Week 2: In-Class Activities: Prompting With RAG

Mizanur Rahman Jewel

February 8, 2026

Here is GitHub URL for the work

Team Members

- Mujahid Swadhin
- Nuzaer Omar

Failure Patterns Observed

When we ran `pytest` using the original refactoring prompt, we observed a high error count (18 failing tasks). The failures consistently clustered into two major patterns:

1. **NameError / Missing-symbol failures (API breakage).** Many tasks failed immediately because the tests attempted to call a function name that was not defined at runtime. This indicated that refactoring often renamed, removed, or otherwise broke the public function contract expected by the test harness.
2. **AssertionError / Behavior drift failures (semantic change).** A smaller subset of tasks failed because the refactored code returned an incorrect value or produced different behavior than expected. This suggested that some refactors introduced unintended logic changes (e.g., edge-case differences, type handling changes, or incorrect transformations).

This separation was important because *API breakage* and *behavior drift* require different controls in the prompt: the former is primarily a naming/interface stability issue, while the latter is a semantics-preservation issue.

What the RAG Chat Retrieved That Was Useful

The RAG chat was most useful when it retrieved concrete evidence from the failing test outputs, especially:

- Exact examples of **NameError** failures listing the missing symbol names the tests were calling.
- Exact examples of **AssertionError** mismatches showing that the function existed but returned the wrong output.

This retrieved evidence helped us diagnose whether each failure was caused by (i) missing/renamed function definitions versus (ii) incorrect logic. That distinction guided how we revised the prompt.

How the Evidence Changed Our Prompt

Based on the above failure patterns, we updated the prompt to add guardrails that remain broadly applicable to refactoring, but specifically prevent the recurring issues:

- We explicitly instructed the model to *not change original function names*. This directly targets the dominant NameError pattern.
- We added guidance that if a function is called incorrectly elsewhere, the model should adjust the call site rather than rename the function itself.
- We added an explicit constraint to *not change logic at all*, emphasizing that refactoring should focus on readability, structure, and safety without altering behavior.
- We included general refactoring safety hints such as using temporary variables for swaps to prevent overwriting, and avoiding unnecessary type conversions (e.g., redundant `complex()` wrapping) that can change edge-case behavior.

Overall, the revised prompt shifted from "make it cleaner" to "make it cleaner under strict interface and semantic invariants."

Successful Refactorings Achieved

Using the original prompt, we observed **18 failing tasks**. After updating the prompt, the failures dropped to **4 failing tasks**. Therefore:

- **Failure reduction:** 18 → 4 (a reduction of 14 failures)
- We also found that the 4 cases that it missed, also have wrong test cases

Notably, after the prompt update, the remaining failures were concentrated in **AssertionError** cases (behavior mismatches) rather than **NameError** cases (missing/renamed symbols). This indicates the revised prompt substantially improved interface stability and reduced "contract-breaking" refactors.

Keeping the Final Prompt Generalizable (Not Task-Specific)

A key goal was to avoid overfitting the prompt to specific failing functions. We kept the prompt generalizable in the following ways:

- We did not add instructions like "for function X, return Y." Instead, we addressed broad failure classes: API stability and semantics preservation.
- We framed requirements around preserving externally observable behavior validated by tests, which applies to any refactoring context.
- Rules such as "do not rename public functions" and "do not change logic" are domain-agnostic and suitable for refactoring across many tasks, languages, and codebases.
- We kept the response structure consistent (single code block + short checklist) to support repeatable evaluation without embedding task-specific content.