Parallel work: worktrees, tmux sessions & landing

Key takeaways

1. Parallel progress starts with a dependency graph, not more terminals. Split the goal into tasks that each name an output and a merge point.

2. A worktree gives a task its own files and exists to bring them back: rebase, fast-forward main, remove the worktree. tmux keeps a job alive when your connection drops.

3. Switch only while work runs unattended, and return on a named event such as a finished run. Past two to five live sessions, coordination costs more than it buys.

Common misunderstandings of parallel development work

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

"Working in parallel means opening more terminals and keeping every one of them busy."

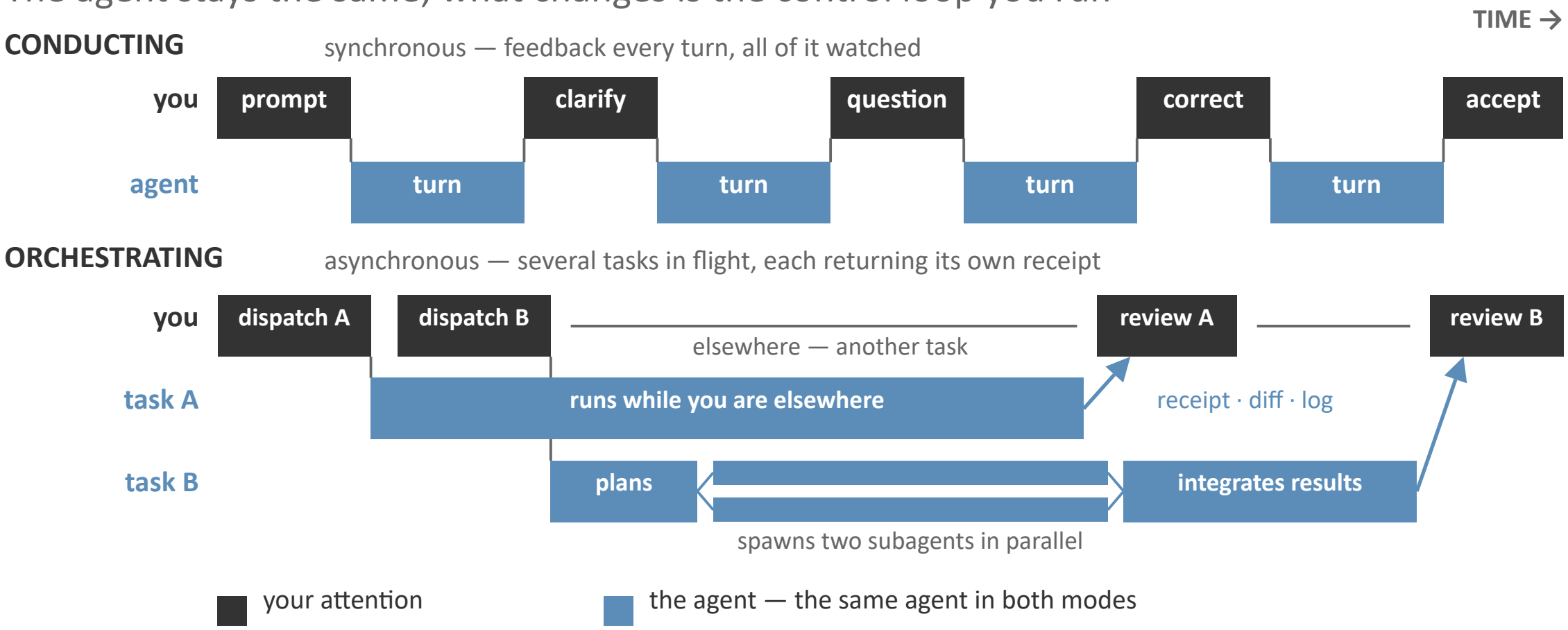
"A git branch gives each concurrent task its own working files."

"A merge commit and a rebase leave main with the same history."

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

Conducting and orchestrating

The agent stays the same; what changes is the control loop you run

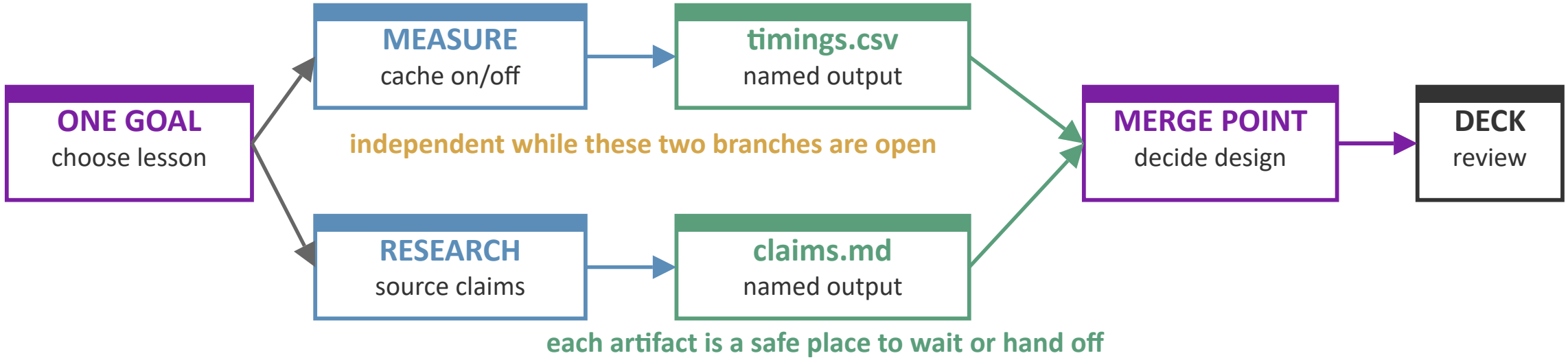


Conduct when each step needs your eyes; orchestrate when a receipt can stand in for them.

source: verb framing adapted from the conductor/orchestrator roles in Osmani, Saboo & Kartakis, The New SDLC With Vibe Coding (Google, May 2026)

Parallel work means resumable state

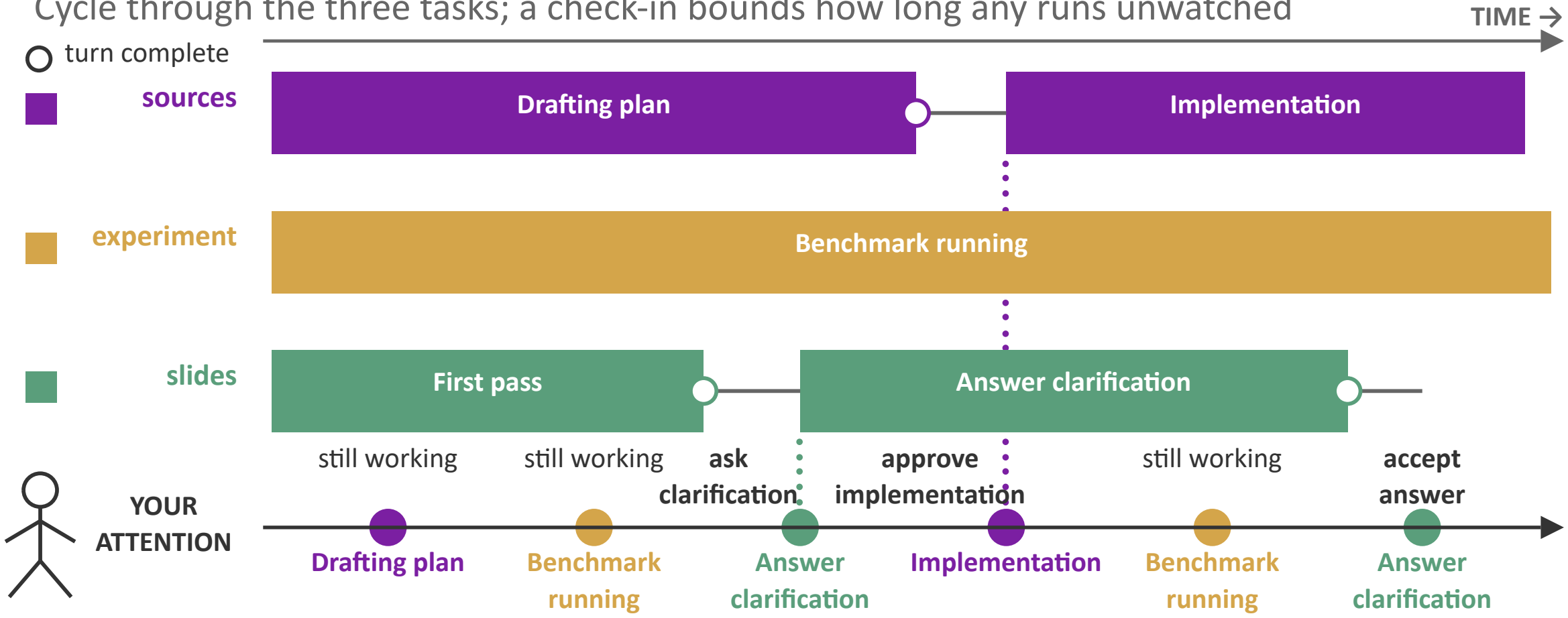
- Parallel progress is not typing in two terminals at once.
- It is dividing one goal into tasks that can wait without losing their state.
- tmux preserves the conversation with the machine; a worktree isolates the files.



A task is ready to run in parallel when you can name its output and its merge point.

Switch when work can continue without you

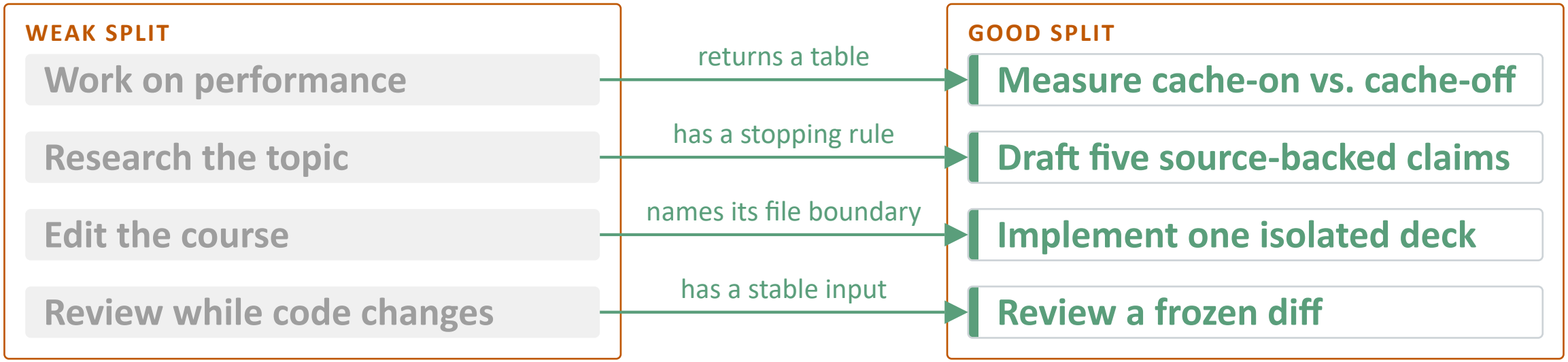
Cycle through the three tasks; a check-in bounds how long any runs unwatched



Act on a completed turn or move on; let /tasks or a notify hook call the next visit.

Split so no task touches a file being written

Two writers overwrite each other; a reader gets a moving input, so it waits for the write to land

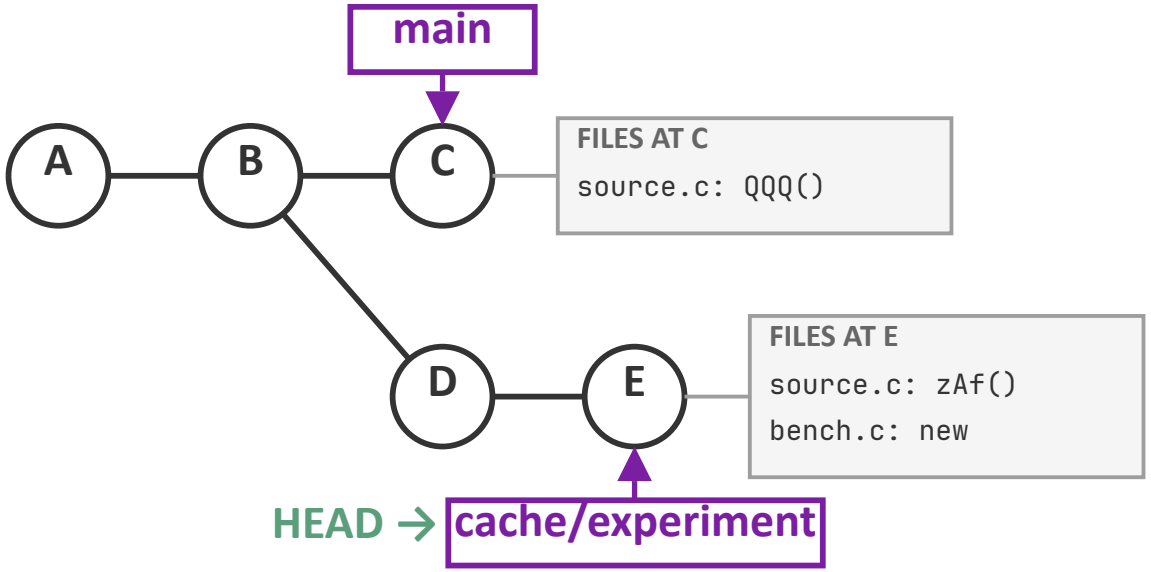


Sixteen parallel agents building a C compiler each hit the same bug, each fixed it, and each overwrote the others. The fix was splitting the work per source file, not adding agents (Anthropic, 2026-02-05).

Two tasks that write the same file, or the same decision, are one task pretending to be two.

A branch is a movable name for a commit

Cheap to create and switch between; but one checkout shows one branch at a time



A commit is a snapshot of every file, so C and E differ in which files exist and what they hold. A branch names its newest commit, and each new commit moves the name. HEAD records which branch this checkout shows.

```
git switch -c cache/experiment
    # a new name at HEAD, checked out
git commit       # moves only that name
git switch main  # rewrites the files
```

ONE CHECKOUT SHOWS ONE BRANCH AT A TIME

One HEAD per checkout
the files match one branch's tip

Switching rewrites those files
under any build, editor, or agent

Uncommitted edits get in the way
the next task waits for a commit or stash

Two branches in one checkout take turns: the files belong to whichever is checked out.

Instapoll

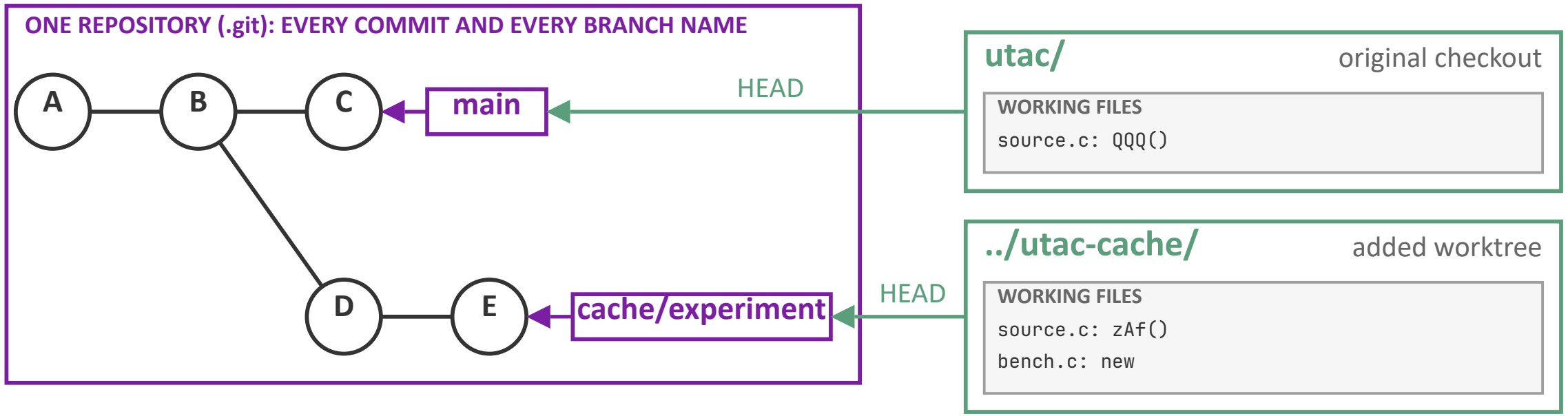
Instapoll · Review · Sep 15 B

A git worktree exports a second view of one repository into a new directory, on its own branch and sharing the same commits: git worktree add ../utac-cache -b cache/experiment makes one. What does that buy two agents working on the same repository at once?

- A. Nothing a branch does not already: each agent checks out its own branch in the one directory
- B. Each agent edits its own files on its own branch, so neither sees the other’s unfinished work
- C. An agent’s session survives a dropped connection, since its work sits in its own directory
- D. Both agents see each other’s edits as they happen, so they can coordinate on shared files

A worktree gives each task its own files

One repository, two checkouts: each has its own HEAD and its own working files



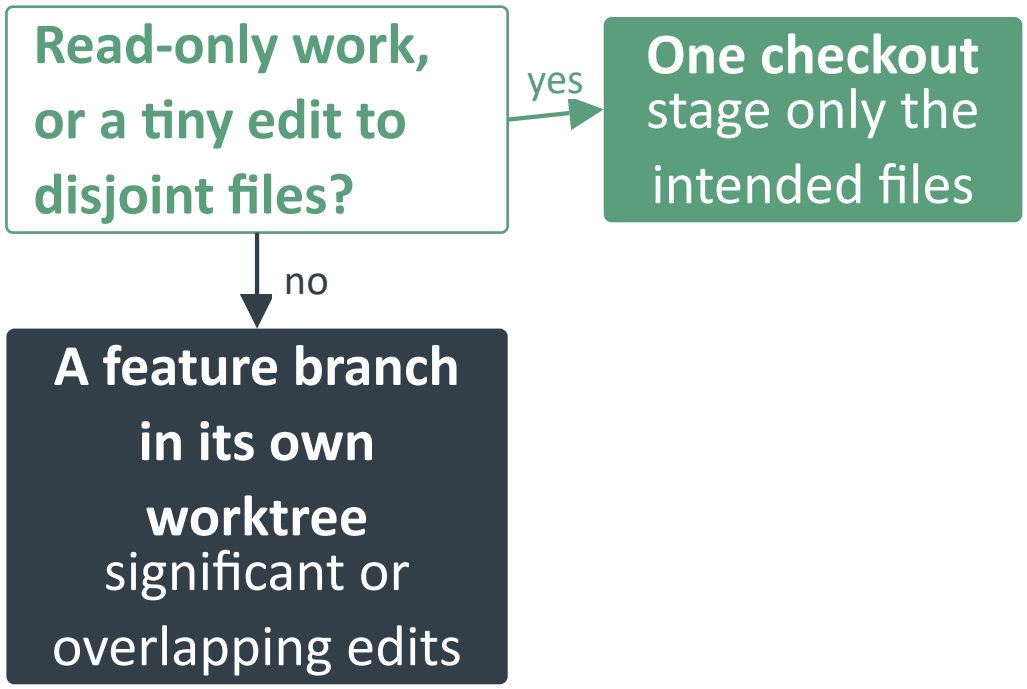

```
git worktree add ../utac-cache -b cache/experiment   # new checkout, new branch
git worktree list                                    # every checkout and its branch
git worktree remove ../utac-cache                    # once the work is on main
```

Significant or overlapping agent work needs its own worktree.

A worktree’s work comes back to main

Choose the smallest boundary that isolates the work, and plan its return before you start

CHOOSE THE BOUNDARY



THEN BRING IT BACK

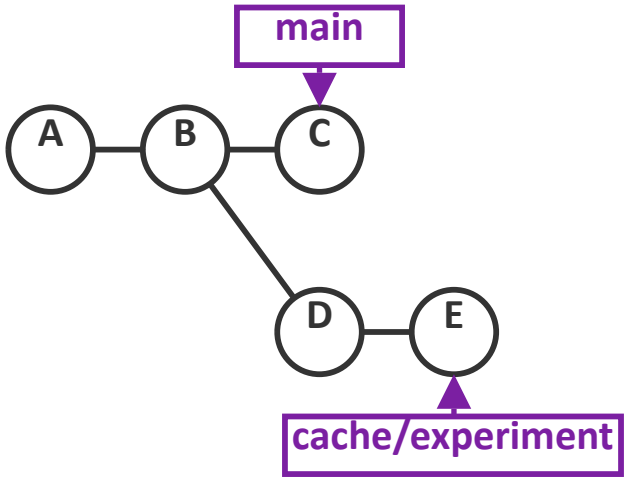
1. **Rebase onto main**
   put your commits on top of main’s tip
2. **Fast-forward main**
   main moves to your tip; no merge commit
3. **Remove the worktree, delete the branch**
   main has the work; nothing stays behind

The worktree and its branch are scaffolding; the deliverable is commits on main.

Merge keeps the fork; rebase keeps one line

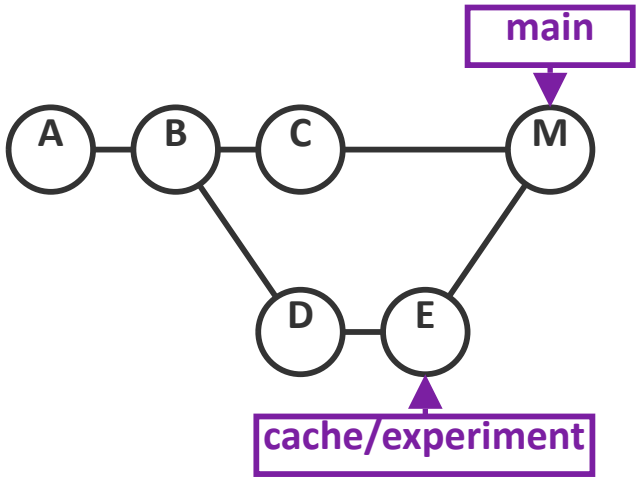
Both bring D and E onto main; only one leaves a history that reads straight through

BEFORE: TWO TIPS



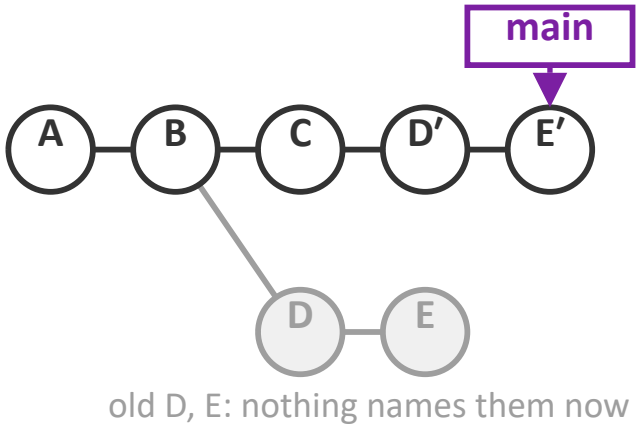
The worktree's commits D and E are not on main yet.

MERGE: A COMMIT WITH TWO PARENTS



Merge adds M, whose parents are C and E; the fork stays in the history.

REBASE, THEN FAST-FORWARD: ONE LINE



Rebase replays D and E onto C as D′ and E′, new ids; main then moves to E′.

```
git rebase main                          # in the worktree: D, E replayed on C
git merge --ff-only cache/experiment  # in utac/: main moves to E′
git worktree remove ../utac-cache; git branch -d cache/experiment
```

Rebase a branch that is yours alone, then fast-forward: history reads as one line.

tmux separates job lifetime from connection lifetime

The server and job live on the remote host; the client may detach

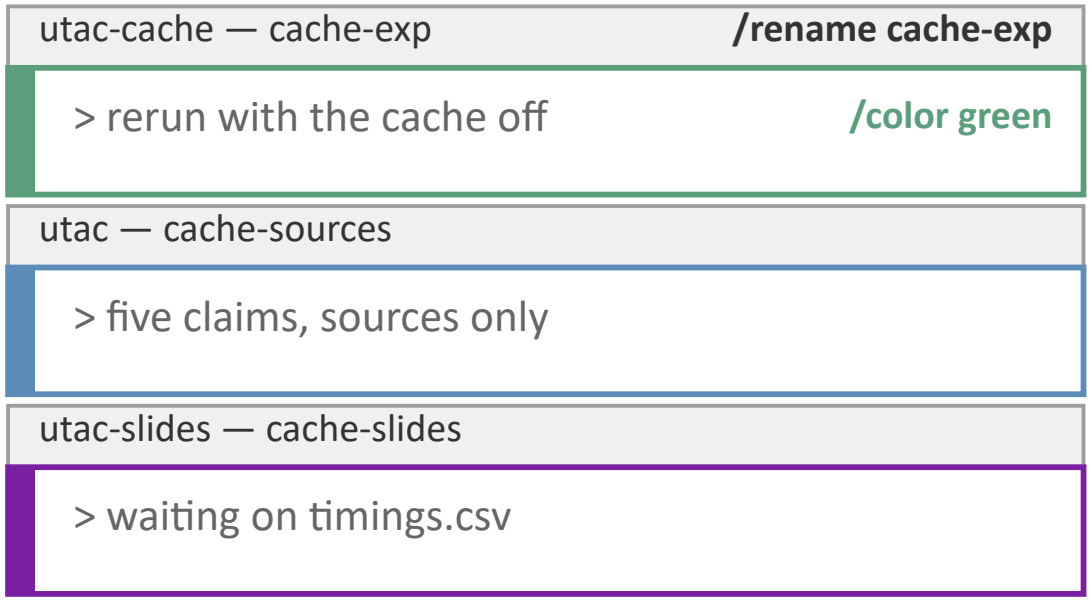
```
# First connection: record the owner, then start the long job
ssh HOST
hostname -f
tmux new-session -s cache-exp -c ../utac-cache
./run-benchmark 2>&1 | tee evidence/cache-exp.log
# C-b d detaches; the tmux server and job keep running

# Later connection: return to the SAME host
ssh HOST
tmux list-sessions
tmux attach-session -t cache-exp
```

A different host may see the files, but it does not own this tmux server.

Color is the fastest way to tell sessions apart

A glance says which task this is; reading says what it waits on



Eight colors: red, blue, green, yellow, purple, orange, pink, cyan; /color default resets. Codex has /rename but no color: use tmux window names or terminal tab colors there.

WHAT A GLANCE TELLS YOU

Which task this is
the color; no reading required

What it is waiting on
the last prompt line

Where its files live
the worktree named in the tab

Assign each task a color once; then you can switch sessions without reading.

More sessions can mean less progress

Brooks’s law holds for agents: both vendors put the ceiling at two to five sessions

Every live session adds polling, reorientation, and integration demand; ten parallel sessions also spend quota ten times faster.

many unnamed shells	no task boundary	rename or close them
same files in two sessions	false independence	serialize the work
constant conflict resolution	split crossed an interface	move the boundary
forgotten finished jobs	no polling cadence	keep an active-session list
long reorientation	weak handoff note	leave the next question

source: claude.com, The AI-native SDLC playbook (2026-08-21); openai.com, Symphony (2026-04-27) and Shipping Sora for Android with Codex (2025-11); code.claude.com agent-view page for the quota figure

Key Takeaways

1. Parallel progress starts with a dependency graph, not more terminals. Split the goal into tasks that each name an output and a merge point, and split on artifact boundaries so no task touches a file another is writing.

2. A worktree gives a task its own files and exists to bring them back: rebase a branch that is yours alone, fast-forward main, remove the worktree. tmux keeps a job alive when your connection drops.

3. Switch only while work runs unattended, and return on a named event such as a finished run. Past two to five live sessions, coordination costs more than it buys: each one adds files to keep apart, a job to poll, and a context to rebuild when you return.