LLM Testing & Evaluation Solution — PromptFoo + TruLens + LangChain

Author: ChatGPT (GPT-5 Thinking mini) **Date:** 2025-11-23

Executive summary

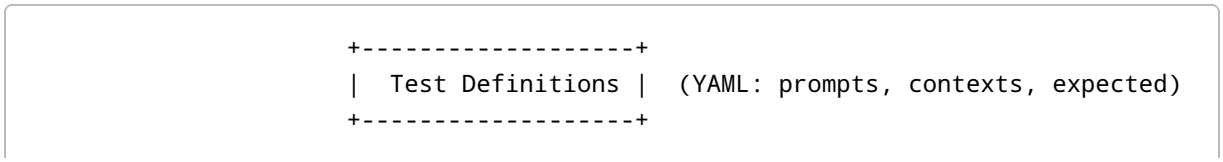
This document describes a complete, production-ready solution to test, evaluate, and monitor LLM-driven pipelines that extract structured data (e.g., financial figures from annual reports) and generate text outputs. The stack uses **LangChain** for pipeline/orchestration and context (RAG), **PromptFoo** for automated multi-run regression testing and pass/fail gating, and **TruLens** for deep evaluation (faithfulness, hallucination detection, traceability). The framework supports **custom REST LLMs** and **Google Gemini** (or other model endpoints) and integrates into CI/CD.

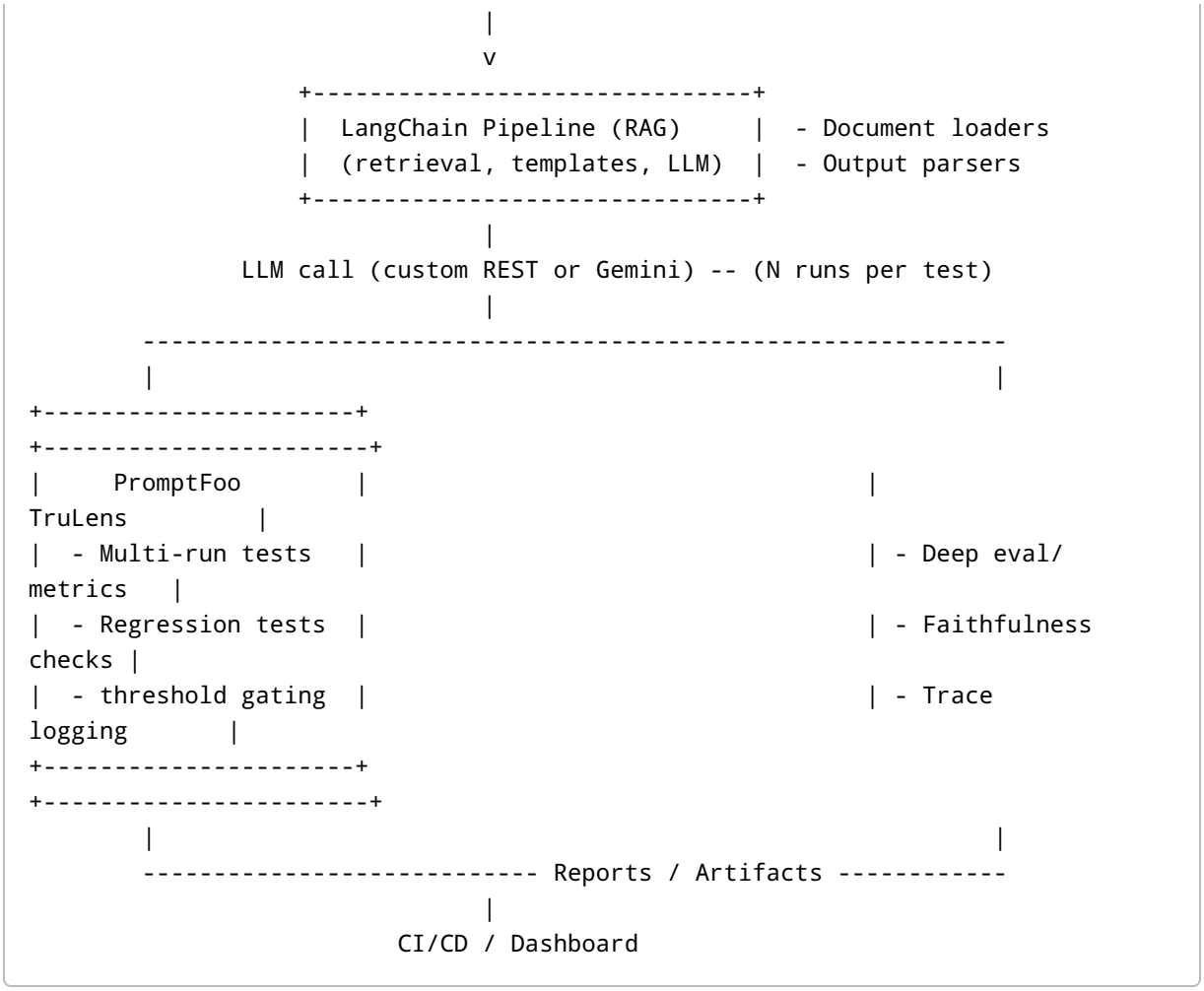
Goals: - Provide deterministic test harness behavior for *probabilistic* models via multiple runs + thresholds. - Support both numeric and free-text evaluations (exact, tolerant, and semantic similarity). - Be model-agnostic (work with custom REST LLMs and Gemini). - Provide observability and deep evaluation for hallucination/factuality. - Integrate into CI/CD for gating prompt/context/model changes.

Table of contents

---

1. Architecture (high level)





---

2. Components and responsibilities

LangChain

- Document loaders: PDF, DOCX, HTML, plain text
- Chunking & retriever (Vector DB / local search)
- Prompt templates and output-enforcement (request JSON schema)
- LLM adapters: custom REST wrapper + Gemini wrapper
- Output parsers: robust numeric extraction, currency normalization, unit handling
- Single entrypoint function that runs the pipeline and returns structured output (JSON)

PromptFoo

- Declarative test definitions (YAML)
- Runs tests multiple times (N runs)
- Supports custom assertions (javascript/python) and thresholds
- Produces test reports & visualizations

- Works against any REST endpoint or custom provider; can call LangChain pipeline via a small HTTP wrapper if desired

TruLens

- Attach to LangChain pipeline (or wrap pipeline call)
- Collect chain traces & intermediate activations
- Compute groundedness, hallucination risk, context relevance
- Produce dashboards for qualitative review

CI/CD

- Run PromptFoo test suites on PR / merge to main
- Gate deployments if average accuracy < threshold or if TruLens detects regressions
- Store artifacts: PromptFoo report HTML, TruLens logs, LangChain call logs

---

3. Test case data model (YAML schema)

Use YAML to store test metadata; keep contexts as file references so Git can version them.

```yaml
# tests/finance/extract_revenue.yaml
id: finance-revenue-001
description: "Extract Total Revenue (FY 2024) from annual report"
prompt_template: |
  You are a financial data extractor. Extract the requested fields from the context and return JSON.
  Return JSON object: {"revenue": number, "currency": string}
context_file: docs/annual_report_2024.pdf
model: custom-llm  # or gemini
runs: 5
pass_threshold: 0.8  # fraction of runs that must pass
evaluation:
  type: numeric
  field: revenue
  expected: 43560000000
  tolerance: 0  # exact match; can be absolute or percent
post_processing:
  - normalize_currency: true
  - convert_to_inr: true

# Optional: semantic eval for text
text_test_id: finance-summary-001
prompt_template: "Summarize the management commentary in 50 words"
evaluation:
  type: semantic
  field: summary
```

```
  similarity_threshold: 0.82
  reference: tests/finance/expected_summary.txt
```

Notes: - Keep `context_file` small by pre-extracting relevant sections when possible. - For PDFs, include a pipeline step that extracts text and normalizes numeric formats.

---

4. LangChain pipeline design and code snippets

4.1 Custom REST LLM adapter

```python
# src/langchain_adapters/custom_rest_llm.py
from langchain.llms.base import LLM
import requests

class CustomRestLLM(LLM):
    """Simple LangChain LLM wrapper for any REST endpoint returning JSON or
text."""
    def __init__(self, api_url: str, api_key: str = None, timeout: int = 30):
        self.api_url = api_url
        self.api_key = api_key
        self.timeout = timeout

    def _call(self, prompt: str, stop=None) -> str:
        payload = {"prompt": prompt}
        headers = {"Content-Type": "application/json"}
        if self.api_key:
            headers["Authorization"] = f"Bearer {self.api_key}"

        resp = requests.post(self.api_url, json=payload, headers=headers,
timeout=self.timeout)
        resp.raise_for_status()
        data = resp.json()
        # Commonly the text is data["output"] or data["answer"]; adapt to your
API
        return data.get("output") or data.get("answer") or data.get("text")

    @property
    def _identifying_params(self):
        return {"api_url": self.api_url}
```

4.2 Gemini wrapper (example using Google GenAI SDK) — or use REST if required

```python
# src/langchain_adapters/gemini_llm.py
from langchain_google_genai import ChatGoogleGenerativeAI
```

```
def create_gemini(model_name: str, api_key: str):
    return ChatGoogleGenerativeAI(model=model_name, google_api_key=api_key)
```

4.3 Document loader & retriever

- Use `UnstructuredPDFLoader` or `PyPDF2` to extract text.
- Chunk text (e.g., 500–1200 tokens) and index them into a vector store (FAISS, Milvus, or local Chroma).

```
from langchain.document_loaders import UnstructuredPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

loader = UnstructuredPDFLoader("docs/annual_report_2024.pdf")
docs = loader.load()
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1200,
chunk_overlap=200)
chunks = text_splitter.split_documents(docs)
# use Chroma or FAISS to index
```

4.4 Prompt template + structured output enforcement

- Ask LLM to return JSON and give a sample schema.
- Optionally, use `jsonschema` validator after the LLM returns.

```
prompt_template = """
You are a financial data extractor. Given the context, return a JSON object
containing:
{
  "revenue": number,
  "currency": string
}
Context:
{context}
"""

# call:
final_prompt = prompt_template.format(context=context)
raw_output = llm(final_prompt)
# parse JSON robustly with heuristics
```

4.5 Robust output parsing

- The LLM may add commentary. Use heuristics to extract JSON (find the first `{` to last `}`), then `json.loads`.

- Normalize numbers: remove commas, parentheses (for negatives), and currency symbols. Convert words ("forty million") using `word2number` libs if needed.

```
import re, json

def extract_json_from_text(text: str) -> dict:
    # heuristic: find first { and last }
    start = text.find('{')
    end = text.rfind('}')
    if start == -1 or end == -1:
        raise ValueError("No JSON-looking text found")
    jtext = text[start:end+1]
    return json.loads(jtext)

def normalize_number(s: str) -> int:
    s = s.replace(',', '')
    s = s.replace('(', '-').replace(')', '')
    # remove non numeric except dot and minus
    s = re.sub(r'[^0-9.-]', '', s)
    if s == '':
        return None
    return int(float(s))
```

---

5. PromptFoo test examples

5.1 PromptFoo model configuration (supporting custom REST LLM)

```
# promptfoo.config.yaml
models:
  - id: custom_llm
    type: custom
    apiBaseUrl: https://api.my-llm.company/v1/generate
    apiKey: ${CUSTOM_LLM_KEY}

  - id: gemini
    provider: google
    model: gemini-pro
    apiKey: ${GEMINI_KEY}
```

5.2 PromptFoo test definition (financial extraction)

```
# tests/promptfoo/extract_revenue.yaml
tests:
```

```
  - name: extract_revenue_2024
    model: custom_llm
    prompt: |

You are a financial data extractor. From the following context extract the
company's Total Revenue for FY 2024
      Return only JSON: {"revenue": number, "currency": string}

    context: "{{ file:docs/annual_report_2024.txt }}"

    runs: 5
    threshold: 0.8
    asserts:
      - type: javascript
        code: |
          const expected = 43560000000;
          const out = JSON.parse(output);
          const actual = Number(out.revenue);
          Math.abs(expected - actual) <= 1
```

5.3 Running PromptFoo

```
# run suite
promptfoo eval tests/promptfoo/extract_revenue.yaml
# view results
promptfoo view
```

PromptFoo will produce a report (HTML) and show pass/fail and sample outputs per run.

---

6. TruLens evaluation setup

TruLens attaches to your pipeline to collect traces and evaluate groundedness. It can run in-process around your LangChain function or via wrapper.

6.1 Installation

```
pip install trulens_eval
```

6.2 Basic wrapper example

```
# src/trulens_wrapper.py
from trulens_eval import Tru, Feedback
```

```python
from trulens_eval.feedback import Groundedness

tru = Tru(app_id="financial-pipeline")

# define feedback probes
grounded = Groundedness()
feedback = Feedback(grounded.groundedness_measure, name="groundedness")

# example: wrap the call results
def run_with_trulens(prompt, context):
    # create an example object or run directly
    with tru.run(example={"input": prompt, "context": context}) as r:
        output = run_financial_extraction(prompt, context)
        r.add_output({"text": output})
        tru.add_feedback(feedback)
        return output
```

6.3 Metrics TruLens provides

- Groundedness / Faithfulness score
- Hallucination markers (claims not supported by context via retrieval similarity)
- Relevance of retrieved chunks
- Attention to source passages (via chain traces)

TruLens dashboard can be run locally and exported as JSON for CI comparisons.

---

7. Evaluation functions & scoring

Numeric evaluation (recommended)

- **Absolute tolerance**: `abs(actual - expected) <= tolerance`
- **Relative tolerance**: `abs(actual - expected) / expected <= pct_tolerance`
- **Rounding tolerance**: compare at appropriate magnitude (e.g., billions)

Return 1 for pass, 0 for fail for each run. Aggregate across N runs to compute accuracy.

Text evaluation

- **Exact match** for canonical strings
- **Semantic similarity** using sentence-transformers / OpenAI embeddings. Compute cosine similarity and compare against threshold.
- **Key-phrase presence**: check if required phrases appear

Composite scoring

- Weighted average of multiple sub-scores (e.g., numeric accuracy 70% + groundedness 30%).

- Example rule: `pass` if composite_score >= 0.8

Example evaluator (python)

```python
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

def numeric_eval(actual, expected, tolerance=0):
    return abs(actual - expected) <= tolerance

def semantic_eval_text(output_text, reference_text, embed_fn, threshold=0.82):
    v_out = embed_fn(output_text)
    v_ref = embed_fn(reference_text)
    sim = cosine_similarity([v_out], [v_ref])[0][0]
    return sim >= threshold, sim
```

---

8. CI/CD: GitHub Actions example

8.1 Workflow: run PromptFoo + TruLens checks on PR

```yaml
name: llm-tests
on:
  pull_request:
    types: [opened, synchronize, reopened]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.11'
      - name: Install deps
        run: |
          pip install -r requirements.txt
          npm install -g promptfoo
      - name: Run PromptFoo
        env:
          CUSTOM_LLM_KEY: ${{ secrets.CUSTOM_LLM_KEY }}
          GEMINI_KEY: ${{ secrets.GEMINI_KEY }}
        run: |
          promptfoo eval tests/promptfoo --output reports/promptfoo.json || true
```

```
          promptfoo view --open false || true
      - name: Evaluate PromptFoo results
        run: |
          python scripts/parse_promptfoo_report.py reports/promptfoo.json
          # This script should exit non-zero if thresholds fail
      - name: Run TruLens checks
        run: |
          python scripts/run_trulens_checks.py || true
      - name: Upload artifacts
        uses: actions/upload-artifact@v4
        with:
          name: llm-test-reports
          path: |
            reports/
            trulens_logs/
```

Notes: - `parse_promptfoo_report.py` should parse PromptFoo output and cause job to fail if overall accuracy below configured thresholds. - TruLens steps may upload logs; use them for historical comparisons.

---

9. Folder structure and repo layout

```
llm-test-framework/
├─ docs/
│  └─ annual_report_2024.pdf
├─ src/
│  ├─ langchain_adapters/
│  │  ├─ custom_rest_llm.py
│  │  └─ gemini_llm.py
│  ├─ pipeline/
│  │  └─ run_financial_extraction.py
│  ├─ evaluators/
│  │  ├─ numeric_eval.py
│  │  └─ semantic_eval.py
│  └─ trulens_wrapper.py
├─ tests/
│  └─ promptfoo/
│     └─ extract_revenue.yaml
├─ scripts/
│  ├─ parse_promptfoo_report.py
│  └─ run_trulens_checks.py
├─ requirements.txt
```

```
├─ promptfoo.config.yaml
└─ .github/workflows/llm-tests.yml
```

---

10. Security, cost, and operational considerations

Security

- Protect API keys using CI secrets.
- Limit test data exposure; redact PII from contexts before storing in repo.
- If using internal LLM endpoints, restrict network access to CI runners (VPC/GitHub self-hosted runners).

Cost

- Multi-run tests (5–20 runs) multiply API usage. Cache inexpensive tests. Use smaller models for wide regression tests and run expensive models less frequently.
- Use sampling/partial context during CI; run full extensive suites on scheduled nightly pipelines.

Observability

- Centralize logs (PromptFoo reports + TruLens logs + LangChain traces).
- Retain artifacts for a reasonable duration to debug regressions.

---

11. Runbook & troubleshooting

When a test fails

1. Open PromptFoo report and inspect failing runs.
2. Re-run the specific test locally with increased logging.
3. Check TruLens metrics for groundedness drop or hallucination increase.
4. Compare prompt versions (PromptLayer history) and model version used.
5. If context changed (document update), re-check text extraction and normalization pipeline.

Common failure modes

- JSON parse errors: improve output enforcement and add robust heuristics.
- Currency/unit mismatch: normalize currency in parser and include per-field unit checks.
- Off-by-scale errors (millions vs thousands): include unit hints in prompt and evaluator tolerant logic.

---

12. Next steps & roadmap

Short-term (1–2 weeks)

- Create skeleton repo with LangChain pipeline + sample PDF loader.
- Add CustomRestLLM adapter and a tiny local mock LLM (for offline tests).
- Create 2 PromptFoo test YAMLs (numeric and text) and run locally.
- Integrate TruLens with one feedback probe (groundedness).

Mid-term (1–2 months)

- Harden parsers (tables, inline numeric mentions, footnotes).
- Add richer TruLens probes (hallucination detection, retrieval relevance).
- Implement CI/CD gating and automated report uploads.

Long-term

- Add historical metrics dashboard and alerting for regressions.
- Add dataset drift detection for context documents.
- Build a lightweight UI for triaging failing prompts and examples.

Appendix A — Useful snippets & tools

- sentence-transformers: semantic embeddings for text similarity
- faiss/chroma: vector stores
- Unstructured / Apache Tika / pdfplumber: PDF extraction
- jsonschema: validate LLM JSON outputs
- word2number: convert words to numbers

If you want, I can now: - generate the **starter repo** with the files and code above, - or produce individual files (LangChain adapter, PromptFoo YAML, TruLens wrapper) in the canvas for you to copy.

Which would you like me to create next?

8. DeepEval Comparison and Considerations

DeepEval is another evaluation framework for LLM systems. Here is how it compares to the tools chosen in this solution (PromptFoo, TruLens, LangChain), especially **when avoiding LLM-as-judge**, which is a requirement for your architecture.

8.1 Strengths of DeepEval

- Built-in metrics (factual consistency, hallucination detection, answer relevancy, etc.)
- Supports multi-run evaluations similar to PromptFoo

- Offers embedding-based and rule-based evaluators (no LLM-as-judge required)
- Python-native and integrates with LangChain
- Flexible custom evaluators suitable for deterministic financial extraction tests

8.2 Limitations Compared to PromptFoo + TruLens

- Less suited for large matrix testing compared to PromptFoo
- Lacks deep observability and tracing (TruLens advantage)
- Many examples assume LLM-as-judge (avoidable but requires customization)
- PromptFoo provides stronger CI/CD gating and regression testing workflows

8.3 Role of DeepEval in This Architecture

DeepEval is recommended as a **supplementary evaluator** used for: - Embedding similarity scoring - Rule-based validation - Automated metrics (precision, recall, hallucination rate)

8.4 Recommended DeepEval Evaluators (No LLM-As-Judge)

- Exact Match Evaluator → deterministic numeric extraction
- Regex Match Evaluator → financial figure validation
- Embedding Similarity Evaluator → natural language outputs
- Custom Evaluators → domain-specific checks

8.5 Final Recommendation

DeepEval works best *alongside* PromptFoo and TruLens: - **PromptFoo** → regression testing, CI gating, multi-run evaluations - **TruLens** → traceability, context attribution, explainability - **DeepEval** → metrics-level scoring, semantic evaluation, deterministic rules

This combination keeps the system robust while avoiding LLM-as-judge entirely.