Prompt Engineering / Prompt Optimization

Generate test cases that previously broke the prompt and verify they stay fixed after changes.
Difficulty: Advanced
Model: GPT-4 / Claude / Gemini
Use Case: Prompt Regression Testing, Quality Assurance, Version Control
Updated: May 2026
Why This Prompt Exists
You fixed a bug. Then you introduced two new bugs. Without regression testing, you'll never know.

You get:

- fixing one failure and breaking five previously correct cases
- no way to know if a change made things better or worse overall
- manual testing that misses regressions
- fear of changing prompts (because you don't know what will break)
- prompts that degrade slowly over time as fixes accumulate

But regression tests catch issues:

- failure cases: inputs that produced wrong outputs (must stay fixed)
- success cases: inputs that produced correct outputs (must not break)
- edge cases: tricky inputs that test boundaries
- performance tests: latency and token usage baselines

Without regression tests, every change is a risk.

This prompt builds a regression test suite for any prompt.

The Prompt

```
Assume the role of a QA engineer who builds prompt regression tests.

Your task is to create a test suite that prevents regressions.

Generate:

1. TEST CASE INVENTORY (from historical failures)

| Test ID | Input | Expected Output | Last Known Status | Reason for
inclusion |
|---------|-------|-----------------|-------------------|------------
--------|
| FAIL-01 | [input] | [output] | Fixed (v1.2) | Failed in v1.0 |
| SUCC-01 | [input] | [output] | Always passed | Core use case |
| EDGE-01 | [input] | [output] | Intermittent | Tests boundary
condition |

2. REGRESSION TEST MATRIX

| Test ID | v1.0 | v1.1 | v1.2 (current) | v1.3 (proposed) |
|---------|------|------|----------------|-----------------|
| FAIL-01 | FAIL | PASS | PASS | ? |
| SUCC-01 | PASS | PASS | PASS | ? |
| EDGE-01 | FAIL | FAIL | PASS | ? |

3. TEST EXECUTION PROTOCOL
   - Run all tests before any prompt change (baseline)
```

- Run all tests after change (compare)
- Any test that passes before and fails after = regression
- Any test that fails before and passes after = improvement

4. REGRESSION CATCHES (if any)
- Tests that passed in baseline but fail in proposed version
- Severity (Critical / Major / Minor)

5. PASSING IMPROVEMENTS (if any)
- Tests that failed in baseline but pass in proposed version

6. GO/NO-GO RECOMMENDATION
- Deploy (no regressions, or only minor regressions on low-priority tests)
- Fix regressions first (critical or major regressions present)
- More testing needed (mixed results, unclear winner)

7. REGRESSION TEST PROMPT (ready to use)
- A prompt that runs this test suite automatically

INPUTS:

Current prompt version (baseline):
[PASTE OR DESCRIBE]

Proposed prompt version (candidate):
[PASTE OR DESCRIBE]

Historical failures (inputs that broke previous versions):

```
[PASTE 5-10 FAILURE EXAMPLES]

Core success cases (inputs that should always work):
[PASTE 5-10 SUCCESS EXAMPLES]

Task type:
[CLASSIFICATION / GENERATION / EXTRACTION / OTHER]

RULES:
- Build test suite incrementally (add each failure as a regression test)
- Run regression tests before every prompt change (not just major releases)
- Automate testing where possible (manual regression testing is too slow)
- If a test fails intermittently, run it multiple times (3-5) to establish baseline flakiness
- Keep test suite up to date (remove tests for features you no longer support)
```

How To Use It

- Add every failure you fix to your regression test suite.
- Run regression tests before every prompt change — not just major releases.
- Automate testing (use LLM-as-judge or exact match where possible).
- If a test is flaky (passes sometimes, fails sometimes), run it multiple times.
- Don’t deploy if critical regressions are present — fix them first.

Example Input

**Current prompt version:**

“Classify customer emails as URGENT, NORMAL, or LOW. Respond with one word.”

**Proposed prompt version:**
“Classify customer emails as URGENT (requires immediate action), NORMAL (can wait 24 hours), or LOW (informational only). Respond with one word. If email contains ‘urgent’, ‘asap’, or ‘deadline’, prioritize URGENT.”

**Historical failures:**
“Input: ‘My account is locked and bill due tomorrow’ → v1.0 gave NORMAL (wrong)”
“Input: ‘URGENT: Payment failed’ → v1.0 gave NORMAL (wrong)”

**Core success cases:**
“Input: ‘Thanks for your help’ → LOW”
“Input: ‘Question about pricing’ → NORMAL”

Why It Works
Most prompt changes are tested manually on a few examples — which almost guarantees regressions will slip through.

This framework improves outcomes by forcing:

- test case inventory (what has broken before?)
- regression test matrix (compare versions systematically)
- execution protocol (test before and after every change)
- regression detection (identify what broke)
- go/no-go decision (deploy or fix first?)

Great regression testing doesn’t prevent all bugs — it catches them before they reach production.

## Build Better AI Systems

Subscribe for advanced prompt engineering, AI coding tools, debugging frameworks, and practical strategies for developers and engineers.

Carefully engineered prompts for people doing real work.

### Share this:

- Share on Facebook (Opens in new window) Facebook
- Share on X (Opens in new window) X

See also Token Efficiency Optimizer