



Version Control with Agentic Development

Week 4b • CIS 3990 / 2500

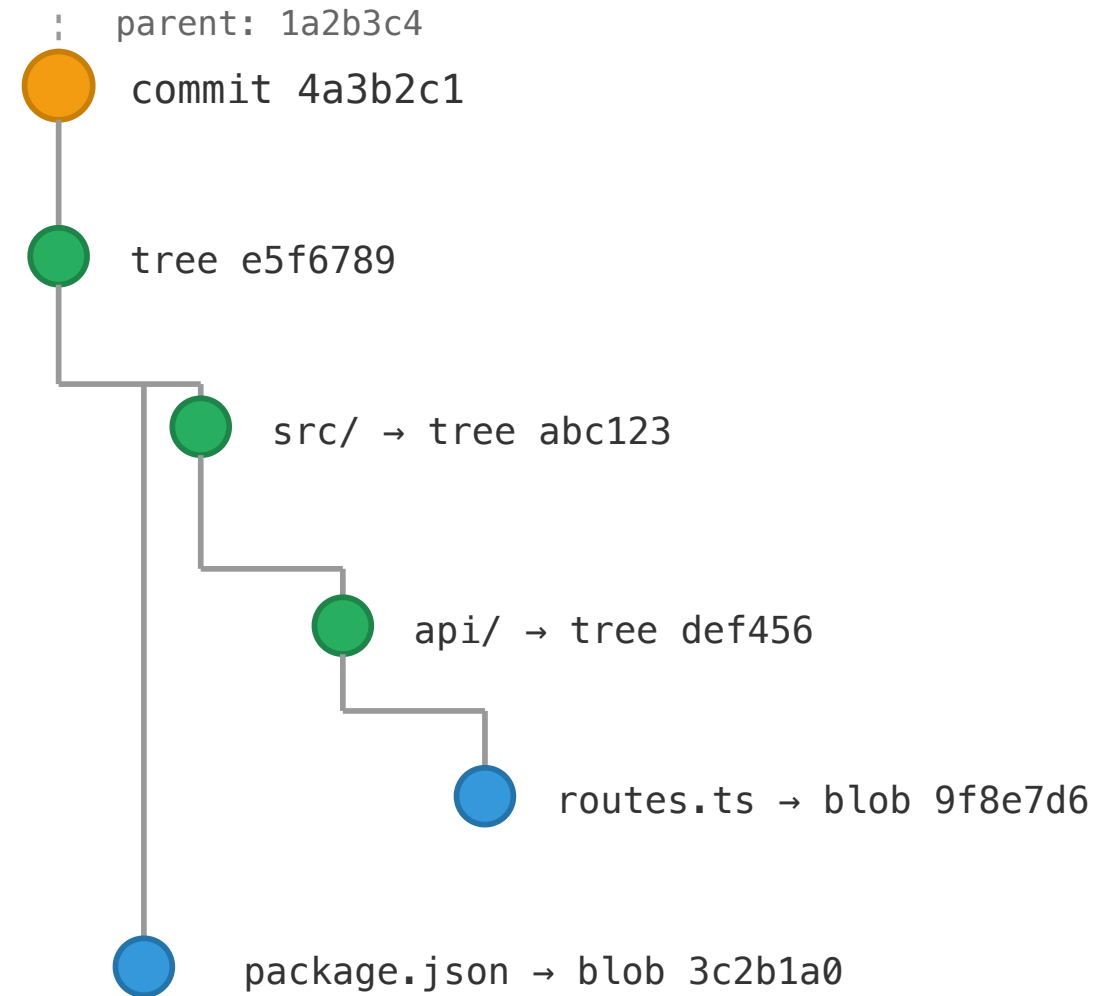
Zachary G. Ives & Harry Smith
University of Pennsylvania
Fall 2026

How can we use Git/Version Control at an expert level to help us in development?

- On the human level, can we develop some greater facility with Git?
- On the agentic level, what can we do to move more quickly and protect ourselves?

Git's Data Model: Blobs, Trees, Commits

- **Blob** — a file's content (immutable, content-addressed)
- **Tree** — a directory snapshot (maps filenames → blobs or other trees)
- **Commit** — a tree snapshot + metadata (parent commit, author, message, timestamp)



Building a Commit: The Staging Area

Git distinguishes between **tracked** and **untracked** files, and between **staged** and **unstaged** changes.

Tracked files are files Git knows about (previously committed or explicitly added).

Untracked files are new files Git hasn't been told to track yet.

Staged changes are modifications you've marked with `git add` to include in the next commit.

Unstaged changes are modifications to tracked files that haven't been staged yet.

Building a Commit: Workflow

You pick the changes that go into a commit!

```
git status          # see what's tracked, staged, and unstaged
git add <file>      # stage a file (or specific changes)
git add .           # stage all changes in current directory
git commit -m "message" # commit only the staged changes
```

You can work on multiple changes simultaneously but commit them as separate, focused units. This makes review easier and history clearer.

`git diff` for Looking at Changes

- `git diff` shows you what you have **not** accepted yet.
- `git diff --staged` shows you what you **have**.



As you review the changes that are implemented, you'll accept them piece by piece. It's important to be able to check your progress!

Reviewing by the Hunk with **-p**

```
git add -p          # walk the change one hunk at a time
diff --git a/slides/week-04a.md b/slides/week-04a.md
index c659ce8..71f49d2 100644
--- a/slides/week-04a.md
+++ b/slides/week-04a.md
@@ -35,7 +35,7 @@ Fall 2026
</div>
```

```
The answer is similar whether we are talking about agents or humans!
-- **Architectural patterns** help reduce complexity globally
+- Good **architectural design** and communication help reduce complexity globally
---
(1/2) Stage this hunk [y,n,q,a,d,j,J,g/,e,?]
```

Git shows you a hunk and waits. Press **y** to stage it, **n** to leave it alone, **s** to split it into something smaller, and **q** to stop.

`git add .`: the Secret Leaker

Agents produce files nobody else needs to see. `git add .` stages every one of them by default.

- a lockfile,
- a build directory,
- a config with a real password in it.

Remember the bumpers: use `.gitignore` to let the system do the work.

```
# .gitignore
node_modules/
dist/
.env
config.json
```

What Makes a Good Commit? Narrow Focus.

Each commit should represent a single coherent change:

- **one** bug fix
- **one** feature addition
- **one** refactoring step

(We are bad at this.)

Commits on Jul 6, 2026

added syntax highlighting to code blocks in slides; first outline of week-03 slides.



sharry29 committed 2 weeks ago · ✓ 2 / 2

Commits on Jul 4, 2026

Mermaid setup; lab sketches; partial progress on Week-02

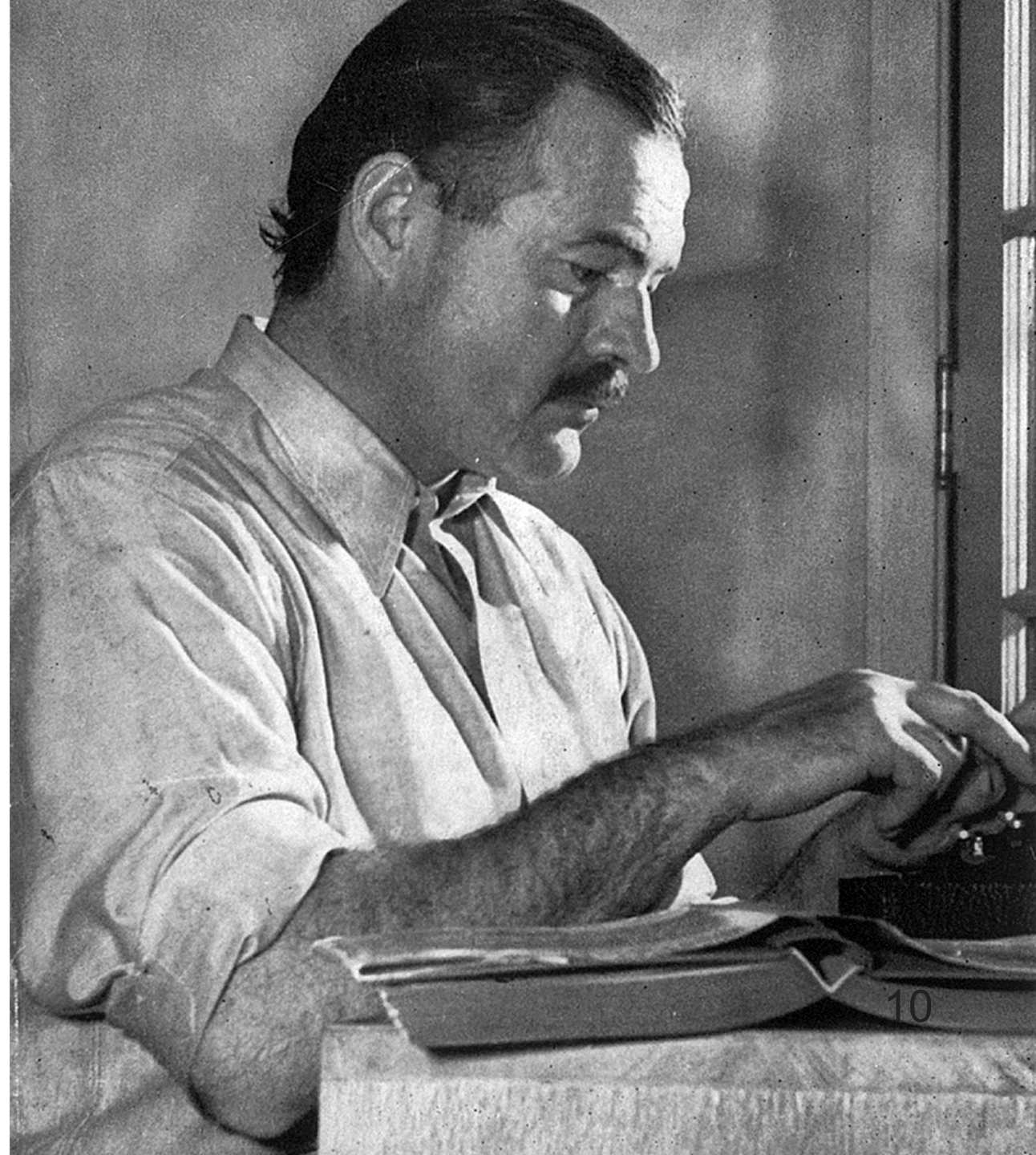


zackives committed 3 weeks ago · ✓ 2 / 2

What Makes a Good Commit? Good Messaging.

The commit message should explain **why** the change was made

- **what** changed is in the diff anyway!
- the message should provide some context and intent.
- the character limit is fairly small.



What Makes a Good Commit? No Breaking.

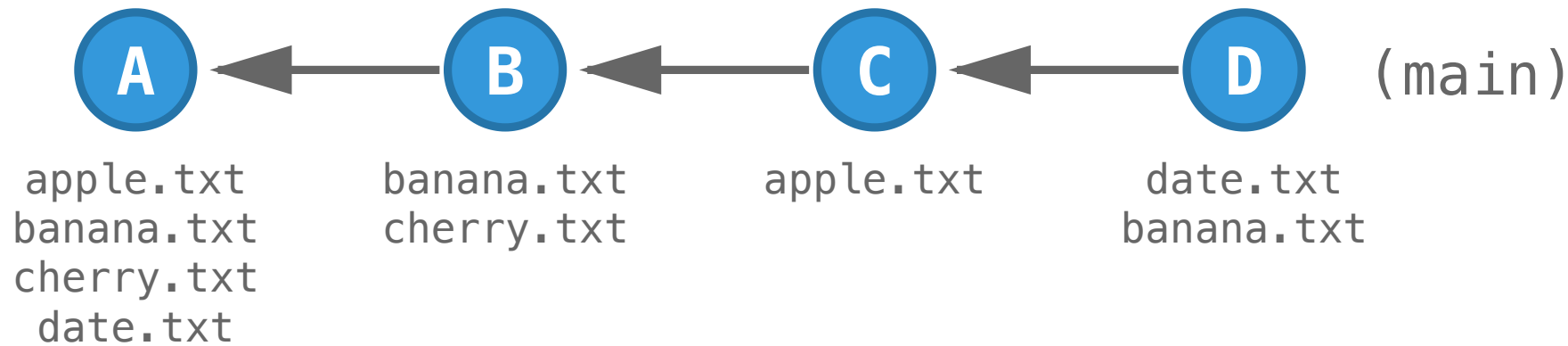
A good commit passes tests and doesn't break the build.

- A good **CI** setup should enforce this for you!
- It should leave the codebase in a working state, not halfway through a multi-step change.

In summary: Small, focused commits are easier to review, easier to understand in `git log`, and easier to revert if something goes wrong.

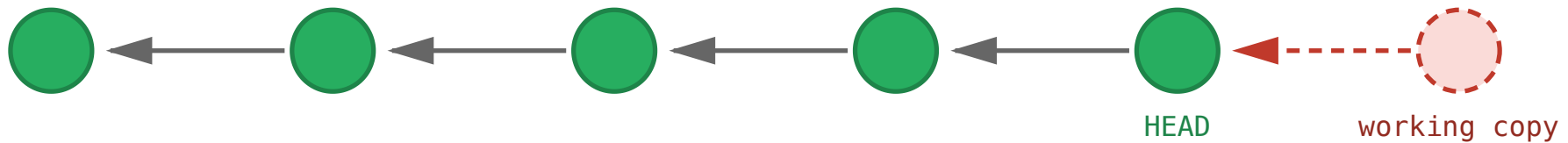
Commits On a Branch Form a Path

Each commit points to its parent. **A, B, C, D** are commits on `main`



New commits only require space to store files that have changed; unchanged files simply keep a pointer to their original blob.

A Commit Is Also a Savepoint



The red circle is the stuff that you're currently working on.

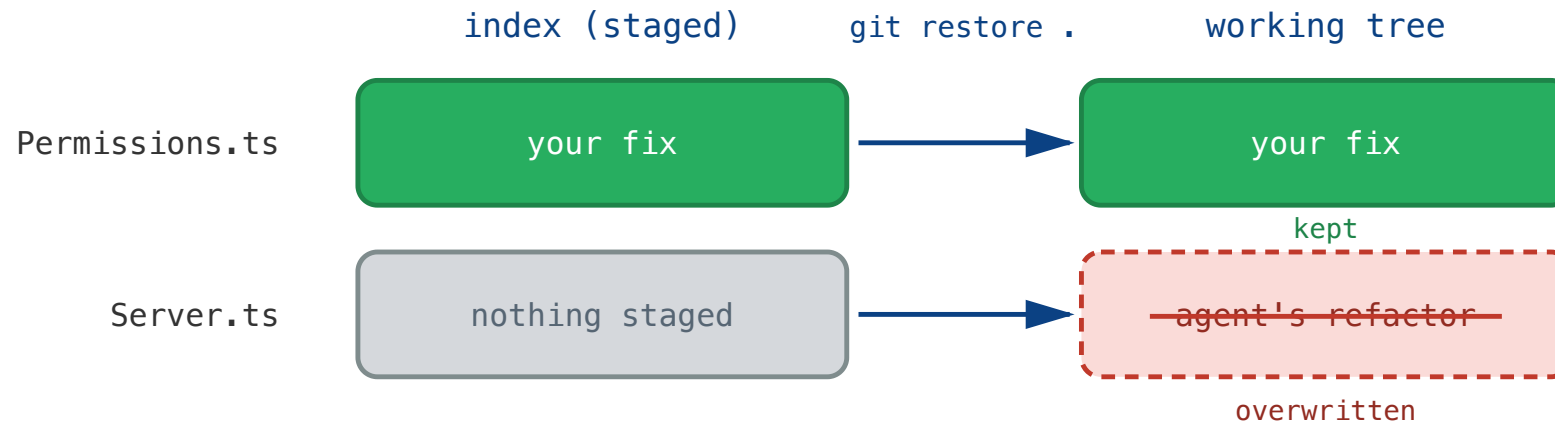
Before you hand a task to an agent, commit!

- Ideally, CI will pass and you'll have one new "known good" point to return to
- If you hand off to an agent in the middle of some other changes, you can't reliably walk back.

"the agent made a mess, let me throw it away" vs. "the agent made a mess somewhere inside two hours of my own uncommitted work."

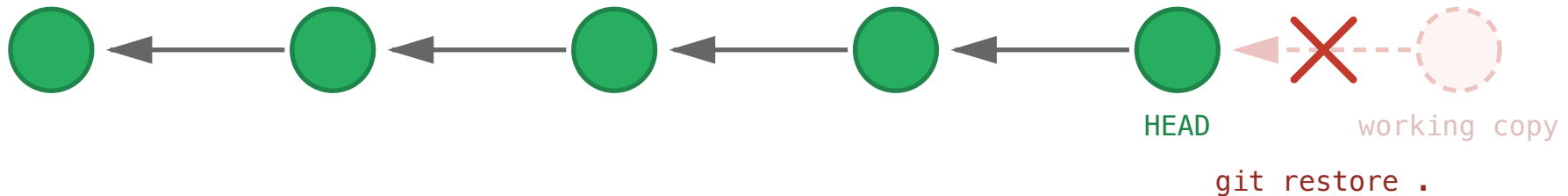
Cleaning up the Working Copy

```
git restore <file>      # discard unstaged changes to one file
git restore .           # ... to every tracked file
```



Cleaning up the Working Copy

```
git reset --hard HEAD    # working tree and index both back to the last commit
```



Both `reset` and `restore` destroy uncommitted work silently—nothing in the log to say it happened.

Neither one touches **untracked** files, so whatever the agent newly created survives. `git clean -nd` lists what is left; `-fd` deletes it.

Who Gets to Undo

While we're talking about them...

Put `git reset --hard` and `git push --force` on the deny list.

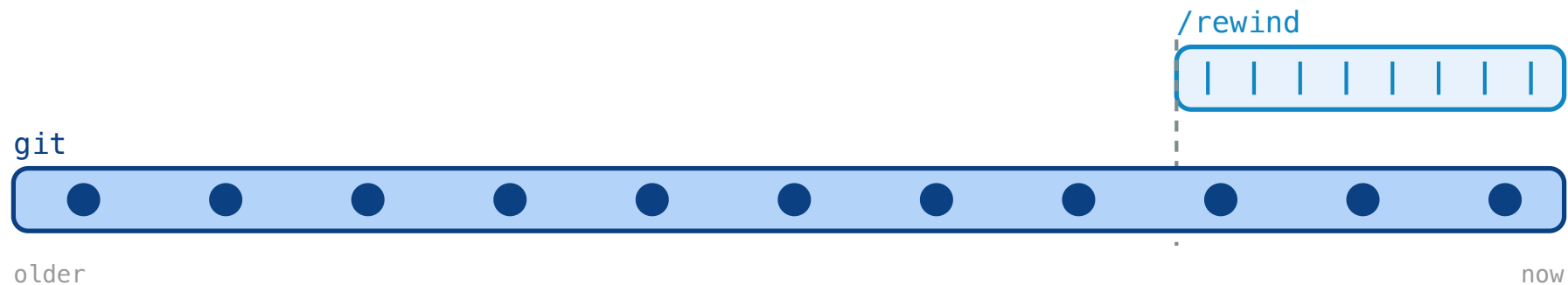
- In theory, an agent might be able to undo your own work
- In practice, this doesn't usually happen. Usually.

ALLOW	DENY
<code>git add</code> , <code>git commit</code>	<code>git reset --hard</code> — discards uncommitted work for good
<code>git status</code> , <code>git diff</code> , <code>git log</code>	<code>git push --force</code> — rewrites history others already pulled
<code>git branch</code> , <code>git blame</code>	<code>git clean -fd</code> — deletes untracked files outright

`/rewind` Is Not the Same Safety Net

Claude Code checkpoints your code at each prompt, and `/rewind` walks those checkpoints backward.

- Good for small steps backwards in a given session
- Does not maintain checkpoints very far back!



git diff: What Changed

`git diff` shows what changed between two commits (or between your working directory and the last commit).

Common uses:

```
git diff           # unstaged changes
git diff --staged   # staged changes
git diff main feature # difference between two branches
git diff HEAD~1 HEAD # last commit's changes
```

Context-saver: Before asking an agent to "fix the build," run `git diff` to see exactly what changed. Paste that diff into your prompt to suggest good places to investigate.

git blame: Who Changed It

`git blame` shows, for each line in a file, who touched it last.

```
git blame src/util/util/Permissions.ts
```

When something goes wrong or looks weird, you can answer some important questions:

- "Who wrote this line, so I can ask them why?"
- "Did a person actually write this, or was it added by an agent?"
 - Particularly within a session, agents are optimistic about code they wrote—often helpful to ask a separate model to review/fix/debug.



Tags and Releases: Stable Snapshots

A **tag** is a named pointer to a specific commit — unlike branches, tags do not move.

```
git tag v1.0.0 4a3b2c1
git push origin v1.0.0
```

Open-source projects use tags to mark stable releases.

VERSION	DESCRIPTION
v2.0.0	Initial release, new major version with potentially breaking changes
v2.1.0	New minor version, perhaps with a feature addition
v2.1.1	A patch version with some bug fixes

Feature Branches

A **feature branch** is a separate line of development for one feature or bug fix. (You already used this in Lab 1.)

```
git checkout -b feature-X      # create and switch to new branch
# ... make changes, commit
git push -u origin feature-X  # push to remote
# ... open PR, review, merge
```

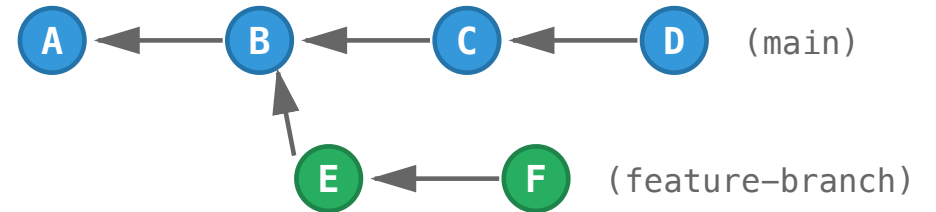
This workflow keeps `main` stable while you experiment. Multiple people (or agents) can work on different features simultaneously. Code review happens on the branch before merging.

Commits Form a Directed Acyclic Graph

Each commit points to its parent. Branches are just pointers (labels) to commits.

- **A, B, C, D** are commits on `main`
- **E, F** are commits on `feature-branch`, which diverged from B
- `main` and `feature-branch` are pointers

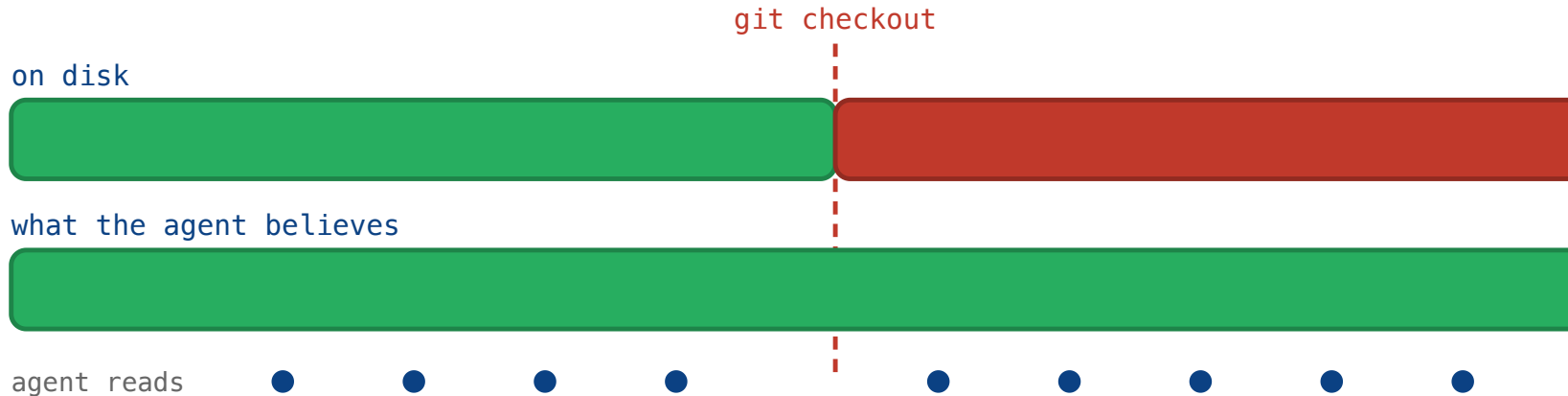
When you create a branch, Git does not copy files — it just creates a new pointer.



Use Caution When Switching Around

`git checkout` rewrites the files in your one working directory.

Do that while an agent is mid-task, and the agent's next file read returns content from a branch it has never seen! Context clash, poisoning, take your pick...



In this and in all things, start regularly from a fresh context.

Handling Two Things at Once?

Imagine that you've started an agent on a big refactor task in worktree A. Mid-task, a "critical bug" comes in. Instead of stopping the agent, what can you do?

git worktree: Parallel Universes

A **worktree** is a separate working directory pointing to a different branch *in the same repo*.

You are working on feature A. You want an agent to explore feature B simultaneously — worktrees make this possible.

```
# Current directory: working on feature-A
git worktree add ../spacebar-feature-B

# Now you have two directories:
# ~/spacebar/                (feature-A)
# ~/spacebar-feature-B (with brand new feature-B branch)
```

You can now create worktree B, fix the bug there, merge/push, and come back — worktree A's agent never stopped running.

Feature Branches vs. Worktrees: When to Use Which

Feature branches

- One person/agent working at a time
- One working directory
- `git checkout` changes which branch you're on

Worktrees

- For multiple experiments in parallel, or multiple agents working at a time
- Useful for comparing two implementations side by side
- Creates multiple working directories, each points to a different branch

Cool idea for worktrees: prototype two implementations in parallel and compare them. More on this soon.

Worktrees Share History, Not Your Untracked Files

A new worktree is a fresh checkout of **tracked** files and nothing else. Everything in `.gitignore` is simply absent, so the project does not build there yet.

```
git worktree add ../spacebar-feature-B
cd ../spacebar-feature-B

npm install                # node_modules/ did not come along
cp ../spacebar/config.json .  # neither did your config or .env
```

This is kind of annoying and slow if you do it a lot.

Cleaning Up

```
git worktree list           # where are they all  
git worktree remove ../spacebar-feature-B  
git worktree prune         # forget ones deleted by hand
```

Deleting the directory in Finder/File Explorer leaves git's bookkeeping behind in `.git/worktrees/`, which is what `prune` exists to sweep up.

Maybe You Don't Always Need Worktrees

I haven't adopted git worktrees yet: if I want to run two agents in isolation against the same repo I do a fresh checkout, often into /tmp.

For riskier tasks I'm currently using asynchronous coding agents—usually Codex Cloud—so if anything goes wrong the worst that can happen is my source code getting leaked (since I allow it to have network access while running). Most of what I work on is open source anyway so that's not a big concern for me.

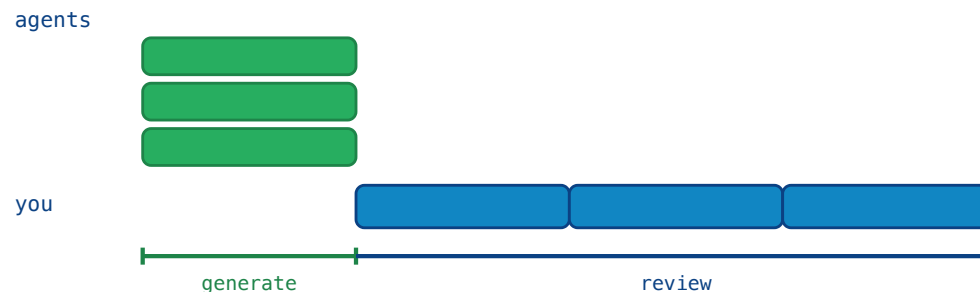
Simon Willison, [Parallel Coding Agents](#) from Oct '25

With Great Parallelism Comes Great Reviewing

Three worktrees means three agents, three branches, and three diffs that one person has to read carefully enough to be accountable for.

Starting work is nearly free now. Reviewing it costs exactly what it always did, and it does not parallelize.

How much you can delegate is not bounded by how many agents you can run. It is bounded by how much output you can actually review!



Recap: Git's Model Makes Parallelism Cheap

- **Blobs, trees, and commits are why branching is nearly free** — a new branch is just a pointer, not a copy of the codebase.
- **Small, focused commits with intent-explaining messages** are easier to review, `git blame`, and revert — for both humans and agents reading history later.
- **`git diff` is a context-saver for agents**: paste the actual change instead of describing it, and the agent has exactly what changed instead of a guess.
- **worktrees parallelize** — when you or an agent needs to work on two things in the same repo simultaneously, without switching branches.

Jujutsu: A Different Bet on the Same `.git`

Git is not the only way to model version control, and one of the alternatives is worth knowing about because of how well it happens to fit agents.

Jujutsu (`jj`) is a version control system that reads and writes an ordinary Git repository. It works on the same objects with a slightly different model.

```
jj git clone git@github.com:upenn-cis-2500/spacebar.git  
jj git init --colocate      # or adopt an existing git checkout in place
```

What changes is the model you work in, not where the data lives.

The Working Copy Is Already a Commit

There is no staging area and no `git add`. The files on disk *are* a commit, and every `jj` command snapshots them into it before doing anything else.

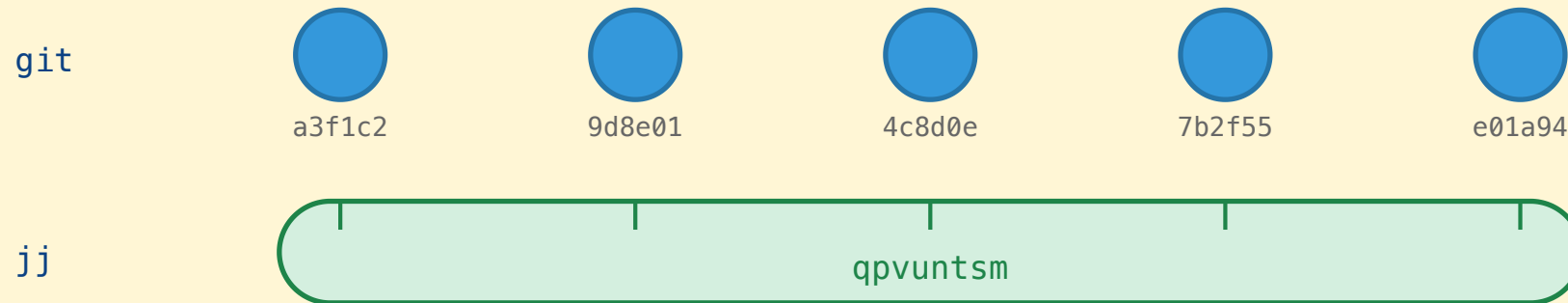
```
jj st          # what is in the working-copy commit right now
jj describe -m "fix admin bypass on channel overwrites"
jj new         # set that change aside, start a fresh one on top
```

An agent that dies mid-task still leaves a snapshot behind!

We'll talk more later about *squashing*, a way to reshape history to improve clarity.

Twenty SHAs, or One Change

Agents do not write a change once. They revise the same logical change twenty times, and git hands you twenty unrelated SHAs with no thread connecting them. `jj` gives that sequence a name you can keep referring to.



Should You Use It? Not Yet.

Get better at Git first and let `jj` mature.

- Something to keep an eye on...
- the training data is overwhelmingly git — ask an agent to use `jj` and it might hallucinate more than is worth dealing with.
- [Jujutsu documentation](#) — tutorial and reference
- [Git comparison](#) — the model differences, stated precisely

Further Reading

- [Pro Git: Branching](#) — free online
- [git worktree](#) — official docs
- [Understanding Git's Data Model](#) — Pro Git chapter

Questions?

`zives@engineering.upenn.edu` · `sharry@engineering.upenn.edu`