
CDU
Cloud And Data Universe
VERSION
CONTROL
TRACK • MANAGE • COLLABORATE
EVERY CHANGE. EVERY TIME. ALWAYS UNDER CONTROL.
HISTORY
COLLABORATION
RELIABILITY
FLEXIBILITY

CDU
Cloud And Data Universe

WHAT IS VERSION CONTROL?

Before learning Git, we need to understand the problem that Git was created to solve.

git

1 IMAGINE YOU'RE WRITING A PYTHON PROGRAM

Suppose you're creating a simple Python application.

1. You start with:

```
print("Hello World")
```

2. You save it as: `hello.py`
3. Everything is fine.

THE PROBLEM WITHOUT VERSION CONTROL

After some time, you make more changes, add features, fix bugs...

But what if something breaks?

hello.py → hello_v2.py → hello_v3.py → hello_final.py → hello_final_v2.py → hello_final_v2.py → hello_final_latest.py → hello_final_latest_new.py

- Who changed the file?
- What exactly changed?
- When was the change made?
- Which version was working?
- Can I go back to an older version?

VERY HARD TO MANAGE!

VERSION CONTROL

is a system for **managing and tracking changes** to files over time.

THINK OF IT AS A TIMELINE

Version 1	Version 2	Version 3	Version 4	Version 5
Created application	Added login	Added registration	Fixed bug	Added validation

Instead of many copies, version control maintains the **history of changes.**

WHY VERSION CONTROL?

HISTORY
See how the project evolved over time.

TRACKING
Know what changed, who changed it, when and why.

COLLABORATION
Multiple people can work together on the same project.

RECOVERY
Go back to previous versions when something goes wrong.

EXPERIMENTATION
Try new ideas safely without affecting the main project.

ACCOUNTABILITY
Know who made a change and why.

KEY TAKEAWAY: Version control is the practice of **tracking and managing changes** to files over time. **Git** is a version control system that helps us do exactly that.

CDU
Cloud And Data Universe

NOW IMAGINE A TEAM

Suppose three developers are working on the same Python project.

git

NOW WE HAVE QUESTIONS:

- Who changed this code?
- What did they change?
- When did the change happen?
- Which version should we keep?
- What happens if two people change the same code?

MANAGING THIS MANUALLY BECOMES EXTREMELY DIFFICULT.

VERSION CONTROL

is a system for managing and tracking changes to files over time.

THINK OF IT AS A TIMELINE

VERSION 1	VERSION 2	VERSION 3	VERSION 4	VERSION 5
Created application	Added login	Added registration	Fixed bug	Added validation

Instead of many copies, version control maintains the history of changes.

WHY VERSION CONTROL?

HISTORY
See how the project evolved over time.

TRACKING
Know what changed, who changed it, when and why.

COLLABORATION
Multiple people can work together on the same project.

RECOVERY
Go back to previous versions when something goes wrong.

EXPERIMENTATION
Try new ideas safely without affecting the main project.

ACCOUNTABILITY
Know who made a change and why.

KEY TAKEAWAY: Version control is the practice of tracking and managing changes to files over time. | Git is a version control system that helps us do exactly that.

CDU
Cloud And Data Universe

THIS IS WHERE VERSION CONTROL COMES IN

git

Version control is a system for managing changes to files over time.

WITHOUT VERSION CONTROL

Instead of manually creating:

v1 → v2 → v3 → v4

- Too many copies
- No idea what changed
- Hard to go back
- Which is the latest?
- Risk of losing working code
- Very difficult to manage

Vs

WITH VERSION CONTROL

we maintain a **history of changes.**

Each version represents a **state of the project** at a particular point in time.

THINK OF VERSION CONTROL LIKE A TIMELINE

Imagine your project has evolved like this:

Instead of having five different copies of the entire project,
a version control system maintains the history of changes.

CDU
— Cloud And Data Universe —

THINK OF VERSION CONTROL LIKE A TIMELINE

git

Imagine your project has evolved like this:

Instead of having five different copies of the entire project, a **version control system** maintains the **history of changes**.

WHAT DOES VERSION CONTROL ACTUALLY TRACK?

It can track things like:

WHAT CHANGED?

`Added a new function`

Exactly what modifications were made to the code or files.

WHO CHANGED IT?

`Developer A`

Identify who made the change.

WHEN WAS IT CHANGED?

`August 8`

Know the exact date and time of the change.

WHY WAS IT CHANGED?

`Fix incorrect password validation`

Understand the reason behind the change.

★ This creates a **history** of the project.

CDU
— Cloud And Data Universe —

VERSION CONTROL IS NOT ONLY FOR PROGRAMMERS

git

This is important for a generic course.
Version control can be useful anywhere you have files that **change over time**.

FOR EXAMPLE:

SOFTWARE DEVELOPMENT

- Python
- Java
- JavaScript
- SQL

Any programming language, framework, or codebase **benefits from version control**.

BUT ALSO

- **Documentation**
  Notes, guides, policies, manuals
- **Configuration**
  Config files, settings, environments
- **Infrastructure code**
  Terraform, ARM templates, CloudFormation
- **Websites**
  HTML, CSS, content, themes
- **Data projects**
  Datasets, notebooks, pipelines
- **Scripts**
  Automation scripts, utilities, batch jobs

Any type of file or content that changes over time can be **versioned and tracked.**

The underlying problem is the same:

HOW DO WE MANAGE AND TRACK CHANGES OVER TIME?

Keep history | Enable collaboration | Prevent conflicts | Recover easily | Maintain integrity

CDU
Cloud And Data Universe

WHAT ARE THE BENEFITS?

git

Version control gives us several important capabilities.

1 HISTORY

- Jan 10 Initial commit
- Jan 15 Added login
- Jan 20 Added registration
- Feb 02 Fixed login bug
- Feb 08 Added validation

We can see how the project evolved.

2 TRACKING

We can identify changes.

3 COLLABORATION

Multiple people can work on the same project.

4 RECOVERY

- v5 Added validation
- v4 Fixed login bug
- v3 Added registration
- v2 Added login
- v1 Initial commit

We can inspect previous versions when something goes wrong.

5 EXPERIMENTATION

We can work on new ideas without immediately affecting the main version.

6 ACCOUNTABILITY

- Author Developer A
- Date Aug 08, 2025
- Change Fix incorrect password validation

We can identify who made a particular change.

VERSION CONTROL
MAKES OUR WORK ORGANIZED, SAFE & SCALABLE.

HISTORY | TRACKING | COLLABORATION | RECOVERY | EXPERIMENTATION | ACCOUNTABILITY

CDU
Cloud And Data Universe

GIT IS NOT THE ONLY VERSION CONTROL SYSTEM

git

There have been several version control systems over the years.

WHAT IS A VERSION CONTROL SYSTEM?

A system that helps us manage and track changes to files over time.

POPULAR VERSION CONTROL SYSTEMS

Version Control Systems

- **Git** — Most Popular
- **Subversion** (SVN)
- **Mercurial**
- **Others** (CVS, Perforce, Bazaar, etc.)

WHY DID GIT BECOME SO POPULAR?

- **Powerful Capabilities** — Supports modern development workflows
- **High Performance** — Very fast even for large projects
- **Flexible** — Works with any type of project
- **Distributed Architecture** — Every developer has a full local copy
- **Strong Community** — Backed by millions of developers
- **Industry Standard** — Used by almost every company worldwide

Version Control is the Concept

- The general idea and principles
- Defines how we manage changes
- Applicable in many tools
- Not tied to a specific software

≠

Git is a Tool

- A specific software that implements the version control concept
- Most popular implementation
- Open source and free
- Used for software development

This distinction is very important!

KEY TAKEAWAY: | "Version control is the concept. | "Git is a tool that implements version control."

CDU
Cloud And Data Universe

Every great project has a history.
Git helps you write it.

git

INTRODUCTION TO GIT

The first step to better code, collaboration and version control.

START YOUR JOURNEY WITH GIT TODAY!

WHAT YOU'LL LEARN

- Track Changes and Maintain History
- Work with Branches and Merge
- Sync with Remote Repositories
- Collaborate Effectively in Teams
- Manage Code with Confidence and Safety
- Build Professional Workflows and Best Practices

ONE TOOL. ENDLESS POSSIBILITIES. LET'S BUILD SOMETHING AMAZING TOGETHER!

Learn. Commit. Push. Repeat.

CDU
Cloud And Data Universe

WHAT IS GIT?

git

1 THE PROBLEM

Projects **change over time**, and managing those changes manually becomes **difficult**.

2 THE CONCEPT

Version control is the practice of **tracking** and **managing changes** to files over time.

3 THE NEXT QUESTION

So, how do we actually **implement** version control?

ONE ANSWER: git.

GIT IN ONE SENTENCE

> **Git is a distributed version control system that tracks changes in files and maintains their history.**

That's the formal definition.
But let's break it down because **every word matters.**

VERSION CONTROL

The practice of tracking and managing changes to files over time.

DISTRIBUTED

Each developer has a complete repository locally. Work can happen anywhere, even offline.

Version control is the concept.
It's the idea of tracking and managing changes over time.

Git is the tool that implements it.
It's a distributed system that helps us put this concept into practice.

```
$ git init
$ git add .
$ git commit -m "Initial commit"
$ git push origin main
```

★ VERSION CONTROL IS THE CONCEPT. | GIT IS A VERSION CONTROL SYSTEM THAT IMPLEMENTS THAT CONCEPT. ★

CDU
— Cloud And Data Universe —

FIRST, REMEMBER THE PROBLEM

git

1. THE PROBLEM

Imagine you have a **Python** project:

```
python-project/
├── app.py
├── calculator.py
└── README.md
```

You make changes **every day.**

Version 1 → Version 2 → Version 3 → Version 4

You want to know:

- What **changed**?
- **When** did it change?
- **Who** changed it?
- **Why** did it change?
- Can I see an **older version**?
- Can **multiple developers** work on it?

⚠ This is the **version control problem**.
Git provides a system for managing all of this.

3. GIT THINKS IN TERMS OF HISTORY

One of **Git's** most important ideas is:

> "Your project has a **history**."

Imagine your **Python** application evolving:

PROJECT HISTORY

Version 1	Created application
Version 2	Added login
Version 3	Added registration
Version 4	Fixed login bug
Version 5	Added validation

Git records this history.
Later, you can **inspect** it and **understand** how your project evolved.

Git's power comes from **tracking changes** over time and maintaining the **complete history** of your project.

CDU
— Cloud And Data Universe —

WHAT DOES GIT ACTUALLY TRACK?

git

1. SUPPOSE YOU HAVE:

calculator.py (Version 1)

```
1  def calculate_total(price, tax):
2      return price + tax
```

2. LATER YOU CHANGE IT TO:

calculator.py (Version 2)

```
1  def calculate_total(price, tax, discount):
2      return price + tax - discount
3
```

So **Git** isn't simply storing a bunch of files.
It's maintaining information about the evolution of those files.

3. GIT CAN HELP YOU DETERMINE:

What changed?
Added discount parameter

When?
August 8

Who?
Developer A

Why?
Support discounted orders

4. GIT RECORDS ALL THIS

Example Commit
7f3a9c2 — Aug 8, 2026 10:30 AM

What changed
Added discount parameter and updated calculation logic.

When
August 8, 2026 10:30 AM

Who
Developer A
(developer.a@example.com)

Why (Commit Message)
Support discounted orders

Project History

Git tracks content changes + context (when, who, why)

THAT'S THE POWER OF VERSION CONTROL!

CDU
Cloud And Data Universe

WHAT IS A GIT REPOSITORY?

git

To use Git for a project, we create something called a repository. | Usually we simply call it a: **repo**

1. SUPPOSE WE HAVE:

python-project/
- app.py
- calculator.py
- README.md

This is just a normal Python project.

2. WE TELL GIT:

"This is a Git project."

Git then creates its repository information.

3. CONCEPTUALLY:

python-project/
- .git/
- app.py
- calculator.py
- README.md

The .git directory contains Git's internal information about the repository.

This is what allows Git to maintain the project's history.

WHAT THE REPOSITORY ENABLES

- Track changes over time
- Store commits and history
- Manage versions
- Enable collaboration with others
- Keep your project safe and organized

★ A Git repository (repo) is the heart of every Git project.

REPO = HISTORY + CHANGES + CONTEXT

CDU
— Cloud And Data Universe —

THE REPOSITORY IS MORE THAN YOUR FILES

git

This distinction is important.

1. YOUR PROJECT CONTAINS YOUR ACTUAL FILES

- app.py
- calculator.py
- README.md

2. THE GIT REPOSITORY CONTAINS INFORMATION THAT ALLOWS GIT TO MANAGE THE HISTORY OF THOSE FILES.

This is not your code. It's the data that helps Git track every change.

THINK OF IT LIKE:

PROJECT

Your files

- app.py
- calculator.py
- README.md

These are the files you create and edit.

Git data

- history
- commits
- branches
- etc.

This is the Git information that records everything about your project.

3. WHERE IS THE GIT DATA STORED?

The Git data is stored inside the `.git` directory.

```
python-project/
→ .git/   ⟷ Git's internal repository information
→ app.py
→ calculator.py
→ README.md
```

This is what allows Git to maintain your project's history safely and reliably.

In short:
Your files are the content. | The `.git` directory is the intelligence. | Together, they make it a Git repository.

CDU
Cloud And Data Universe

WHAT IS A COMMIT?

git

A commit is a recorded point in the history of your project.

1 WHAT IS A COMMIT?

A commit is a recorded point in the history of your project.

IMAGINE THIS:

- Commit 1 "Create application"
- Commit 2 "Add login"
- Commit 3 "Add registration"
- Commit 4 "Fix login bug"

YOUR HISTORY BECOMES: Commit 1 → Commit 2 → Commit 3 → Commit 4

Each commit records a particular state/change in the project's history.

2 THINK OF A COMMIT LIKE A CHECKPOINT

Imagine you're playing a game.

You reach an important point and save your game.

Checkpoint 1 → Checkpoint 2 → Checkpoint 3

Git commits work somewhat like checkpoints. You reach a meaningful point in your work and secord it. Later, you can inspect those checkpoints.

Project → Commit → Project history

3 A COMMIT HAS INFORMATION

A commit isn't just: **"Save the files."**

It contains information such as:

Commit
- What changed
- Who made the change
- When it was made
- Commit message
- Unique identifier

FOR EXAMPLE:

- Commit message: "Add user authentication"
- Author: Developer A
- Date: August 8, 2026
- Commit ID: a1b2c3d4e5f6g7h8i9j0

This is why Git history becomes useful for teams.

KEY TAKEAWAY

- A commit captures a moment in time.
- You can explore, compare and go back if needed.
- It brings clarity, accountability and collaboration.
- Better commits lead to better projects!

CDU
Cloud And Data Universe

GIT IS DISTRIBUTED

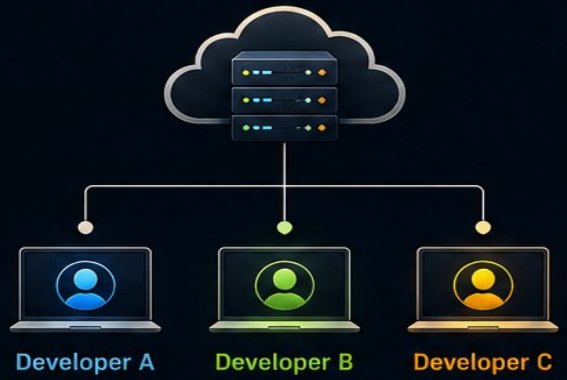
Understanding the Power of Git's Distributed Model

git

TRADITIONAL CENTRALIZED MODEL

Imagine a company has a central repository.

CENTRAL SERVER



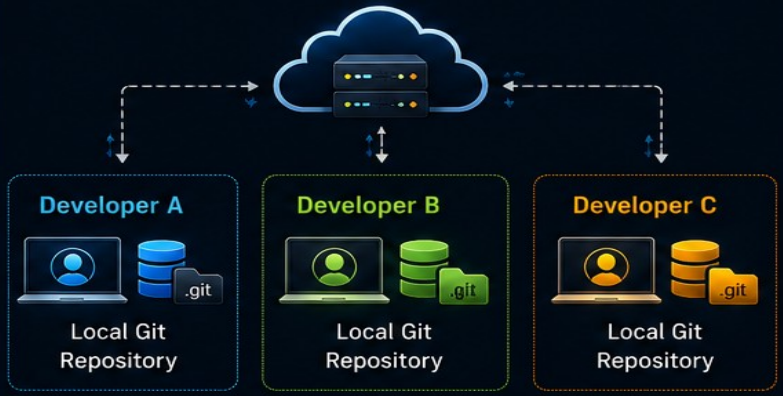
The developers depend heavily on that central server.

This is the basic idea behind **centralized version control.**

GIT USES A DISTRIBUTED MODEL

Git works differently.
Each developer can have a **complete** Git repository **locally.**

REMOTE REPOSITORY



Each developer's local repository can contain the project's history.

That's why Git is called:

DISTRIBUTED VERSION CONTROL SYSTEM

WHY IS THIS POWERFUL?

Suppose you're traveling and **don't have internet access.**

You can **still work** on your Git project.

- Make changes
- Create commits
- View history
- Create branches
- Compare changes

because much of the Git functionality **exists locally.**

Later, when you're connected again, you can **synchornize** with a remote repository.

DISTRIBUTED MODEL GIVES YOU:

- Work offline with full history
- Faster performance
- No single point of failure
- More flexibility and freedom
- Better collaboration at any scale

CDU
Cloud And Data Universe

LOCAL vs REMOTE

Understanding the Two Important Concepts in Git

git

LOCAL REPOSITORY

The Git repository on your computer.

REMOTE REPOSITORY

A repository hosted somewhere else.

For example:

- GitHub
- GitLab
- Azure DevOps

CONCEPTUALLY

We'll learn how these communicate later in the tutorial.

IMPORTANT: GIT DOES NOT REQUIRE GITHUB

This is another common beginner misconception.
You can use Git **completely locally.**

For example:

```
Your Computer
python-project/
├── app.py
├── README.md
└── .git/
```

You can create commits and maintain history **without GitHub.**

GitHub is useful when you want things like:

- Remote storage
- Collaboration
- Pull requests
- Code review
- Team workflows

Similarly, you could use **GitLab** or **Azure DevOps.**

KEY TAKEAWAY

- Local repository lives on **your computer.**
- Remote repository lives on a server somewhere else on the **internet.**
- Git lets you work locally and **sync** with remote when needed.
- Git is the tool. GitHub/GitLab/Azure DevOps are **platforms** built around Git.

CDU
Cloud And Data Universe

GIT vs GITHUB

git

Let's make the distinction very clear.

GIT

Version control system

Git is the core technology that tracks changes in your files and maintains history.

VS

GITHUB

Platform built around Git

GitHub is a service that provides hosting, collaboration and additional features on top of Git.

THE ECOSYSTEM

GIT
Version Control System

- **GitHub** — Cloud platform for hosting Git repositories, collaboration, pull requests, issues, actions & more.
- **GitLab** — DevOps platform with Git repos, CI/CD, issue tracking, security & more.
- **Azure DevOps** — Microsoft's DevOps platform with Git repos, CI/CD pipelines, boards, artifacts & more.

KEY DIFFERENCE

Git is a tool (software).
GitHub is a platform (service).

TOOL → PLATFORM

THE BOTTOM LINE

All three (GitHub, GitLab, Azure DevOps) can work with Git repositories.

But Git itself is the underlying version control technology.

Think of it this way: Git is the engine, platforms are the vehicles that use that engine!

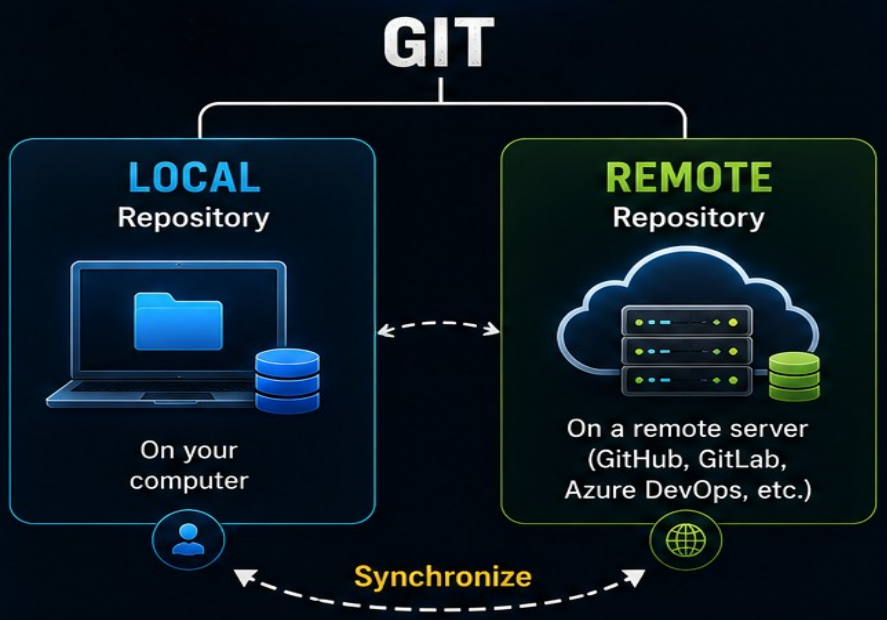
CDU
Cloud And Data Universe

GIT'S BASIC ARCHITECTURE

git

Understanding how Git works at a high level

AT A HIGH LEVEL



Changes made locally can be synchronized (pushed) to the remote repository, and updates from remote can be pulled to your local repository.

INSIDE YOUR LOCAL WORKFLOW

We'll eventually have this flow:

DON'T WORRY ABOUT THE STAGING AREA YET!

We will learn about the staging area in detail in the next lesson.
For now, focus on the big picture.

Next Major Concept:
The Three States in Git

CDU
Cloud And Data Universe
★ Your Learning Universe ★

GIT CONFIG

TELL GIT WHO YOU ARE

Git records every change you make in the form of commits.
Each commit stores the author's **name** and **email**.
That's what `git config` helps us do.

git

KEY IDEA
You configure Git once, and it is used for all your repositories (with `--global`).

WHY CONFIGURE?

- Git needs to know who you are when you create commits.
- Adds your name and email to every commit.
- You set it once, and it is used across all your projects.

HOW TO CONFIGURE (Run in VS Code Terminal)

1. Set your name

```
git config --global user.name "Your Name"
# Example:
git config --global user.name "Yusuf Didighar"
```

2. Set your email

```
git config --global user.email "youremail@example.com"
# Example:
git config --global user.email "yusuf@example.com"
```

3. Verify your configuration

```
git config --global --list
```

CONFIGURATION LEVELS

`--global` (Default)
Applies to your user account on this computer.
(Recommended for most users)

`--local`
Applies only to the current repository.

`--system`
Applies to the entire system.
(Requires admin rights)

CHECK INDIVIDUAL VALUES

```
git config user.name
git config user.email
```

★ Shows the value for that setting.

EXAMPLE OUTPUT

```
user.name=Yusuf Didighar
user.email=yusuf@example.com
```

Example Commit

```
Author: Yusuf Didighar <yusuf@example.com>
Date:   Wed Aug 14 2026 10:30:00 +0530

Initial commit
```

WHEN TO CONFIGURE?

- Do it before your first commit.
- You rarely need to change it again.

Set it once.
Focus on your code!

★ `git config` sets your identity. Git uses it to give credit to your work.

CONFIG ONCE, COMMIT WITH CONFIDENCE!

CDU
— Cloud And Data Universe —
Learn • Build • Grow

GIT INIT

Turn a Folder into a Git Repository

`git init` initializes a new Git repository in your project.

Steps in VS Code

1. **Create Project Folder**
   Create a folder (e.g., git-demo) and add your files.
2. **Open in VS Code**
   Open the folder in VS Code.
3. **Open Terminal**
   Terminal → New Terminal
4. **Run git init**
   `git init`
   Initialize a Git repository.
   ✓ You will see: Initialized empty Git repository
5. **Check in VS Code**
   Source Control panel will now show your project as a Git repository.

What does `git init` do?

- ✓ Creates a `.git` directory
- ✓ Initializes a Git repository
- ✓ Sets up Git to track changes in your project

Before `git init`

```
git-demo/
├── app.py
├── calculator.py
└── README.md
```

✗ Just a normal project folder

After `git init`

```
git-demo/
├── .git/
├── app.py
├── calculator.py
└── README.md
```

✓ Git repository ready

Terminal Example

```
PS C:\git-demo> git init
Initialized empty Git repository in C:/git-demo/.git/
PS C:\git-demo>
```

What is .git?

The `.git` directory is where Git stores all the repository's internal data — **history**, **commits**, **branches** and more.

```
.git/
├── history
├── commits
├── branches
└── config
```

Key Commands

`git init`	Create a new Git repository
`git status`	Check repository status
`git config --global`	Set your identity

Remember:

- ✓ `git init` does not commit your files.
- ✓ Your files remain unchanged.
- ✓ It only enables Git to track changes.

Run `git init` in VS Code and make your project a Git repository!

CDU — Cloud And Data Universe —

CDU
Cloud And Data Universe
Learn • Build • Grow

GIT WORKFLOW – THE CORE 3 COMMANDS

git status → git add → git commit

From changes in your files to a saved snapshot in Git history.

git

1. Make Changes — Edit files in your working directory.
→ **2. git status** — Check what changed.
→ **3. git add** — Stage the changes you want to commit.
→ **4. git commit** — Save the staged changes into Git history.
→ **5. History** — Your project history grows commit by commit.

1. git status

Shows the current state of your working directory.

In VS Code (Source Control)

M = **Modified** (changed)
U = **Untracked** (new file)
D = **Deleted**

In Terminal

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
        modified:   app.py
        modified:   README.md

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

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

Use `git status` frequently to know what Git sees before you stage or commit.

2. git add

Stages changes to be included in the next commit.

In VS Code (Staging Changes)

Click the ⊕ icon next to files to stage them, or click the ⊕ at the top to stage all.

In Terminal

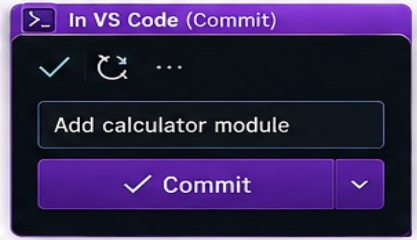
```
$ git add app.py              # stage a specific file
$ git add README.md           # stage another file
$ git add .                   # stage all changes
$ git status                  # check again

On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   calculator.py
        modified:   app.py
        modified:   README.md
```

Staging is like making a selection.
Only staged changes will be committed.

3. git commit

Saves staged changes as a snapshot in the repository history.

In VS Code (Commit)



Write a meaningful commit message explaining what changes you made and why.

In Terminal

```
$ git commit -m "Add calculator module"
[main 7f3a9c1] Add calculator module
 3 files changed, 12 insertions(+)
 create mode 100644 calculator.py

$ git status
On branch main
nothing to commit, working tree clean
```

Each commit is a checkpoint (snapshot) of your project at a point in time.

THE BIG PICTURE

Edit — Make changes to files
→ **git status** — See what changed
→ **git add** — Stage the changes
→ **git commit** — Save to history
→ **History** — Your project history grows

BEST PRACTICES

- Check status before every commit.
- Stage only what you want to commit. (Use `git add -p` for selective staging)
- Write clear, meaningful commit messages.

KEY TAKEAWAY

status → add → commit
Inspect. Select. Save.
This is the heart of Git!

★ Master these 3 commands and you can control your project history with confidence!

Happy Coding!

CDU – Cloud And Data Universe

GIT DIFF

SEE EXACTLY WHAT CHANGED

`git diff` shows you the differences between changes.
It is your superpower to understand what, why and how your code has changed.

THE BIG IDEA

Working Directory → Compare (Diff) → Last Commit / Reference

`git diff` compares your current files with a reference and shows the exact changes.

WHAT IS GIT DIFF?

Every change you make in your Working Directory compared to the last commit (or any reference) can be viewed using `git diff`.

Working Directory → `git diff` → Last Commit / Reference

It helps you **review** changes before staging and committing.

MOST COMMON USAGE

`git diff`
Shows changes between Working Directory and the last commit.

`git diff --staged`
Shows changes between Staging Area and the last commit.

`git diff HEAD`
Same as `git diff` (compares Working Directory with the last commit).

`git diff <commit1> <commit2>`
Shows changes between two commits.
Example: `git diff a1b2c3d d4e5f6g`

EXAMPLE: CHANGES IN app.py

LAST COMMITTED VERSION

```
1  print("Welcome to my Python application")
2  version = "1.0"
3  
4  print("Thank you!")
```

VS

CURRENT VERSION (WORKING DIRECTORY)

```
1  name = input("Enter your name: ")
2  print(f"Welcome, {name}")
3  version = "1.1"
4  
5  print("Thank you so much!")
```

OUTPUT OF `git diff`

```
diff --git a/app.py b/app.py
index e69de29..b1c2d3e 100644
--- a/app.py
+++ b/app.py
@@ -1,4 +1,5 @@
-print("Welcome to my Python application")
-version = "1.0"
-
+name = input("Enter your name: ")
+print(f"Welcome, {name}")
+version = "1.1"
+
+print("Thank you so much!")
```

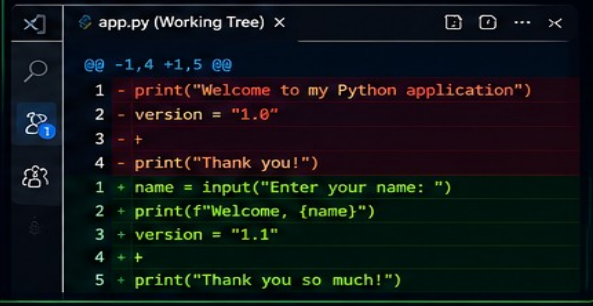
LEGEND

- `-` Removed
- `+` Added
- `@@` Context (line numbers)

- `-` (minus) lines are removed
- `+` (plus) lines are added
- `@@` shows the line numbers around the change (context)

IN VS CODE – VISUAL DIFF

Click on a file in the Source Control panel to see the diff in VS Code.



KEY POINTS TO REMEMBER

- `git diff` does NOT change anything.
- It only shows you the differences.
- It helps you review changes before staging and committing.
- Always review your diff before `git add` and `git commit`.
- It's one of the most important Git commands.

PRACTICAL EXAMPLES

`git diff`	See changes not yet staged
`git diff --staged`	See what you are about to commit
`git diff HEAD`	See changes compared to last commit
`git diff main`	See changes compared to main branch
`git diff a1b2c3d d4e5f6g`	Compare two commits

WHERE DOES GIT DIFF FIT?

1. **EDIT** — You modify files in your Working Directory.
2. **CHECK** — Run `git diff` / `git status` to see what changed.
3. **STAGE** — Run `git add` to move changes to Staging Area.
4. **REVIEW** — Run `git diff --staged` to review what you will commit.
5. **COMMIT** — Run `git commit` to save changes to history.

> Good developers **review** changes.
> Great developers **understand** changes.

CDU
CLOUD AND DATA UNIVERSE

GIT LOG

YOUR PROJECT'S HISTORY BOOK

`git log` shows the commit history of your repository.
It tells you **WHO** made the change, **WHEN**, and **WHAT** was changed (through the commit message).

THE BIG IDEA

Commits → `git log` → History

See your project's journey from start to now.

WHAT IS GIT LOG?

Displays a list of all commits in reverse chronological order (newest first). Each entry shows:

Author	–	Who made the commit
Date	–	When it was made
Message	–	What the commit is about
Commit ID	–	Unique identifier for the commit

It helps you understand the history, track changes, and find when something was introduced or fixed.

EXAMPLE: `git log`

```
PS D:\git-demo-bk> git log

commit 8f42c91a9e3b7c2f4e6d3a1b8c9d0e7f12345678
Author: John Doe <john@example.com>
Date:   Sun Aug 16 11:20:00 2026 +0530

    Fix greeting message

commit 31ac7de2b8f6a1c0d4e5f67890abcde123456789
Author: John Doe <john@example.com>
Date:   Sun Aug 16 10:45:00 2026 +0530

    Add user greeting

commit a72bc11c9e5d4f6a8b7c6d5e4f3a2b1c0d9e8f7a
Author: John Doe <john@example.com>
Date:   Sun Aug 16 10:15:00 2026 +0530

    Initial commit
```

NEWEST ↓ OLDEST

`git log` shows the newest commit first and goes back in time.
Read it from **TOP** to **BOTTOM**.

KEY POINTS TO REMEMBER

- `git log` shows the commit history.
- Latest commit appears first.
- Each commit has an ID, author, date, and message.
- Use `git log --oneline` for a compact view.
- You can compare any two commits using their IDs.
- Short commit IDs work as long as they are unique.

USEFUL OPTIONS

`git log --oneline`	Compact one-line view
`git log --oneline --graph`	Show branch history (graph)
`git log -n <number>`	Show last N commits
`git log --author="name"`	Filter by author name
`git log --since="2026-08-01"`	Show commits since date
`git log --until="2026-08-16"`	Show commits until date
`git log -- <file>`	Show history for a file

UNDERSTANDING THE OUTPUT

```
commit 8f42c91a9e3b7c2f4e6d3a1b8c9d0e7f12345678
Author: John Doe <john@example.com>      ← Author
Date:   Sun Aug 16 11:20:00 2026 +0530   ← Date

    Fix greeting message                 ← Commit Message
```

Commit ID	:	Unique SHA-1 hash of the commit
Author	:	Who created the commit
Date	:	When the commit was created
Message	:	What this commit represents

A COMPACT VIEW: `git log --oneline`

```
PS D:\git-demo-bk> git log --oneline
8f42c91 Fix greeting message
31ac7de Add user greeting
a72bc11 Initial commit
```

`--oneline` shows each commit in a single line. Great for quick overview!

PRACTICAL EXAMPLES

```
git log --oneline                    # Quick history
git log --oneline --graph            # See branches visually
git log -n 5                         # Last 5 commits
git log --author="John"              # Commits by John
git log --since="2026-08-01"         # Since Aug 1, 2026
git log 985e4d6..d1f5bb6             # Commits between two IDs
```

WHERE DOES GIT LOG FIT?

1. **EDIT** — Make changes to your files.
2. **REVIEW CHANGES** — Use `git diff` to see changes.
3. **STAGE** — Use `git add` to stage changes.
4. **COMMIT** — Use `git commit` to save changes.
5. **VIEW HISTORY** — Use `git log` to see the full history.
6. **CONTINUE** — Keep building and improving.

SUMMARY

> Every commit you make is a step in your project's journey. `git log` helps you look back, understand the story, and move forward with confidence.

CDU CLOUD AND DATA UNIVERSE

GIT SHOW

INSPECT A COMMIT IN DETAIL

`git show` lets you explore a commit. It shows the commit **information** and the **changes** introduced by that commit.

WHERE ARE YOU? HEAD

HEAD points to your current commit (D) on branch master

WHAT IS GIT SHOW?

Shows details about a commit:

- Commit information (author, date, message)
- The changes (diff) introduced by the commit
- Files affected in that commit
- Statistics and summary of changes

Think of it as "opening" a commit to see what happened inside.

EXAMPLE: git show

```
PS D:\git-demo-bk> git show
commit b158d9e16fbb48f3ece6e502dc4bc59535df64ed (HEAD -> master)
Author: Yusuf Didighar <yusuf@example.com>
Date:   Mon Aug 19 11:42:10 2026 +0530

    Add user greeting

diff --git a/app.py b/app.py
index e69de29..7c3a1b2 100644
--- a/app.py
+++ b/app.py
@@ -0,0 +1,3 @@
+name = input("Enter your name: ")
+print(f"Welcome, {name}!")
```

Commit Information

Changes (diff) introduced by this commit

Running `git show` without arguments shows the commit HEAD points to.

COMMON FORMS

`git show`	Show the commit at HEAD (current commit)
`git show HEAD`	Same as above (explicit)
`git show <commit-id>`	Show a specific commit by ID (short or full)
`git show HEAD~1`	Show the previous commit
`git show HEAD~2`	Show two commits before HEAD
`git show HEAD~n`	Show n commits before HEAD
`git show <commit>:<file>`	Show a file as it existed in that commit
`git show <commit> -- <file>`	Show changes in that commit for a file

Use short commit IDs whenever possible (e.g., `b158d9e`).

ANATOMY OF THE OUTPUT

- Commit ID
- Author
- Date
- Commit Message
- Diff (Changes) — Exact changes made in that commit

```
commit b158d9e16fbb48f3ece6e502dc4bc59535df64ed
Author: Yusuf Didighar <yusuf@example.com>
Date:   Mon Aug 19 11:42:10 2026 +0530

    Add user greeting

diff --git a/app.py b/app.py
index e69de29..7c3a1b2 100644
--- a/app.py
+++ b/app.py
@@ -0,0 +1,3 @@
+name = input("Enter your name: ")
+print(f"Welcome, {name}!")
```

NAVIGATE HISTORY USING HEAD

`git show HEAD`	Show current commit
`git show HEAD~1`	Show previous commit
`git show HEAD~2`	Two commits before
`git show HEAD~n`	n commits before
`git show HEAD^`	Same as HEAD~1
`git show HEAD^^`	Same as HEAD~2

HEAD~n is very helpful to move backward through history.
Example: `git show HEAD~3` → 3 commits before current.

USEFUL OPTIONS

`git show --no-patch <commit>`	Show commit info only (no diff)
`git show --name-only <commit>`	Show only the names of files changed
`git show --name-status <commit>`	Show files with status (A/M/D)
`git show --stat <commit>`	Show statistics (insertions/deletions)
`git show <commit>:<file>`	Show a file as it existed in that commit
`git show <commit> -- <file>`	Show changes for a specific file in that commit

A = Added M = Modified D = Deleted R = Renamed

PRACTICAL EXAMPLES

1. Show current commit

```
PS D:\git-demo-bk> git show
commit b158d9e (HEAD -> master)
Author: Yusuf <yusuf@example.com>
Date:   Mon Aug 19 11:42:10 2026 +0530
    Add user greeting
... diff output ...
```

2. Show previous commit

```
PS D:\git-demo-bk> git show HEAD~1
commit 985e4d6
Author: Yusuf <yusuf@example.com>
Date:   Sun Aug 18 16:20:00 2026 +0530
    Add calculator
... diff output ...
```

3. Show file as it was in a commit

```
PS D:\git-demo-bk> git show HEAD~1:app.py

def add(a, b):
    return a + b

print("This is calculator")
```

4. Show only changed files

```
PS D:\git-demo-bk> git show --name-only HEAD

app.py
README.md
```

5. Show summary stats

```
PS D:\git-demo-bk> git show --stat HEAD

app.py      | 4 +++-
README.md   | 1 +
2 files changed, 4 insertions(+), 1 deletion(-)
```

WHERE DOES GIT SHOW FIT?

1. Make Changes — Edit your files
2. Stage Changes — `git add`
3. Commit Changes — `git commit -m "..."`
4. See History — `git log`
5. Inspect a Commit — `git show <commit>`

QUICK TIPS

- `git show` is read-only, it does not change anything.
- Use short IDs to quickly inspect any commit.
- Combine with `HEAD~n` to explore history easily.
- Great for code reviews and debugging.

SUMMARY

`git show` opens a commit and shows you everything you need to know about it — who made it, when, what message they wrote, and exactly what changes were introduced.

CDU
CLOUD AND DATA UNIVERSE

GIT BRANCH & SWITCH

Create branches, switch between them and see divergence in action.

CDU — Cloud And Data Universe —

1 CHECK CURRENT BRANCH

See which branch you are currently on.

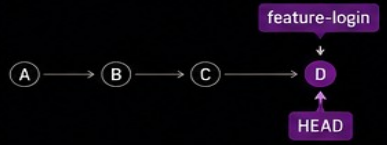
```
PS D:\git-demo> git branch
* master
```

* indicates the current branch (HEAD points here).

2 CREATE & SWITCH TO A NEW BRANCH

Create a new branch and switch to it in one step.

```
PS D:\git-demo> git switch -c feature-login
```

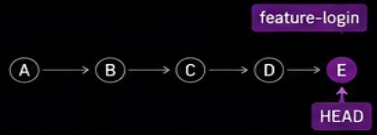


New branch starts at the same commit. No divergence yet.

3 MAKE CHANGES, STAGE & COMMIT

Make some changes and commit them to the current branch.

```
PS D:\git-demo> git status
PS D:\git-demo> git add app.py
PS D:\git-demo> git commit -m
 "Add login feature"
```

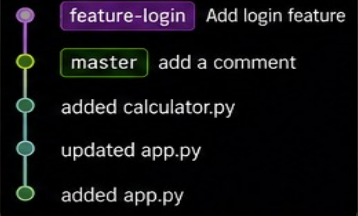


feature-login moved forward. Now branches have diverged!

4 VIEW IN GIT GRAPH

Open the Git Graph extension to see the divergence.

Git Graph View



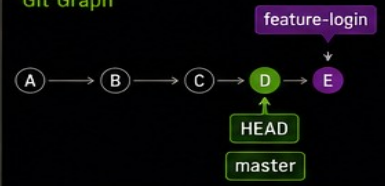
Two branches, two different commits.

5 SWITCH BACK TO MASTER

Switch back to master branch.

```
PS D:\git-demo> git switch master
```

Git Graph



HEAD now points to master. Your feature commit is safe in * feature-login.

6 LIST BRANCHES

See all branches in the repository.

```
PS D:\git-demo> git branch
  feature-login
* master
```

* master → current branch (HEAD)

feature-login → another branch

Use this often to know where you are.

7 CREATE A BRANCH FROM ANOTHER BRANCH

Create a new branch starting from an existing branch.

```
PS D:\git-demo> git branch
 feature-validation feature-login
```

Git Graph

Both branches point to the same commit (E).

8 CREATE A BRANCH FROM A SPECIFIC COMMIT (OPTIONAL)

You can start a branch from any commit using its ID.

```
PS D:\git-demo> git branch
 experiment C
```

Git Graph

Branch 'experiment' starts from commit C.

9 SWITCH TO PREVIOUS BRANCH (QUICK TIP)

Go back to the branch you were on before.

```
PS D:\git-demo> git switch -
```

Example

master → feature-login

git switch -

Back to master

Very handy when you switch back and forth.

10 KEY TAKEAWAYS

- A branch is a pointer to a commit.
- Creating a branch does NOT create divergence.
- Switch to the branch, make changes and commit.
- Git Graph helps you see what's happening.
- Your feature work is isolated until you merge it.

PRACTICE FLOW (HANDS-ON SEQUENCE)

1. Check Branch — `git branch`
2. Create & Switch — `git switch -c`
3. Make Changes — Edit your files
4. Stage Changes — `git add`
5. Commit Changes — `git commit -m "..."`
6. View Graph — Git Graph
7. Switch Back — `git switch master`

QUICK TIPS

- `git show <commit>` → full commit details
- Use short IDs for speed
- Combine with `HEAD~` to explore history
- Great for debugging & code reviews

NEXT STEP

Merging Branches

Bringing your work back together.

CLOUD AND DATA UNIVERSE
LEARN • BUILD • GROW

GIT MERGE
FAST FORWARD, MERGE WITH COMMIT & MERGE WITH COMMIT (NO FAST FORWARD)

Understand the 3 Most Common Ways to Integrate Changes in Git

WHAT IS MERGE?

Merging in Git is the process of integrating changes from one branch into another. Git provides different strategies to combine the history of branches.

KEY POINTS

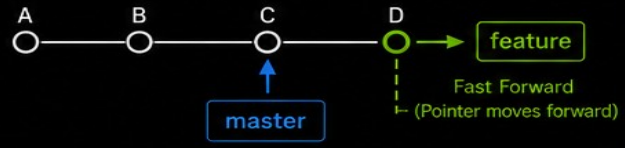
- Merge is used to bring changes from one branch into another.
- Git decides how to integrate based on the commit history.
- You can allow Git to Fast Forward, or create a Merge Commit to preserve branch history.

1 GIT MERGE (FAST FORWARD)

If the target branch has not diverged, Git simply moves the pointer forward. No new merge commit is created.

SCENARIO

feature → master (no divergence)



RESULT

master moves to D. History remains linear.

```
git checkout master
git merge feature    # Fast Forward
```

WHEN TO USE?

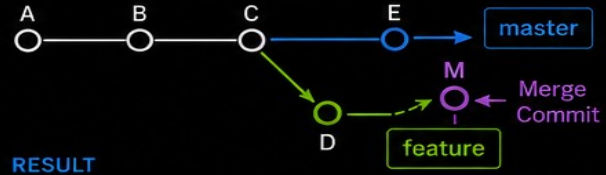
- Use when the branch has a straight line of commits.
- Keeps history clean and simple.

2 GIT MERGE WITH COMMIT (NO FAST FORWARD)

If the branches have diverged, Git creates a new merge commit to combine the histories.

SCENARIO

feature → master (diverged)



RESULT

A new merge commit (M) is created.
History shows both branches and the merge point.

```
git checkout master
git merge feature    # Creates a merge commit
                     # (no fast forward)
```

WHEN TO USE?

- Use when branches have diverged.
- Preserves full history of both branches.
- Useful for collaboration and auditing.

3 GIT MERGE WITH COMMIT (NO FAST FORWARD) EXPLICITLY (–no-ff)

Even if fast forward is possible, you can force Git to create a merge commit using –no-ff.

SCENARIO

feature → master (no divergence, but force merge commit)

RESULT

A merge commit (M) is created even though Fast Forward was possible.

```
git checkout master
git merge feature --no-ff   # Force merge commit
```

WHEN TO USE?

- Use when you want a merge commit for tracking, even if fast forward is possible.
- Helps maintain consistency in team workflows.

QUICK TIP

Always pull the latest changes before merging.

Understand your history. Better collaboration starts with clarity.

Practice these scenarios and master Git like a pro!

Happy Learning!
– CDU Team

GIT MERGE CONFLICT

Git found different changes in the same place.
You decide how to bring them together.

CDU — Cloud And Data Universe —

1 WHAT IS A MERGE CONFLICT?

Two branches change the same part of a file differently. Git can't decide automatically.

YOUR BRANCH

```
def calculate():
    return 200
```

OTHER BRANCH

```
def calculate():
    return 500
```

Git stops and shows a MERGE CONFLICT.
You decide which changes to keep.

2 HOW VS CODE SHOWS IT

Conflicting file will look like this:

```
<<<<<<< HEAD
  print("Hello from main")
=======
  print("Hello from feature")
>>>>>>> feature
```

- `<<<<<<< HEAD` → Your current branch (HEAD)
- `=======` → Separator
- `>>>>>>> feature` → Incoming branch

Conflict is NOT an error.
It's Git asking you to make the right decision.

MERGE CONFLICT

- **CURRENT CHANGE** — Your branch
- **INCOMING CHANGE** — Other branch
- **YOU DECIDE**
  - **KEEP CURRENT** — Accept Current Change — Keep your version
  - **KEEP INCOMING** — Accept Incoming Change — Take incoming version
  - **KEEP BOTH** — Accept Both Changes — Keep both versions
- **OR MANUALLY COMBINE** — Edit the file to create the correct final version
- **SAVE** — Save the resolved file
- `git add` — Stage the resolved file
- `git commit` — Complete the merge

3 AFTER RESOLVING CONFLICTS

1. Check status

```
PS D:\git-demo> git status
You have unmerged paths.
  (fix conflicts and run "git commit")
  both modified:      app.py
```

2. Stage resolved files

```
PS D:\git-demo> git add app.py
```

3. Commit to complete merge

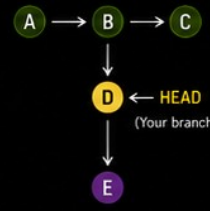
```
PS D:\git-demo> git commit
Merge branch 'feature' into 'main'
```

Merge completed!
Your branches are now combined.

4 TIPS

- Use `Compare Changes` in VS Code to understand both versions.
- Don't blindly accept either change. Understand the business logic.
- Resolve all files with conflicts before committing.
- `git add` marks the conflict as resolved.
- Always test your code after merging.

WHERE ARE YOU?



HEAD points to your current commit (D) during merge.

QUICK CONFLICT EXAMPLE

HEAD (main)

```
value = 100
```

INCOMING (feature)

```
value = 200
```

CONFLICT VIEW

```
<<<<<<< HEAD
 value = 100
=======
 value = 200
>>>>>>> feature
```

RESOLVED MANUALLY

```
value = 150  # final decision
```

MERGE CONFLICT RESOLUTION FLOW (HANDS-ON SEQUENCE)

1. Conflict Occurs — `git merge feature`
2. Open in VS Code — See conflict markers
3. Understand Changes — Compare both versions
4. Resolve Conflict — Edit & save file
5. `git add` — Stage resolved file
6. `git commit` — Complete merge

REMEMBER

Git can't know which change is correct for your project.

YOU ARE THE DECIDER.

CLOUD AND DATA UNIVERSE
Learn. Build. Succeed.
www.cloudanddatauniverse.com

GIT RM — REMOVE A TRACKED FILE

Remove a file from your project and tell Git about it — in one step.

`git rm <file>`
Removes the file **AND** stages the deletion.

CDU — Cloud And Data Universe —

1 THE SCENARIO

Suppose `config.py` is tracked by Git and you no longer need it.

Your project
config.py

DON'T DO THIS (Manual Way)

```
rm config.py
```

Now the file is gone from your working directory, but Git sees:

```
$ git status
    deleted:   config.py
```

2 THE RIGHT WAY: LET GIT HANDLE IT

Use `git rm` to remove the file and stage the deletion.

```
$ git rm config.py
```

What just happened?

- `config.py` is deleted from your working directory.
- The deletion is staged for your next commit.

3 CHECK THE STATUS

Run:

```
$ git status
```

You'll see:

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:    config.py
```

The deletion is staged!

4 COMMIT THE CHANGE

Run:

```
$ git commit -m "Remove config.py"
```

`config.py` is now permanently removed from the repository history (in a new commit).

5 VERIFY

Check again:

```
$ git status
```

You'll see:

```
On branch main
nothing to commit, working tree clean
```

WHAT HAPPENS?

`git rm config.py`
↓
File removed from working directory
↓
Deletion staged (in the index)
↓
Commit to save permanently

KEY IDEA

`git rm` removes the file AND stages the deletion.

It keeps Git and your filesystem in sync — the right way!

BEFORE vs AFTER

BEFORE (File Tracked)

config.py (tracked) → Git Index → Repository

AFTER `git rm config.py`

config.py (removed) → Deletion Staged → (Commit next)

COMMAND SUMMARY

`git rm <file>`	`git status`	`git commit -m "msg"`	`git status`
Remove & stage deletion	Verify staged deletion	Commit the deletion	Clean working tree

REMEMBER

Use `git rm` instead of manually deleting files so Git knows exactly what changed.

CLOUD AND DATA UNIVERSE
Learn. Build. Succeed.
www.cloudanddatauniverse.com

GIT MV — RENAME OR MOVE A TRACKED FILE

Rename or move a file in your project and let Git handle everything — in one step.

```
git mv <source> <destination>
```

Renames or moves the file AND stages the change.

— Cloud And Data Universe —

1 THE SCENARIO

Suppose:

old_name.py

should become:

new_name.py

MANUAL WAY (NOT IDEAL)

```
mv old_name.py new_name.py
```

You renamed the file, but Git may see it as a deletion + new file!

2 THE RIGHT WAY: USE GIT MV

Run:

```
$ git mv old_name.py new_name.py
```

Git handles:

- Renames (or moves) the file
- Stages the rename for your next commit

3 CHECK THE STATUS

Run:

```
$ git status
```

You may see:

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    old_name.py -> new_name.py
```

The rename is staged!

WHAT HAPPENS?

```
$ git mv old_name.py new_name.py
```

↓

File renamed (or moved)

↓

Rename staged (in the index)

↓

Commit to save permanently

4 COMMIT THE CHANGE

Run:

```
$ git commit -m "Rename old_name.py to new_name.py"
```

The rename is now saved in your repository history.

5 VERIFY

Check again:

```
$ git status
```

You'll see:

```
On branch main
nothing to commit, working tree clean
```

6 MOVING TO A FOLDER

Suppose:

script.py

should move into:

src/script.py

Run:

```
$ git mv script.py src/script.py
```

Git handles both the move and staging.

KEY IDEA

`git mv` renames/moves a tracked file and stages the change.

It keeps Git and your filesystem in sync — the right way!

BEFORE vs AFTER

BEFORE (File Tracked)

old_name.py (tracked) → Git Index → Repository

AFTER `git mv old_name.py new_name.py`

old_name.py (renamed) → new_name.py (staged) → (Commit next)

COMMAND SUMMARY

1. `git mv <source> <destination>`
2. `git status` — See rename staged
3. `git commit -m "msg"` — Commit the rename
4. `git status` — Clean working tree

REMEMBER

Use `git mv` instead of manually renaming or moving files so Git understands the change correctly.

CLOUD AND DATA UNIVERSE

Learn. Build. Succeed.

www.cloudanddatauniverse.com

GIT RESTORE — UNDO WORKING-DIRECTORY CHANGES

Undo accidental changes in your files and get back to the last committed version.

`git restore <file>`

Restores the file in your working directory to the last commit.

— Cloud And Data Universe —

1 THE SCENARIO

Suppose the committed version of app.py contains:

app.py (committed)

```
def greet():
    print("Hello World")
```

You modify it:

app.py (modified)

```
def greet():
    print("Hello, Git!")  # changed
```

But then realize:

"I don't want these changes."

2 THE RIGHT WAY: USE GIT RESTORE

Run:

```
$ git restore app.py
```

What happens?

- Git discards your local changes to app.py.
- The file is restored to match the last committed version.

3 CHECK THE RESULT

Check the file:

```
$ cat app.py
```

You'll see:

app.py (restored)

```
def greet():
    print("Hello World")  # back to committed version
```

Back to the last committed version!

WHAT HAPPENS?

```
$ git restore app.py
```

↓

Git replaces the contents in your working directory

↓

File matches the last committed version

↓

Changes in working directory are gone

! IMPORTANT

`git restore` is primarily about restoring file contents.

It does NOT move your branch backward.

It does NOT delete any commits.

It only affects your working directory (and optionally the staging area).

4 SEE THE DIFFERENCE

BEFORE CHANGE (Committed)	AFTER CHANGE (Modified)	AFTER RESTORE (Back to committed)
app.py	app.py	app.py
`def greet():` `  print("Hello World")`	`def greet():` `  print("Hello, Git!")`	`def greet():` `  print("Hello World")`
COMMITTED	MODIFIED	RESTORED

KEY IDEA

Use `git restore` when you want to discard unwanted changes in your working directory and get the file back to the way it was in the last commit.

COMMON VARIATIONS

Restore multiple files

```
$ git restore file1.py file2.py
```

Restore all changed files in the current folder

```
$ git restore .
```

COMMAND SUMMARY

Make unwanted changes → Realize the mistake → `git restore <file>` Restore file to last committed version → File is back to normal

REMEMBER

`git restore` = Undo local changes (working directory) NOT history.

CLOUD AND DATA UNIVERSE

Learn. Build. Succeed.

www.cloudanddatauniverse.com

GIT RESTORE --STAGED — UNSTAGE A FILE

Remove a file from the staging area
but keep your changes in the working directory.

```
git restore --staged <file>
```
Unstages the file but keeps your changes.

— Cloud And Data Universe —

1 THE SCENARIO

Suppose you modify app.py:

app.py (modified)
```
def add(a, b):
    return a + b  # updated logic
```

Then you stage it:

app.py (staged)
```
def add(a, b):
    return a + b  # updated logic
```

Now:

app.py is staged for commit.

2 OOPS! CHANGE OF PLAN

You realize:

"Oops, I don't want to commit this yet."

You want to:

- Keep the changes in your file
- But remove it from staging

3 THE RIGHT WAY: UNSTAGE IT

Run:

```
$ git restore --staged app.py
```

What happens?

- Git removes app.py from the staging area.
- Your changes in app.py are kept in the working directory.

4 CHECK THE STATUS

Run:

```
$ git status
```

You'll see:

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update
    what will be committed)
        modified:   app.py
```

app.py is no longer staged
but your changes are still there!

WHAT HAPPENS?

```
$ git restore --staged app.py
```

↓

File removed from staging area (index)

↓

Changes kept in working directory

↓

Ready for later staging or more changes

THIS IS IMPORTANT

`git restore` → Discard the working-directory changes

`git restore --staged` → Remove from staging but KEEP the changes

5 BEFORE vs AFTER

BEFORE (Staged)
```
$ git status
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   app.py
```
STAGED

AFTER git restore --staged
```
$ git status
Changes not staged for commit:
  (use "git add <file>..." to update what
    what will be committed)
        modified:   app.py
```
NOT STAGED (Changes kept)

AFTER git commit
```
$ git status
nothing to commit, working tree clean
```
COMMITTED

KEY IDEA

Use `git restore --staged` when you want to unstage a file but keep all your changes.

Safe and powerful!

VISUAL SUMMARY

Modify file → `git add` (stage it) → `git restore --staged` (unstage it) → Changes kept (not staged) → `git add` (again, when ready) → `git commit` (commit it)

COMMAND SUMMARY

`git restore <file>`	Discard changes in working directory (restore to last commit)
`git restore --staged <file>`	Unstage the file but keep the changes

REMEMBER

Unstaged is not lost.
It's just not ready to be committed yet.

CLOUD AND DATA UNIVERSE

Learn. Build. Succeed.

www.cloudanddatauniverse.com

GIT RESET — MOVE THE BRANCH/HEAD

Move the current branch pointer to a different commit.
Powerful and useful — use with care!

`git reset <commit> [--soft | --mixed |-hard]`
Moves the current branch (and HEAD) to the specified commit.

— Cloud And Data Universe —

1 THE SCENARIO

Your commit history looks like this:

- A Initial commit
- B Added login
- C Fixed bug
- D Added feature

You want `master` to move back to B.

(Maybe C and D were mistakes.)

2 THE RIGHT WAY: USE GIT RESET

Run:

```
$ git reset --mixed B
```

Now the history looks like this:

The branch pointer moved backward.
Commits C and D are no longer part of `master`.

3 QUICK CHECK

Run:

```
$ git log --oneline --graph
```

You'll see:

```
* b123456 (HEAD -> master) Added login
* a1b2c3d Initial commit
```

HEAD and `master` now point to B.

IMPORTANT

`git reset` moves the branch (and HEAD).
It does NOT affect other branches.

Commits C and D are not deleted — they're just no longer reachable from `master`.

You can still access them using:

```
$ git reflog
```

(Use with care!)

THREE IMPORTANT RESET MODES

RESET --SOFT

Moves the branch back but keeps changes staged.

Changes from C and D are:
- ✓ Kept in STAGING AREA
- ✓ Ready to commit again

Use when:
You want to rewrite history but keep everything staged.

RESET --MIXED (DEFAULT)

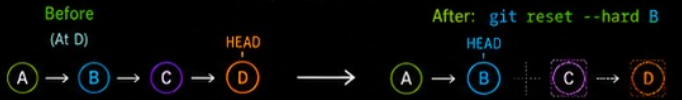
Moves the branch back and keeps changes in the working directory, unstaged.

Changes from C and D are:
- ✓ Kept in WORKING DIRECTORY
- ✓ Unstaged

Use when:
You want to unstage commits but keep your changes.

RESET --HARD

Moves the branch back and makes the working directory match that commit.



Changes from C and D are:
- ✗ Discarded from working directory
- ✗ Lost permanently

Use when:
You want a clean rollback to that commit.

⚠ DANGER: This will delete uncommitted changes. Use with extreme care!

WHAT HAPPENS UNDER THE HOOD?

- `git reset` moves the branch pointer (e.g., `master`).
- HEAD follows the branch.
- Commits after the target are just no longer part of that branch.

SUMMARY COMPARISON

Mode	Moves Branch?	Staging Area	Working Directory	Use Case
--soft	✓	Keep (staged)	Keep	Rewrite history, keep staged changes
--mixed (default)	✓	Clear (unstaged)	Keep	Unstage commits, keep changes
--hard	✓	Clear	Reset to target commit	Discard everything, clean rollback

KEY IDEA

`git reset` moves the branch/HEAD.
How your changes are treated depends on the mode you choose.

SOFT = Keep staged | MIXED = Keep changes | HARD = Discard changes

REMEMBER

Always double-check the target commit before running `git reset`.
When in doubt, don't use `--hard`!

CLOUD AND DATA UNIVERSE
Learn. Build. Succeed.
www.cloudanddatauniverse.com

GIT REVERT — UNDO A COMMIT SAFELY

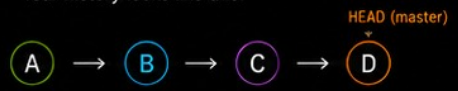
Undo the changes from a commit by creating a new commit that reverses it.
Safe, shareable, and history-friendly.

`git revert <commit>`
Creates a new commit that undoes the changes from <commit>.

— Cloud And Data Universe —

1 THE SCENARIO

Your history looks like this:



- A Initial commit
- B Added login
- C Added feature (BUG!)
- D More work

Commit C introduced a bug.

2 THE RIGHT WAY: USE GIT REVERT

Run:

```
$ git revert C
```

What happens?

- Git finds the changes introduced in C.
- It creates a NEW COMMIT (R) that reverses those changes.
- Your history stays intact!

Safe to use on shared branches.

3 CHECK THE RESULT

Run:

```
$ git log --oneline --graph
```

You'll see:

```
* d4e5f67 (HEAD -> master) Revert "Add feature (BUG!)"
* e3c4d56 More work
* a1b2c3d Add feature (BUG!)
* b1a2c3d Added login
* 9f8e7d6 Initial commit
```

A new commit (R) is added.
History is preserved.

WHY USE REVERT?

- Does not rewrite history. Perfect for shared/public branches.
- Creates a traceable record of the undo.
- Safer than reset when others have pulled your commits.

Use revert when in doubt!

REVERT vs RESET — KNOW THE DIFFERENCE

git reset

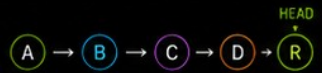
Moves the branch pointer backward.

- Rewrites history.
- Can discard commits.
- Dangerous on shared branches.

VS

git revert

Creates a new commit that reverses the change.



- Keeps history intact.
- Adds a new commit.
- Safe on shared branches.

WHAT DOES THE REVERT COMMIT DO?

Suppose commit C changed file app.py like this:

```
# Before C
return True

# Commit C (bug introduced)
return False
```

Running `git revert C` creates a new commit (R) that changes it back:

```
# After revert (R)
return True
```

Revert = apply the opposite changes.

WHEN NOT TO USE REVERT?

- If the commit you want to undo hasn't been pushed yet, reset may be simpler.
- If the changes are very complex, a revert might cause conflicts.
- Revert does not remove the commit — it only adds a new one.

Rule of thumb:
If others might have your commits, use revert. If not, reset is OK.

EXAMPLE WORKFLOW

Bug found in production from commit C.

1. Identify the bad commit (C caused the bug)
2. Run `git revert C`
3. Git creates a new commit (R) that undoes C
4. Push normally
   Everyone stays in sync!

COMMON SCENARIO

You pushed C, but it introduced a bug.

```
$ git revert C
```

→ Push the new revert commit → Teammates pull and are safe

COMMAND SUMMARY

`git log` — Find the commit (e.g., C)
→ `git revert C` — Create a revert commit
→ New commit (R) reverses C
→ `git push` — Share safely

REMEMBER

Revert undoes changes by adding a new commit.
Reset undoes changes by moving the branch.
One adds history, the other rewrites it.

CLOUD AND DATA UNIVERSE

Learn. Build. Succeed.

www.cloudanddatauniverse.com

GIT STASH — TEMPORARILY PUT WORK ASIDE

Save your uncommitted changes for later and get back to a clean workspace.
Switch branches, fix issues, do what you need — then come back and continue.

`>_ git stash`
Temporarily stores your changes and cleans your working directory.

— Cloud And Data Universe —

1 THE SCENARIO

You're working on feature/login and have unfinished changes:

app.py (modified)

```
def login():
    print("Login started")
    print("TODO: add validation")  # WIP
```

You haven't committed yet.

Suddenly:
"I need to switch to another branch."

2 THE PROBLEM

Git may prevent the switch:

```
error: Your local changes to the following files would be overwritten by checkout:
    app.py
Please commit your changes or stash them before you switch.
```

Or you don't want to carry those unfinished changes around.

Solution: Use `git stash`

3 THE RIGHT WAY: STASH IT

Run:

```
$ git stash
```

What happens?

- Git saves your changes on a stash stack.
- Your working directory becomes clean.
- You can switch branches freely.

4 CHECK THE RESULT

Run:

```
$ git status
```

You'll see:

```
On branch feature/login
nothing to commit, working tree clean
```

Your workspace is clean!

5 SWITCH AND WORK

Now you can:

- `$ git checkout master`
- Fix a production issue
- Create a hotfix
- Do whatever you need

No unfinished work blocking you.

6 GET YOUR WORK BACK

Later, return to your work.

Run:

```
$ git stash pop
```

What happens?

- Your changes are restored.
- The stash is removed from the stash list.

Back to where you left off!

HOW IT WORKS (VISUAL FLOW)

`git stash`

Uncommitted changes (dirty workspace) → Stash Stack (Git stores it) → Working directory becomes clean → Switch branch, do other work

Think of stash like a shelf:
You put your work on the shelf, come back later, and take it down.

USEFUL GIT STASH COMMANDS

Save your changes (stash)	`git stash`	Save tracked changes (unstaged and staged)
Save with a message	`git stash push -m "WIP login"`	Add a message to your stash
List all stashes	`git stash list`	See all saved stashes
Apply the latest stash (keep it in the list)	`git stash apply`	Restore changes, stash remains
Apply and remove the latest stash	`git stash pop`	Restore changes and remove from list
Apply a specific stash	`git stash apply stash@{2}`	Use stash index
Drop (delete) a stash	`git stash drop stash@{0}`	Delete specific stash
Clear all stashes	`git stash clear`	Remove all stashes

STASH STATUS EXAMPLE

Before stash (dirty)

```
$ git status
On branch feature/login
Changes not staged for commit:
    modified:   app.py
Untracked files:
    notes.txt
```

→ After `git stash`

```
$ git status
On branch feature/login
nothing to commit,
working tree clean
```

Clean workspace. Move freely.

WHAT GETS STASHED?

By default

- Modified tracked files
- Staged changes
- Untracked files

Ignored files are NOT stashed.

Include ignored files?

`git stash push -u`

Include untracked files

Include ignored files too?

`git stash push -a`

Include untracked + ignored

MULTIPLE STASHES (STACK)

```
$ git stash list
stash@{0}: On feature/login: WIP login flow
stash@{1}: On master: temp changes
stash@{2}: On dev: refactor code
```

stash@{0} is the most recent stash.

IMPORTANT NOTES

- Stash is local to your machine.
- Stash does not create a commit.
- Stash is perfect for short-term context switching.
- Don't forget old stashes — they can pile up!

Use stash wisely!

COMMAND SUMMARY

Dirty workspace → `git stash` Save changes → Clean workspace → Do other work → `git stash pop` Restore changes → Back to your work!

REMEMBER

`git stash` = Shelve your work.
`git stash pop` = Take it back.
Work clean. Switch fast. Stay productive.

CLOUD AND DATA UNIVERSE

Learn. Build. Succeed.
www.cloudanddatauniverse.com

GIT CLEAN — REMOVE UNTRACKED FILES

Remove files that Git is NOT tracking.
Clean your working directory.

```
>_ git clean [options]
```
Remove untracked files from your working directory.

— Cloud And Data Universe —

1 THE SCENARIO

Your project has:

```
project/
├── app.py      (tracked)
├── utils.py    (tracked)
├── data.csv    (tracked)
├── logs/       (untracked)
├── temp/       (untracked)
└── debug.log   (untracked)
```

Git knows about app.py, utils.py, data.csv but not logs/, temp/, debug.log

2 CHECK STATUS

Run:

```
$ git status
```

You might see:

```
On branch main
nothing to commit, working tree clean
Untracked files:
 (use "git add <file>..." to include
  in what will be committed)
        logs/
        temp/
        debug.log
```

These files are untracked.

3 PREVIEW: WHAT WILL BE REMOVED

Run (dry run – preview only):

```
$ git clean -n
```

You'll see what would be removed:

```
Would remove the following:
  logs/
  temp/
  debug.log
```

Nothing is deleted with -n. Safe to run.

4 PREVIEW INCLUDING UNTRACKED DIRECTORIES

By default, -n doesn't show untracked directories.
Use -nd to see everything:

```
$ git clean -nd
```

You'll see:

```
Would remove the following:
  logs/
  temp/
  debug.log
```

-nd = include directories in the preview.

5 CLEAN: REMOVE THEM

When you're sure, run:

```
$ git clean -d
```

(-d = remove untracked directories)

It removes the untracked files and folders.

```
logs/
temp/
debug.log
```
are deleted.

Your working directory is now clean.

6 CHECK AGAIN

Run:

```
$ git status
```

You'll see:

```
On branch main
nothing to commit,
working tree clean
```

No untracked files left!

IMPORTANT WARNINGS

- ⊗ git clean PERMANENTLY deletes files.
- ⊗ There is no undo.
- ⊗ These files are not in Git history.
- ⊗ Always preview with -n first.
- ⊗ Don't run git clean -fd unless you are 100% sure.

Be careful. Be safe. Preview first!

COMMON OPTIONS

OPTION	DESCRIPTION
-n	Dry run. Show what would be removed.
-d	Remove untracked directories in addition to files.
-f	Force. Required unless clean.requireForce is false.
-i	Interactive mode. Ask before removing each item.
-x	Also remove ignored files (not recommended normally).
-X	Remove only ignored files, keep other untracked files.

MORE EXAMPLES

$ git clean -n	Preview untracked files (not directories)
$ git clean -nd	Preview files and directories
$ git clean -fd	Remove untracked files and directories
$ git clean -fdx	Remove untracked and ignored files/dirs
$ git clean -i	Interactive mode
$ git clean -nX	Preview ignored files only

INTERACTIVE MODE (-i)

Run:

```
$ git clean -i
```

You'll be prompted:

```
1: clean       Remove files
2: filter      Filter by pattern
3: select      Select by numbers
4: ask each    Ask before removing
5: quit        Quit
Choose an option:
```

KEY IDEA

git clean removes files that Git doesn't know about.
Use it to keep your workspace tidy.

BEFORE vs AFTER

COMMAND SUMMARY

REMEMBER

Preview first (-n).
Use -d to include directories.
Use with care!

CLOUD AND DATA UNIVERSE

Learn. Build. Succeed.
www.cloudanddatauniverse.com

GIT TAG — MARK AN IMPORTANT COMMIT

Tags let you mark specific points in history as important — like snapshots or releases.

`>_ git tag [tag-name] [commit]`

Mark a commit with a tag. Easy to reference later.

CDU — Cloud And Data Universe —

1 THE SCENARIO

Your project reaches a stable milestone. History looks like this:

- A Initial commit
- B Added login
- C Stable version (v1.0)
- D New feature
- E Bug fixes

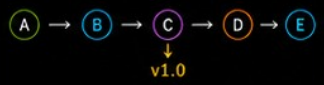
You want to mark C as the v1.0 release.

2 CREATE A LIGHTWEIGHT TAG

Run:

```
$ git tag v1.0 C
```

Now C is tagged as v1.0.



✓ Tag created!

3 SEE TAGS

Run:

```
$ git tag
```

You'll see:

```
v1.0
```

✓ List all tags.

4 CREATE AN ANNOTATED TAG

Better for releases — includes message, author, date, etc.

Run:

```
$ git tag -a v1.0.1 C -m
"Release v1.0.1 - Stable version"
```

Now you created an annotated tag.

✓ Annotated tags store more info.

5 SEE TAG DETAILS

Run:

```
$ git show v1.0.1
```

You'll see:

```
tag v1.0.1
Tagger: You <you@example.com>
Date:   Sat May 24 12:00:00 2025

 Release v1.0.1 - Stable version

commit c3d4e5f (tag: v1.0.1, tag: v1.0)
Author: You <you@example.com>
Date:   Sat May 24 11:45:00 2025
    Stable version (v1.0)
```

✓ See tag info and the commit it points to.

6 DELETE A TAG

Delete a local tag:

```
$ git tag -d v1.0
```

You'll see:

```
Deleted tag 'v1.0' (was c3d4e5f)
```

Delete a remote tag:

```
$ git push origin :refs/tags/v1.0
```

⚠ Deletes the tag on the remote (use with care!).

LIGHTWEIGHT vs ANNOTATED TAGS

LIGHTWEIGHT

- ✓ Simple pointer to a commit
- ✓ No extra metadata
- ✓ Good for personal use
- ✓ Created with: `git tag <name>`

ANNOTATED

- ✓ Full metadata (message, author, date)
- ✓ Stored as objects in Git
- ✓ Best for releases & collaboration
- ✓ Created with: `git tag -a <name> -m "msg"`

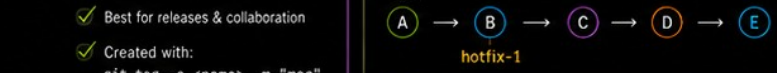
TAGGING A COMMIT DIRECTLY

Tag any commit using its ID (short or full).

Run:

```
$ git tag hotfix-1 b1a2c3d
```

Now:



✓ Tag points exactly to that commit.

LIST TAGS WITH MORE INFO

Run:

```
$ git tag -n
```

You'll see:

```
v1.0      Stable version (v1.0)
v1.0.1    Release v1.0.1 - Stable version
```

✓ `-n` shows the message (if annotated).

SORT TAGS BY DATE

Run:

```
$ git tag --sort=-creatordate
```

You'll see (newest first):

```
v1.0.1
v1.0
hotfix-1
```

✓ Sort tags by creation date.

WHERE DO TAGS LIVE?

Push tags explicitly (they are not pushed by default).

```
$ git push origin --tags
```

USE CASES

Release Versions
Mark official releases (v1.0, v2.0, etc.)

Milestones
Mark project checkpoints (alpha, beta, RC)

Hotfixes
Tag emergency fixes (hotfix-1, fix-urgent)

Backup Points
Easy to reference stable commits

COMMAND SUMMARY

`$ git tag <name>`	Create lightweight tag
`$ git tag -a <name> -m "msg"`	Create annotated tag
`$ git tag`	List tags
`$ git tag -n`	List tags with messages
`$ git show <tag>`	Show tag details
`$ git tag -d <name>`	Delete local tag
`$ git push origin --tags`	Push all tags to remote

REMEMBER

Tags don't move. They always point to the same commit.

Use annotated tags for releases.

Push tags when sharing with others.

CLOUD AND DATA UNIVERSE

Learn. Build. Succeed.

www.cloudanddatauniverse.com

DELETE A BRANCH

Clean up branches after the work is done.

`git branch -d <branch-name>`
Delete a merged branch safely.

CDU — Cloud And Data Universe —

1 LIST BRANCHES

Run:

```
$ git branch
```

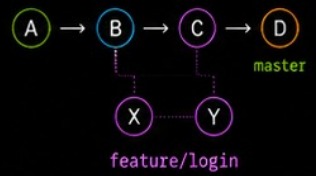
You'll see:

```
* master
  feature/login
  feature/report
  bugfix/typo
```

✓ * means your current branch.

2 OUR SCENARIO

Suppose you worked on feature/login and merged it.



The feature is finished and merged into master.

3 DELETE MERGED BRANCH

Run:

```
$ git branch -d feature/login
```

You'll see:

```
Deleted branch feature/login
(was Y7f3a12).
```

✓ Git removes the branch pointer, not the commits.

4 CHECK AGAIN

Run:

```
$ git branch
```

You'll see:

```
* master
  feature/report
  bugfix/typo
```

✓ feature/login is gone. History is still intact.

5 VERIFY HISTORY IS INTACT

Run:

```
$ git log --oneline --graph --all
```

You'll still see all commits:

```
*   d4e5f67 (HEAD -> master) Merge
|\          feature/login
| * y7f3a12 Add login validation
| * x9b1cde Add login UI
* | c3a2b10 Add reporting
|/
* b12c3d4 Initial commit
```

✓ Commits from feature/login are part of master now.

6 CANNOT DELETE UNMERGED BRANCH

Suppose feature/report is NOT merged.

Run:

```
$ git branch -d feature/report
```

You'll see:

```
error: The branch 'feature/report'
  is not fully merged.
  If you are sure you want to delete it,
  run 'git branch -D feature/report'.
```

⚠ Git prevents accidental data loss. Use -d first (safe).

FORCE DELETE (DANGEROUS VERSION)

If you are absolutely sure:

Run:

```
$ git branch -D feature/report
```

You'll see:

```
Deleted branch feature/report (was ab12cd3).
```

⚠ -D forces deletion. Use with extreme care!

HOW -d vs -D WORKS

OPTION	-d (safe)	-D (force)
When to use	Branch is fully merged and you're cleaning up	Branch is NOT merged and you really want to delete
What it checks	Verifies branch is merged into current branch	Does NOT check merge status
Risk level	Low	High
Bottom line	✓ Prefer -d	⚠ Use -D only when you know what you're doing

COMMON WORKFLOW

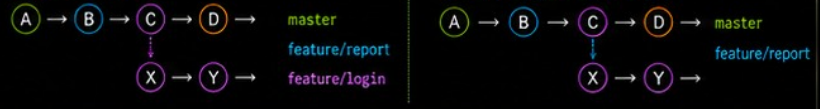
Create branch → Do work & commit → Merge into master → Verify it works → Delete the branch

Keep your branch list clean.
Easy to navigate, easier to collaborate.

TIPS & BEST PRACTICES

- ✓ Delete branches after the work is merged and verified.
- ✓ Keep master/main clean and stable.
- ✓ Avoid force deleting unless absolutely necessary.
- ✓ Regular cleanup reduces confusion.

VISUAL: BEFORE vs AFTER



REMEMBER

Deleting a branch does NOT delete commits. It only removes the branch pointer (name).

CLOUD AND DATA UNIVERSE

Learn. Build. Succeed.

www.cloudanddatauniverse.com

RENAME A BRANCH

Change a branch name without losing any history.

`git branch -m <new-name>`
Rename a branch.

— Cloud And Data Universe —

1 THE SCENARIO

Suppose you have a branch named:

`feature/login`

but you want to rename it to:

`user-auth`

Renaming a branch only changes its name.
The commits and history stay the same.

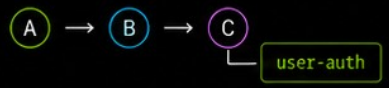
2 RENAME THE CURRENT BRANCH

If you are ON the branch you want to rename:

Run:

```
git branch -m user-auth
```

Now:



The branch is renamed in-place.
You stay on the same branch.

3 RENAME ANOTHER BRANCH

If you are NOT on the branch you want to rename:

Run:

```
git branch -m feature/login user-auth
```

This renames the branch from `feature/login` to `user-auth`

Works even if you are on a different branch (like `main` or `master`).

4 CHECK THE RESULT

Run:

```
git branch
```

You'll see:

```
* main
  user-auth
  bugfix/typo
  hotfix/ui
```

The new name appears.
The old name is gone.

IMPORTANT

Renaming a branch does NOT change any commits. It only changes the branch pointer (name).

BEFORE vs AFTER

The commits (A, B, C) are exactly the same.
Only the branch name changed.

QUICK REFERENCE

What you want to do	Command	Example
Rename current branch	`git branch -m <new-name>`	`git branch -m user-auth`
Rename another branch	`git branch -m <old-name> <new-name>`	`git branch -m feature/login user-auth`
Check branches	`git branch`	`git branch`

THINGS TO KNOW

- Renaming a local branch is safe and simple.
- If the branch is already pushed to a remote, you'll need to update the remote too:

  After renaming:
  ```
  git push origin -u user-auth
  git push origin --delete feature/login
  ```

- Other team members must also update their local references.

COMMON EXAMPLE

Rename a branch from:

`feature/login` → `user-auth`

COMMAND SUMMARY

REMEMBER

Rename to make your branch names clearer and more consistent.
Clean names, `clean` repo!

CLOUD AND DATA UNIVERSE

Learn. Build. Succeed.

www.cloudanddatauniverse.com

GIT REBASE

Take the changes from your commits and reapply them on top of another base to create a clean, linear history.

`git rebase <base-branch>`
Reapply your commits on top of <base-branch>.

CDU — Cloud And Data Universe —

PRACTICAL DEMO: PUT FEATURE WORK ON TOP OF THE LATEST MASTER

1 INITIAL COMMIT

Create the first commit on master.


```
echo "Version 1" > app.txt
git add .
git commit -m "Initial version"
```

We have one commit (A). Both master and future branches will start here.

2 CREATE FEATURE BRANCH

Create and switch to feature branch.


```
git switch -c feature
```

Now master and feature point to the same commit (A).

3 MAKE A FEATURE COMMIT

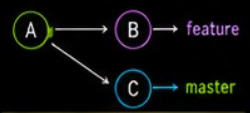
Do some work on feature and commit.


```
echo "Feature 1" >> app.txt
git add .
git commit -m "Add feature 1"
```

Feature branch is now ahead with commit B.

4 SWITCH BACK TO MASTER

Switch to master and do some work.

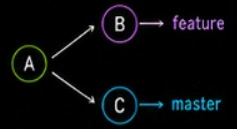



```
git switch master
echo "Main update" >> app.txt
git add .
git commit -m "Add main update"
```

Now both branches have diverged.

5 VISUALIZE DIVERGENCE

See the diverged history.

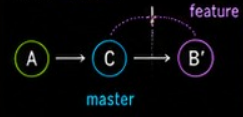



```
git log --oneline --graph --all
```

We have:
A → B (feature)
A → C (master)

6 PERFORM REBASE

Rebase feature onto the latest master.




```
git switch feature
git rebase master
```

Git takes commit B, removes it temporarily, then reapplies its changes on top of C.

7 HISTORY AFTER REBASE

Feature is now ahead of master in a linear way.


```
git log --oneline --graph --all
```

A new commit B' is created (same changes as B, new commit ID). Clean & linear!

8 RESULT

Your feature work is now on top of the latest master.

Ready to merge with a fast-forward (no merge commit)!

This makes history easier to read and understand.

BEFORE vs AFTER REBASE

Rebase rewrites history by creating new commits (B'). Commit IDs change, but your changes stay the same.

REBASE vs MERGE

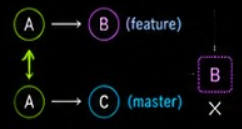
VS

History has a merge commit (M).
Preseves the timeline of both branches.

No merge commit.
Linear history is easier to read.

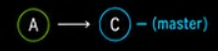
WHAT ACTUALLY HAPPENS DURING REBASE

1 Remove your commits



Git temporarily removes B from the branch.

2 Move base to new branch



The base is now the target branch (master).

3 Reapply your commits

Git applies the changes from B on top of C as a new commit (B').

WHEN TO USE REBASE

- You're working on a local feature branch.
- You want a clean, linear project history.
- You want to keep your commits organized before merging.

WHEN NOT TO USE REBASE

- If the branch is already pushed/shared with others.
- If others are working on the same branch.
- It rewrites history and can cause conflicts.

INTERACTIVE REBASE (NEXT LEVEL)

Open an editor to rewrite, reorder, squash, edit or drop commits.

```
git rebase -i HEAD~3
```

Common actions in editor:

```
pick   - keep commit
reword - change commit message
edit   - modify commit
squash - combine with previous
drop   - remove commit
```

QUICK COMMAND REFERENCE

Command	Meaning
`git rebase <branch>`	Rebase current branch on top of <branch>
`git rebase --continue`	Continue after resolving conflicts
`git rebase --abort`	Abort rebase and go back
`git rebase -i HEAD~N`	Interactive rebase (last N commits)
`git log --oneline --graph --all`	View history (graphical)

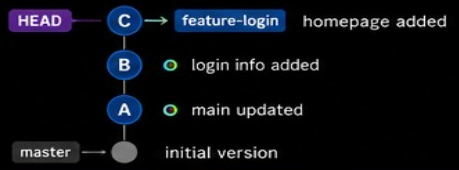
GIT REBASE

Rewrite history. Keep it clean.
Interactive rebase lets you
PICK, DROP, SQUASH, EDIT
or ABORT when needed.

```
>_ git rebase -i HEAD~3
```
Edit the todo list, save & close.
Git will apply your instructions.

CDU
— Cloud And Data Universe —

EXAMPLE HISTORY (BEFORE REBASE)



HOW TO START

```
git rebase -i HEAD~3
```
Start an interactive rebase for the last 3 commits.

YOU WILL SEE THE TODO LIST

```
pick  a1b2c3  main updated
pick  d4e5f6  login info added
pick  g7h8i9  homepage added
```
Change the commands, save & close.
Git will apply your instructions.

COMMANDS SUMMARY

pick	keep the commit (default)
drop	remove the commit
squash	combine with previous commit
edit	stop at this commit to modify it

AFTER REBASE

Git rewrites the history and moves your branch to the new commits.

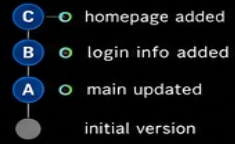
① PICK (KEEP)

Default action. Keeps the commit as it is.

TODO LIST (example)

```
pick  a1b2c3  main updated
pick  d4e5f6  login info added
pick  g7h8i9  homepage added
```

RESULT
History remains the same.



Use pick for commits you want to keep unchanged.

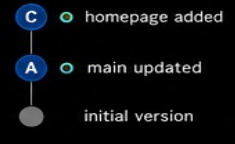
② DROP (REMOVE)

Drops (removes) the commit from history.

TODO LIST (example)

```
pick  a1b2c3  main updated
drop  d4e5f6  login info added
pick  g7h8i9  homepage added
```

RESULT
The dropped commit is gone.



Use drop to remove a commit completely.

③ SQUASH (COMBINE)

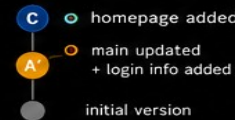
Combines this commit with the one above it.

TODO LIST (example)

```
pick    a1b2c3  main updated
squash  d4e5f6  login info added
pick    g7h8i9  homepage added
```

After saving, Git opens the editor to combine the commit messages.

RESULT
Commit B is combined into A.



Use squash to keep changes but merge commit messages.

④ EDIT (MODIFY)

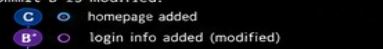
Stops at this commit so you can modify files, then **amend** the commit.

TODO LIST (example)

```
pick  a1b2c3  main updated
edit  d4e5f6  login info added
pick  g7h8i9  homepage added
```

WHAT HAPPENS
1. Git stops at commit B.
2. You modify files as needed.
3. `git add .`
4. `git commit --amend` (to update the commit)
5. `git rebase --continue` (to continue rebase)

RESULT
Commit B is modified.



Use edit to change the content of a commit.

⑤ ABORT (CANCEL REBASE)

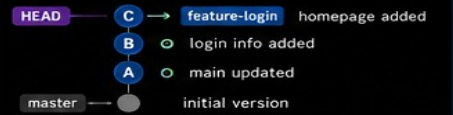
Stops the rebase and returns to the state before it started.

SCENARIO
- You started rebase
- Changed the todo list
- Git is in the middle of rebasing...

RUN THIS COMMAND

```
git rebase --abort
```

RESULT
Everything is back to normal.



Use --abort if something goes wrong or you change your mind.

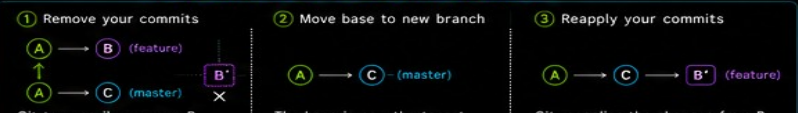
REBASE CONTROL COMMANDS (during rebase)

`git rebase --continue`	Continue after resolving or amending
`git rebase --skip`	Skip the current commit
`git rebase --abort`	Cancel the rebase and go back

BEFORE vs AFTER REBASE

Rebase rewrites history to create a clean, linear commit line.

WHAT ACTUALLY HAPPENS DURING REBASE



IMPORTANT NOTES

Rebase rewrites history.
- Never rebase commits that are already shared.
- Use rebase to keep history clean before pushing.
- When in doubt: `git rebase --abort` is your safety net.

WHEN TO USE REBASE

- You're working on a local feature branch.
- You want a clean, linear project history.
- You want to keep your commits organized before merging.

WHEN NOT TO USE REBASE

- If the branch is already pushed/shared.
- If others are working on the same branch.
- It rewrites history and can cause conflicts.

INTERACTIVE REBASE CHEAT SHEET

```
git rebase -i HEAD~N

pick   = keep commit
reword = change commit message
         (use r instead of pick)
edit   = modify commit
squash = combine with previous
drop   = remove commit
```

QUICK COMMAND REFERENCE

Command	Meaning
`git rebase <branch>`	Rebase current branch on top of <branch>
`git rebase --continue`	Continue after resolving conflicts
`git rebase --abort`	Abort rebase and go back
`git rebase -i HEAD~N`	Interactive rebase (last N commits)
`git log --oneline --graph --all`	View history (graphical)

GIT CHERRY-PICK

Apply the changes from one or more specific commits from another branch to your current branch.

`git cherry-pick <commit-id>`

Takes the change introduced by the given commit and creates a new commit on your current branch.

CDU — Cloud And Data Universe —

WHEN TO USE CHERRY-PICK

- You want a specific fix/feature from another branch.
- You don't want the entire branch history.
- You want to bring a commit to multiple branches.
- You need a hotfix from one branch to another.

CHERRY-PICK vs REBASE

Cherry-pick	Rebase
• Picks specific commit(s).	• Replays a series of commits.
• Creates new commit(s).	• Creates new commit(s).
• History may diverge.	• Keeps history linear.

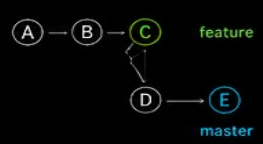
BASIC CHERRY-PICK WORKFLOW

1. **Find the commit** — Locate the commit ID you want to pick.

```
$ git log --oneline
eeb4221  main updated feature
21f1e23  login added
4c18ea8  homepage added
0a1b2c3  initial commit
```

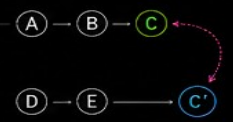
2. **Switch to target branch** — Go to the branch where you want to apply it.

```
$ git switch master
```

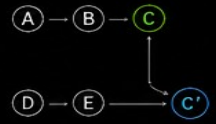


3. **Cherry-pick the commit** — Apply the commit to current branch.

```
$ git cherry-pick eeb4221
```



4. **New commit created** — A new commit (C') with same changes, different ID.

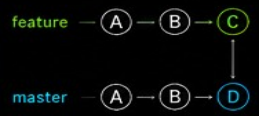


CHERRY-PICK COMMANDS

`git cherry-pick <commit>`	Pick a specific commit.
`git cherry-pick <c1> <c2> ...`	Pick multiple commits (in the given order).
`git cherry-pick <c1>^..<c2>`	Pick a range of commits (from c1 exclusive to c2 inclusive).
`git cherry-pick --continue`	Continue after resolving conflicts.
`git cherry-pick --skip`	Skip the current commit.
`git cherry-pick --abort`	Abort and return to the state before cherry-pick.

DEMO 1: BASIC CHERRY-PICK

1. **Initial State**



A = initial
B = main updated
C = login added (only in feature)
D = bug fix (only in master)

2. **Pick commit C to master**

```
$ git switch master
$ git cherry-pick C
```

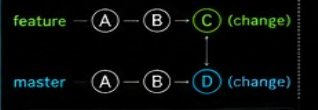
New commit C' is created.

3. **Result** — master now has the change from commit C.

Commit C and C' have the same changes, but different commit IDs.

DEMO 2: CHERRY-PICK WITH CONFLICT

1. **Diverged Changes** — Both branches modify the same line differently.



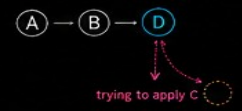
file.txt on both branches

```
Original line
C: changed by FEATURE
D: changed by MASTER
```

2. **Cherry-pick C to master**

```
$ git cherry-pick C
```

CONFLICT occurs!



3. **Resolve Conflict** — Git marks conflict in file. Open file in VS Code.

```
<<<<<<< HEAD (master)
changed by MASTER
=======
changed by FEATURE
>>>>>>> C (feature)
```

Edit the file and keep your desired content.

4. **Continue**

```
$ git add file.txt
$ git cherry-pick --continue
```

New commit C' is created.

★ **Tip:** If you don't want to apply the commit, run: `git cherry-pick --abort`

PICK MULTIPLE COMMITS

Pick commits one by one (creates new commits).

```
$ git cherry-pick c1 c2 c3
```

Pick a range (c1 is exclusive, c3 is inclusive).

```
$ git cherry-pick c1^..c3
```

Commits are applied in the order you list them.

WHAT HAPPENS UNDER THE HOOD?

- Git takes the changes introduced by the commit.
- Applies them on top of your current branch.
- If clean → creates a new commit.
- If conflicts → stops for you to resolve.
- You continue, skip, or abort.

BEFORE vs AFTER (EXAMPLE)

C (original) ≠ C' (new commit with same changes)

IMPORTANT NOTES

- ⚠ Cherry-pick creates new commits (new IDs).
- ⚠ It's safe for public branches.
- ⚠ Perfect for bug fixes and hotfixes.
- ⚠ Conflicts may happen—resolve carefully.
- ⚠ Always verify with: `git log --oneline --graph --all`

QUICK COMMAND REFERENCE

Command	Meaning
`git cherry-pick <commit>`	Pick a single commit
`git cherry-pick <c1> <c2> ...`	Pick multiple commits
`git cherry-pick <c1^..c2>`	Pick a range of commits
`git cherry-pick --continue`	Continue after resolving conflicts
`git cherry-pick --skip`	Skip the current commit
`git cherry-pick --abort`	Cancel cherry-pick and rollback

Golden Rule: Cherry-pick what you need, not everything.

Smaller, intentional, and easy to manage.

GITHUB – INTRODUCTION

GitHub is a website where we can **store Git repositories online.**

If Git manages your code and its history on your computer, GitHub gives you a place to store, share, and collaborate on that Git repository online.

KEY IDEA

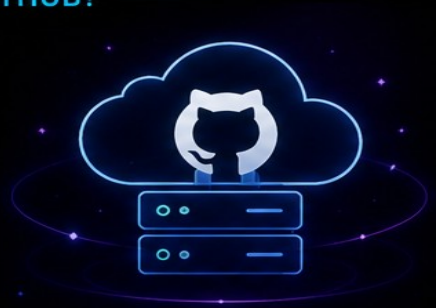
Git works locally on your computer. GitHub brings your repositories online and enables collaboration.

CDU — Cloud And Data Universe —

WHAT IS GITHUB?

GitHub is an online platform built around Git.

- Store repositories online
- Share code and collaborate
- Review code and track changes
- Manage projects and tasks



GIT VS GITHUB – AT A GLANCE

YOUR COMPUTER
(Local Repository)

Git manages everything locally on your machine.

GITHUB
(Remote Repository)

GitHub stores your repositories and makes them accessible online.

GIT VS GITHUB

This is the first thing you should understand:

GIT

A tool for **version control.**

It helps you manage:

- files
- commits
- branches
- history
- merges
- rebase

VS

GITHUB

An online platform built around Git.

It helps you:

- store repositories online
- share code
- collaborate with others
- review code
- manage projects

So: Git → Version Control | GitHub → Online collaboration platform

WHY DO WE NEED GITHUB?

BACKUP
Your code is safe in the cloud.

COLLABORATION
Work with others from anywhere.

ACCESS ANYWHERE
Access your code from any device.

OPEN SOURCE
Share your projects with the world.

SECURITY
Secure and reliable hosting.

THE BIG PICTURE

1. YOU WORK LOCALLY
Make changes, commit, and manage history using Git.

→

2. PUSH TO GITHUB
Send your commits to GitHub and store them online.

→

3. OTHERS COLLABORATE
Team members fetch, pull, review, and contribute.

→

4. IMPROVE TOGETHER
Review changes, merge, and make the project better.

★ **Remember:** Git is the engine. | GitHub is the platform. | Together, they make powerful teamwork possible!

GIT REMOTE

Manage connections between your local repository and remote repositories.

COMMON GIT REMOTE COMMANDS

1 git remote

List the names of remote repositories.

```
$ git remote
origin
```

Shows the remote names connected to your local repository.

2 git remote -v

List remotes along with their URLs.

```
$ git remote -v
origin   https://github.com/user/project.git (fetch)
origin   https://github.com/user/project.git (push)
```

Shows the URLs for both fetch (download) and push (upload).

3 git remote add <name> <url>

Add a new remote repository.

```
$ git remote add origin \
  https://github.com/user/project.git
```

Connects your local repository to a remote using the name you give (e.g., origin).

4 git remote remove <name>

Remove a remote repository.

```
$ git remote remove origin
```

Removes the remote connection with the specified name.

5 git remote rename <old> <new>

Rename an existing remote.

```
$ git remote rename origin github
```

Changes the remote name (e.g., origin → github).

HOW IT WORKS

`git remote` only manages the connection (address).
It does NOT upload or download your code.
Use `git push`, `git pull`, or `git fetch` for that.

COMMON USES

- View connected remote repositories
- See where your remotes are located (URLs)
- Add a new remote repository
- Remove a remote repository connection
- Rename remotes for easy identification
- Work with multiple remotes

REMOTE WORKFLOW EXAMPLE

1. ADD REMOTE
`git remote add origin <url>`
Connect your local repo to a remote.

→

2. CHECK REMOTE
`git remote -v`
Verify the remote connection.

→

3. PUSH CHANGES
`git push -u origin main`
Send your commits to the remote.

→

4. COLLABORATE
Others fetch, pull, and contribute.

→

5. STAY UPDATED
`git fetch / git pull`
Get updates from the remote.

IMPORTANT NOTES

- origin is just a convention, not a requirement. You can use any name you like.
- You can have multiple remotes.

```
$ git remote add upstream https://...
$ git remote add origin   https://...
```

- Removing a remote does not delete the remote repository on GitHub.

REMEMBER:

`git remote` = Manage remote connections

It connects your local repository to remote repositories.

It does not transfer any code. Use `push`, `pull`, or `fetch` for that.

Essential for collaborating and working with remote repositories.

GIT PUSH

Send your local commits to a remote repository.

git push updates the remote repository with the commits from your current branch.

KEY IDEA

You work locally, commit your changes, then push to share those changes with others by updating the remote repository.

WHEN DO WE USE GIT PUSH?

- Upload your local commits to a remote repository.
- Share your work with teammates.
- Backup your work on GitHub.
- Keep the remote repository up to date.

BASIC SYNTAX

```
$ git push <remote> <branch>
```

- `<remote>` → remote name (e.g., origin)
- `<branch>` → branch name you want to push (e.g., master, main)

EXAMPLES

`$ git push origin master`	Push master branch to origin remote
`$ git push origin main`	Push main branch to origin remote
`$ git push upstream develop`	Push develop branch to upstream remote

PUSH WITH UPSTREAM (TRACKING)

The `-u` (or `--set-upstream`) option sets the upstream (tracking) relationship. After that, you can use just `git push`.

FIRST TIME PUSH OF A NEW BRANCH

```
$ git push -u origin feature/login
```

This does two things:

1. Pushes your branch to the remote.
2. Sets the remote branch (`origin/feature/login`) as upstream.

AFTER THAT

```
$ git push
```

Git knows where to push your branch.

CHECK UPSTREAM

```
$ git branch -vv
```

Shows your branches and their upstream tracking info.

MULTIPLE REMOTES – HOW TO PUSH

You can have multiple remotes in one repository.

origin	upstream	backup
https://github.com/your-username/project.git	https://github.com/company/project.git	https://gitlab.com/your-username/project.git

Push to the specific remote you want:

`$ git push origin master`	Push to origin
`$ git push upstream master`	Push to upstream
`$ git push backup master`	Push to backup

Git does NOT push to all remotes automatically.

WHAT HAPPENS WHEN YOU PUSH?

1. **LOCAL COMMITS** — Commits exist only on your computer.
2. **PUSH** — Git sends your commits to the remote.
3. **REMOTE UPDATED** — The remote repository is updated.
4. **OTHERS CAN SEE** — Your teammates can fetch/pull the changes.

COMMON OUTPUT (SUCCESS)

```
$ git push origin master
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compresion using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 512 bytes | 512.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To https://github.com/your-username/project.git
   abc1234..def5678  master -> master
```

Your branch is now up to date on the remote.

IMPORTANT NOTES

- You can only push what you have committed.
- You must have permission to push to the remote repository.
- If someone else pushed changes, you may need to pull first to avoid conflicts.
- For the first push of a new branch, use `-u` to set upstream.

ERROR EXAMPLE

```
$ git push origin master
! [rejected]          master -> master (fetch first)
error: failed to push some refs to 'origin'
hint: Updates were rejected because the remote contains work
that you do not have locally. This is usually caused by another
repository pushing to the same ref.
hint: See the 'Note about fast-forwards' in 'git push --help'.
```

SOLUTION

```
$ git pull origin master      # Get remote changes
! Resolve conflicts if any
$ git push origin master      # Push again
```

QUICK REMEMBER

`git push <remote> <branch>`

- Send your commits
- Where (remote)
- What (branch)

Push = Upload
Pull = Download

If it's not pushed, it's not backed up.

Commit locally.
Push confidently.
Collaborate effectively.

Push your code.
Share your work.
Build together!

GIT FETCH

Download changes from a remote repository **without** changing your current branch.

KEY IDEA

`git fetch` gets the latest changes, commits, and branches from the remote and updates your remote-tracking branches (`origin/*`). It does NOT change your working files or your current branch.

CDU — Cloud And Data Universe —

WHAT IS GIT FETCH?

`git fetch` downloads commits, files and references from a remote repository to your local repository.

It updates your remote-tracking branches (like `origin/main`) but does NOT change your working files or your current branch.

It lets you see what others have changed without actually applying those changes to your branch.

HOW GIT FETCH WORKS

REMOTE (GitHub)

origin

FETCH (Download)

`>_ git fetch origin`

Downloads new data from the remote.

LOCAL REPOSITORY

Updates remote-tracking branches (`origin/*`)

IMPORTANT

Your current branch (e.g., `main`) and your working files remain UNCHANGED after `git fetch`.

USAGE

```
$ git fetch origin
```

- `origin` is the name of the remote.
- This updates all remote-tracking branches under `origin/*`.
- It does NOT merge changes into your current branch.

WHEN TO USE?

- To see what changes others have made.
- To inspect new commits before merging.
- To keep your local information up to date without affecting your work.

VISUAL EXAMPLE

Fetch brings the changes to your local repository (`origin/*`) for you to review. It does not apply them to your current branch.

RELATED COMMANDS

`git pull`	= fetch + merge (integrate changes)
`git merge origin/main`	= integrate fetched changes into your current branch
`git log origin/main`	= see commits on remote branch
`git diff main..origin/main`	= see differences between your branch and remote

FETCH VS PULL

`git fetch`

- Downloads changes and updates `origin/*`
- Does NOT change
- Safe to run anytime

VS

`git pull`

- Fetches changes and merges (integrates) into your current branch
- May change files and create

WHY DO WE NEED GIT FETCH?

SAFETY — Check changes before integrating. No risk to your work.

COLLABORATION — Stay informed about what others have pushed.

INSPECTION — Review commits, differences, and history before merging.

INFORMED DECISIONS — Helps you decide when and how to integrate changes.

UP TO DATE — Keeps your remote information current without interruptions.

EXAMPLE WORKFLOW

1. **WORK LOCALLY** — Make changes, commit, and work on your branch.
2. **FETCH** — `git fetch origin` — Get latest changes from remote.
3. **INSPECT** — Check logs, diffs, and what others have changed.
4. **MERGE (WHEN READY)** — `git merge origin/main` — Bring changes into your branch.
5. **COLLABORATE** — Push your work and continue building together!

REMEMBER: | GIT IS THE ENGINE. | GITHUB IS THE PLATFORM. | TOGETHER, THEY MAKE POWERFUL TEAMWORK POSSIBLE!

GIT PULL

Fetch changes from a remote repository and integrate them into your current branch.

CDU — Cloud And Data Universe —

KEY IDEA

`git pull = git fetch + integrate`

It gets the latest changes from the remote and merges (or rebases) them into your current branch.

WHAT IS GIT PULL?

`git pull` downloads new commits, files, and references from a remote repository and integrates them into your current branch.

- Fetches the latest changes from the remote (like `origin/main`).
- Integrates those changes into your current branch (merge by default, or rebase if configured).
- Updates your working files if needed.

In short: It keeps your local branch up to date with the remote.

HOW GIT PULL WORKS

1. FETCH	2. INTEGRATE	3. UPDATED
Downloads new data from the remote.	Merges (or rebases) the changes into your current branch.	Your current branch is now up to date.
origin	(merge / rebase)	your branch

```
$ git pull origin main
```
`remote name` `branch name`

USAGE

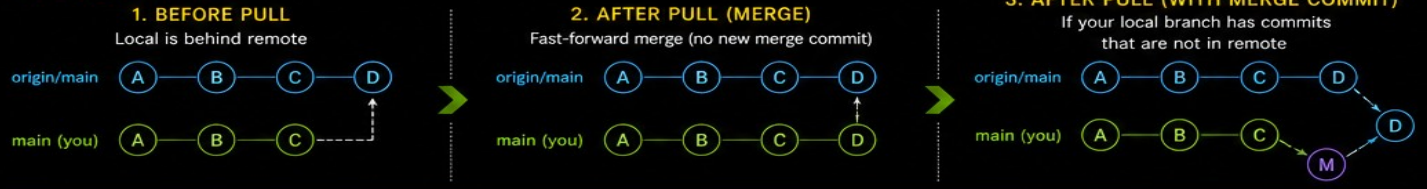
```
$ git pull origin main
```

- `origin` → name of the remote
- `main` → branch to pull
- If your branch has an upstream branch set, you can simply use: `git pull`

WHEN TO USE?

- To get the latest changes from the remote.
- To update your local branch with others' work.
- Before you start working to ensure your branch is up to date.

VISUAL EXAMPLE



`git pull` first fetches the latest changes (updates `origin/*`), then integrates them into your current branch.

RELATED COMMANDS

`git fetch origin`	= fetch updates without integrating
`git pull`	= fetch + integrate
`git pull --rebase`	= fetch + rebase (linear history)
`git log origin/main`	= see remote branch commits
`git status`	= check your working tree status
`git branch -vv`	= shows upstream tracking info

PULL VS FETCH

`git fetch origin`	VS	`git pull origin main`
• Downloads changes only.		• Downloads and integrates.
• Does NOT merge.		• Updates your current branch.
• Your files do not change.		• Your files may change.
• Safe to explore changes first.		• Keeps your work in sync.

WHAT IF THERE ARE CONFLICTS?

If changes conflict, Git will stop and ask you to resolve.

1. Git stops — Shows conflict in files.
2. Resolve — Edit files and keep the changes you want.
3. Stage — `git add <file>` or `git add .`
4. Commit — `git commit` to complete.

EXAMPLE WORKFLOW

Work on your branch → `git pull origin main` → Resolve conflicts (if any) → Now your branch is up to date!

REMEMBER: `Pull = Get latest + Integrate` | Keep pulling often to stay in sync. | Up-to-date code = Better collaboration!

GIT UPSTREAM

Upstream is the remote-tracking branch that your local branch is configured to follow.

KEY IDEA

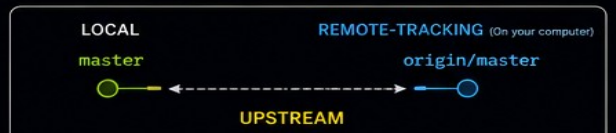
Upstream is your branch's default remote destination/source.

- `git push` → sends your commits to upstream
- `git pull` → gets changes from upstream

CDU — Cloud And Data Universe —

WHAT IS UPSTREAM?

An upstream branch is the remote-tracking branch that your local branch is configured to track.



If `master` tracks `origin/master`, then `origin/master` is the upstream branch of `master`.

WHY DO WE NEED UPSTREAM?

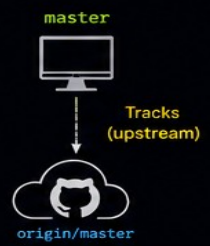
Without upstream, you must always specify remote and branch.

Without upstream (have to be explicit)

```
git pull origin master
git push origin master
```

With upstream (Git remembers the relationship)

```
git pull
git push
```



HOW DO WE SET UPSTREAM?

The most common way is using `-u` (set upstream).

```
git push -u origin master
```

This command does TWO things:

1. **Pushes commits**
   Pushes local master to origin/master.
   master ⟷ origin/master
2. **Sets upstream relationship**
   Remembers that master tracks origin/master.
   master ←-------- (tracks) ------→ origin/master

REMOTE vs UPSTREAM

REMOTE (Repository)	VS	UPSTREAM BRANCH (Remote-tracking branch)
origin		origin/master
Name of the remote repository.		The specific remote branch that your local branch tracks.

`origin` = remote name
`origin/master` = upstream branch of local `master`

HOW DO I SEE UPSTREAM?

Use the command:

```
git branch -vv
```

Example output:

```
* master    abc1234 [origin/master] Update login.py
  main      def5678 [origin/main]   Add README.md
```

This shows that local master is tracking origin/master.

EXAMPLE DIAGRAM

`master` tracks `origin/master` (upstream).
So `git pull` and `git push` work without extra arguments.

UPSTREAM CAN BE DIFFERENT FOR DIFFERENT BRANCHES

master	feature-login	feature-payment
↓	↓	↓
origin/master	origin/feature-login	origin/feature-payment

Each local branch can have its own upstream.

MENTAL MODEL

COMMON SCENARIOS

Push
master → origin/master
`git push` (uses upstream)

Pull
origin/master → master
`git pull` (uses upstream)

IMPORTANT COMMANDS

- `git push -u origin master` — Push and set upstream.
- `git push` — Push to upstream.
- `git pull` — Pull from upstream.
- `git branch -vv` — Show upstream info.
- `git branch --set-upstream-to=origin/master master` — Set or change upstream manually.
- `git branch --unset-upstream` — Remove upstream relationship.

REMEMBER:

- Upstream = default remote branch your local branch tracks.
- Upstream makes `git pull` and `git push` simple.
- Check with `git branch -vv` to see your upstream.
- Keep your branches in sync with their upstream.
- Up-to-date code = Better collaboration!

GIT CLONE

`git clone` is used to make a local copy of an existing repository that already exists on a remote (e.g., GitHub).

KEY IDEA

`git clone` copies the entire repository, including history, files, branches and remote configuration to your computer.

CDU — Cloud And Data Universe —

WHAT IS GIT CLONE?

It is used when a Git repository already exists somewhere else (like GitHub) and you want to get a full copy of it on your computer.

BASIC SYNTAX

```
git clone <repository-url>
```

Example:

```
git clone https://github.com/cdulearning/myrepo.git
```

Git will create a folder named `myrepo` (or the name you specify) and put the repository inside it.

WHAT HAPPENS WHEN YOU CLONE?

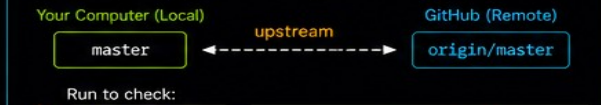
1. Downloads the repository from the remote.
2. Downloads the full Git history.
3. Creates a new local working directory.
4. Initializes a Git repository.
5. Adds a remote named `origin`.
6. Checks out the default branch (e.g., `master/main`).

PRACTICAL EXAMPLE

1	`git clone https://github.com/cdulearning/myrepo.git`	Clone the repository from GitHub.
2	`cd myrepo`	Go into the cloned directory.
3	`git status`	Check repository status.
4	`git remote -v`	See remote (`origin`) configured automatically.
5	`git branch -vv`	Check upstream tracking (e.g., `[origin/master]`).
6	`git log --oneline`	View commit history downloaded from GitHub.

UPSTREAM AFTER CLONE

After cloning, your local branch is usually set to track the corresponding remote-tracking branch.



Run to check:

```
git branch -vv
```

```
* master    abc1234 [origin/master] Initial commit
```

THEN YOU CAN SIMPLY USE:

Pull latest changes	Push your changes
`git pull`	`git push`

(No need to specify remote and branch every time!)

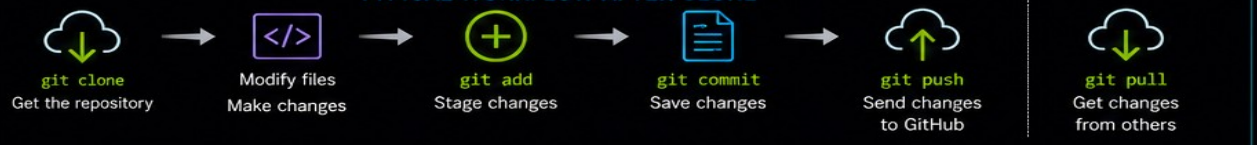
GIT CLONE vs GIT INIT

`git clone`	`git init`
Copies an existing repository	Creates a new repository
Downloads history & files	Starts with no history
Creates remote (origin) automatically	No remote by default
Ready to work with	You add files & commit

MORE CLONE OPTIONS

Clone into a different folder	`git clone <url> my-project`	Creates folder `my-project` and clones inside it.
Clone a specific branch	`git clone -b feature-login <url>`	Clones the repository and checks out `feature-login`.
Clone a shallow copy (latest only)	`git clone --depth 1 <url>`	Downloads only the latest snapshot (not full history).

TYPICAL WORKFLOW AFTER CLONE



TIP

Cloning is not the same as downloading a ZIP!

Clone gets you:
files + history + branches + remote config

REMEMBER

`git clone` is the starting point for working with any existing project on GitHub.

It gives you everything you need to start collaborating!

IMPORTANT COMMANDS

`git clone <url>`	Clone a repository.
`git pull`	Pull from upstream.
`git push`	Push to upstream.
`git branch -vv`	Show upstream info.
`git remote -v`	Show remotes.

REMEMBER:

- Upstream = default remote your local branch tracks.
- Upstream makes `git pull` and `git push` simple.
- Always check with `git branch -vv` to see your upstream.
- Keep your branches in sync with their upstream.

GIT PULL REQUESTS

A Pull Request (PR) is a request to merge changes from one branch into another branch on GitHub.

KEY IDEA

A pull request lets you propose your changes, get them reviewed, discuss improvements, and merge them into the target branch.

CDU — Cloud And Data Universe —

WHAT IS A PULL REQUEST?

A Pull Request (PR) is a way to propose and review changes before merging them into a branch (usually `main`).

PULL REQUEST vs git pull

git pull	Pull Request
A Git command	A GitHub feature
Runs on your computer	Created on GitHub
Gets latest changes from remote	Proposes changes for review
Updates your local branch	Used to merge branches
No review required	Review and discussion can happen
`$ git pull`	Completely different things!

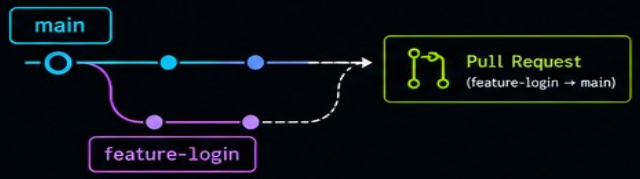
VS

WHY DO WE USE PULL REQUESTS?

- Keeps the main branch stable and clean.
- Allows code review and quality checks.
- Team members can discuss the changes.
- Helps catch bugs before merging.
- Supports collaboration in a safe and organized way.

BRANCHING WORKFLOW

Create a new branch, make changes, and push it to GitHub before creating a Pull Request.



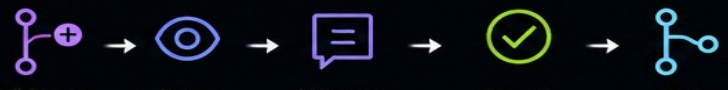
HANDS-ON EXAMPLE (STEP BY STEP)

1	`git switch -c feature-login`	Create a new branch.
2	`# make changes to files`	Modify your files.
3	`git add .`	Stage the changes.
4	`git commit -m "Add login feature"`	Commit the changes.
5	`git push -u origin feature-login`	Push the branch to GitHub.
6	Go to GitHub and click "Compare & pull request"	Create the Pull Request.
7	Set base: main, compare: feature-login	Review and create the PR.

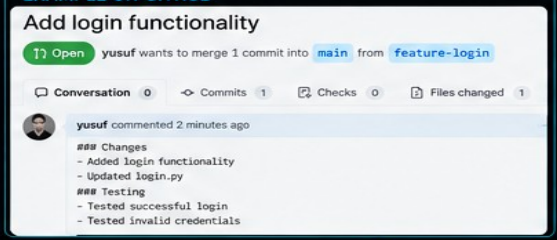
CREATING A PULL REQUEST ON GITHUB

1. Go to your repository on GitHub.
2. Click "Compare & pull request" (or "New pull request").
3. Set the branches: `base: main` ← `compare: feature-login`
4. Add a title (e.g., "Add login functionality").
5. Write a description (what and why).
6. Click "Create pull request".

WHAT HAPPENS AFTER YOU CREATE A PR?



EXAMPLE ON GITHUB



MERGING A PULL REQUEST

Once the changes are approved:

1. Click "Merge pull request".
2. Choose a merge method:
   - Create a merge commit (default)
   - Squash and merge
   - Rebase and merge
3. Confirm the merge.
4. Optionally, delete the branch.

PULL REQUEST FLOW

feature-login (Your changes)
↓
Pull Request (on GitHub)
↓
Review & Approve
↓
Merge
↓
main (Updated)

BEST PRACTICES

- Use descriptive branch names (e.g., feature-login).
- Write clear PR titles and descriptions.
- Keep your changes small and focused.
- Request reviews from team members.
- Address feedback and keep the discussion professional.
- Delete the branch after merging (if not needed).

TIP

A Pull Request is not just about code. It's about collaboration, learning and building better software together!

REMEMBER:

- `git pull` and `Pull Request` are different.
- PRs help keep code quality high.
- Start using Pull Requests in your projects.

Better code.
Better teamwork.
Brighter future.