

GIT & **GITHUB**

Handbook

A comprehensive guide to distributed version control, collaborative workflows, and repository management. Master branching, merging, and pull requests.

Git Handbook

Version Control Fundamentals

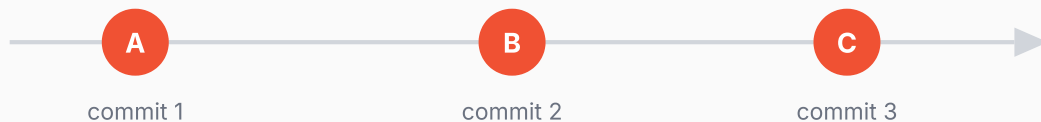
CHAPTER 01

What is Git?

Git is a version control system that tracks changes to files over time. It helps you save progress, return to earlier versions, and work without losing important edits.

Think of Git as a timeline for your project. Each saved point is called a **commit**. You can review changes, compare versions, and undo mistakes when needed.

A Commit Timeline



In the above diagram, each letter is one commit. New commits extend the line to the right.

Git is often used with **GitHub**, which gives you a place to store repositories online, share code, and collaborate with others.

Use Git When You Want

01

A clear history of changes made to your project over time.

02

Safe backup points for your work that you can return to anytime.

03

Easier teamwork when multiple people work on the same project.

04

The ability to undo mistakes without losing important edits.

A Simple Workflow

- 1 Edit your files
- 2 Save the changes
- 3 Commit them
- 4 Push them to GitHub

Git Handbook

Version Control Fundamentals

CHAPTER 01

A Brief History of Git

Git was created by **Linus Torvalds** in 2005 while he was developing the Linux kernel. He needed a fast and reliable tool to manage changes across a large team.

Before Git, developers often used older version control systems that were slower or harder to work with. Git solved that by making branching and merging simple and efficient.

Its design was focused on speed, data integrity, and support for distributed work. That is why Git became the standard tool for modern software development.

Key Reasons Git Became Popular

01

It works well for large projects with many contributors.

02

It supports branching without much overhead.

03

It is fast and reliable, even on large codebases.

04

It is free and open source.

Quick Recap

- Git tracks changes to files using commits.
- Each commit represents a saved point in your project's timeline.
- GitHub is commonly used alongside Git to store and share repositories.
- Git was built by Linus Torvalds in 2005 for managing the Linux kernel.
- Its speed, reliability, and branching support made it the industry standard.

Git Handbook

CHAPTER 02

Understanding Version Control Systems

Understanding Version Control Systems

Version control systems (VCS) track file changes over time. They let you save snapshots, compare versions, and collaborate without overwriting each other's work.

Centralized Version Control

In a centralized model, one server holds the official copy of the project. Developers check out files, make changes, and check them back in.

Examples: SVN, CVS.

Downside: if the central server goes down, collaboration stops.

Distributed Version Control

In a distributed model, every developer has a full copy of the project history on their machine. You commit locally first, then sync with a remote when ready.

Examples: Git, Mercurial.

Benefit: you can work offline and still have full history.

Centralized vs Distributed

	Centralized	Distributed
Full history	On server only	On every machine
Offline work	Limited	Full commits locally
Backup	Single point of failure	Many copies exist

Git Handbook

CHAPTER 02

Understanding Version Control Systems

Why Git Uses a Distributed Model

Git was built by Linus Torvalds in 2005 for large teams working on the Linux kernel. A distributed model means faster operations, safer backups, and flexible workflows like branching and merging.

So in Short

- Centralized VCS relies on one server.
- Distributed VCS gives everyone a full copy.
- Git is distributed by design.

Git Handbook

CHAPTER 03

Git vs GitHub

Git vs GitHub

Git and GitHub are related but not the same thing.



What is Git?

Git is the version control tool that runs on your computer. It tracks changes, creates commits, and manages branches.

What is GitHub?

GitHub is a website that hosts Git repositories online. It adds collaboration features like pull requests, issues, and code review.

Git Repository

A repository (repo) is a project folder tracked by Git. It contains your files plus the hidden `.git` folder with all history.

Local Repository vs Remote Repository

- **Local repo:** on your machine, where you commit daily work.
- **Remote repo:** on GitHub (or similar), where you push and pull to share code.

GitHub Alternatives

Other hosting platforms that work with Git:

GitLab

Bitbucket

Codeberg



Git Handbook

Git vs GitHub

CHAPTER 03

Git vs GitHub Comparison

	Git	GitHub
What it is	Tool	Hosting service
Runs on	Your computer	The cloud
Needs internet	No (for local work)	Yes (to sync)
Stores history	Yes	Yes (as a remote copy)



Git Handbook

CHAPTER 04

Installing Git

Installing Git

Install Git once, then use it from the terminal, VS Code, or GitHub Desktop. The recommended approach is to download it from the official Git website at **git-scm.com**.

Installing Git on Windows

Download the installer from **git-scm.com** and run it with default options.

Or use winget:

```
winget install --id Git.Git -e --source winget
```

Installing Git on macOS

Install Xcode Command Line Tools:

```
xcode-select --install
```

Or use Homebrew:

```
brew install git
```

Installing Git on Linux

Debian / Ubuntu:

```
sudo apt update  
sudo apt install git
```

Git Handbook

CHAPTER 04

Installing Git

Verifying Installation

After installing, confirm Git is available by running:

```
git --version
```

You should see something like `git version 2.x.x`.

Choosing Your Workflow

You can primarily use Git in three different ways:

Command Line Full control, works everywhere. The most powerful and portable way to use Git.

VS Code Built-in Source Control panel, good for daily edits alongside your code.

GitHub Desktop Visual interface, great for beginners who prefer a graphical experience.

All three use the same Git commands under the hood.

Quick Comparison by Platform

WINDOWS

git-scm.com

Download the installer and run with default options, or use winget in the terminal.

MACOS

Xcode / Homebrew

Install via Xcode Command Line Tools or through Homebrew with `brew install git`.

LINUX

apt / Package Manager

Use your distro's package manager. On Debian/Ubuntu: `sudo apt install git`.

Git Handbook

CHAPTER 05

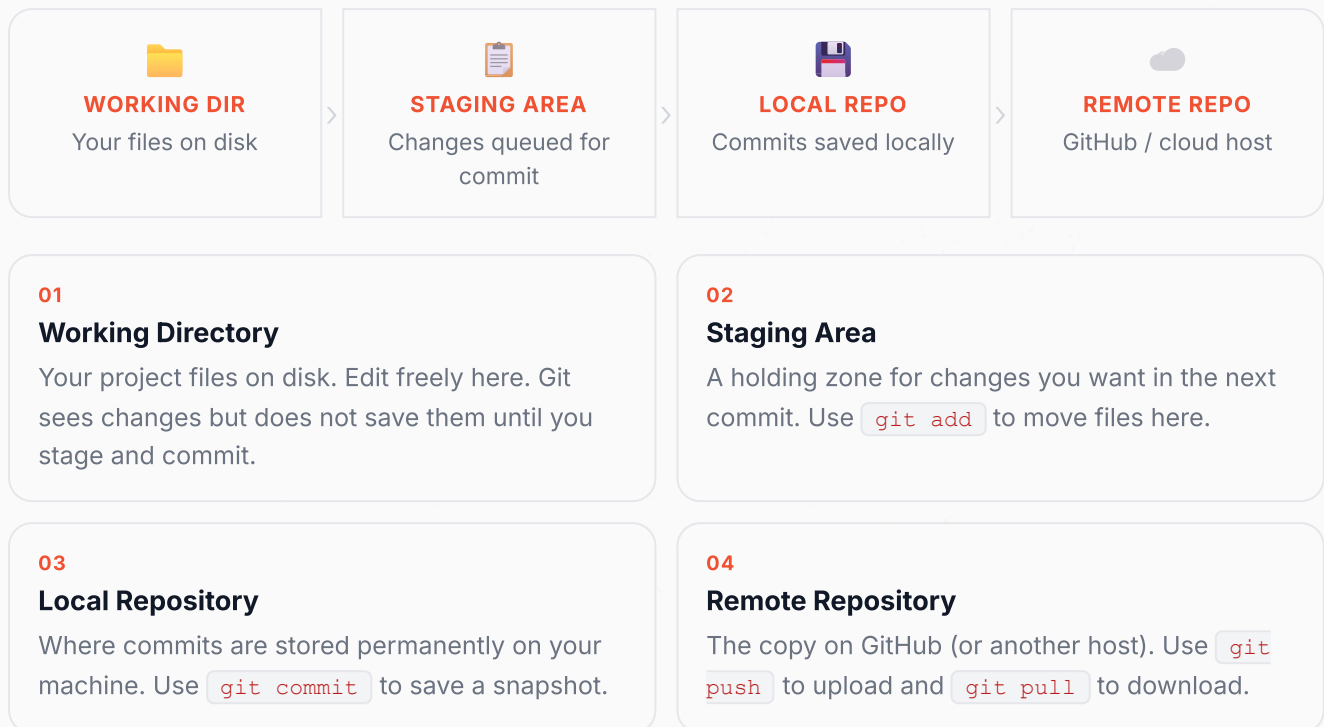
How Git Works

How Git Works

Git moves your changes through **four areas** before they reach GitHub.



The Four Areas of Git



Git Handbook

CHAPTER 05

How Git Works

The Git Workflow

A full cycle from editing to pushing looks like this:

```
# 1. Edit files in your working directory
```

```
# 2. Stage changes
```

```
git add .
```

```
# 3. Commit to local repository
```

```
git commit -m "describe your change"
```

```
# 4. Push to remote repository
```

```
git push origin main
```

Understanding Snapshots

Each commit is a snapshot of your entire project at one moment. Git stores these efficiently, so you can jump back to any point in history.

```
# see all snapshots
```

```
git log --oneline
```

Git Handbook

CHAPTER 06

Configuring Git

Configuring Git

Set your identity once so every commit is attributed to you.



Setting Your Username

```
git config --global user.name "Your Name"
```

Setting Your Email

Use the same email as your GitHub account:

```
git config --global user.email "you@example.com"
```

Viewing Git Configuration

```
# all settings
git config --list

# specific values
git config user.name
git config user.email
```

Global vs Local Configuration

GLOBAL

`--global` flag

Applies to all repos on your machine. Use this for your name and email so every commit is attributed correctly.

LOCAL

No flag (inside a repo)

Applies to one project only. Useful for overriding your email for a specific work or client repository.

Git Handbook

Configuring Git

CHAPTER 06

Overriding Config for One Repo

Navigate into a project folder and set a local config value to override the global setting:

```
# override email for one repo only
cd my-project
git config user.email "work@company.com"
```

Useful Defaults

```
# default branch name for new repos
git config --global init.defaultBranch main

# colored output
git config --global color.ui auto
```

Common Config Commands at a Glance

What it sets	Command
Username	<code>git config --global user.name "Name"</code>
Email	<code>git config --global user.email "e@mail.com"</code>
Default branch	<code>git config --global init.defaultBranch main</code>
Colored output	<code>git config --global color.ui auto</code>
View all settings	<code>git config --list</code>

Git Handbook

CHAPTER 07

Your First Git Repository

Your First Git Repository

A Git repository starts with one command in your project folder.

Creating a Repository

```
mkdir my-project
cd my-project
git init
```

This creates a hidden `.git` folder that stores all history.

git init

Run `git init` in any folder to start tracking it. Only run it **once per project**.

```
# start tracking the current folder
git init

# start tracking a new folder
git init my-new-project
```

git clone

If the repository already exists on GitHub, use `git clone` to download a complete copy to your local machine.

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



Git Handbook

CHAPTER 07

Your First Git Repository

Understanding Repository Structure

```
ls -la
```

```
my-project/  
├── .git/ ← hidden folder, all history lives here  
├── index.html  
├── style.css  
└── app.js
```

⚠ **Never delete or edit the `.git` folder manually.** Doing so destroys the entire version history of your project.

Checking Repository Status

```
git status
```

Shows which files are new, modified, or staged. **Run this often.**

```
# short summary  
git status -s
```

Reading git status Output

```
On branch main  
Changes to be committed:  
  new file:   index.html  
  
Changes not staged for commit:  
  modified:   style.css  
  
Untracked files:  
  app.js
```

GREEN - Staged

Files added to the staging area and ready to be included in the next commit.

YELLOW - Modified

Files changed on disk but not yet staged. Run `git add` to stage them.

RED - Untracked

New files Git sees but has never tracked. Stage them with `git add`.

04

Use `git status -s` for a compact two-column summary of all changes.



Git Handbook

CHAPTER 08

The Big Four Git Commands

The Big Four Git Commands

These four commands cover most of your daily Git work.

`git status` Shows the current state of your repo.

```
git status
git status -s
```

`git add` Stages changes for the next commit.

```
# stage one file
git add README.md

# stage all changes
git add .

# stage all files of one type
git add *.py
```

`git commit` Saves a snapshot with a message.

```
git commit -m "add readme file"

# stage and commit tracked files in one step
git commit -am "fix typo in main script"
```

Note: `git commit -am` only works for files Git already tracks.

Git Handbook

CHAPTER 08

The Big Four Git Commands

`git log` Shows commit history.

```
git log

# compact view
git log --oneline

# last 5 commits
git log --oneline -5

# graph view
git log --oneline --graph --all
```

The Git Lifecycle

```
# edit files, then:
git status          # what changed?
git add .           # stage it
git commit -m "msg" # save it
git log --oneline   # review it
```

Command Reference at a Glance

Command	What it does
<code>git status</code>	Shows staged, modified, and untracked files in the working directory.
<code>git status -s</code>	Compact two-column summary of current repository state.
<code>git add .</code>	Stages all changes in the current directory for the next commit.
<code>git commit -m</code>	Creates a new snapshot with a descriptive message.
<code>git commit -am</code>	Stages tracked files and commits in a single step.
<code>git log --oneline</code>	Displays a compact one-line summary of each commit.
<code>git log --graph --all</code>	Visualises the full branch and merge history as a graph.

Git Handbook

CHAPTER 09

Your First Project with Git

Your First Project with Git

Let me walk you through a complete project from scratch.



Complete Walkthrough

1 Create a Simple Project

```
mkdir hello-git  
cd hello-git  
git init  
echo "# Hello Git" > README.md
```

2 Make Changes

```
echo "print('hello')" > main.py  
git status
```

3 Stage Files

```
git add README.md  
git add main.py  
  
# or stage everything at once  
git add .
```

4 Create a Commit

```
git commit -m "initial commit with readme and main script"
```



Git Handbook

CHAPTER 09

Your First Project with Git

Viewing History

```
git log --oneline
```

Best Practices While Committing

- Commit when a logical piece of work is done.
- One idea per commit.
- Use present tense messages: `add login form` not `added login form`.

```
# good commit flow
git status
git add .
git commit -m "add user input validation"
git log --oneline -3
```

The Full Workflow in One View

Every project follows the same repeating loop: **edit** → **stage** → **commit** → **review**. Repeat this cycle for every logical change.

EDIT

Make your changes to files in the working directory. Git notices but does not save them yet.

STAGE

Use `git add` to move changes into the staging area, ready to be committed.

COMMIT

Use `git commit -m` to save a permanent snapshot with a clear message.

REVIEW

Use `git log --oneline` to inspect your commit history and confirm everything looks right.

Git Handbook

CHAPTER 10

Ignoring Files

Ignoring Files

Not every file belongs in Git. Some should stay local.

Why Some Files Should Not Be Tracked

- Passwords & API keys `(.env)`
- Build output `dist/` , `__pycache__/`
- OS junk `.DS_Store` , `Thumbs.db`
- Large data files
- Dependency folders `node_modules/`

Understanding `.gitignore`

Create a `.gitignore` file in your repo root. Git skips anything listed there.

```
touch .gitignore
```

Example `.gitignore`:

```
# environment variables
.env

# Python
__pycache__/
*.pyc
venv/

# OS files
.DS_Store
Thumbs.db

# build output
dist/
build/
```

Git Handbook

CHAPTER 10

Ignoring Files

Common .gitignore Rules

```
# ignore a specific file
secret.txt

# ignore a folder and everything inside
node_modules/

# ignore all files of a type
*.log

# but keep one specific file
!important.log
```

We read `!important.log` as *not* `important.log`, so it will not be ignored by Git. `!` is the not operator.

Pattern Reference

Pattern	What it ignores
<code>secret.txt</code>	A specific file named <code>secret.txt</code> anywhere in the repo.
<code>node_modules/</code>	An entire folder and all of its contents recursively.
<code>*.log</code>	All files with the <code>.log</code> extension.
<code>!important.log</code>	Exception: keep this file even if <code>*.log</code> is ignored above it.
<code>.env</code>	Environment variable file with secrets and API keys.

Empty Folders & .gitkeep

Git does not track empty folders. To keep one, add a placeholder:

```
mkdir logs
touch logs/.gitkeep
git add logs/.gitkeep
```

If a file was already tracked before you added it to `.gitignore`:

```
git rm --cached filename
git commit -m "stop tracking filename"
```



Git Handbook

Working Alone with Git

CHAPTER 11

Working Alone with Git

Even solo projects benefit from Git. It is your safety net and project diary.



Why Even Solo Developers Need Git

- Undo mistakes safely.
- Try ideas on branches without breaking main code.
- Track what you changed and when.
- Push to GitHub as a backup.

Typical Personal Workflow

MORNING START WORK	<pre># morning: start work git status git pull origin main</pre>
DURING THE DAY MULTIPLE TIMES	<pre># during the day (multiple times) git add . git commit -m "add chart to dashboard"</pre>
END OF DAY BACK UP	<pre># end of day: back up to GitHub git push origin main</pre>

Rolling Back Changes

Discard unstaged edits to one file	<code>git restore filename.py</code>
Discard all unstaged edits	<code>git restore .</code>
Undo the last commit, keep file changes	<code>git reset --soft HEAD~1</code>



Git Handbook

Working Alone with Git

CHAPTER 11

Tracking Project Progress

```
# what changed since yesterday
git log --oneline --since="1 day ago"

# who changed what in a file
git log --oneline -- filename.py

# see actual code changes
git diff
```

Rolling Back: Full Reference

git restore <file>

Throws away unstaged edits for a specific file, restoring it to the last committed version.

git restore .

Discards all unstaged edits in the working directory. Use with caution: changes are lost.

git reset --soft HEAD~1

Undoes the most recent commit but keeps all file changes staged so you can re-commit.

git diff

Shows line-by-line differences between your working directory and the last commit.



Git Handbook

CHAPTER 12

Working with Teams

Working with Teams

Git shines when multiple people work on the same codebase.

Collaboration in Git

Each person works on their own local copy. Changes sync through a shared remote on GitHub.

Sharing Code

GIT PULL

Get latest team changes

get the latest team changes

```
git pull origin main
```

GIT PUSH

Send your commits to team

send your commits to the team

```
git push origin main
```

Always pull before you push to avoid conflicts.

Working with Multiple Developers

Best practice: each person works on their own branch.

1

CREATE & SWITCH TO A FEATURE BRANCH

```
git checkout -b feature/login
```

2

MAKE CHANGES AND COMMIT

make changes

```
git add .
```

```
git commit -m "add login page"
```

3

PUSH AND OPEN A PULL REQUEST

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

Then open a **pull request** on GitHub for review.

Git Handbook

CHAPTER 12

Working with Teams

Keeping Code Organised

MAIN

Holds stable, working code.
Never commit experimental work here directly.

FEATURE BRANCHES

Hold work in progress. Each feature or fix gets its own branch.

PULL REQUESTS

Let the team review and discuss changes before merging into main.

```
# see all branches (local and remote)
```

```
git branch -a
```

```
# see who changed what recently
```

```
git log --oneline --author="name" -10
```

Team Workflow at a Glance

01 - Start

Pull the latest changes from `main` before starting any new work.

02 - Branch

Create a feature branch with `git checkout -b feature/name` for every piece of work.

03 - Push

Push the branch to GitHub and open a pull request for team review.

04 - Merge

After approval, merge the PR into `main` and delete the feature branch.

Git Handbook

CHAPTER 13

Branching

Branching

Branches let you work on features without touching stable code.

What is a Branch?

A branch is an independent line of development. `main` is the default branch.

Branching at `B` lets you build a feature without touching `main`:

```
graph LR
    A-B-C --> main
    A-B-C --> D-E[feature/search]
```

Each letter is a commit. Lines show parent links. Time flows left to right.

Why Branches Exist

- They help build features in isolation
- They help fix bugs without breaking production code
- They help experiment safely

Creating a Branch

```
git branch feature/search
```

Switching Branches

```
git checkout feature/search
```

```
# modern alternative
git switch feature/search
```


Git Handbook

CHAPTER 13

Branching

Creating and Switching Together

```
git checkout -b feature/search
```

```
# modern alternative
```

```
git switch -c feature/search
```

Viewing Branches

```
# local branches (* shows current)
```

```
git branch
```

```
# all branches including remote
```

```
git branch -a
```

```
# current branch name only
```

```
git branch --show-current
```

Understanding the Git Graph

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

This shows how branches split and merge over time. A typical graph looks like:

```
      D-E
     /   \
    /       \
A-B-C-----M
```

M is a merge commit where two lines of work came back together.

Git Handbook

CHAPTER 14

Merging Branches

Merging Branches

Merging combines the work from one branch into another.

What is Merge?

Merge takes commits from a feature branch and adds them to your target branch (usually `main`).

Each letter is a commit. Lines show parent links. Time flows left to right.

Merging a Feature Branch

```
# switch to the branch you want to merge INTO
git switch main

# bring in the feature branch
git merge feature/search
```

Fast Forward Merge

Happens when `main` has no new commits since the branch was created. Git simply moves the pointer forward.

```
A-B-C-D
      ↑
    main
```

Git moves `main` to D. No merge commit needed.

```
git switch main
git merge feature/search
# output: Fast-forward
```

Git Handbook

CHAPTER 14

Merging Branches

Merge Commit

Happens when both branches have new commits. Git creates a merge commit to combine them.

After branching at `B`, the feature branch has C and D while `main` gains E:

```
      C-D
     /
    /
   A-B-E
```

Running `git merge feature/search` on `main` creates merge commit M:

```
      C-D
     /  \
    /      \
   A-B-E—M
```

M has two parents (D and E). History keeps the fork visible.

```
git switch main
git merge feature/search
# output: Merge made by the 'ort' strategy
```

Deleting Branches

After a successful merge, delete the feature branch:

```
# delete local branch
git branch -d feature/search
```

```
# delete remote branch
git push origin --delete feature/search
```

Why Deleted Branches Are Still Safe

The commits live in Git history. You can always find them:

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

Git Handbook

CHAPTER 15

Git Stash

Git Stash

Stash saves uncommitted work temporarily so you can switch tasks.

Why Git Stash Exists

You are mid-feature but need to fix an urgent bug on `main`. Stash lets you park your changes without committing half-finished work.

Saving Work Temporarily

```
git stash

# with a label so you remember what it is
git stash push -m "wip on search feature"
```

Viewing Stashes

```
git stash list
```

Restoring a Stash

```
# apply most recent stash (keeps it in the list)
git stash apply

# apply a specific stash
git stash apply stash@{1}
```



Git Handbook

CHAPTER 15

Git Stash

Applying vs Popping

```
# apply: keeps the stash in the list
```

```
git stash apply
```

```
# pop: applies and removes the stash
```

```
git stash pop
```

Real World Example

```
# save current work
```

```
git stash push -m "half-done login form"
```

```
# switch to fix a bug
```

```
git switch main
```

```
git switch -c hotfix/typo
```

```
# fix, commit, merge...
```

```
# return to your feature
```

```
git switch feature/login
```

```
git stash pop
```

```
# delete a stash you no longer need
```

```
git stash drop stash@{0}
```

```
# clear all stashes
```

```
git stash clear
```

Git Handbook

CHAPTER 16

Merge Conflicts

Merge Conflicts

Conflicts happen when Git cannot automatically combine two changes to the same lines.

Why Conflicts Happen

Common causes:

- two people edit the same line
- you edit a file locally while someone else pushes changes to it
- you merge or rebase branches with overlapping edits

Both branches changed the same lines after splitting at `B`:

```
      C (you changed line 10)
    /
A-B-E (main also changed line 10)
```

Git cannot choose a winner automatically and that is a conflict.

Resolving Conflicts

```
# try to merge
git merge feature/search
# output: CONFLICT in filename.py
```

Open the conflicted file. You will see markers like:

Git Handbook

CHAPTER 16

Merge Conflicts

Conflict Markers

```
<<<<<<< HEAD
your version
=====
their version
>>>>>> feature/search
```

Edit the file to keep the correct code, remove the markers, then:

```
git add filename.py
git commit -m "resolve merge conflict in filename.py"
```

Understanding git diff

`git diff` is used to see the differences between the working directory with the staging area and the staging area with the repository.

- `git diff` → working tree vs staging area
- `git diff --staged` → staging area vs last commit (`HEAD`)

```
# see unstaged changes
git diff

# see staged changes
git diff --staged

# see conflict markers in a file
git diff filename.py
```

Git Handbook

CHAPTER 16

Merge Conflicts

Best Practices to Avoid Conflicts

```
# pull before starting work
git pull origin main

# pull before pushing
git pull origin main
git push origin main

# keep branches short-lived
# communicate with teammates on shared files
```

Abort a Merge if You Are Stuck

```
git merge --abort
```

git merge --abort undoes the in-progress merge and returns your files to the state before you ran

```
git merge .
```


Git Handbook

CHAPTER 17

Git History

Git History

Your commit history is a searchable record of every change.



Commit IDs

Every commit gets a unique hash (SHA). You only need the first 7 characters most of the time.

```
git log --oneline
# a1b2c3d add login page
```

Viewing Commit History

```
# full log
git log

# one line per commit
git log --oneline

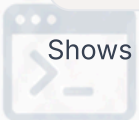
# with file changes
git log --stat

# history for one file
git log --oneline -- README.md
```

Understanding Commit Graphs

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

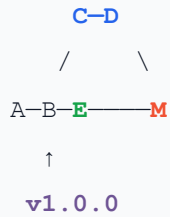
Shows branches, merges, and tags visually. For example:



Git Handbook

CHAPTER 17

Git History



Read it left to right: branches split, work happens in parallel, then merge commit `M` joins them. Tag `v1.0.0` stays pinned on `B` even as `main` moves forward.

Inspecting Previous Versions

see what changed in a commit

```
git show a1b2c3d
```

see file at a past commit

```
git show a1b2c3d:README.md
```

compare two commits

```
git diff a1b2c3d b2c3d4e
```

temporarily view old code

```
git checkout a1b2c3d
```

```
git switch main # return to latest
```

search commit messages

```
git log --oneline --grep="login"
```

see commits by date

```
git log --oneline --since="2025-01-01" --until="2025-06-01"
```

Git Handbook

CHAPTER 18

Undoing Mistakes

Undoing Mistakes

Git gives you several ways to undo work depending on how far gone it is.

Undoing Working Directory Changes

Discard edits to one file (not yet staged):

```
git restore filename.py
```

Discard all unstaged edits:

```
git restore .
```

Undoing Staged Changes

Unstage a file (keeps your edits):

```
git restore --staged filename.py
```

Unstage everything:

```
git restore --staged .
```

Git Reset

Move the branch pointer to undo commits.

```
Before:  A-B-C
          ↑
        main
```

Git Handbook

CHAPTER 18

Undoing Mistakes

```
git reset --hard HEAD~1
```

```
After:  A-B
        ↑
      main
```

Hard Reset

```
git reset --hard HEAD~1
```

This permanently deletes uncommitted work and the last commit. Use with care.

When NOT to Use Hard Reset

- After you have already pushed to GitHub (use `git revert` instead)
- When you are unsure what you will lose
- On shared branches other people depend on

Safer alternative for pushed commits:

```
git revert HEAD
git revert a1b2c3d
```

`git revert` creates a new commit that undoes a previous one without rewriting history.



Git Handbook

CHAPTER 18

Undoing Mistakes

Reverting a Commit

```
A-B-C-C'  
      ↑  
    main
```

`C'` reverses the changes from `C`. The original commit stays in history and is safe for shared branches.



Git Handbook

CHAPTER 19

Git Rebase

Git Rebase

Rebase moves your branch commits on top of another branch for a cleaner history.

What is Rebase?

Instead of creating a merge commit, rebase replays your commits on top of the latest `main`.

Your feature branch diverged from `main` at `B`. You have C and D; `main` moved to `E`:

```
    C-D      ← feature branch
  /
A-B-E      ← main
```

After `git rebase main`, Git replays your commits on top of E:

```
A-B-E-C'-D' ← feature (linear)
      ↑
      main
```

C' and D' are new commits with the same changes as C and D.

```
git switch feature/search
git rebase main
```

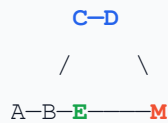
Git Handbook

CHAPTER 19

Git Rebase

Rebase vs Merge

Merge keeps the fork and adds a merge commit `M`:



Rebase replays your work for a straight line without commit M:

A-B-E-C'-D'

	MERGE	REBASE
History	Preserves branch structure	Linear, cleaner
Merge commit	Yes	No
Safe on shared branches	Yes	No

When to Use Rebase

- Before opening a pull request (clean up your branch)
- To sync a feature branch with updated `main`
- On branches only you work on

```
# update your feature branch with latest main
git switch feature/search
git fetch origin
git rebase origin/main
```

Git Handbook

CHAPTER 19

Git Rebase

Best Practices

```
# rebase your branch before pushing a PR
git switch feature/search
git rebase main
git push --force-with-lease origin feature/search
```

Rules:

- Never rebase commits already pushed to a shared branch
- If rebase gets messy, abort and try again:

```
git rebase --abort
```



Git Handbook

CHAPTER 20

Git Tags

Git Tags

Tags mark important points in history, usually releases.



Why Tags Exist

- Mark version releases (`v1.0.0`)
- Bookmark stable checkpoints
- Reference a specific commit permanently

Unlike branches, tags do not move when you add new commits:

```
A-B-C-D
      ↑ ↑
    v1.0.0 main

# after two more commits on main

A-B-C-D-E-F
      ↑   ↑
    v1.0.0 main
```

`v1.0.0` stays on C forever. `main` keeps moving.

Lightweight Tags

A simple pointer to a commit. No extra metadata.

```
git tag v1.0.0
```

Annotated Tags



Recommended for releases. Stores author, date, and message.



Git Handbook

CHAPTER 20

Git Tags

```
git tag -a v1.0.0 -m "first stable release"
```

Creating Tags

```
# tag current commit
git tag v1.0.0

# tag a specific commit
git tag v0.9.0 a1b2c3d

# annotated tag with message
git tag -a v1.0.0 -m "release version 1.0.0"
```

Listing Tags

```
# list all tags
git tag

# list tags matching a pattern
git tag -l "v1.*"

# see tag details
git show v1.0.0
```

Push tags to GitHub:

```
# push one tag
git push origin v1.0.0

# push all tags
git push origin --tags
```

Delete a tag:

```
git tag -d v1.0.0
git push origin --delete v1.0.0
```

Git Handbook

GitHub Essentials

CHAPTER 21

GitHub Essentials

Connect your local Git repo to GitHub and start sharing code.



Creating a GitHub Account

Sign up at github.com. Choose a username you are happy to put on a resume.

Creating Your First Repository

On GitHub: click **New repository**, name it, choose public or private, and create it. Do not initialize with a README if you already have a local repo.

Public vs Private Repositories

PUBLIC

Anyone can see your code. Good for portfolios and open source.

PRIVATE

Only you and invited collaborators can see it.

Connecting Local Git with GitHub

```
git remote add origin https://github.com/username/repo-name.git
git remote -v
```



Git Handbook

GitHub Essentials

CHAPTER 21

Understanding origin

`origin` is the default name for your remote repository. You can rename it but most people keep `origin`.

```
git remote rename origin upstream
```

Understanding main

`main` is the default branch name. Older repos may use `master`.

```
git branch -M main
```

First Push

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

The `-u` flag sets `main` as the default upstream so future pushes need only `git push`.

Pushing Future Changes

```
git add .  
git commit -m "update homepage"  
git push
```

Pulling Changes

```
git pull origin main
```

Git Handbook

GitHub Essentials

CHAPTER 21

Fetch vs Pull

```
# download changes without merging
```

```
git fetch origin
```

```
# download and merge in one step
```

```
git pull origin main
```

Your teammate pushed commit `D` while you were still at `C`:

```
GitHub (origin/main):  A—B—C—D
```

```
Your machine (main):  A—B—C
```

- `git fetch` downloads D but leaves your working branch unchanged
- `git pull` downloads D and fast-forwards your `main` to match

Use `git fetch` when you want to review changes before merging.

Git Handbook

CHAPTER 22

Cloning Existing Projects

Cloning Existing Projects

Cloning downloads a full copy of a remote repository to your machine: project files, full Git history, and remote connection all in one step.

Clone Methods

HTTPS

Easiest to Start

Works without any key setup. Requires your GitHub username and password (or a personal access token) when pushing.

SSH

Recommended for Daily Use

Requires setting up an SSH key pair once. After that, no passwords are needed for any push or pull.

git clone

```
# HTTPS (easiest to start)
git clone https://github.com/username/repo-name.git

# clone into a specific folder name
git clone https://github.com/username/repo-name.git my-folder

# SSH (after setting up SSH keys)
git clone git@github.com:username/repo-name.git
```

Exploring the Repository

Command	What it shows
<code>cd repo-name</code>	Navigate into the cloned folder.
<code>git status</code>	Current state of the working directory.
<code>git log --oneline -5</code>	Last five commits in compact form.
<code>git branch -a</code>	All local and remote branches.
<code>git remote -v</code>	The fetch and push URLs for the remote.

Git Handbook

CHAPTER 22

Cloning Existing Projects

Making Changes & Pushing Back

1 Create a feature branch

```
git switch -c feature/my-change
```

2 Edit, stage, and commit

```
# edit files
git add .
git commit -m "add my contribution"
```

3 Push and open a pull request

```
git push -u origin feature/my-change
```

Then open a **pull request** on GitHub for the team to review your changes.

If You Have Write Access

You can merge directly into `main` without a pull request:

```
git switch main
git pull origin main
git merge feature/my-change
git push origin main
```

GitHub Handbook

chapter 23

GitHub Desktop: Getting Started

GitHub Desktop

GitHub Desktop gives you a visual interface for Git without typing every command.



Why Use GitHub Desktop?

01

No terminal required for basic workflows.

02

Easy to see changed files side by side.

03

Simple branch and merge management.

04

Good stepping stone before the command line.

Download GitHub Desktop from desktop.github.com.

Installing GitHub Desktop

Install the application, sign in with your GitHub account, and authorize the app to connect to your repositories.

Cloning Repositories

1

Open the clone dialog

Go to **File > Clone Repository**.

2

Choose a repository

Pick one from your GitHub account, or paste a URL directly.

3

Clone it locally

Choose a local path and click **Clone**.



GitHub Handbook

chapter 23

GitHub Desktop: Getting Started

Committing Changes

- 1 Edit your files**
Make changes in your editor as usual.
- 2 Review changed files**
GitHub Desktop lists all changed files in the left panel.
- 3 Describe the change**
Write a short summary and, optionally, a longer description.
- 4 Commit**
Click **Commit to main** (or your current branch).



Push and Pull

Action	What it does
Push origin	Sends your commits to GitHub.
Fetch origin	Checks for remote changes without merging them.
Pull origin	Downloads and merges remote changes into your branch.

Trigger these from the toolbar buttons or the **Repository** menu.

Managing Branches

```
# create a new branch
Branch > New Branch

# change branch
Branch > Switch Branch

# combine changes into current branch
Branch > Merge into current branch

# remove after merge
Branch > Delete
```

Viewing History

Click the **History** tab to see all commits, who made them, and what changed.

GitHub Handbook

GitHub Desktop: Getting Started

chapter 23

Resolving Conflicts

When a merge conflict occurs, GitHub Desktop highlights the conflicted files.

- 1 Open the conflicted files**
Open each flagged file in your editor.
- 2 Choose what to keep**
Decide which changes should remain in the final version.
- 3 Mark as resolved**
Tell GitHub Desktop the conflict has been handled.
- 4 Commit the merge**
Commit the resolved changes to complete the merge.

Publishing Repositories

For a local repository that is not yet on GitHub, use the Repository menu to publish it:

Repository > Publish repository

Choose **public** or **private**, then publish.

GitHub Handbook

chapter 23

GitHub Desktop: Getting Started

What is Forking?

Forking creates your own copy of someone else's GitHub repository under your account. Since the fork belongs to you, you can freely experiment, make changes, and develop independently without affecting the original project.

01

Creates your own copy of an existing repository.

02

Lets you work on projects even when you don't have write access.

03

Make changes safely without modifying the original repository.

04

Submit your improvements later using a Pull Request.

Example: If you want to improve an open-source project but don't have permission to edit it directly, first fork the repository, make your changes, and then create a Pull Request for the maintainers to review.

How to Fork a Repository

1

Open the repository

Visit the GitHub repository you want to contribute to.

2

Click Fork

Click the **Fork** button in the top-right corner to create your own copy under your GitHub account.

3

Clone your fork

Open GitHub Desktop and clone your forked repository to your local machine.

4

Commit and create a Pull Request

Make your changes, push them to your fork, and create a Pull Request to propose your improvements.



Fork vs Clone

Fork	Clone
Creates a copy on GitHub.	Creates a copy on your computer.
You own the copied repository.	You work with a local copy of a repository.
Used when contributing to projects you don't own.	Used for day-to-day development.

Good Practice: Fork repositories when you genuinely want to learn, fix bugs, or contribute meaningful improvements. Avoid creating spam or unnecessary Pull Requests, as they waste maintainers' time and may cause your future contributions to be ignored.

GitHub Handbook

Git in Visual Studio Code

chapter 24

Source Control Panel

VS Code has built-in Git support in the Source Control panel, showing changed files, the staging area, and a commit box, all without leaving the editor.

WINDOWS / LINUX

Ctrl + Shift + G

Opens the Source Control panel.

MACOS

Cmd + Shift + G

Opens the Source Control panel.

Staging Changes

01

Click the  next to a file to stage it.

02

Click  next to **Changes** to stage everything at once.

03

Click  to unstage a file.

04

Every click maps to a real Git command underneath.

```
# terminal equivalent  
git add filename.py  
git add .
```

Creating Commits

Type a message in the text box at the top of Source Control and click the checkmark to commit.

```
git commit -m "your message"
```



GitHub Handbook

Git in Visual Studio Code

chapter 24

Synchronizing Changes

Click the sync icon (circular arrows) in the status bar to pull and push in one step.

```
git pull origin main
git push origin main
```

Viewing File Differences

Click any changed file in Source Control to open a side-by-side diff view.

```
git diff
git diff --staged
```

Timeline View

Right-click a file and choose **Open Timeline** to see its full commit history.

```
git log --oneline -- filename.py
```

Managing Branches

Click the branch name in the bottom-left status bar to create or switch branches.

```
git switch -c feature/new-branch
git switch main
```

GitHub Handbook

Git in Visual Studio Code

chapter 24

Merge Conflict Resolution

VS Code highlights conflicts directly in the editor with inline action buttons.

KEEP MINE

Accept Current keeps your local version of the conflicting block.

KEEP THEIRS

Accept Incoming keeps the incoming version instead.

KEEP BOTH

Accept Both combines both versions into the file.

Publishing to GitHub

Click **Publish to GitHub** in Source Control when starting a new local repository.

```
git remote add origin https://github.com/username/repo.git
git push -u origin main
```

Git Graph Extension

Install the **Git Graph** extension for a visual branch history.

Extensions > search "Git Graph" > Install

Then run **Ctrl+Shift+P** > **Git Graph: View Git Graph**.

GitHub Handbook

Modern Git Workflow

chapter 25

Git in the AI Era

AI tools change how fast you write code. Git keeps that speed safe.

AI can generate files quickly, but it can also introduce bugs or overwrite working code. Git lets you review, commit, and roll back AI output like any other change.

Using Git While Vibe Coding

1 Commit before letting AI make large changes

```
git add .  
git commit -m "working state before AI refactor"
```

2 Review and commit after AI edits

```
git diff  
git add .  
git commit -m "AI-generated dashboard layout"
```

3 Roll back if the result is bad

```
git restore .  
# or  
git reset --hard HEAD~1
```


GitHub Handbook

Modern Git Workflow

chapter 25

Reviewing AI Generated Code

Never commit AI output blindly.

```
# see every line AI changed
git diff

# see changes file by file
git diff --stat
```

AI Generated Commit Messages

Some tools suggest commit messages. Edit them to be specific rather than generic.

BAD

Too Vague

```
git commit -m "update files"
```

GOOD

Specific

```
git commit -m "add input validation to
signup form"
```

Daily Git Workflow

```
# start of session
git pull origin main
git switch -c feature/todays-work

# during session (commit often)
git add .
git commit -m "describe each logical step"

# end of session
git push -u origin feature/todays-work
# open pull request on GitHub
```

GitHub Handbook

Modern Git Workflow

chapter 25

Quick Reference

Command	When to use it
<code>git diff</code>	Review every line changed before staging.
<code>git diff --stat</code>	See a summary of changes, file by file.
<code>git restore .</code>	Discard uncommitted changes after a bad AI edit.
<code>git reset --hard HEAD~1</code>	Roll back the last commit entirely.
<code>git switch -c name</code>	Start a new branch for today's session.
<code>git push -u origin name</code>	Publish the branch and open a pull request.



GitHub Handbook

Git Best Practices

chapter 26

Git Best Practices

Good habits prevent most Git headaches.



Core Commit Habits

01

Commit frequently. Small commits are easier to review, revert, and understand.

02

Keep commits small. One logical change per commit: don't bundle unrelated fixes.

03

Write good messages. A short summary, written in present tense.

04

These three habits together make history easy to read and safe to roll back.

```
git commit -m "add email field to form"
git commit -m "fix null check in user parser"
git commit -m "add dark mode toggle to settings"
```

One Feature Per Branch

Isolate each feature in its own branch so unrelated work never mixes.

```
git switch -c feature/dark-mode
# work only on dark mode here
```



GitHub Handbook

Git Best Practices

chapter 26

Pull Before Push

Always sync with the remote before pushing to avoid unnecessary conflicts.

```
git pull origin main
git push origin main
```

Never Commit Secrets

No API keys, passwords, or `.env` files should ever enter Git history.

```
git rm --cached .env
# stops tracking the file but keeps it on your machine

echo ".env" >> .gitignore
git commit -m "remove env file from tracking"
```

If a secret was ever committed, **rotate it immediately** ; removing it from Git does not undo the exposure.

Use .gitignore

Set it up on day one, before your first commit.

```
echo ".env" >> .gitignore
echo "__pycache__/" >> .gitignore
echo ".DS_Store" >> .gitignore
git add .gitignore
git commit -m "add gitignore"
```

GitHub Handbook

Git Best Practices

chapter 26

Review Before Every Commit

A quick review pass catches mistakes before they enter history.

1 Check what's changed

```
git status
```

2 Review the actual edits

```
git diff  
git diff --staged
```

3 Commit with a clear message

```
git commit -m "your message"
```

Quick Reference

Command	What it does
<code>git status</code>	Shows what's changed before staging.
<code>git diff --staged</code>	Reviews exactly what a commit will contain.
<code>git rm --cached file</code>	Stops tracking a file without deleting it locally.
<code>git switch -c name</code>	Creates an isolated branch for one feature.

GitHub Handbook

Git Cheat Sheet

chapter 27

Git Command Reference

Quick copy-paste reference for commands you will use again and again.

Repository Commands

Command	What it does
<code>git init</code>	Starts a new repository.
<code>git clone <url></code>	Copies a remote repository.
<code>git status</code>	Checks the current state.
<code>git status -s</code>	Short, compact status output.

Tracking Commands

Command	What it does
<code>git add <file></code>	Stages one file.
<code>git add .</code>	Stages all changes.
<code>git add -p</code>	Stages changes interactively, hunk by hunk.
<code>git commit -m "msg"</code>	Saves a snapshot of staged changes.
<code>git commit -am "msg"</code>	Stages tracked files and commits in one step.

Branch Commands

Command	What it does
<code>git branch</code>	Lists local branches.
<code>git branch -a</code>	Lists all branches, local and remote.
<code>git switch <branch></code>	Switches to an existing branch.
<code>git switch -c <branch></code>	Creates a branch and switches to it.
<code>git merge <branch></code>	Merges a branch into the current one.
<code>git branch -d <branch></code>	Deletes a local branch.

GitHub Handbook

Git Cheat Sheet

chapter 27

Remote Commands

Command	What it does
<code>git remote -v</code>	Shows configured remotes.
<code>git remote add origin <url></code>	Connects the repo to GitHub.
<code>git push -u origin main</code>	First push, sets the upstream branch.
<code>git push</code>	Pushes commits using the tracked upstream.
<code>git pull origin main</code>	Pulls and merges remote changes.
<code>git fetch origin</code>	Downloads remote changes without merging.

History Commands

Command	What it does
<code>git log</code>	Shows the full commit history.
<code>git log --oneline</code>	Compact, one-line-per-commit history.
<code>git log --oneline --graph --all</code>	Visual graph of all branches.
<code>git show <commit></code>	Inspects a single commit in detail.
<code>git diff</code>	Shows unstaged changes.
<code>git diff --staged</code>	Shows staged changes.

Use `git fetch` when you want to see remote changes before deciding to merge; `git pull` does both at once.



GitHub Handbook

Git Cheat Sheet

chapter 27

Undo Commands

Command	What it does
<code>git restore <file></code>	Discards unstaged edits to a file.
<code>git restore --staged <file></code>	Unstages a file, keeping the edits.
<code>git reset --soft HEAD~1</code>	Undoes the last commit, keeps changes staged.
<code>git reset --hard HEAD~1</code>	Undoes the last commit and discards changes.
<code>git revert <commit></code>	Safely undoes a commit that's already pushed.
<code>git stash</code>	Saves uncommitted work temporarily.
<code>git stash pop</code>	Restores the most recently stashed work.

Prefer `git revert` over `git reset --hard` once a commit has been pushed and shared with others.

Quick Workflow

```
git pull origin main
git switch -c feature/my-feature
# edit files
git add .
git commit -m "add my feature"
git push -u origin feature/my-feature
# open pull request on GitHub
```


GitHub Handbook

Conclusion

chapter 28

You Now Have a Solid Foundation

You started with the basics: what version control is, how commits and branches work, and the core commands that drive every workflow. From there, you learned to collaborate with others, resolve conflicts, undo mistakes, and use GitHub to share and review code.

Keep Practicing

The real skill comes from practice. Use Git on every project, even small ones. Commit often, write clear messages, and pull before you push. When something goes wrong, you have the tools to fix it.

