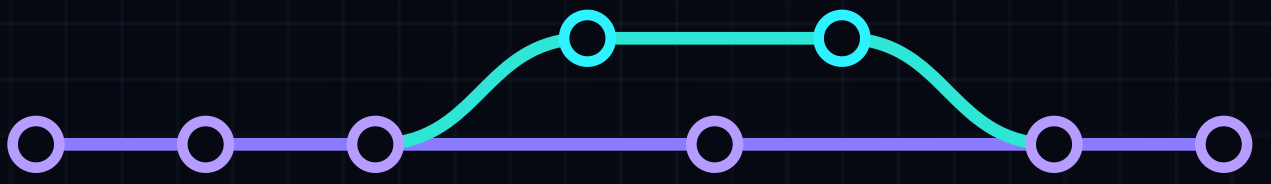
SBCCAS · AMROLI COLLEGE, SURAT

STUDENT EDITION

CERTIFICATE COURSE · 2 CREDITS · 30 HOURS

Git & GitHub Fundamentals

Version control from your first commit to your first pull request — told through the Amroli Canteen project.



Sutex Bank College of Computer Applications & Science
Amroli College · Affiliated to Veer Narmad South Gujarat University (VNSGU) · Jivan Jyot Trust

sbccas.github.io/git-github-fundamentals/
Edition 1 · October 2026

About this book

The course

Course	Git & GitHub Fundamentals (certificate course)
Credits · hours	2 credits · 30 hours · 15 sessions of 2 hours
Units	1. Introduction to Version Control (4 h) 2. Getting Started with Git (8 h) 3. Branching and Merging (6 h) 4. Advanced Git Commands (6 h) 5. Introduction to GitHub (4 h)
For	BCA · BCA – AI · B.Sc. Data Science · M.Sc. IT students
Practice	15 browser missions · 4 GitHub labs · 5 unit quizzes · final self-check

How to use this book

Read a chapter, then **practise in your browser** — the course website runs real Git on your phone or a lab PC, with no installation:

- **Playground:** sbccas.github.io/git-github-fundamentals/playground/
- **Missions** (15 story challenges): sbccas.github.io/git-github-fundamentals/missions/
- **Unit quizzes, labs and your Git Passport** are on the website too.

Commands in grey boxes are real: type the lines starting with `$`, without the `$`.

The college

Sutex Bank College of Computer Applications & Science (Amroli College)
Amroli, Surat, Gujarat, India
Affiliated to Veer Narmad South Gujarat University (VNSGU) · Managed by Jivan Jyot Trust
www.amrolicollege.ac.in · info@amrolicollege.ac.in
Instagram @amrolicollege.official

Faculty



Hitesh Patel
Department Coordinator · B.Sc. Data Science
github.com/contact2hitech

Book and website created by Hitesh Patel.

Licence

Text © Sutex Bank College of Computer Applications & Science, licensed CC BY-SA 4.0: you may copy, print and share it with credit. Student Edition · Edition 1 · October 2026. Learning material, not an official syllabus document or certificate. Source: github.com/sbccas/git-github-fundamentals.

Contents

About this book 2

Chapter 1 · Introduction to Version Control 5

The Amroli Canteen problem 5

What is version control? 6

Three kinds of version control 7

Benefits of Git and GitHub 8

Installing Git on Windows 9

Telling Git who you are: git config 10

Getting help: git help 12

Creating your GitHub account 12

Unit 1 summary 13

Chapter 2 · Getting Started with Git 14

Your terminal survival kit 14

Local and remote repositories 15

Creating a repository: git init 16

The three stages 17

Checking where things are: git status 18

Staging changes: git add 19

Saving a snapshot: git commit 20

Your first repository on GitHub 21

Uploading: git remote and git push 22

Downloading a project: git clone 23

Getting your team's work: git fetch and git pull 24

The daily loop 25

Chapter 3 · Branching and Merging 27

Why branches? 27

What a branch really is 28

Creating and listing branches 28

Switching branches 29

Merging: fast-forward 30

Three-way merges 31

Merge conflicts 32

Practise resolving conflicts 33

Deleting branches 34

Branches on GitHub 34

Team branching workflow 35

Unit 3 summary 36

Chapter 4 · Advanced Git Commands 37

The ₹0 mystery 37
Reading history: git log 38
Inspecting one commit: git show 39
Seeing changes: git diff 40
Parking work: git stash 41
Marking releases: git tag 42
Undoing changes: which command? 43
Discarding edits: git restore (and checkout --) 44
Moving back in time: git reset 45
Undoing a pushed commit: git revert 46
Visiting the past: detached HEAD 46
Unit 4 summary 47

Chapter 5 · Introduction to GitHub 48
Going public 48
A tour of a repository page 49
Your profile and finding projects 50
Creating a repository on GitHub 50
Managing a repository 51
Issues: tracking work 52
Pull requests: proposing changes 53
Forking: contributing to other projects 54
Open source etiquette 56
Your Git cheat sheet 56
Unit 5 summary 57

Appendix A · Command cheat sheet 58

Appendix B · Glossary 61

Appendix C · Practise online 64

Appendix D · Course at a glance 65

CHAPTER 1 · UNIT 1 · 4 HOURS

Introduction to Version Control

Why version control exists, the three kinds of VCS, and setting up Git and GitHub on Windows.

Local, centralized & distributed VCS | Benefits of Git & GitHub | Installing Git on Windows | git config | git help | Creating a GitHub account

Practise online: missions M-01, M-02

The Amroli Canteen problem

Meet four BCA final-year students at SBCCAS. For their project they are building **Amroli Canteen**, a web app that lets students order samosas and chai *before* the lunch break, so nobody waits in the queue.

Student	Builds
Krish	the menu and cart pages
Dhruvi	the order system in Python
Meet	the database
Hetvi	team lead: plans, reviews and writes the README

In the first two weeks they share code the way many students do: WhatsApp, pen drives and email. Here is what happened.

1 · Lost work

Krish's login page worked on Monday. On Tuesday he "improved" it and broke it. Monday's version is gone.

With Git

Every commit is a saved snapshot. Krish can go back to Monday's version in one command.

2 · File-name chaos

`menu.html`, `menu_v2.html`, `menu_FINAL.html`, `menu_FINAL_use_this.html` …

With Git

One file name. History keeps every version, with a message saying what changed.

3 · Overwritten work

Dhruvi and Krish both edited `app.py` and sent it on WhatsApp. Whoever sent last "won".

With Git

Changes are merged. If two people edit the same line, Git stops and asks a human to decide.

4 · Who broke it?

The total price shows ₹0. Nobody knows when it broke or why.

With Git

The log shows who changed what, when and why. You can find the exact change.

5 · No backup

Meet's laptop crashed a week before the review. His database work was only on that laptop.

With Git + GitHub

Every teammate and GitHub hold a full copy. A crashed laptop costs minutes, not weeks.

Hetvi, the team lead, says: *"Professional teams don't work like this. They use **Git** and **GitHub**. Let's learn it properly."* That is exactly what you will do in this course.

What is version control?

A **version control system (VCS)** is software that records changes to files over time, so you can see who changed what and bring back any earlier version.

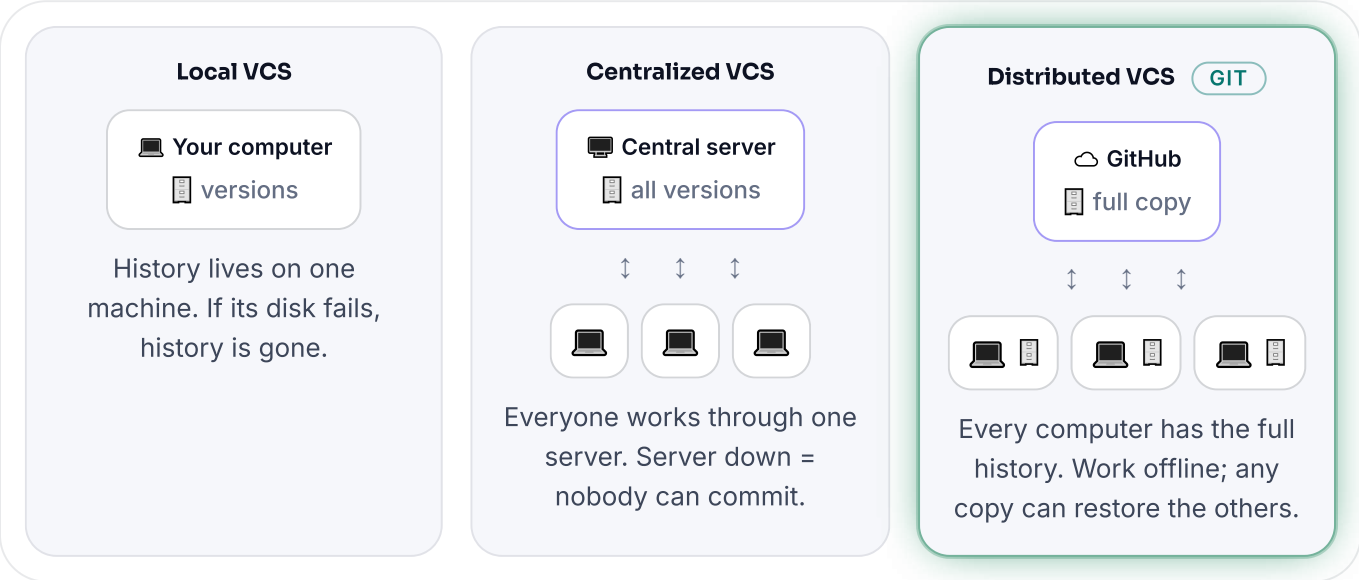
> 🧠 **A time machine for your project**
>
> Think of a video game that saves your progress at every checkpoint. If you fall into lava, you restart from the last checkpoint instead of from the beginning. A VCS gives your code the same superpower, and each checkpoint is called a **commit**.

A VCS answers four questions about every change:

- **What** changed? (exactly which lines)
- **Who** changed it?
- **When** was it changed?
- **Why**? (the message the author wrote)

Three kinds of version control

Version control has grown up in three stages.



1. Local VCS. Versions are kept in a small database on one computer. An early example is RCS (1982). *Like a personal diary in your room: useful, but only you can read it, and if the house floods it is gone.*

2. Centralized VCS. One central server holds all versions. Developers *check out* files from it and send changes back. Examples: CVS and Subversion (SVN). *Like one library*

book the whole class must share: if the library is closed, nobody can study. The server is a **single point of failure**.

3. Distributed VCS. Every developer has a **complete copy** of the project *and its whole history* on their own computer. Examples: **Git** and Mercurial. *Like every student owning a full photocopy of the book: if one copy is lost, any other copy can replace it.*

	Local	Centralized	Distributed (Git)
Where history lives	One computer	One server	Every computer
Work offline?	Yes	No	**Yes**
Team work	No	Yes	**Yes**
If one machine dies	History lost	Everyone stuck	**Nothing lost**

> ✨ **Why Git won**
>
> Git was created in 2005 by **Linus Torvalds**, who also created Linux, to manage the Linux kernel's thousands of contributors. It is fast, works offline and makes branching cheap. Today almost every software company uses it.

Benefits of Git and GitHub

Git A TOOL

💻 Installed on your computer

- Tracks every change in a folder
- Works offline
- Free, open source (2005, Linus Torvalds)
- Commands: `init`, `add`, `commit`, `log` ...

GitHub A WEBSITE

☁ github.com

- Stores Git repositories online
- Needs the internet
- Adds teamwork: issues, pull requests, forks
- Your public developer portfolio

Git works without GitHub. GitHub needs Git. Similar sites: GitLab, Bitbucket.

Git is the tool on your computer. **GitHub** is a website that hosts Git repositories so a team can share them. Together they give you:

- **Complete history**: every change, with who, when and why.
- **Safe experiments**: try an idea on a *branch* without breaking working code (Unit 3).
- **Teamwork**: many people work on the same project at the same time.
- **Undo**: go back to any earlier version in seconds (Unit 4).
- **Backup**: your code is safe on GitHub even if your laptop is not.
- **Career**: your GitHub profile is a portfolio that recruiters look at.

> ⚠ **Git ≠ GitHub**
>
> A common exam mistake is writing "GitHub is a version control system". **Git** is the version control system. **GitHub** is a hosting service built around Git. GitLab and Bitbucket are similar services.

Installing Git on Windows

The SBCCAS labs use Windows, so here is the Windows installer step by step. (On macOS, Git comes with the Xcode tools; on Ubuntu, run `sudo apt install git`.)

1. Open **git-scm.com** and click **Download for Windows**. Choose the 64-bit installer.
2. Run the installer and click **Next** on the licence screen.
3. **Select Components:** keep the defaults.
4. **Choosing the default editor:** change *Vim* to **Notepad** or **Visual Studio Code**. (Vim confuses beginners. If it ever opens, type `:wq` and press Enter to leave.)
5. **Adjusting the name of the initial branch:** choose **Override the default branch name** and type `main`. GitHub uses `main` too.
6. **Adjusting your PATH:** keep **Git from the command line and also from 3rd-party software** (recommended).
7. **SSH executable:** keep *Use bundled OpenSSH*.
8. **HTTPS transport backend:** keep *Use the OpenSSL library*. (If the college network blocks GitHub, your lab admin may choose *Windows Secure Channel*.)

9. **Line ending conversions:** keep *Checkout Windows-style, commit Unix-style line endings*.
10. **Terminal emulator:** keep *Use MinTTY*.
11. **Default behaviour of `git pull`:** keep *Fast-forward or merge*.
12. **Credential helper:** keep **Git Credential Manager**. It opens a browser window to sign in to GitHub.
13. Keep the remaining defaults and click **Install**, then **Finish**.

Now open **Git Bash** from the Start menu and check the installation:

```
Git Bash

$ git --version
git version 2.47.1.windows.1
```

Your number may be newer. Any version 2.x is fine.

> 💡 **No laptop? No problem.**
>
> Everything in Units 1–4 can also be practised in this site's Playground, which runs real Git inside your browser, even on a phone.

Telling Git who you are: `git config`

Every commit records an **author name** and **email**. Before your first commit, tell Git who you are. You only need to do this once per computer:

```
Git Bash

$ git config --global user.name "Krish Patel"
$ git config --global user.email "krish@example.com"
```

Check your settings:

Git Bash

```
$ git config --list
init.defaultbranch=main
user.name=Krish Patel
user.email=krish@example.com
$ git config user.name
Krish Patel
```

Option	Where the setting is saved	Applies to
`--global`	`~/.gitconfig` in your home folder	every repository for your user account
(no option, inside a repository)	`.git/config`	only this repository

> ⚠️ **Use the same email as GitHub**
>
> Use the email address you will use for your GitHub account. GitHub uses it to link your commits to your profile and to show the green squares on your contribution graph.

> 📝 **Shared lab computers**
>
> On a shared lab PC, set your name **without** `--global` inside your own project folder, so the next student's commits are not labelled with your name.

If you forget to configure Git, your first commit stops with this message. Now you know how to fix it:

```
Author identity unknown

*** Please tell me who you are.

Run
```

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

Getting help: `git help`

Git has its own manual built in. There are three ways to ask it for help:

Command	What you get
`git help`	the list of the most common Git commands
`git help <command>`	the full manual page for one command, e.g. `git help commit`
`git <command> -h`	a short summary of options, right in the terminal

Git Bash

```
$ git commit -h
usage: git commit [-a] [-m <msg>] [--amend] [--allow-empty]

    -m <msg>      use the given message
    -a, --all     stage modified and deleted tracked files first
    --amend       replace the last commit
```

> 💡 **Read the error messages**
>
> Git's messages are unusually helpful. They often tell you the exact command to run next. Before searching the internet, read what Git printed.

Creating your GitHub account

1. Go to **github.com** and click **Sign up**.
2. Enter your **email**, the same one you used in `git config`.
3. Create a strong **password**.
4. Choose a **professional username**: `krish-patel` or `krishpatel-dev`, not `coolboy007`. It becomes your portfolio address: `github.com/krish-patel`.
5. Solve the puzzle, then type the **code GitHub emails you**.

6. Choose the **Free** plan.
7. Turn on **two-factor authentication (2FA)**: *Settings → Password and authentication → Enable two-factor authentication*. Use an authenticator app or the GitHub Mobile app, and **save your recovery codes** somewhere safe. GitHub requires 2FA for everyone who contributes code.
8. Add a profile photo, your name, and "BCA student at SBCCAS, Surat" in your bio.

> 🧪 **Lab L-01: your profile README**
>
> Create a repository named **exactly the same as your username** (for example `krish-patel/krish-patel`) and tick *Add a README file*. Whatever you write in that README appears on your GitHub profile page, like a business card. The full steps are on the Labs page.

Unit 1 summary

- A **version control system** records every change: what, who, when and why.
- **Local** VCS keeps history on one machine; **centralized** VCS on one server; **distributed** VCS (Git) on every computer.
- **Git** is the tool on your computer; **GitHub** is the website that hosts repositories for teams.
- On Windows, install **Git for Windows**, and set the default branch to `main` and the editor to Notepad or VS Code.
- `git config --global user.name` and `user.email` tell Git who you are.
- `git help`, `git help <command>` and `git <command> -h` are your built-in manual.
- Create a GitHub account with a professional username and **turn on 2FA**.

Next, practise with **Missions M-01 and M-02** below, then take the Unit 1 quiz.

CHAPTER 2 · UNIT 2 · 8 HOURS

Getting Started with Git

Local and remote repositories, the three stages, and the everyday commands — from your first commit to your first push.

Local vs remote repositories | init · status · add · commit | push · clone | pull · fetch | Your first repository

Practise online: missions M-03, M-04, M-05, M-06, M-07

Your terminal survival kit

Git is used from a **terminal**: on Windows that is **Git Bash**, on a Mac the Terminal app. You only need a handful of commands to move around:

Command	Does	Example
`pwd`	**p**rint **w**orking **d**irectory: where am I?	`pwd` → `/home/student`
`ls`	list files (`ls -a` also shows hidden ones like `.git`)	`ls -a`
`cd`	change directory (`cd ..` = up one level, `cd ~` = home)	`cd amroli-canteen`
`mkdir`	make a directory (folder)	`mkdir amroli-canteen`
`touch`	create an empty file	`touch notes.txt`
`echo "…" > file`	write text into a file (`>>` adds to the end)	`echo "Samosa 15" > menu.txt`
`cat`	show a file	`cat menu.txt`

```
Git Bash

$ pwd
/home/student
$ mkdir amroli-canteen
$ cd amroli-canteen
```

> **The number one beginner mistake**
>
> Running Git in the wrong folder. If Git says `fatal: not a git repository`, run `pwd` and `cd` into your project. The prompt shows your folder: `~/amroli-canteen (main)`.

Local and remote repositories

A **repository** (or *repo*) is a project folder that Git tracks: your files plus their complete history.



	Local repository	Remote repository
Lives on	your computer (the `.git` folder)	a server such as GitHub
Needs internet?	No	Yes, to sync
Who can see it?	only you	your team, or the whole world if public
If your laptop dies	gone	**safe**
You send work with	n/a	`git push`
You get work with	n/a	`git clone`, `git fetch`, `git pull`

> 🧠 **Your notebook and the class notice board**
>
> Your local repo is your own notebook: you can write in it anywhere, even with no network. The remote repo is the class notice board: everyone reads from it, and you **pin** (push) your work there when it is ready.

By convention the main remote is called `origin`.

Creating a repository: `git init`

Git Bash

```
$ cd ~/amroli-canteen
$ git init
Initialized empty Git repository in /home/student/amroli-canteen/.git/
$ ls -a
./  ../  .git/
```

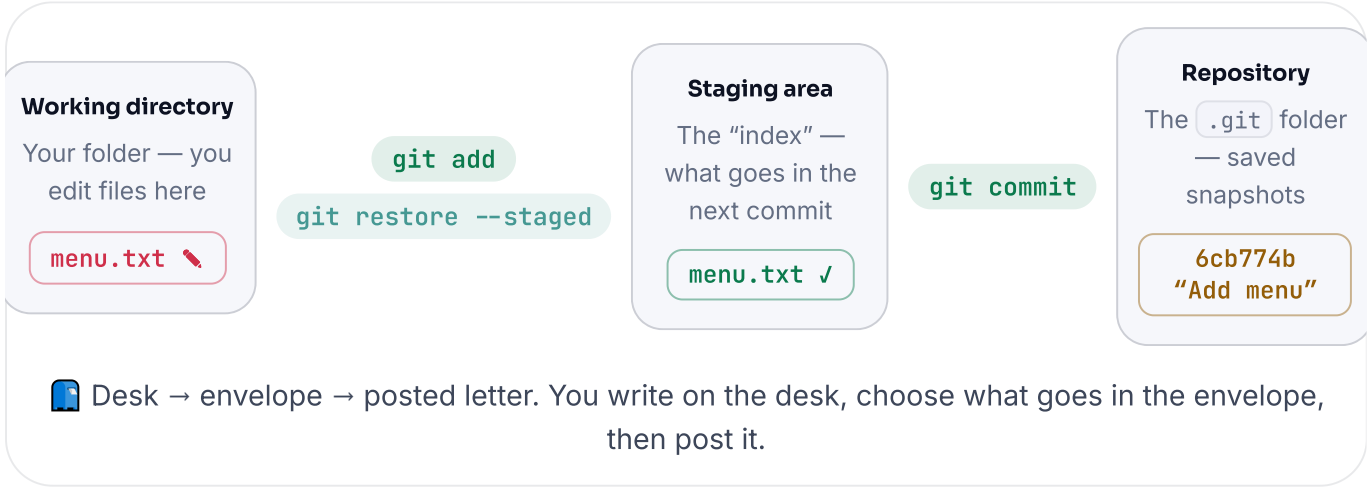
`git init` creates one hidden folder, `.git`. That folder *is* the repository: all history, branches and settings live inside it. Delete `.git` and the folder becomes an ordinary folder again.

```
amroli-canteen/
├── menu.txt              ← your files (working directory)
└── .git/                 ← the repository — Git manages this, don't edit it
    ├── HEAD              ← which branch you are on
    ├── config            ← settings for this repository
    ├── objects/          ← every saved version of every file
    ├── refs/             ← branch and tag names
    ├── hooks/            ← optional scripts
    └── info/
```

> ⚠️ **Never edit .git by hand**
>
> Look inside if you are curious (`ls .git`), but let Git manage it. Also, never run `git init` in your home folder or on your Desktop: only inside a project folder.

The three stages

This is **the most important idea in Git**. Every change passes through three places:



Desk → envelope → posted letter. You write on the desk, choose what goes in the envelope, then post it.

1. **Working directory**: the files you see and edit.
2. **Staging area** (also called the *index*): the changes you have *chosen* for the next snapshot.
3. **Repository**: the snapshots (**commits**) saved permanently in `.git`.

> 🧠 **The post office**
>
> You write letters at your **desk** (working directory). You put the ones that are ready into an **envelope** (staging area); you don't have to send every letter you wrote. When you **post** the envelope (commit), it is sent and recorded.

Why have a staging area at all? Because real work is messy. You may fix a bug *and* start a new feature in the same hour. Staging lets you commit them as two clean, separate commits.

Checking where things are: `git status`

`git status` is the command you will type most often. Run it before and after everything.

Git Bash

```
$ echo "Samosa 15" > menu.txt
$ git status
On branch main

No commits yet

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

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

Untracked means Git sees the file but has never saved it. Notice that Git even tells you what to do next.

What status says	Meaning	Colour in Git Bash
Untracked files	new file Git has never saved	red
Changes not staged for commit	a saved file was edited, but not staged	red
Changes to be committed	staged, ready for the next commit	green
nothing to commit, working tree clean	everything is saved	n/a

There is also a short format: `git status -s` prints `??` for untracked, `M` for modified (left column = staged, right column = not staged).

Staging changes: `git add`

Git Bash

```
$ git add menu.txt
$ git status
On branch main

No commits yet

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

Ways to stage:

Command	Stages
`git add menu.txt`	one file
`git add menu.txt cart.js`	several files
`git add *.css`	every `.css` file in this folder
`git add .`	**everything** changed in this folder and below

Staged something by mistake? Unstage it, and your edit stays safe in the file:

```
Git Bash

$ git add .
$ git restore --staged menu.css
$ git status -s
M  menu.txt
?? menu.css
```

Saving a snapshot: `git commit`

```
Git Bash

$ git commit -m "Add menu"
[main (root-commit) 6cb774b] Add menu
 1 file changed, 1 insertion(+)
 create mode 100644 menu.txt
```

Read that output: on branch **main**, the very first (**root**) commit, its short ID **6cb774b**, your message, and what changed. Every commit gets a unique 40-character ID (a SHA-1 hash). Git usually shows the first seven characters.

```
Git Bash

$ git log
commit 6cb774b24074eb78d807a9a2420e109ab07b3a89 (HEAD → main)
Author: Krish Patel <krish@example.com>
Date:   Thu Oct 22 22:55:39 2026 +0530

    Add menu
```

Krish edits the menu and adds a stylesheet. He commits them **separately**, because they are two different changes:

Git Bash

```
$ echo "Vada Pav 20" >> menu.txt
$ git diff
diff --git a/menu.txt b/menu.txt
index 4b00757..b0be448 100644
--- a/menu.txt
+++ b/menu.txt
@@ -1 +1,2 @@
 Samosa 15
+Vada Pav 20
$ git add menu.txt
$ git commit -m "Add vada pav to menu"
[main b0d4373] Add vada pav to menu
 1 file changed, 1 insertion(+)
$ git add menu.css
$ git commit -m "Add menu styles"
[main 92bd5b5] Add menu styles
 1 file changed, 1 insertion(+)
 create mode 100644 menu.css
$ git log --oneline
92bd5b5 (HEAD → main) Add menu styles
b0d4373 Add vada pav to menu
6cb774b Add menu
```

💡 **Good commit messages**

Write what the commit *does*, starting with a verb: **"Add vada pav to menu"**, **"Fix total when cart is empty"**. Not "update", "changes" or "final". Your future self, and your teammates, read these messages to understand the history.

📝 **Shortcut:** `git commit -am "message"`

`-a` stages every **modified tracked** file before committing. It does **not** add new (untracked) files; those still need `git add`.

Your first repository on GitHub

There are two ways to connect a project to GitHub. Both end in the same place.

	Method 1: local first	Method 2: GitHub first
Start with	a folder on your computer (`git init`)	a new repo on github.com
On GitHub, create the repo…	**empty**: don't add a README	**with** a README
Then run	`git remote add origin <url>` and `git push -u origin main`	`git clone <url>`
Best when	you already have code	you are starting fresh

To create a repo on GitHub: click **+** (top right) → **New repository** → type a name → choose Public or Private → **Create repository**. GitHub then shows the exact commands to run.

> ⚠ **"refusing to merge unrelated histories"**
>
> This error appears if you create the GitHub repo **with** a README *and* also run `git init` locally, so the two have different first commits. Pick one method. (The fix is `git pull origin main --allow-unrelated-histories`.)

Uploading: `git remote` and `git push`

Krish uses Method 1. He created an empty repo on GitHub, so he connects his local repo to it and pushes:

```
Git Bash

$ git remote add origin https://github.com/krish-patel/amroli-canteen.git
$ git remote -v
origin  https://github.com/krish-patel/amroli-canteen.git (fetch)
origin  https://github.com/krish-patel/amroli-canteen.git (push)
$ git push -u origin main
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Writing objects: 100% (9/9), done.
Total 9 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/krish-patel/amroli-canteen.git
 * [new branch]      main → main
branch 'main' set up to track 'origin/main'.
```

- `origin` is the nickname for the GitHub URL.
- `-u` (set **u**pstream) links your `main` to `origin/main`. From now on, plain `git push` and `git pull` know where to go.

> 📝 **Signing in**
>
> On a real computer the first push opens a browser window from **Git Credential Manager** to sign in to GitHub. GitHub no longer accepts your account password typed in the terminal. In this site's Playground the GitHub is simulated, so no sign-in is needed.

Downloading a project: `git clone`

Hetvi created a repo for project notes. Dhruvi wants a copy on her laptop:

```
Git Bash

$ git clone https://github.com/hetvi-desai/canteen-docs.git
Cloning into 'canteen-docs'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
$ cd canteen-docs
$ git remote -v
origin  https://github.com/hetvi-desai/canteen-docs.git (fetch)
origin  https://github.com/hetvi-desai/canteen-docs.git (push)
```

`git clone` downloads **every file and the whole history**, and sets up `origin` for you. That's why you don't run `git init` or `git remote add` after cloning.

> ⚠ **Clone ≠ Download ZIP**
>
> GitHub's *Download ZIP* button gives you only the files: no history and no connection to GitHub. You can't push or pull from it.

Getting your team's work: `git fetch` and `git pull`

Overnight, Dhruvi pushed a new file, `orders.py`. Next morning Krish gets her work.

`git fetch`

📦 The parcel arrives at your door.

- Downloads new commits into `origin/main`
- Your branch and files do **not** change
- Look first: `git log main..origin/main`

`git pull`

📦➡🏠 The parcel is opened and unpacked.

- = `git fetch` + `git merge`
- Your branch moves forward
- Your files update

Look before you leap with `git fetch`:

Git Bash

```
$ git fetch
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/krish-patel/amroli-canteen
   92bd5b5..e423aad  main → origin/main
$ git status
On branch main
Your branch is behind 'origin/main' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
$ git log --oneline main..origin/main
e423aad (origin/main) Add orders list
```

Then bring it in with `git pull` (fetch + merge):

Git Bash

```
$ git pull
Updating 92bd5b5..e423aad
Fast-forward
 orders.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 orders.py
```

> ⚠ **"rejected ... (fetch first)"**
>
> If a teammate pushed before you, Git refuses your push. Never panic and never force it: run `git pull`, then `git push` again.

The daily loop

`git pull` → edit files → `git status` → `git add` → `git commit -m` → `git push`

- A **repository** is a folder plus its history in `.git`. `git init` creates one.
- Changes move **working directory** → **staging area** → **repository** with `git add` and `git commit`.
- `git status` tells you where every file is. Read it often.
- **Local first:** `git remote add origin <url>` then `git push -u origin main`. **GitHub first:** `git clone <url>`.
- `git fetch` downloads without changing your files; `git pull` downloads and merges.
- Commit small, commit often, and write messages that start with a verb.

Practise with **Missions M-03 to M-07**, then take the Unit 2 quiz.

CHAPTER 3 · UNIT 3 · 6 HOURS

Branching and Merging

Work in parallel with branches, bring work together with merges, and calmly resolve conflicts.

Introduction to branches | Creating & switching branches | Merging & resolving conflicts | Deleting branches

Practise online: missions M-08, M-09, M-10, M-11

Why branches?

The Amroli Canteen app works, and Hetvi has shown it to the canteen manager. Now Krish wants to try a **new menu design**, and Dhruvi wants to add **online payment**. Both changes will take days, and both might break things.

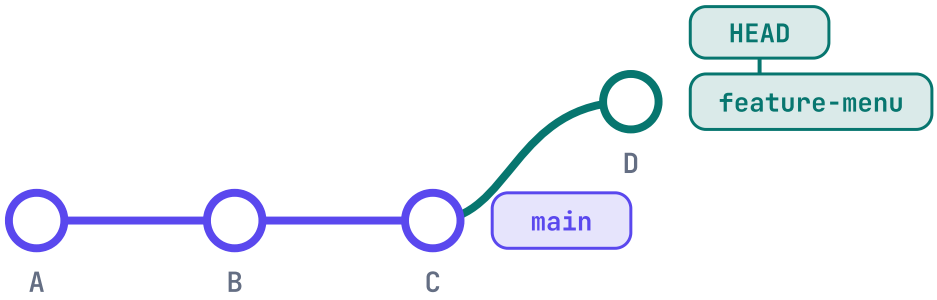
If they edit `main` directly, the working app is broken until they finish. If they don't, they can't save their progress. **Branches** solve this.

> 🧠 **Parallel universes**
>
> A branch is a parallel universe for your project. Krish can try his new design in his own universe. If it works, he brings it back to the main universe (a **merge**). If it fails, he deletes that universe, and `main` never even knew.

With branches:

- `main` always stays **working**, ready to demo.
- Each feature gets its **own branch**: `feature-menu`, `feature-payment`.
- Several people work **in parallel** without disturbing each other.
- Finished work is **merged** back into `main`.

What a branch really is



A branch is just a **movable label** on a commit. **HEAD** shows which branch you are on. Committing moves that label forward.

A branch is not a copy of your files. It is a tiny **label that points to one commit**, which is why creating a branch is instant.

- When you commit, the **current branch's label moves forward** to the new commit.
- **HEAD** is a special label that says *which branch you are on*. Git Bash shows it in the prompt: `~/amroli-canteen (feature-menu)`.

Creating and listing branches

Git Bash

```
$ git branch
* main
$ git branch feature-menu
$ git branch
  feature-menu
* main
```

- `git branch` lists branches. The `*` marks the one you are on.
- `git branch <name>` **creates** a branch at your current commit, but it does **not** switch to it.

> 💡 **Naming branches**
>
> Use short, descriptive, lowercase names with hyphens: `feature-menu`, `fix-cart-total`, `docs-readme`. No spaces.

Switching branches

Git Bash

```
$ git switch feature-menu
Switched to branch 'feature-menu'
$ echo "Vada Pav 20" >> menu.txt
$ git commit -am "Add vada pav to menu"
[feature-menu b97cb83] Add vada pav to menu
 1 file changed, 1 insertion(+)
$ git log --oneline --graph --all
* b97cb83 (HEAD -> feature-menu) Add vada pav to menu
* 8391a89 (main) Add opening time
* 3c16dcc Add menu
```

Now switch back to `main` and look at the menu:

Git Bash

```
$ git switch main
Switched to branch 'main'
$ cat menu.txt
Samosa 15
```

The *Vada Pav* line is gone. That's not lost work: it is safely on `feature-menu`.
Switching branches changes the files in your folder to match that branch.

Create **and** switch in one step:

New way (Git 2.23+)	Classic way	Does
`git switch <branch>`	`git checkout <branch>`	move to an existing branch
`git switch -c <new>`	`git checkout -b <new>`	create a branch and move to it

Both work. `checkout` is older and does many jobs (it also restores files), so newer Git added the clearer `switch`. You will see `checkout` in many tutorials, so know both.

> ⚠ **Commit before you switch**
>
> If you have **uncommitted** changes to a file that is different on the other branch, Git refuses to switch: *"Your local changes to the following files would be overwritten by checkout"*. Commit your work (or stash it, Unit 4) first.

Merging: fast-forward

Krish is happy with the new menu. To bring `feature-menu` into `main`, **switch to the branch that receives the changes**, then merge:

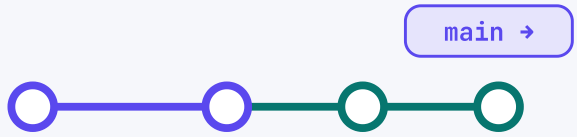
Git Bash

```
$ git switch main
$ git merge feature-menu
Updating 8391a89..b97cb83
Fast-forward
 menu.txt | 1 +
 1 file changed, 1 insertion(+)
```

Fast-forward means `main` had no new commits of its own, so Git simply moved the `main` label forward to Krish's commit. No new commit was needed.

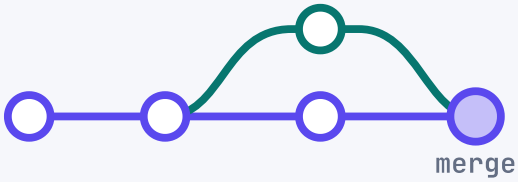
Three-way merges

Fast-forward



main had no new commits, so Git just **moves the label** forward. No merge commit.

Three-way merge



Both branches moved on, so Git joins them with a new **merge commit** that has two parents.

Meanwhile Krish built the cart on `feature-cart`, and someone also added tea to `main`. Now **both** branches have new commits:

Git Bash

```
$ git merge feature-cart
Merge made by the 'ort' strategy.
 cart.js | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 cart.js
$ git log --oneline --graph
*   de0d069 (HEAD -> main) Merge branch 'feature-cart'
|\
* | 3776e7d Add tea
| * 7c7c199 (feature-cart) Add cart
* | b97cb83 (feature-menu) Add vada pav to menu
|/
* 8391a89 Add opening time
* 3c16dcc Add menu
```

Git found the common ancestor (`8391a89`), combined both sets of changes, and created a **merge commit** with two parents. Read the graph from the top: the `|\` is where the history splits; the `|/` is where it started.

> 📝 **Opens an editor?**
>
> On a real computer, `git merge` may open your editor with the message *Merge branch 'feature-cart'*. Just save and close it to accept.

Merge conflicts

Sometimes Git cannot decide alone. Dhruvi changed the opening time to **8 AM** on her branch; on `main`, Krish changed the same line to **10 AM**.

Git Bash

```
$ git merge change-timing
Auto-merging timing.txt
CONFLICT (content): Merge conflict in timing.txt
Automatic merge failed; fix conflicts and then commit the result.
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

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

A **conflict** happens when two branches change **the same lines** of the same file. Git stops and asks a human to decide. That isn't an error: it is Git refusing to guess.

Git writes **both versions** into the file, between markers:

```
<<<<<<< HEAD
Open: 10 AM
=======
Open: 8 AM
>>>>>>> change-timing
```

- `<<<<<<< HEAD` starts **your version** (the branch you are on)
- `=======` divides the two versions
- `>>>>>>> change-timing` ends **their version** (the branch being merged)
- **To resolve:** keep the right text, delete all three marker lines, then `git add` and `git commit`

Resolving a conflict, step by step:

1. Open the file (`edit timing.txt` in the Playground, or your editor).
2. Decide the correct content. Keep one side, the other, or a mix.
3. **Delete all three marker lines.**
4. Mark it resolved: `git add timing.txt`.
5. Finish the merge: `git commit`.

Git Bash

```
$ echo "Open: 8 AM" > timing.txt
$ git add timing.txt
$ git status
On branch main
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   timing.txt
$ git commit --no-edit
[main e41ab0c] Merge branch 'change-timing'
```

> 💡 **The escape hatch**
>
> Changed your mind halfway? `git merge --abort` puts everything back exactly as it was before the merge.

Practise resolving conflicts

Tap lines to delete them until each file matches what the team decided, with no markers left.

Loading the conflict game...

Deleting branches

After a branch is merged, its label is no longer needed. Deleting it keeps `git branch` tidy (the commits stay in `main`'s history):

Git Bash

```
$ git branch -d feature-menu
Deleted branch feature-menu (was b97cb83).
```

Git protects you from deleting work that was **never merged**:

Git Bash

```
$ git branch -d experiment
error: the branch 'experiment' is not fully merged
hint: If you are sure you want to delete it, run 'git branch -D experiment'
hint: Disable this message with "git config set advice.forceDeleteBranch false"
$ git branch -D experiment
Deleted branch experiment (was 84079e6).
```

Command	Deletes
`git branch -d <name>`	a **merged** branch (safe)
`git branch -D <name>`	any branch, **even unmerged** (its unique commits are abandoned)
`git push origin --delete <name>`	the branch **on GitHub**

You can't delete the branch you are standing on; switch away first.

Branches on GitHub

Branches can be pushed like `main`, so teammates can see or review them:

```
Git Bash

$ git push -u origin feature-payment
To https://github.com/krish-patel/amroli-canteen.git
 * [new branch]      feature-payment → feature-payment
branch 'feature-payment' set up to track 'origin/feature-payment'.
$ git branch -a
  change-timing
  feature-cart
* feature-payment
  main
  remotes/origin/feature-payment
  remotes/origin/main
```

`git branch -a` shows **all** branches, including the remote-tracking ones (`remotes/origin/…`).

Team branching workflow

Hetvi's four rules for the Amroli Canteen team:

1. **Never commit directly to `main`.** `main` must always work.
2. **One feature = one branch:** `feature-cart`, `fix-total`.
3. **Pull before you start**, so you branch from the latest `main`.
4. **Merge small and often.** Small branches mean small conflicts.

```
Git Bash

$ git switch main
$ git pull
$ git switch -c feature-search
$ git add .
$ git commit -m "Add menu search box"
$ git switch main
$ git merge feature-search
$ git branch -d feature-search
```

In Unit 5 you'll replace the local merge with a **pull request** on GitHub, so a teammate reviews the work first.

Unit 3 summary

- A **branch** is a movable label on a commit; **HEAD** marks the branch you are on.
- `git branch` lists/creates; `git switch` (or `checkout`) moves; `-c` / `-b` create and move.
- Merge **into** the branch you are on: `git switch main` then `git merge feature-x`.
- **Fast-forward** just moves the label; a **three-way merge** creates a merge commit.
- A **conflict** = both sides changed the same lines. Edit, remove markers, `git add`, `git commit` (or `git merge --abort`).
- `git branch -d` deletes merged branches; `-D` forces; `git push origin --delete` removes it from GitHub.

Practise with **Missions M-08 to M-11**, then take the Unit 3 quiz.

CHAPTER 4 · UNIT 4 · 6 HOURS

Advanced Git Commands

Read history like a detective, park work with stash, mark releases with tags, and undo mistakes safely.

log · diff · show | stash | Tags | reset | Undoing: revert, reset, checkout

Practise online: missions M-12, M-13, M-14, M-15

The ₹0 mystery

A week before the Spectrum 2026 tech fest, Mr. Desai, the canteen manager, phones Hetvi: *"The cart says the total is ₹0!"* Nobody remembers touching the cart code. In Unit 1 this would have been a disaster. With Git, it's a detective story: the history records **every change, who made it, when and why**.

In this unit you will learn to:

- **read** history (`git log`, `git show`, `git diff`),
- **park** unfinished work (`git stash`),
- **mark** releases (`git tag`),
- and **undo** mistakes safely (`git restore`, `git reset`, `git revert`).

Reading history: `git log`

```
Git Bash

$ git log
commit be6e58670ebd8d0c94d25d66a5ca7b213c10e50c (HEAD → main)
Author: Krish Patel <krish@example.com>
Date:   Fri Oct 23 06:18:46 2026 +0530

    Add tea to menu

commit 03474ce19d00da77f18110754b73539ef13411e2
Author: Dhruvi Patel <dhruvi@example.com>
Date:   Fri Oct 23 06:18:46 2026 +0530

    Refactor cart total
```

The full log is long; these options make it useful:

Command	Shows
`git log --oneline`	one line per commit
`git log -n 3` or `git log -3`	only the last 3 commits
`git log --oneline --graph --all`	every branch, drawn as a graph
`git log --author="Dhruvi"`	commits by one person
`git log --grep="cart"`	commits whose **message** contains "cart"
`git log -- cart.js`	commits that **changed** `cart.js`
`git log --stat`	which files each commit changed

The detective work begins. Which commits touched the cart?

```
Git Bash

$ git log --oneline
be6e586 (HEAD → main) Add tea to menu
03474ce Refactor cart total
63a4905 Add vada pav
1d297a5 Add menu
$ git log --oneline -- cart.js
03474ce Refactor cart total
```

> 💡 **Press q to quit**
>
> If the log is longer than the screen, Git Bash shows it page by page. Use ↑ ↓ or Space to scroll and **q** to quit.

Inspecting one commit: `git show`

```
Git Bash

$ git show HEAD~1
commit 03474ce19d00da77f18110754b73539ef13411e2
Author: Dhruvi Patel <dhruvi@example.com>
Date:   Fri Oct 23 06:18:46 2026 +0530

    Refactor cart total

diff --git a/cart.js b/cart.js
new file mode 100644
index 0000000..0cd9610
--- /dev/null
+++ b/cart.js
@@ -0,0 +1 @@
+total = 0
```

Found it: `total = 0`. Now the team knows **what** changed, **who** changed it and **when**, and can ask Dhruvi **why**.

Naming commits. You can refer to a commit by its short hash (`03474ce`), a branch or tag name (`main`, `v1.0`), or relative to HEAD:

Name	Means
`HEAD`	the commit you are on now
`HEAD~1`	one commit before HEAD (its parent)
`HEAD~3`	three commits back
`main~2`	two commits before the tip of `main`

`git show HEAD:menu.txt` prints a file **as it was** in that commit.

Seeing changes: `git diff`

`git diff` shows **line-by-line** differences. Lines starting with `-` were removed, `+` were added.

Git Bash

```
$ echo "Coffee 20" >> menu.txt
$ git diff
diff --git a/menu.txt b/menu.txt
index 49bfe2c..ef592dd 100644
--- a/menu.txt
+++ b/menu.txt
@@ -1,3 +1,4 @@
 Samosa 15
 Vada Pav 20
 Tea 10
+Coffee 20
$ git add menu.txt
$ git diff
$ git diff --staged
diff --git a/menu.txt b/menu.txt
index 49bfe2c..ef592dd 100644
--- a/menu.txt
+++ b/menu.txt
@@ -1,3 +1,4 @@
 Samosa 15
 Vada Pav 20
 Tea 10
+Coffee 20
```

Notice that after `git add`, plain `git diff` shows **nothing**: the change moved to the staging area. Each form of `diff` compares two of the three stages:

Command	Compares
`git diff`	working directory ↔ staging area (unstaged changes)
`git diff --staged`	staging area ↔ last commit (what you're about to commit)
`git diff main feature-cart`	two branches
`git diff HEAD~2 HEAD`	two commits (add `--stat` for a summary)

Parking work: `git stash`

Krish is halfway through the cart when Hetvi messages: *"The samosa price is wrong on the live demo. Fix it NOW!"* His cart code is broken and not ready to commit.

> 🧠 **Saving your game mid-level**
>
> `git stash` is like pausing a game and saving it in a slot. Your half-done changes are put in a drawer, the working tree becomes clean, you fix the urgent bug, then you take your changes back out exactly as they were.

Git Bash

```
$ git status -s
 M cart.js
$ git stash
Saved working directory and index state WIP on main: 88f8fde Add coffee
$ git status
On branch main

nothing to commit, working tree clean
$ git stash list
stash@{0}: WIP on main: 88f8fde Add coffee
```

Now fix the urgent bug and commit it, then bring the cart work back:

Git Bash

```
$ git stash pop
On branch main

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   cart.js

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (37aff8fdb12a0f05787c4165c766f0ca3261096d)
```

Command	Does
`git stash` / `git stash push -m "msg"`	save changes and clean the working tree
`git stash list`	list saved stashes (`stash@{0}` is the newest)
`git stash pop`	bring back the newest stash **and remove it** from the list
`git stash apply`	bring it back but **keep** it in the list
`git stash drop`	delete a stash without applying it

> 📝 **Untracked files are not stashed**
>
> By default, `git stash` saves changes to **tracked** files. Brand-new files you never added stay where they are.

Marking releases: `git tag`

On the day of the tech fest, Hetvi wants a permanent name for "the version we demonstrated". Commit hashes are hard to remember, and branches move. **Tags don't move.**

Git Bash

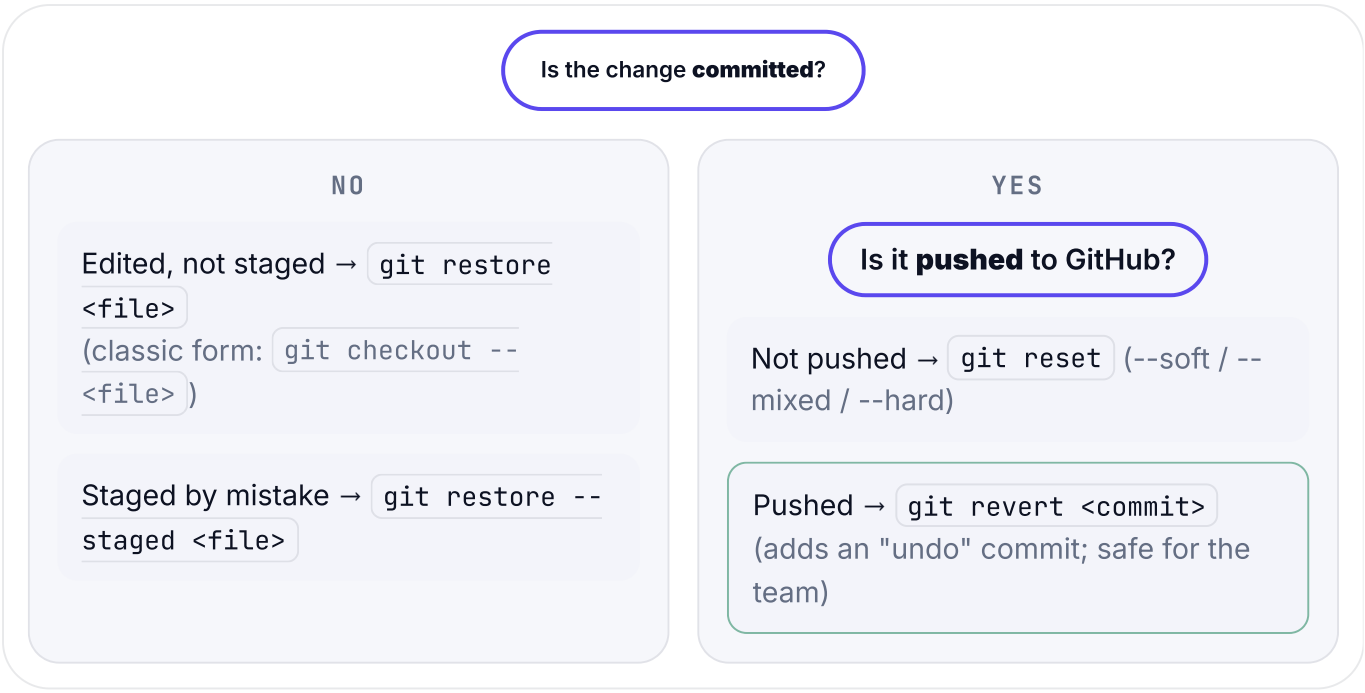
```
$ git tag v1.0
$ git tag -a v1.1 -m "Menu and cart released for Spectrum 2026"
$ git tag
v1.0
v1.1
$ git log --oneline -3
07f793e (HEAD → main, tag: v1.0, tag: v1.1) Finish cart
26ff36d Fix samosa price
88f8fde Add coffee
```

Kind	Command	Stores
Lightweight	`git tag v1.0`	just a name for the commit
Annotated (for releases)	`git tag -a v1.1 -m "…"`	name + message + who tagged it + date

`git show v1.1` shows the tag's message and the commit. Tags are **not pushed automatically**: use `git push origin v1.1`, or `git push origin --tags` for all of them. Delete a tag with `git tag -d v1.0`.

Undoing changes: which command?

Git has several "undo" commands. Choosing the right one depends on two questions:



> ⚠ **The golden rule of undo**
>
> **Not pushed yet?** `reset` is fine: you're only rewriting your own history. **Already pushed?** Use `revert`. It adds a new commit and never rewrites history your team already has.

Discarding edits: `git restore` (and `checkout --`)

You edited a file, it's a mess, and you want the last committed version back:

Git Bash

```
$ echo "oops" >> menu.txt
$ git restore menu.txt
$ git status -s
```

The classic command does the same thing: `git checkout -- menu.txt`. (The `--` means "what follows is a file, not a branch".)

> ⚠ **No undo for restore**
>
> `git restore <file>` permanently throws away the **uncommitted** edits in that file. Git never saved them, so it can't bring them back.

Moving back in time: `git reset`

`git reset` moves the current branch back to an earlier commit. The three modes decide what happens to the changes from the commits you "undo":

Command	Moves the branch back	Staging area	Your files
`git reset --soft HEAD~1`	✓	changes stay **staged**	kept
`git reset HEAD~1` (--mixed)	✓	changes become **unstaged**	kept
`git reset --hard HEAD~1`	✓	cleared	⚠ changes **deleted**

Git Bash

```
$ echo "Dosa 40" >> menu.txt && git commit -qam "add dosa"
$ git reset --soft HEAD~1
$ git status -s
M  menu.txt
$ git commit -qm "Add dosa to menu"
$ git reset HEAD~1
Unstaged changes after reset:
M       menu.txt
$ git commit -qam "Add dosa to menu"
$ git reset --hard HEAD~1
HEAD is now at 07f793e Finish cart
```

- `--soft`: perfect for fixing a bad commit message or adding a forgotten file. Changes stay **staged**.

- `--mixed` (the default): undo the commit and decide again what to stage.
- `--hard`: throw the commit **and** its changes away. ⚠️ Use with care.

`git reset <file>` (without a mode) simply **unstages** a file, like `git restore --staged`.

Undoing a pushed commit: `git revert`

Krish pushed *Finish cart*, and it broke the live demo. His teammates have already pulled it, so rewriting history with `reset` would cause chaos. Instead he **reverts** it:

Git Bash

```
$ git revert HEAD --no-edit
[main c47f6d3] Revert "Finish cart"
 1 file changed, 1 deletion(-)
$ git log --oneline -3
c47f6d3 (HEAD → main) Revert "Finish cart"
07f793e (tag: v1.0, tag: v1.1) Finish cart
26ff36d Fix samosa price
```

`git revert <commit>` creates a **new commit that does the opposite** of the old one. The history stays honest: anyone can see the mistake *and* its fix. Without `--no-edit`, Git opens the editor with a ready-made message.

	`reset`	`revert`
How	moves the branch back	adds a new "opposite" commit
History	rewritten (commits disappear)	kept (nothing disappears)
Safe after pushing?	⚠️ No	✅ Yes

Visiting the past: detached HEAD

`git checkout <commit>` lets you **look at** an old version of the whole project:

```
Git Bash

$ git checkout HEAD~2
Note: switching to 'HEAD~2'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.
…
HEAD is now at 26ff36d Fix samosa price
$ git switch main
Switched to branch 'main'
```

Detached HEAD means HEAD points to a commit instead of a branch. Look around, run the old code, then `git switch main` to come back. If you want to keep work done there, create a branch first: `git switch -c old-idea`.

> 💡 **Time travel in the Playground**
>
> In the Playground, open the **Graph** tab and press ⏳ **Time travel** to slide through your history and read any file at any commit, without changing anything.

Unit 4 summary

- `git log` (with `--oneline`, `--graph --all`, `-n`, `--author`, `--grep`, `--file`) reads history; `git show` inspects one commit.
- `git diff` = unstaged; `git diff --staged` = staged; `git diff A B` = two commits or branches.
- `git stash` parks work; `git stash pop` brings it back.
- `git tag -a v1.0 -m "…"` marks a release; push tags explicitly.
- Undo: `restore` (uncommitted edits), `reset` (local commits: soft / mixed / hard), `revert` (pushed commits).
- `git checkout <commit>` visits the past (detached HEAD); `git switch main` returns.

Practise with **Missions M-12 to M-15**, then take the Unit 4 quiz.

CHAPTER 5 · UNIT 5 · 4 HOURS

Introduction to GitHub

Find your way around GitHub, manage repositories, use issues, and contribute with forks and pull requests.

Navigating the GitHub interface | Creating & managing repositories | Issues & pull requests | Forking repositories | Git cheat sheet

Practise online: GitHub labs L-01, L-02, L-03, L-04

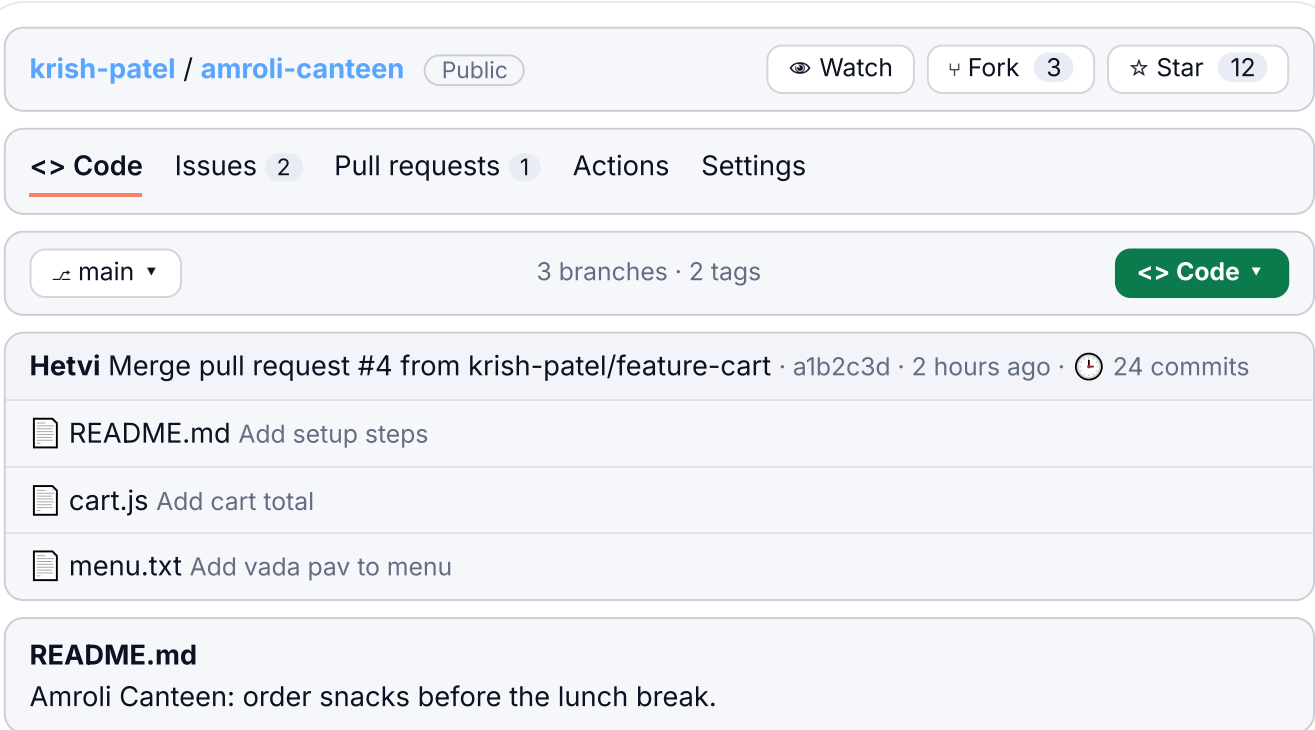
Going public

The Amroli Canteen app is a success. Mr. Desai wants to report bugs directly, juniors from another college want to use the code for *their* canteen, and Hetvi wants every change reviewed before it reaches `main`.

So far you have used GitHub as a place to **store** commits. In this unit you'll use it as a place to **work together**: browsing projects, managing repositories, tracking work with **issues**, reviewing changes with **pull requests**, and contributing to other people's projects with **forks**.

> 📝 **GitHub's design changes**
>
> GitHub updates its website regularly. Buttons may move or look slightly different from these notes, but the ideas (repositories, issues, pull requests, forks) stay the same.

A tour of a repository page



1. **Owner / repository** and whether it is public or private
2. **Watch · Fork · Star**: follow, copy to your account, bookmark
3. **Tabs**: Code, Issues (tasks and bugs), Pull requests (proposed changes), Actions (automation), Settings (owners only)
4. **Branch menu** to view any branch; counts of branches and tags
5. **Code button**: the clone URL (HTTPS or SSH) and Download ZIP
6. **Latest commit** and the full commit history
7. **README.md** is shown below the files: the project's front page

Some useful things to know:

- Press `t` on a repository page to search for a file, and `.` to open it in a web code editor.
- Click 🕒 **commits** to see the history (like `git log`), and click any commit to see its diff (like `git show`).
- Open a file and choose **History** (its commits) or **Blame** (who last changed each line).
- The **branch menu** lets you look at any branch without cloning.

Your profile and finding projects

Your profile page (`github.com/<username>`) shows your profile README (Lab L-01), pinned repositories and the **contribution graph**: the green squares that grow with every commit linked to your email.

Action	What it does
⭐ **Star**	bookmark a project and show appreciation; stars measure popularity
👁 **Watch**	get notifications about a repository's activity
⑂ **Fork**	make your own copy of someone else's repository (below)
🔍 **Search / Explore**	find projects by topic, language or name

> 💡 **Your portfolio**
>
> Recruiters look at GitHub profiles. Pin your best three repositories, give each a clear README, and commit regularly. Steady green squares tell a story.

Creating a repository on GitHub

Click **+** → **New repository**. The form asks for:

Field	Advice
Repository name	short, lowercase, hyphens: `amroli-canteen`
Description	one sentence: what it is
Public / Private	public = anyone can see; private = only people you invite
Add a README	yes if you will `git clone` it; **no** if you will push an existing project
.gitignore template	pick your language (Python, Node, C...) to ignore build files
License	for open source, e.g. MIT; without one, others may not legally reuse your code

You can also create a file directly on GitHub (**Add file → Create new file**) and **commit** it in the browser. That creates a real commit, which you then `git pull` to your computer.

Managing a repository

Everything under **Settings** is visible only to owners and admins:

Setting	Use
General → Repository name	rename (GitHub redirects old links)
General → Default branch	the branch people see first (usually `main`)
Collaborators	give teammates push access (they must accept the invitation)
Branches → Branch protection	e.g. "require a pull request before merging into `main`"
Pages	publish a website from the repository (see Bonus)
Danger zone	change visibility, transfer, archive or **delete** (no undo)

Releases turn a tag (Unit 4) into a downloadable version with notes: *Releases → Draft a new release → choose tag* `v1.0`.

Issues: tracking work

An **issue** is a discussion about one task: a bug, a feature request, a question. Every issue gets a number (`#3`).

> 🍽️ **Mr. Desai reports a bug**
>
> **Title:** Cart total wrong when the same item is added twice **Body:** Steps: 1) add Samosa, 2) add Samosa again, 3) open cart. Expected ₹30, shows ₹15. Phone: Android, Chrome.

A good issue has:

- a **clear title** that names the problem,
- **steps to reproduce**, **expected** vs **actual** result,
- screenshots or error messages if possible.

Then the team:

- adds **labels** (`bug`, `enhancement`, `good first issue`),
- **assigns** someone,
- discusses in **comments** (Markdown works; mention people with `@username`).

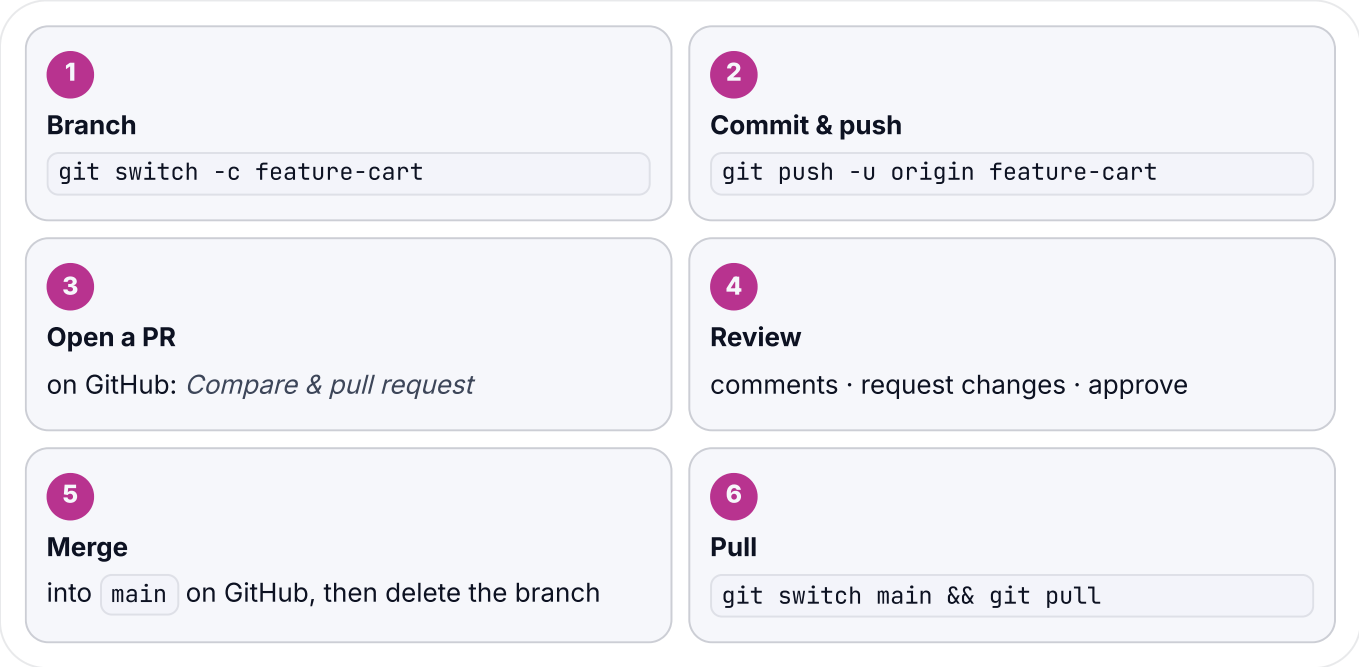
Close an issue from a commit by writing a keyword and the number in the message:

```
Git Bash
$ git commit -m "Fix cart total for repeated items (fixes #3)"
$ git push
```

When that commit reaches the default branch, GitHub closes issue #3 automatically. Keywords that work include `fixes`, `closes` and `resolves`.

Pull requests: proposing changes

A **pull request (PR)** says: *"Here is my branch. Please review it and merge it into* `main`*."* It is how professional teams merge work, because a teammate checks the code first.



Opening a PR: after `git push -u origin feature-cart`, GitHub shows a yellow **Compare & pull request** banner. Choose **base:** `main` ← **compare:** `feature-cart`, write a title and a description (what, why, how to test, `Closes #3`), and click **Create pull request**.

Reviewing a PR (Hetvi's job):

- **Files changed** tab: the diff, like `git diff main feature-cart`.
- Click a line to add a **comment**.
- Finish the review: **Comment**, **Approve**, or **Request changes**.
- The author fixes things by **pushing more commits to the same branch**. The PR updates by itself.

Merging a PR offers three buttons:

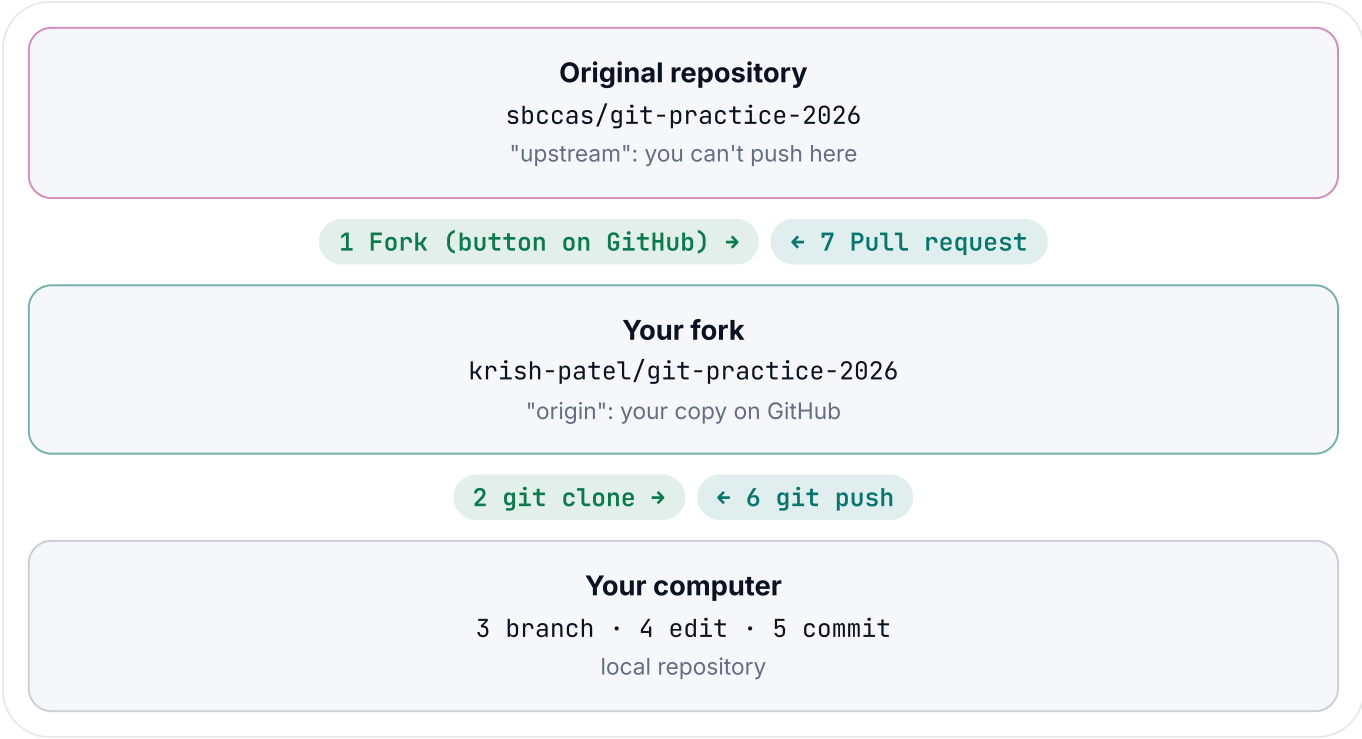
Option	Result
Create a merge commit	keeps all commits plus a merge commit (like `git merge --no-ff`)
Squash and merge	combines the branch into one commit on `main`
Rebase and merge	replays the commits on top of `main`, no merge commit

After merging, click **Delete branch**, then update your computer: `git switch main` and `git pull`.

> ⚠ **Conflicts in a PR**
>
> If `main` changed the same lines, GitHub says *"This branch has conflicts that must be resolved"*. Fix it exactly as in Unit 3: merge `main` into your branch locally, resolve, commit and push. (Small conflicts can also be resolved in GitHub's web editor.)

Forking: contributing to other projects

You can't push to a repository you don't own. To contribute anyway, you **fork** it: GitHub copies it to your account, you change your copy, and you send a pull request back.



	Fork	Clone	Branch
Creates	a copy **on GitHub**, under your account	a copy **on your computer**	a new line of work **inside** a repository
Done with	the **Fork** button	`git clone`	`git branch` / `git switch -c`

The seven-step contribution flow:

Git Bash

```
$ git clone https://github.com/krish-patel/git-practice-2026.git
$ cd git-practice-2026
$ git remote add upstream https://github.com/sbccas/git-practice-2026.git
$ git switch -c add-krish-patel
$ echo "# Krish Patel" > students/krish-patel.md
$ git add .
$ git commit -m "Add krish-patel profile card"
$ git push -u origin add-krish-patel
```

Then open the pull request from your fork to the original. Later, to get the original's new commits: `git pull upstream main`, or press **Sync fork** on your fork's GitHub page.

Open source etiquette

Well-run projects contain a few standard files:

File	Purpose
`README.md`	what the project is, how to install and use it
`LICENSE`	what others may legally do with the code
`CONTRIBUTING.md`	how to contribute: branch names, code style, PR rules
`.gitignore`	files Git should not track

Before contributing: read `CONTRIBUTING.md`, look for issues labelled **good first issue**, keep each PR small and focused, and be polite in reviews.

Your Git cheat sheet

The full, searchable, printable list is on the Cheat sheet page. The ten commands you'll use most:

Command	Does
`git status`	where is everything?
`git add <file>` / `git add .`	stage
`git commit -m "…"`	save a snapshot
`git pull`	get the team's work
`git push`	share your work
`git switch -c <branch>`	start a feature
`git merge <branch>`	bring a branch in
`git log --oneline --graph --all`	see the history
`git diff`	see what changed
`git restore <file>`	discard an edit

Unit 5 summary

- A repository page shows code, history, branches, issues, pull requests and settings; the README is its front page.
- Create repositories with a README (to clone) or empty (to push); choose visibility, `.gitignore` and licence.
- **Issues** track bugs and tasks; `fixes #3` in a commit closes issue 3.
- **Pull requests** propose merging a branch; teammates review, then merge (merge commit, squash or rebase).
- **Fork** = your copy on GitHub; contribute with fork → clone → branch → commit → push → PR; `upstream` points to the original.

Practise with **Labs L-01 – L-04**, then take the Unit 5 quiz and the final Self-check.

Appendix A · Command cheat sheet

⚙️ Setup (once per computer)

`git --version`
check that Git is installed U1

`git config --global user.name "Your Name"`
set your name for all repositories U1

`git config --global user.email "you@example.com"`
set your email (same as GitHub) U1

`git config --global init.defaultBranch main`
new repositories start on main U1

`git config --list`
show all settings U1

`git help <command>`
full manual for a command U1

`git <command> -h`
short options summary U1

🚀 Start a project

`git init`
turn the current folder into a repository U2

`git clone <url>`
copy a repository (and its history) from GitHub U2

`git remote add origin <url>`
connect a local repo to GitHub U2

`git remote -v`
show remote URLs U2

`git push -u origin main`
first push; sets the upstream U2

📦 Everyday work

`git status`
what changed? what is staged? U2

`git status -s`
short status U2

`git add <file>`
stage one file U2

`git add .`
stage everything in this folder U2

`git commit -m "Add menu page"`
save a snapshot of the staged changes U2

`git commit -am "Fix cart total"`
stage tracked files and commit U2

`git restore --staged <file>`
unstage a file (keep the edit) U2

☁️ Sync with GitHub

`git pull`
download and merge the team's work U2

`git fetch`
download only; look before merging U2

`git log --oneline main..origin/main`
commits on GitHub you don't have yet U2

`git push`
upload your commits U2

`git push origin <branch>`
upload a branch U3

`git push origin --delete <branch>`
delete a branch on GitHub U3

🌿 Branches & merging

`git branch`
list branches (* = current) U3

`git branch -a`
list local and remote branches U3

`git branch <name>`
create a branch (don't switch) U3

`git switch <name>`
move to a branch U3

`git switch -c <name>`
create and move to a branch U3

`git checkout -b <name>`
classic form of switch -c U3

`git merge <branch>`
merge a branch into the current one U3

`git merge --abort`
cancel a conflicted merge U3

`git branch -d <name>`
delete a merged branch U3

`git branch -D <name>`
force-delete an unmerged branch U3

🔍 Inspect history

`git log`
full history U4

`git log --oneline`
one line per commit U4

`git log --oneline --graph --all`
all branches as a graph U4

`git log --author="Dhruvi"`
commits by one person U4

`git log --grep="price"`
search commit messages U4

`git log -- <file>`
commits that changed a file U4

`git show <commit>`
one commit with its diff U4

`git show HEAD~1:<file>`
a file as it was in a commit U4

`git diff`
unstaged changes U4

`git diff --staged`
staged changes U4

`git diff main feature`
compare two branches U4

🏷️ Stash & tags

`git stash`
park unfinished changes U4

`git stash list`
list stashes U4

`git stash pop`
bring the newest stash back U4

`git tag -a v1.0 -m "First release"`
annotated release tag U4

`git tag`
list tags U4

`git push origin --tags`
upload all tags U4

↩️ Undo

`git restore <file>`

discard uncommitted edits (classic: git checkout -- <file>) U4

`git reset --soft HEAD~1`

undo last local commit, keep changes staged U4

`git reset HEAD~1`

undo last local commit, keep changes unstaged U4

`git reset --hard HEAD~1`

⚠️ undo last local commit and delete its changes U4

`git revert <commit>`

safely undo a pushed commit (new commit) U4

`git commit --amend -m "Better message"`

replace the last (unpushed) commit U4

🤝 GitHub workflow

`git remote add upstream <original-url>`

track the repository you forked U5

`git pull upstream main`

update from the original U5

`git push -u origin <branch>`

push a branch, then open a pull request U5

`git commit -m "Fix total (fixes #3)"`

close issue #3 when merged U5

Appendix B · Glossary

.gitignore U2

A file listing patterns of files Git should not track, such as passwords, build output or node_modules/.

Annotated tag U4

A tag that also stores a message, the tagger's name and a date. Created with git tag -a. Preferred for releases.

Branch U3

A movable label pointing to a commit, used to work on something in parallel without disturbing main.

Centralized version control U1

A VCS where one central server holds the only full history; clients depend on it. Examples: CVS, SVN.

Clone U2

A full copy of a repository, with all files and history, downloaded to your computer with git clone. The original becomes the remote 'origin'.

Collaborator U5

A person given permission to push to a GitHub repository (Settings → Collaborators).

Commit U2

A saved snapshot of the project, with an author, date, message and a link to its parent commit(s).

Commit hash U2

The unique 40-character ID of a commit (e.g. 6cb774b...). Usually shown shortened to 7 characters.

Commit message U2

The text describing what a commit does and why. Good messages start with a verb: 'Add vada pav to menu'.

Conflict U3

When two branches changed the same lines, Git can't merge automatically. It marks both versions in the file and asks you to choose.

Credential Manager U2

The tool installed with Git for Windows that signs you in to GitHub through the browser and remembers the login.

Detached HEAD U4

The state when HEAD points directly to a commit instead of a branch, e.g. after git checkout <commit>. Return with git switch main.

Diff U4

A line-by-line comparison showing what changed: lines starting with - were removed, + were added. Shown by git diff and git show.

Distributed version control U1

A VCS where every copy of a repository holds the full history, so you can work offline and any copy can restore the others. Git is distributed.

Fast-forward U3

A merge where the receiving branch had no new commits, so Git just moves its label forward. No merge commit is created.

Fetch U2

Download new commits from a remote into remote-tracking branches (e.g. origin/main) without changing your branch or files.

Fork U5

Your own copy of someone else's repository, on GitHub, under your account. Used to propose changes with a pull request.

Git U1

Free, open-source distributed version control software that runs on your computer. Created by Linus Torvalds in 2005.

Git Bash U1

The terminal installed with Git for Windows. It understands Linux-style commands such as ls, cd and pwd.

GitHub U1

A website that hosts Git repositories and adds teamwork features: issues, pull requests, forks, reviews and more.

HEAD U3

Git's pointer to where you are now: usually the current branch (shown in the prompt as (main)).

Issue U5

A numbered GitHub discussion about one task: a bug, a feature request or a question.

Local repository U2

The repository on your own computer: the .git folder with the full history.

Merge U3

Bring the commits of one branch into another. Results in a fast-forward or a merge commit.

Merge commit U3

A commit with two parents, created when two branches that both moved on are joined.

origin U2

The default name of the remote repository you cloned from or pushed to first (usually your GitHub repository).

Pull U2

git fetch followed by git merge: download the team's new commits and merge them into your current branch.

Pull request U5

A GitHub request to review a branch and merge it into another branch, usually main.

Push U2

Upload your commits to a remote repository such as GitHub.

README U5

The project's front page: a Markdown file explaining what the project is and how to use it. GitHub shows it below the file list.

Remote U2

A named copy of the repository on another computer or server, such as GitHub. Managed with git remote.

Remote-tracking branch U2

Git's read-only bookmark of where a remote branch was at your last fetch, pull or push, e.g. origin/main.

Repository U2

A project tracked by Git: the files plus their complete history (stored in the .git folder).

Reset U4

Move the current branch back to an earlier commit. --soft keeps changes staged, --mixed unstaged, --hard deletes them.

Restore U4

Discard uncommitted edits in a file (git restore file) or unstage it (git restore --staged file).

Revert U4

Create a new commit that undoes an earlier one. Safe for commits already pushed, because history is kept.

Root commit U2

The very first commit in a repository; it has no parent.

Staging area U2

The list of changes chosen for the next commit. git add puts changes here; also called the index.

Star U5

A GitHub bookmark for a repository you like; the number of stars shows popularity.

Stash U4

Temporarily park uncommitted changes so the working tree is clean; bring them back with git stash pop.

Tag U4

A permanent name for a commit, usually a release such as v1.0. Unlike a branch, it never moves.

Tracked file U2

A file Git already knows about (it was added and committed). New files are untracked until you git add them.

Two-factor authentication U1

A second login step (a code from an app or phone) that protects your GitHub account even if your password leaks.

Untracked file U2

A file in your folder that Git has never saved. git status lists it under 'Untracked files'.

Upstream U2

The remote branch your local branch tracks (set with git push -u). In forks, 'upstream' also names the original repository.

Version control system U1

Software that records changes to files over time, so you can see who changed what and bring back earlier versions.

Working directory U2

The files you see and edit in your project folder.

Appendix C · Practise online

Every mission runs in your browser with real Git; it checks your work and tells you when each goal is done. Labs use your own free account on github.com.

Mission	Unit	Title	Time
M-01	1	Who am I?	5 min
M-02	1	Ask the manual	5 min
M-03	2	Day One	10 min
M-04	2	Pack only what matters	10 min
M-05	2	Ship it to origin	10 min
M-06	2	Teammate pushed overnight	10 min
M-07	2	Join the project	10 min
M-08	3	Parallel universe	10 min
M-09	3	Bring it home	10 min
M-10	3	Two edits, one line	15 min
M-11	3	Spring cleaning	10 min
M-12	4	Detective	15 min
M-13	4	Drop everything!	10 min
M-14	4	Release day	10 min
M-15	4	Undo, the safe way	15 min

Lab	Title	Link
Unit 1 lab	Set up Git and GitHub	sbccas.github.io/git-github-fundamentals/labs/#unit-1
L-01	Your profile README	sbccas.github.io/git-github-fundamentals/labs/#l-01
Unit 2 lab	Your first project on GitHub	sbccas.github.io/git-github-fundamentals/labs/#unit-2
Unit 3 lab	Branches and a real conflict	sbccas.github.io/git-github-fundamentals/labs/#unit-3
Unit 4 lab	Detective, stash, tag and undo	sbccas.github.io/git-github-fundamentals/labs/#unit-4
L-02	Create and configure a repository	sbccas.github.io/git-github-fundamentals/labs/#l-02
L-03	Report and fix an issue	sbccas.github.io/git-github-fundamentals/labs/#l-03
L-04	Fork and pull request	sbccas.github.io/git-github-fundamentals/labs/#l-04

Appendix D · Course at a glance

Session	Unit	Topic	Practice
1	1	Why version control?	—
2	1	Installing Git, config, help & GitHub account	M-01, M-02, L-01
3	2	Terminal basics, git init & the three stages	M-03
4	2	status, add and commit in depth	M-04
5	2	Local vs remote: push, clone & your first GitHub repo	M-05, M-07
6	2	Teamwork: fetch vs pull	M-06
7	3	Branches: create & switch	M-08
8	3	Merging & resolving conflicts	M-09, M-10
9	3	Deleting branches & team branching workflow	M-11
10	4	Reading history: log, diff, show	M-12
11	4	stash & tags	M-13, M-14
12	4	Undoing changes: restore, reset, revert	M-15
13	5	The GitHub interface, repositories & issues	L-02, L-03
14	5	Forks, pull requests & the cheat sheet	L-04
15	—	Final practical, self-check & showcase	—

15 sessions × 2 hours = 30 hours · 2 credits.