SBCCAS · AMROLI COLLEGE, SURAT

FACULTY EDITION

CERTIFICATE COURSE · 2 CREDITS · 30 HOURS

Teaching Git & GitHub with Stories

The faculty handbook: the Amroli Canteen story bible, deeper explanations and classroom examples for every unit.



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 **before** you teach its unit. The faculty guide says how to run each session; this book says what to know. Slides for all 15 sessions are on the Teach page.

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. Faculty 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

Story bible: the Amroli Canteen 5

Why teach Git with a story? 5

The project 5

The cast 6

The story arc 7

Other story settings for variety 8

Tips for telling the story 9

Chapter 1: Introduction to Version Control 10

Learning outcomes 10

Episode 1: The final_FINAL disaster 10

The ideas in depth 11

More story examples 14

Board diagrams to draw 14

Questions students ask 15

Misconceptions to correct 15

Self-check for faculty 16

10-minute recap 16

Chapter 2: Getting Started with Git 17

Learning outcomes 17

Episode 2: First commit, first push 17

The ideas in depth 18

Worked walkthrough (real output) 21

More story examples 21

Board diagrams to draw 21

Questions students ask 22

Misconceptions to correct 23

Self-check for faculty 23

10-minute recap 24

Chapter 3: Branching and Merging 25

Learning outcomes 25

Episode 3: Parallel universes 25

The ideas in depth 26

Worked walkthrough (real output) 28

More story examples 28

Board diagrams to draw 29

Questions students ask 29

Misconceptions to correct 30

Self-check for faculty 30
10-minute recap 31

Chapter 4: Advanced Git Commands 32

Learning outcomes 32
Episode 4: Detective work 32
The ideas in depth 33
Worked walkthrough (real output) 36
More story examples 36
Board diagrams to draw 37
Questions students ask 37
Misconceptions to correct 38
Self-check for faculty 38
10-minute recap 38

Chapter 5: Introduction to GitHub 40

Learning outcomes 40
Episode 5: Going public 40
The ideas in depth 41
Running Lab L-04 smoothly 42
More story examples 43
Board diagrams to draw 43
Questions students ask 43
Misconceptions to correct 44
Self-check for faculty 44
10-minute recap 45

Appendices 46

A. Course at a glance 46
B. Where everything is on the course site 47
C. Story quick reference 48
D. Further reading for faculty 48

Story bible: the Amroli Canteen

Every unit of this course is taught through one continuing story. Lessons, missions, quizzes and slides all use the same characters, so students meet the same people again and again and remember the Git idea attached to each of their problems. This section is the reference every faculty member should use, so the story stays consistent across classes and teachers.

Why teach Git with a story?

Git commands are abstract: *staging area*, *remote-tracking branch*, *fast-forward*. Students forget definitions, but they remember **what went wrong for someone and how it was fixed**. Each Git concept in this course is introduced as the solution to a problem a character actually has:

- "Meet's laptop crashed" → why distributed version control matters.
- "Dhruvi and Krish both changed the opening time" → merge conflicts.
- "The price shows ₹0 since last week. Who changed it?" → `git log`, `git diff`.

When a student later asks "why do we need `git fetch`?", you can answer "remember the morning Dhruvi pushed overnight?" and the class already knows.

The project

Amroli Canteen is a BCA final-year project at SBCCAS: a small web app where students order snacks *before* the lunch break and pick them up without queuing.

Part	File(s)	Built by
Menu page	`index.html`, `menu.txt`, `menu.css`	Krish
Cart	`cart.js`	Krish
Order system	`app.py`, `orders.py`	Dhruvi
Database	`schema.sql`, `seed.sql`	Meet
Documentation	`README.md`, `team.md`, planning notes repo `canteen-docs`	Hetvi
Opening time	`timing.txt` (used in the conflict story)	everyone, and that's the problem

GitHub repositories used in the course (simulated in the Playground, real in labs): `krish-patel/amroli-canteen` (main project) and `hetvi-desai/canteen-docs` (planning notes).

The cast

Character	Role	Personality you can play up	Typical mistake (and lesson)
Krish Patel	frontend: menu & cart	Fast, enthusiastic, "I'll just quickly change it"	Commits everything with `git add .`, writes "update" as a message, breaks working code. Teaches staging, good messages, branches, undo.
Dhruvi Patel	backend: orders	Careful, organised, pushes late at night	Her overnight pushes cause Krish's rejected pushes. Teaches fetch, pull, rejected pushes.
Meet Shah	database	Quiet, works alone on his laptop	Laptop crashes with unpushed work. Teaches backups, distributed VCS, push often.
Hetvi Desai	team lead	Plans, reviews, protects `main`	Insists on branches and pull requests; tags releases. Teaches workflow, reviews, issues, tags.

Supporting characters you may use: **Mr. Desai**, the canteen manager who reports bugs (*issues* in Unit 5), and **the juniors** who later *fork* the project for another college (Unit 5).

The story arc

Unit	Episode	What happens	Git ideas	Missions
1	*The final_FINAL disaster*	Two weeks of WhatsApp and pen-drive sharing: lost work, file chaos, overwrites, no history, Meet's crash. Hetvi says: "we learn Git." Everyone installs Git and makes GitHub accounts.	VCS types, Git vs GitHub, install, `config`, `help`, GitHub account	M-01, M-02
2	*First commit, first push*	Krish creates the repo and commits the menu. He pushes to GitHub. Dhruvi clones it, adds the order system and pushes overnight. Next morning Krish fetches and pulls.	`init`, three stages, `status` / `add` / `commit`, `remote`, `push`, `clone`, `fetch`, `pull`	M-03 – M-07
3	*Parallel universes*	Krish builds the cart on a branch while main keeps working. Dhruvi and Krish both change the opening time; the merge conflicts and they resolve it together. Old branches are cleaned up.	branches, `switch`, merge (fast-forward / 3-way), conflicts, deleting branches	M-08 – M-11
4	*Detective work*	Prices show ₹0. The team reads the history to find the bad commit. During a demo, an urgent fix interrupts half-done work (stash). They	`log`, `diff`, `show`, `stash`, tags, `reset`, `revert`, `restore`	M-12 – M-15

Unit	Episode	What happens	Git ideas	Missions
		tag v1.0 for the college tech fest and safely undo a pushed mistake.		
5	*Going public*	The repo goes public. Mr. Desai reports bugs as issues. Every change now goes through a pull request reviewed by Hetvi. Juniors from another college fork the project.	GitHub interface, repos, issues, pull requests, forks	L-01 – L-04

Other story settings for variety

Use these when a class has heard the canteen story enough, or when you want students to *transfer* the idea to a new situation. These settings are used in the handbook chapters and make good exam questions.

Setting	Use it for
Spectrum 2026 tech-fest website: 6 volunteers edit the event schedule page the night before the fest	conflicts, branches per event, tags for "final schedule"
College library management system: a C/Python project passed from one batch to the next	clone, history, `log`, why commit messages matter years later
PCPM class assignment: a student's folder of C programs (`practical1.c` … `practical20.c`)	`init`, staging, small commits, undoing a broken program, a first personal GitHub portfolio
Thesis / project report in Word: `report_final_v7.docx`	the "why version control" hook for non-programmers

Tips for telling the story

1. **Name the character, then the command.** "Krish needs to save his work → `git commit`."
2. **Let students predict.** "Dhruvi pushed last night. Krish pushes now. What will happen?" Then show the rejected push.
3. **Use the same file names** as the lessons (`menu.txt`, `orders.py`, `timing.txt`) so demos match what students see in missions.
4. **Show the mistake first.** Each episode starts with something going wrong. The fix only makes sense after students feel the problem.
5. **Keep it light.** Students enjoy Krish's mistakes; they will make the same ones.

Chapter 1: Introduction to Version Control

Sessions 1–2 · 4 hours · Missions M-01, M-02 · Lab L-01

Learning outcomes

By the end of Unit 1, a student can:

1. Explain in their own words what a version control system is and the four questions it answers (what, who, when, why).
2. Compare local, centralized and distributed version control, and say which one Git is and why that matters.
3. Explain the difference between Git and GitHub, and name two alternatives to GitHub.
4. Install Git for Windows with sensible options and verify it.
5. Configure `user.name` and `user.email`, and explain `--global` vs local configuration.
6. Find help with `git help`, `git help <command>` and `git <command> -h`.
7. Create a secure, professional GitHub account with two-factor authentication.

Episode 1: The final_FINAL disaster

> It is the second week of the final-year project. Hetvi has a WhatsApp group called *Amroli Canteen* 🍽 with forty-three unread messages. Most of them are files.
>
> On Monday, Krish sends `menu.html`. It works. On Tuesday he sends `menu_v2.html` with a new design, and the cart button stops working. Nobody can find Monday's version: it was on his laptop, and he saved over it.
>
> Dhruvi edits `app.py` to add order numbers and sends it at 11 pm. Krish, who had downloaded the previous `app.py` that morning, adds a discount feature and sends *his* copy at midnight. Dhruvi's order numbers vanish. Nobody notices for three days.
>
> On Friday the total price shows ₹0. Everyone says "I didn't touch that."
>
> The next Monday, Meet's laptop will not start. The database tables he designed over two weeks were only on that laptop.
>
> Hetvi calls a meeting in the lab. "This is how students work," she says. "It is not how companies work. They use something called **version control**. This week we all learn Git."

Use this episode as the opening of Session 1. Ask students to count the problems: there are five (lost work, file-name chaos, overwritten work, no history, no backup). Each is solved by a feature they will learn.

The ideas in depth

What a version control system really stores

A version control system keeps a **history of snapshots** of a project. In Git, each snapshot is a **commit**. A commit records:

- the complete state of every tracked file at that moment (Git stores *snapshots*, not just differences, and saves space by reusing unchanged files and compressing data),
- the **author** and **committer** (name, email, time),
- a **message** explaining the change,
- a link to the **parent** commit(s), so the history forms a chain (and later, with branches, a graph).

Each commit is identified by a **hash**, a 40-character hexadecimal ID calculated from its contents (SHA-1 by default). Change anything in a commit and its hash changes, which is why history in Git is very hard to corrupt silently. Students usually see the first seven characters, e.g. `6cb774b`.

A short history of version control

Year	System	Type	Note for class
1972	SCCS (Bell Labs)	local	one of the first: tracked versions of single files
1982	RCS (Walter Tichy)	local	the "local VCS" example used in Unit 1
1990	CVS	centralized	first widely used team system
2000	Subversion (SVN)	centralized	"CVS done right"; still found in some companies
2005	**Git** (Linus Torvalds)	distributed	built for the Linux kernel
2005	Mercurial	distributed	similar idea to Git; less common today

The Git origin story (students enjoy it): from 2002 the Linux kernel used a commercial distributed system called BitKeeper for free. In April 2005 that free licence was withdrawn. Linus Torvalds wrote the first version of Git in about ten days, aiming for speed, a distributed model and strong protection against corrupted history. Since mid-2005 Git has been maintained by Junio Hamano.

Local vs centralized vs distributed, more precisely

- **Local:** a version database on one machine. Good for one person; no collaboration; one disk failure loses everything.
- **Centralized:** a single server holds the *only* full history. Clients have just a working copy. You need the server for almost every operation (commit, log, diff against old versions). Server failure = nobody can work; server data loss = history lost.
- **Distributed:** every clone is a **complete repository** with the full history. Commit, log, diff and branch all work offline; you only need the network to *share* (push, fetch, pull). Any clone can rebuild the server. In practice teams still agree on one shared copy (GitHub), but it is a convention, not a technical necessity.

Git vs GitHub vs the alternatives

	What it is	Notes
Git	free, open-source version control software	runs on your computer; command line or GUI
GitHub	website and company that hosts Git repositories	founded 2008, owned by Microsoft since 2018; issues, pull requests, Actions, Pages
GitLab	similar hosting service	can also be installed on a company's own server
Bitbucket	similar hosting service	from Atlassian; integrates with Jira
Gitea	lightweight self-hosted service	common for private college / company servers

The commands students learn are identical everywhere; only the website changes.

Configuration levels

Git reads settings from three places, and a later (more specific) one overrides an earlier one:

Level	Option	File	Applies to
system	`--system`	Git's install folder (`etc/gitconfig`)	every user on the computer
global	`--global`	`~/.gitconfig`	one user account
local	*(default inside a repo)*	`.git/config`	one repository

`git config --list --show-origin` shows which file each setting comes from. This is useful when a student's commits show the wrong name on a lab PC.

Lab PCs: recommend students set name and email **locally** (without `--global`) inside their project folder, so the next user of the PC doesn't commit under their name.

Privacy: GitHub can hide a user's real email. Under *Settings → Emails*, tick *Keep my email addresses private* and use the address GitHub shows (`ID+username@users.noreply.github.com`) as `user.email`.

Getting help

Command	Output
`git help`	list of common commands, grouped by task
`git help <command>` / `git <command> --help`	full manual page (opens in a browser on Git for Windows)
`git <command> -h`	short usage summary printed in the terminal
`git help -g`	list of concept guides (e.g. `git help everyday`, `git help glossary`)

Train the habit: **read the error message first.** Git's messages often contain the exact fix.

GitHub accounts

- **Username:** short, professional, permanent-looking (`krish-patel`). It becomes the portfolio URL.
- **Two-factor authentication:** GitHub requires 2FA for everyone who contributes code (rolled out during 2023). Students should use an authenticator app or GitHub Mobile and **store the**

recovery codes. A lost phone without recovery codes can lock them out permanently.

- **Passwords are not used for Git commands.** Since August 2021 GitHub rejects account passwords for Git over HTTPS. On Windows, Git Credential Manager handles sign-in through the browser; alternatives are personal access tokens or SSH keys.
- **Profile README:** a repository named exactly like the username shows its README on the profile page (Lab L-01).

More story examples

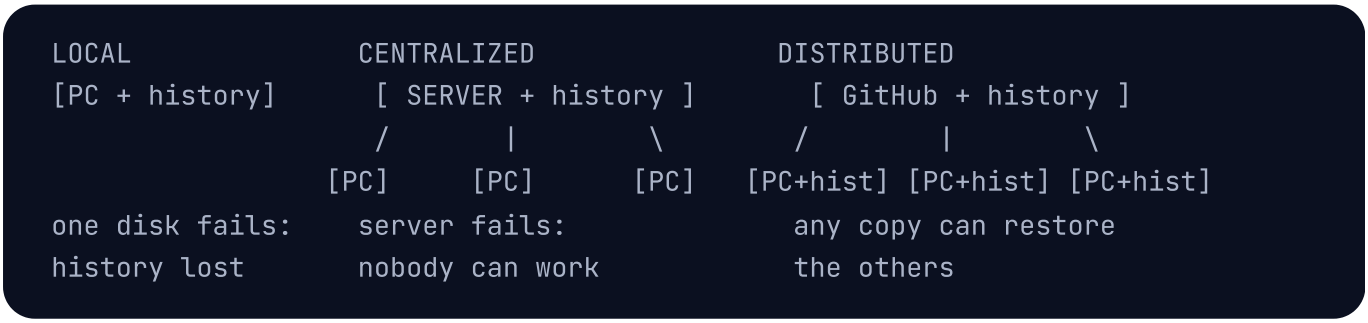
Spectrum 2026 tech-fest website. Six volunteers edit `schedule.html` the night before the fest. Two of them change the time of the robotics event. In the morning the printed schedule is wrong and nobody knows whose copy was printed. *Use to show:* why a central "source of truth" plus history matters.

The PCPM assignment folder. A first-year student keeps `practical1.c` … `practical20.c`. Practical 12 compiled on Friday; on Monday, after "a small fix", it doesn't. *Use to show:* that version control is useful even for one person; a commit after each working practical is a checkpoint.

The library management system. The college library software was written by a 2019 batch. A 2026 student must fix a bug. With Git, `git log` shows every change since 2019 with reasons; without it, there is just a ZIP file. *Use to show:* history and messages are documentation for the future.

Board diagrams to draw

The three kinds of VCS (draw in three columns):



Git vs GitHub: a laptop labelled *Git (tool)* with an arrow labelled *push* to a cloud labelled *GitHub (website)*.

Questions students ask

"Is Google Drive version control?" Partly. Drive keeps old versions of single files, but not a snapshot of the whole project at a moment, no messages explaining why, no branches and no merging of two people's changes. Git does all of these.

"Do I need the internet to use Git?" No. Everything except sharing (push, fetch, pull, clone) works offline.

"Is GitHub free?" Yes for students and most uses: unlimited public and private repositories. The GitHub Student Developer Pack adds free extras with a verified student email.

"Can people see my code on GitHub?" Only if the repository is **public**. Private repositories are visible only to people you invite.

"Why does the installer ask so many questions?" Most defaults are fine. The important choices are the editor (Notepad or VS Code, not Vim) and the default branch name (`main`).

"I set the wrong email. Are my old commits ruined?" No. Fix `user.email` for future commits. (Rewriting old commits is possible but not needed in this course.)

"What is the difference between `git help commit` and `git commit -h`?" The first opens the full manual; the second prints a short summary in the terminal.

Misconceptions to correct

Misconception	Correction
"GitHub is a version control system."	Git is the VCS; GitHub hosts Git repositories.
"Version control is only for big teams."	It helps one student too: undo, history, backup.
"Distributed means there is no main copy."	Teams still agree on a shared copy (GitHub); distributed means *every* copy is complete.
"`user.name` is my GitHub login."	It is just text written into commits. The **email** links commits to the GitHub account.
"Git saves only the differences."	Conceptually Git stores snapshots; compression is an internal detail.

Self-check for faculty

1. Which VCS type keeps the only full history on one server? *Centralized.*
2. Name the four questions a VCS answers. *What, who, when, why.*
3. Who created Git, and what event triggered it? *Linus Torvalds, 2005, when Linux lost its free BitKeeper licence.*
4. What file does `git config --global` write to? *`~/.gitconfig`*.
5. Which setting wins: global or local? *Local.*
6. Which command prints a short usage summary? *`git <command> -h`*.
7. Why should `user.email` match the GitHub account? *So GitHub links commits to the profile and contribution graph.*
8. Why can't students push with their GitHub password? *GitHub stopped accepting passwords for Git over HTTPS in 2021; use Credential Manager, a token or SSH.*
9. What makes a repository a "profile README"? *Its name equals the username and it contains a README.*
10. Give one non-programming example of version-control pain. *`report_final_v7.docx`*.

10-minute recap

Use at the start of Session 3 or in a revision lecture:

1. Show the `project_FINAL_really.zip` list. Ask: "What problem is this?" (no history)
2. Draw the three VCS types; ask which one Git is and why that protects Meet.
3. Ask a student to type the two `git config --global` commands from memory.
4. One-sentence answers: Git vs GitHub; `git help commit` vs `git commit -h`.

Chapter 2: Getting Started with Git

Sessions 3–6 · 8 hours · Missions M-03 – M-07 · Unit 2 lab

Learning outcomes

By the end of Unit 2, a student can:

1. Navigate folders in Git Bash (`pwd`, `ls`, `cd`, `mkdir`) and avoid running Git in the wrong folder.
2. Explain local vs remote repositories and what `origin` means.
3. Create a repository with `git init` and describe what the `.git` folder holds.
4. Explain the three stages (working directory, staging area, repository) and move changes between them with `git add`, `git restore --staged` and `git commit`.
5. Read `git status` in long and short form.
6. Write clear commit messages.
7. Connect a local repository to GitHub (`git remote add`, `git push -u`) and copy a remote one (`git clone`).
8. Explain and use `git fetch` vs `git pull`, and recover from a rejected push.

Episode 2: First commit, first push

> Monday morning, lab 3. Krish opens Git Bash and makes a folder, `amroli-canteen`. He types `git init`. "That's it?" he asks. "Nothing happened." Hetvi tells him to type `ls -a`, and a hidden `.git` folder appears. "That folder *is* the repository," she says.
>
> Krish writes the first menu line, `Samosa 15`, and tries to commit. Git says nothing is staged. He learns `git add`, and then makes his first commit: `[main (root-commit) 6cb774b] Add menu`. He grins: "Root commit. Like the first checkpoint."
>
> By lunch he has three commits. He tried `git add .` once, and his personal `notes-private.txt` nearly went in with the project. He unstaged it with `git restore --staged` just in time.
>
> Hetvi creates an empty repository on GitHub. Krish connects it (`git remote add origin …`) and pushes. The files appear on the GitHub page and the class claps.
>
> That night Dhruvi clones the repository, adds `orders.py`, and pushes at 11:40 pm.

> Tuesday morning, Krish adds a new menu item and pushes. **Rejected.** "Updates were rejected because the remote contains work that you do not have locally." Hetvi smiles: "Dhruvi was up late. Fetch first, look, then pull."

Session mapping: Session 3 = the first half (init, three stages); Session 4 = staging decisions and messages (the private-notes moment); Session 5 = remote, push, clone; Session 6 = the rejected push, fetch and pull.

The ideas in depth

What is inside `.git`

Item	Holds
`objects/`	every version of every file (**blobs**), every folder listing (**trees**), every **commit**
`refs/heads/`	branch names, each a file containing a commit hash
`refs/remotes/origin/`	**remote-tracking branches**: Git's memory of where GitHub's branches were at the last fetch/push
`HEAD`	which branch you are on (usually `ref: refs/heads/main`)
`index`	the **staging area** (created on the first `git add`)
`config`	settings for this repository, including remotes

Three object types matter for teaching: a **blob** is a file's content, a **tree** is a folder (names → blobs/trees), and a **commit** points to one tree (the snapshot) plus its parent commit(s), author, date and message. That is the whole model; branches are just labels pointing to commits.

The three stages, precisely

- The **working directory** is ordinary files.
- The **staging area (index)** is *the snapshot Git will save next*. `git add file` copies the file's current content into it.
- The **repository** holds the saved commits.

A file therefore has a lifecycle:

```
untracked --git add⟶ staged --git commit⟶ unmodified --edit⟶ modified --
git add⟶ staged …
```

Important subtlety for strong students: `git add` stages the content *as it is at that moment*. If you edit the file again after `git add`, `git status` shows it **both** under "Changes to be committed" and "Changes not staged". Commit now and only the first version is saved. Demonstrate this; it explains why the staging area exists.

Reading `git status`

Section	Meaning	Next command
Untracked files	never added	`git add <file>`
Changes not staged for commit	tracked file edited	`git add <file>` or `git restore <file>` to discard
Changes to be committed	staged	`git commit`, or `git restore --staged <file>` to unstage
Your branch is ahead of 'origin/main' by N commits	local commits not pushed	`git push`
Your branch is behind 'origin/main' … can be fast-forwarded	fetched commits not merged	`git pull`

Short form (`git status -s`): two columns, **left = staging area, right = working directory**: `M` staged change, `M` unstaged change, `MM` both, `A` new file staged, `??` untracked.

Commit messages

A good message completes the sentence *"If applied, this commit will …"*:

- ✅ `Add vada pav to menu` · `Fix total when cart is empty` · `Remove unused CSS`
- ❌ `update` · `changes` · `final` · `asdf` · `done`

Keep the first line under about 50 characters. For bigger changes, leave a blank line and explain *why* in a body (`git commit` without `-m` opens the editor).

Remotes, `origin` and upstream

- A **remote** is a named URL. `git remote add origin <url>` creates one called `origin` (a convention, not a keyword).
- `git push -u origin main` uploads `main` **and** records the **upstream** (`branch.main.remote=origin`, `branch.main.merge=refs/heads/main` in `.git/config`). After that, plain `git push` / `git pull` know where to go, and `git status` can say "ahead/behind".
- `origin/main` is a **remote-tracking branch**: a read-only bookmark of where GitHub's `main` was at the last contact. It moves only on `fetch`, `pull` or `push`, never on its own.

`clone` vs `init`

	`git init`	`git clone <url>`
Starts with	an existing folder of yours	a repository on GitHub
Creates	empty history	full history copied
Remote	none (add one yourself)	`origin` set automatically
Branch tracking	none until `push -u`	`main` already tracks `origin/main`

`fetch`, `pull` and rejected pushes

- `git fetch` downloads new commits and moves `origin/main`. Your `main` and your files don't change.
- `git pull` = `git fetch` + `git merge origin/main` (the default on Git for Windows is *fast-forward or merge*).
- A push is **rejected** when GitHub has commits you don't have. Git refuses to overwrite them. Correct response: `git pull` (resolve any conflict), then `git push`. Never teach `--force` as a fix at this stage.

Signing in (real computers)

- **Git Credential Manager** (installed with Git for Windows) opens a browser window to sign in on the first push, then stores a token in Windows Credential Manager.
- Shared lab PCs: remove the stored credential after class (*Control Panel → Credential Manager → Windows Credentials → git:https://github.com*).
- If GitHub is blocked by a proxy: `git config --global http.proxy http://host:port`.

Worked walkthrough (real output)

This is exactly what students see in the Playground (Mission M-03 and Unit 2 lesson):

```
Git Bash

$ mkdir amroli-canteen && cd amroli-canteen
$ git init
Initialized empty Git repository in /home/student/amroli-canteen/.git/
$ echo "Samosa 15" > menu.txt
$ git status -s
?? menu.txt
$ git add menu.txt
$ git commit -m "Add menu"
[main (root-commit) 6cb774b] Add menu
 1 file changed, 1 insertion(+)
 create mode 100644 menu.txt
$ git log --oneline
6cb774b (HEAD → main) Add menu
```

Points to narrate: `(root-commit)` means the first commit; `6cb774b` is the short hash; `HEAD -> main` means "you are on main and it points here".

More story examples

Spectrum 2026 schedule (staging). A volunteer fixed a typo in `schedule.html` *and* started redesigning `events.html`, which is half finished. With staging she commits only the typo fix now (`git add schedule.html`), and the redesign later. *Ask:* "Without a staging area, what would happen?"

Library system (clone). A new student joins the library project. Instead of copying a ZIP from a senior's pen drive, she runs `git clone` and gets all 400 commits since 2019. *Ask:* "What does she lose if she uses *Download ZIP* instead?"

PCPM portfolio (push). A student commits each C practical after it compiles and pushes to a public repository `pcpm-practicals`. At placement time, the GitHub page shows steady work over a semester. *Ask:* "What story do the commit dates tell a recruiter?"

Board diagrams to draw

Three stages with arrows:

```
Working directory  --git add⟶  Staging area  --git commit⟶  Repository
  (edit files)     ⟵git restore --staged--      (index)          (.git)
```

Fetch vs pull:

```
GitHub main:      A---B---C
git fetch:        origin/main moves to C, your main stays at B
git pull:         origin/main → C, and your main fast-forwards to C
```

Questions students ask

"Why not commit directly without `add`?" Because you often change several things at once and want to save them as separate, meaningful commits. (`git commit -a` exists for the simple case, but it never adds new files.)

"Where is the staging area? I can't see it." It is a file, `.git/index`. You see it through `git status` and `git diff --staged`.

"Do I have to type `-u` every time?" No, only on the first push of each branch.

"What happens if two people push at the same time?" The second push is rejected. They pull, merge, and push again. Git never silently overwrites someone else's commits.

"Can I rename `origin`?" Yes (`git remote rename origin github`), but everyone uses `origin` by convention.

"I created the GitHub repo with a README and now `git push` fails." Both sides have a different first commit. Use `git pull origin main --allow-unrelated-histories`, or next time create the GitHub repo empty (or clone it instead of `init`).

"Is `git pull` dangerous?" It can create a merge or a conflict, but it never deletes your committed work. Commit first, then pull.

Misconceptions to correct

Misconception	Correction
"`git add` saves my work."	It only stages. `git commit` saves.
"`git commit` uploads to GitHub."	Commit is local; `git push` uploads.
"`git init` creates a GitHub repository."	It only creates `.git` locally.
"`git fetch` didn't work because my files didn't change."	Fetch updates `origin/main` only. That's the point.
"Clone and Download ZIP are the same."	ZIP has no history and no connection to GitHub.
"Pushing again with `--force` fixes a rejected push."	It deletes teammates' commits on GitHub. Pull first.

Self-check for faculty

1. What does `git init` create? *A `.git` folder: the repository.*
2. Where is the staging area stored? *`.git/index`.*
3. A file shows `MM` in `git status -s`. What does that mean? *Staged change plus another unstaged change.*
4. What does `-u` record, and where? *The upstream, in `.git/config` (`branch.main.remote/merge`).*
5. What is `origin/main`? *A remote-tracking branch: where GitHub's `main` was at the last fetch/push.*
6. When does `origin/main` move? *Only on fetch, pull or push.*
7. Push rejected with "(fetch first)": correct sequence? *`git pull`, resolve if needed, `git push`.*
8. Clone vs init: two differences. *Clone copies history and sets `origin` and tracking; init starts empty with no remote.*
9. Why might `git commit -am` not include a new file? *`-a` only stages tracked files.*
10. Give a good and a bad commit message. *"Fix total when cart is empty" vs "update".*

10-minute recap

1. Draw the three stages; ask students to label the arrows (`add`, `commit`, `restore --staged`).
2. Show a `git status` output and ask which command comes next.
3. "Krish's push was rejected. What happened, and what does he type?"
4. One sentence each: `fetch` vs `pull`; `clone` vs `init`.

Chapter 3: Branching and Merging

Sessions 7–9 · 6 hours · Missions M-08 – M-11 · Conflict game · Unit 3 lab

Learning outcomes

By the end of Unit 3, a student can:

1. Explain what a branch is (a movable pointer) and what HEAD is.
2. Create, list and switch branches with `git branch`, `git switch` / `git checkout`, `-c` / `-b`.
3. Predict how the working directory changes when switching branches.
4. Merge a branch, and tell a fast-forward from a three-way merge (with a merge commit).
5. Resolve a merge conflict: read the markers, edit, `git add`, `git commit`, or `git merge --abort`.
6. Delete local branches (`-d`, `-D`) and remote branches (`git push origin --delete`).
7. Follow a simple feature-branch workflow in a team.

Episode 3: Parallel universes

> The app is working, and Hetvi demonstrates it to Mr. Desai, the canteen manager, every Friday. "Nobody touches `main` unless it works," she says.
>
> Krish wants to redesign the menu page, and Dhruvi wants to add online payment. Both will take a week. Hetvi draws two lines branching off the main line on the whiteboard: "Each of you gets a branch."
>
> Krish types `git switch -c feature-menu`, adds *Vada Pav*, commits, and switches back to `main` to check the demo. He gasps: "My Vada Pav is gone!" Hetvi laughs. "It's on your branch. `main` didn't change. That's the point."
>
> On Thursday Krish merges his finished menu into `main`: *Fast-forward*. Then Dhruvi merges the cart: *Merge made by the 'ort' strategy*, and a new merge commit appears in the graph.
>
> Friday morning, disaster. Dhruvi changed the opening time to 8 AM on her branch; Krish changed the same line on `main` to 10 AM. `git merge` stops: **CONFLICT**. The file shows

> both lines between strange arrows. Hetvi decides on 8 AM, Krish deletes the markers, `git add`, `git commit`, and the demo is saved.
>
> Afterwards the repository has eleven branches. "Spring cleaning," says Hetvi. Merged branches are deleted with `-d`; the failed `experiment` branch needs `-D`.

The ideas in depth

Branches are pointers

A branch is a file in `.git/refs/heads/` containing one commit hash. That's all. Creating a branch writes a 41-byte file, which is why it is instant even in huge projects. When you commit, Git creates the commit and then **moves the current branch's pointer** to it.

HEAD is usually a *symbolic* reference: the file `.git/HEAD` contains `ref: refs/heads/main`. "Switching branches" means: change HEAD to the other branch, then update the working directory and index to match that branch's commit.

Detached HEAD (Unit 4) is when `.git/HEAD` contains a commit hash directly instead of a branch name.

What switching really does

1. Git compares the current commit's tree with the target commit's tree.
2. Files that differ are rewritten in your folder; files that exist only in the target appear; files that exist only in the current commit disappear.
3. Untracked files and unrelated uncommitted edits are left alone.
4. If an uncommitted edit would be **overwritten**, Git refuses (*"Your local changes … would be overwritten by checkout"*).

This explains the classic surprise: "my file vanished when I switched". It is in the other branch's commit.

`checkout` vs `switch` / `restore`

`git checkout` historically did two unrelated jobs: switching branches **and** restoring files. Since Git 2.23 (2019) there are two clearer commands, `git switch` (branches) and `git restore` (files). The syllabus names `checkout`, and most online material uses it, so teach both and show the mapping:

Task	Classic	Modern
Switch branch	`git checkout main`	`git switch main`
Create + switch	`git checkout -b x`	`git switch -c x`
Discard file edits	`git checkout -- file`	`git restore file`
Visit an old commit	`git checkout a1b2c3d`	`git switch --detach a1b2c3d`

How a merge works

To merge branch B into the current branch A, Git finds the **merge base**, the most recent common ancestor of A and B.

- If the merge base **is A itself** (A has no new commits), the merge is a **fast-forward**: A's pointer simply moves to B. No new commit.
- Otherwise Git performs a **three-way merge**: it compares *base → A* and *base → B* for every file. Changes to different files, or to different parts of a file, are combined automatically. The result is saved as a **merge commit with two parents**.
- The default strategy in modern Git is called **ort** (*"Ostensibly Recursive's Twin"*), which is why the message says *Merge made by the 'ort' strategy*.
- `git merge --no-ff` forces a merge commit even when a fast-forward is possible. Some teams like this to keep a visible record of each feature.

Conflicts, precisely

A conflict happens when both sides changed **the same lines** (or one side edited a file the other deleted). Git:

- writes both versions into the file between markers (`<<<<<<< HEAD`, `=======`, `>>>>>>> branch`),
- records the file as **unmerged** in the index (`git status` shows *both modified*),
- writes `MERGE_HEAD` so the next `git commit` creates a merge commit with two parents.

The student's job: produce the correct content, remove the markers, `git add` (marks it resolved), `git commit`. `git merge --abort` returns to the state before the merge.

Important teaching point: Git checks for unmerged files, **not** for leftover marker text. If a student runs `git add` on a file that still contains `<<<<<<<`, Git accepts it and the markers end up committed. Train them to check the file before `git add`.

Deleting branches

- `git branch -d x` deletes the label only if its commits are already contained in the current branch (merged).
- `git branch -D x` deletes it regardless. Commits that were only on that branch become unreachable and are eventually garbage-collected. (Advanced: `git reflog` can rescue them for a while.)
- `git push origin --delete x` deletes the branch on GitHub. Other people's `origin/x` remains until they run `git fetch --prune`.

Worked walkthrough (real output)

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.
$ cat timing.txt
<<<<<<< HEAD
Open: 10 AM
=======
Open: 8 AM
>>>>>>> change-timing
$ echo "Open: 8 AM" > timing.txt
$ git add timing.txt
$ git commit --no-edit
[main e41ab0c] Merge branch 'change-timing'
```

Narrate: *HEAD* = the branch you are on (main, Krish's 10 AM); *change-timing* = the branch being merged (Dhruvi's 8 AM). The final commit has two parents; show it in `git log --oneline --graph`.

More story examples

Spectrum 2026 schedule. The robotics team and the coding-contest team both edit `schedule.html` on their own branches. They change *different* rows, so the merge is automatic. Then both change the *lunch break* row, which is a conflict. *Ask:* "Why did the first merge succeed and the second stop?"

Library system. A senior's half-finished "fine calculator" feature sits on an unmerged branch for months. A new student is about to delete it with `-D`. *Ask:* "What happens to that work? How could the team have avoided this?" (Push branches; merge small and often; name branches clearly.)

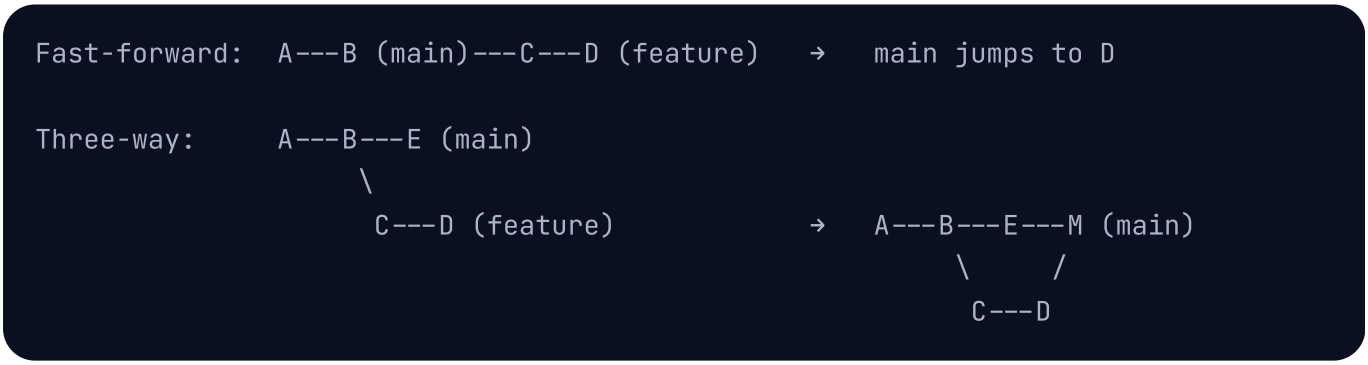
PCPM assignment. A student keeps the working version of `practical7.c` on `main` and tries a faster algorithm on `faster-sort`. The experiment fails, so `git switch main` and `git branch -D faster-sort`. *Use to show:* branches make experiments cheap.

Board diagrams to draw

Branch pointers and HEAD:



Fast-forward vs three-way:



Questions students ask

"Does creating a branch copy my project? Will it fill my disk?" No. A branch is a 41-byte file pointing to a commit.

"Which branch do I merge from?" Always switch to the branch that **receives** the work, then `git merge <other>`.

"Why didn't Git make a merge commit?" It was a fast-forward: the receiving branch had no new commits.

"Can a conflict delete my work?" No. Both versions are in the file. If you're unsure, `git merge --abort` restores everything.

"Why does the editor open after merge?" Git asks you to confirm the merge commit message. Save and close to accept (or use `--no-edit`).

"I deleted a branch by mistake!" If it was merged, nothing is lost: the commits are in `main`. If not, `git reflog` can usually find the commit for a few weeks (beyond this course; show it if asked).

"Should every small change have a branch?" In a team, yes: it keeps `main` working and makes review easy (Unit 5).

Misconceptions to correct

Misconception	Correction
"A branch is a copy of the files."	It's a pointer to a commit.
"Switching branches lost my work."	The work is committed on the other branch; switch back.
"`git merge x` merges the current branch into x."	It merges **x into** the current branch.
"A conflict means Git is broken / I did something wrong."	It means two people changed the same lines; Git asks a human.
"`git add` checks that markers are removed."	It doesn't. Check the file yourself.
"Deleting a branch deletes its commits."	Merged commits stay in history; only the label goes.

Self-check for faculty

1. Where is a branch stored, and what does it contain? `.git/refs/heads/<name>`, *one commit hash.*

2. What does `.git/HEAD` normally contain? *`ref: refs/heads/<branch>`*.
3. When is a merge a fast-forward? *When the receiving branch is an ancestor of the other branch (no new commits).*
4. What is a merge base? *The most recent common ancestor of the two branches.*
5. What makes a merge commit special? *It has two parents.*
6. What does Git write while a conflict is in progress? *Markers in the file, unmerged index entries, `MERGE_HEAD`*.
7. How do you cancel a conflicted merge? *`git merge --abort`*.
8. `git branch -d` refuses. Why, and what are the options? *Unmerged commits; merge first or force with `-D`*.
9. Classic equivalent of `git switch -c x`? *`git checkout -b x`*.
10. Delete a branch on GitHub from the terminal? *`git push origin --delete x`*.

10-minute recap

1. Draw A–B–C with `main` and `feature` labels; ask what moves when you commit on `feature`.
2. Ask: "Krish is on `main` and types `git merge feature-cart`. Into which branch does the work go?"
3. Show a conflict block; students label HEAD, divider, theirs, and say the four resolution steps.
4. One sentence each: `-d` vs `-D`; fast-forward vs merge commit.

Chapter 4: Advanced Git Commands

Sessions 10–12 · 6 hours · Missions M-12 – M-15 · Unit 4 lab

Learning outcomes

By the end of Unit 4, a student can:

1. Search history with `git log` and its options (`--oneline`, `--graph --all`, `-n`, `--author`, `--grep`, `-- <file>`).
2. Inspect a commit with `git show`, and refer to commits as `HEAD~n`, short hashes, branch and tag names.
3. Choose the right `git diff` (unstaged, `--staged`, two commits, two branches) and read a diff.
4. Park and restore unfinished work with `git stash` (push, list, pop, apply, drop).
5. Create lightweight and annotated tags and push them.
6. Choose the right undo command: `restore` / `checkout --` (uncommitted), `reset` (local commits; soft, mixed, hard), `revert` (pushed commits).
7. Visit an old commit (detached HEAD) and return.

Episode 4: Detective work

> One week before the Spectrum 2026 tech fest, Mr. Desai phones Hetvi: "Your app says a samosa costs zero rupees!"
>
> Nobody remembers changing prices. Hetvi opens Git Bash: `git log --oneline -- menu.txt`. Three commits touched the menu. `git show` on the second one reveals the culprit: *Update price list* by Meet changed `Samosa 15` to `Samosa 0`, a typo. Meet is embarrassed, but nobody had to search through files for an hour. "Next time," Hetvi says, "write *why* you changed prices in the message."
>
> The day before the fest, Krish is halfway through a new cart feature when the price bug resurfaces on the live demo. His cart code is broken, so he can't commit it. He types `git stash`, fixes the price, commits, pushes, then `git stash pop`, and his half-done cart is back.
>
> On fest day Hetvi tags the demonstrated version: `git tag -a v1.0 -m "Released at Spectrum 2026"` and pushes the tag.

> That evening Krish pushes *Change prices*, which accidentally sets everything to ₹99. Dhruvi has already pulled it. Krish wants to `git reset --hard`, but Hetvi stops him: "It's pushed. Use `git revert`, so nobody's history breaks."

The ideas in depth

Naming commits (revisions)

Form	Meaning
`a1b2c3d`	a commit by (unique prefix of) its hash; 4+ characters
`main`, `feature-cart`, `origin/main`	the commit a branch points to
`v1.0`	the commit a tag points to
`HEAD`	the current commit
`HEAD~1`, `HEAD~3`	1 or 3 first-parent steps back
`HEAD^2`	the second parent of a merge commit
`main..feature`	commits on `feature` that are not on `main`

`HEAD~n` is **relative**: after the next commit, `HEAD~2` means something different. That's why scripts and notes should record hashes or tags.

`git log` options worth knowing

`--oneline`, `--graph --all`, `-n 5` / `-5`, `--author="name"`, `--grep="text"` (messages), `-- path` (commits that changed a path), `--stat` (files changed), `-p` (full patches), `--since="2 weeks ago"`, `main..origin/main` (what you haven't merged yet).

Reading a diff

```
@@ -1,3 +1,4 @@
 Samosa 15
 Vada Pav 20
```

```
 Tea 10
+Coffee 20
```

`@@ -1,3 +1,4 @@` means "the old file's lines 1–3 became the new file's lines 1–4". Lines starting with a space are unchanged context, `-` removed, `+` added. A modified line appears as one `-` and one `+`.

The three stages explain the three `diff`s: `git diff` = working directory vs index; `git diff --staged` (`--cached`) = index vs HEAD; `git diff A B` = commit vs commit.

How stash works

`git stash` saves the current changes as special commits (one for the index, one for the working directory) referenced by `refs/stash`, then resets your files to HEAD. `stash@{0}` is the newest. `pop` = `apply` + `drop`. If applying would overwrite local changes, Git refuses and keeps the stash. By default untracked files are not stashed (`git stash -u` includes them).

Good uses: an urgent fix, pulling while you have uncommitted work, switching branches with unfinished changes. Bad use: long-term storage. Stashes are easy to forget, so use a branch instead.

Tags

- **Lightweight** (`git tag v1.0`): a ref in `refs/tags/` pointing to a commit, like a branch that never moves.
- **Annotated** (`git tag -a v1.0 -m "…"`): a tag **object** with tagger, date and message. Use these for releases; `git show v1.0` displays the message.
- Tags are pushed only explicitly (`git push origin v1.0` or `--tags`). On GitHub, tags appear under *Tags*, and *Releases* are built on them (Unit 5).
- Versioning convention many teams use: **semantic versioning** `MAJOR.MINOR.PATCH`, e.g. v1.2.3.

The undo toolbox

Situation	Command	What changes
Unwanted edit, not staged	`git restore f` / `git checkout -- f`	working file ← index
Staged by mistake	`git restore --staged f` / `git reset f`	index ← HEAD
Last commit wrong, not pushed, keep work staged	`git reset --soft HEAD~1`	branch moves
… keep work unstaged	`git reset HEAD~1` (mixed)	branch + index move
… throw work away	`git reset --hard HEAD~1`	branch + index + files
Commit already pushed	`git revert <commit>`	new commit added
Just fix the last message, not pushed	`git commit --amend -m "…"`	last commit replaced

Why `reset` is dangerous after pushing: it removes commits from *your* branch, but teammates still have them. Your next push is rejected (non-fast-forward); forcing it deletes their base and causes confusion. `revert` only adds a commit, so everybody can pull it normally.

Safety net (advanced, mention only): `git reflog` lists where HEAD has been, so even a `reset --hard` can usually be undone within a few weeks, unless the changes were never committed.

Detached HEAD

`git checkout <commit>` (or `git switch --detach <commit>`) points HEAD directly at a commit. You can look around, build and run the old code. New commits made here belong to no branch; create one (`git switch -c name`) before leaving if you want to keep them. `git switch main` (or `git switch -`) returns.

Worked walkthrough (real output)

Git Bash

```
$ git log --oneline -- cart.js
03474ce Refactor cart total
$ git show 03474ce
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
```

Narrate as detective work: *log* narrows the suspects, *show* examines the evidence, the author line tells you whom to ask.

More story examples

Spectrum 2026 schedule (tags). The printed schedule must match the website exactly. The team tags `schedule-final` when printing. When someone edits the site later, `git diff schedule-final` shows exactly what changed since the printout.

Library system (revert). A 2024 change to the fine calculation is wrong, and three batches have built on top of it. Resetting is impossible; `git revert <old commit>` undoes just that change. *Ask:* why is reset not an option?

PCPM assignment (stash). Halfway through `practical15.c`, the teacher asks for a quick fix to `practical14.c` before marking. `git stash`, fix, commit, `git stash pop`.

Thesis report (diff). Supervisor asks "what did you change since my last review?" `git diff review-1` answers instantly (with text formats like Markdown or LaTeX).

Board diagrams to draw

Reset modes vs the three stages:

```
                    branch   index   working files
reset --soft          ✓        -          -
reset (mixed)         ✓        ✓          -
reset --hard          ✓        ✓          ✓   ⚠️
```

Reset vs revert:

```
reset:   A---B---C (main)   →   A---B (main)              C is gone from main
revert:  A---B---C (main)   →   A---B---C---C' (main)     C' undoes C; nothing
disappears
```

Questions students ask

"How do I get out of `git log`?" Press `q`. It's the pager, not Git being stuck.

"Why is `git diff` empty when I changed a file?" Either the file is already staged (`git diff --staged`), or it's untracked (diff only shows tracked files).

"Is a stash saved on GitHub?" No. Stashes are local only and are not pushed.

"What's the difference between a tag and a branch?" Both are names for commits; a branch moves when you commit, a tag never moves.

"I did `git reset --hard` and lost my work!" If it was committed, `git reflog` can find it. If it was never committed, Git never stored it. This is why `--hard` deserves respect.

"Can I undo a revert?" Yes: revert the revert commit.

"What's the difference between `HEAD~1` and `HEAD^`?" For normal commits they're the same (the parent). They differ only for merge commits: `^2` means the second parent.

Misconceptions to correct

Misconception	Correction
"`git revert` deletes a commit."	It adds a new opposite commit; history keeps both.
"`git reset` is always safe."	`--hard` discards changes; any reset after pushing causes trouble for the team.
"Stash is a backup on GitHub."	It's local and temporary.
"Tags are pushed with `git push`."	They must be pushed explicitly.
"`git diff` shows everything I changed."	Only unstaged changes to tracked files.
"Detached HEAD means something is broken."	It just means you're looking at a commit instead of a branch.

Self-check for faculty

1. Show commits that changed `menu.txt`. *`git log -- menu.txt`*.
2. What does `@@ -1,3 +1,4 @@` mean? *Old lines 1–3 became new lines 1–4.*
3. Show staged changes. *`git diff --staged`*.
4. `stash pop` vs `apply`? *pop removes the stash after applying.*
5. Why prefer annotated tags for releases? *They record message, tagger and date.*
6. Push all tags. *`git push origin --tags`*.
7. Undo the last local commit but keep changes staged. *`git reset --soft HEAD~1`*.
8. Why not reset a pushed commit? *Teammates still have it; the next push is rejected or forcing breaks their history.*
9. What does `git revert` create? *A new commit applying the inverse change.*
10. Return from detached HEAD. *`git switch main`* *(or `git switch -`).*

10-minute recap

1. Give a "₹0" scenario; students name the three commands to find the culprit (`log -- file`, `show`, author line).

2. Draw the undo decision guide; students fill in the commands.
3. "You're halfway through a feature and need to fix a bug on main." (stash, or commit on a branch)
4. One sentence each: lightweight vs annotated tag; reset vs revert.

Chapter 5: Introduction to GitHub

Sessions 13–14 · 4 hours · Labs L-01 – L-04 · Unit 5 quiz · Final self-check

Learning outcomes

By the end of Unit 5, a student can:

1. Navigate a GitHub repository page: code, history, branches, README, clone URL, issues, pull requests, settings.
2. Create a repository with suitable options (visibility, README, `.gitignore`, licence) and manage it (collaborators, default branch, branch protection, releases).
3. Write a useful issue, label and assign it, and close it from a commit message.
4. Open, review and merge a pull request, and explain the three merge options.
5. Fork a repository, contribute through a pull request, and keep the fork up to date with `upstream`.
6. Use the course cheat sheet confidently.

Episode 5: Going public

> The Amroli Canteen app has been running for a month. Mr. Desai keeps phoning Hetvi with problems, and she loses track of them. "Write them on GitHub," she tells him, and creates his first **issue**: *Cart total wrong when the same item is added twice*. Krish fixes it and writes `fixes #3` in his commit message. When the commit reaches `main`, the issue closes itself. Mr. Desai is impressed.
>
> Then Hetvi turns on **branch protection**: nobody may push to `main` directly any more. Krish's next feature arrives as a **pull request**. Hetvi leaves a comment on line 12 ("this crashes if the cart is empty"), Krish pushes a fix to the same branch, the PR updates, and Hetvi approves and merges it.
>
> A month later, students from another college find the repository. They can't push to it, so they **fork** it, add a feature for their own canteen, and open a pull request back. Krish reviews code from strangers for the first time. "This is open source," says Hetvi.

The ideas in depth

GitHub is Git plus a website

Everything under *Code* is plain Git: commits, branches, tags. GitHub adds **collaboration features stored outside Git**: issues, pull request discussions, reviews, stars, Actions. They don't travel with `git clone`. Make sure students see the line between "Git does this" and "GitHub does this".

Visibility and access

- **Public:** anyone can see and clone; only collaborators can push.
- **Private:** only invited people can see it.
- **Collaborators** (personal repositories) get push access; organisations like `sbccas` use teams and roles.
- Pushing needs permission **and** authentication (browser sign-in, token or SSH).

Issues

Issues are numbered per repository (`#1`, `#2` ...), and pull requests share the same numbering. Labels, assignees and milestones organise them. Closing keywords (`close`, `closes`, `closed`, `fix`, `fixes`, `fixed`, `resolve`, `resolves`, `resolved`) followed by `#n` close the issue when the commit or PR reaches the **default branch**. Mention people with `@username`, and link issues by writing `#n`.

Pull requests

A PR is a request to merge a **head** branch (yours) into a **base** branch (usually `main`). It isn't a Git object. It lives on GitHub and is updated automatically whenever the head branch receives new commits.

Reviews come in three kinds: *Comment*, *Approve*, *Request changes*. With branch protection, a PR may require approvals and passing checks before merging.

Merge options:

Option	History on `main`	When teams choose it
Merge commit	all commits + a merge commit	keep full detail of each feature
Squash and merge	one commit per PR	tidy history; small PRs
Rebase and merge	commits replayed linearly	linear history without merge commits

Draft PRs signal work in progress. A PR that conflicts with `main` must be updated: merge `main` into the branch locally, resolve, push.

Forks

A fork is a server-side copy under the user's account, linked to the original. The standard remote names:

- `origin` = **your fork** (you push here),
- `upstream` = **the original** (you pull updates from here).

Keep a fork current with `git pull upstream main` then `git push`, or GitHub's **Sync fork** button. Contributors typically make a **branch per PR** in their fork, so several PRs can be open at once.

Open-source etiquette and licences

Without a licence, code is "all rights reserved" by default, even if it is public. **MIT** is a short, permissive licence; this course's content uses **CC BY-SA 4.0**. Good contributors read `CONTRIBUTING.md`, start with *good first issue* labels, keep PRs small and respond politely to review.

Running Lab L-04 smoothly

- Create the class practice repository before Session 14 (faculty guide, Appendix A) with a `students/` folder and a `CONTRIBUTING.md`.
- Each student adds **only** `students/<username>.md`, so PRs never conflict.
- Review on the projector: open *Pull requests*, pick one, show *Files changed*, comment, approve, **Squash and merge**. Students see their work land live.

- For a conflict demonstration, ask two volunteers to edit the same line of `README.md`.
- Common problems: PR opened against **their own fork** instead of the class repository (check *base repository*); committing the whole folder of another student's file; branch pushed but PR not created.

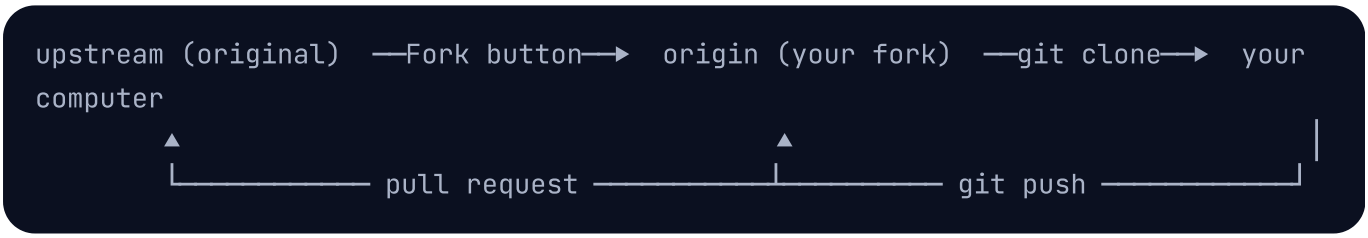
More story examples

Spectrum 2026 website (issues). Visitors report broken links as issues; volunteers pick issues labelled *good first issue*. *Ask:* how does an issue list help a team of volunteers who never meet?

Library system (pull requests). A junior proposes a new fine rule as a PR. The senior asks for changes because a test case is missing. *Ask:* what would have happened without review?

PCPM portfolio (fork). A student finds a classmate's C programs repository with a bug in the factorial program. They fork, fix it on a branch and open a PR. *Ask:* why can't they push directly?

Board diagrams to draw





Questions students ask

"Is a pull request the same as `git pull`?" No. Despite the name, a pull request is a GitHub *review-and-merge request*. `git pull` downloads commits.

"Do issues and PRs come with `git clone`?" No. They live on GitHub, not in the Git repository.

"Can I make a PR inside my own repository?" Yes: branch → push → PR into `main`. That's how teams with branch protection work.

"Someone forked my repository. Can they change my code?" No. They change their copy. Their changes reach yours only if you merge their PR.

"Should I make my college projects public?" Usually yes, as a portfolio, unless they contain secrets or your college forbids sharing assignments.

"Which merge button should we use?" Any is fine for this course. Many teams use *Squash and merge* for small PRs.

Misconceptions to correct

Misconception	Correction
"A pull request downloads code."	It asks to merge a branch, after review.
"Forking and cloning are the same."	A fork is on GitHub; a clone is on your computer.
"Public means anyone can push."	Anyone can read; only collaborators can push.
"Closing an issue fixes the bug."	The fix is a commit; the keyword just links and closes the issue.
"No licence means free to use."	No licence means all rights reserved.

Self-check for faculty

1. Name three GitHub features that are not part of Git. *Issues, pull requests, reviews (also stars, Actions).*
2. What happens when `fixes #3` reaches a feature branch only? *Nothing until it reaches the default branch.*
3. Head vs base in a PR? *Head = the branch with changes; base = where they will be merged.*
4. What does *Squash and merge* produce? *One commit on the base branch.*
5. In a fork workflow, what are `origin` and `upstream`? *Your fork; the original.*
6. How do you keep a fork updated? *`git pull upstream main` then push, or Sync fork.*
7. How do you stop direct pushes to `main`? *Branch protection rule requiring a PR.*
8. Why should each L-04 student change only their own file? *To avoid conflicts between PRs.*
9. Default rights on public code without a licence? *All rights reserved.*
10. Where does a release come from? *A tag, published with notes under Releases.*

10-minute recap

1. Show a repository page; students point out the clone URL, history, issues and PR tabs.
2. "Mr. Desai found a bug." Students write the issue title and the commit message that closes it.
3. Draw the PR flow; students fill in the six steps.
4. One sentence each: fork vs clone; `origin` vs `upstream`.

Appendices

A. Course at a glance

Unit	Sessions	Key commands	Practice	Badge
1 · Version control	1–2	`config`, `help`, `--version`	M-01, M-02, L-01	🧭 Version Control Explorer
2 · Getting started	3–6	`init`, `status`, `add`, `commit`, `remote`, `push`, `clone`, `fetch`, `pull`	M-03 – M-07, Unit 2 lab	🚀 Repository Launcher
3 · Branching	7–9	`branch`, `switch`, `checkout`, `merge`, `branch -d`	M-08 – M-11, conflict game, Unit 3 lab	🌿 Branch Master · ⚔️ Conflict Crusher
4 · Advanced	10–12	`log`, `show`, `diff`, `stash`, `tag`, `restore`, `reset`, `revert`	M-12 – M-15, time travel, Unit 4 lab	🕵️ History Detective · ⏳ Time Traveller
5 · GitHub	13–14	issues, pull requests, forks, `upstream`	L-02 – L-04	🤝 Open Source Collaborator
Assessment	15	all	practical, self-check, showcase	🏆 Course Complete

B. Where everything is on the course site

You need	Go to
Student lessons	`/unit-1/` … `/unit-5/`
Real Git in the browser	`/playground/` (sandboxes, Graph, Files, GitHub tabs, ⏳ time travel)
Story missions	`/missions/` (each in its own sandbox; Restart resets it)
Real-GitHub labs	`/labs/`
Command reference	`/cheatsheet/` (printable)
Terms	`/glossary/` (also tap-to-define inside lessons)
Final practice test	`/self-check/`
Slides, session plans, this handbook	`/teach/`
Students moving devices	`/about/#progress` (export / import progress)

Pre-filling a command for students: link to `/playground/?cmd=git%20status` and the command appears ready to run.

C. Story quick reference

Character	Remember them by	Unit moments
Krish Patel	fast, makes mistakes	first commit (U2), `feature-menu` (U3), stash emergency and pushed ₹99 mistake (U4), PR reviewed (U5)
Dhruvi Patel	careful, pushes late at night	overnight push → fetch/pull (U2), 8 AM opening time conflict (U3)
Meet Shah	quiet, laptop crash	crash → why Git (U1), "Update price list" typo → detective (U4)
Hetvi Desai	team lead	introduces Git (U1), team rules (U3), tags v1.0 (U4), branch protection and reviews (U5)
Mr. Desai	canteen manager	reports the ₹0 bug (U4) and issues (U5)

D. Further reading for faculty

- **Pro Git** (Scott Chacon & Ben Straub), free online at git-scm.com/book: the standard reference; chapters 1–3 and 6 match this course.
- **GitHub Docs** (docs.github.com): repositories, issues, pull requests, forks.
- **GitHub Skills** (skills.github.com): free interactive GitHub exercises for students who finish early.
- **Learn Git Branching** (learngitbranching.js.org): visual branching puzzles; good extension for Unit 3.
- **Git reference manual** (git-scm.com/docs): the same pages as `git help <command>`.