VERSION CONTROL SYSTEM

A Version Control System (VCS) is a tool that helps track changes to files over time. It allows multiple people to collaborate on a project, keeps a history of modifications, and enables reverting to previous versions if needed.

Main thing is sharing and versioning

**Sharing**

Eg: 2 developers dev1 and dev 2 working in the same project like developing a calculator.

     Dev1 working for additional functionality and dev2 with subtraction

End of the day to build the application they have to combine both functionality

This one we can share via email but in real world we have 1000 of files and we cant send it through email.

**Versioning:**

Suppose starting code developed for addition of 2 number later changed to 2 then 3 ,  
But after some point of time they released no one using 4 digit calculation and got new requirement go back to addition of 2 number

Using this version control system we can revert as we have all versions of the code we modified.

VCS 2 types

1. Centralized VCS eg: SVN
2. Distributed VCS eg : git

A Centralized Version Control System (CVCS) is a system where a single central server stores all versions of files,. If the server fails, all history may be lost.

A Distributed Version Control System (DVCS) allows each user to have a full copy of the entire repository, A fork is essentially a personal copy of a repository

|  |  |  |
| --- | --- | --- |
| Feature | Centralized VCS (CVCS) | Distributed VCS (DVCS) |
| Internet | Requires internet to access history | Can work offline and sync later |
| Backup recovery | If the central server fails, data may be lost | Each user has a full backup, reducing data loss risk |

Github:

Githhub is a cloud based service that uses **Git** for version control.

It store your Git repositories and makes it easy to collaborate with other developers on projects.

A **Git repository** is a **directory** or **storage space** where your project files are stored along with their **version history**.

Git:

<https://lms.simplilearn.com/courses/2823/GIT/syllabus>

**Git** is a **version control system** (VCS) that helps developers manage and track changes to their code over time.

Installation of git in linux os

1. **sudo apt-get install git**
2. **Git –version**
3. **Create your project and navigate to that path**
4. **Initialize a new Git repository: git init**
5. **ls -la**

            If the repository is initialized, you should see a hidden .git folder in the output

         If you delete the .git folder from your project directory, you will lose all version control     history and tracking information

Inside .git below files are present:

**ls .git**

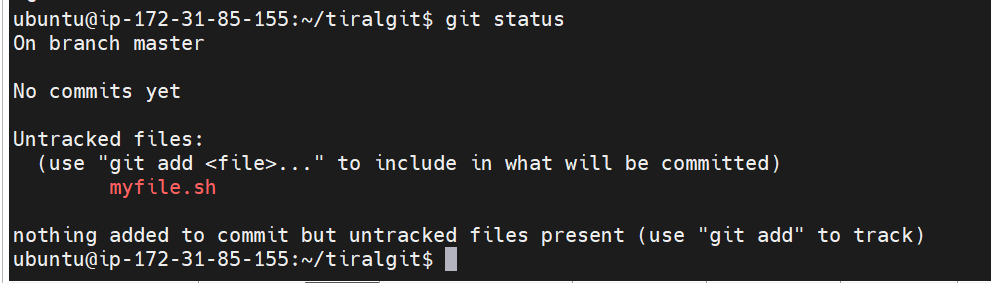
HEAD  branches  config  description  hooks  info  objects  refs

Object: Stores the actual file contents in a compressed format

    Config: Contains repository-specific configurations, such as user details, remote URLs, and branch settings**.**

1. **Git status**

ubuntu@ip-172-31-85-155:~/tiralgit$ git status



git status output means:

1. You are on the master branch.
   * But there are no commits yet (this means your repository is empty).
2. You have an untracked file: myfile.sh
   * Git is not tracking this file yet.
   * It won’t be part of any commit unless you add it to Git.

1. **Add files to staging: git add filename**

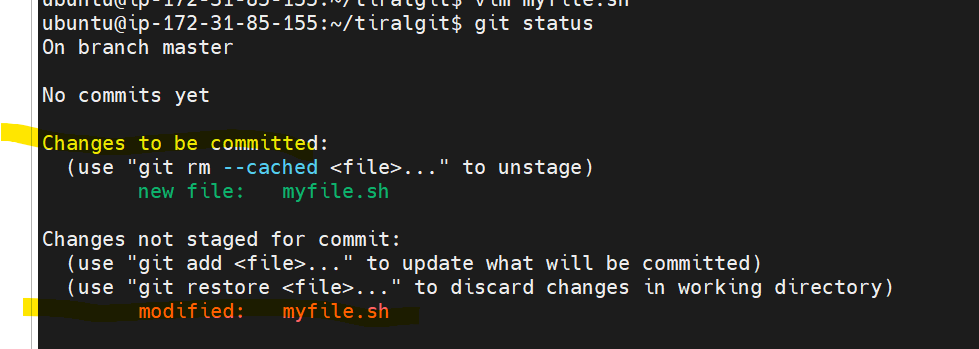
Staging It holds the changes you want to commit but haven’t committed yet.

1. **Commit:** git commit -m " this is my first version"

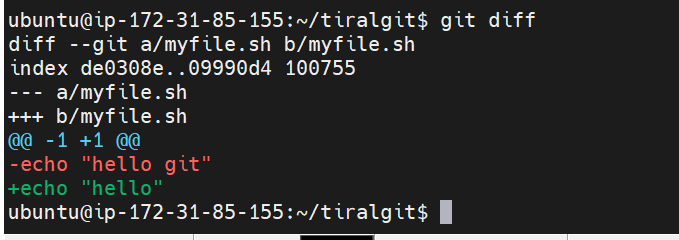
                m for message

           Commit records the changes permanently.so you can track, revert,

           Every commit creates a snapshot of your files.

1. **Suppose ur modifying the file from hello git to hello**
2. **Check git status: git status**
3. ****

1. **Check git diff : git diff**

****

1. **Add files for staging : git add filename**
2. **After staging if you want to check diff use the command**

**git diff --staged myfile.sh**

1. **Then commit: git commit -m “ this is my second version”**

1. **git log:  The git log command shows the history of commits in your repository, Helping you track who made changes and when.**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Pushing the code to the github**

1. **Create a New Repository on GitHub**

**Go to GitHub -> Create repository  -> Create new repository**

1. **Enter Repository Details:**

**Repository name: Enter a unique name (e.g., myproject).**

**Description (optional): Add a short description.**

**Visibility:**

**Public: Anyone on the internet can see this repository.**

**Private: Only you and selected collaborators can access it.**

1. **Click "Create repository" at the bottom.**
2. **Copy the repository URL (e.g.,** [**https://github.com/your-username/myproject.git**](https://github.com/your-username/myproject.git)**).**
3. **Connect Local Repository to GitHub**
4. **N**avigate to your local project folder
5. Use the copied GitHub repository URL to link it with your local repo

***git remote add origin https://github.com/ChinjuVarghesee/Kubernetes\_Practise.git***

***Remot: informing git that we are working with remote repository***

***Add: adding new repository***

***Origin: represent main remote repository***

1. Verify that the remote repository is added correctly

***git remote -v***

Exected o/p:

origin  https://github.com/your-username/myproject.git (fetch)

origin  https://github.com/your-username/myproject.git (push)

This confirm**s that your local repository is connected to the remote GitHub repository**

1. **Push the Code to GitHub**

***git push -u origin main***

***-u : set upstream, so next time no need to add -u***

* First-time authentication:
  + If prompted for a username, enter your GitHub username.
  + If prompted for a password, DO NOT enter your GitHub password.
  + Instead, you need to use a Personal Access Token (PAT).

Generating a GitHub Personal Access Token (PAT)

1. Go to GitHub Settings (Click on your profile picture → "Settings").
2. In the left sidebar, go to Developer settings → Personal access tokens.
3. Click Generate new token → Generate new token (classic).
4. Set an expiration date (e.g., 90 days).
5. Under scopes, check repo (for repository access).
6. Click Generate token and copy it. This token is shown only once
7. Verify the Code on GitHub

GIT branching Strategy

→ Git and GitHub do not enforce the same branching strategy, but GitHub follows the branching    strategies that you set up in your Git repository

→ main and master are the primary/default branch of a repository

→ When you create a new Git repository, Git automatically creates a branch named master

→ 2020, GitHub announced a change to use main as the new default branch

→ Both main and master serve the same function

→ to check which branch we are in   
   ubuntu@ip-172-31-85-155:~/new$ **git branch**

    \* master

→ to rename your **master** branch to **mini**,

**git branch -m master mini**

**Feature Branch:**

Feature Branching is a Git workflow where developers create separate branches for each new feature. This helps in keeping the master (or main) branch stable,

Becx this is very new feature we dont know when it will complete so we r creating feature branch

A **release branch** is a special type of branch used in Git to prepare for a new software release.

Github program for Release Branching , feature branching

1. Setting Up the Git Repository

**mkdir math\_operations**

**cd math\_operations**

**git init**

**ls -la : check whether repository create or not**

1. Create a file named math\_operations.py and add the basic arithmetic functions:

#!/usr/bin/env python3

# math\_operations.py

def add(a, b):

    return a + b

def subtract(a, b):

    return a - b

def multiply(a, b):

    return a \* b

a=float(input("enter first digit"))

b=float(input("enter second digit"))

# Get user input for the operation to perform

#select the opeartions to be performed

print("ADD=1")

print("SUB=2")

print("MUL=3")

action=input("enter the number corresponding to the operation")

if action == '1':

    result=add(a,b)

elif action == '2':

    result=subtract(a,b)

elif action == '3':

    result =multiply(a,b)

else:

    print("entered invalid number")

print("result",result)

1. Commit the basic math operations library to the repository.

**git add .**

**git commit -m "Initial commit - Add basic math operations library"**

1. Create **a New Repository on GitHub and copy the url**

**git remote add origin** [**https://github.com/ChinjuVarghesee/Basic-Math-Operations-Library.git**](https://github.com/ChinjuVarghesee/Basic-Math-Operations-Library.git)

**git remote -v**

**git push -u origin master**

1. Now again make some changes in the file which we created before math\_operations

For that Create a Feature Branch (e.g., add modulus function)

**git checkout -b feature/add-modulus**

**Git branch : it should be in feature**

           Now edit the file add modulus function.

         #!/usr/bin/env python3

# math\_operations.py

def add(a, b):

    return a + b

def subtract(a, b):

    return a - b

def multiply(a, b):

    return a \* b

def modulus(a, b):

    return a % b

a=float(input("enter first digit"))

b=float(input("enter second digit"))

# Get user input for the operation to perform

#select the opeartions to be performed

print("ADD=1")

print("SUB=2")

print("MUL=3")

print("Mod 4")

action=input("enter the number corresponding to the operation")

if action == '1':

    result=add(a,b)

elif action == '2':

    result=subtract(a,b)

elif action == '3':

    result =multiply(a,b)

elif action == '4':

      result=modulus(a,b)

else:

    print("entered invalid number")

print("result",result)

1. Commit the changes:

**Git add .**

**git commit -m "Add modulus function"**

1. Merge **the Feature Branch into Main:**

**git checkout master**

**git merge feature/add-modulus   # Merge feature branch into main**

1. **Pull the Latest Main** (to ensure you are up-to-date with the latest changes):

**git pull origin master**

1. Create **a Release Branch for Finalizing the Library:** After adding the necessary features, create a release branch for preparing the final version of the library:

**git checkout -b release/v1.0**

1. Finalize the code, add a README if necessary

**touch README.md**

**vim README.md**

# Basic Math Operations Library

A simple Python library for performing basic arithmetic operations.

## Features:

- Addition

- Subtraction

- Multiplication

- Division

- Modulus

## Example Usage:

```python

from math\_operations import add, subtract

print(add(5, 3))  # Output: 8

print(subtract(5, 3))  # Output: 2

1. Commit changes

**git add .**

**git commit -m "Prepare for release v1.0 - Finalize math operations library"**

1. Once the release branch is ready, merge it into the main branch:

**git checkout master**

**git merge release/v1.0**

1. Tag the Release:

**git tag -a v1.0 -m "Release version 1.0"**

 Git tags help in managing software versions, tracking stable releases, and deploying correct versions.

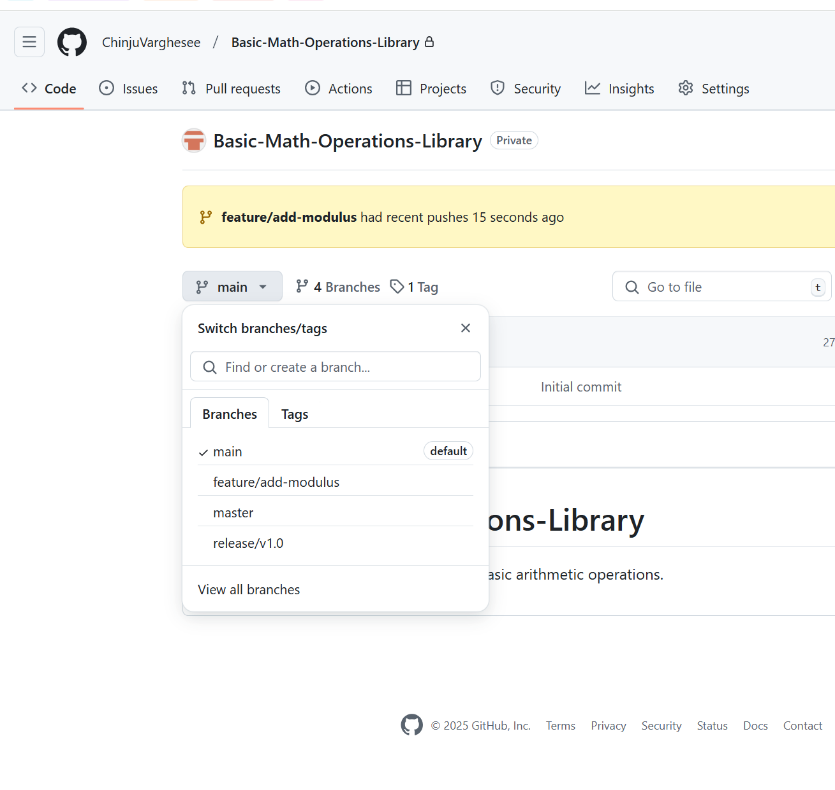
1. Push the Changes to GitHub

**git push origin master**

**git push origin release/v1.0**

**git push origin feature/add-modulus**

**git push --tags**

****

**Git Basics commands**

1. **Git Setup:**
   * git config --global user.name "Your Name"
   * git config --global user.email "your-email@example.com"
2. **Repository Initialization:**
   * git init – Initialize a new Git repository
   * git clone <repository-url> – Clone a repository from GitHub
3. **Basic Git Workflow:**
   * git add . – Stage all changes
   * git commit -m "message" – Commit changes with a message
   * git status – Check the status of the repository
   * git log – View the commit history

**Branching and Merging:**

1. **Create and Switch Branches:**
   * git branch <branch-name> – Create a new branch
   * git checkout <branch-name> – Switch to a different branch
   * git checkout -b <branch-name> – Create and switch to a new branch
2. **Merging:**
   * git merge <branch-name> – Merge changes from another branch into the current branch
   * git rebase <branch-name> – Rebase changes from another branch
3. **Branch Management:**
   * git branch -d <branch-name> – Delete a local branch if the branch is merged
   * git branch -D <branch-name> →Delete a local branch if the branch is un merged
   * git push origin --delete branch-name -> Delete a remote branch:
   * git push origin <branch-name> – Push a branch to the remote repository
   * git pull origin <branch-name> – Pull changes from the remote repository.

Diff between rabase and merge?  
**"Rebase** is the process of taking the changes from one branch and reapplying them on top of another branch, which results in a clean, linear history without merge commits.

* Main branch A-B-C
* From branch C a feature branch created like E-F-G
* Later main branch files changed to A-B-C-D
* So we need the latest contents from the main branch apply rebase

                      git checkout feature-branch

                      git rebase main

* So now feature branch changed to A-B-C-D-E-F-G
* Main branch remains unchanged

**Merge**, on the other hand, combines two branches, preserving their full history and adding a merge commit that shows where the branches merged.

* Main branch A-B-C
* From branch C a feature branch created like E-F-G
* Later main branch files changed to A-B-C-D
* So we need to merge the changes feature branch into the main branch

          git checkout main

                      git merge feature-branch

* So now Main branch changed to A-B-C-D-M
* Feature branch remains unchanged

The **main branch** is updated **only** after you merge the feature branch into it.

The rebase only modifies the feature branch's history, not main.

**For a small, personal project or if you prefer a cleaner, linear history**, **use rebase**. It helps in maintaining a tidy commit history and makes it easier to track the flow of changes.

**For a collaborative project or if you need to preserve all commit histories**, **use merge**. This way, you maintain context and avoid rewriting history, making it safer for team collaboration.

Merge Conflicts:

Merge conflicts occur when Git cannot automatically combine changes from two branches. This happens when the same line is modified in both branches, a file is deleted in one branch but edited in another, or when a file is renamed in one branch but modified in another. To avoid conflicts, always pull the latest changes before editing, communicate with teammates, and resolve conflicts manually when needed."

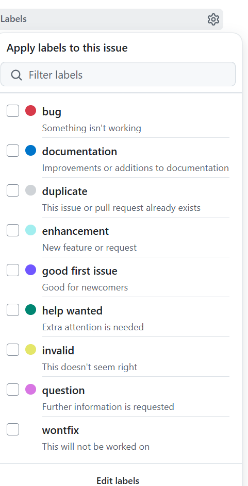
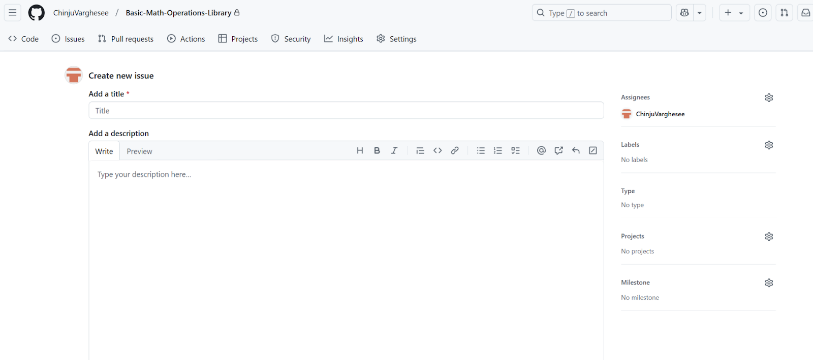
**GitHub for DevOps:**

1. **Managing Issues and Pull Requests:**
   * **Use GitHub Issues to track tasks, bugs, and features.**
   * **Leverage GitHub PR templates to standardize contributions.**
2. **Code Reviews and Merging PRs:**
   * **Review pull requests and use comments for collaboration.**
   * **Use merge strategies like squash and merge or rebase and merge.**
3. **GitHub Actions for CI/CD:**
   * **Learn to write YAML files for continuous integration workflows.**
   * **Set up automated deployment pipelines, run tests, and build Docker images using GitHub Actions.**

Managing issues in github

**Use GitHub Issues to track tasks, bugs, and features.**

**Go to issue either u can create an issue ,  assign people, add labels, and track progress.**

****

PR: PULL REQUEST

What is a Pull Request in GitHub?  
→ A PR is a request to merge changes from one branch to another. It is used for code review, collaboration, and safe merging in Git projects.

Why do we need Pull Requests instead of merging directly?  
→ PRs allow code review, prevent errors, ensure team collaboration, and maintain clean version control.

What happens after a Pull Request is approved?  
→ Once approved, it is merged into the main branch, and the feature branch is usually delete**d.**

**How to create and check pull request.**

One ur files are pushed got to github→ pull request→ create pull request→ give description

To check and approve

Pull request→Check for open pull request→ check the code → approve

Commands to Go Back to a Previous Version in Git

**HEAD**: Refers to the most recent commit in the current branch (the "tip" of the current branch).

**HEAD~1**: Refers to the **commit just before** the current HEAD commit (the parent of the most recent commit).

**HEAD~2**: Refers to the **commit before the parent** of the current HEAD (i.e., two commits ago).

1. Keep changes but undo commit: git reset --soft HEAD~1

 The last commit is undone, but the changes stay in the staging area.

1. Undo commit and unstaged: git reset --mixed HEAD~1
2. Undo commit and delete changes:: git reset --hard HEAD~1

Git reflog:

git reflog is an essential Git tool that enables you to recover lost work,

git reflog

0e3d4d3 HEAD@{0}: reset: moving to HEAD~1

1c7b7e9 HEAD@{1}: commit: Added a third line

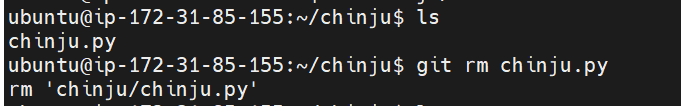
e4f09e5 HEAD@{2}: commit: Added a second line

c1d2e3f HEAD@{3}: commit: Initial commit

We performed a hard reset and third line removed to recover

Git reset –hard HEAD@{1}

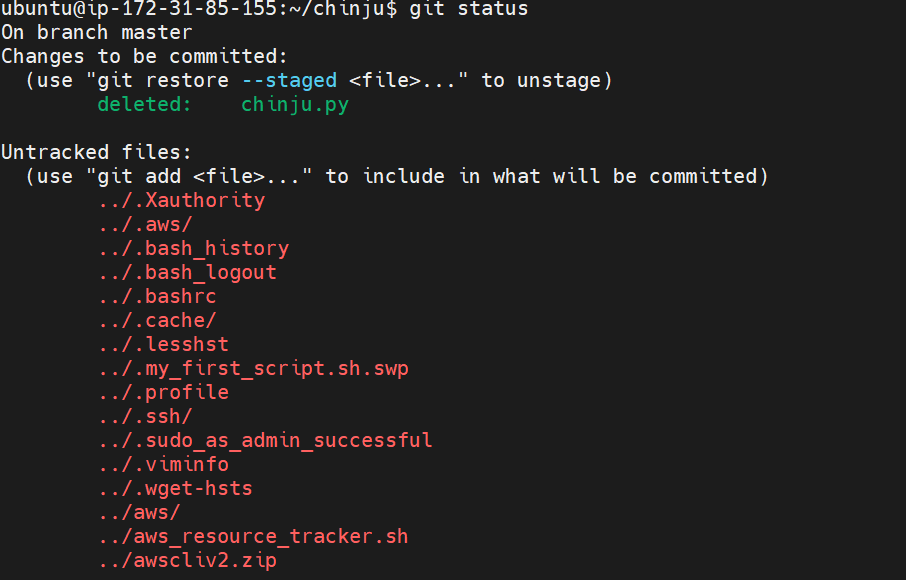
Git rm :Deletes the file from both Git and the working directory.



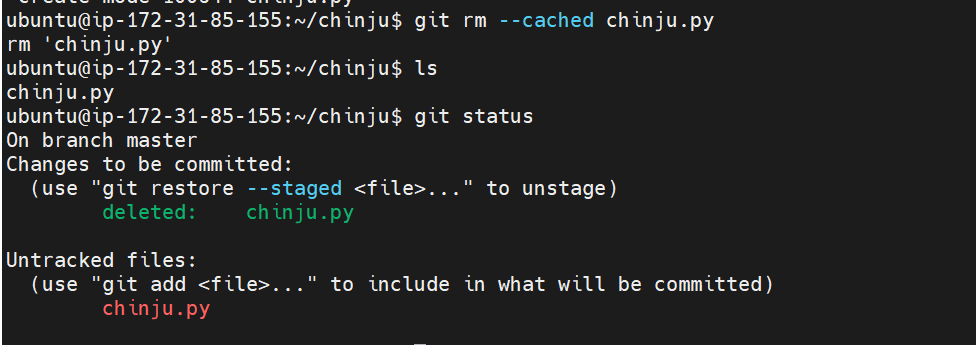
To check weather file deleted from working directory

https://lh7-rt.googleusercontent.com/docsz/AD_4nXdh4g0RvWk-AXNmdGSPoTAkD5H0jBMVatpDQOmZMxVODPOFpxRDBtxv1HIMWCBBTrDDmkvw8CqAX-xenio8rfZgHtbuhdi_a-uDlyg7TR9TQVLAOL5BejFMUlNyRWBkJXHxV_TQJQ?key=HecxA9SgWMbG4bzCe4AvyRBk

To check whether file deleted from git



Git rm –cached filename: Removes the file from Git tracking but keeps it in the working directory.



Ls -lrt

hot fix branch

A **hotfix branch** is a special type of **Git branch used for urgent bug fixes** in a production environment. It allows you to fix critical issues **without disrupting ongoing development**.

Since hotfixes address **production issues**, they are created from main (or master):

git checkout main

git pull origin main  # Ensure you have the latest version