FWE 458 Environmental Data Science

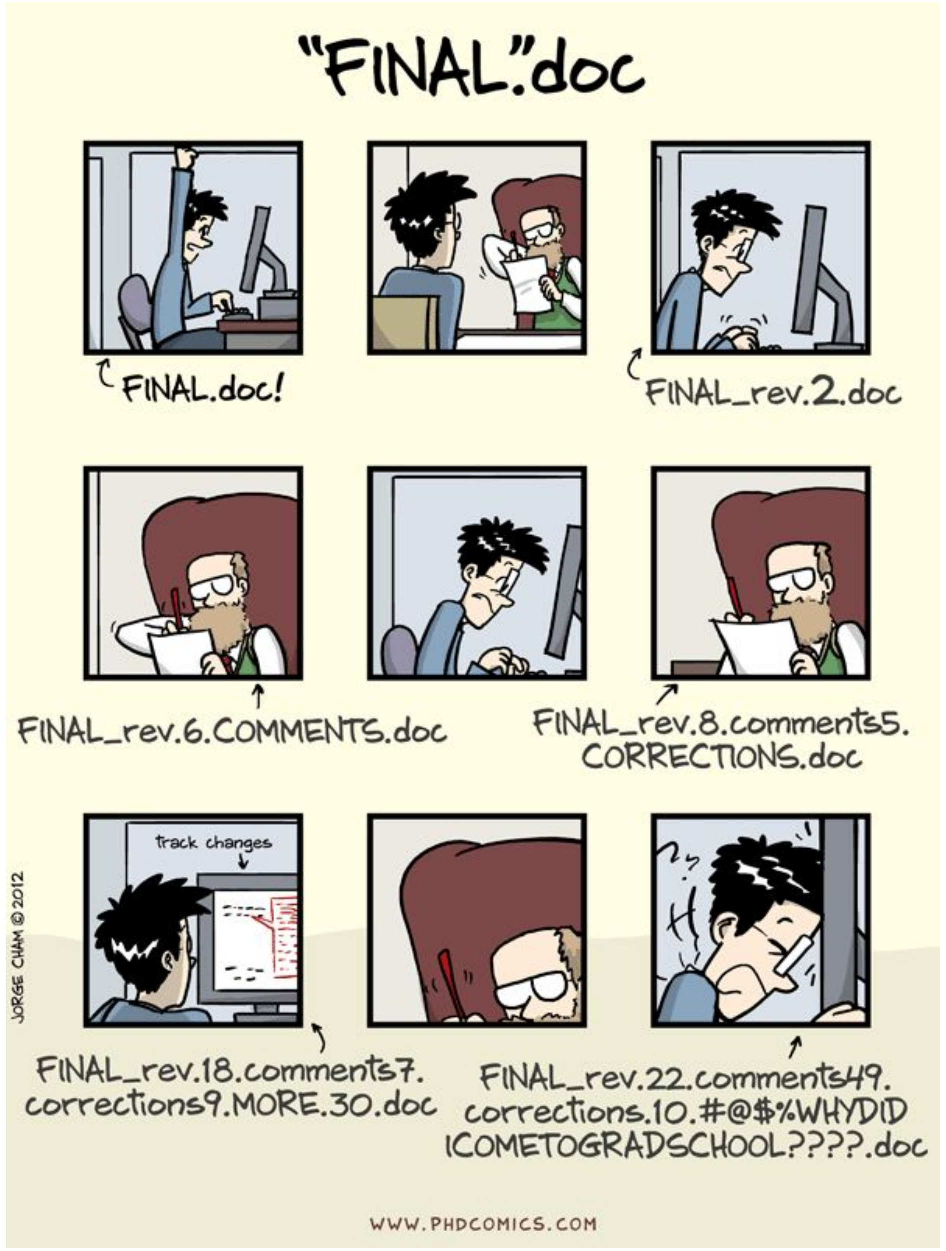
git and github

Spring 2026
Instructor: Min Chen

Version control

Have you ever met these issues:

- overwriting each other's work
- losing track of the latest version
- not being able to revert to a previous state if something goes wrong


"FINAL".doc
FINAL.doc!
FINAL_rev.2.doc
FINAL_rev.6.COMMENTS.doc
FINAL_rev.8.comments5.CORRECTIONS.doc
track changes
FINAL_rev.18.comments7.corrections9.MORE.30.doc
FINAL_rev.22.comments49.corrections.10.#@$%WHYDID ICOMETOGRADSCHOOL????.doc
JORGE CHAM © 2012
WWW.PHDCOMICS.COM

Version Control

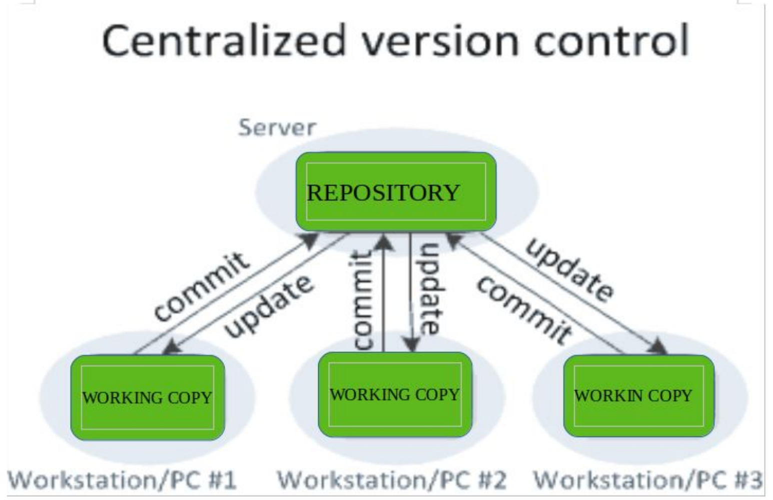
- a system that records changes to a file or set of files over time so that you can recall specific versions later
- "repository" (a storage location for your project), "commit" (saving changes), and "revision" (each change or set of changes).
- think about ‘Google Doc’

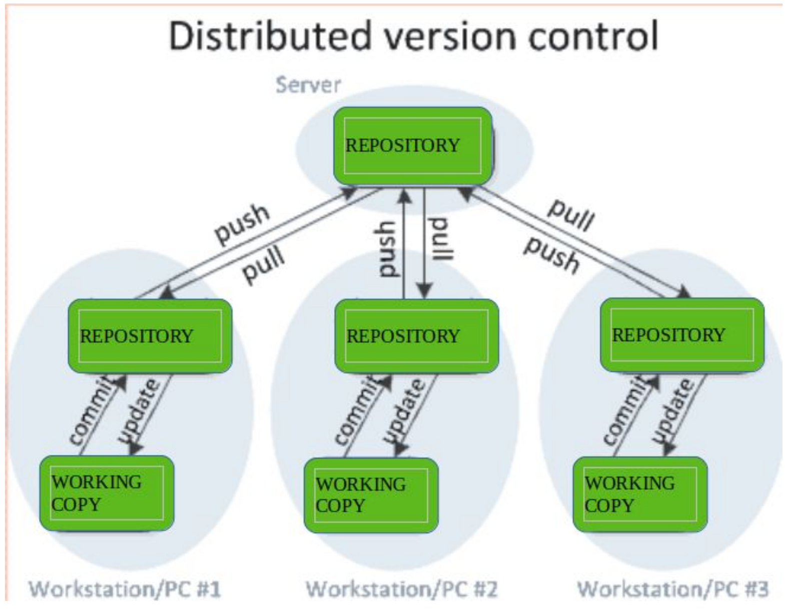
Benefit of Version Control

- **Collaboration**: Allows multiple people to work on the same project without interfering with each other's changes.
- **Tracking Changes:** Keeps a history of who changed what and when, which is crucial for understanding the evolution of a project and for debugging issues.
- **Reverting**: The ability to revert to a previous state if a mistake is made or if a project direction changes.
- **Experimentation**: Facilitates trying new ideas without the fear of losing the original work, as changes can be made in separate branches.

Types of Version Control Systems

- Local Version Control Systems: Simple database that keeps all the changes to files under revision control.
- Centralized Version Control Systems (CVCS): Systems like SVN, where a single server contains all the versioned files, and various clients check out files from this central place.
- Distributed Version Control Systems (DVCS): clients don't just check out the latest snapshot of the files; they fully mirror the repository including its full history.





https://www.geeksforgeeks.org/version-control-systems/

DVCS vs CVCS

Feature	DVCS Advantage	CVCS Limitation
Resilience	No single point of failure	Server dependency creates risks.
Offline Work	Full functionality offline	Requires server for most operations.
Performance	Local operations are fast	Server-reliant operations are slower.
Collaboration	Peer-to-peer sharing is possible	Requires central repository for collaboration.
Backup	Every clone is a backup	Central server is a single backup source.
Branching/Merging	Easy, fast, and efficient	Slower and more error-prone.
Flexibility	Supports diverse workflows	Enforces centralized workflow.

Git

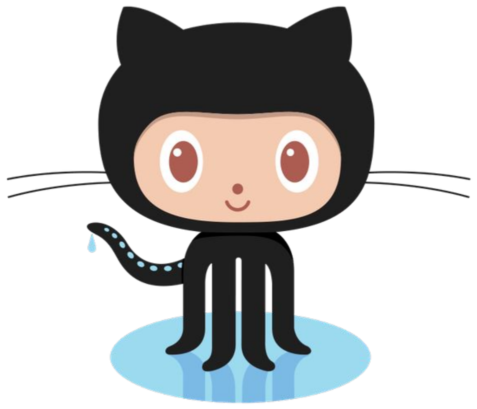
A popular distributed version control system

Git was originally authored by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development.

https://git-scm.com/

GitHub

- A web-based platform used for version control and collaboration
- The GitHub service started in February 2008. The company, GitHub, Inc., has existed as of 2007 and is located in San Francisco. On February 24, 2009, GitHub announced that within the first year of being online, GitHub had accumulated over 46,000 public repositories. The number became 372 million in 2023.

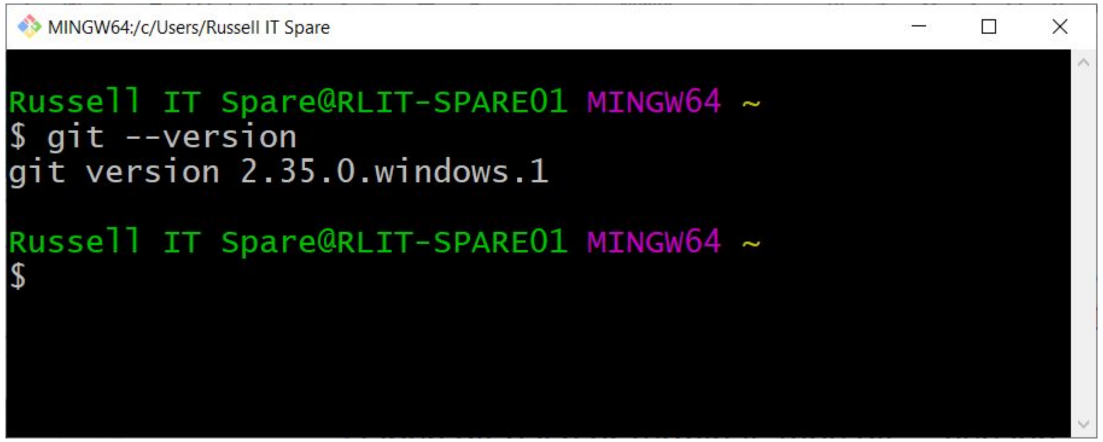


Git vs GitHub

- **Git is the tool, GitHub is the service:**
  - Git: Git is a distributed version control system, a tool that tracks changes in source code during software development. It’s designed for coordinating work among programmers, but it can be used to track changes in any set of files.
  - GitHub: a hosting service for Git repositories. It provides a web-based graphical interface. It also provides access control, collaboration features, task management, and more.
- **Local vs. Remote:**
  - Git: Operates on your local computer. You can use Git without GitHub (or any other central repository) to manage your version control.
  - GitHub: Acts as a remote repository. It hosts your code and offers a place for collaboration on that code. It's accessible from anywhere via the internet.
- **Collaboration:**
  - Git: Handles the version control part but doesn’t offer built-in mechanisms for team collaboration like commenting on changes or approving them.
  - GitHub: Provides a platform for collaborative features like pull requests (a way to merge changes from one branch to another after peer review), issues, code review, and team management.

Download and install git

- Windows users: https://git-scm.com/download/win
- Mac users: typically git is already installed
- Open git bash or terminal, type: git -- version



```
Russell IT Spare@RLIT-SPARE01 MINGW64 ~
$ git --version
git version 2.35.0.windows.1

Russell IT Spare@RLIT-SPARE01 MINGW64 ~
$
```

Work with Git

Command line tools

windows: git bash

mac: terminal

What we will do

Config git

Clone a repository

Making Changes locally

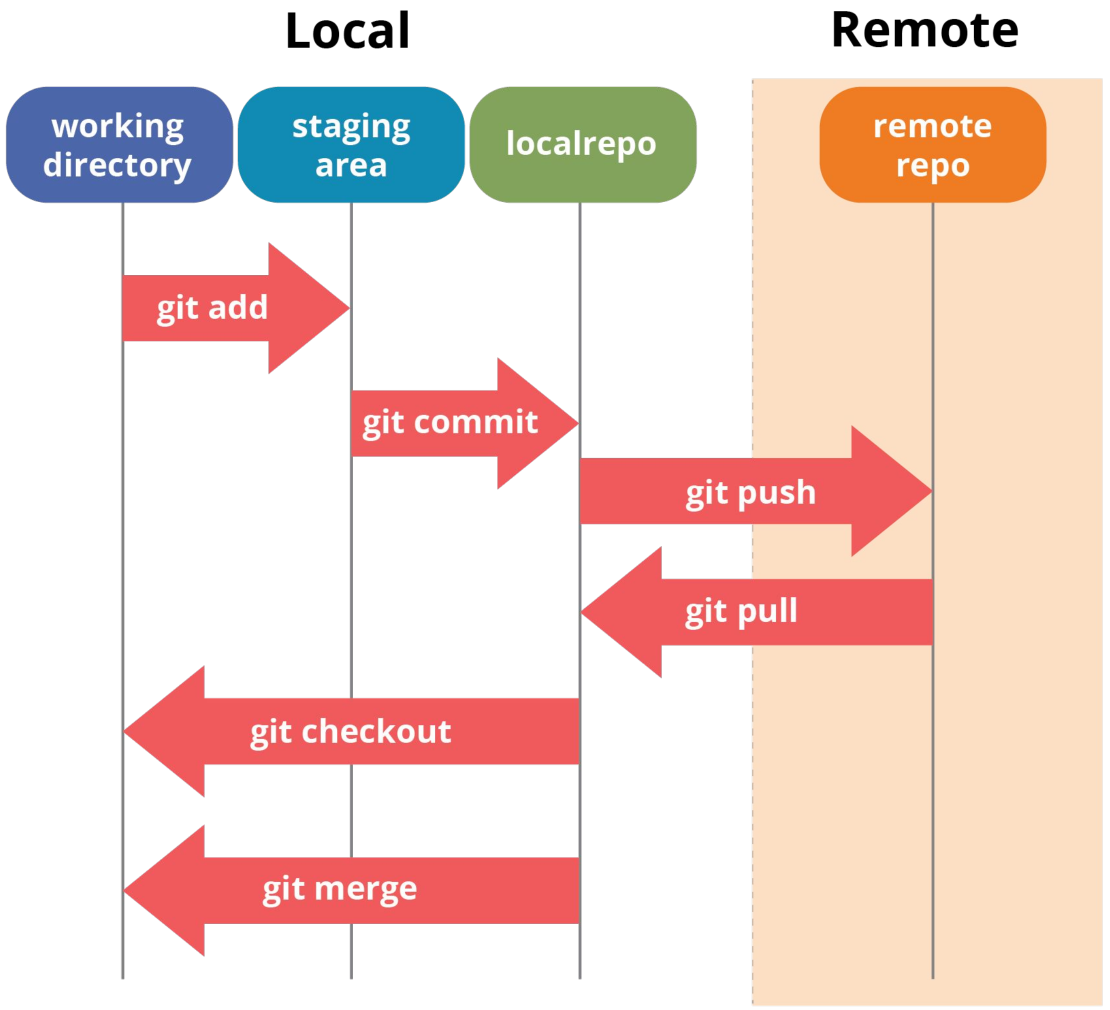
Pushing Changes to GitHub

Pull Changes and Manage Upstream

Handle Merges and Conflicts

https://education.github.com/git-cheat-sheet-education.pdf

Git Cheat Sheet



https://www.edureka.co/blog/git-tutorial/

Concepts in Git

1. Workspace (Your Camera Roll)

The **workspace** is like your **camera roll** where all your photos are stored.

- You take new photos, edit existing ones, or delete some as you go.
- These photos are **not yet organized** or ready for sharing.

In Git, this is where you make changes to your files—add, edit, or delete—but the changes aren't yet tracked unless you explicitly stage them.

Concepts in Git

2. Staging Area (Your Selection Folder)

The **staging area** is like a **"selected photos" folder** where you collect the specific photos you want to include in your online album.

- After reviewing your camera roll, you pick and move the best photos into this folder.
- Only the photos in this folder will be added to the final album.

In Git, the `git add` command is like selecting the photos. You are saying, "These changes are ready to be committed."

Concepts in Git

3. Local Repository (Your Computer)

The **local repository** is like saving your selected photos to a special folder on your computer.

- Once you're happy with the selection, you save them as a snapshot of your project.
- This folder is your local copy of the album, but it isn’t shared yet.

In Git, the `git commit` command saves the staged changes to your local repository. This creates a version of your project that you can refer back to later.

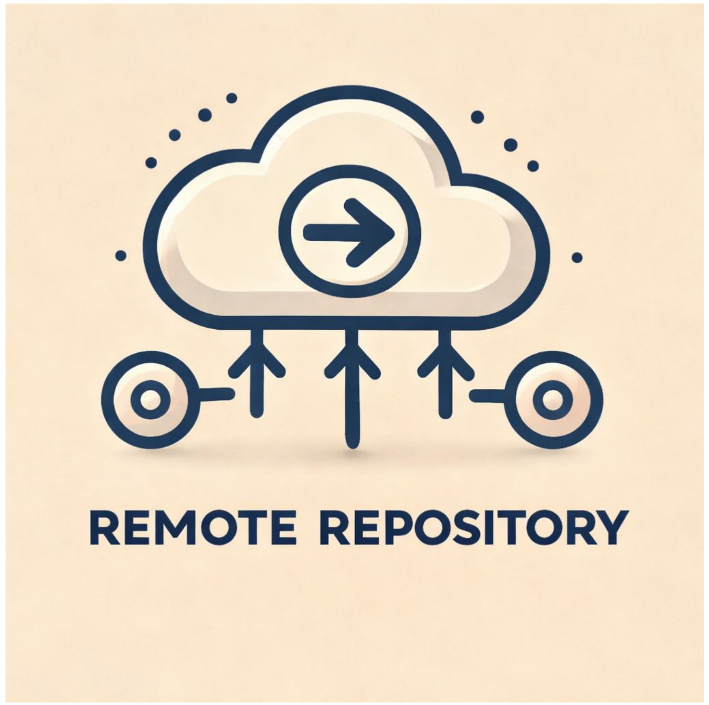
Concepts in Git

4. Remote Repository (The Online Album)

The **remote repository** is like your **online album** where you upload the photos to share with friends.

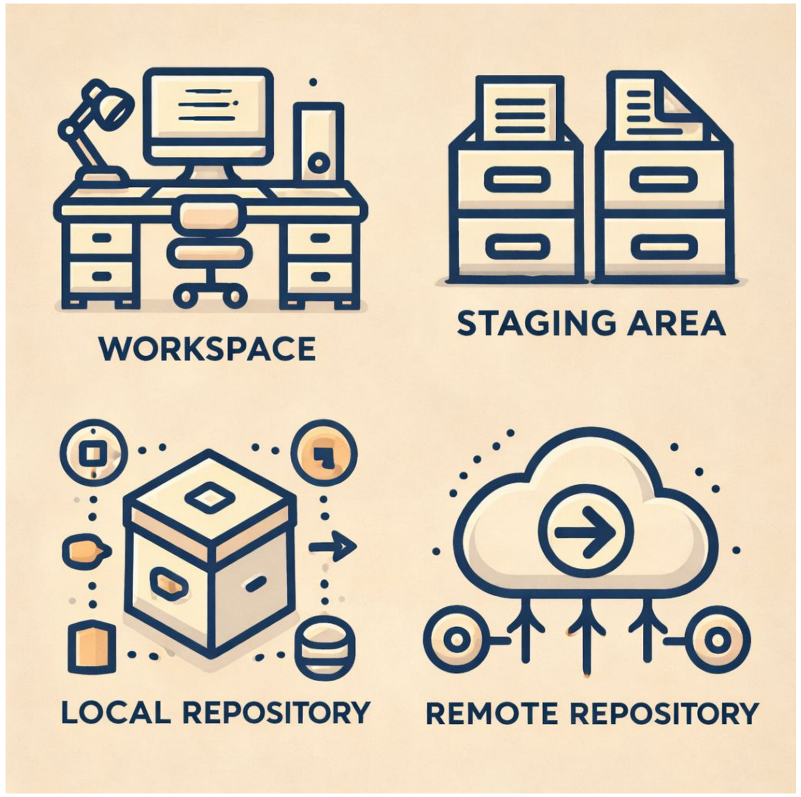
- Once the photos are saved on your computer (local repository), you can upload them to the cloud or an online platform.
- Now others can view and collaborate on the album.

In Git, the `git push` command uploads your committed changes to a remote repository like GitHub or Bitbucket.



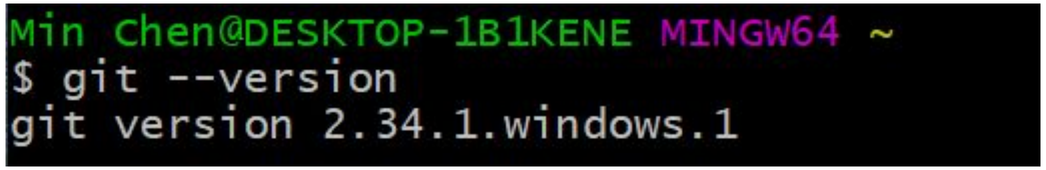
Summary

Git Concept	Example	Purpose
Workspace	Camera Roll	Where you make and edit changes (files).
Staging Area	Selected Photos Folder	Prepares the changes/files for the next commit.
Local Repository	Computer Folder	Saves a snapshot/version of the project.
Remote Repository	Online Album	Shares your project with others.



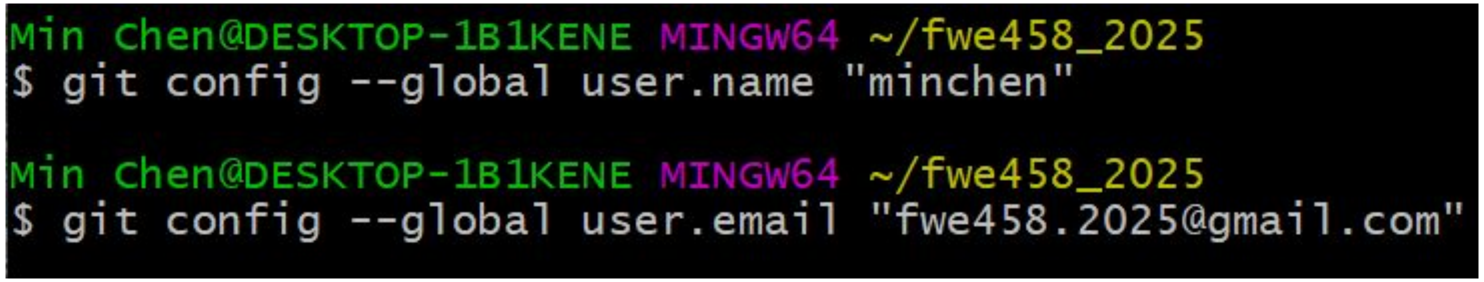
Start git!

Using Git with Command Line: (Mac Terminal; Windows: Git Bash)



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~
$ git --version
git version 2.34.1.windows.1
```

Let Git know who you are (registration)



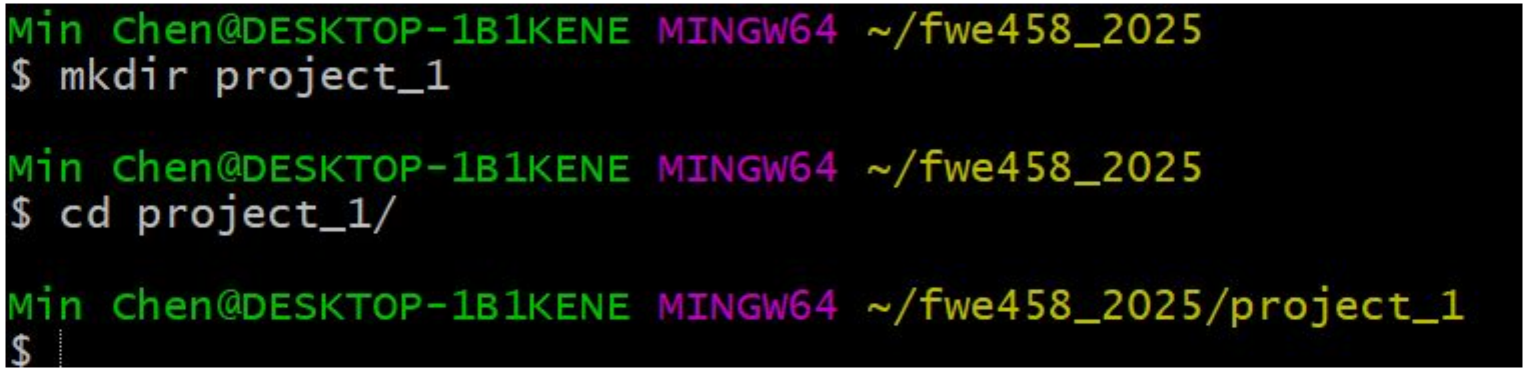
```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025
$ git config --global user.name "minchen"

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025
$ git config --global user.email "fwe458.2025@gmail.com"
```

Use the `--global` flag to set the username and email for all repositories on your computer.

If you want to set the username and email for only the current repository, omit the `--global` flag.

Git practices

Make a project folder

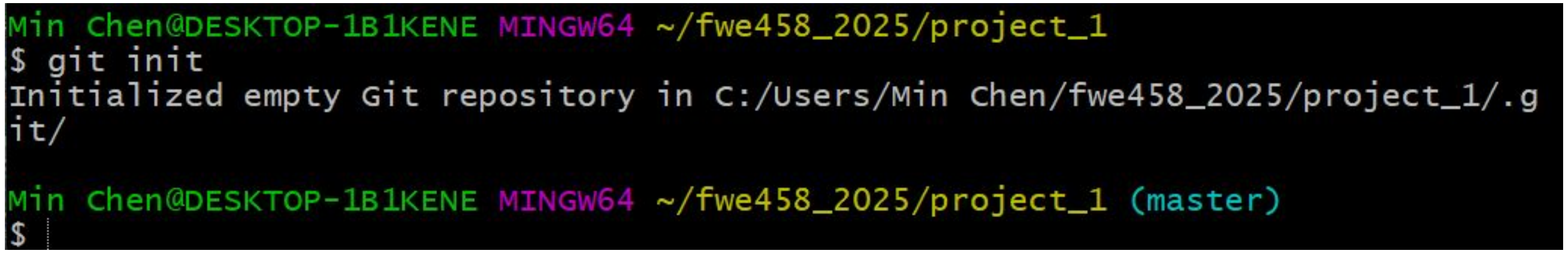


```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025
$ mkdir project_1

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025
$ cd project_1/

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1
$
```

If you already have a folder or directory you'd like to use with Git:

1. **Command Line**: Navigate to the folder using the command line (e.g., `cd path/to/your/folder`).
2. **File Explorer**: Open the folder in your file explorer, right-click inside it, and select **"Git Bash Here"** to open Git Bash directly in that directory.

Initialize Git



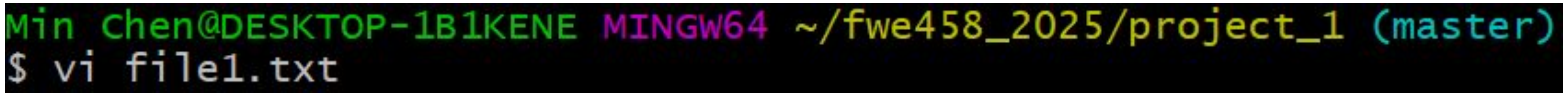
```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1
$ git init
Initialized empty Git repository in C:/Users/Min Chen/fwe458_2025/project_1/.g
it/

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$
```

When you initialize Git in a folder, Git starts tracking the folder and its contents.

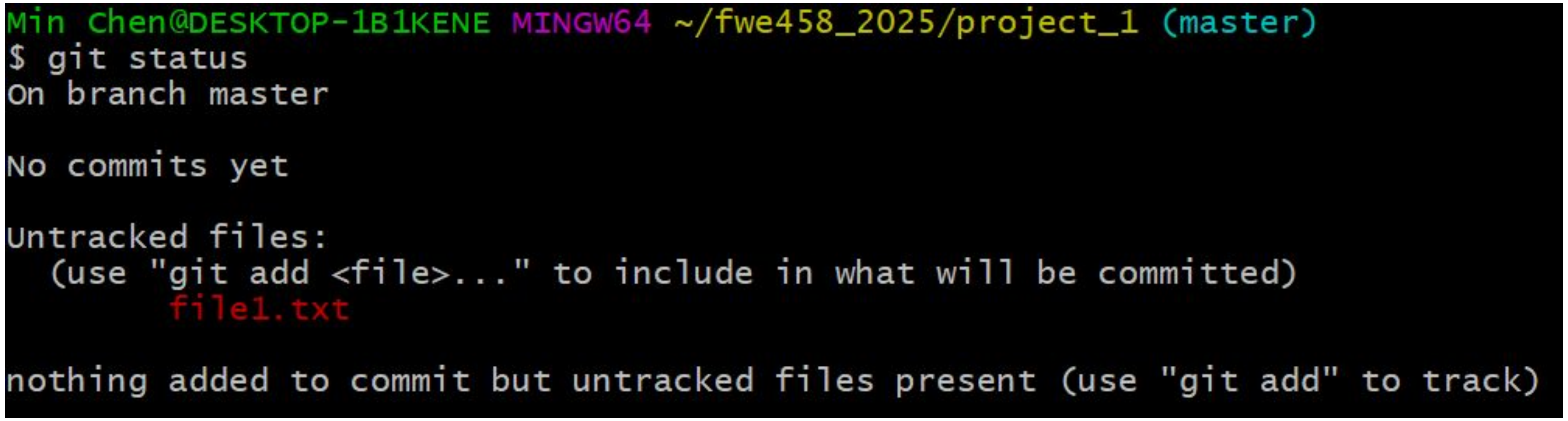
Git creates a hidden directory named `.git` inside the folder. This directory stores all the information Git needs to track changes, manage commits, and maintain version history.

Git practices

Edit files



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ vi file1.txt
```



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Git is now aware of the file, but it hasn't been added to the repository yet!

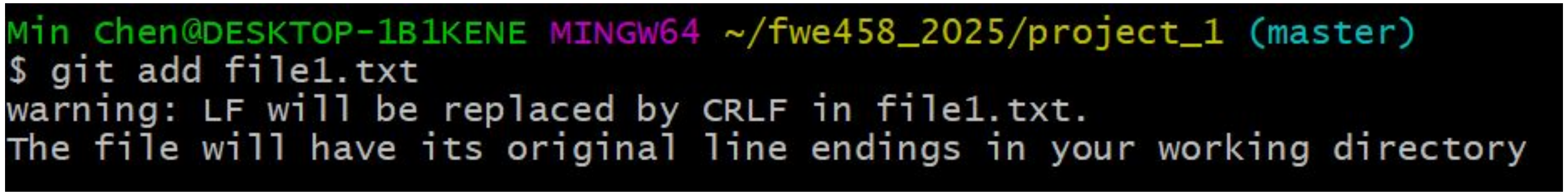
Files in your Git repository can be in one of two states:

1. **Tracked**: Files that Git is monitoring. These files have been committed or staged, and Git tracks their changes.
2. **Untracked**: Files in your working directory that Git is not yet monitoring. Git recognizes their existence but does not track any changes.

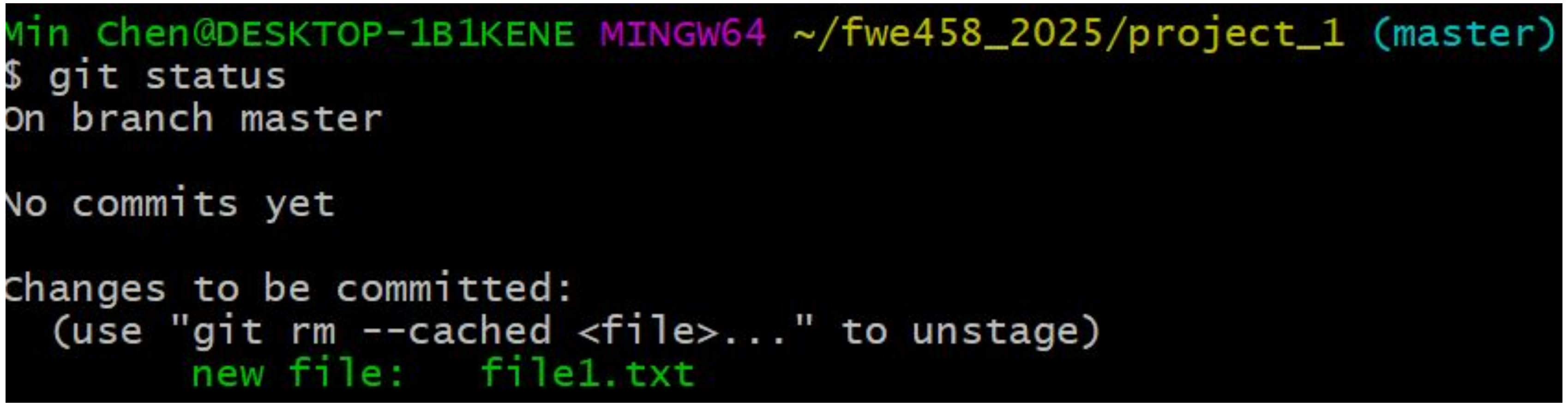
When you first create a repository, all files are untracked. To start tracking them, you need to **stage** them by adding them to the **staging area**. This step prepares the files to be included in your next commit. You can stage files using the `git add` command.

Git practices

Stage files



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ git add file1.txt
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory
```



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt
```

You can also stage multiple files at once using the following commands:

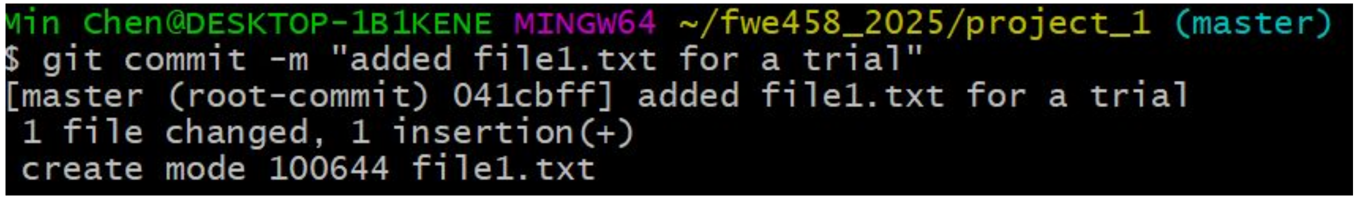
git add .

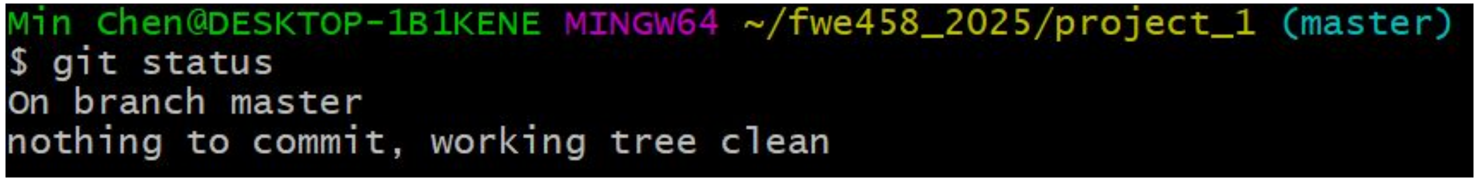
git add –all

git add -A

Git practices

Git commit


```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ git commit -m "added file1.txt for a trial"
[master (root-commit) 041cbff] added file1.txt for a trial
 1 file changed, 1 insertion(+)
 create mode 100644 file1.txt
```


```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Now that we've finished our work, we're ready to move changes from the staging area to a commit in our repository.

Commits serve as checkpoints that track our progress and record changes as we work. Each commit represents a "save point" in the project. If you encounter a bug or need to revisit an earlier state, you can easily return to a specific commit.

When creating a commit, it’s important to include a clear message.

You can use git commit -a -m "Commit message" to skip the staging step. But highly not recommended!!

Git Branch

In Git, a **branch** is like a separate version of your main project, allowing you to work on updates or fixes without impacting the main codebase.

Let’s break this down with an example: Imagine you’re working on a large project and need to update its design.

How Would This Work Without Git?

1. **Create File Copies**:
   You make copies of all relevant files to avoid affecting the live version.
2. **Start Updating the Design**:
   As you work, you realize other dependent files also need changes.
   You copy and update those files too, ensuring dependencies reference the correct filenames.
3. **Handle Emergencies**:
   - Suddenly, there’s an unrelated critical error in the project.
   - You save all your current work and note the file names of your copies.
   - Fix the unrelated error in the original files.
4. **Resume Design Work**:
   - Go back to your design updates, finish them, and update the live version by copying or renaming files.
5. **Problems Later**:
   - Two weeks later, you realize the critical error fix wasn’t included in the design version because it was based on old copies of the files.
   - Now, you need to repeat the fix in the design version as well.

Git Branch

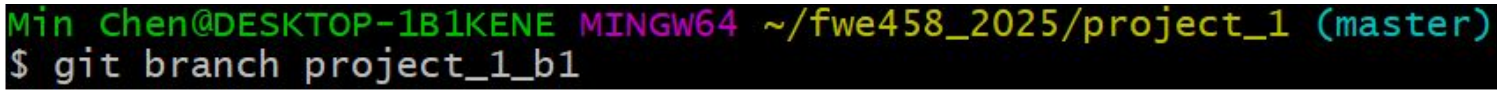
How Would This Work With Git?

1. **Create a Branch for the Design**:
   - Create a new branch called `new-design` and edit the code directly without affecting the main branch.
2. **Handle Emergencies Easily**:
   - If a critical error arises, create another branch called `small-error-fix` from the main branch.
   - Fix the error in the `small-error-fix` branch and merge it into the main branch.
3. **Continue Design Work**:
   - Switch back to the `new-design` branch and finish the design updates.
   - When you merge `new-design` into `main`, Git automatically alerts you to changes made in the `small-error-fix` branch, ensuring no fixes are missed.

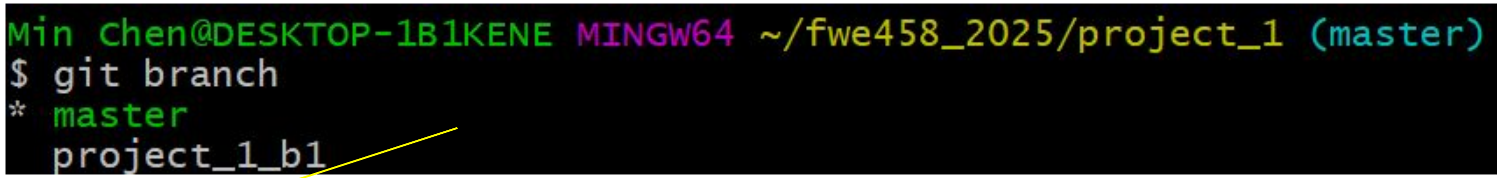
Why Use Git Branches?

- **Isolation**: Each branch is an independent workspace, so changes in one branch don't affect others.
- **Merging**: Once the work is done, you can combine branches and resolve any conflicts easily.
- **Switching**: Move between branches effortlessly to work on different parts of the project simultaneously.
- **Efficiency**: Branches in Git are lightweight and fast, making them ideal for both small fixes and large-scale updates.

Work on Git Branch

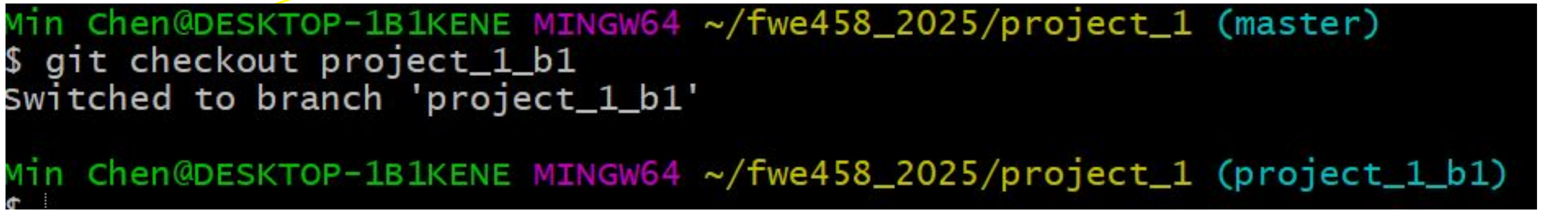


```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ git branch project_1_b1
```



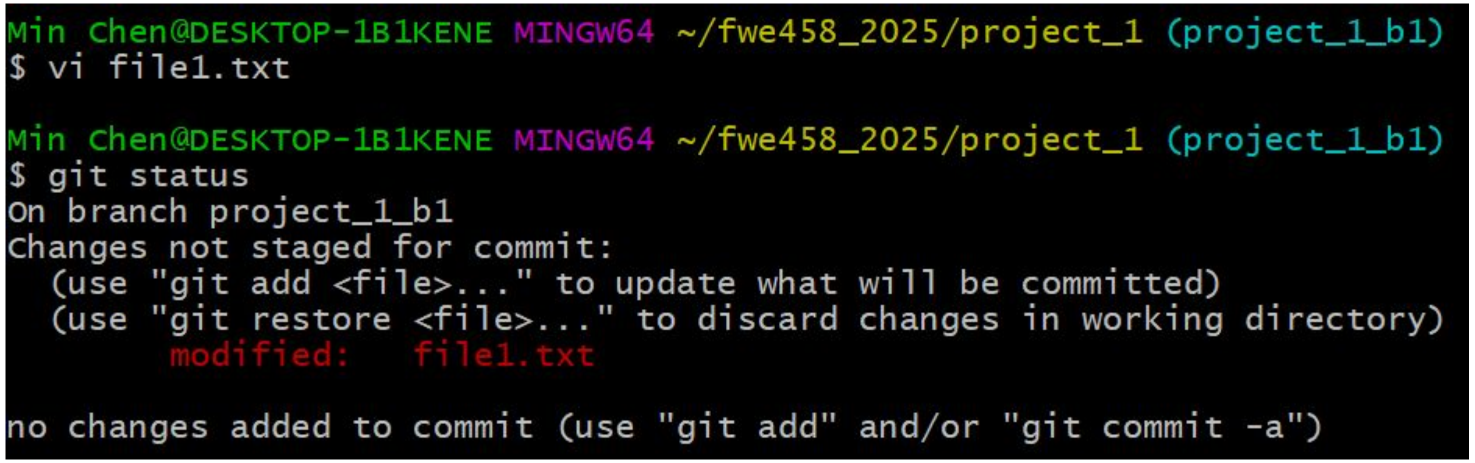
```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ git branch
* master
  project_1_b1
```

We can see the new branch with the name "hello-world-images", but the `*` beside `master` specifies that we are currently on that `branch`.



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ git checkout project_1_b1
Switched to branch 'project_1_b1'

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (project_1_b1)
$
```

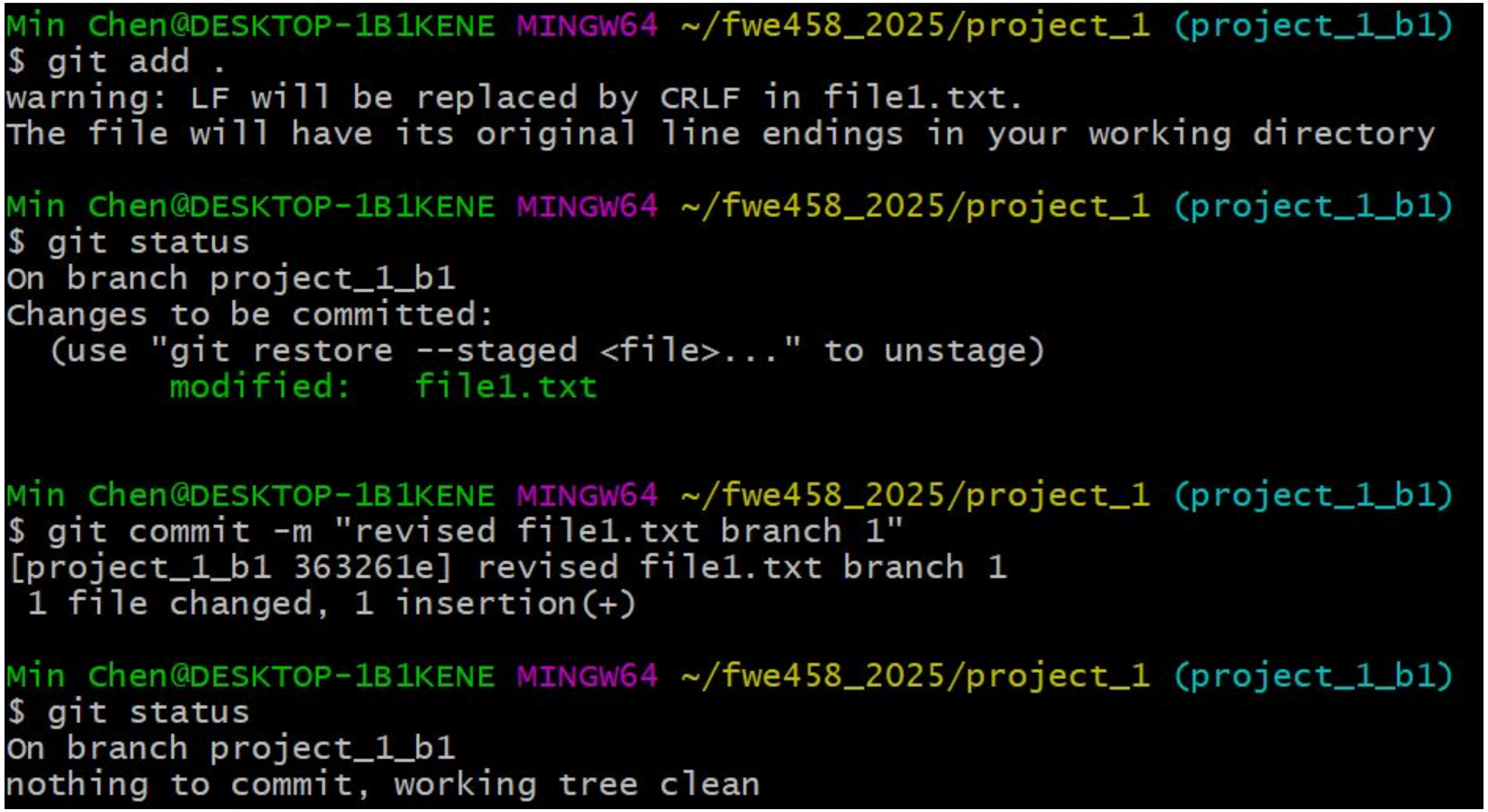
`checkout` is the command used to check out a `branch`. Moving us from the current `branch`, to the one specified at the end of the command:



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (project_1_b1)
$ vi file1.txt

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (project_1_b1)
$ git status
On branch project_1_b1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

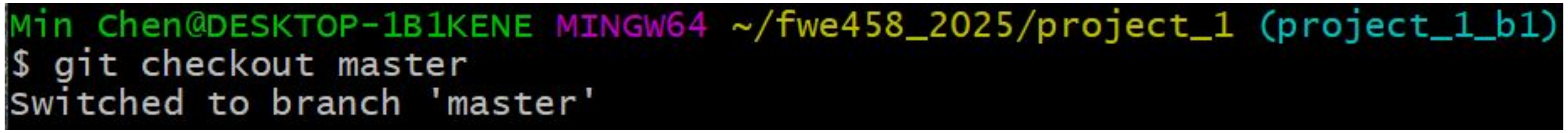
Made some changes

Work on Git Branch



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (project_1_b1)
$ git add .
warning: LF will be replaced by CRLF in file1.txt.
The file will have its original line endings in your working directory

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (project_1_b1)
$ git status
On branch project_1_b1
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.txt


Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (project_1_b1)
$ git commit -m "revised file1.txt branch 1"
[project_1_b1 363261e] revised file1.txt branch 1
 1 file changed, 1 insertion(+)

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (project_1_b1)
$ git status
On branch project_1_b1
nothing to commit, working tree clean
```

Switch between branches

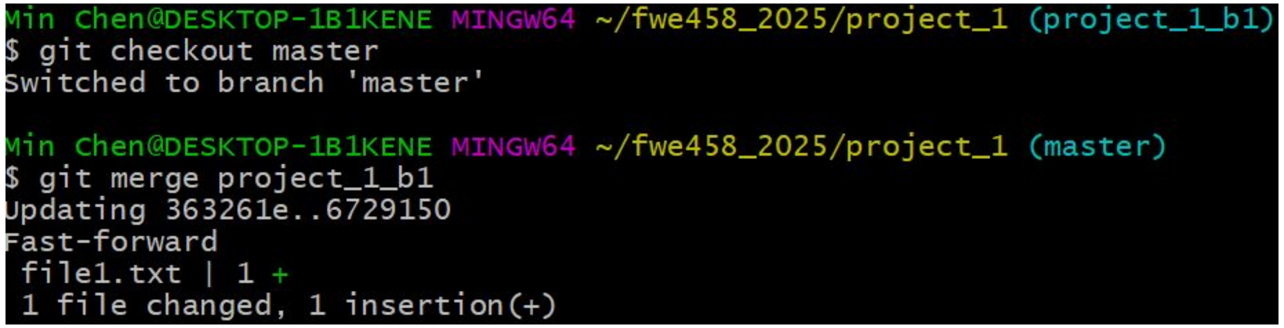


```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (project_1_b1)
$ git checkout master
Switched to branch 'master'
```

Alternative command: git switch -c <branch-name>

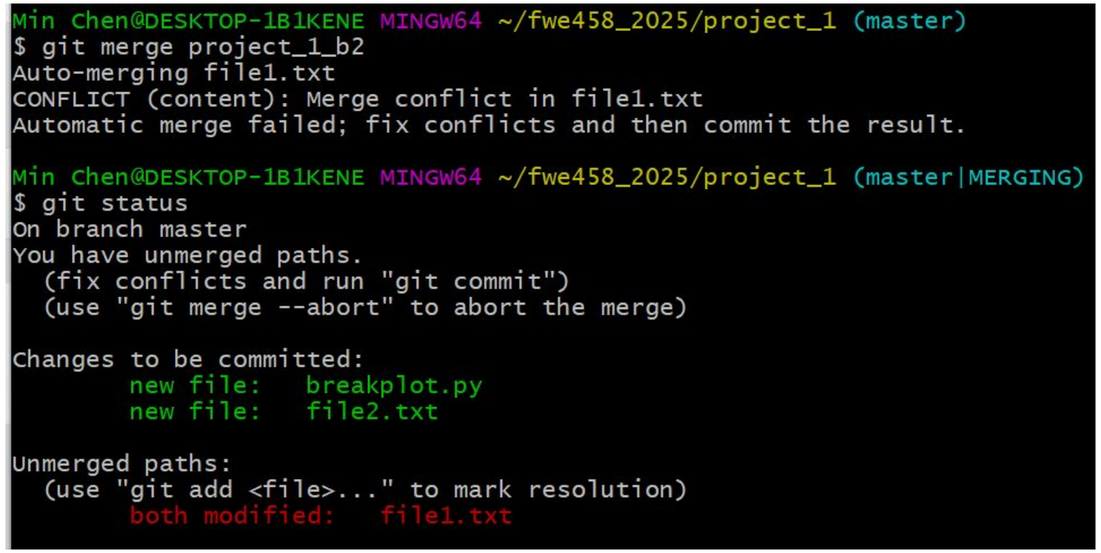
You can delete a branch by git branch -d <branch-name>

Merge



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (project_1_b1)
$ git checkout master
Switched to branch 'master'

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ git merge project_1_b1
Updating 363261e..6729150
Fast-forward
 file1.txt | 1 +
 1 file changed, 1 insertion(+)
```

A **merge conflict** occurs in Git when two branches being merged have conflicting changes in the same part of the same file. Git cannot automatically determine which change to keep, so it requires human intervention to resolve the conflict.



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ git merge project_1_b2
Auto-merging file1.txt
CONFLICT (content): Merge conflict in file1.txt
Automatic merge failed; fix conflicts and then commit the result.

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master|MERGING)
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:
        new file:   breakplot.py
        new file:   file2.txt

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   file1.txt
```

When a merge conflict occurs, Git:

- Stops the merge process.
- Marks the conflicting areas in the affected file(s) like this:

text
CopyEdit

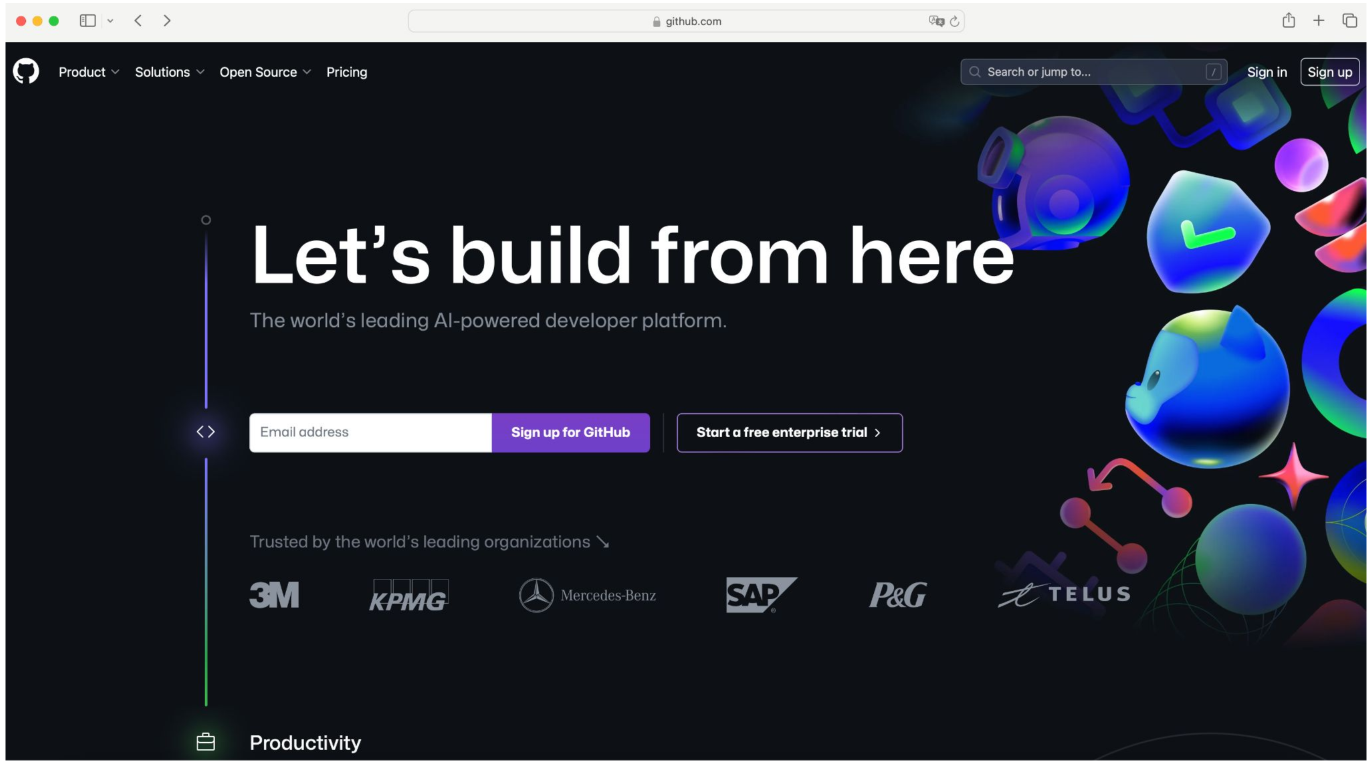
```
<<<<<<< HEAD
Welcome to the Git tutorial!
=======
Git is a powerful tool for collaboration.
>>>>>>> branch-name
```

- The part between `<<<<<<< HEAD` and `=======` represents the changes from your current branch.
- The part between `=======` and `>>>>>>> branch-name` represents the changes from the branch being merged.
- resolve it; stage and commit your new edits
- delete the new branch by **git branch -d <branch-name>**

Create your GitHub account

Navigate to https://github.com

Follow the steps after clicking "Sign Up"



Create your repository on your GitHub

Sign In to Your Account:

- Log in to your GitHub account.

Go to Repository Creation Page:

- Once logged in, you can create a new repository by clicking the "+" icon in the top right corner of the GitHub interface, then selecting "New repository".

Repository Details:

- Repository Name: Give your repository a name. This name should be descriptive and indicate what the repository is for.
- Description (Optional): Provide a brief description of your repository. This helps others understand what your repository is about.
- Visibility: Choose whether the repository is Public (anyone can see this repository) or Private (only you and collaborators you choose can see it).

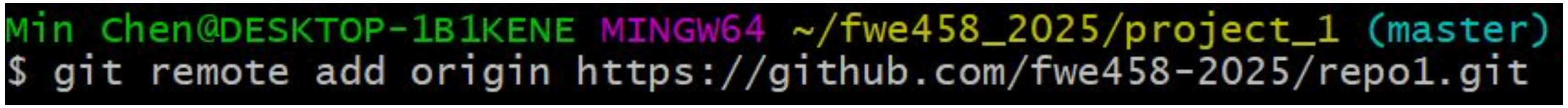
Initialize the Repository (Optional):

- You can choose to initialize the repository with a README, which is a document explaining what your repository is about.
- You might also want to add a .gitignore file, which tells Git which files or folders to ignore in a project.
- Optionally, you can choose a license for your repository. If you’re creating an open-source project, this is important to specify.

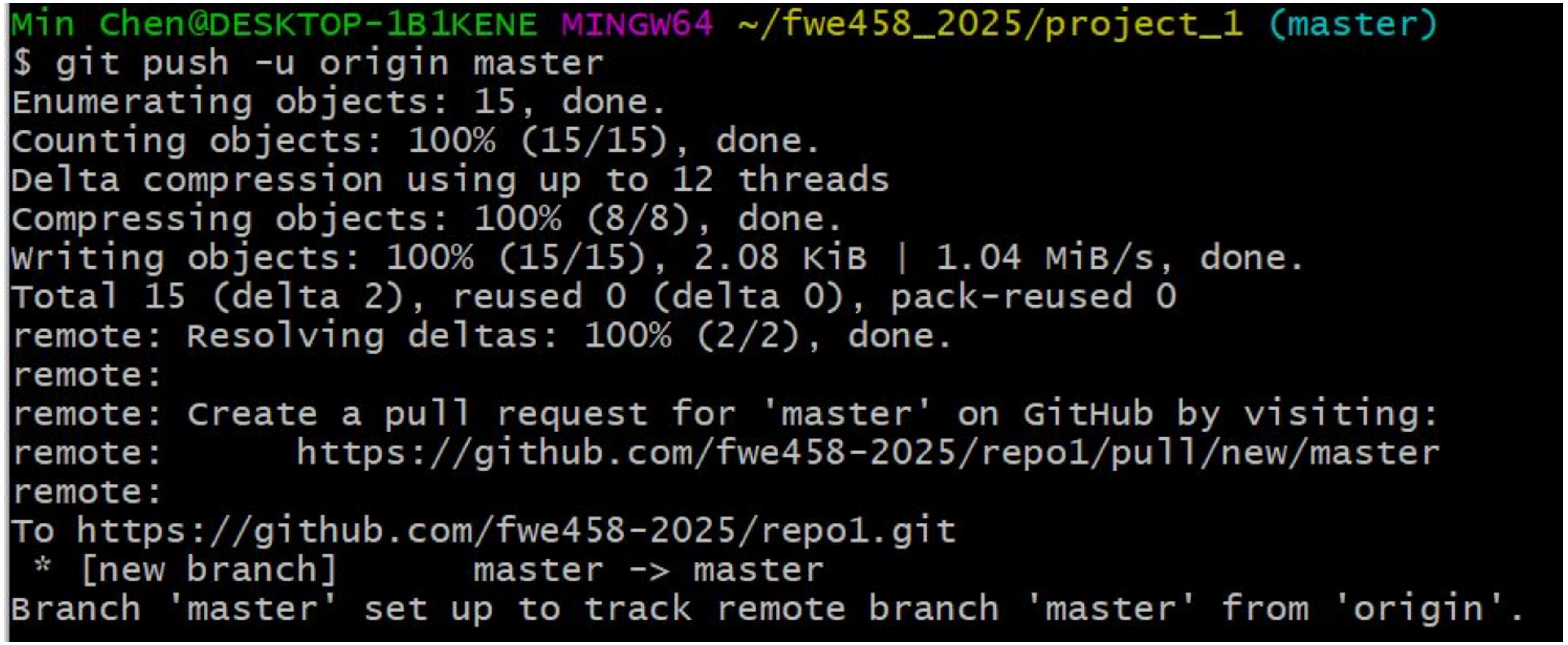
Create Repository:

- Click the "Create repository" button to create your new repository.

Push to GitHub

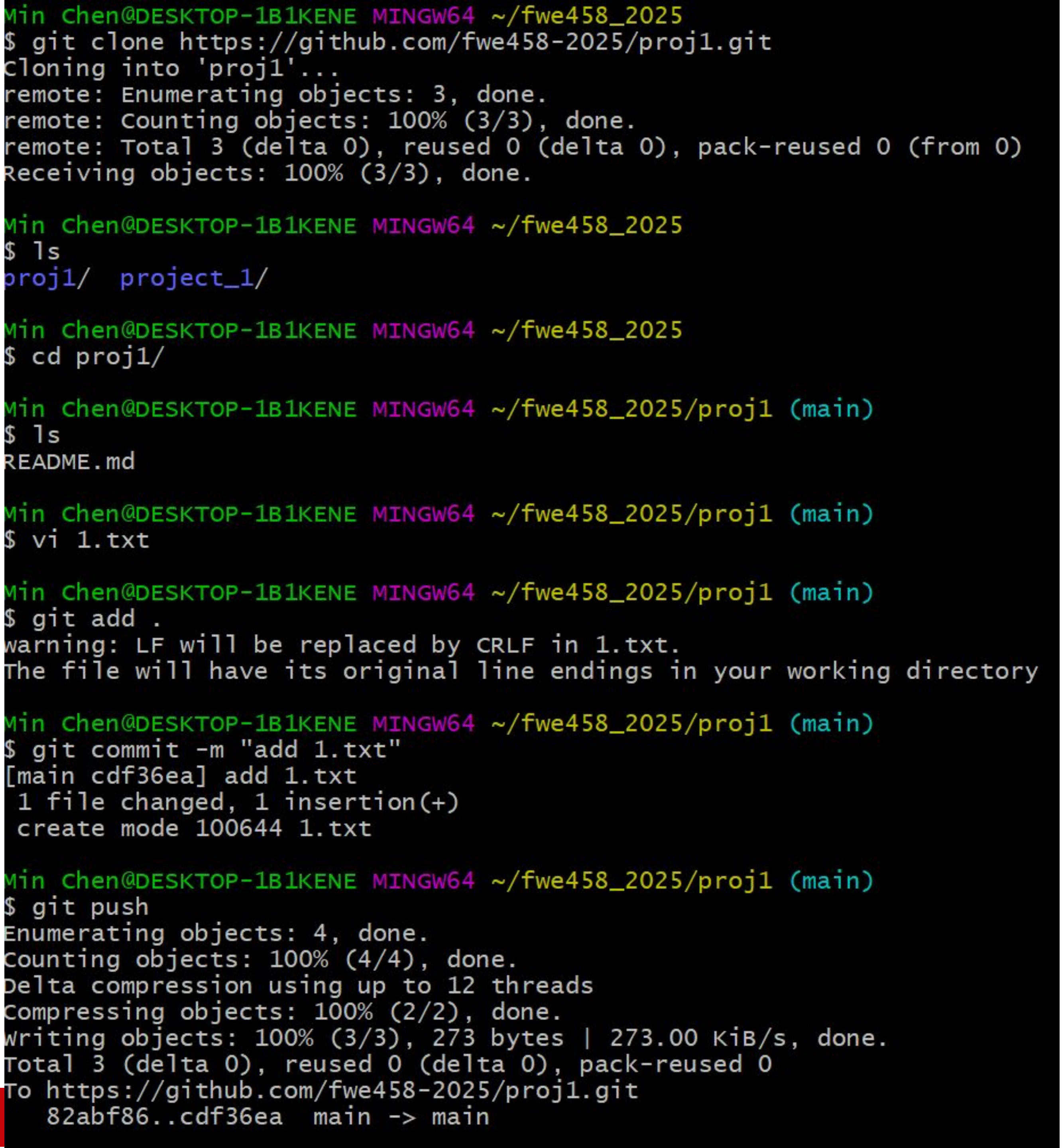


```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ git remote add origin https://github.com/fwe458-2025/repo1.git
```



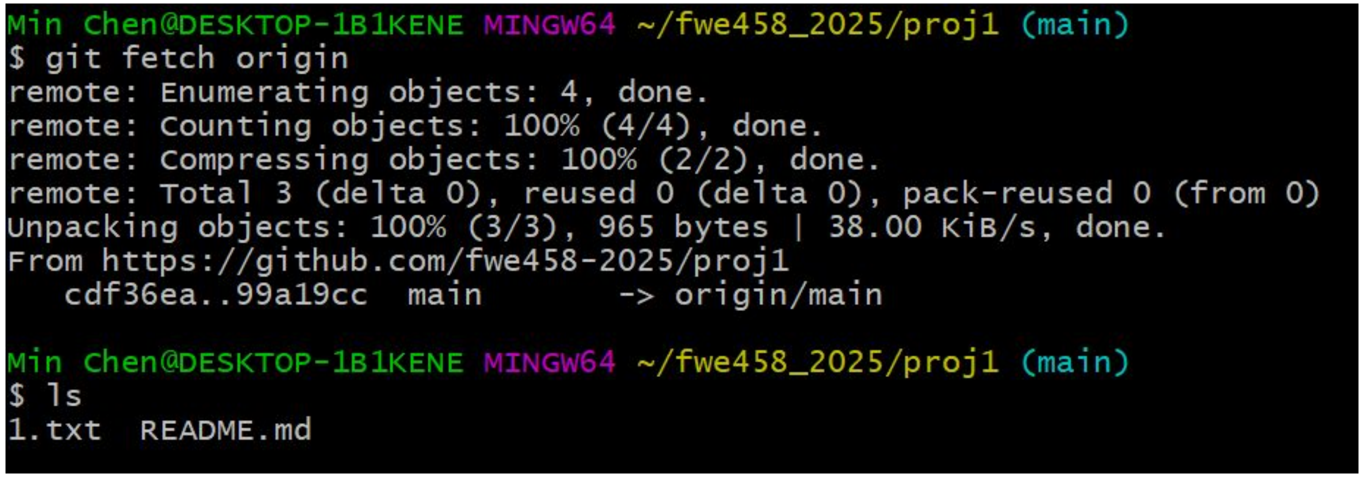
```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/project_1 (master)
$ git push -u origin master
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 12 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (15/15), 2.08 KiB | 1.04 MiB/s, done.
Total 15 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), done.
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:      https://github.com/fwe458-2025/repo1/pull/new/master
remote:
To https://github.com/fwe458-2025/repo1.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Work with Github

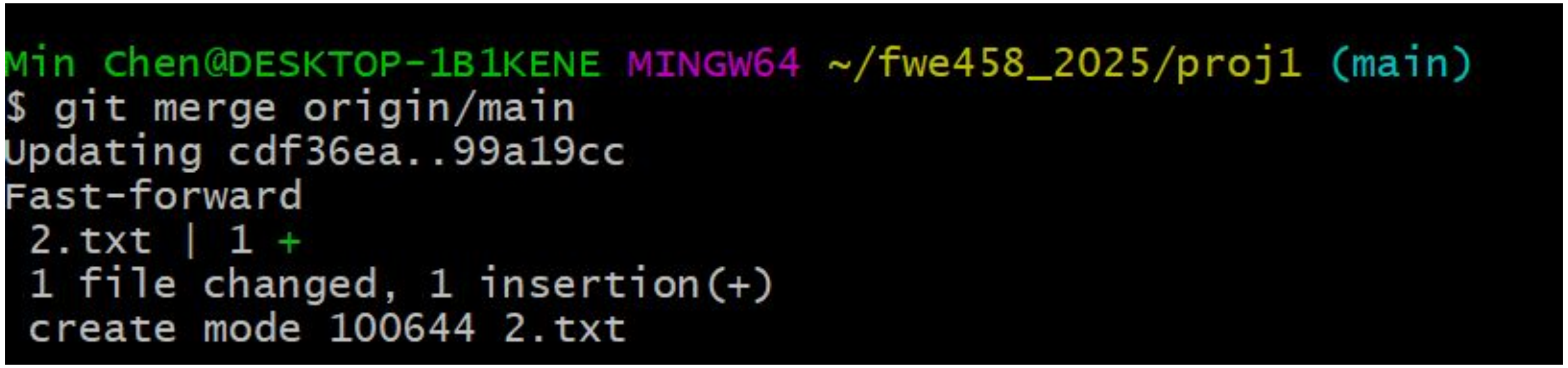


```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025
$ git clone https://github.com/fwe458-2025/proj1.git
Cloning into 'proj1'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (3/3), done.

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025
$ ls
proj1/  project_1/

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025
$ cd proj1/

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ ls
README.md

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ vi 1.txt

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ git add .
warning: LF will be replaced by CRLF in 1.txt.
The file will have its original line endings in your working directory

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ git commit -m "add 1.txt"
[main cdf36ea] add 1.txt
 1 file changed, 1 insertion(+)
 create mode 100644 1.txt

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 273 bytes | 273.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/fwe458-2025/proj1.git
   82abf86..cdf36ea  main -> main
```

Git Pull

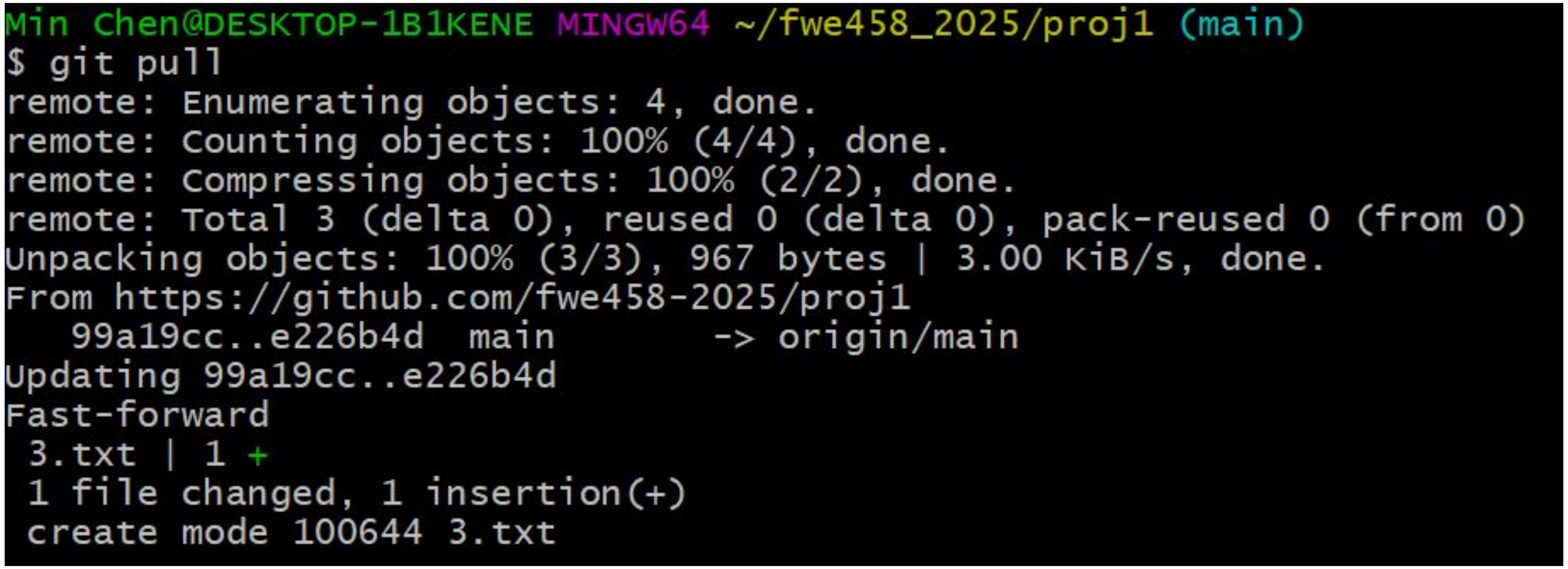


```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ git fetch origin
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 965 bytes | 38.00 KiB/s, done.
From https://github.com/fwe458-2025/proj1
   cdf36ea..99a19cc  main       -> origin/main

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ ls
1.txt  README.md
```



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ git merge origin/main
Updating cdf36ea..99a19cc
Fast-forward
 2.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 2.txt
```

Git Pull



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 967 bytes | 3.00 KiB/s, done.
From https://github.com/fwe458-2025/proj1
   99a19cc..e226b4d  main       -> origin/main
Updating 99a19cc..e226b4d
Fast-forward
 3.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 3.txt
```

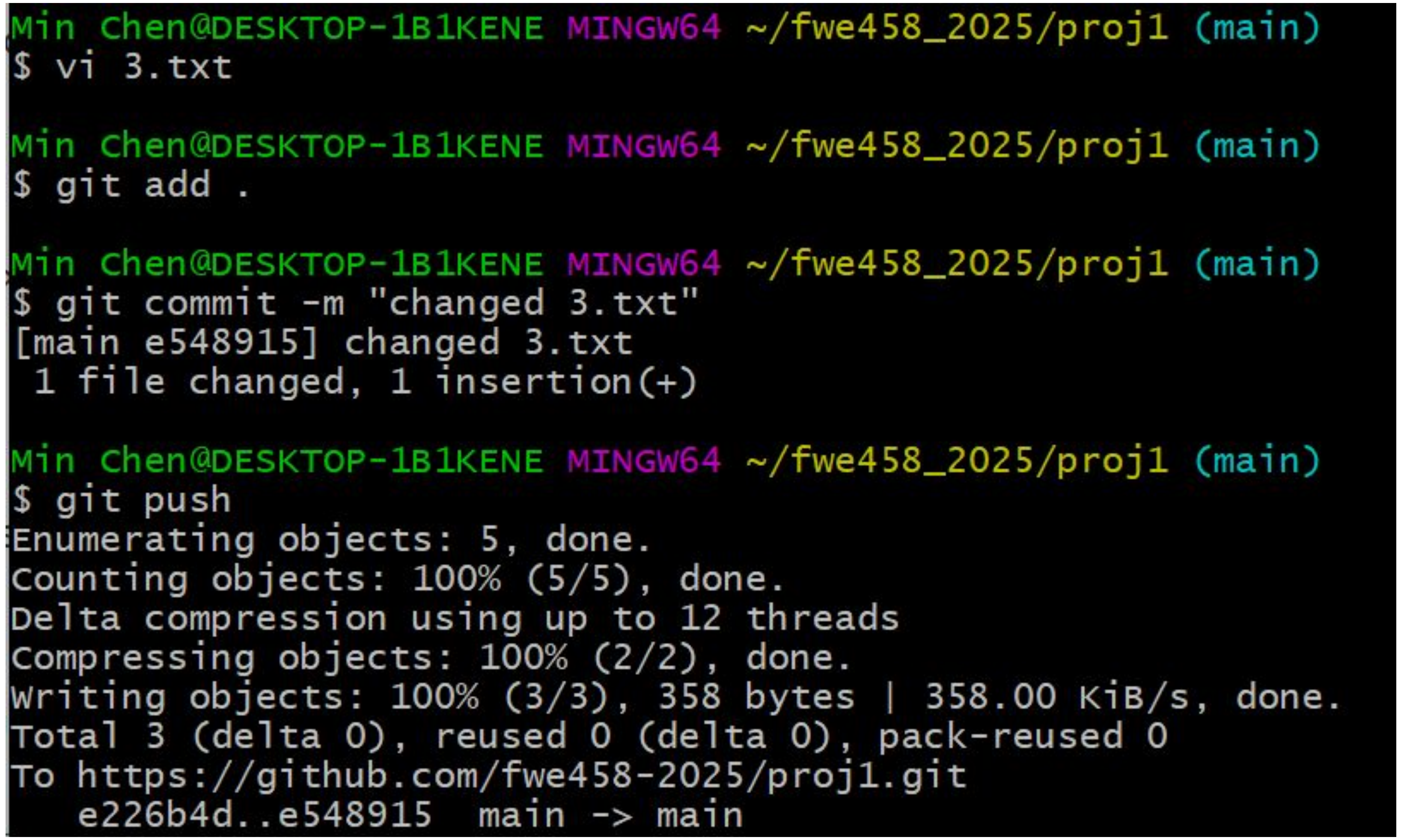
Any time you start working on a project, you should get the most recent changes to your local copy.

With Git, you can do that with `pull`.

`pull` is a combination of 2 different commands:

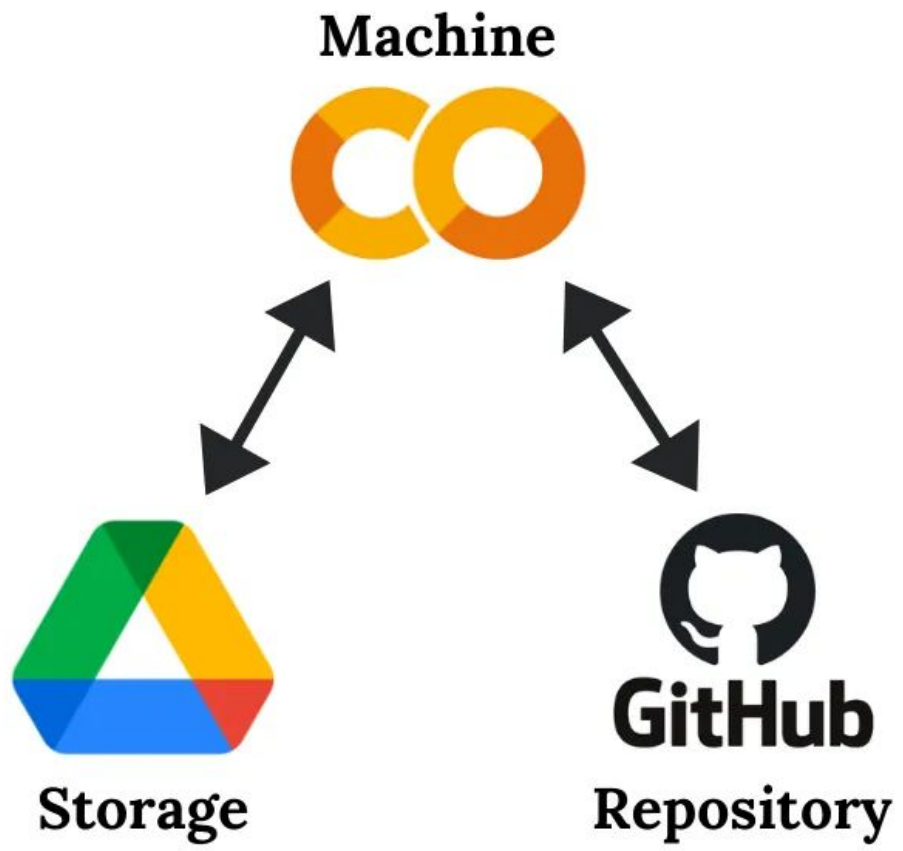
- `fetch`
- `merge`

Local Change and Git Push



```
Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ vi 3.txt

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ git add .

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ git commit -m "changed 3.txt"
[main e548915] changed 3.txt
 1 file changed, 1 insertion(+)

Min Chen@DESKTOP-1B1KENE MINGW64 ~/fwe458_2025/proj1 (main)
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 358 bytes | 358.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/fwe458-2025/proj1.git
   e226b4d..e548915  main -> main
```

Use Google Colab with GitHub via Google Drive



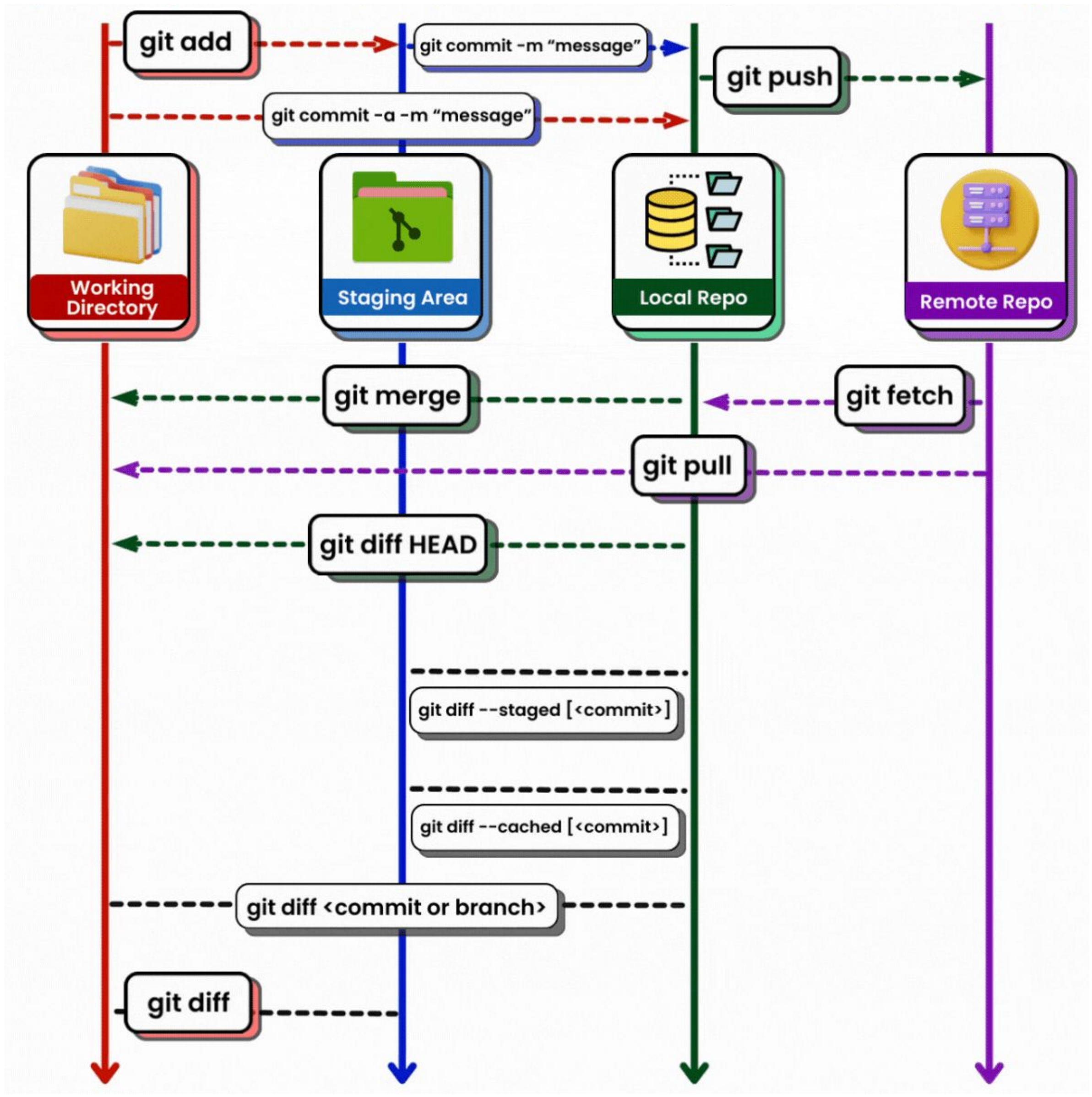
- **Colab** enables anyone to write and run Python code of any kind directly in a web browser.
- **GitHub** serves as a platform for hosting code, facilitating both version control and collaborative work. It provides a space where individuals can collectively contribute to projects from any location, ensuring seamless teamwork while maintaining the integrity of the original project.
- **Google Drive** is a service for storing and synchronizing files. It allows users to save their files on cloud servers, keep files updated across various devices, and share them with others.

https://medium.com/analytics-vidhya/how-to-use-google-colab-with-github-via-google-drive-68efb23a42d

Next Lecture: Fundamentals of Python

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Git Workflow



Source: Brij kishore Pandey on LinkedIn