ai # guide # tooling

Claude Code Workflows in Practice

TCSS 460 — Client/Server Programming

The earlier guides in this section cover what a coding agent is and how to set one up. This page takes a different angle: here are three real Claude Code sessions from a working project, lightly edited and annotated so you can see what it looks like when a developer is genuinely *directing* the work — architecting, planning, reviewing, and pushing back — rather than accepting whatever the agent produces first.

You are still the engineer. The agent is a tool. Nobody calls a scrum lead a "vibe manager," and nobody should describe a developer working with a coding agent as a "vibe coder." The sessions below are what the work actually looks like.

Why read these?

You can read a list of best practices — "use plan mode," "push back on the first design," "file an issue instead of expanding scope" — and nod along. You will still, very reliably, skip all of them the first time you are under pressure to ship. These transcripts exist because seeing someone else do the thing, with the messy parts left in, sticks better than being told.

These are **not** TCSS 460 sessions. They come from a Java/Swing desktop application — a graph analysis tool used by environmental systems researchers. **Read them for the *process*, not the code.** The specific bug being fixed, the class names, the Maven commands, the `gh` CLI calls — all irrelevant. What matters is:

- How the user steers Claude away from a shortcut
- When Claude proposes multiple options vs. commits to one
- When the user asks a UX question before an implementation question
- What plan mode actually looks like in practice
- How smoke testing catches things automated tests cannot
- When the right move is to revert and file an issue, not ship a partial fix

The three sessions

Interaction 1 — Design, implement, smoke-test, and triage a fix

Claude has just reported "all phases done, tests pass." The user does not take that as "shippable" — they ask Claude to propose a smoke test, run it, and find two bugs. The session then walks through four progressively deeper design options, a clarifying UX question, and a push-not-poll refactor before landing in plan mode. Plan v1 is written, a follow-up question produces plan v2, plan v2 is approved, code ships. Re-testing reveals one smoke-test step still fails — but rather than patch it in this PR, the user and Claude enrich a separate tracking issue with a regression criterion.

Behaviors to watch for:

- Pushing back on minimal fixes
- Asking UX questions before code questions
- Using plan mode for non-trivial changes
- Routing discovered bugs to existing tech-debt issues instead of scope-creeping the current PR

Interaction 2 — Fix, then revert when a follow-up doesn't hold

The immediate continuation of interaction 1. Claude is asked to fix the autosave bug noted during the earlier smoke test. First fix ships. Happy-path smoke test passes. An edge-case test uncovers a new error — Claude's first hypothesis turns out to be wrong, the user clarifies, Claude ships a second fix. The error persists because the stale data on disk was written *before* the fix existed. The user calls it: stop, revert both commits, file an issue.

Behaviors to watch for:

- Edge-case testing beyond the happy path
- Claude's first hypothesis being wrong, and the clarifying detail that redirects it
- Reverting a partial fix being a healthy outcome, not a failure
- Resisting "while I'm here" creep — knowing when to stop and route additional bug-fixing or feature work to its own issue instead of letting the current PR grow

Interaction 3 — Wrapping an epic: routing a cross-cutting feature to its own story

A fresh context window the next day. The user opens by checking memory ("what did we ship last?") instead of diving back in. A focused smoke test of a sibling issue exposes a gap — there's no UI to reset quadrant-driven overrides. Claude offers three fix options, leaning toward the minimal one. The user reframes: this is a cross-cutting feature affecting every panel where factors can be selected, not just quadrants. Session ends with a new story filed, the closing issue amended, and a separate investigation logged for a fresh context window.

Behaviors to watch for:

- Starting sessions with memory checks
- Pushing back when Claude's "small fix" should actually be a new feature story
- Consciously splitting work across context windows instead of bloating one

Conventions used in the transcripts

All three files follow the same formatting:

- **User messages** are rendered as blockquotes, with typos left in. The typos are part of the "real interaction" feel — real sessions do not happen in polished prose.
- **Claude's prose responses** are rendered as normal body text.
- **Claude's tool-use and status lines** (*Explore agent...*, *Read 2 files*, *Entered plan mode*, *Recap:* ...) are rendered in italics, so you can see what Claude was doing under the hood without mistaking it for narrative prose.
- **Long stretches of code or tool output** have been compressed to a single italic line — e.g., *(Claude implements the plan across 4 files.)* The conversation is the point, not the execution.

How to read these

The first time through, skim. Do not try to follow the Java.

On a second pass, for each user turn, ask yourself: **what did the user do here that redirected the conversation?** You will see patterns — asking for a smoke test before `/close-story`, asking a UX question before an implementation question, saying "don't implement yet, let me review," routing a bug to a separate issue. Those patterns are the transferable skill.