Git data model

Key takeaways

1. Git stores immutable snapshots and movable refs; learn that model first.
2. Prefer rebase and fast-forward: a linear history has one order of events.
3. Give concurrent agents separate worktrees; push remote work before leaving.

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.

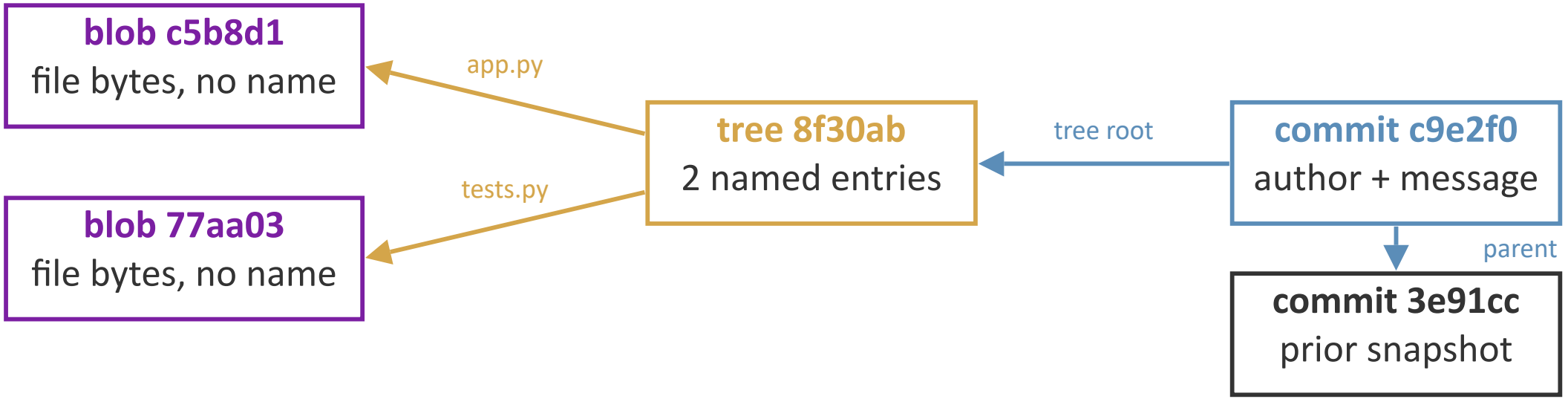
What git stores

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

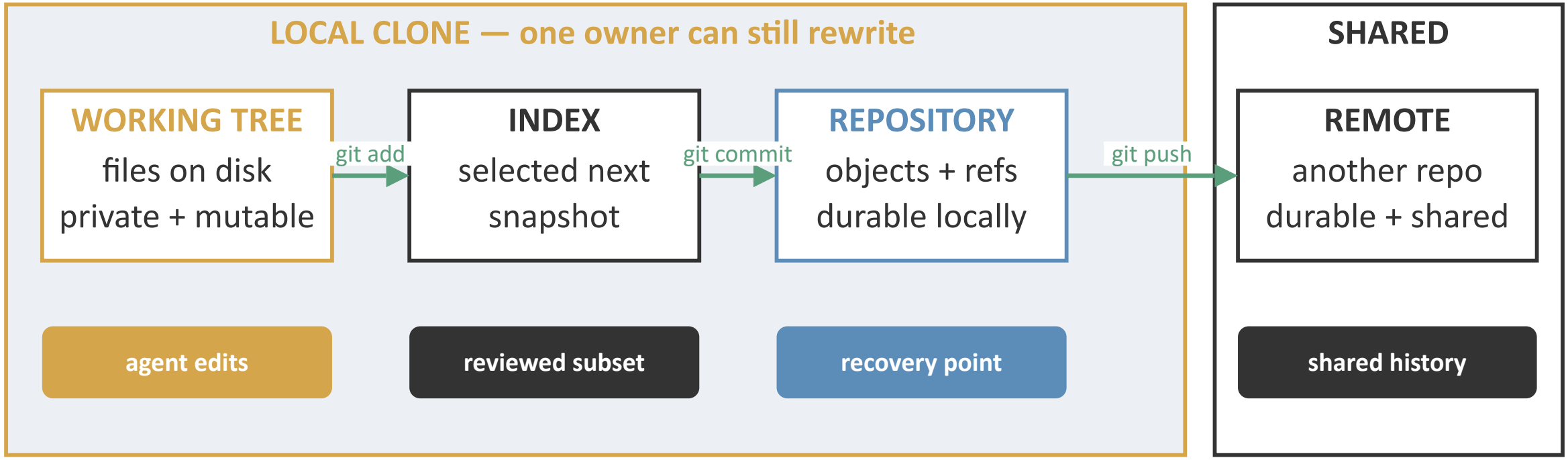
History is a graph of immutable snapshots

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

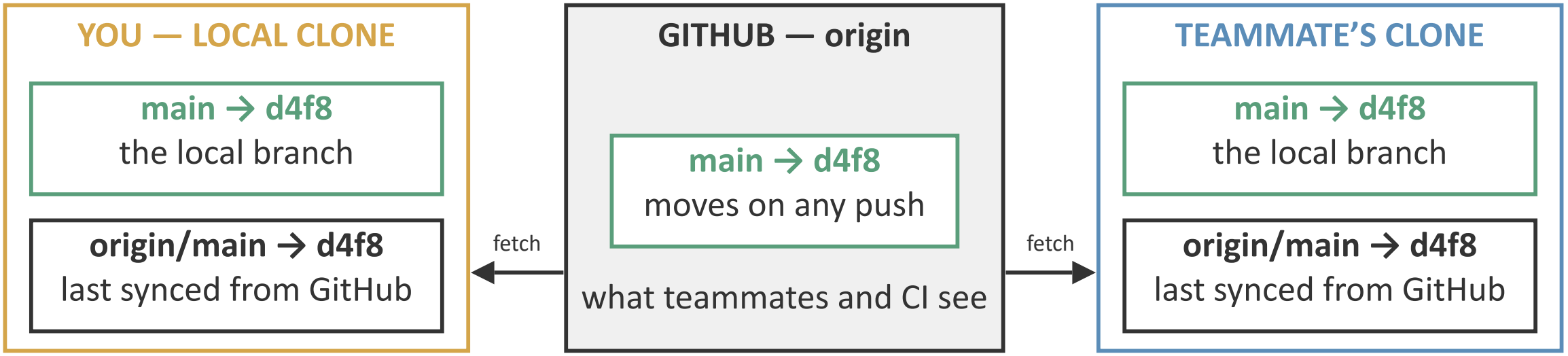
Most agent mistakes are easier to diagnose with this model



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

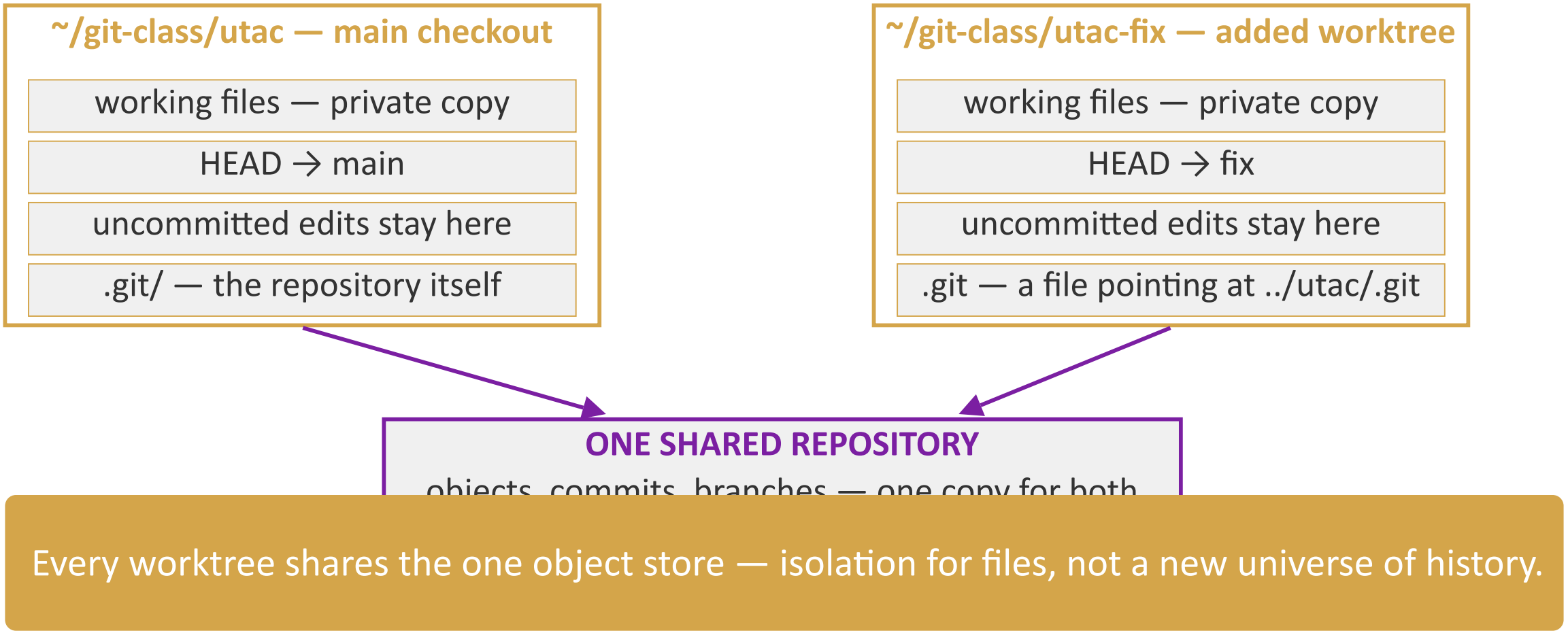
A bad edit should cost you a reset to a known commit, not a forensic project reconstructing what was there.

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

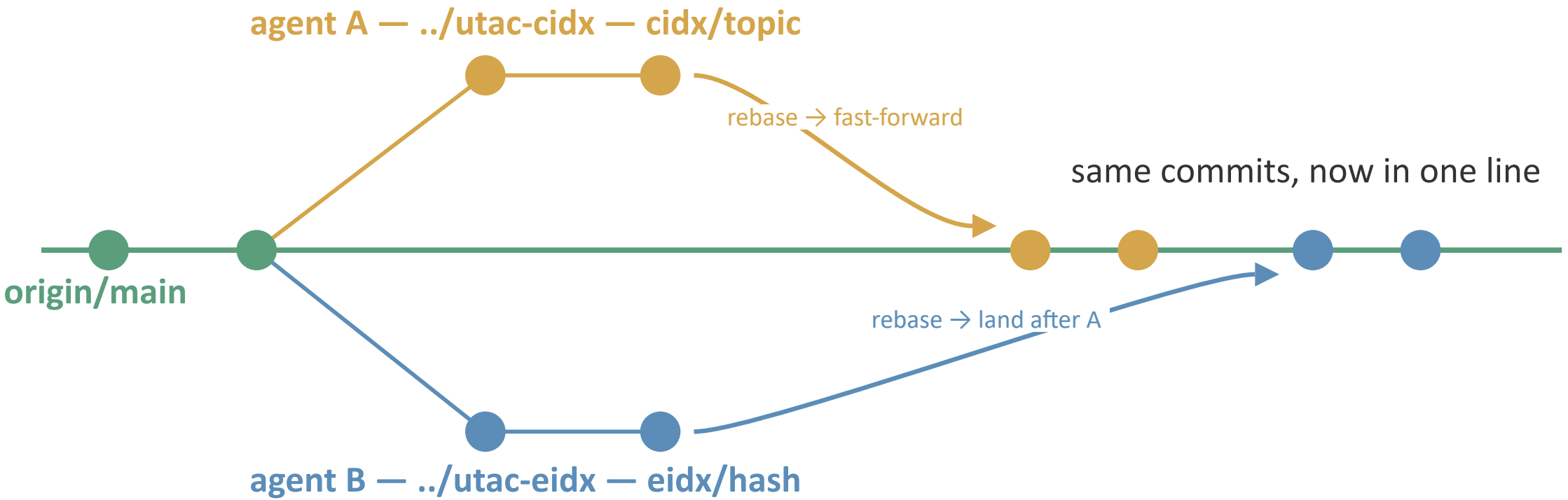
Multiple checkouts, one shared repository history



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

The worktree lifecycle

Branch out in parallel; land back in series



Branch from origin/main, work in a private worktree, rebase, fast-forward, delete the worktree.

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

Instapoll

Instapoll · Review · Oct 13 A

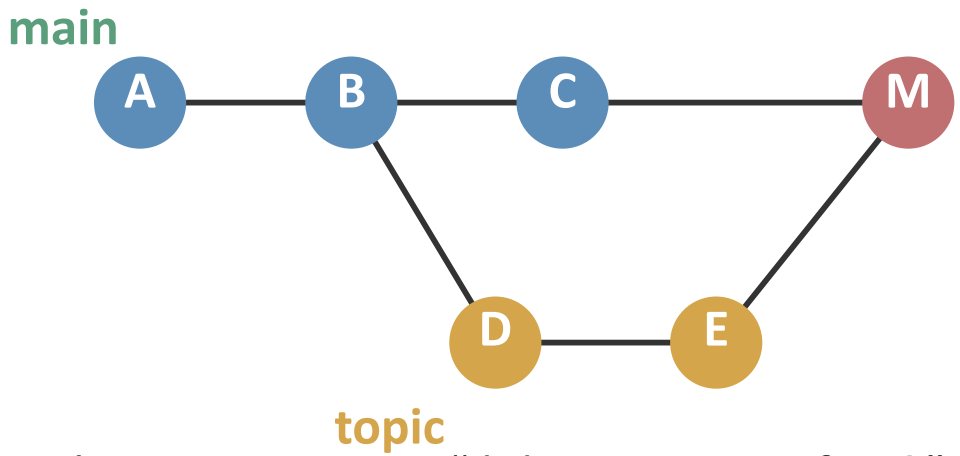
You create a second worktree for a parallel agent on a repository whose history contains a 1 GB file. What happens to disk usage?

- A. The whole object database is duplicated, so disk usage roughly doubles
- B. Only the checked-out files are added; the object store stays shared
- C. Git refuses the worktree until the large blob is removed from history
- D. The large blob is copied but ordinary source files are hard-linked

The shape of history: merge vs. rebase

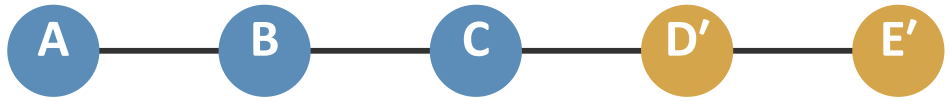
Merge preserves the fork; rebase replays it into a line

MERGE: THE FORK IS PRESERVED



M has two parents: “did C or E come first?” has no answer without walking a graph.

REBASE: THE LINE IS RESTORED



The same work replayed on top of C: one order, one parent each, new hashes D’ and E’.

Prefer rebase then fast-forward whenever the branch is yours alone: a linear history keeps review, bisect, and revert simple.

Rebase vs. merge

Both move branch names; only one rewrites your commit ids

Operation	Storage-model view	Course habit
fast-forward	move a branch ref to a descendant commit	preferred final integration when history is already linear
merge	create one new commit with two parents	use when preserving a shared branch history matters
rebase	copy your commits onto a new parent, producing new hashes	use before review so your branch reads as a clean story
reset	move a branch ref without copying or merging	dangerous around uncommitted work; know exactly what moves

If other people may have pulled your commits, do not rebase them without coordination.

Even with git, work can get lost

Git helps only after the work is in the right place

Failure	Why git cannot save it yet	Course habit
Checkout/reset overwrites in-progress edits	uncommitted bytes are just files on disk	commit or stash before switching tasks
Work on another machine is abandoned	the fix exists only in that machine's clone, maybe uncommitted	record hostname, branch, and git status
The wrong clone gets edited	laptop, WSL, and a department host each hold their own copy	pwd, git remote -v, and branch before editing
A commit was never pushed	it exists in one clone; no fetch from GitHub can find it	push before leaving a machine for the day
An agent repeats work already done elsewhere	the other branch was never fetched into origin/*	fetch, list branches, and inspect the graph

A commit is recoverable. A memory that you edited something on another host is not.

Read the state: working tree and index

These states matter more than exact command spelling

- Dirty working tree
  - Files differ from the last commit; review before switching tasks.
- Staged vs. unstaged
  - Only staged files become the next commit; staging is a selection step.

Column 1 is the index, column 2 is the working tree

```
$ git status -sb
## main...origin/main
 M app.py       # edited, NOT staged
M  tests.py     # staged, will commit
MM notes.md     # staged, then edited again
?? scratch.py   # untracked
```

Two states no agent should push through

Ahead/behind you already read off git branch -vv; these two need a human

- Detached HEAD — you are on a commit, not a branch; new commits can be lost.
- Merge conflict — two snapshots changed one region; a human chooses the result.

Detached HEAD: commits with no name

```
$ git switch --detach 3e91cc
$ git status
HEAD detached at 3e91cc

# commits made here belong to no branch;
# switching away leaves them reachable
# only through git reflog
```

Merge conflict: one region, two answers

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

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

Instapoll

Instapoll · Review · Oct 13 B

An agent rebases a private branch and the old branch tip disappears. Where would you look first for the previous commit identifier?

- A. The local reflog for the rewritten branch or HEAD
- B. The remote repository’s default branch
- C. The working tree’s file modification times
- D. The staging area’s current index entries

Human vs. agent ownership

Do not outsource the repository boundary

Your Responsibility
Choose the branch/worktree boundary
Review staged changes before commit
Decide conflict resolution intent
Own what reaches GitHub

Agent’s Responsibility
Summarize diffs and risky files
Prepare narrow commits when asked
Explain conflict choices
Run tests before handoff

Key Takeaways

1. Git stores immutable snapshots and movable refs; that model explains branches, rebase, merge, and worktrees.
2. Rebase then fast-forward keeps history a line; reserve merge commits for genuinely shared branches.
3. Use worktrees for concurrent agents, and commit/push remote-host work before it is forgotten.