Lab 4 — Review and Explain

AI-Assisted Programming for PhD Researchers

So far you have kept the agent on a short leash: a spec, then tests. This lab does the opposite on purpose. You will **vibe-code**: hand the agent a fast, loose request, accept whatever it writes without reading it, and watch it *probably work*. That feeling (it ran, so it must be fine) is the trap this whole session is built to spring. The code lives on a **branch**, walled off from `main`, and it only earns the merge once you have reviewed it line by line, had a second agent review it independently, and fixed the issues that are actually real. Vibe fast; merge slow.

Budget about **40 minutes**. Continue in the same repository, on the state you reached at the end of Lab 3.

Task 1 — Vibe on a Branch

Goal: generate a batch of new code the fast, unguarded way, but do it on a branch so nothing touches `main` until it has passed review.

Create the branch first. Everything you are about to accept unread stays quarantined here:

```
git switch -c vibe-features
```

Now hand the agent a broad, build-fast request and — deliberately, this once — **accept everything without reading it**:

> ℹ Prompt
>
> Add three things, fast, without asking me questions:
>
> - `src/pipeline/plots.py` with `temperature_timeline(df, out_path)` plotting one line per station over time;
> - `src/pipeline/report.py` with `write_report(df, out_path)` writing a markdown summary (row counts, a per-station table from `stats.station_summary`, and a link to the plot);
> - `src/pipeline/cli.py` with an argparse `main` so `pipeline run --raw data/raw/measurements.csv --out out/` runs everything.
>
> Register the `pipeline` script in `pyproject.toml`.

Let it run to completion and accept each change unread. Then run the whole thing end to end:

```
uv run pipeline run --raw data/raw/measurements.csv --out out/
```

It probably works. You get an `out/plot.png` and an `out/report.md`, no error, no complaint. **That feeling is the trap.** "It ran" tells you the code did not crash on one input; it tells you nothing about whether the plot is readable, the report is correct, or the next input will blow up.

> Checkpoint
>
> You are on branch `vibe-features` (`git branch --show-current` confirms it), three new files exist under `src/pipeline/`, `pyproject.toml` has a `pipeline` script, and `uv run pipeline run ...` produced `out/plot.png` and `out/report.md`. You have not yet read a single line of the generated code.

> Explain it
>
> 1. You just ran the pipeline once and it succeeded. List three distinct *specific* things that could still be wrong with this code that a single successful run on this one dataset would not reveal. Be concrete: name the file and the failure, not "it might have bugs".
> 2. Why did you put this work on a branch instead of committing straight to `main`? State the one concrete thing the branch protects you from.

Task 2 — The Review

Goal: replace the good feeling with actual evidence. You will look at the diff, catch yourself doing the thing everyone does, and then review the code against a real checklist.

Start with the overview:

```
git diff main --stat
```

Look at what you just did with that output. You saw three new files, a few dozen added lines each, nothing alarming, and some part of you filed it as "looks reasonable". That reflex is **impressionistic scanning**: judging AI code by its size and shape rather than reading it. It is not a personal failing: it is the documented *default* way programmers review AI-generated code [1]. The checklist below is the deliberate correction: it forces your eyes onto the specific places where generated code goes wrong.

Now open each of the three files (`plots.py`, `report.py`, `cli.py`) and read them **against this checklist**, one file at a time:

Review checklist

1. **Match**: does each function actually do what was asked, with the signature that was asked for?
2. **Silent data changes**: does anything drop, fill, or coerce rows or values without telling you (`dropna`, `fillna`, a silent `astype`)?
3. **Plot correctness**: are the axes labeled *with units*? Are timestamps sorted before plotting, or will the lines zig-zag?
4. **Dependencies**: is every newly imported package real, actually needed, and licensed for your use? (Check `pyproject.toml` against the imports.)
5. **Hardcoded paths**: any absolute paths, or output locations baked into the code instead of taken from arguments?
6. **Edge cases**: what happens on empty input, a missing file, or a station with no readings? Does the output directory get created if it does not exist?
7. **Explainability**: is there any line you cannot explain? A `try/except` that swallows errors, a `sys.exit` buried in a library function, a magic constant?

Work through the checklist for every file and **write down at least three concrete issues** you find: file, line, and what is wrong. Not "could be cleaner"; specific defects a checklist item flags.

Note

A note from me: what people typically find here. The unguarded generator tends to produce the same class of problems: timestamps left **unsorted** so the plot lines become spaghetti; the report **overwriting** an existing file with no warning; a **bare** `except:` that hides real failures; a `sys.exit()` sitting inside library code where an exception belongs; the **output directory not created**, so a missing `out/` crashes the run; and **axes without units** on the plot. If you found three of these, you are reviewing, not scanning.

Checkpoint

You have read all three files line by line and have a written list of **at least three concrete, located issues**: each tied to a specific checklist item, not a matter of taste.

⚠ Explain it

1. Distinguish your finds into two piles: **real defects** (wrong output, a crash on a plausible input, a silently corrupted number) versus **taste** (naming, formatting, a style you would have chosen differently). Which of your three-plus issues are genuinely in the first pile, and why?
2. Before the checklist, your gut said the diff "looked reasonable". Which single checklist item caught the issue you are least proud of missing, and what was it about that issue that let it slide past a size-and-shape scan?

Task 3 — The Second Reviewer

Goal: get an independent read. You reviewed your own accepted code: you are attached to it. A fresh agent with no memory of the session has no such attachment.

Open a **new terminal** and start a **fresh OpenCode session** in the same repository. Give it only the diff and the project's own documents to judge against, and explicitly forbid it from touching anything:

ℹ Prompt

Review the diff between `main` and `vibe-features` against `docs/spec-cleaning.md` and `AGENTS.md`. List the problems ranked by severity, most serious first. Do not fix anything. I only want the list.

Read its list next to yours. Some issues will overlap. Some it caught that you missed. Some you caught that it missed. That gap between the two lists is the whole point of a second reviewer.

Checkpoint

You have two independent problem lists (yours and the fresh agent's) and you have marked, for each issue, whether it appears on both lists, only yours, or only the agent's.

⚠ Explain it

1. **One it found that you missed.** Pick an issue on the agent's list that is not on yours. Why did you miss it: was it hiding in a hunk that looked small or boring in `git diff --stat`? What does that tell you about trusting the `--stat` overview?
2. **One you found that it missed.** Pick an issue on your list that is not on the agent's. Why might it have missed that: did the defect depend on context the diff alone does not carry (the data's real shape, how the output is used)?
3. **Restate it in your own words.** Take *one specific real defect* from either list: name it. In your own words, not the agent's, explain **exactly why it would corrupt a result** (what wrong number or wrong figure would a reader end up trusting?) **and what input would trigger it** (describe a concrete row or file that sets the bug off). If your answer is "it just looks wrong", you have not found the mechanism yet: keep going until you can name the cause and the trigger.

Task 4 — Fix and Merge

Goal: close the loop. Fix the issues that are actually real (and, just as importantly, *leave the taste issues alone*). Verify the tests still pass, then let the branch earn its merge into `main`.

Switch the agent to **plan mode** and have it fix only the agreed real defects. Restraint is the skill here: every "while I'm at it" change is unreviewed code sneaking back in.

ℹ Prompt

Here is the list of confirmed real defects we agreed on: [paste your reconciled list]. Fix exactly these, nothing else: no renames, no reformatting, no extra "improvements". Keep each change minimal. Do not touch the taste issues.

Read the plan, approve it, and let it apply the fixes. Then confirm nothing regressed:

```
uv run pytest
uv run pipeline run --raw data/raw/measurements.csv --out out/
```

Tests green and the pipeline still runs? The branch has earned the merge:

```
git switch main
git merge vibe-features
```

Checkpoint

`vibe-features` is merged into `main`, `uv run pytest` is green, and the fixed `plots.py`, `report.py`, and `cli.py` are on `main`. Your repository now matches the reference state for this lab, tagged `lab-04-done`.

If you fell behind, catch up with:

```
git fetch --tags && git checkout tags/lab-04-done -b catchup-04
```

Explain it

1. You deliberately left the taste issues unfixed. Argue the case for that restraint: what is the cost of every extra "while I'm here" change you fold into a review-and-merge, in terms of what you can still vouch for afterward?
2. Compare this task's workflow to Labs 2 and 3. There you led with a spec or a test; here you led with vibes and cleaned up after. State the one situation in your own research where vibe-then-review is the right trade, and the one where it is clearly the wrong one.

If You Are Ahead

Give the CLI a `--station` filter, and build it the disciplined way: test first, agent second (the Lab 3 movement, applied to a new feature).

1. Write a failing test that runs the pipeline logic with `--station Alpha` (or calls the underlying function directly) on a small two-station frame and asserts that only Alpha's rows survive into the plot and report. Run it and confirm it is red for the right reason: the option does not exist yet.
2. Hand the agent the test as a fixed contract: "Make this test pass. Add a `--station NAME` optional argument to the `run` subcommand that filters the cleaned data to one station before plotting and reporting. Do not modify the test."
3. Run the suite until green, then read the diff: did it filter in the one place you expected, or did it also quietly change the default (no-filter) behavior?

You have now used every tool in the kit (spec, tests, and review) on the same small pipeline. Which one you reach for first is a judgment call; that this code is *reviewable at all* is because you kept it small enough to read.

Note

That's it for this lab. If anything is unclear, ask now — the next session builds on this state. If you fell behind, use the checkpoint tag from the last checkpoint box to catch up.

Bibliography

[1] A. Sarkar and I. Drosos, "Vibe coding: programming through conversation with artificial intelligence," in *Proceedings of the 36th Annual Conference of the Psychology of Programming Interest Group (PPIG 2025)*, Sept. 2025. doi: 10.48550/ARXIV.2506.23253.