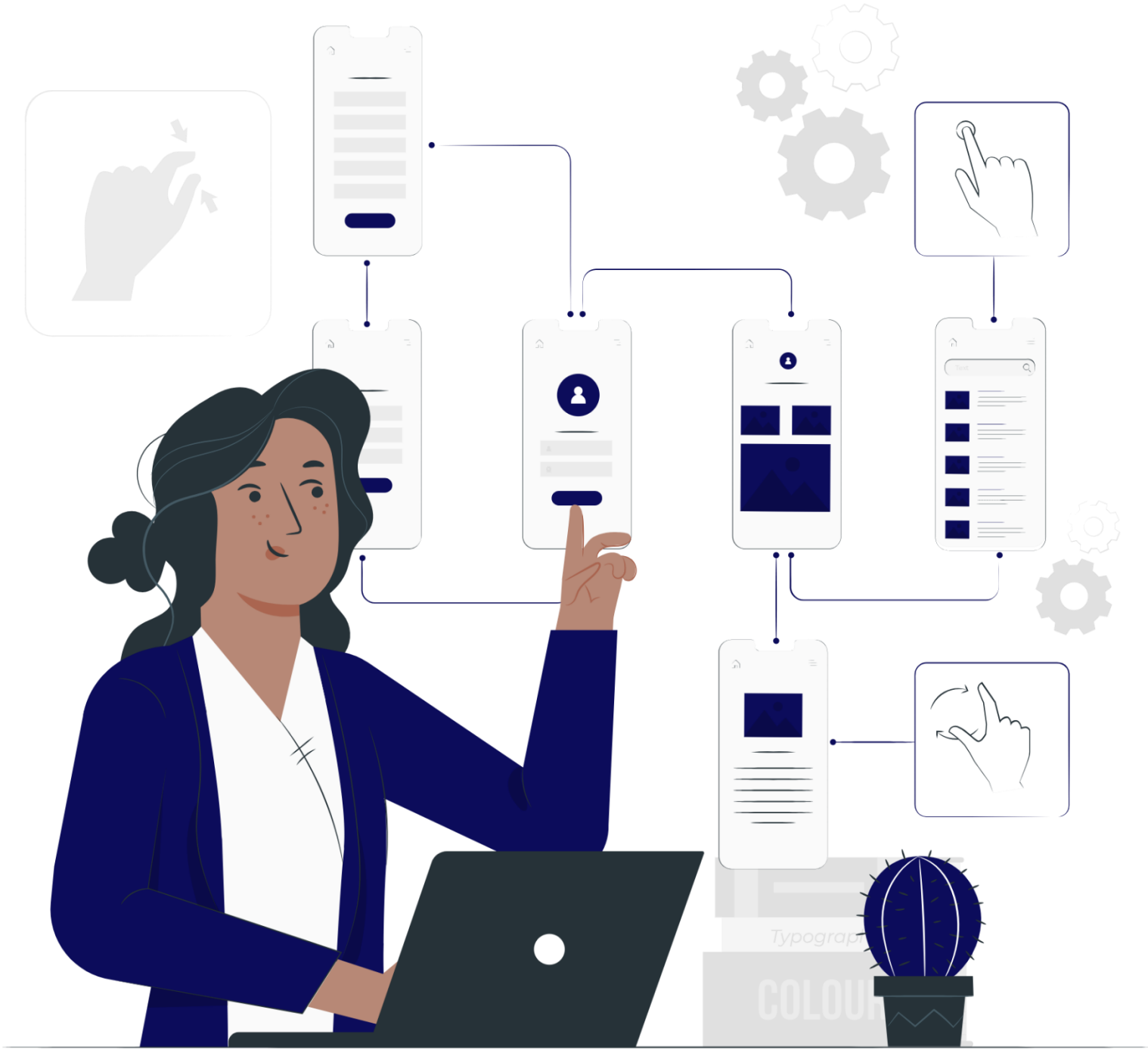
Git and Version Control


Typograph
COLOUR

Welcome to your first step in becoming a proficient developer! This week, we'll equip you with Git, an essential tool for managing your code, collaborating with others, and protecting your work. Think of Git as a powerful "time machine" for your code that also allows multiple people to work on the same project without chaos.

---

1. The Problem Git Solves: Why Version Control?

Imagine you're writing an important essay. You create `essay.docx`. Then you make changes and save it as `essay_final.docx`. More changes: `essay_final_v2.docx`. Then your friend edits it: `essay_final_v2_friend_edits.docx`. Soon, you have a folder full of files, no idea which is the absolute latest, what changed between versions, or what your friend did. If you accidentally delete a paragraph, can you easily get it back? This is the chaos of manual versioning.

Version Control Systems (VCS) are software tools that solve this problem for code (and any text-based files).

1.1 What is a Version Control System (VCS)?

At its core, a VCS is a system that:

- **Tracks Changes:** It records every modification made to your files over time, rather than just saving new versions. This creates a detailed history.
- **Allows Collaboration:** It enables multiple people to work on the same project simultaneously without overwriting each other's work. It helps merge their individual changes together.

- **Provides a Safety Net:** You can easily revert files or even the entire project back to a previous, stable state if something breaks. No more fear of losing work!
- **Compares Versions:** You can see exactly what changed between any two points in your project's history.
- **Identifies Authorship:** You can see who made specific changes and when.

1.2 Types of VCS: Centralized (CVCS) vs. Distributed (DVCS)

There are two main philosophies for how VCS works:

- **Centralized Version Control Systems (CVCS) - (e.g., SVN, Perforce)**
    - **Analogy:** Imagine a single, giant library (the central server) that holds all versions of every book. To read or change a book, you must "check it out" from the library. Once you're done, you "check it in" back to the library.
    - **How it works:** All versioned files are stored on a central server. Developers "check out" files, make changes, and then "check in" their updated files.
    - **Pros:** Simpler to understand initially, as there's one definitive "source of truth."
    - **Cons:**
        - **Single Point of Failure:** If the central server goes down, no one can work or save changes. All history is lost if the server's data is corrupted.
        - **Requires Network Access:** You must be connected to the server to perform most VCS operations.
        - **Scalability Challenges:** Can become slow for very large teams or projects.
- **Distributed Version Control Systems (DVCS) - (e.g., Git, Mercurial)**
    - **Analogy:** Instead of a single library, imagine everyone gets their *entire own copy* of the library (the full project history). You work on your copy, and when you want to share changes, you can

exchange copies directly with others, or push your changes to a designated "remote" copy that everyone uses.

- **How it works:** Every developer has a complete clone (a full copy) of the entire repository, including its full history, on their local machine. Changes are committed locally first, then pushed to a shared remote repository (e.g., GitHub).
- **Pros:**
    - **No Single Point of Failure:** Even if the shared remote server goes down, every developer has a full backup on their machine.
    - **Faster Operations:** Most operations (committing, viewing history) are done locally, so they are incredibly fast.
    - **Offline Work:** You can commit changes and manage your project history even without an internet connection.
    - **Excellent for Branching & Merging:** Designed from the ground up to handle complex parallel development flows seamlessly.
    - **Flexibility:** You can set up multiple remote repositories if needed.
- **Cons:cc**
    - Slightly steeper initial learning curve due to the distributed nature and multiple "areas" (which we'll cover next).

Why Git is a DVCS:
Git is a Distributed Version Control System. This fundamental design choice is why Git is so powerful, fast, and resilient, and why it has become the industry standard for managing code in modern software development.

---

2. The Core Git Workflow: Understanding the Areas

Before diving into commands, it's crucial to understand the four main areas where your code resides or is processed when you use Git. Think of it as a pipeline or a series of states your changes go through.

2.1 The Working Directory (or Working Tree)

- **What it is:** This is simply the directory (folder) on your computer where you're currently working. It contains the actual files that you see and edit.
- **What happens here:** You open your code editor (like VS Code), type, delete, and modify your project files. These changes are just ordinary files on your hard drive at this point.
- **Git's view:** Git sees these as "untracked" files (if new) or "modified" files (if existing files have changed).

2.2 The Staging Area (or Index)

- **What it is:** This is a crucial intermediate area in Git. It's not a physical folder, but rather a file that Git uses to keep track of changes that you've *prepared* to be part of your *next commit*.
- **Analogy:** Imagine you're at a grocery store, filling your shopping cart. You decide which items (changes) you want to buy (commit) right now. The shopping cart is your Staging Area.
- **What happens here:** You selectively add specific changes (files, or even parts of files) from your Working Directory to the Staging Area. This allows you to group related changes into a single commit.
- **Git's view:** Git now sees these as "staged" changes.

2.3 The Local Repository (or Local Repo)

- **What it is:** This is the hidden `.git` directory inside your project folder on your local machine. It's where Git stores your project's entire history – all your commits, branches, and metadata.
- **Analogy:** After you've paid for your groceries, they're now permanently "yours" and organized in your pantry. Your local repository is your private, organized collection of all the "saves" you've made.
- **What happens here:** When you perform a `git commit`, all the changes from the Staging Area are permanently recorded as a new snapshot in your Local Repository. Each commit has a unique ID.
- **Git's view:** These are your "committed" changes, part of your project's history.

2.4 The Remote Repository (or Remote Repo)

- **What it is:** This is a version of your repository hosted on a server, typically on platforms like GitHub, GitLab, Bitbucket, or a company's own Git server.
- **Analogy:** After packing your groceries at home, you send them to a shared community pantry where others can also contribute and take items.
- **What happens here:**
    - You **push** (git push) your local commits from your Local Repository to the Remote Repository to share them with collaborators and back them up.
    - You **pull** (git pull) changes from the Remote Repository to bring other collaborators' work into your Local Repository.
- **Git's view:** This is the shared, "central" version of the project that everyone works from and contributes to.

2.5 The Flow: From Idea to Shared Code

Let's visualize the typical flow:

1. **Start/Modify:** You make changes to files in your **Working Directory**.
    - *(You are editing code)*
2. **Stage:** You decide which specific changes you want to save. You move them to the **Staging Area** using git add.
    - git add [filename] or git add .
3. **Commit:** You save the staged changes as a permanent snapshot in your **Local Repository** using git commit. This is a "save point" in your history.
    - git commit -m "Your descriptive message"
4. **Push:** You send your newly committed changes from your **Local Repository** to the **Remote Repository** to share them and back them up.
    - git push origin main

5. **Pull:** You receive updates from the **Remote Repository** (changes made by others) and integrate them into your **Local Repository** (and sometimes your Working Directory).
    - git pull origin main

---

3. Essential Git Commands: Your Daily Toolkit

These are the commands you will use almost every single day when working with Git.

3.1 Getting Started with a Repository

- **To start a brand new project and put it under Git control:**
    - **git init**
        - **Purpose:** Initializes (creates) a new, empty Git repository in the current directory. This command creates the hidden .git folder, which is your Local Repository.
        - **When to use:** When you're starting a project from scratch and want to begin tracking its changes.
        - **Example:**

          Bash

          ```
          # Create a new empty folder for your project
          mkdir my_awesome_project
          # Navigate into the new folder
          cd my_awesome_project
          # Initialize Git in this folder
          git init
          # Output: Initialized empty Git repository in
          /path/to/my_awesome_project/.git/
          ```
        - **Important:** You only run git init ONCE per project, in the project's root directory.
- **To get a copy of an existing project from a remote server (like GitHub):**
    - **git clone [repository-URL]**
        - **Purpose:** Downloads an entire existing Git repository (including its full history) from a specified URL to your local

machine. It also automatically sets up the remote connection.

- **When to use:** When you're joining an existing project, or when you want to get a local copy of a project hosted online.
- **Example:**

  Bash

  ```
  # Clones the project into a new folder named 'my-repo'
  git clone https://github.com/username/my-repo.git
  # Output: Cloning into 'my-repo'...
  #         remote: Enumerating objects: ..., done.
  #         ...
  # Now you can navigate into the cloned folder
  cd my-repo
  ```
- **Important:** git clone creates a new folder for the repository, so don't run it inside an already initialized Git repo.

3.2 Checking the Status of Your Files

- **git status**
  - **Purpose:** Shows you the current state of your Working Directory and Staging Area. It tells you:
    - Which files are **untracked** (new files Git doesn't know about yet).
    - Which files are **modified** but not yet staged.
    - Which files are **staged** and ready for the next commit.
    - Which branch you are currently on.
  - **When to use: Constantly!** It's your compass in the Git workflow. Run it before git add, before git commit, after making changes.
  - **Example Scenarios:**
    - **After creating a new file:**

      Bash

      ```
      # Create a new file
      ```

```
touch new_feature.txt
git status
# Output:
# Untracked files:
#   new_feature.txt
# nothing added to commit but untracked files present (use "git add" to track)
```

- **After modifying an existing file:**

  Bash

  ```
  # Modify README.md
  echo "Hello again" >> README.md
  git status
  # Output:
  # Changes not staged for commit:
  #   modified: README.md
  # no changes added to commit (use "git add" and/or "git commit -a")
  ```

- **After staging a file:**

  Bash

  ```
  git add new_feature.txt
  git status
  # Output:
  # Changes to be committed:
  #   new file: new_feature.txt
  # Changes not staged for commit:
  #   modified: README.md # README.md is still modified but not staged
  ```

3.3 Saving Changes Locally (Stage and Commit)

- **git add [file(s)]**
  - **Purpose:** Moves changes from your Working Directory to the Staging Area. It tells Git, "I want to include these specific changes in my next save point (commit)."
  - **When to use:** After you've made modifications to files or created new files that you want to be part of your next commit.
  - **Examples:**
    - **Add a specific file:** git add my_script.py
    - **Add multiple specific files:** git add index.html style.css

        - **Add all modified and new files in the current directory and its subdirectories:** git add .
        - **Add all modified and deleted files that are already tracked (but not untracked new files):** git add -u (Useful if you've only updated/deleted existing files)
        - **Add interactively (patch mode):** git add -p (Allows you to stage specific hunks/sections of a modified file. Advanced but powerful!)
    - **Important:** git add does *not* save the changes permanently. It only prepares them for the next commit. Always follow git add with git status to verify.

- **git commit -m "Your descriptive message"**
    - **Purpose:** Permanently records the staged changes (everything currently in the Staging Area) into your Local Repository as a new snapshot. This is your "save point."
    - **When to use:** After you've staged all the changes that logically belong together.
    - **Commit Message Best Practices (Revisited):**
        - **Short, Summary Line (Subject):** Max 50-72 characters. Describes *what* changed.
        - **Blank Line:** Follow the subject with a blank line.
        - **Detailed Body (Optional):** Explain *why* the change was made, what problem it solves, any relevant context.
        - **Imperative Mood:** Start the subject line with a strong verb in the imperative mood (e.g., "Fix bug," "Add feature," "Update documentation," not "Fixed bug," "Added feature").
    - **Examples:**

      Bash

      ```
      git commit -m "feat: Implement user login functionality"
      # OR for a more detailed commit (this will open your default text editor):
      git commit
      # (Then type your message in the editor, save, and close)
      ```

- **Important:** Each commit has a unique identifier (a SHA-1 hash), and it points to its parent commit(s), forming a chronological history.

3.4 Synchronizing with Remote Repositories

- **Connecting your local repo to a remote (usually done once per clone/init):**
  - **git remote add [name] [URL]**
    - **Purpose:** Adds a new remote repository connection to your local Git configuration. The common name origin is the default alias for the primary remote.
    - **When to use:** After you git init a project locally and then create an empty repository on GitHub/GitLab.
    - **Example:**

      Bash

      ```
      # Usually after you've initialized and made your first commit locally
      git remote add origin https://github.com/yourusername/your-repo.git
      # You can verify remotes with:
      git remote -v
      # Output:
      # origin  https://github.com/yourusername/your-repo.git (fetch)
      # origin  https://github.com/yourusername/your-repo.git (push)
      ```
- **Sending your local commits to the remote:**
  - **git push [remote-name] [branch-name]**
    - **Purpose:** Uploads your new local commits from your Local Repository to the specified branch on the Remote Repository.
    - **When to use:** After you've made one or more commits locally and want to share them with collaborators or back them up.
    - **First time pushing the main branch:** You often need to set the upstream remote tracking branch.

      Bash

```
git push -u origin main
# The -u or --set-upstream flag tells Git to remember that your local 'main' branch
# should push to and pull from 'origin/main' by default.
# Subsequent pushes on this branch can just be: git push
```

- **Subsequent pushes:**

  Bash

  ```
  git push
  ```

- **Example:** git push origin feature-x (if you are on feature-x branch)

- **Important:** Before pushing, it's a good practice to git pull to integrate any changes from the remote that others might have pushed.

- **Getting updates from the remote and integrating them:**
  - **git pull [remote-name] [branch-name]**
    - **Purpose:** Fetches (downloads) changes from the specified remote repository and then automatically merges them into your current local branch. It's a shortcut for git fetch followed by git merge.
    - **When to use:** Before you start working for the day, or before pushing your own changes, to ensure your local repository is up-to-date with others' work.
    - **Example:**

      Bash

      ```
      git pull origin main
      ```

    - **If you've set an upstream branch (e.g., with git push -u):**

      Bash

      ```
      git pull
      ```

  - **Important:** git pull can sometimes lead to merge conflicts if local changes overlap with remote changes. We'll cover resolving these soon.

3.5 Viewing Your Project's History

- **git log**
    - **Purpose:** Displays the commit history of your current branch. Each entry shows the commit hash, author, date, and commit message.
    - **When to use:** To review past changes, find specific commits, or understand the project's evolution.
    - **Useful variations:**
        - **git log --oneline:** Shows a condensed log with one commit per line. Very useful for a quick overview.
        - **git log --graph --oneline --all:** Shows a visual ASCII graph of the commit history, including all branches. Excellent for understanding branching and merging.
        - **git log -p [filename]:** Shows the full diff (changes) introduced by each commit for a specific file.
        - **git log --author="Your Name":** Filters commits by author.

---

4. Branching and Merging: Your Collaborative Superpowers

Branches are Git's most powerful feature for enabling parallel development and managing different versions of your codebase.

4.1 What is a Branch?

- **Analogy:** Imagine your project's history as a straight timeline. A branch is like creating a new, parallel timeline that diverges from the main one. You can make changes on this new timeline without affecting the original.
- **Purpose:**
    - **Isolate Work:** Work on new features, bug fixes, or experiments in isolation without breaking the main, stable version of your code.
    - **Parallel Development:** Multiple developers can work on different features simultaneously on their own branches.

    - **Organize Development:** Keep experimental code separate from production-ready code.

- By default, when you initialize or clone a repository, you are typically on the main (or master) branch. This is considered the primary, stable line of development.

4.2 Creating and Switching Branches

- **git branch**
    - **Purpose:** Lists all local branches in your repository. The active branch is usually highlighted (e.g., with an asterisk *).
    - **Example:**

      Bash

      ```
      git branch
      # Output:
      # * main
      #   feature/login
      ```
- **git branch [new-branch-name]**
    - **Purpose:** Creates a new branch named [new-branch-name] but *does not* switch your Working Directory to it. You remain on your current branch.
    - **Example:**

      Bash

      ```
      git branch feature/user-profile
      git status # Still on 'main' branch
      ```
- **git checkout [branch-name]**
    - **Purpose:** Switches your Working Directory to the specified [branch-name]. When you switch, Git updates your files to match the state of that branch.
    - **Example:**

      Bash

      ```
      git checkout feature/user-profile
      # Output: Switched to branch 'feature/user-profile'
      ```

- **git checkout -b [new-branch-name] (Common Shortcut!)**
    - **Purpose:** This is a very common and convenient shortcut. It both **creates** a new branch named [new-branch-name] AND **switches** your Working Directory to it in one command.
    - **When to use:** Almost always, when you start working on a new feature or bug fix.
    - **Example:**

      Bash

      ```
      git checkout -b bugfix/login-issue
      # Output: Switched to a new branch 'bugfix/login-issue'
      ```
- **git branch -vv**
    - **Purpose:** Shows all your local branches and their associated remote tracking branches, which helps you see which local branch is connected to which remote branch.

4.3 The Merge Process: Combining Branch Histories

Once you've completed work on a feature branch, you'll want to integrate those changes back into the main branch. This is done through **merging**.

- **Steps to Merge:**
    1. **Switch to the target branch:** First, ensure you are on the branch where you want to *bring in* the changes (e.g., main).

       Bash

       ```
       git checkout main
       ```
    2. **Ensure target branch is up-to-date:** It's a good practice to pull the latest changes from the remote main branch to avoid conflicts:

       Bash

       ```
       git pull origin main
       ```
    3. **Merge the feature branch:** Now, merge your feature branch into the current branch (main).

       Bash

```
git merge [feature-branch-name]
# Example: git merge feature/user-profile
```

- **Types of Merges:** Git handles merging in two primary ways:
    - **Fast-Forward Merge:**
        - **When it happens:** This occurs when the target branch (e.g., main) *has not had any new commits* since your feature branch diverged from it. Essentially, the main branch's history is a direct ancestor of your feature branch.
        - **How it works:** Git simply moves the pointer of the target branch (main) forward to the latest commit of the feature branch. No new "merge commit" is created. The history remains linear.
        - **Visual:** main -> commit A -> commit B (feature) -> commit C (feature) *After fast-forward merge:* main also points to commit C.
    - **Three-Way Merge (Recursive Merge):**
        - **When it happens:** This is the most common type of merge. It occurs when *both* the target branch (e.g., main) and your feature branch have new, independent commits since they last shared a common commit. Their histories have diverged.
        - **How it works:** Git finds a "common ancestor" commit, then takes the changes from both branches from that common ancestor, and combines them. This process creates a **new commit** (called a "merge commit") that has two parent commits (the latest commit from the main branch and the latest commit from the feature branch).
        - **Visual:**
        - A -- B (main)
        - \ /
        - C -- D (feature)
        - \ /
        - E (Merge Commit)
        - **Important:** This type of merge is where **merge conflicts** can occur.

4.4 Handling Merge Conflicts (The Developer's Rite of Passage)

Merge conflicts happen when Git cannot automatically reconcile changes between two branches. This usually occurs when the *same lines* of code in the *same file* have been modified differently in both branches that are being merged.

- **How Git Indicates Conflicts:**
    - When a conflict occurs during `git merge` (or `git pull`), Git will pause the merge and tell you there are conflicts.
    - The conflicted files will be marked in your Working Directory with special "conflict markers":
    - `<<<<<<< HEAD`
    - This is the content from the current branch (e.g., main).
    - `=======`
    - This is the content from the branch you are merging (e.g., feature/login).
    - `>>>>>>> feature/login`
    - `HEAD` refers to your current branch (the one you `git checkout` to before merging).
    - The lines between `<<<<<<< HEAD` and `=======` are from your current branch.
    - The lines between `=======` and `>>>>>>> [branch-name]` are from the branch you're trying to merge.
- **Steps to Resolve a Merge Conflict:**
    1. **Identify Conflicts:** Git will tell you which files have conflicts. Run `git status` to see them listed under "Unmerged paths."
    2. **Open the Conflicted File(s):** Use your code editor to open each file marked with a conflict.
    3. **Manually Edit the File:**
        - Carefully review the changes from both sides.
        - Decide which version you want to keep, or manually combine both versions.
        - **DELETE** the `<<<<<<<`, `=======`, and `>>>>>>>` conflict markers. Your goal is to make the file's content exactly as you want it.

4. **Add the Resolved File:** Once you've fixed the file and removed all markers, stage the file to tell Git it's resolved:

   Bash

   git add [resolved-file-name.ext]
5. **Commit the Merge:** After all conflicts are resolved and staged, commit the merge:

   Bash

   git commit -m "Merge branch 'feature/login' into main and resolve conflicts"
   # Git often pre-populates the commit message for you. You can just save and close.

- **Important:** You must resolve *all* conflicts and stage *all* conflicted files before you can commit the merge.

4.5 Deleting Branches

Once a feature branch is merged and no longer needed, it's good practice to delete it to keep your repository clean.

- **git branch -d [branch-name]**
  - **Purpose:** Deletes the specified local branch *if it has already been fully merged* into its upstream branch. Git will prevent deletion if there are unmerged changes to prevent accidental data loss.
  - **Example:** git branch -d feature/my-new-feature
- **git branch -D [branch-name]**
  - **Purpose: Force-deletes** the specified local branch, regardless of whether it's been merged or not.
  - **When to use:** Use with caution! Only if you are absolutely sure you don't need the changes on that branch. This is useful for cleaning up abandoned or incorrectly created branches.
  - **Example:** git branch -D old-experiment-branch
- **Deleting a remote branch:**
  - git push origin --delete [branch-name] (or git push origin :[branch-name])
  - **Example:** git push origin --delete feature/my-new-feature

---

5. Git Best Practices: Working Smart, Not Hard

Using Git effectively goes beyond just knowing commands. These best practices will make your life easier and improve team collaboration.

5.1 Atomic Commits (Small, Focused Changes)

- **What it means:** Each commit should represent a single, logical, and complete unit of work. Don't commit unrelated changes together.
- **Why it's good:**
    - **Easier Review:** Colleagues can understand your changes more quickly.
    - **Easier Debugging:** If a bug is introduced, it's easier to pinpoint the exact commit that caused it.
    - **Easier Reverts:** You can undo specific changes without affecting other work.
- **How to achieve:** Use `git add` to selectively stage only the relevant files/changes for a particular commit.

5.2 Excellent Commit Messages

- **Structure:**
    1. **Subject Line (50-72 chars max):** Concise summary, imperative mood (e.g., "Add user authentication," "Fix bug in login form").
    2. **Blank Line:** Always.
    3. **Body (Optional):** Detailed explanation of *why* the change was made, *what problem it solves*, and any relevant context, trade-offs, or further details.
- **Why it's good:**
    - **History is Readable:** Makes `git log` genuinely useful for understanding project evolution.
    - **Context for Reviewers:** Helps others understand your thought process.
    - **Future You Thanks You:** You'll forget why you made a change in 3 months; a good message reminds you.
- **Good Example:**

- feat: Add basic user authentication
-
- Implements a basic user registration and login system.
- Uses bcrypt for password hashing and JWT for session management.
- Connects to the 'users' table in the database.
- Fixes #123 (issue tracking reference).
- **Bad Example:**

- stuff
- Fixed bug and added new feature and updated something.

5.3 Branching Strategies (Brief Overview)

While many exist, understanding the **Feature Branch Workflow** is most critical for individual projects and small teams.

- **Feature Branch Workflow (Most Common for Beginners):**
    - The `main` branch always remains stable and deployable.
    - For every new feature, bug fix, or experiment, create a new branch from `main`.
    - Work on your branch in isolation.
    - Once complete and tested, merge your branch back into `main`.
    - Delete the feature branch.
    - **Pros:** Clear separation of work, safe for `main`.
    - **Cons:** Can lead to "long-lived" branches that diverge significantly, making merging difficult (merge hell).
- **Trunk-Based Development:** All developers commit small, frequent changes directly to the `main` branch. Relies heavily on robust automated testing and continuous integration.
- **Git Flow:** A more complex, highly structured workflow with dedicated branches for `develop`, `feature`, `release`, and `hotfix`. Good for very large, strict release cycles.

5.4 Pull Before You Push

- **Rule of thumb:** Always run `git pull origin [your-branch]` before you `git push` your changes to the remote.
- **Why it's critical:** This ensures you incorporate any changes pushed by collaborators since your last `pull`. It helps you resolve potential merge

conflicts on your local machine, which is generally easier than trying to resolve them on the remote (especially if others are pushing simultaneously).

5.5 git status is Your Friend!

- Seriously, run `git status` frequently. It's your primary way to understand what's going on in your repository, what files are changed, and what actions Git expects from you. It gives you invaluable context for your next command.

---

6. Hands-On Lab: Your First Git Journey (Detailed Steps)

Let's put theory into practice! For this lab, you'll need a terminal (like Git Bash on Windows, Terminal on macOS/Linux, or your IDE's integrated terminal) and a free account on a Git hosting service (GitHub, GitLab, or Bitbucket).

Estimated Time: 60 minutes

6.1 Setup: Ensure Git is Ready

1. **Verify Git Installation:**
    - Open your terminal.
    - Type `git --version` and press Enter. You should see the Git version number (e.g., `git version 2.39.2`). If not, you need to install Git.
2. **Configure Git (First-time setup):**
    - Set your name (this will appear in your commits):

      Bash

      ```
      git config --global user.name "Your Name"
      ```
    - Set your email (use the same email associated with your GitHub/GitLab account):

      Bash

      ```
      git config --global user.email "your_email@example.com"
      ```
    - You can check your configurations with `git config --list`.
3. **Create an Account on a Git Hosting Service:**
    - If you don't have one, sign up for a free account on:

        - GitHub (Most popular, highly recommended)
        - GitLab
        - Bitbucket
    - This is where your Remote Repository will live.

6.2 Task 1: Initialize and Push a New Project

In this task, you'll start a brand new project locally, make your first commit, and push it to a remote repository (e.g., GitHub).

1. **Create a New Local Folder for your Project:**
    - Open your terminal.
    - Navigate to a location where you store your projects (e.g., your Desktop, Documents, or a `dev` folder).
    - Create a new directory (folder) for your project:

      Bash

      mkdir my-first-git-project
    - Navigate into your new project directory:

      Bash

      cd my-first-git-project
2. **Initialize a New Git Repository:**
    - Inside the `my-first-git-project` folder, type:

      Bash

      git init
    - **Expected Output:** Initialized empty Git repository in /path/to/my-first-git-project/.git/
    - This creates the hidden `.git` folder, making your project a Git repository.
3. **Create Your First File:**

- Let's create a simple README.md file (a common file that describes a project).
- Type:

  Bash

  ```
  echo "# My First Git Project" > README.md
  ```
- You can check the file content: cat README.md (or type README.md on Windows Command Prompt/PowerShell).

4. **Check Git Status (First Time):**
   - Type:

     Bash

     ```
     git status
     ```
   - **Expected Output:**
   - On branch master (or main) # Git typically defaults to 'master' or 'main'
   - No commits yet
   - Untracked files:
   - (use "git add <file>..." to include in what will be committed)
   - README.md
   -
   - nothing added to commit but untracked files present (use "git add" to track)
   - **Explanation:** Git sees README.md but doesn't track it yet.
5. **Stage Your First File:**
   - Tell Git to prepare README.md for the next commit:

     Bash

     ```
     git add README.md
     ```
   - Check status again: git status
   - **Expected Output:**
   - On branch master (or main)
   - No commits yet
   - Changes to be committed:
   - (use "git rm --cached <file>..." to unstage)
   - new file: README.md
   - **Explanation:** README.md is now in the Staging Area.

6. **Make Your First Commit:**
    - Permanently save the staged changes to your Local Repository:

      Bash

      git commit -m "feat: Initial project setup and README"
    - **Expected Output:**
    - [master (root-commit) 67c1c1a] feat: Initial project setup and README
    - 1 file changed, 1 insertion(+)
    - create mode 100644 README.md
    - **Explanation:** You've created your first snapshot! The `67c1c1a` is a short version of your commit's unique ID.
7. **Create a New Empty Repository on GitHub/GitLab:**
    - Go to your Git hosting service (GitHub.com, GitLab.com, or Bitbucket.org).
    - Click "New repository" or "Create project."
    - Give it the same name as your local folder: `my-first-git-project`.
    - Choose "Public" or "Private" (Public is fine for this demo).
    - **Important: DO NOT** initialize with a README, .gitignore, or license. We want an *empty* repository.
    - Click "Create repository."
    - After creation, you'll see instructions. Copy the **HTTPS URL** (e.g., `https://github.com/yourusername/my-first-git-project.git`).
8. **Connect Your Local Repository to the Remote:**
    - In your terminal, inside `my-first-git-project`, add the remote connection:

      Bash

      git remote add origin https://github.com/yourusername/my-first-git-project.git
      # Replace the URL with the one you copied from GitHub/GitLab!
    - Verify the connection:

      Bash

git remote -v

- **Expected Output:**
- origin https://github.com/yourusername/my-first-git-project.git (fetch)
- origin https://github.com/yourusername/my-first-git-project.git (push)

9. **Push Your First Commit to the Remote:**
    - Send your local main branch to the origin remote:

      Bash

      git push -u origin main
    - **Expected Output:** (You might be prompted for your username and password/personal access token)
    - Enumerating objects: ..., done.
    - ...
    - To https://github.com/yourusername/my-first-git-project.git
    - * [new branch]   main -> main
    - Branch 'main' set up to track remote branch 'main' from 'origin'.
    - **Explanation:** Your local main branch is now linked to the main branch on your remote origin.
    - **Verify:** Go back to your GitHub/GitLab repository page in your browser and refresh. You should see README.md!

2

6.3 Task 2: Feature Branch Workflow

Now, let's practice working on a new feature in isolation using branches and then merging it back.

1. **Create a New Feature Branch:**
    - Ensure you are on the main branch (git status).
    - Create a new branch and switch to it in one go:

      Bash

      git checkout -b feature/add-greeting
    - **Expected Output:** Switched to a new branch 'feature/add-greeting'
    - **Verify:** git branch (you should see * feature/add-greeting)
2. **Make Changes on the Feature Branch:**

- Edit your README.md file. Add a new line at the end (e.g., using a text editor or echo):

  Bash

  ```
  echo "This project will greet you!" >> README.md
  ```
- Create a new file for your greeting script:

  Bash

  ```
  echo 'print("Hello, Git!")' > greet.py
  ```
- Check status: git status (You should see README.md modified, greet.py untracked).

3. **Stage and Commit Changes on the Feature Branch:**
   - Stage all changes in the current directory:

     Bash

     ```
     git add .
     ```
   - Commit these changes:

     Bash

     ```
     git commit -m "feat: Add simple greeting feature with Python script"
     ```
4. **Switch Back to the main Branch:**
   - **Important:** Before merging, you must be on the branch you want to *merge into*.
   - Bash
   - git checkout main
   - **Expected Output:** Switched to branch 'main'
   - **Verify:** Check your files (ls or dir). Notice greet.py is gone, and README.md is back to its previous state (without the "This project will greet you!" line). This is because main doesn't have those changes yet!
5. **Merge the Feature Branch into main:**
   - Now, bring the changes from feature/add-greeting into main:

     Bash

git merge feature/add-greeting

- **Expected Output:** (Likely a Fast-Forward merge since main hasn't changed)

- Updating ...
- Fast-forward
- README.md | 1 +
- greet.py | 1 +
- 2 files changed, 2 insertions(+)
- create mode 100644 greet.py
- **Verify:** Check your files (ls or dir). greet.py should reappear, and README.md should have the new line. git status should show "nothing to commit, working tree clean."

6. **Delete the Feature Branch (Optional but Recommended):**
    - Since feature/add-greeting is now fully integrated into main, you can delete it locally:

      Bash

      git branch -d feature/add-greeting
    - **Expected Output:** Deleted branch feature/add-greeting (...).
    - **Verify:** git branch (the branch should be gone).
7. **Push the Merged Changes to Remote main:**
    - Now that main has the new feature, push it to GitHub/GitLab:

      Bash

      git push origin main
    - **Verify:** Check your remote repository in the browser. You should see the new greet.py file and the updated README.md.

3

6.4 Task 3: Simulating and Resolving a Simple Merge Conflict

Merge conflicts are inevitable. Learning to resolve them is a critical skill.

1. **Prepare for the Conflict (on main):**
    - Ensure you are on the main branch: git checkout main.
    - Add a line to README.md that you will later conflict with:

Bash

```
echo "## Project Goals" >> README.md
echo "- Learn Git" >> README.md
```

- Stage and commit these changes:

  Bash

  ```
  git add README.md
  git commit -m "docs: Add project goals section"
  ```

- Push to remote (optional, but good practice): `git push origin main`

2. **Create a Conflict Branch:**
   - Create and switch to a new branch for the conflicting change:

     Bash

     ```
     git checkout -b conflict-feature
     ```

3. **Make Conflicting Changes on `conflict-feature`:**
   - Edit `README.md`. **Change the EXACT same line you added in step 1** (e.g., `- Learn Git` to `- Understand Version Control`) or add a new line immediately below it in a conflicting way.

     Bash

     ```
     # Open README.md in a text editor (e.g., nano README.md or code README.md)
     # Change: - Learn Git
     # To:     - Master Git Basics
     ```

   - Save the file.
   - Stage and commit these changes on `conflict-feature`:

     Bash

     ```
     git add README.md
     git commit -m "feat: Refine Git learning goal"
     ```

4. **Switch Back to `main` and Introduce Another Change:**
   - Switch back to the `main` branch:

     Bash

     ```
     git checkout main
     ```

- Make a *different* change to README.md (e.g., add a different line or modify a different section) or even a new file. **Do NOT touch the line you just changed in conflict-feature.**

  Bash

  ```
  echo "- Collaborate Effectively" >> README.md
  ```
- Stage and commit this change on main:

  Bash

  ```
  git add README.md
  git commit -m "docs: Add collaboration goal to main"
  ```
- **Explanation:** Now, both main and conflict-feature have diverged and made changes that Git can't automatically combine.

5. **Attempt to Merge and Observe the Conflict:**
   - Try to merge conflict-feature into main:

     Bash

     ```
     git merge conflict-feature
     ```
   - **Expected Output (CRITICAL!):** Git will stop and inform you of the conflict.
   - Auto-merging README.md
   - CONFLICT (content): Merge conflict in README.md
   - Automatic merge failed; fix conflicts and then commit the result.
   - Check status: git status
   - On branch main
   - You have unmerged paths.
   - (fix conflicts and run "git commit")
   - (use "git merge --abort" to abort this merge)
   -
   - Unmerged paths:
   - (use "git add <file>..." to mark resolution)
   - both modified: README.md
   -
   - no changes added to commit (use "git add" and/or "git commit -a")
6. **Resolve the Conflict Manually:**
   - Open README.md in your text editor. You will see the conflict markers:

Markdown

```
## Project Goals
<<<<<<< HEAD
- Learn Git
- Collaborate Effectively
=======
- Master Git Basics
>>>>>>> conflict-feature
```

- **Your Task:** Decide which version you want to keep, or combine them.
    - **Option A (Keep `main`'s version for that line):** Delete the lines from `=======` to `>>>>>>> conflict-feature`.
    - **Option B (Keep `conflict-feature`'s version):** Delete the lines from `<<<<<<< HEAD` to `=======`.
    - **Option C (Combine/New version):** Modify the line to `- Learn Git and Master Git Basics` and delete all conflict markers.
- **Let's go with Option C to see a combination:**

  Markdown

```
## Project Goals
- Learn Git and Master Git Basics
- Collaborate Effectively
```

- **Save the `README.md` file.**

7. **Stage and Commit the Resolved Merge:**
    - Tell Git that you've resolved the conflicts for `README.md`:

      Bash

```
git add README.md
```

    - Check status: `git status` (It should now show `README.md` as "all conflicts fixed but you are still merging").
    - Commit the merge. Git usually provides a default message, just save and close the editor.

      Bash

```
git commit
# (This will open your default text editor with a pre-populated message. Save and close.)
# OR, if you prefer, you can use:
# git commit -m "Merge branch 'conflict-feature' into main and resolve conflicting project goals"
```

- **Expected Output:**
- [main ad6791e] Merge branch 'conflict-feature'

8. **Clean Up and Push:**
   - Delete the now-merged `conflict-feature` branch locally:

     Bash

     ```
     git branch -d conflict-feature
     ```
   - Push your merged `main` branch to the remote:

     Bash

     ```
     git push origin main
     ```
   - **Verify:** Check your remote repository in the browser. The `README.md` should reflect your merged changes.

Task

7. Online Activity (Task-Based): Debugging & Restoring Changes

This activity will deepen your understanding of Git's powerful history management capabilities, allowing you to debug issues and restore previous versions of your code. This is a crucial skill for any developer!

Learning Objective: Students will be able to use `git log`, `git diff`, `git stash`, and `git reset` (or `git revert`) to identify erroneous commits, temporarily save work, and restore previous versions of code.

Estimated Time: 1-2 hours

7.1 Scenario: The Accidental Bug and Unfinished Work

Imagine you're developing a simple Python script. You've been making a series of commits, adding features and making improvements. However, a few commits ago, you accidentally introduced a critical bug (e.g., a syntax error or logic flaw that breaks a core function). To make matters worse, you've already made several more good commits *on top* of the buggy one, and you also have some unfinished, unsaved work in your current files.

You need to:

1. Figure out exactly when the bug was introduced.
2. Temporarily save your current, unsaved work.
3. Fix the bug by undoing the problematic commit (or its changes).
4. Restore your unsaved work.

7.2 Simulation Steps (Set up your local repository for the task):

1. **Start Clean:**
    - Go to your my-first-git-project from the hands-on lab, or create a *new* empty directory and git init it.
    - Make sure you are on the main branch: git checkout main.
    - Create a simple Python file named calculator.py:

      Python

      ```
      # calculator.py
      def add(a, b):
        return a + b

      def subtract(a, b):
        return a - b

      print(f"Adding 5 and 3: {add(5, 3)}")
      print(f"Subtracting 10 and 4: {subtract(10, 4)}")
      ```
    - Initial commit:

      Bash

      ```
      git add calculator.py
      git commit -m "feat: Add basic calculator functions (add, subtract)"
      ```
2. **Add multiply function (Good Commit 1):**
    - Modify calculator.py:

      Python

      ```
      # calculator.py
      def add(a, b):
        return a + b
      ```

```
def subtract(a, b):
  return a - b

# Add this new function
def multiply(a, b):
  return a * b

print(f"Adding 5 and 3: {add(5, 3)}")
print(f"Subtracting 10 and 4: {subtract(10, 4)}")
print(f"Multiplying 6 and 7: {multiply(6, 7)}")
```

- Commit:

  Bash

  ```
  git add calculator.py
  git commit -m "feat: Add multiply function"
  ```

3. **Introduce the Bug (Erroneous Commit):**
   - Modify calculator.py again. **Intentionally introduce a bug in the `subtract` function.**

     Python

     ```
     # calculator.py
     def add(a, b):
       return a + b

     def subtract(a, b):
       # THIS IS THE BUG! Should be 'a - b'
       return a + b # Intentional bug: changed to addition!

     def multiply(a, b):
       return a * b

     print(f"Adding 5 and 3: {add(5, 3)}")
     print(f"Subtracting 10 and 4: {subtract(10, 4)}")
     print(f"Multiplying 6 and 7: {multiply(6, 7)}")
     ```

   - Commit the bug:

     Bash

     ```
     git add calculator.py
     git commit -m "fix: Optimize subtract function logic (introducing bug)"
     # Note: A real commit message wouldn't say "introducing bug"!
     ```

- *Test the bug*: Run python calculator.py. Notice that "Subtracting 10 and 4" outputs "14" instead of "6".

4. **Add divide function (Good Commit 2 - *After* the bug):**
   - Modify calculator.py again:

     Python

     ```
     # calculator.py
     def add(a, b):
       return a + b

     def subtract(a, b):
       # THIS IS THE BUG! Should be 'a - b'
       return a + b # Intentional bug: changed to addition!

     def multiply(a, b):
       return a * b

     # Add this new function
     def divide(a, b):
       if b == 0:
         return "Cannot divide by zero!"
       return a / b

     print(f"Adding 5 and 3: {add(5, 3)}")
     print(f"Subtracting 10 and 4: {subtract(10, 4)}")
     print(f"Multiplying 6 and 7: {multiply(6, 7)}")
     print(f"Dividing 20 by 5: {divide(20, 5)}")
     ```
   - Commit:

     Bash

     ```
     git add calculator.py
     git commit -m "feat: Add divide function with zero check"
     ```
5. **Make Uncommitted Changes (Work in Progress):**
   - Modify calculator.py one more time, but **DO NOT COMMIT OR ADD IT.**

     Python

     ```
     # calculator.py (at the very end of the file)
     # Some new experimental print statement
     print("Experimental feature: Fibonacci sequence next")
     ```

- Check status: git status (You should see calculator.py as "modified").

7.3 Your Task: Debugging and Restoration

Follow these steps using Git commands:

1. **Identify the Erroneous Commit:**
    - Use git log --oneline to view your commit history.
    - From the output, visually identify the short commit hash of the commit titled "fix: Optimize subtract function logic (introducing bug)".
    - Copy this commit hash.
    - Use git diff [commit-hash-of-bug]~1 [commit-hash-of-bug] calculator.py to specifically see what changes were introduced by *that particular commit* in calculator.py. Replace [commit-hash-of-bug] with the actual hash you copied.
        - **Explain in your deliverables:** What does this git diff command show you? How did it help you confirm the bug?
2. **Temporarily Save Your Uncommitted Work:**
    - Use git stash to save your current uncommitted changes to a temporary storage.
        - **Explain in your deliverables:** Why is git stash useful in this scenario?
3. **Fix the Erroneous Commit (Choose ONE method and justify):**
    - Now, you need to undo the changes of the buggy commit. Choose *either* git reset --hard *or* git revert.
    - **Option A: git reset --hard [hash-of-commit-BEFORE-the-bug] (USE WITH EXTREME CAUTION! This rewrites history)**
        - Find the commit hash of the commit *immediately BEFORE* your buggy commit using git log --oneline.
        - Execute: git reset --hard [hash-of-commit-BEFORE-the-bug]
        - **Explain in your deliverables:** Why did you choose git reset --hard? What are the implications of using --hard? Why is it generally discouraged on shared branches? (Consider that

this method effectively removes the buggy commit and all subsequent commits from your *local* history). You will likely need to `git push -f origin main` (force push) to update the remote, which is generally dangerous.

- **Option B: `git revert [hash-of-buggy-commit]` (Recommended for shared history)**
    - Execute: `git revert [hash-of-buggy-commit]` (Use the exact hash of the *buggy* commit you identified in step 1).
    - Git will create a new commit that undoes the changes of the specified buggy commit. It will open your text editor for a default commit message. Save and close.
    - **Explain in your deliverables:** Why did you choose `git revert`? What does `git revert` do differently from `git reset`? Why is `git revert` generally safer for shared branches?
- **Verify the fix:** Run `python calculator.py` again. The `subtract` function should now output `6` (if you chose `git revert`, or if you manually re-added `divide` after `git reset --hard`).

4. **Restore Your Temporarily Saved Work:**
    - Bring your stashed changes back into your Working Directory:

      Bash

      ```
      git stash pop
      ```
    - **Explain in your deliverables:** What is the difference between `git stash pop` and `git stash apply`? Why did you choose `pop` (or `apply`)?
5. **Final Push to Remote:**
    - If you used `git revert`, you can now `git push origin main`.
    - If you used `git reset --hard`, you will likely need to `git push --force origin main` (or `git push -f`). Be very clear that force pushing rewrites history and is generally *not* done on shared main branches without explicit team agreement.

7.4 Deliverables for Online Activity:

Submit a document (e.g., a Word document, PDF, or Markdown file) that includes:

1. **A list of all Git commands you used** for this task, in order of execution.
2. **Screenshots of your terminal output** after each significant command (e.g., git log, git diff, git stash, git reset/git revert output, git stash pop, git push).
3. **Your final `calculator.py` file** after the fix.
4. **Written Explanations:**
    - Briefly explain how git log and git diff helped you identify the erroneous commit.
    - Explain why git stash was necessary in this scenario.
    - **Crucially, explain your choice between `git reset --hard` and `git revert`**. Detail the pros and cons of the method you chose in this specific scenario, and when you would use the *other* method.
    - Briefly explain the difference between git stash pop and git stash apply.

Congratulations! By completing this week's activities, you've gained foundational skills in Git that are indispensable for any software development role. Keep practicing, and don't be afraid to experiment in a test repository. Git has a safety net for almost everything!