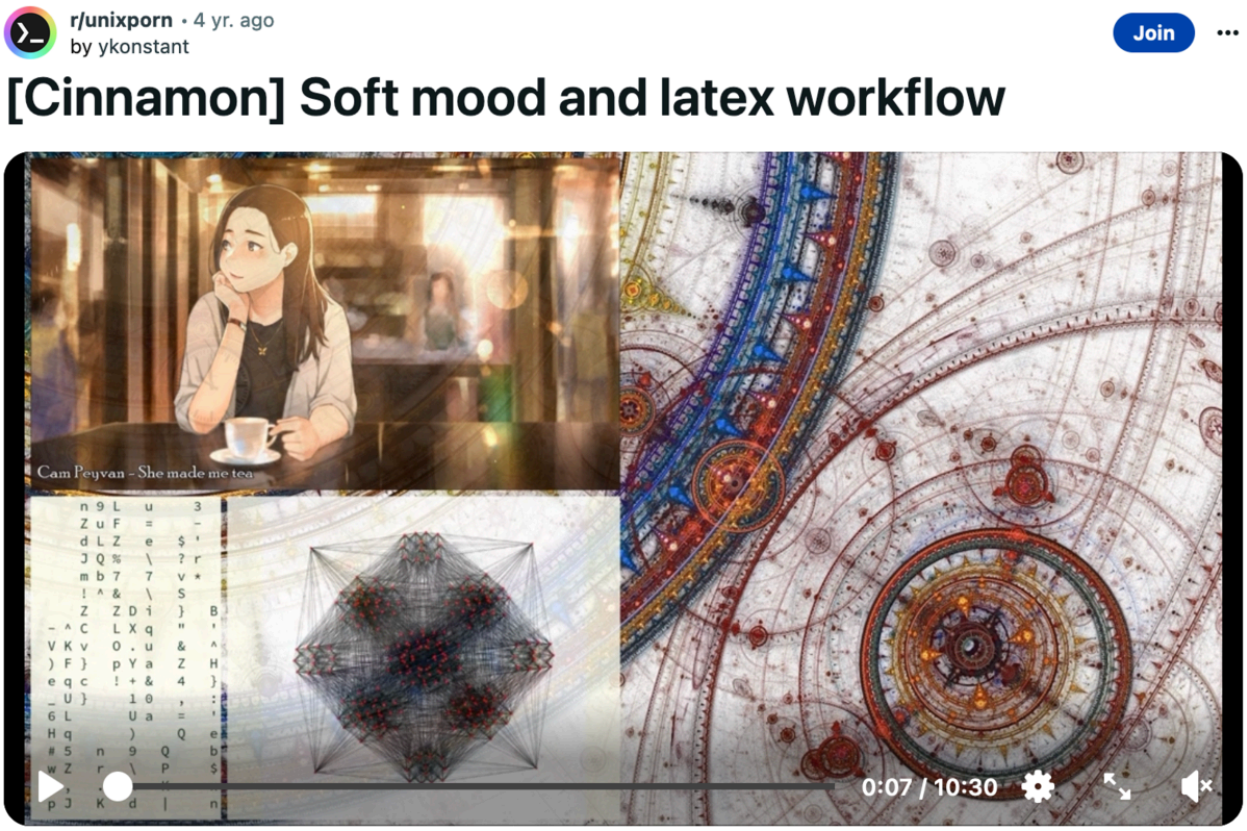

Figure 3: Soft atmospheric image suggesting focused work

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

7 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. You can experiment freely on a branch without affecting the stable version of your 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.

8 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):

1. Log into GitHub and navigate to the repository (for example, `rgt47/x24`).
2. Select **Settings** in the top row of tabs.
3. Select **Collaborators** in the left panel.
4. Click the green **Add people** button.
5. Search for the collaborator's GitHub username (for example, `rgt4748`).
6. Select **Add rgt4748** to send the invitation.

Accepting the invitation (collaborator):

1. Log into GitHub.
2. Click the notification icon in the upper right corner.
3. Select the invitation (for example, "Invitation to join rgt47/x24 from rgt47").
4. Click the green **Accept Invitation** button.

Cloning and contributing (collaborator):

The collaborator clones the repository to their workstation:

```
git clone \
  https://github.com/rgt47/x24.git
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:

On GitHub, the collaborator clicks **Contribute**, then **Open pull request**, enters a title and description, and submits the request.

Reviewing and merging (repository owner):

The repository owner reviews the pull request, checks the **Files changed** tab, and merges if the changes are acceptable. This workflow keeps the main branch clean and provides a record of every contribution.

9 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.

9.1 Use the Imperative Mood

A valuable practice involves writing commit messages as though the commit, when applied, will perform a specific action. Construct your 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.

9.2 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.

9.3 The Angular Convention

The Angular project provides an exemplary model for structured commit messages. Their convention uses specific prefixes that categorize each commit:

- `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

By incorporating these prefixes, the commit history becomes a structured resource for understanding the nature and purpose of each change. For example:

```
git commit -m "feat: Add cross-validation to model"
git commit -m "fix: Correct off-by-one in data merge"
git commit -m "docs: Update README with new workflow"
```

I adopted a simplified version of this convention for my data science projects and found that it made navigating project history substantially easier.

10 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 master:analysis.Rmd
```

This displays the file contents as they existed on the specified branch or commit, without modifying your 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 places you in a "detached HEAD" state where you can inspect the old version. To return to the current state:

```
git checkout main
```

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.

11 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 you commit a large data file 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, verify that your SSH key is added to the agent:

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

12 Verification

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

```
git --version
```

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

```
ssh -T git@github.com
```

```
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 your name and email, SSH authenticates successfully, and the test commit appears in the log, the setup is complete.

13 Daily Workflow

Task	Command
Check status	`git status`
Stage files	`git add analysis.R data/clean.csv`
Commit changes	`git commit -m "feat: Add model"`

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

13.1 Things to Watch Out For

1. **Never commit credentials or API keys.** Add `.env`, `.Renviron`, and credential files to `.gitignore` before your 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 you push.** 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.

14 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 remove SSH keys used for GitHub:

```
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.

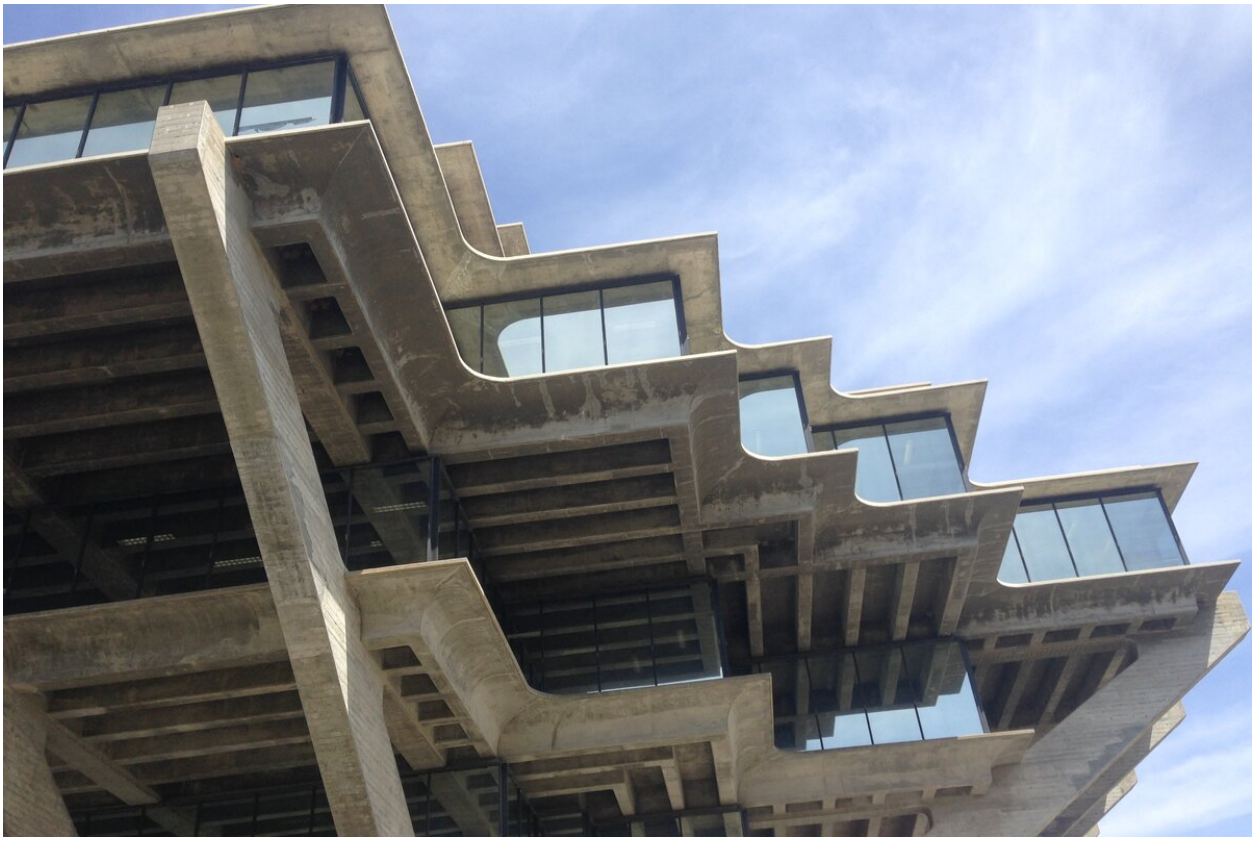

Figure 4: UCSD Geisel Library, a landmark of research and learning

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

15 What Did We Learn?

15.1 Lessons Learnt

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.

- SSH key authentication eliminates password friction and is more secure than HTTPS token-based access for regular use.
- 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.

15.2 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 commit message conventions described here reflect one widely adopted standard (Angular). Other projects use different conventions, 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.

15.3 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.

16 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 you think about your work. Once every change is tracked, you become more willing to experiment, more disciplined about documentation, and more confident that nothing important will be lost.

For readers getting started with Git, my 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 with a clear prefix.
- Add a `.gitignore` before your first commit.
- Pull before you push when collaborating.

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

17 See Also

Related blog posts:

- Setting Up Dotfiles on GitHub – Managing configuration files with Git
- Configuring the Terminal for Data Science – Terminal setup that complements a Git workflow

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.

18 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 with SSH key configured

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

19 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.