Module 4: Intelligent Refactoring and CI/CD, Lesson 3: Automating the Safety Net

Hands-On: AI-Driven Unit & Integration Tests — Live Pricing Engine Demo

Overview

This demo proves three claims from the lecture in real time:

- A strong test prompt with five components produces dozens of correct tests in one shot — including the boundaries you would have missed manually
- Boundary value analysis is where AI consistently outperforms human testers
- Constraining the mock framework in the prompt makes generated tests immediately runnable — no twenty-minute cleanup

You will scaffold a full unit-test suite, generate boundary and error-state tests with mocks, and end with an integration test — using exactly three prompts in the same AI chat session. The student lab extends this to a second function with a service dependency.

Before You Start

- Open your AI tool of choice: Claude.ai, Cursor, or GitHub Copilot Chat
- Have a terminal ready with: pip install pytest pytest-mock pytest-cov
- Save the starter file pricing.py (provided in resources) — it contains the calculate_discount function we will test
- Keep the five-part test prompt template visible on screen or a second monitor throughout
- Do NOT start a new chat between prompts — session continuity is the point

> **Remember**
>
> Every strong test prompt has the same five components. We will prove it by building a complete suite for a Pricing Engine — happy path first, then boundaries and errors, then integration — all in one session. Watch the pattern repeat.

During Hands-On — Keep Visible if Possible

1. ROLE — who the AI is ("senior QA engineer, pytest")
2. CONTEXT — paste the function, its docstring, its dependencies
3. FRAMEWORK — pytest, unittest.mock, project conventions
4. COVERAGE — happy path, errors, boundaries, edge cases
5. OUTPUT FORMAT — naming pattern, AAA structure, one-line "what would break" comment

The Function Under Test

All three prompts target this single function. Skim it now — note the external dependency on get_user_tier, which is a database call. We will mock it.

```
# pricing.py
from decimal import Decimal
from db import get_user_tier   # external dependency — must be mocked

def calculate_discount(user_id: str, cart_total: Decimal) -> Decimal:
    """Return the discounted cart total for a given user.

    Discount rules:
      - tier "premium":  20% off any cart > $0
      - tier "standard": 10% off if cart >= $50, else no discount
      - tier "guest":    no discount
    Raises:
      ValueError on invalid cart_total (None, negative, NaN)
      LookupError if get_user_tier returns an unknown tier
    """
    if cart_total is None or cart_total < 0:
        raise ValueError("cart_total must be non-negative")
    tier = get_user_tier(user_id)
    if tier == "premium":
        return cart_total * Decimal("0.80")
    if tier == "standard":
        return cart_total * Decimal("0.90") if cart_total >= 50 else
cart_total
    if tier == "guest":
        return cart_total
    raise LookupError(f"unknown tier: {tier}")
```

Prompt 1 — Happy-Path Baseline (Establish the Anchor)

> **What to Learn**
> Always start green. Before you ask AI for boundaries or errors, get one passing test that documents correct behavior. That green result is your anchor — every later test builds from it. Notice the five components in action: role, context (function pasted in), framework, narrow coverage target, output format.

Paste this into the AI chat:

```
ROLE: You are a senior QA engineer specializing in Python testing with
pytest
and unittest.mock.

CONTEXT: Test the calculate_discount function below. It takes a user_id
and a
cart_total (Decimal) and returns the discounted total. It calls the
external
function get_user_tier(user_id) which hits a database and must be mocked.

<paste the full pricing.py contents here>

FRAMEWORK: pytest. Use unittest.mock.patch to mock pricing.get_user_tier.
Show the mock setup inside each test.

COVERAGE: Happy path ONLY for this prompt. One test per tier:
  - premium user with cart_total = 100  -> expect 80
  - standard user with cart_total = 100 -> expect 90
  - standard user with cart_total = 49  -> expect 49 (below threshold)
  - guest user with cart_total = 100    -> expect 100

OUTPUT FORMAT:
  - Name every test using: function_when_condition_then_expected
  - Arrange-Act-Assert structure with blank-line separators
  - One-line comment above each test stating what failure it would catch
  - Do NOT generate boundary or error tests yet — those come in the next
prompt
```

> **What you should get**
>
> Four tests, all green when you run them. Each test patches get_user_tier with a return_value, calls calculate_discount, and asserts the expected Decimal. Names look like calculate_discount_when_user_is_premium_then_returns_80_percent_of_cart. The one-line comment above each test reads like documentation.

You should ask:

"Why did we tell the AI not to generate boundaries or errors yet?"

Answer: A passing happy-path test is your behavioral baseline. Without it you have no green anchor — when a boundary test fails later, you cannot tell whether the boundary logic is broken or whether the basic call is broken. One concern at a time.

Prompt 2 — Boundaries + Error States (Where AI Beats Humans)

> **Remember**
> Same chat window. The function and the mock pattern are already in context — I do not repeat them. I just reference what is there and ask for the next layer. This is the entire payoff of session continuity.

Continue in the same chat session — do NOT start a new one:

```
Same context. Now generate two more groups of tests for
calculate_discount.

GROUP A — BOUNDARY VALUES:
  Identify every numeric boundary in the function and produce one test
for
  each value just inside and just outside each limit. At minimum cover:
    - cart_total = 0       (exact lower bound)
    - cart_total = -0.01   (just below — must raise ValueError)
    - cart_total = 49.99   (just below standard threshold)
    - cart_total = 50.00   (exactly at standard threshold)
    - cart_total = 50.01   (just above standard threshold)
  Find any other boundary I missed and add tests for those too.

GROUP B — ERROR STATES:
  - cart_total = None              -> ValueError, message contains "non-
negative"
  - cart_total = -100              -> ValueError
  - get_user_tier returns "vip"    -> LookupError, message contains
"unknown tier"
  - get_user_tier raises TimeoutError -> propagates unchanged
  For every error test: assert on exception TYPE and MESSAGE, and verify
  no mutation occurred before the failure (e.g., the mock was called at
  most once, depending on which error).

MOCK CONSTRAINT: Continue using unittest.mock.patch for get_user_tier.
Do not introduce new mock libraries.

OUTPUT FORMAT: Same naming pattern and AAA structure as Prompt 1. Group
the tests with a blank line and a # ---- BOUNDARIES ---- / # ---- ERRORS
----
header comment so the file reads top-to-bottom in increasing risk.
```

> **What you should get**

Roughly 9–11 additional tests. Boundaries cover both sides of every numeric edge in the function. Error tests assert on exception type AND message text. Mock setup is consistent with Prompt 1 — patches at the same import path, same return-value pattern. Run pytest -v and watch the count jump from 4 to 13+ tests with no manual writing.

Bonus: the AI almost always finds a boundary you forgot — usually around Decimal vs int comparison or the implicit zero case. Count how many of those it caught.

Tip: Run pytest --cov=pricing --cov-branch now. Branch coverage should already be at 100% on calculate_discount. That is the boundary-test payoff — every if/else path executes because the AI generated a test for each side of every boundary.

Prompt 3 — Integration Test (Cross the Module Boundary)

What to Learn

Unit tests proved calculate_discount works in isolation. An integration test proves it works inside a real call flow — through a checkout service that reads from the database, calls our function, and returns a response. We mock only the true external third party (the email receipt service), not the layer we are testing.

Third prompt — still the same session:

```
Same context. Now generate a pytest integration test for the checkout
flow
in services/checkout.py (provided below). The flow is:

<paste services/checkout.py here — it calls calculate_discount and then
 send_receipt_email>

SCOPE OF MOCKING (read carefully):
  - Use a real test database (sqlite in-memory) so get_user_tier returns
    real values from a seeded users table
  - Call the REAL calculate_discount — do NOT mock it, that is the layer
    we are testing
  - Mock ONLY send_receipt_email (it is the external transport layer)

TEST CASES (one function each, descriptive names):
  1. checkout for premium user with cart=100 -> response.total == 80,
     send_receipt_email called once with the discounted total
  2. checkout for standard user with cart=49 -> total == 49, no discount
```

```
  3. checkout for unknown user_id -> 404 response, send_receipt_email
     never called

FIXTURES:
  - test_db: sqlite in-memory, seeded with one premium and one standard
user,
    torn down after each test
  - mock_email: pytest-mock fixture that patches send_receipt_email

OUTPUT FORMAT: Same naming and AAA conventions as before. Add a one-line
comment above each test stating what cross-module behavior it protects.
```

> **What you should get**
>
> Three integration tests with proper fixture-based setup and teardown. The mock_email fixture asserts call counts and arguments. The test_db fixture seeds and tears down cleanly between tests. Critically, calculate_discount is called for real — if you broke it, the integration tests would also fail, which is exactly what you want.

Run the full suite:

```
pytest -v --cov=pricing --cov=services --cov-branch
```

At least one test may need a small fix. This is intentional:

- The AI wrote 95% of the suite in under three minutes. You verify and own the 5%.
- If a test fails, read the error. The fix is usually a wrong import path or a Decimal-vs-float assertion.
- A failing test the AI wrote is still infinitely better than no test at all — and easier to fix than to write from scratch.

Wrap-Up — The Pattern, The Assignment Handoff

> **Remember**
>
> Three prompts. One session. We got a full happy-path baseline, comprehensive boundary coverage, error-state assertions with mocks, and an integration test that crosses module boundaries. The AI found at least one boundary you would have missed. The mock framework was consistent because we constrained it in every prompt. That is what AI-assisted testing looks like.

Key takeaways:

- Happy path first, always — it is your green anchor for everything that follows
- Five-part prompt: role, context, framework, coverage, output format
- Boundary value analysis is where AI consistently outperforms manual testing — let it find the edges you missed
- Constrain the mock framework in the prompt or your tests will hit real databases in CI
- Integration tests mock only true third parties; never mock the layer you are testing
- Session continuity is leverage — the AI carries the function, the mock pattern, and your conventions forward for free

Student lab assignment: Add a second function — apply_promo_code(cart_total, code) — to pricing.py. Use the same three-prompt pattern in a fresh AI session to scaffold its full test suite. Submit pricing.py, test_pricing.py, and a one-paragraph reflection on which boundaries the AI surfaced that you would not have written yourself.