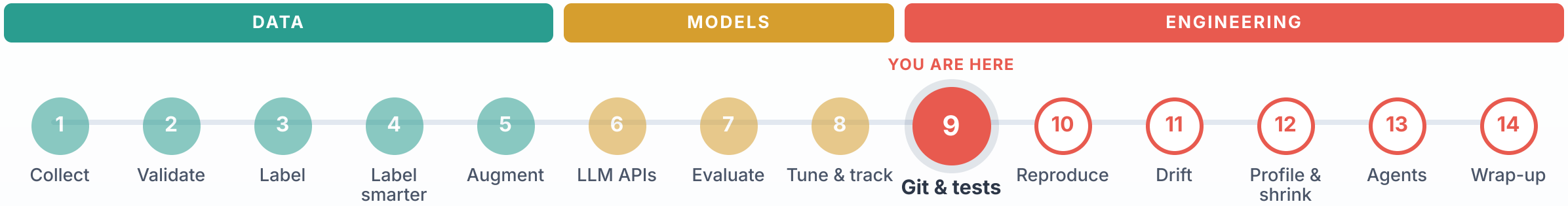
Git, GitHub & Testing for AI Projects

Week 9 · CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra
IIT Gandhinagar

Live demo: GitQuest + `lecture-demos/week09/git_followalong.sh`

Where we are · Week 9 of 14



TODAY'S QUESTION

Can a team change the code without breaking it?

WHAT OUR PROJECT GAINS TODAY

A GitHub repo with history, tests and CI

NEXT

Week 10 · Reproduce: Can anyone re-run it, on any machine?

Today's Plan

Part	Question	Tools
1 · Time machine	How do I never lose work again?	`status` `diff` `add` `commit` `restore`
2 · Parallel universes	How do I try risky ideas safely?	`switch -c` `merge` conflicts
3 · Teamwork	How do 4 people edit one project?	GitHub, `push` `pull`, pull requests
4 · ML hygiene	What must *never* go into git?	`.gitignore`, `.env`, HF Hub, `nbstripout`
5 · Safety nets	How do we know a change didn't break anything?	`pytest`, GitHub Actions

YOU ALREADY KNOW THIS

Week 8: we logged every training run in TrackIO. Today we version the *code* that produced those runs, so a result can always be traced to the exact code behind it.

Part 0: The Problem

Why do we need version control?

The Dark Ages




```
train.py   train_v2.py   train_v2_FINAL.py   train_v2_FINAL_actually_final.py
train_v2_FINAL_actually_final_nipun_edits.py   train_v2_FINAL_ab_fixed(1).py
```

Which one produced the 91% accuracy you reported last week? **Nobody knows.**

Three Real Disasters (All Preventable)

Disaster	What happened	What would have saved you
"It worked yesterday"	A small edit broke training; the old version is gone	A commit before the edit
The overwrite	Two teammates emailed `model.py` back and forth; one copy won	Branches + merge
The leaked key	`GEMINI_API_KEY = "AIza..."` pushed to a public repo	`.env` + `.gitignore`

PREDICT FIRST

A real API key is pushed to a **public** GitHub repo. How long until automated bots find it: a month, a day, or a few minutes?

Answer: Minutes

Bots continuously scan every public push for strings that look like keys.

- GitHub runs **secret scanning** and may alert the provider, but don't count on it
- Deleting the file in a new commit **does not help**: the key is still in history
- **The only real fix:** revoke/rotate the key on the provider's dashboard, *then* clean up

> A secret that touched a public repo is a burned secret. Rotate first, ask questions later.

Enter Git: Four Superpowers

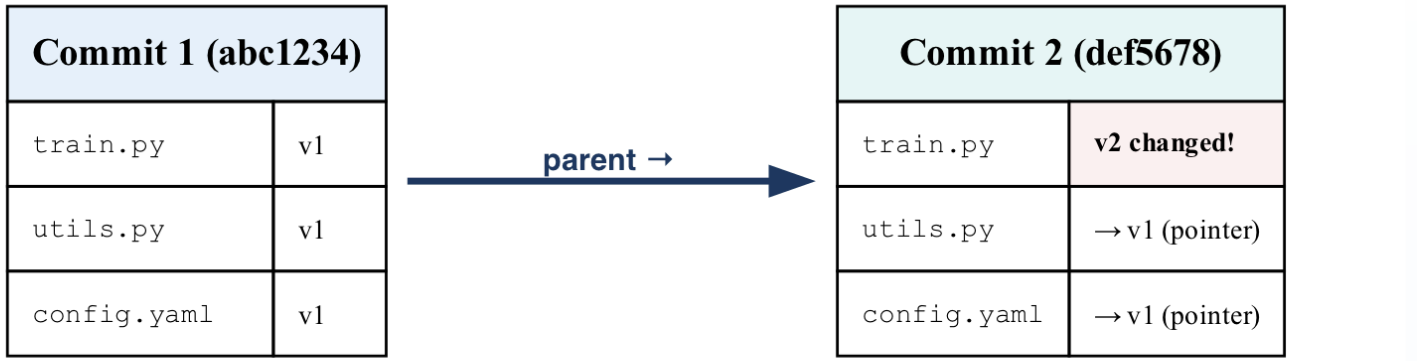
Without Git	With Git
Five files named `*_FINAL_v3`	One file, full history
"Which version had that feature?"	`git log` : every change, with a reason
"I broke everything!"	`git restore` : go back in time
"Let me try something risky…"	`git switch -c` : a parallel universe
"Who changed this line, and why?"	`git blame` + commit message

1. **Time machine** · 2. **Parallel universes** · 3. **Collaboration** · 4. **Safety net** (with tests + CI)

Part 1: The Personal Time Machine

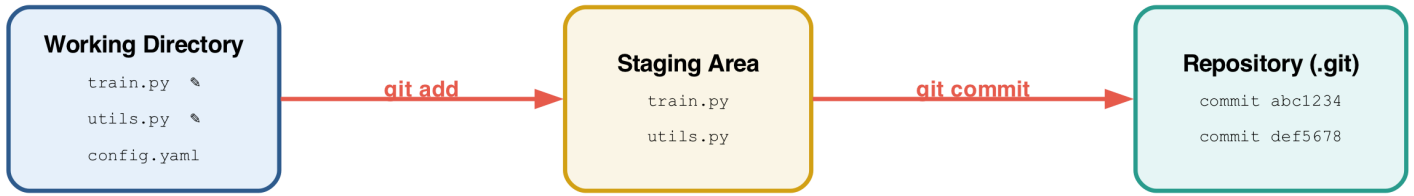
Snapshots, the three areas, and undo

Git Stores Snapshots, Not Diffs



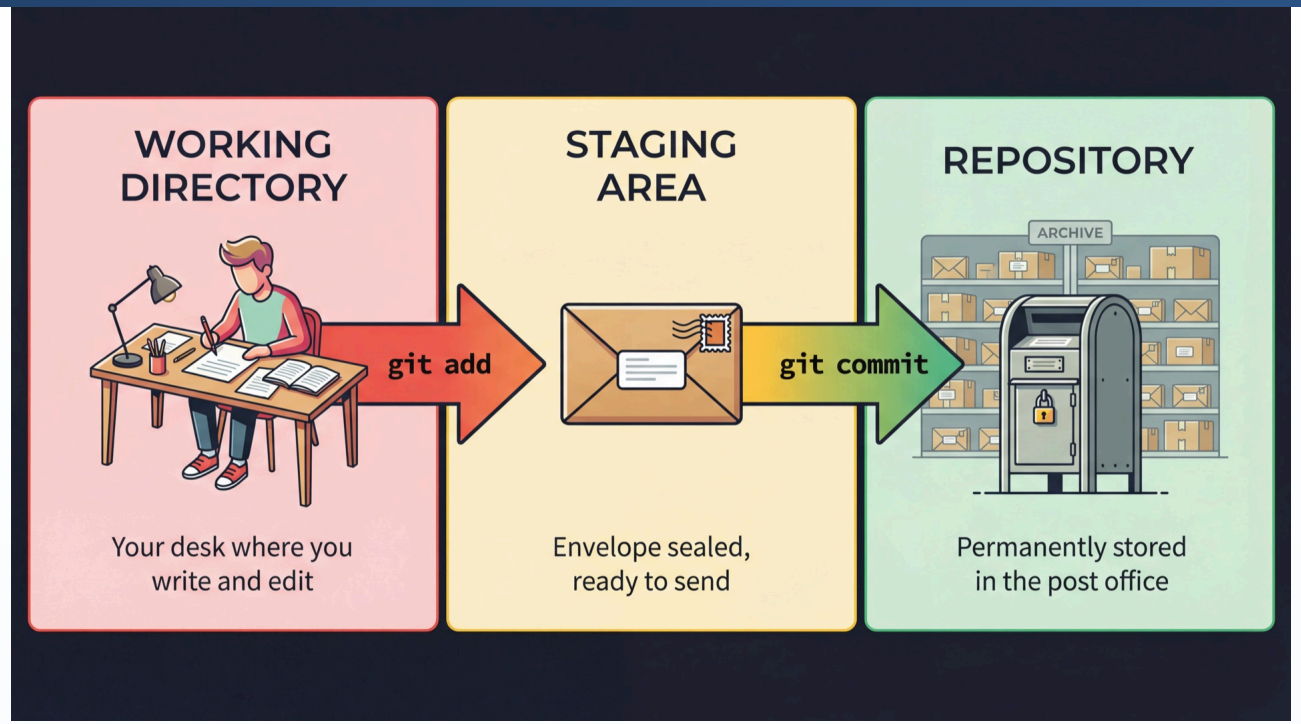
Each **commit** is a snapshot of the whole project at a moment in time, plus a pointer to the previous commit. Unchanged files are not copied again: Git just reuses them.

The Three Areas of Git



Working directory	Staging area	Repository
Your desk	The envelope	The vault
You edit files here	`git add` puts changes here	`git commit` seals them in

The Letter Analogy



Step	Analogy	Git command
1	Write the letter	Edit your files
2	Put chosen pages in the envelope	`git add data.py`
3	Seal and post it	`git commit -m "Validate runtimes"`

The envelope lets you **choose** what goes into each snapshot.

One-Time Setup

```
git config --global user.name  "Your Name"
git config --global user.email "you@iitgn.ac.in"
git config --global init.defaultBranch main
```

Then, in each new project:

```
mkdir movie-predictor && cd movie-predictor
git init                # creates the hidden .git/ folder: the vault
```

> **TAKEAWAY**
>
> Everything Git knows lives in `.git/`. Delete that folder and the history is gone; your files stay.

The Core Loop

```
git status              # 1. What changed?
git diff                # 2. Show me exactly how
git add clean.py        # 3. Put it in the envelope
git commit -m "Reject movies with negative runtime"   # 4. Seal it
git log --oneline       # 5. See the history
```

```
$ git log --oneline
3f9c2a1 Reject movies with negative runtime
a1b2c3d Add OMDb collection script
7e8d9f0 Initial commit: README and .gitignore
```

Rhythm: edit → `status` → `diff` → `add` → `commit`. Repeat dozens of times a day.

The Life of a File



Untracked → `add` → staged → `commit` → unmodified → edit → modified → `add` → ...

Predict the Output

PREDICT FIRST

```
git add model.py          # stage the current version
echo "lr = 0.01" >> model.py
git status
```

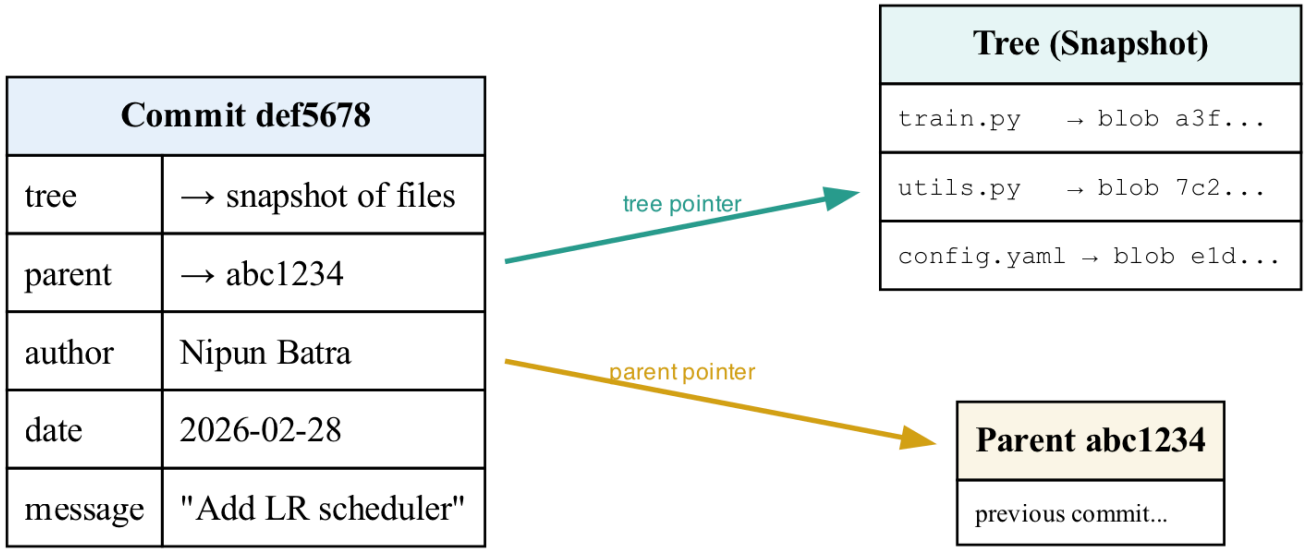
Is `model.py` listed as *staged*, *modified*, or **both**?

Answer: Both

```
Changes to be committed:
        modified:   model.py      ← the version you added
Changes not staged for commit:
        modified:   model.py      ← the newer edit
```

`git add` stages a **snapshot of the file at that moment**, not "the file forever".
Run `git add model.py` again to include the new line.

Anatomy of a Commit



Hash (unique ID) · parent(s) · author · date · message · snapshot. Change any byte, and the hash changes.

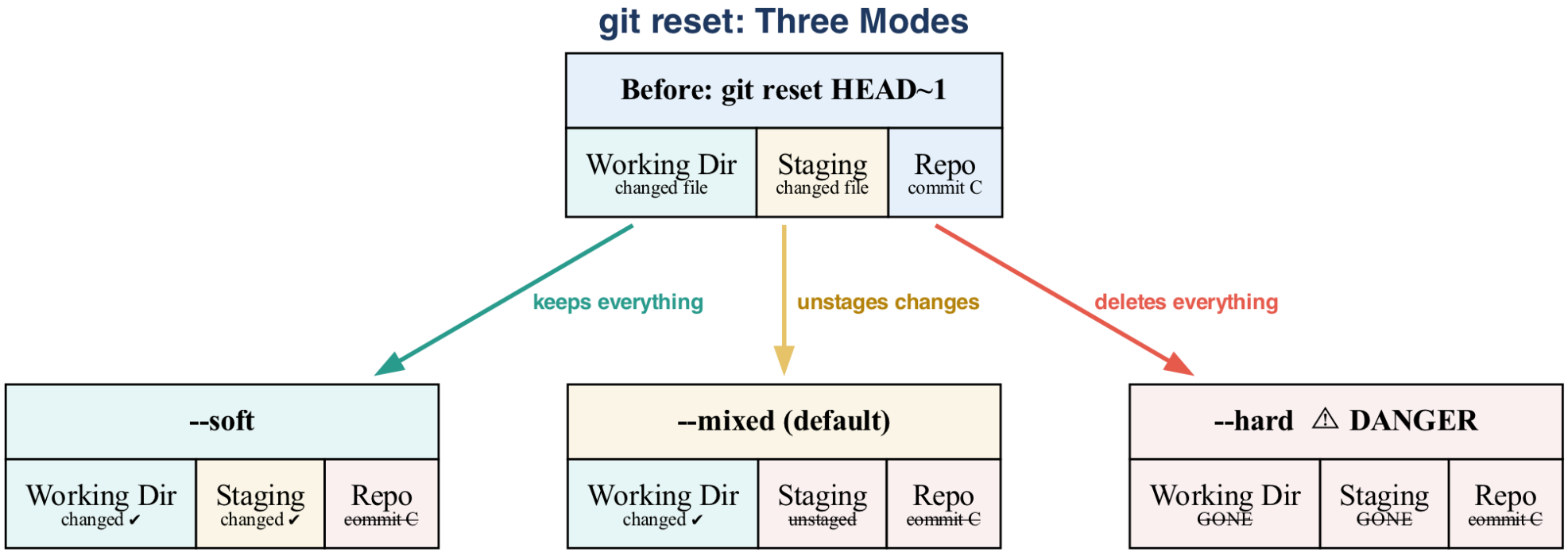
Writing Good Commit Messages

Rule: complete the sentence *"If applied, this commit will ___."*

Bad	Good
`stuff`	`Add stratified 5-fold CV to evaluate.py`
`fix`	`Fix leakage: fit scaler inside the CV pipeline`
`changes`	`Drop movies with missing box office before training`
`final!!!`	`Log git commit hash with every TrackIO run`

Small commits, one idea each. You will thank yourself in week 13.

git reset: Three Modes



`--soft` keeps changes staged · `--mixed` (default) keeps them in your files · `--hard` **throws them away**.

Undo: Your Time-Travel Toolkit

I want to...	Command
Throw away edits to a file	`git restore model.py`
Unstage a file (keep edits)	`git restore --staged model.py`
Undo my last commit, keep the changes	`git reset --soft HEAD~1`
Undo a commit that is **already pushed**	`git revert <hash>` (adds a new "undo" commit)
Find "lost" commits after a bad reset	`git reflog`

⌨ Live Demo 1: GitQuest Levels 1–6

Open nipunbatra.github.io/learn-git

- Watch the three areas update after every command
- Predict **before** each command: what moves?

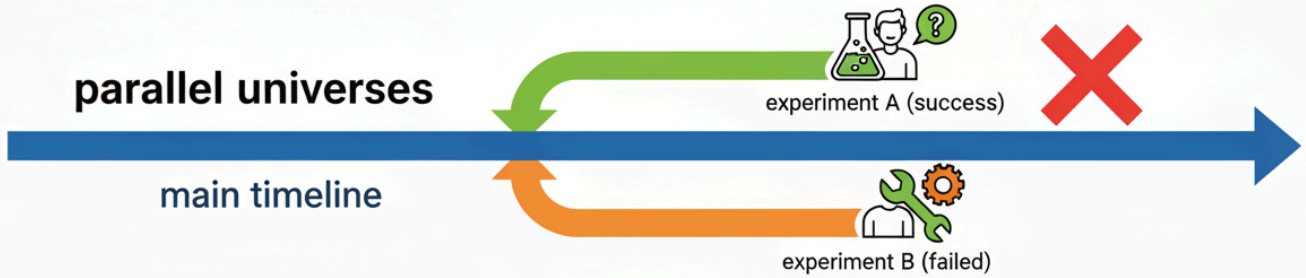
Or in your own terminal:

```
bash lecture-demos/week09/git_followalong.sh
# Acts 1-3: the mess, init, the core loop
```

Part 2: Parallel Universes

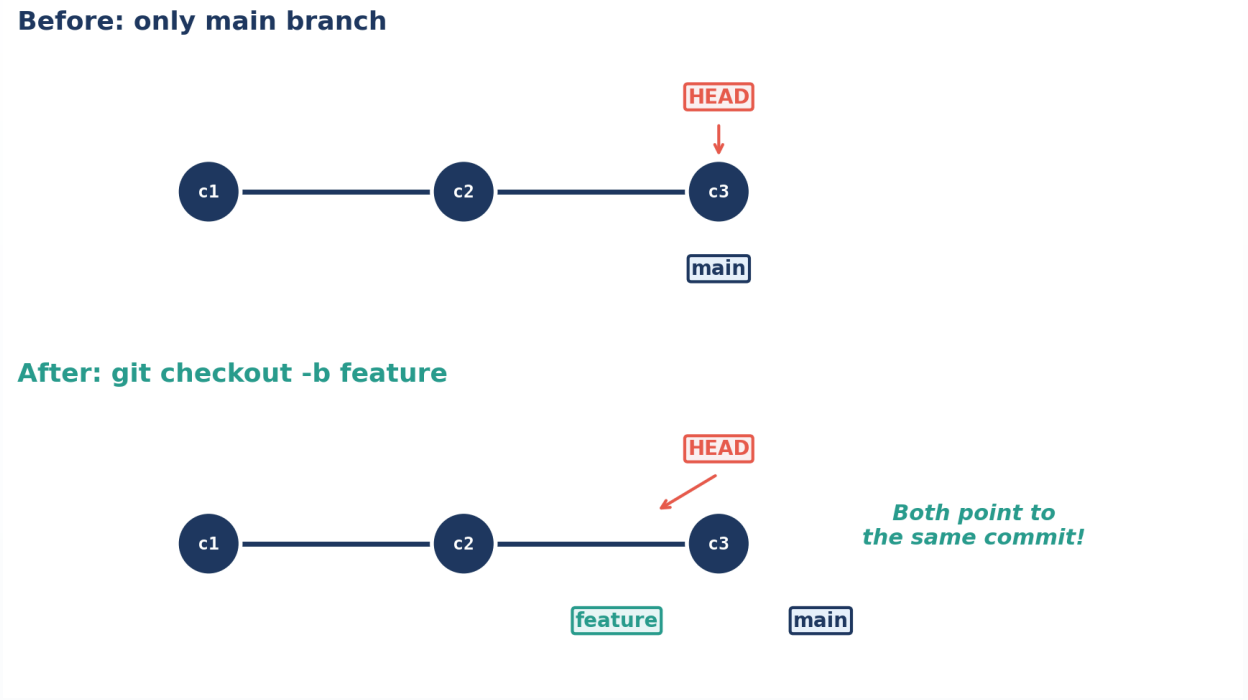
Branches, merges, and conflicts

Branches: The "What If?" Machine



- `main` always holds working code
- Try a risky idea on `exp/xgboost`
- It works → merge it. It fails → delete the branch. No harm done.

A Branch Is Just a Pointer



A branch is a **sticky note on a commit**, not a copy of your files. `HEAD` marks where you are.

```
git switch -c exp/xgboost     # create + switch (older: git checkout -b)
git switch main               # switch back
git branch                    # list branches; * marks the current one
```

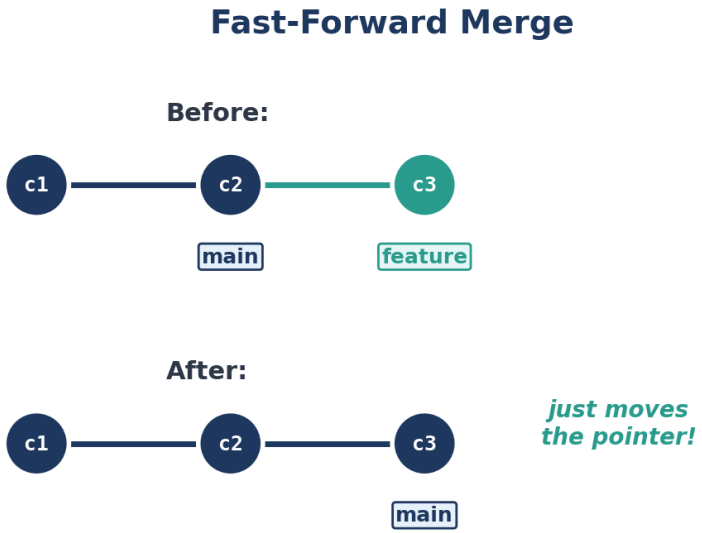
Branches for ML Experiments

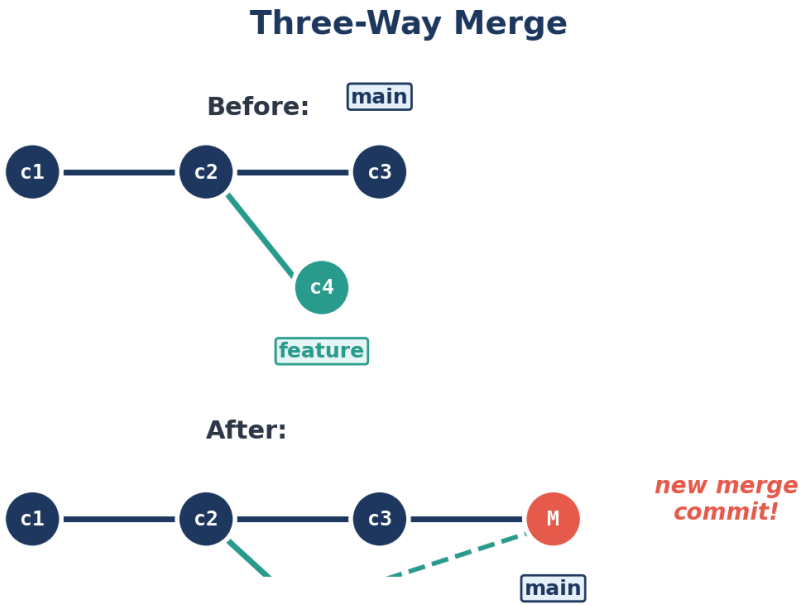
```
git switch -c exp/lr-sweep
# edit train.py, run the sweep, log to TrackIO ...
git commit -am "Sweep lr in {1e-4, 1e-3, 1e-2}; 1e-3 wins (F1 0.81)"
```

Log the commit hash **with every run**, so every number on your dashboard maps to exact code:

```
import subprocess, trackio
commit = subprocess.check_output(["git", "rev-parse", "--short", "HEAD"], text=True).strip()
trackio.init(project="movie-predictor", config={"lr": 1e-3, "git_commit": commit})
```

Merging: Two Kinds





Fast-forward	Three-way merge
`main` has not moved since you branched	Both branches have new commits
Git just slides the pointer forward	Git creates a **merge commit** with two parents

Predict: Which Merge?

PREDICT FIRST

1. You branch `feat/augment` from `main` and make 3 commits. Nobody touches `main`. You run `git merge feat/augment` on `main`.
2. Same, but your teammate pushed 1 commit to `main` in the meantime.

Fast-forward or three-way merge, in each case?

Answer: (1) fast-forward: the pointer just moves. (2) three-way: the histories have diverged.

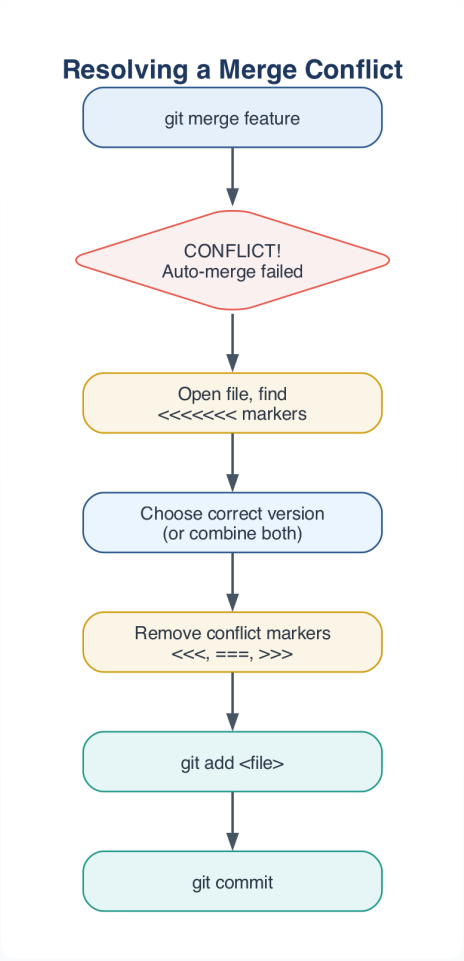
Merge Conflicts: Two People, One Line

```
def learning_rate():
<<<<<<< HEAD
    return 0.001    # main says 0.001
=======
    return 0.01     # the feature branch says 0.01
>>>>>>> feat/tuning
```

Marker	Meaning
`<<<<<<< HEAD`	Your current branch's version
`=======`	Divider
`>>>>>>> feat/tuning`	The incoming branch's version

Git is not broken. Two edits touched the same line, and a human has to decide.

Resolving a Conflict

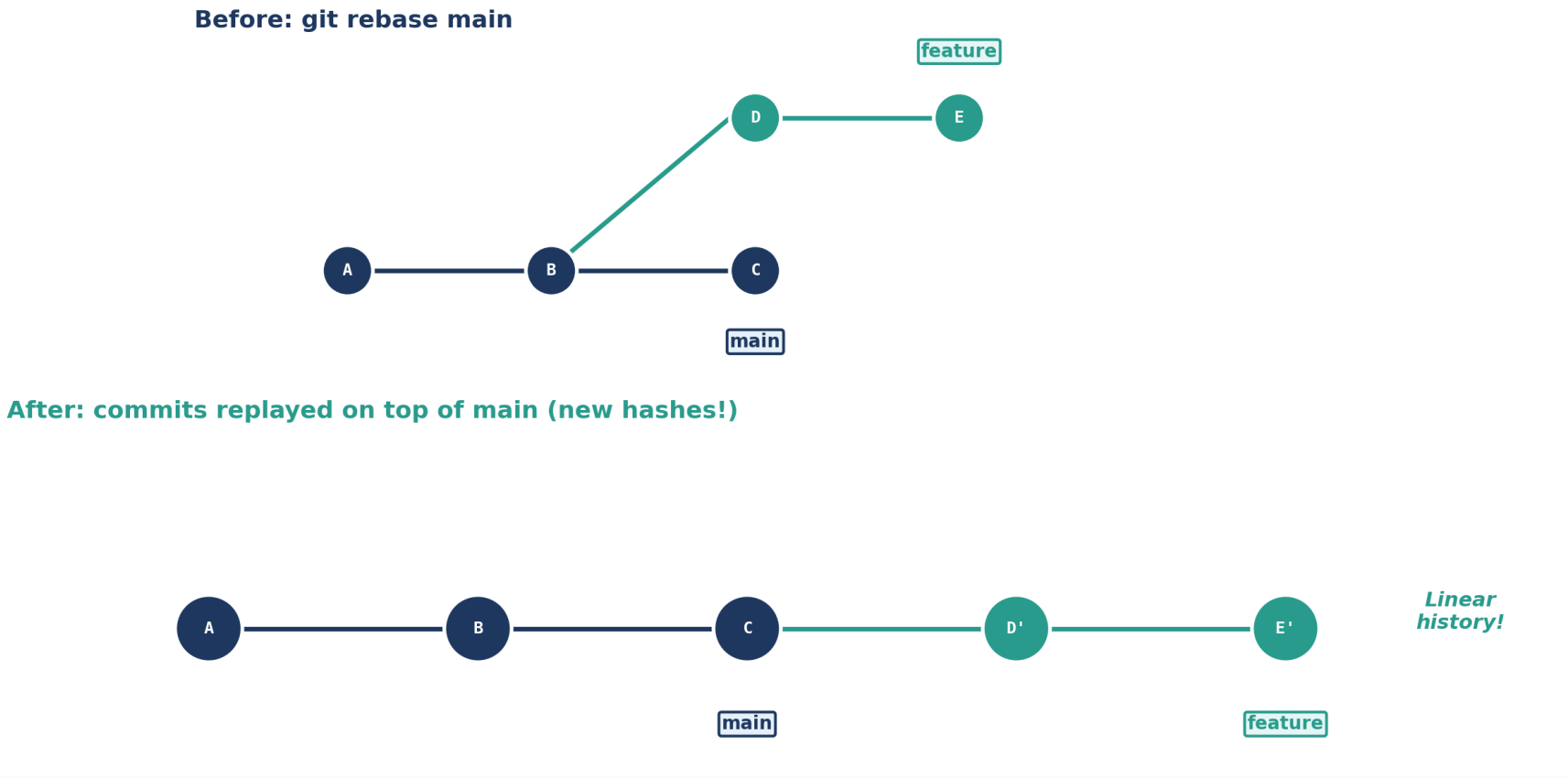


1. Open the file, find the markers
2. Decide: keep one, the other, or combine. **Delete the markers.**
3. `git add model.py`
4. `git commit`

Read both versions and understand the *intent* before choosing.

`git merge --abort` gets you out if you panic.

Rebase (In One Slide)



`git rebase main` replays your commits on top of the latest `main`, giving a straight-line history.

Golden rule: never rebase commits that others have already pulled. When in doubt, merge.

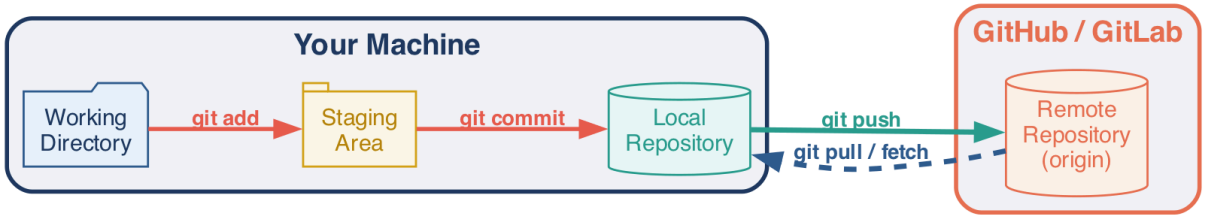
⌨ Live Demo 2: GitQuest Levels 7–12

- Create a branch, commit on both sides, predict: fast-forward or merge commit?
- Manufacture a conflict on `learning_rate()` and resolve it live
- Follow-along: **Acts 4–8** in `git_followalong.sh`

Part 3: Teamwork with GitHub

Remotes and pull requests

Git vs GitHub: Camera vs YouTube



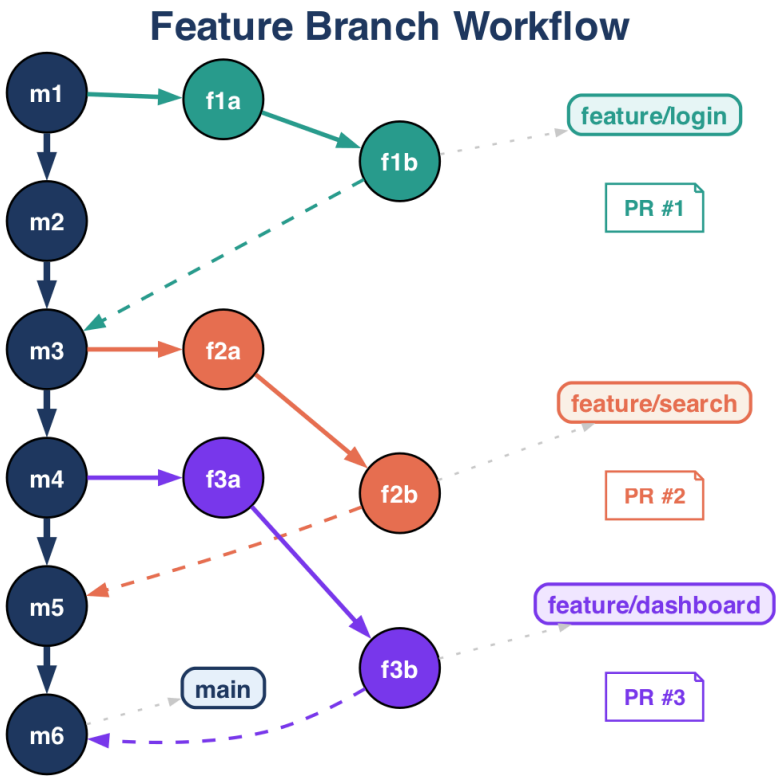
Git	GitHub
The **camera**: records history locally	**YouTube**: hosts and shares it
Works offline, free, open source	A service (GitLab, Bitbucket and Hugging Face are others)

Four Commands Talk to the Remote

Command	Direction	What it does
`git clone <url>`	remote → local	First full download (history included)
`git fetch`	remote → local	Download new commits, **don't touch my files**
`git pull`	remote → local	`fetch` + merge into my branch
`git push`	local → remote	Upload my commits

```
git remote add origin https://github.com/<you>/movie-predictor.git
git push -u origin main      # first push links local main to origin/main
```

The Feature-Branch Workflow



1. **Branch** from `main`
2. **Commit** small steps
3. **Push** the branch
4. Open a **pull request**
5. **Review** + automatic **checks**
6. **Merge**, delete the branch

`main` only receives reviewed, tested changes.

Anatomy of a Pull Request

Part	Why it matters
Title + description	*What* changed and *why*; link the issue
Diff	Reviewers read every changed line
Review comments	Questions, suggestions, approvals
Checks (CI)	Tests run automatically; a red ✗ blocks the merge

```
git switch -c fix/runtime-units
git commit -am "Convert runtime from seconds to minutes in loader"
git push -u origin fix/runtime-units      # GitHub prints a "Create PR" link
```

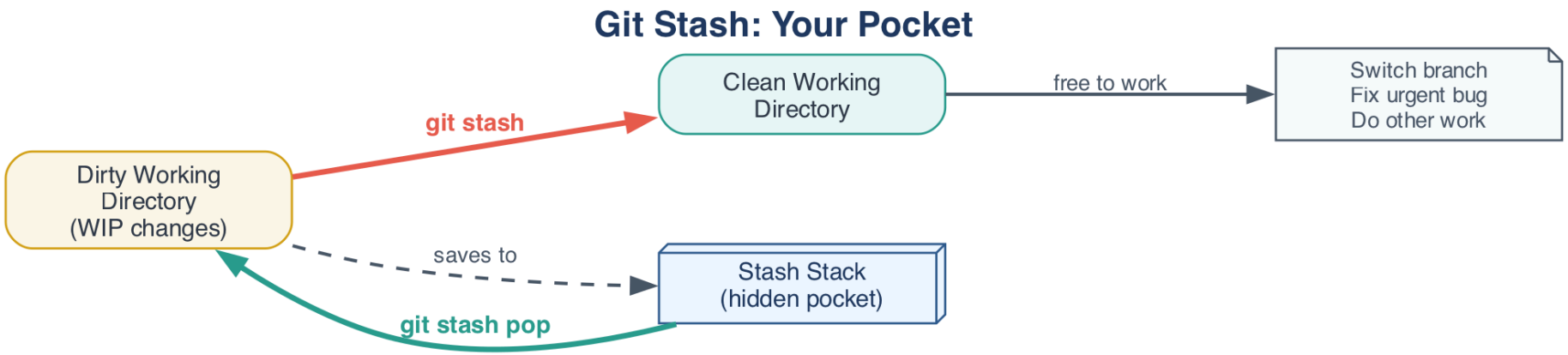
Reviewing an ML Pull Request

Ask these before clicking **Approve**:

- Does any preprocessing step `fit` on test data? (Week 7: leakage)
- Are random seeds fixed and logged? (Week 10 preview)
- Does it commit data, model weights, notebook outputs or secrets? (Part 4)
- Did the metric change, and is the comparison fair (same split, same seed)?
- Is there a test for the new behaviour? (Part 5)

> **TAKEAWAY**
>
> A review is a conversation about correctness, not a rubber stamp.

git stash: Pause Button




```
git stash          # park half-finished work
git switch main    # fix the urgent bug, commit, come back
git switch feat/x
git stash pop      # resume where you left off
```

⌨ Live Demo 3: Your First Pull Request

- GitQuest levels 13–18: remotes, fetch vs pull, a remote that is ahead of you
- Follow-along **Acts 9–10**: stash, push to GitHub, clone, pull
- Then: a real PR on the class demo repo. Branch, change one line, push, open the PR, request a review

Part 4: What Never Goes into Git

ML projects have special hazards

The .gitignore for an ML Project

```
# secrets
.env
*.key
# data and models
data/
*.csv
*.parquet
models/
*.pt
*.onnx
```

```
# logs and caches
trackio/
mlruns/
__pycache__/
.ipynb_checkpoints/
.venv/
# OS junk
.DS_Store
```

Create it **before** your first commit. `git status` should never show data or weights.

Secrets: Use .env, Never Code

```
# .env  (git-ignored, lives only on your machine)
GEMINI_API_KEY=AIza...your-key...
```

```
from dotenv import load_dotenv     # pip install python-dotenv
import os
load_dotenv()
client = genai.Client(api_key=os.environ["GEMINI_API_KEY"])
```

Commit a `.env.example` with empty values, so teammates know which variables to set.
On Colab, use the **Secrets** panel (🔑) instead of a `.env` file.

Predict: Why Is This Repo 2 GB?

PREDICT FIRST

A course repo has 300 slide images and 13 decks, yet `git clone` downloads **2 GB**. The images are now 60 MB in total. Why is the clone still huge?

Answer: history never forgets. Every 14 MB image ever committed is still in `.git/`, even after it is replaced. (This is our own course repo.)

> Big binary files in git are forever. Keep them out from day one.

Where Data and Models Should Live

Artifact	Put it in	Why
Code, configs, small CSVs (< 1 MB)	Git	Diffs, review, history
Datasets	Hugging Face Hub datasets, DVC, cloud bucket	Versioned and streamable
Model weights	Hugging Face Hub, git-lfs	Large binaries
Experiment logs	TrackIO / MLflow	Built for metrics
Secrets	`.env`, Colab Secrets, GitHub Secrets	Never in history

GitHub warns above 50 MB per file and **rejects files above 100 MB**.

Notebooks in Git

Notebook outputs (images, tables) make diffs unreadable and repos heavy.

```
pip install nbstripout
nbstripout --install          # strip outputs automatically on every commit
```

Better still: move reusable code into `.py` modules and keep notebooks thin, so the logic can be **imported and tested**.

Part 5: Safety Nets

Tests and continuous integration

ML Code Fails Silently

> **YOU ALREADY KNOW THIS**
>
> Week 2: `pd.to_numeric(errors="coerce")` silently turned 479 bad box-office values into `NaN`. No error, just wrong numbers.

A program that crashes is easy to fix. ML bugs usually **don't crash**:

- a scaler fitted on test data → accuracy looks great
- runtime in seconds instead of minutes → the model learns nonsense
- shuffled labels → training "works", predictions are random

Tests turn silent assumptions into loud failures.

pytest in 60 Seconds

```python
# clean.py
def clean_runtime(text: str) -> int | None:
    """'148 min' -> 148; 'N/A' -> None"""
    if text == "N/A":
        return None
    return int(text.split()[0])
```

```python
# test_clean.py   (pytest finds files named test_*.py)
from clean import clean_runtime

def test_parses_minutes():
    assert clean_runtime("148 min") == 148

def test_missing_is_none():
    assert clean_runtime("N/A") is None
```

```
pytest -q        # ..  2 passed in 0.01s
```

What to Test in an ML Project

Test	Example assertion
Data contract	every row passes the Pydantic schema; `runtime` in 1–400
No leakage	`set(train_ids).isdisjoint(test_ids)`
Shapes	`model.predict(X[:5]).shape == (5,)`
Probabilities	`np.allclose(model.predict_proba(X).sum(axis=1), 1)`
Determinism	two runs with `seed=42` give identical predictions
Sanity baseline	the model beats `DummyClassifier` on a tiny fixture set

Tests should be **fast** (seconds) and use **tiny** fixture data, not the full dataset.

Predict: Which Test Catches It?

PREDICT FIRST

A teammate's PR changes the loader so `runtime` is now in **seconds**. Training still runs and accuracy drops slightly. Which test from the previous slide fails?

Answer: the **data contract** (`runtime` in 1–400 minutes): 8,880 seconds is out of range. The other tests still pass. That is why we test *data*, not just code.

Continuous Integration: Tests on Every Push

```yaml
# .github/workflows/ci.yml
name: tests
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7
      - uses: actions/setup-python@v7
        with:
          python-version: "3.12"
      - run: pip install -r requirements.txt pytest
      - run: pytest -q
```

Every push and every PR gets a ✓ or ✗ from a clean machine, not just *your* laptop.

What CI Gives a Team

- **"Works on my machine" dies:** tests run on a fresh Ubuntu VM every time
- **Red ✗ on the PR** → reviewers see it before merging
- **Branch protection:** make `main` require passing checks + one approval
- **History of green builds:** you know exactly which commit broke things

> **YOU ALREADY KNOW THIS**
>
> Next week (Reproducibility): the same idea goes one step further. Pin the *whole environment* with `uv` and Docker, so "a fresh machine" means *exactly* our machine.

Git + AI Coding Assistants

Coding agents (Claude Code, Copilot, Cursor, Gemini CLI) edit many files at once. Git is your seatbelt:

1. **Commit before** you let an agent loose: a clean starting point
2. Let it work on a **branch**
3. **Read** `git diff` before accepting. You are the reviewer.
4. **Run the tests.** Tests are the contract the agent must satisfy
5. Bad result? `git restore .` or `git switch main`, and nothing is lost

> TAKEAWAY
>
> Agents make writing code cheap. Git and tests make *checking* code cheap.

Quick Reference

Task	Command
What changed?	`git status` · `git diff`
Save a snapshot	`git add <files>` → `git commit -m "why"`
History	`git log --oneline --graph`
New branch / switch	`git switch -c <name>` · `git switch <name>`
Merge	`git merge <branch>`
Discard / unstage	`git restore <file>` · `git restore --staged <file>`
Undo pushed commit	`git revert <hash>`
Park work	`git stash` · `git stash pop`
Sync	`git clone` · `git fetch` · `git pull` · `git push`
Run tests	`pytest -q`

This Week's Lab

1. Turn your movie project into a Git repo with a proper `.gitignore` and `.env.example`
2. Make 5+ meaningful commits; one experiment on a branch, merged back
3. Resolve a planted merge conflict
4. Write 4 tests: data contract, no-leakage, prediction shape, determinism
5. Push to GitHub, add `ci.yml`, open a pull request and get a green ✓
6. Review a classmate's PR using the ML review checklist

The Golden Rule

As long as you **commit**, you can almost always get your work back.

Commit early · commit often · write why · test what matters.

*Next week: version the **environment**. `uv`, seeds and Docker.*