



Department of Econometrics and Business Statistics

Git and GitHub Practical Guide

Kunal Kapoor | 34144161

May 2025

Abstract

This guide provides a structured walkthrough of using Git and GitHub for efficient version control and collaboration. It covers essential concepts, practical commands, and workflows to help individuals track changes, manage branches, resolve conflicts, and maintain organized project histories.



AACSB
ACCREDITED



1 Git Guide: Version Control and Collaboration

1.1 What is Git?

Git is a **distributed version control system** that tracks changes in files and helps teams collaborate safely on projects.

With Git, you can:

- Save snapshots of your work (called **commits**)
- Restore previous versions when needed
- Work on multiple ideas at once (using **branches**)
- Merge changes from different contributors
- Experiment safely without disturbing the main project

Git is the most widely used version control system today, trusted by individual developers, companies, and open-source communities worldwide.

1.2 Why Use Git?

Using Git offers major benefits:

- **Version Control:** Track every change made to your project files.
- **Collaboration:** Multiple people can work on the same project without overwriting each other's changes.
- **Backup:** Push your project to online services like GitHub to keep it safe.
- **Experimentation:** Create branches to safely try new features without affecting the main project.
- **Mistake Recovery:** Easily undo mistakes by rolling back to earlier commits.
- **Transparency:** Clearly see who made changes, when, and why.

Whether working alone or with a team, Git makes your projects safer, more organized, and more professional.

1.3 Key Concepts in GIT

Table 1: *Key concepts and terms used in Git, explained in simple language.*

Term	Meaning
Repository	A folder that Git tracks; contains project files and the Git history.
Commit	A saved snapshot of your project at a specific point in time.
Branch	A separate line of development; allows safe parallel work.
Merge	Combining changes from different branches together.
Conflict	When two branches edit the same part of a file differently; must be resolved manually.
Remote	A version of your repository hosted online (e.g., GitHub).
Tag	A fixed label for a specific commit, often used for marking releases (e.g., v1.0).

1.4 Setting Up Git

Before starting with Git, make sure you have the following:

- **Git Installed:**

Install Git on your computer.

→ You can download it from git-scm.com or install it through your operating system's package manager (e.g., Homebrew on Mac, apt on Linux, etc.).

- **GitHub Account:**

Create a free account at github.com to store your repositories online and collaborate with others.

- **Code Editor:**

You can use any text editor or IDE (like Visual Studio Code, RStudio, or even plain Notepad++) to work with Git projects.

Git itself does not depend on any specific software.

1.5 Practical Walkthrough: Applying Git

In this section, we apply Git basics through a real mini-project example.

We will:

- Set up a project folder
- Track changes with Git
- Push to GitHub
- Create branches
- Handle conflicts
- Tag versions

And more!

These steps cover the full cycle of using Git for version control and collaboration.

Step 1: Initialize Git Repository and Create Project File

First, we set up Git tracking inside our project folder and create our first file.

Commands we ran in our command line interface

```
git init
```

The above command Initializes Git tracking in the folder.

Then we create a file `example.qmd` (can be done through your code editor) and knit it into an HTML output to confirm it works.

Step 1.2:

1. Open your code editor (e.g., RStudio).
2. Create a new Quarto document (`example.qmd`).
3. Add simple content like a title and a few lines of text.
4. Save the file inside your project folder.
5. Render (knit) the `example.qmd` to produce an HTML file (e.g., `example.html`).

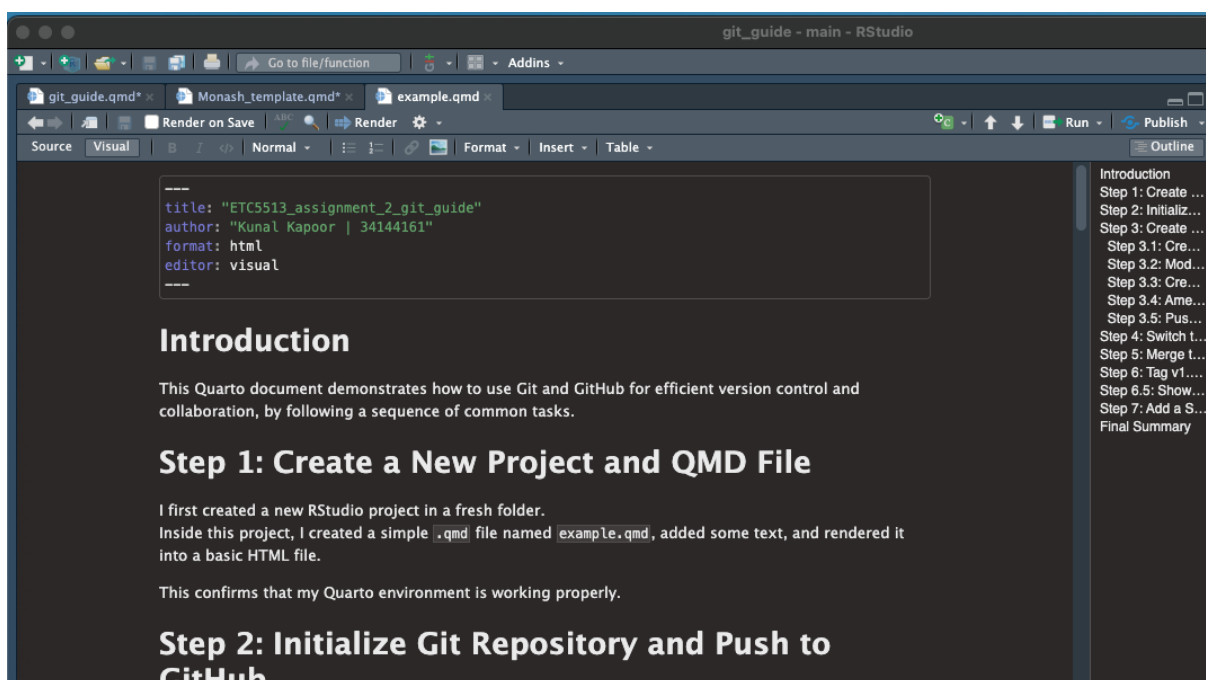


Figure 1: File before rendering

ETC5513_assignment_2_git_guide

AUTHOR

Kunal Kapoor | 34144161

Introduction

This Quarto document demonstrates how to use Git and GitHub for efficient version control and collaboration, by following a sequence of common tasks.

Step 1: Create a New Project and QMD File

I first created a new RStudio project in a fresh folder.

Inside this project, I created a simple `.qmd` file named `example.qmd`, added some text, and rendered it into a basic HTML file.

This confirms that my Quarto environment is working properly.

Step 2: Initialize Git Repository and Push to GitHub

Figure 2: Rendered HTML

Step 2: First Commit

We save our project's first version into Git history.

Commands:

```
git add example.qmd
```

```
git commit -m "First Commit - Added example.qmd and rendered HTML files"
```

This creates the first checkpoint of our project.

Note: Please run these above commands one by one

Step 3: Link to GitHub and Push

Now we connect our local project to GitHub and upload it.

Commands:

```
git remote add origin <your-repository-URL>  
git push -u origin main
```

This makes sure our work is backed up online.

Step 4: Create a New Branch and Make Changes

We create a separate branch to work safely without disturbing the main branch.

Commands:

```
git checkout -b testbranch
# Create and switch to a new branch called 'testbranch'
# Edit example.qmd (e.g., add a new line)
git add example.qmd
git commit -m "Step 3.2: Added a line to example.qmd in testbranch"
git push -u origin testbranch
```

Changes are safely made inside testbranch.

Note: Whenever you see # followed by text in the command examples, it is a comment. Comments are for humans to read — the computer ignores them when running commands.

Step 5: Add Data Folder and Amend Previous Commit

We add a new folder data/ (e.g., containing Assignment 1 files) and update our last commit to include it.

Commands:

```
mkdir data
```

```
# Create a new folder named 'data'
```

```
# Move your Assignment 1 data files into the 'data' folder manually
```

```
git add data/
```

```
# Stage the entire 'data' folder for the next commit
```

```
git commit --amend --no-edit
```

```
# Update the last commit to include the staged data folder without changing the commit message
```

```
git push --force
```

```
# Force push the amended commit to GitHub
```

The previous commit now includes the new folder without making an extra commit.

Step 6: Modify Main Branch to Create Conflict

We make different changes directly on main branch to simulate a merge conflict later.

Commands:

```
git checkout main
# Switch back to the main branch
# Edit example.qmd (modify the same line differently)
git add example.qmd
# Stage the changes
git commit -m "creating a conflicting change in main branch"
# Save the conflicting changes
git push
# Push the conflicting changes to GitHub
```

Now main and testbranch both have different edits to the same file.

Step 7: Merge Branches and Resolve Conflict

We attempt to merge testbranch into main. Git detects the conflict, and we fix it manually.

Commands:

```
git merge testbranch

# Attempt to merge 'testbranch' into 'main'
# Git will detect a conflict and stop the merge
# Open example.qmd manually
# Look for conflict markers like:
# <<<<<<< HEAD
# (your main branch content)
# =====
# (your testbranch content)
# >>>>>> testbranch
# Decide which version to keep or combine them, then save the file

git add example.qmd
# Stage the resolved file

git commit -m "Fixed merge conflict between main and testbranch"
# Save the merge result

git push
# Push the resolved version to GitHub
```

Changes from both branches are combined after resolving the conflict.

Step 8: Tagging a Version

We mark the current state of the project as version 1.0.

Commands:

```
git tag -a v1.0 -m "Version 1.0 release"
git push origin v1.0
```

Versioning helps you identify stable points in your project.

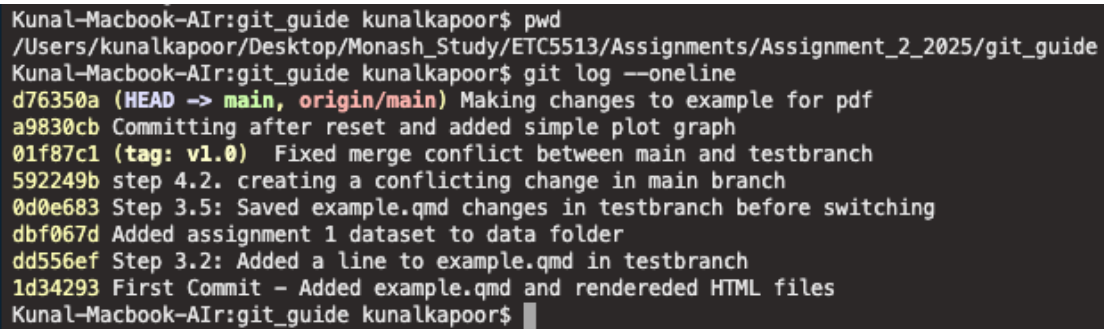
Step 9: Show Commit Log in Condensed Form

We display a simple, one-line summary of the project's commit history.

Commands:

```
git log --oneline
```

Shows a compact list of all commits



```
Kunal-Macbook-Air:git_guide kunalkapoor$ pwd
/Users/kunalkapoor/Desktop/Monash_Study/ETC5513/Assignments/Assignment_2_2025/git_guide
Kunal-Macbook-Air:git_guide kunalkapoor$ git log --oneline
d76350a (HEAD -> main, origin/main) Making changes to example for pdf
a9830cb Committing after reset and added simple plot graph
01f87c1 (tag: v1.0) Fixed merge conflict between main and testbranch
592249b step 4.2. creating a conflicting change in main branch
0d0e683 Step 3.5: Saved example.qmd changes in testbranch before switching
dbf067d Added assignment 1 dataset to data folder
dd556ef Step 3.2: Added a line to example.qmd in testbranch
1d34293 First Commit - Added example.qmd and rendered HTML files
Kunal-Macbook-Air:git_guide kunalkapoor$
```

Figure 3: *Git log --oneline condensed form*

Step 10: Clean Up Old Branch

We delete the testbranch locally and remotely since it's no longer needed.

Commands:

```
git branch -d testbranch
```

```
git push origin --delete testbranch
```

Keeping only necessary branches keeps the project tidy.

Step 11: Undo a Commit (Without Losing Changes)

Suppose we add a new section (like a plot) to `example.qmd`, but realize we committed too early. We can undo the last commit while keeping changes intact.

Commands:

```
# After editing and committing  
git reset --soft HEAD~1
```

The last commit is undone, but your edits remain staged, ready for a better commit.

2 Summary of Actions

By following these steps, you learned how to:

- Set up a Git repository
- Track and save changes
- Work safely using branches
- Connect your project to a remote GitHub repository
- Push and pull changes to/from GitHub
- Handle merge conflicts
- Tag versions
- Manage commits and undo mistakes

3 Git Commands Quick Reference

Table 2: Quick reference table summarizing important Git commands and their purposes.

<u>Git Command</u>	<u>Meaning</u>	<u>Why It's Useful</u>
<code>git init</code>	Start tracking the project with Git	Begin version control
<code>git status</code>	Check the status of changes	See staged, unstaged, or untracked files
<code>git add filename</code>	Stage a specific file	Prepare file for committing
<code>git add .</code>	Stage all changes in the working directory	Quickly add everything for commit
<code>git commit -m "message"</code>	Save a snapshot of changes	Record work into Git history
<code>git log</code>	Show commit history	View detailed list of commits
<code>git log --oneline</code>	Condensed commit history	View a brief summary of commits
<code>git branch</code>	List all branches	Manage and view project branches
<code>git branch branch_name</code>	Create a new branch	Work separately without affecting the main
<code>git switch branch_name</code>	Switch to another branch	Move between versions
<code>git switch -c branch_name</code>	Create and switch to a new branch	Shortcut to save time
<code>git merge branch_name</code>	Merge another branch into current	Combine features safely
<code>git push</code>	Upload commits to GitHub	Share work online
<code>git push -u origin main</code>	Push and track a new branch	Set up branch tracking
<code>git pull</code>	Download and merge remote changes	Stay updated with remote
<code>git tag -a v1.0 -m "message"</code>	Create an annotated tag	Mark important project points

<u>Git Command</u>	<u>Meaning</u>	<u>Why It's Useful</u>
<code>git reset --soft HEAD~1</code>	Undo last commit but keep changes staged	Correct mistakes without losing work
<code>git remote add origin url</code>	Connect local repo to GitHub	Set up a remote repository
<code>git remote -v</code>	View remote connections	Confirm remote links
<code>git remote remove origin</code>	Remove a GitHub link	Disconnect remote repository
<code>git branch -d branch_name</code>	Delete a local branch	Clean up after merging
<code>git stash</code>	Temporarily save uncommitted work	Save work without committing
<code>git stash pop</code>	Reapply stashed work	Restore work and continue
<code>git revert commit_id</code>	Undo a specific commit safely	Safe undo for public history
<code>git rebase branch_name</code>	Move branch commits onto another branch	Simplify commit history
<code>git rebase -i HEAD~n(squash inside rebase)</code>	Interactive rebase to squash commits Combine multiple commits into one	Clean multiple commits Tidy commit history

4 Conclusion

Mastering Git provides a strong foundation for any collaborative or individual project work. Whether you are coding, writing, or analyzing data, Git ensures that your progress is organized, secure, and easy to manage.