MONASH University

MONASH BUSINESS SCHOOL

Department of Econometrics

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.

Table of contents

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 1.1: Initialize Git Repository and Create Project File

Before we can track our work with Git, we need to tell Git :
"Hey Git can you start watching this folder"

This is what is called **initializing a Git repository.**

Commands we ran in our command line interface

```
git init
```

In your command line (Terminal), move into your project folder and run:

- **What This Does:**

This command sets up a hidden .git folder inside your project.

That folder will store all the history of your changes, forever. You won't see it normally, but Git will.

Tip:

If you ever want to double-check whether Git is watching your folder, you can run:

```
git status
```

It will tell you if you're inside a Git repo or not.

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:

Now that Git is ready to track our project, let's create the very first file inside our folder.

- Open your code editor — for example, RStudio, Visual Studio Code, or even Notepad++.
- Create a new Quarto document and name it example.qmd.

(A .qmd file is just a special kind of markdown file that can easily be turned into different formats like HTML or PDF.)

```
---
title: "Example QMD"
author: "Kunal Kapoor"
format: pdf
---

# Hello Git

This is my first Git-tracked file!
```

Save the file inside your project folder (the same folder where you ran git init).

Now let's render (or "knit") this .qmd file to produce a real output file we can view, like a webpage.

- In RStudio, just click the Render button.

- If you're using the Quarto command line, you can run:

```
quarto render example.qmd
```

This will create a new file called example.html in your project folder.

- Why This Step Matters:

We now have an actual file that Git can track — and we've confirmed that our project setup works properly!

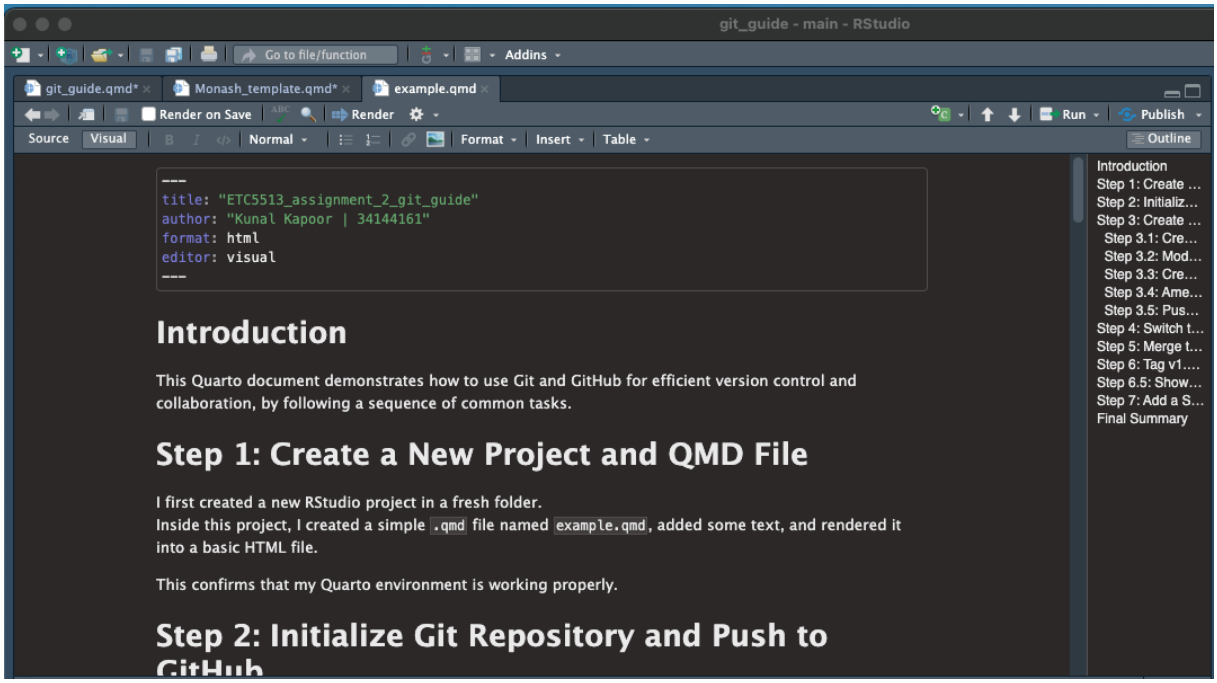

Figure 1: *File in Rstudio environment*

Figure 2: *HTML Render in Web Browser*

Step 2: Let's make our first commit

Now that we have our example.qmd and the rendered HTML file, it's time to save a snapshot of our project using Git.

This snapshot is called a commit.

Think of a commit like saving a version of your project at a particular moment —so if something goes wrong later, you can always rewind back to this point!

Step 2.1: Stage your work

First we check which files Git is already tracking by running a simple command in our command line interface [Guess what can tell us the status of GIT?]

```
git status
```

This shows you which files Git is tracking and which ones are new.

You should see something like:

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    example.qmd
    example.html
```

These are files Git knows exist but hasn't saved yet.

- To stage them (prepare them for committing), run:

```
git add example.qmd
```

This creates the first checkpoint of our project and also tell Git:

"Hey, I want to include these files in the next snapshot."

Step 2.2: Save (Commit) your Snapshot

Now that the files are staged, let's commit them:

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

The **-m** lets you attach a short message describing what this commit is about.

- Why This Step Matters:
  - Staging ensures you're only saving what you really intend to.

– Commit messages are like breadcrumbs — they help you (and your future teammates) understand what changes were made, and why.

Table 2: *Recap of step 2*

STEP	PURPOSE
`git status`	Check what git is tracking
`git add`	Prepare the files for commit
`git commit`	Save a snapshot with a message

Step 3: Link to GitHub and Push it Oneline

Saving work on your computer is good.

But what if your laptop crashes? Or you want to show your work to others?

That's why we push our Git repository to GitHub a safe place on the internet to store your project and collaborate with others.

Step 3.1: Create a new repository on GitHub

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

First, log in to GitHub and create a new empty repository.

- `Don't initialize it with a README, .gitignore, or license.`

(Otherwise Git will get confused when pushing.)

Once created, GitHub will show you the repository URL — something like:

```
https://github.com/your-username/your-repository.git
```

Copy this link — we'll need it next!

Step 3.2: Connect Local project to GitHub (Remote Link)

In your terminal (inside your project folder), run:

```
git remote add origin <your-repository-URL>
```

Replace <your-repository-URL> with the link you copied from GitHub.

- **What this does:**
  - It tells Git:
  - *"Hey, this is the address where you will send (push) all my project snapshots."*

Bonus tip:
To double-check that the connection was made:

```
git remote -v
```

We should see an output like this:

```
origin  https://github.com/your-username/your-repository.git (fetch)
origin  https://github.com/your-username/your-repository.git (push)
```

This shows that Git knows exactly where our project will live online

Step 3.3: Push our work to GitHub

Now's let's send (push) our local commits to GitHub:

```
git push -u origin main
```

The -u flag tells Git to remember that "origin/main" is the default place to push changes next time.

(So from now on, you can just type git push.)

- **Why This Step Matters:**
  - Your work is now backed up safely online.
  - You can access it from anywhere — just log into GitHub.
  - You're ready to collaborate with others if needed!

QUICK TIPS:

1. **Push after every major work session** to avoid losing progress.
2. **Don't push unfinished or broken work** unless you're using a branch meant for testing.

Table 3: *Step 3 | Summary Table*

STEP	PURPOSE
Create Repository on GitHub	Set up an online space for your project
`git remote add origin <your-repository-URL>`	Connect your local Git project to the GitHub repository
`git remote -v`	Confirm that Git knows where to push and fetch from
`git push -u origin main`	Upload your local project history to GitHub and remember the link

Step 4: Create a New Branch and Make Changes

When working on a project, it is a good idea to experiment or make changes without touching your main working version.

In Git, we do this by creating a branch.

A branch is like a safe testing ground where you can make updates, try new features, or fix mistakes without affecting the original project.

Step 4.1: Create and a New Branch

In our command line interface we will run

```
git branch testbranch
```

This only creates the branch. It does not move you into it yet.NOTE:

Step 4.2: Switch to a New Branch

Now, we switch into the new branch we just created by using

```
git checkout testbranch
```

After switching, you are now working inside testbranch.
Any changes you make will stay isolated from your main branch.

SHORTCUT

```
git checkout -b testbrach
```

If we run the above command then both creation of new branch and switching to the new branch happens at the same time.

Step 4.3: Make A change in the Branch

Now that we are inside the testbranch, let's make a small change to practice working safely inside a branch.

To do this:

1. In RStudio, go to the top menu and click
    - File → New File → Quarto Document.
2. Name the new file example.qmd if it is not already.
3. Inside example.qmd, make a small change — for example, add a new heading:
4. Save the file after making your changes.

Step 4.4: Stage and Commit the changes in the Branch

Now that we have made a change inside example.qmd and saved it, we need to tell Git that we are ready to save this new version into the project history.

In Git, this happens in two small actions:

- Staging (preparing the change)
- Committing (saving the change)

Step 4.4.1: Stage the file

First, we need to stage the changed file. In your terminal, run:

```
git add example.qmd
```

This command tells git: *'Please include this updated file in the next snapshot'*

Step 4.4.2: Commit the Change

After staging, we can now commit the change

```
 git commit -m 'Updated the example.qmd file with changes made in testbranch'
```

The -m option allows you to add a **commit message** describing what this change is about.

Step 4.5: Push the Branch to GitHub

Now that we have committed our changes locally inside the testbranch, the next step is to upload this new branch to GitHub.

This way, the changes are safely backed up online, and anyone you collaborate with can also see your work.

Step 4.5.1: Push the Branch

In our command line interface we will run the follwing command:

```
git push origin main
```

What this command does

- git push send the local changes to GitHub
- -u (or — set upstream) tells GitHub to remember that this branch should push and pull from origin/testbranch automatically in the future.

Table 4: *Summary of Step 4*

STEP	PURPOSE
`git branch testbranch`	Create a new branch called *testbranch*
`git checkout testbranch`	Switch to working inside the new branch
Create or edit example.qmd	Make a safe change inside the testbranch
`git add example.qmd`	Stage the modified file for saving
`git commit =-m'YOUR MESSAGE'`	Save the staged changes into Git history
`git push -u origin testbranch`	Upload the testbranch to GitHub and set up tracking

Step 5: Add Data Folder and Amend Previous Commit

Sometimes after making a commit, you realize you forgot to include an important file or folder.

Instead of making a new commit for small things, Git allows you to amend the last commit and update it.

In this step, we will:

1. Create a new folder
2. Add files into it
3. Update (amend) our previous commit to include the new data
4. Push the corrected commit to GitHub

Step 5.1: Create a Data Folder

First, inside your project directory, create a new folder called data.

If you are using your computer's file explorer, simply create a new folder named data.

Or, from the command line, you can run:

```
mkdir data
```

Now, move your Assignment 1 files (or any sample files) into this data folder manually.

Step 5.2: Stage the New Data Folder

After adding our files into the data folder, we will tell git about the new content by staging it

```
git add data/
```

This command stages the entire folder and all the files inside it for the next commit.

data/ - Tells the git to track and stage the entire contents of the data folder.

Step 5.3: Amend the Last Commit

Now, instead of creating a new commit just for the data folder, we will update our previous commit to include it.

Run:

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

The –no-edit flag means you will keep the original commit message, but update the files included.

Your last commit now contains:

- The original changes (e.g., your example.qmd edits)
- Plus the newly added data folder.

Step 5.4: Push the Amended Commit to GitHub

Since we changed the commit history, we need to force push the update to GitHub.

```
git push --force
```

This command tells GitHub to replace the old commit with the new, updated one.

Table 5: *Summary Step 5*

STEP	PURPOSE
`mkdir data`	Create a new folder to strore project data
Moves files into data/	Add Assignment 1 or example files
`git add data/`	Stage the new folder and it's contents
`git commit --amend --no-edit`	Update the last commit to include the new folder
`git push --force`	Push the amended commit to GitHub

Step 6: Modify Main Branch to Create Conflict

In real projects, when multiple people work on the same files, sometimes conflicts happen.

A conflict occurs when Git cannot automatically decide which version of a file is correct because two people (or two branches) have changed the same part of the file differently.

In this step, we will deliberately create a conflict between main and testbranch, so we can practice resolving it.

Step 6.1: Switch back to the Main Branch

First, we need to move back to the main branch to make a separate change there, let's run

```
git checkout main
```

This command switched our working environment from *testbranch* back to *main*.

Step 6.2: Edit the Same File Differently

Now open the example.qmd file again. Change the same part of the file that you edited in testbranch, but make the content different.

For example, you could replace the previous heading with a new line like:

```
## Different change made in main branch
```

Save the file after making the changes.

Step 6.3: Stage and Commit the Change

Now that we have made a conflcting change in the main branch, let's stage and commit the change:

First stage the updated file:

```
git add example.qmd
```

Then let's commit it:

```
git commit -m 'Create conflicting change in the main branch'
```

This saves the new version inside the main branch's history.

Step 6.4: Push the changes to GitHub

Finally, upload the new commit to GitHub:

```
git push
```

Now both your main branch and your testbranch have different changes to the same file — this sets us up perfectly to experience and learn how to resolve a conflict when merging.

Table 6: *Summary step 6*

STEP	PURPOSE
`git checkout main`	Switch back to the main branch
Edit example.qmd	Make a different change in the same part of the file
`git add example.qmd`	Stage the new change
`git commit -m 'YOUR MESSAGE'`	Save the change into Git history
`git push`	Upload the updated main branch to GitHub

Step 7: Merge Branches and Resolve Conflict

Now that both main and testbranch have different changes to the same file, it is time to merge the branches.

Since the changes overlap, Git will not be able to merge them automatically — it will ask you to manually resolve the conflict.

This is an important skill for working in real projects where multiple people edit the same files.

Step 7.1: Merge testbranch into main

First, make sure you are still in the main branch.

Then, try to merge the testbranch into main:

```
git merge testbranch
```

Git will attempt to combine the changes but will detect a conflict because git does not know which change to keep and which change to discard, so ti will show a conflict message.

```
Auto-merging example.qmd
CONFLICT (content): Merge conflict in example.qmd
Automatic merge failed; fix conflicts and then commit the result.
```

Step 7.2: Understand the Conflict

Open example.qmd in your editor.

Git will have inserted conflict markers inside the file, showing you the two versions:

```
<<<<<<< HEAD
This is the version from the main branch.
=======
This is the version from the testbranch.
>>>>>>> testbranc
```

- The section between «««< HEAD and ======= shows what is currently in main.
- The section between ======= and »»»> testbranch shows what is coming from testbranch.
- Our job is to choose which changes to keep, or combine them, and then remove all the conflict markers.

Step 7.3: Fix the Conflict

Decide what the final content should be.

We can either:

1. Keep only the main branch version,
2. Keep only the testbranch version,
3. Combine both versions in a clean way.

Example (after resolving):

```
## Combined final version after conflict resolution


This line includes updates from both branches.
```

This is after fixing the conflict manually, and saving the file.

Step 7.4: Stage and Commit the Resolved file

Now, stage the resolved file:

```
git add example.qmd
```

Add the commit message

```
git commit -m 'Resolved the merge conflict manually between main and testbranch'
```

This officially completes the merge.

Step 7.5: Push the merged changes to GitHub

Finally let's push all our changes to GitHub

```
git push
```

Now our main branch on GitHub includes the merged content from both branches.

Table 7: *Summary Step 7*

STEP	PURPOSE
`git merge testbranch`	Merge changes from testbranch into main
Resolve the conflict manually	Edit the file to remove the conflict markers
`git add example.qmd`	Stage the resolved file
`git commit -m 'YOUR MESSAGE'`	Save the merged result
`git push`	Upload the merged branch to GitHub

Step 8: Tagging a Version

When working on a project, it is useful to mark important milestones for example, when you complete a feature, fix a major bug, or reach a stable release.

In Git, you can tag a specific commit with a label to make it easy to find later. Tags are often used to mark official version releases like v1.0, v2.0, and so on.

In this step, we will tag our project as version v1.0 after successfully merging the branches.

Step 8.1: Create an Annotated Tag

In your terminal, run the following command:

```
git tag – a v1.0 –m 'Version 1.0 release'
```

Explanation:

- -a v1.0 creates an annotated tag named v1.0.
- -m "…" provides a short message describing the tag.

Annotated tags store important information like:

1. Tag name
2. Tag message
3. Tagger's name
4. Date and time

This makes it more useful than a simple lightweight tag.

Step 8.2: Push the Tag to GitHub

By default, tags are not automatically pushed when you run git push and we must push them separately.

Push the tag using:

```
git push origin v1.0
```

This command sends your tag to GitHub, so it is stored along with your project. Now anyone who views your repository can see that version v1.0 has been marked

Table 8: *Summary step 8*

STEP	PURPOSE
`git tag -a v1.0 -m "Version 1.0 release"`	Create an annotated tag for the current project state
`git push origin v1.0`	Upload the tag to GitHub

Step 9: Show Commit Log in Condensed Form

As you continue working with Git, your project history will grow with many commits over time.

It can sometimes become difficult to quickly understand what has been done without scrolling through long commit messages.

Git allows you to view the commit history in a condensed and easy-to-read format, listing just short summaries.

In this step, we will view the condensed commit log.

Step 9.1: View a Condensed Commit Log

In your terminal, run:

```
git log --oneline
# Shows a compact list of all commits
```

What this does:

- Shows each commit on a single line.
- Displays the commit ID (shortened) and the commit message.

Example output:

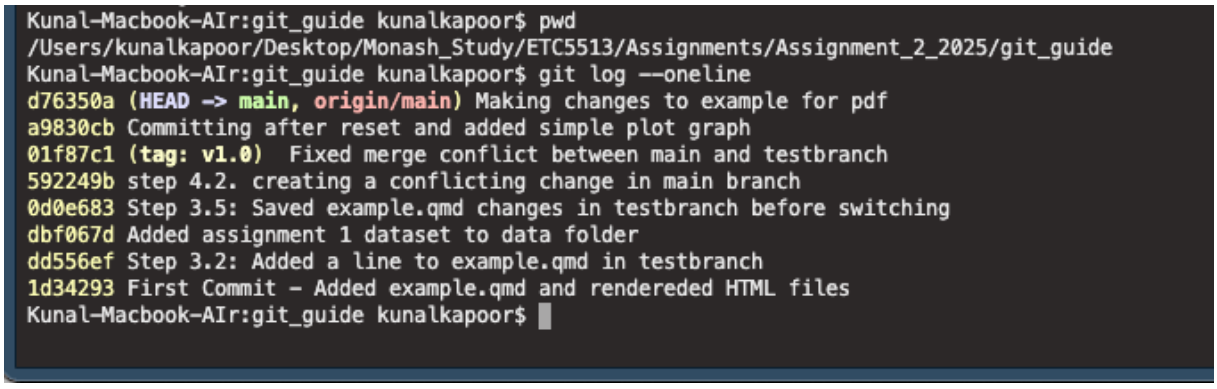

Figure 3: *Git log –oneline condensed form*

STEP	PURPOSE
`git log --oneline`	View a simplified one line summary of all commits

Step 10: Clean Up Old Branch

Once you have successfully merged your changes from a branch into the main project,

it is good practice to delete the branch if it is no longer needed.

This helps keep your repository organized and avoids confusion later.

In this step, we will delete the testbranch both locally (on your computer) and remotely (on GitHub).

Step 10.1: Delete the Branch Locally

First, delete the branch from your local machine.

In your terminal, run:

```
git branch -d testbranch
```

What this does:

- -d stands for “delete”.
- It safely deletes the branch, but only if it has already been merged.
- Git will prevent you from deleting unmerged branches unless you force it.

If you see an error that says the branch is not fully merged, and you still want to delete it (not recommended unless you are sure), you could use:

```
git branch -D testbranch
```

(But for our case, we use -d because we merged it correctly.)

Step 10.2: Delete the Branch Remotely

Now, delete the branch from GitHub as well.

Run:

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

What this does:

- origin is the name of your remote repository on GitHub.
- –delete testbranch tells Git to remove the testbranch from GitHub.

After this, testbranch will no longer appear in the list of branches on GitHub.

Table 10: *Summary Step 10*

STEP	PURPOSE
`git branch -d testbranch`	Delete the local testbranch after merge
`git push origin --delete testbranch`	Delete the remote testbranch from GitHub

Step 11: Undo a Commit (Without Losing Changes)

Sometimes, you might commit your changes too early —

maybe you forgot to edit something, or you want to improve your commit message.

In Git, you can undo the last commit while keeping all your file changes safe.

This is very useful when you want to fix your work without losing anything.

Step 11.1: Reset the last Commit Softly:

To undo the last commit but keep all your changes exactly as they were, use the following command:

```
git reset --soft HEAD~1
```

What this command does:

- reset moves the Git history back one step.
- –soft keeps your files and staged changes exactly as they were.
- HEAD~1 means “one commit before the current one.”

After running this, your changes are still there — they are just uncommitted, waiting to be committed again (correctly this time).

Step 11.2: Make any edits if needed

At this point, you can:

- Edit your files if you want to make more changes.
- Or, if everything is fine, just recommit with a better message.

Step 11.3: Recommit Properly

When you are ready, you can stage (if needed) and commit again with a better or corrected message:

```
git commit -m 'Updated section to include correct plot example'
```

Now you have replaced the rushed commit with a better one — and your work is clean.

Table 11: *Summary step 11*

STEP	PURPOSE
`git reset --soft HEAD~1`	Undo the last commit but keep all local changes
Edit and Recommit	Fix mistakes and save a cleaner commit

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 12: *Quick reference table summarizing important Git commands and their purposes.*

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

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