Module 3: Accelerated Code Generation and Debugging, Lesson 4: Using AI to find edge cases before your users do

Hands-On: The Edge Case Cascade — Live Payment Processing Demo

---

Overview

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

- Edge-case thinking is a systematic discipline, not a creative exercise — five categories cover the entire failure space
- A single well-structured prompt generates dozens of tests the happy path would never surface
- Watching tests fail in real time reveals real bugs — not theoretical ones

You will generate a full edge-case test suite for a payment processing function using one carefully crafted prompt. The student lab extends this to additional payment methods and multi-currency scenarios.

Before You Start

1. Open your AI tool of choice: Claude.ai, Cursor, or GitHub Copilot Chat
2. Have a terminal ready with: `pip install pytest pytest-asyncio`
3. Keep the five edge-case categories visible on screen throughout
4. Do NOT run happy-path tests first — you want to see the delta

> **Remember** Every AI-generated test suite is a cascade across five categories: boundary values, null inputs, concurrency, state transitions, and environmental assumptions. We will prove it by generating all five in one prompt — then running the failures live.

During Hands-On — Keep Visible if Possible

5. `BOUNDARY` — zero, negative, and maximum per-transaction limits
6. `NULL INPUTS` — None / empty string on each parameter
7. `CONCURRENCY` — two simultaneous requests for the same user_id
8. `STATE` — account deleted between lookup and charge
9. `ENVIRONMENTAL` — European locale changes currency formatting behaviour

The Prompt — Edge Case Cascade for payment_process

> **What to Learn** This is a single prompt. It names the function signature, states every category explicitly, and gives the AI the exact failure expectations for each case. Watch how structural completeness in the prompt maps directly to structural completeness in the test output.

Paste this into the AI chat:

```
FUNCTION SIGNATURE:
  payment_process(amount: Decimal, currency: str, user_id: UUID) -> PaymentResult
CONTEXT:
  - amount    : payment value; processor limit is $10,000 per transaction
  - currency  : ISO-4217 code (e.g. 'USD', 'EUR'); must be non-empty
  - user_id   : UUID of an existing, active account
TASK:
  Generate a pytest test suite covering ALL FIVE edge-case categories below.
  One test function per case. Use descriptive function names.
CATEGORY 1 — BOUNDARY VALUES
  - amount = 0          → expect ValueError or PaymentError
  - amount = -1         → expect ValueError or PaymentError
  - amount = 10_001.00  → expect TransactionLimitError
CATEGORY 2 — NULL / EMPTY INPUTS
  - user_id = None      → expect TypeError or ValidationError
  - currency = ''       → expect ValueError or ValidationError
CATEGORY 3 — CONCURRENCY
  - Two simultaneous calls with the same user_id → exactly one succeeds,
    the other raises DuplicateTransactionError (use asyncio.gather)
CATEGORY 4 — STATE TRANSITIONS
  - Account deleted between lookup and charge → expect AccountNotFoundError
    (simulate with a mock that raises on the second call to get_account)
CATEGORY 5 — ENVIRONMENTAL
  - European locale (de_DE.UTF-8) changes decimal separator to comma;
    amount formatting must be locale-independent → assert formatted amount
    never contains a comma as decimal separator
BEHAVIOR:
  - Add a one-line comment above each test stating the bug it would catch
  - Use pytest.mark.asyncio for async tests
  - Mock external I/O (database, payment gateway) with unittest.mock.patch
  - Do NOT generate implementation code — tests only
```

What you should get

Eight pytest functions covering all five categories. Each has a one-line comment stating the exact bug it would catch. The asyncio concurrency test uses asyncio.gather. The state-transition test uses a side_effect mock that raises on its second call. The locale test temporarily overrides locale settings and asserts the formatted string.

Analysis — Reading the Output and Understanding the Failures

What to Learn The tests below are a representative sample of what the AI generates. Study the one-line comment on each function first — that comment is the specification. If the test fails, the comment tells you which invariant the implementation violated.

The AI should return code similar to the following. Read the analysis beneath each block:

Category 1 — Boundary Values

```
# Bug caught: no guard against non-positive amounts allows fraudulent reversals
def test_zero_amount_raises():
    with pytest.raises((ValueError, PaymentError)):
        payment_process(Decimal('0'), 'USD', uuid4())
# Bug caught: negative amounts may silently credit the account
def test_negative_amount_raises():
    with pytest.raises((ValueError, PaymentError)):
        payment_process(Decimal('-50.00'), 'USD', uuid4())
# Bug caught: missing per-transaction ceiling allows unlimited charges
def test_over_limit_amount_raises():
    with pytest.raises(TransactionLimitError):
        payment_process(Decimal('10001.00'), 'USD', uuid4())
```

Analysis These three tests cover the edges of a numeric input that most implementations treat as unbounded. The zero-amount test is the most commonly missed: real payment processors reject it, but application-layer code often lets it through to the gateway — causing a confusing downstream error. The over-limit test forces the developer to decide whether the ceiling is enforced here or upstream; if there is no explicit check, the test fails and the gap is visible.

Category 2 — Null / Empty Inputs

```
# Bug caught: None user_id bypasses account ownership validation
def test_none_user_id_raises():
    with pytest.raises((TypeError, ValidationError)):
        payment_process(Decimal('100.00'), 'USD', None)
# Bug caught: empty currency string may default to a wrong currency silently
def test_empty_currency_raises():
    with pytest.raises((ValueError, ValidationError)):
        payment_process(Decimal('100.00'), '', uuid4())
```

Analysis The None user_id case is dangerous in dynamically typed codebases: if the lookup call treats None as a wildcard or the ORM has a lenient nullable FK, the charge may succeed without being linked to any user. The empty currency test is subtler — some gateway clients default to their configured currency when the string is empty, meaning the error is silent and the transaction is denominated incorrectly.

Category 3 — Concurrency

```
# Bug caught: no idempotency key allows duplicate charges on retry / race
@pytest.mark.asyncio
async def test_concurrent_same_user_id_only_one_succeeds():
    uid = uuid4()
    results = await asyncio.gather(
        payment_process(Decimal('50.00'), 'USD', uid),
        payment_process(Decimal('50.00'), 'USD', uid),
        return_exceptions=True
    )
    successes = [r for r in results if not isinstance(r, Exception)]
    errors    = [r for r in results if isinstance(r, DuplicateTransactionError)]
```

```
    assert len(successes) == 1
    assert len(errors)   == 1
```

Analysis This test is likely to fail on a first implementation because most developers do not add an idempotency key or a distributed lock until a double-charge incident reaches production. asyncio.gather with return_exceptions=True is the correct pattern: it lets both coroutines run concurrently and collects exceptions without stopping. The assertions are the specification — exactly one success, exactly one DuplicateTransactionError.

Category 4 — State Transitions

```
# Bug caught: no re-validation after initial lookup allows orphan charges
def test_account_deleted_between_lookup_and_charge():
    uid = uuid4()
    call_count = {'n': 0}
    def flaky_get_account(user_id):
        call_count['n'] += 1
        if call_count['n'] > 1:
            raise AccountNotFoundError(user_id)
        return FakeAccount(uid, active=True)
    with patch('payments.get_account', side_effect=flaky_get_account):
        with pytest.raises(AccountNotFoundError):
            payment_process(Decimal('100.00'), 'USD', uid)
```

Analysis This is a time-of-check to time-of-use (TOCTOU) race. The function looks up the account once to validate it, then calls the payment gateway — if deletion happens in that window, a charge is created for a non-existent account. The side_effect counter simulates the window precisely: first call succeeds (the check), second call raises (the use). If this test fails, the function has no re-validation guard.

Category 5 — Environmental

```
# Bug caught: locale-dependent formatting sends malformed amount to gateway
def test_european_locale_does_not_affect_amount_formatting():
    import locale
    original = locale.getlocale()
    try:
        locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')
        result = payment_process(Decimal('1234.56'), 'EUR', uuid4())
        # Amount sent to gateway must use period, never comma
        assert ',' not in result.gateway_amount_string
        assert result.gateway_amount_string == '1234.56'
    finally:
        locale.setlocale(locale.LC_ALL, original)
```

Analysis Payment gateways expect amounts in a canonical format (e.g. '1234.56'). If the implementation uses Python's locale-aware formatting functions — str(amount), format(), or locale.currency() — the output changes under a European locale to '1234,56', which the

gateway rejects or, worse, silently misparses. The test uses a try/finally block to guarantee locale restoration even on failure, keeping the test environment clean.

Run the Tests

Execute the full suite against your implementation:

```
pytest test_payment_edge_cases.py -v
```

Expected outcome on a typical first implementation:

- test_zero_amount_raises PASSED ← often already guarded
- test_negative_amount_raises PASSED
- test_over_limit_amount_raises FAILED ← no ceiling check
- test_none_user_id_raises PASSED
- test_empty_currency_raises FAILED ← defaults silently
- test_concurrent_same_user_id… FAILED ← no idempotency key
- test_account_deleted_between… FAILED ← TOCTOU not guarded
- test_european_locale… FAILED ← locale.currency() used

Remember Those four failures are real bugs. They were in the function before you ran a single test. Zero happy-path tests were finding them. That is the entire value of the edge-case cascade.

Wrap-Up — The Pattern, The Assignment Handoff

Remember One prompt. Eight tests. Five categories. Four real bugs. The AI did not know anything about your codebase — it only knew the function signature and the categories you named. That is the entire secret: structure your prompt around the failure taxonomy and the AI will fill every cell in the matrix.

Key takeaways:

- Five categories cover the entire failure space — memorise them
- Naming the expected exception in the prompt forces the AI to write an assertion, not just a call
- The one-line comment above each test is the specification — read it before the code
- A failing AI-generated test is more valuable than a passing test that never ran
- Session continuity is not needed here — the function signature is the entire context