Git data model

Key takeaways

1. Git stores immutable snapshots and movable names that point into them. Learn that model and branches, rebase, merge, and worktrees stop being separate mysteries.

2. Rebase your own branch, then fast-forward it in. A linear history has one order of events, which keeps review, bisect, and revert simple. Save merge commits for shared branches.

3. Give concurrent agents separate worktrees so their edits cannot collide, and push before leaving a machine: a commit in only one clone is invisible to everyone else.

Common misunderstandings of what git actually stores

Which claim would you defend right now? Choose before the evidence.

“A branch or a worktree is a copy of the files, so several of them cost real disk.”

“Rebase and merge end in the same history, so the choice is only cosmetic.”

“Work that is committed is safe work.”

Keep your choice. Look for the mechanism or counterexample that would make it fail.

Learn the model, not the commands

- Agents edit quickly; git is how you see, isolate, review, and recover those edits.
- For this course, understand the model: objects, commits, refs, working trees, and remotes.
- Command syntax is lookup material. The mental model is what keeps agent work safe.

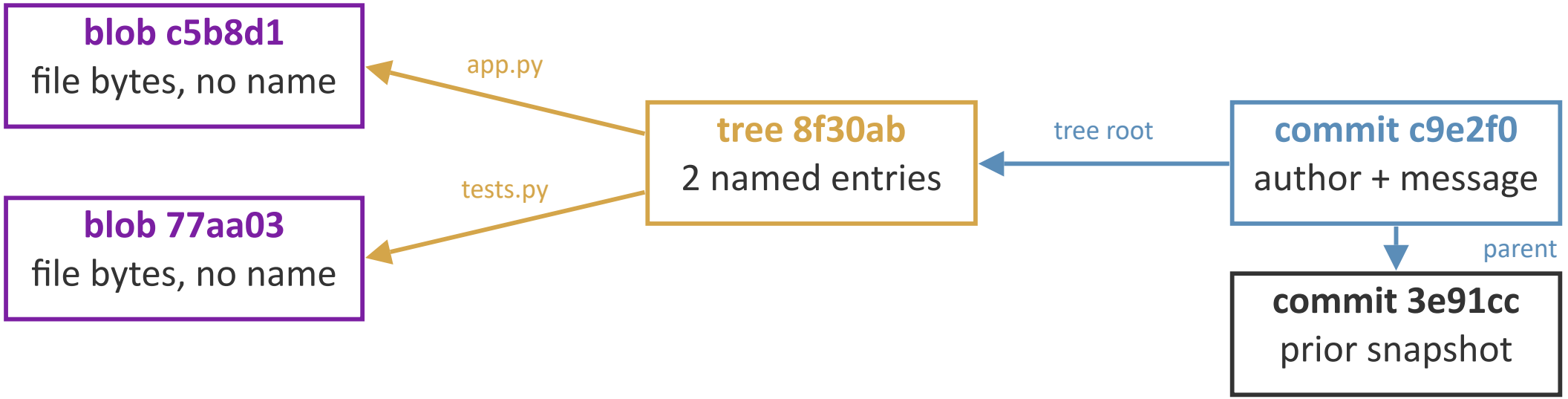
What git stores

Git is a content-addressed object database with names pointing into it

Blob
File contents. Git stores bytes, not the filename.

Tree
Directory listing. Names point to blobs and other trees.

Commit
Snapshot plus parent pointer(s), author, message, and tree root.



A commit is a snapshot with ancestry, not a diff and not a folder.

The commit graph

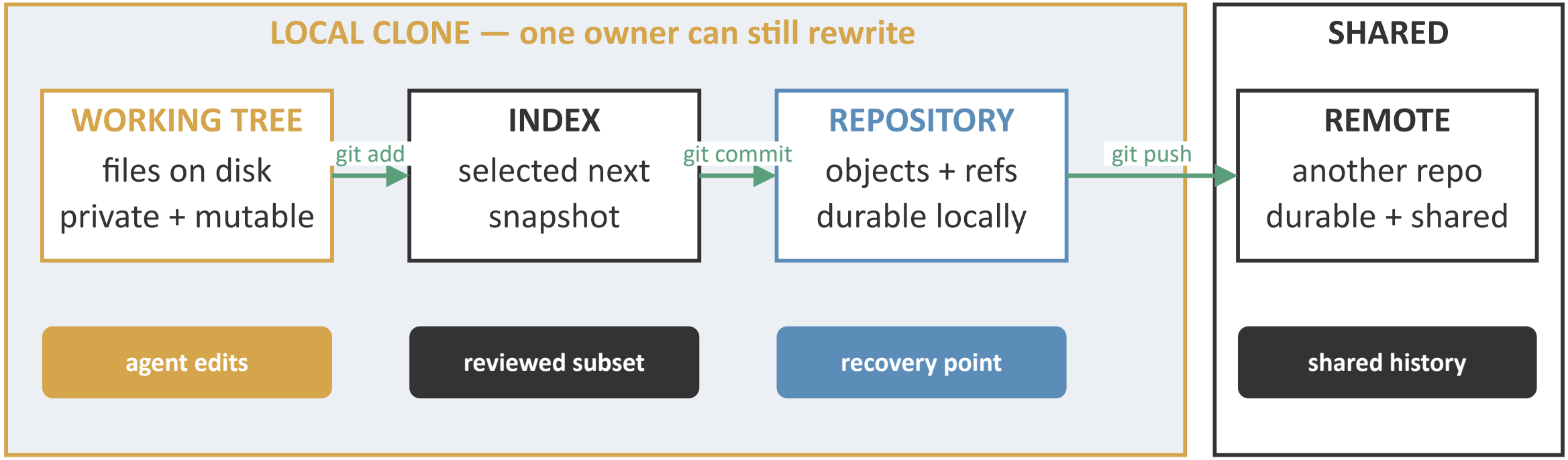
History is a graph of immutable snapshots

Name	Points to	Changes when
commit	one tree + parent commits	never; its hash names its content
branch	one current commit	you commit, merge, rebase, or reset
HEAD	the branch or commit you are looking at	you switch branches or detach
tag	one chosen commit	normally never

- Branches are movable names. Commits are immutable objects.
- Most git confusion starts by treating a branch like a separate copy of files.

Four places your work can live

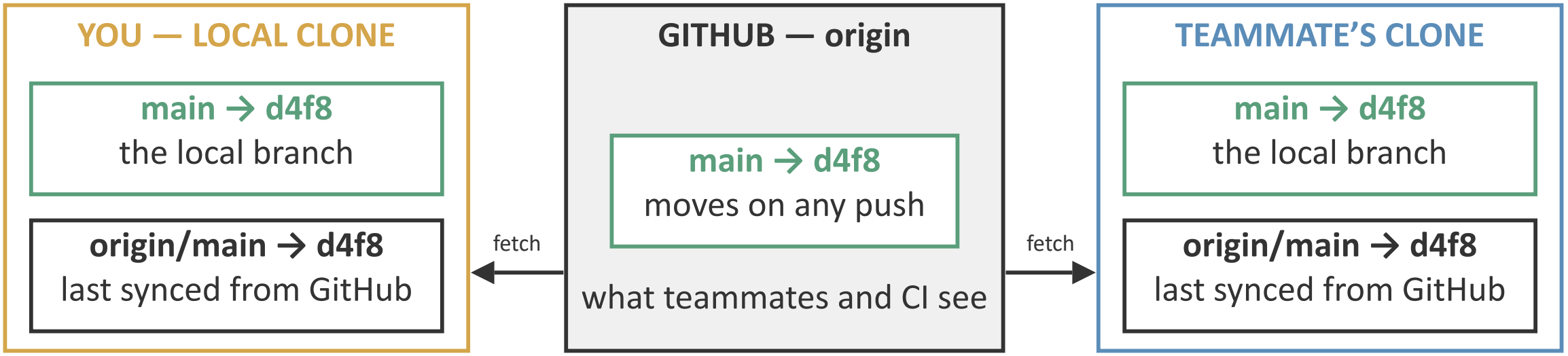
Most agent mistakes are easier to diagnose with this model



Commit preserves a reviewed snapshot; push makes that history shared.

Local branches, remote branches, and origin/*

One example: your clone, GitHub, and a teammate's clone



Why the same hash? You both cloned from GitHub and no one has committed since — the same commit keeps the same hash in every clone.

Commits travel only through GitHub: you push, your teammate fetches. No git command moves work directly between two clones.

Why agents make git more important

Speed changes the failure mode

Large diffs, fast

An agent can touch a dozen files while you read one. You need to see exactly what changed before trusting the result.

Read git diff before you accept the change.

Parallel work

You may run a main agent, a reviewer, and a fix attempt at once. Sharing one checkout means they overwrite each other.

Give each agent its own branch and worktree.

Cheap recovery

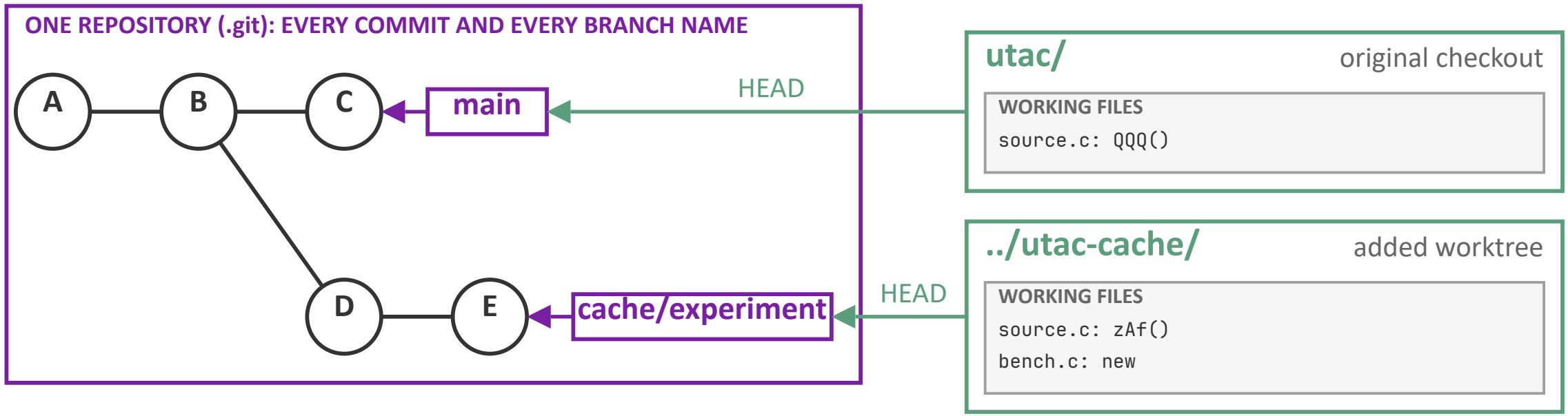
A bad edit should cost you a reset to a known commit, not a forensic project reconstructing what was there.

Commit a known-good state before each run.

Agents do not change what git is. They change how quickly you reach a state that only a commit can undo.

Worktrees

Multiple checkouts, one shared repository history



On disk: `utac/.git/` is the repository; `../utac-cache/.git` is a pointer into it. Uncommitted edits stay in their own checkout.

Every worktree shares the one object store — isolation for files, not a new universe of history.

Instapoll

Instapoll · Review · Sep 24 A

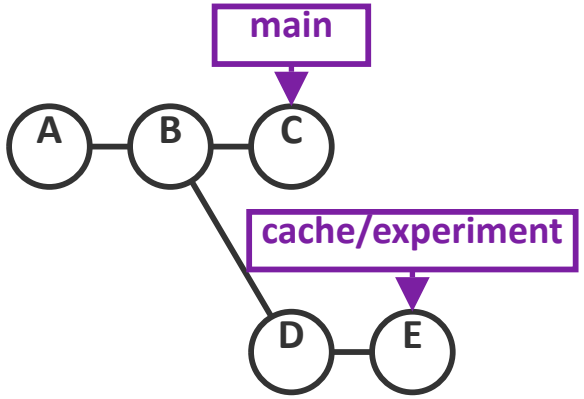
origin/main has advanced beyond your branch’s fork point. Your agent rebases all three of your commits onto that new tip. Afterwards git log shows the same three messages in the same order. Are they the same commits?

- A. Yes — a rebase moves the branch name, and the commits themselves are untouched
- B. No — each is a new commit with a new id, and the originals are still on disk
- C. No — the originals were deleted, which is why rebasing a shared branch is unsafe
- D. Yes for the ones that applied cleanly; only a conflicting commit is recreated

The worktree lifecycle

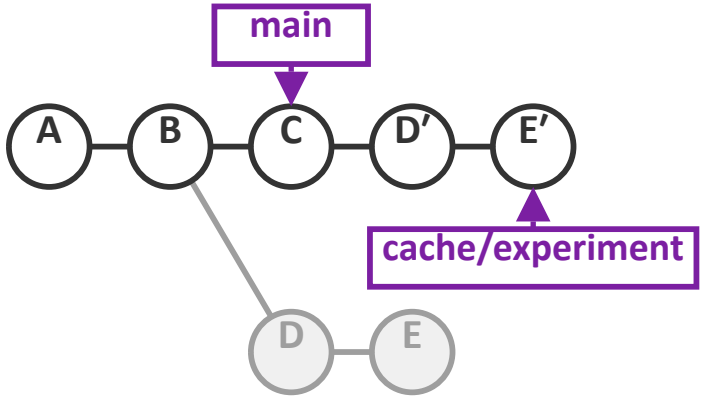
Rebase creates the new commits; fast-forward moves main to their tip

BEFORE



main names C; cache/experiment names E

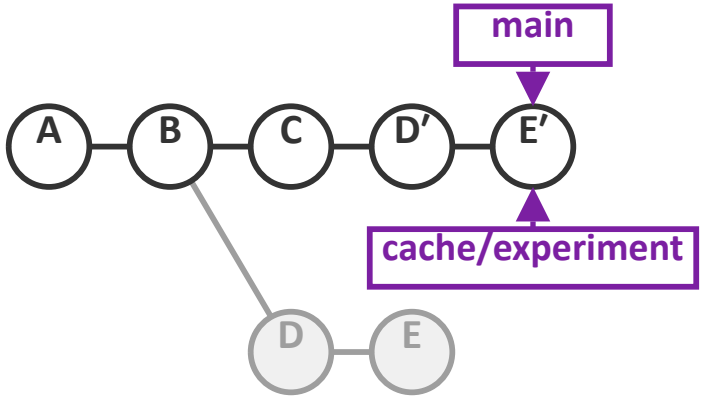
AFTER REBASE



old D and E remain in the object store

D′ and E′ are new; main stays at C

AFTER FAST-FORWARD



old D and E remain in the object store

Only main moves; no new commits

```
git rebase main                        # in the feature checkout: replay D and E
git merge --ff-only cache/experiment   # in the main checkout: move main to E′
git worktree remove ../utac-cache; git branch -d cache/experiment   # clean up
```

- Prefer rebase over merge for private work. Linear history is easier to review, bisect, and revert. Merge preserves commit IDs collaborators already use.

Concurrent development pattern

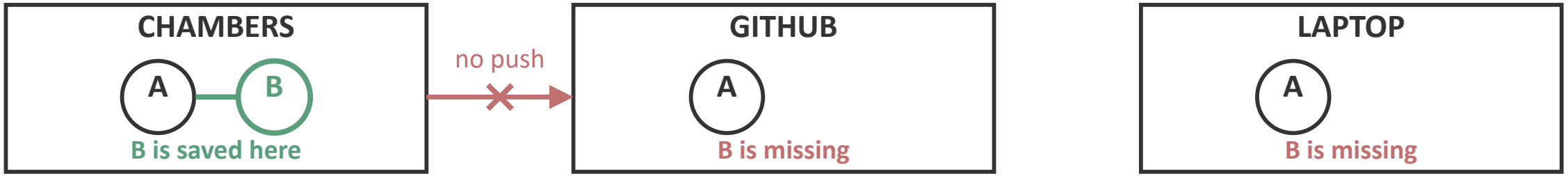
One human can run multiple agents on the same homework

Scenario	Good split	Risk if shared
Two attempts at the same homework	one branch/worktree per attempt	agents overwrite each other
Implementation + tests	separate worktrees, same base commit	test edits chase moving code
Review + fix	reviewer reads; fixer edits separate branch	review feedback mutates mid-read
Experiment	throwaway branch/worktree	failed idea dirties main checkout

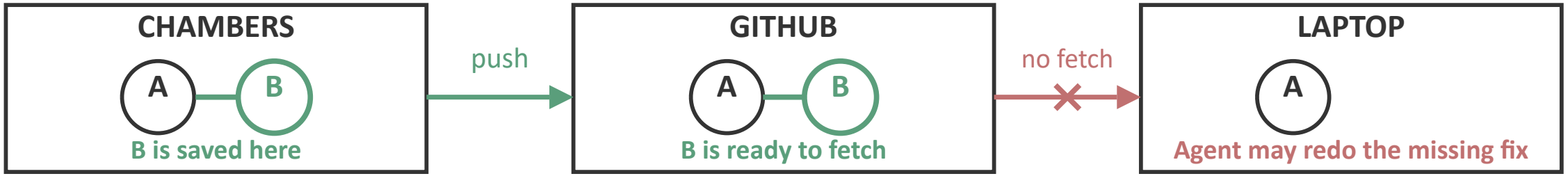
Your next clone may not have the commit

Commit B is the fix. Copies move from Chambers → GitHub → Laptop.

1. Before push



2. After push, before fetch



Before leaving a host, commit and push. On arrival, fetch, list branches, and inspect the graph before redoing work.

Read git status

Start with the branch, then separate staged changes from the remaining edits

Example output (excerpt)

```
$ git status
On branch main
Changes to be committed:
  modified: tests.py

Changes not staged for commit:
  modified: app.py

Untracked files:
  scratch.py
```

- On branch main
  - The next commit advances main.
- Staged
  - The staged version of tests.py goes into the next commit.
- Unstaged
  - The latest edits to app.py are not staged.
- Untracked
  - scratch.py is not in the index yet.

git add updates the staged snapshot. git commit records that snapshot, even if you edit the file again.

One file can be staged AND modified

git add copies the current file into the index. Later edits do not update that copy.

READ DOWN ↓

	HEAD Last commit	INDEX Staged snapshot	APP.PY Working file
1. Edit app.py	retries = 3	retries = 3	retries = 5
2. git add app.py	retries = 3	retries = 5	retries = 5
3. Edit again	retries = 3	retries = 5	retries = 8

After step 3: staged change is 3 → 5

```
Changes to be committed:
  modified: app.py
```

The same file also has an unstaged change: 5 → 8

```
Changes not staged for commit:
  modified: app.py
```

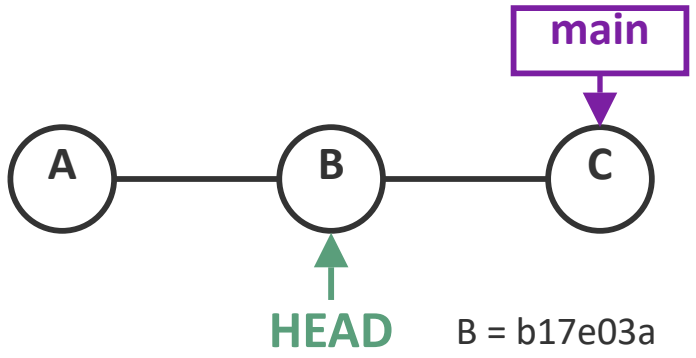
Plain git commit records retries = 5. The working file keeps retries = 8, still unstaged.

Problematic git statuses

A detached HEAD and an unresolved merge need different next steps

Detached HEAD

```
$ git status
HEAD detached at b17e03a
nothing to commit, working tree clean
```



New commits here will not advance main.

Unresolved merge (excerpt)

```
$ git status
On branch main
You have unmerged paths.
  both modified: app.py
```

Inside app.py

```
<<<<<<< HEAD
retries = 3
=======
retries = 5
>>>>>>> fix
```

Detached HEAD: git switch -c rescue keeps new work on a branch. For a conflict, resolve and git add the file, then continue the merge or abort it.

Key Takeaways

1. A commit is an immutable snapshot with ancestry, and a branch is just a movable name pointing at one commit. That single distinction explains branches, rebase, merge, and worktrees, so learn the model rather than memorizing commands.

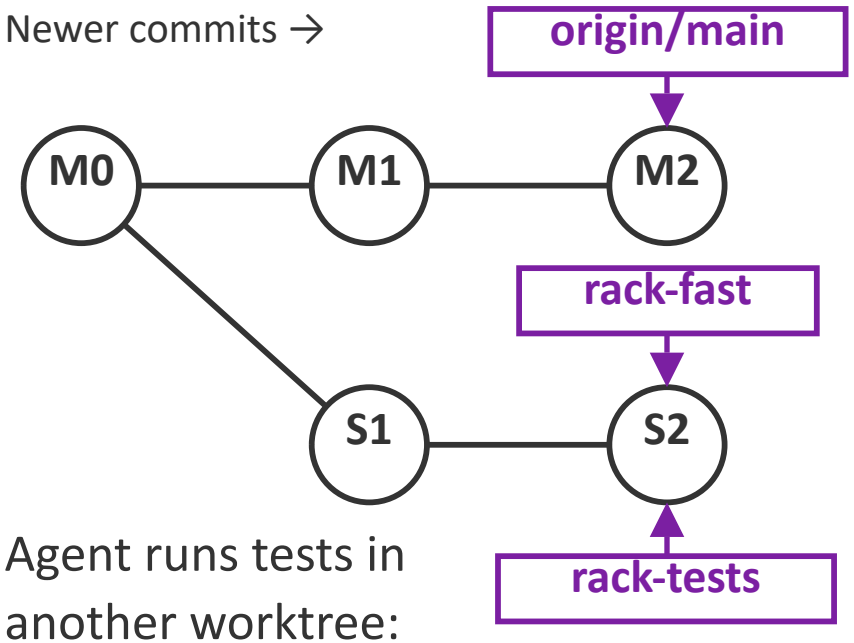
2. Rebase then fast-forward keeps history a straight line, which makes review, bisect, and revert simple. Reserve merge commits for branches other people have already pulled, because rebasing those rewrites commits they hold.

3. Run concurrent agents in separate worktrees: they share one object store, so isolation costs files, not history. Commit and push before leaving a host, because another clone cannot fetch your commits from GitHub until you push them.

Rebase or merge? Draw the graph

10 min

Your clone, after fetching main



Agent runs tests in another worktree:

Classmate's copy of your branch

You pushed; they fetched last night.

- Draw two outcomes, in pairs.
  - Start on rack-fast from the left graph each time:
  - git rebase origin/main
  - git merge origin/main
  - Mark new commits and their parents.
- Explain what the rebase changes.
  - Which commits does your classmate keep?
  - Why is a plain git push refused?
  - Where does rack-tests point? Is S2 still on disk? Why?
- Choose rebase or merge for the next version.
  - Name the cost: replaced IDs or a merge commit.
  - Compare with another pair. Is a fetched branch still private? Defend your rule.