A Comprehensive Guide to PromptFoo

Testing, Red Teaming & Reliability for Large Language Models

Prompt & Context Engineering Team

1. The Production Reality & Risk Analysis

In traditional software engineering, broken code triggers compile errors and immediate pipeline failures. In AI systems, however, broken prompts trigger **silent failures**. Large Language Model (LLM) outputs are inherently non-deterministic. A prompt performing perfectly today can degrade tomorrow following an invisible background model update by the API provider.

In production pipelines — such as automated Quality Assurance, RAG chatbots, and data extraction — a broken prompt is a critical business risk.

Key Failure Modes

- **Output Drift:** The same exact input yields unpredictable response variations over time, breaking downstream JSON parsers.
- **Silent Regressions:** Tweaking a prompt to fix an edge case silently breaks a previously working use case (the "whack-a-mole" problem).
- **Hallucinations:** The model fabricates information or ignores explicit system instructions buried within a massive context window.
- **Cost Spikes:** Unoptimized prompts generate unnecessarily verbose outputs, skyrocketing API token usage.

> Prompt Change → Behavioral Shift → Silent Output Degradation → Unnoticed Business Impact

> Takeaway: Prompts must be treated as compiled code requiring rigorous, automated unit testing.

2. Unit Testing for LLMs

PromptFoo is an open-source Command Line Interface (CLI) tool that brings test-level rigor to prompt engineering. Crucially, your data stays entirely local, ensuring maximum safety for sensitive transcripts and client confidentiality.

Engineers define system prompts, test variables, and rigorous pass/fail criteria in a YAML file. PromptFoo executes these tests concurrently against designated LLMs and instantly surfaces regressions.

Real-World Application: Contact Center QA

Consider a workflow evaluating customer call transcripts manually for tone and compliance. Manual evaluation is unscalable, but using an untested LLM is risky. PromptFoo automates the evaluation of your LLM evaluator.

Listing 1: Example — Basic PromptFoo configuration

```
providers:
  - openai:gpt-4o
prompts:
  - file://prompts/qa_system_prompt_v1.txt
tests:
  - description: "Standard refund request"
    vars:
      transcript: file://transcripts/standard_refund_001.txt
    assert:
      - type: contains
        value: "resolution_status: resolved"
      - type: llm-rubric
        value: "Explicitly state that the agent remained professional."
```

3. Deep Dive: Output Assertions

Testing probabilistic LLMs requires a layered approach to evaluation. PromptFoo provides three distinct tiers of assertions to handle this complexity.

Tier 1: Deterministic Assertions

Fast and cheap. These use traditional software logic to evaluate structural outputs.

- `contains` / `regex`: Validate output structure, such as ensuring a response contains a specific XML tag.
- `is-json`: Validates that the output can be perfectly parsed as JSON.
- `javascript`: Write custom scripts to check string lengths or array sizes.

Tier 2: Semantic Assertions

These use mathematical embeddings to check if the *meaning* of the output aligns with the expected answer, even if the exact words differ.

- `similar`: Compares cosine similarity. (Expected: "The sky is blue." Output: "It's a blue sky today." → PASS).

Tier 3: LLM-as-a-Judge Assertions

These utilize a secondary, highly capable model to grade the primary model.

- `llm-rubric`: You provide a grading rubric. The judge model reads the output and returns a PASS/FAIL along with a rationale.
- `factuality`: Checks if the output aligns perfectly with a provided source of truth, heavily penalizing hallucinations.

4. Evaluating RAG Systems

Retrieval-Augmented Generation (RAG) introduces dynamic context. When evaluating a RAG prompt, you must test how accurately the model interacts with the injected context fetched from your vector database.

- **Faithfulness:** Is the output entirely derivable from the injected context? If the model utilizes outside knowledge, this test fails.
- **Answer Relevance:** Does the output directly address the user's query?

Listing 2: RAG Evaluation Pattern

```
tests:
  - vars:
      user_query: "What is the deductible for the Gold Plan?"
      retrieved_context: "The Gold Plan has a $500 deductible."
    assert:
      - type: factuality
        value: "The deductible is $500"
      - type: llm-rubric
        value: "Answer STRICTLY based on the retrieved context."
```

5. Automated Red Teaming

Deploying an LLM without adversarial testing is equivalent to deploying a web app without a firewall. PromptFoo includes a built-in automated red teaming suite (`promptfoo redteam`) to proactively attack your prompts.

- **Prompt Injection:** Attempts to maliciously override system instructions.
- **Jailbreaks:** Complex cryptographic or role-play attacks designed to bypass API safety filters.
- **PII Leakage:** Tests whether the model can be tricked into outputting sensitive data present in its context window (e.g., SSNs, proprietary data).

Listing 3: Red Teaming Configuration

```
targets:
  - id: openai:gpt-4o
    prompts: [file://system_prompt.txt]
plugins:
  - id: prompt-injection
  - id: pii
strategies:
  - id: jailbreak
```

6. CI/CD Integration

A prompt change must require the exact same pull request, peer review, and automated testing rigor as a standard backend code change.

The Workflow

1. Developer alters `system_prompt.txt` and opens a Pull Request.
2. GitHub Actions automatically triggers a PromptFoo evaluation suite.
3. If regressions occur (e.g., accuracy drops below 95%), the PR is blocked.
4. If tests pass, the PR is merged and safely deployed.

Listing 4: GitHub Actions Workflow snippet

```
steps:
  - name: Install PromptFoo
    run: npm install -g promptfoo
  - name: Run Evaluation Suite
    run: promptfoo eval
  - name: Assert Quality Thresholds
    run: promptfoo check   # Fails pipeline if thresholds aren't met
```

7. Best Practices for Production

To maximize the value of automated LLM testing, teams should adopt these operational methodologies:

- **Modularize Prompts:** Avoid massive, monolithic prompts. Break prompts into atomic steps (route, extract, format) and test each independently.
- **Maintain a "Golden Dataset":** Curate a dataset of 50 to 100 highly diverse, difficult edge cases. This becomes your immutable baseline.
- **Enforce Strict Schemas:** For API integrations, use JSON Schema inside your prompts and validate it with the `is-json` assertion.
- **Benchmark Updates:** When an API provider releases a new model, run your baseline against the old and new versions concurrently before upgrading.

Conclusion

Prompt engineering is no longer a dark art of guessing the right adjectives; it is a rigorous discipline requiring empirical validation. Frameworks like PromptFoo bridge the gap between natural language processing and deterministic software testing, allowing organizations to deploy AI systems with absolute operational confidence.

Revision #3
Created 2026-06-17 13:21:22 UTC by Umar M Ahmed
Updated 2026-06-17 13:25:25 UTC by Umar M Ahmed