

AI Evaluation & Evals

How to measure, test, and improve LLM application quality systematically.

Last reviewed 28 Sep 2026 ·

Prerequisites: [LLM APIs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) · [Prompt Engineering](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/prompt-engineering/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/prompt-engineering/)

Related: [RAG](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) · [AI Observability](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/) · [Data Quality](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) (https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

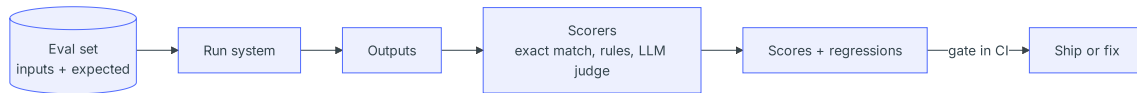
Challenge: A prompt, model, or retrieval change may look better on a handful of examples while silently degrading others. Conventional unit tests do not apply directly: LLM outputs vary between runs and are often open-ended, so `assert output == expected` is not a meaningful check.

Solution: an evaluation ("eval") combines a fixed set of realistic inputs with a method for *scoring* outputs — exact checks where possible, programmatic checks (valid JSON, executable SQL), an LLM grader applying a rubric, or human review. The eval runs on every change, and scores are compared, much like a test suite that reports a rate rather than pass/fail.

```
eval set (inputs + expectations)  →  your LLM app
(version A / B)  →  graders  →  scores

150 real questions, edge cases      prompt v7,
claude-sonnet-5-5      code checks  A: 86%
known failures from production      prompt v8,
claude-sonnet-5-5      LLM judge    B: 91% (ship)
```

Relevance to data engineering: evals apply data quality practices to model outputs — versioned test data, automated checks, thresholds, and a CI gate.



On this page

Basic - [Why Evals Matter](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#why-evals-matter) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#why-evals-matter) - [Types of Evals](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#types-of-evals) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#types-of-evals) - [Unit Tests for LLMs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#unit-tests-for-llms) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#unit-tests-for-llms)

Intermediate - [LLM-as-Judge](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#llm-as-judge) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#llm-as-judge) - [RAG Evaluation Metrics](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#rag-evaluation-metrics) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#rag-evaluation-metrics) - [Building an Eval Dataset](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#building-an-eval-dataset) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#building-an-eval-dataset)

Advanced - [RAGAS Framework](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#ragas-framework) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#ragas-framework) - [Regression Testing](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#regression-testing) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#regression-testing) - [Human Evaluation](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#human-evaluation) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#human-evaluation) - [Eval-Driven Development](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#eval-driven-development) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#eval-driven-development)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/#further-reading)

Why Evals Matter

LLM outputs are probabilistic. Unlike traditional software, there's no assertion that always passes — you need a measurement strategy.

Traditional test:

```
assert sum([1, 2, 3]) == 6    # deterministic – always passes or always fails
```

LLM "test":

```
response = llm("Summarize this doc")
# How do you know if it's good? It changes every run.
# You need evals, not assertions.
```

What evals tell you: - Is this version better than the last? (regression testing) - Does it work on edge cases? (coverage testing) - Where does it fail? (debugging) - Is it worth the cost? (ROI measurement)

Types of Evals

Type	Method	Speed	Cost	Accuracy
Exact match	<code>output == expected</code>	Instant	Free	High (brittle)
Contains check	keyword in output	Instant	Free	Medium
LLM-as-judge	Another LLM scores output	Seconds	Low	Good
Human eval	Humans rate outputs	Slow	High	Best
Model-based metrics	ROUGE, BERTScore	Fast	Free	Medium

Use a mix: cheap automated evals for CI/CD, LLM-as-judge for quality, human eval for calibration.

Unit Tests for LLMs

Start with deterministic checks — fast, free, and catch obvious regressions.

```
import anthropic
import pytest

client = anthropic.Anthropic()

def ask(question: str, system: str = "") -> str:
    resp = client.messages.create(
        model="claude-haiku-4-5-20251001",
        max_tokens=256,
        extra_body={"temperature": 0}, # the Python SDK
        system=system,
        messages=[{"role": "user", "content": question}]
    )
    return resp.content[0].text
```

```

    )
    return next(b.text for b in resp.content if b.type ==
"text").strip()

# — Exact / contains checks

```

```

class TestClassifier:
    SYSTEM = "Classify the SQL query type. Reply with one
word: SELECT, INSERT, UPDATE, DELETE, DDL."

    def test_select(self):
        assert "SELECT" in ask("SELECT * FROM orders",
self.SYSTEM).upper()

    def test_ddl(self):
        assert "DDL" in ask("CREATE TABLE foo (id INT)",
self.SYSTEM).upper()

    def test_insert(self):
        assert "INSERT" in ask("INSERT INTO logs VALUES (1,
NOW())", self.SYSTEM).upper()

class TestExtraction:
    SYSTEM = "Extract the table name from the SQL. Return
only the table name, nothing else."

    def test_simple_select(self):
        result = ask("SELECT * FROM orders WHERE id = 1",
self.SYSTEM)
        assert result.lower() == "orders"

    def test_aliased_table(self):
        result = ask("SELECT o.id FROM orders o JOIN
customers c ON o.cid = c.id", self.SYSTEM)
        assert "orders" in result.lower()

# — Format checks

```

```

import json

def test_json_output():

```

```

    system = "Return ONLY valid JSON with keys: table_name,
row_count."
    result = ask("orders table has 50000 rows", system)
    data = json.loads(result) # will raise if not valid
JSON
    assert "table_name" in data
    assert "row_count" in data
    assert data["table_name"] == "orders"
    assert data["row_count"] == 50000

# — Boundary checks

```

```

def test_refuses_to_hallucinate():
    system = "Answer questions about our data. If you don't
know, say UNKNOWN."
    result = ask("What is the revenue for last Tuesday?",
system)
    assert "UNKNOWN" in result.upper() or "don't have" in
result.lower()

def test_empty_input():
    result = ask("", "Summarize the pipeline")
    assert len(result) > 0 # shouldn't crash

```

LLM-as-Judge

Use a separate LLM call to evaluate quality — more flexible than exact match, cheaper than humans.

```

import anthropic
from dataclasses import dataclass

client = anthropic.Anthropic()
JUDGE_MODEL = "claude-sonnet-5-5" # use a capable model as
judge
# Note: the newest models (Opus 5.5, Sonnet 5.5) reject
non-default temperature/top_p with a 400.
# For consistent judging, rely on a fixed rubric +
structured JSON output instead.

```

```

@dataclass
class EvalResult:
    score: float # 0.0 to 1.0
    reasoning: str
    passed: bool

def judge_faithfulness(question: str, context: str, answer:
str) -> EvalResult:
    """Does the answer only use information from the
context?"""
    prompt = f"""You are evaluating an AI assistant's
answer for faithfulness to the provided context.

Question: {question}

Context:
{context}

Answer:
{answer}

Evaluate whether EVERY claim in the answer is supported by
the context.
- Score 1.0: every claim is explicitly supported
- Score 0.7: mostly supported, minor extrapolations
- Score 0.3: some claims not in context
- Score 0.0: answer contradicts or ignores context

Respond with JSON:
{{"score": <0.0-1.0>, "reasoning": "<one sentence>",
"unsupported_claims": ["<list any claims not in
context>"]}}
"""
    response = client.messages.create(
        model=JUDGE_MODEL,
        max_tokens=256,
        messages=[{"role": "user", "content": prompt}]
    )
    import json
    data = json.loads(next(b.text for b in response.content
if b.type == "text"))

```

```

    return EvalResult(
        score=data["score"],
        reasoning=data["reasoning"],
        passed=data["score"] >= 0.7
    )

def judge_relevance(question: str, answer: str) -> EvalResult:
    """Does the answer actually address the question?"""
    prompt = f"""Does this answer address the question?
    Score 0.0-1.0.

    Question: {question}
    Answer: {answer}

    Respond with JSON: {{ "score": <float>, "reasoning": "<one sentence>" }}"""
    response = client.messages.create(
        model=JUDGE_MODEL,
        max_tokens=128,
        messages=[{"role": "user", "content": prompt}]
    )
    import json
    data = json.loads(next(b.text for b in response.content
    if b.type == "text"))
    return EvalResult(score=data["score"],
    reasoning=data["reasoning"], passed=data["score"] >= 0.7)

def judge_completeness(question: str, answer: str,
    expected_points: list[str]) -> EvalResult:
    """Does the answer cover the expected key points?"""
    points_str = "\n".join(f"- {p}" for p in
    expected_points)
    prompt = f"""Check if this answer covers these key
    points.

    Question: {question}
    Expected points:
    {points_str}
    Answer: {answer}

    For each point, state whether it was covered (YES/NO).

```

```

Then give an overall score (covered_count / total_count).
Respond with JSON: {{"score": <float>, "covered":
[true/false, ...], "reasoning": "<one sentence>"}}"""
    response = client.messages.create(
        model=JUDGE_MODEL,
        max_tokens=256,
        messages=[{"role": "user", "content": prompt}]
    )
    import json
    data = json.loads(next(b.text for b in response.content
if b.type == "text"))
    return EvalResult(score=data["score"],
reasoning=data["reasoning"], passed=data["score"] >= 0.7)

# Run evals
context = "The orders table has order_id (VARCHAR PK),
amount (DECIMAL), status (VARCHAR), created_at
(TIMESTAMP)."
question = "What columns does the orders table have?"
answer = "The orders table has order_id, amount, status,
and created_at columns."

print(judge_faithfulness(question, context, answer))
print(judge_relevance(question, answer))
print(judge_completeness(question, answer, ["order_id",
"amount", "status", "created_at"]))

```

Parsing judge output: the judges call `json.loads` on the raw reply, which raises if the model wraps the JSON in a Markdown code fence or adds a sentence around it. In production, use structured outputs or a tool call to force a schema, or strip fences before parsing, and count unparseable replies as judge failures instead of dropping them.

RAG Evaluation Metrics

The four key metrics for RAG quality:

1. Answer Faithfulness: Does the answer come from the retrieved context?
2. Answer Relevance: Does the answer address the question?



3. Context Precision: Are the retrieved chunks actually useful?
4. Context Recall: Was the relevant information retrieved at all?

```
# Manual RAGAS-style evaluation
def evaluate_rag_response(question: str, answer: str,
                          retrieved_chunks: list[str],
                          reference_answer: str = None) ->
dict:
    context = "\n".join(retrieved_chunks)
    results = {}

    # Faithfulness
    f = judge_faithfulness(question, context, answer)
    results["faithfulness"] = f.score

    # Relevance
    r = judge_relevance(question, answer)
    results["relevance"] = r.score

    # Context precision – how many retrieved chunks were
    useful?
    useful = 0
    for chunk in retrieved_chunks:
        prompt = f"Is this chunk useful for answering
        '{question}'?\n\n{chunk}\n\nAnswer YES or NO."
        resp = client.messages.create(model="claude-haiku-
        4-5-20251001", max_tokens=8,
                                     extra_body=
{"temperature": 0}, messages=[{"role": "user", "content":
prompt}])
        if "YES" in next(b.text for b in resp.content if
b.type == "text").upper():
            useful += 1
    results["context_precision"] = useful /
len(retrieved_chunks) if retrieved_chunks else 0

    # Overall
    results["overall"] = (results["faithfulness"] +
results["relevance"] + results["context_precision"]) / 3
```

```
return results
```

Building an Eval Dataset

```
# Three sources for eval data:

# 1. Hand-crafted: golden QA pairs written by domain
experts
golden_set = [
    {
        "question": "What columns does the orders table
have?",
        "expected_answer": "order_id, customer_id, amount,
status, created_at",
        "expected_keywords": ["order_id", "amount",
"status"],
        "source_doc": "data-dictionary/orders.md"
    },
    {
        "question": "What is the SLA for the gold layer?",
        "expected_answer": "Data available by 6am UTC
daily",
        "expected_keywords": ["6am", "UTC", "daily"],
        "source_doc": "sla-docs/data-freshness.md"
    }
]

# 2. Mined from logs: real questions users asked
import json
from pathlib import Path

def mine_questions_from_logs(log_path: str, min_count: int
= 3) -> list[str]:
    """Extract frequently-asked questions from application
logs."""
    from collections import Counter
    questions = []
    for line in Path(log_path).read_text().splitlines():
        try:
```

```

        entry = json.loads(line)
        if entry.get("type") == "user_query":
            questions.append(entry["query"])
    except json.JSONDecodeError:
        continue

# Return questions asked at least min_count times
counts = Counter(questions)
return [q for q, c in counts.items() if c >= min_count]

# 3. LLM-generated: use an LLM to generate diverse test
cases
def generate_eval_cases(documents: list[str], n: int = 20)
-> list[dict]:
    context = "\n\n".join(documents[:5])
    response = client.messages.create(
        model="claude-sonnet-5-5",
        max_tokens=2048,
        messages=[{"role": "user", "content": f"""
Generate {n} diverse question-answer pairs for evaluating a
RAG system over these documents.
Include easy, medium, and hard questions.
Include questions that the documents DO NOT answer (to test
"I don't know" responses).
Return as a JSON array: [{"question": "...", "answer":
"...", "answerable": true/false}]}

Documents:
{context}
"""}]
    )
    import json
    return json.loads(next(b.text for b in response.content
if b.type == "text"))

```

RAGAS Framework

Version note: RAGAS 0.2+ renamed the dataset fields to `user_input`, `response`, `retrieved_contexts` and `reference`, and this example uses them. The original names (`question`, `answer`, `contexts`, `ground_truth`) are still converted by `evaluate()` in 0.4, but building an `EvaluationDataset` directly from them silently produces empty records. RAGAS uses an LLM as the judge (OpenAI by default); pass your own LLM wrapper to use another provider. Check the [RAGAS docs](https://docs.ragas.io/) (<https://docs.ragas.io/>) for the version you install, and pin it.

```
# ragas 0.4.3 imports
langchain_community.chat_models.vertexai, which langchain-
community 0.4 removed,
# so an unpinned install fails on `import ragas`. Pin
langchain-community below 0.4 until ragas fixes this.
pip install ragas "langchain-community<0.4"
```

```
from ragas import evaluate
from ragas.metrics import faithfulness, answer_relevancy,
context_precision, context_recall
from datasets import Dataset

# Build eval dataset in RAGAS format
data = {
    "user_input": [
        "What columns does the orders table have?",
        "When does the gold layer update?",
    ],
    "response": [
        "The orders table has order_id, customer_id,
amount, status, and created_at.",
        "The gold layer updates daily, with data available
by 6am UTC.",
    ],
    "retrieved_contexts": [
        ["orders table: order_id (VARCHAR), customer_id
(INT), amount (DECIMAL), status (VARCHAR), created_at
(TIMESTAMP)"],
        ["Gold layer tables are updated daily. SLA: data
available by 6am UTC."],
    ],
    "reference": [
```

```

        "order_id, customer_id, amount, status,
created_at",
        "Daily, available by 6am UTC",
    ]
}

dataset = Dataset.from_dict(data)

results = evaluate(
    dataset=dataset,
    metrics=[faithfulness, answer_relevancy,
context_precision, context_recall],
)

print(results)
# {'faithfulness': 0.95, 'answer_relevancy': 0.88,
'context_precision': 0.92, 'context_recall': 0.85}

results.to_pandas()    # DataFrame for deeper analysis

```

Regression Testing

Run evals on every code/prompt change — like CI/CD for LLM quality.

```

import json
from pathlib import Path
from datetime import datetime, timezone

def run_eval_suite(rag_fn, eval_cases: list[dict],
threshold: float = 0.7) -> dict:
    """Run all eval cases and return a summary."""
    results = []
    for case in eval_cases:
        answer = rag_fn(case["question"])
        chunks = case.get("context_chunks", [])
        metrics = evaluate_rag_response(case["question"],
answer, chunks)
        results.append({
            "question": case["question"],
            "answer": answer,

```

```

        "metrics": metrics,
        "passed": metrics["overall"] >= threshold,
    })

    pass_count = sum(1 for r in results if r["passed"])
    fail_count = len(results) - pass_count

    summary = {
        "timestamp":
datetime.now(timezone.utc).isoformat(),
        "total": len(results),
        "passed": pass_count,
        "failed": fail_count,
        "pass_rate": pass_count / len(results),
        "avg_score": sum(r["metrics"]["overall"] for r in
results) / len(results),
        "details": results,
    }
    return summary

# Compare two versions
def compare_versions(fn_v1, fn_v2, eval_cases: list[dict]):
    r1 = run_eval_suite(fn_v1, eval_cases)
    r2 = run_eval_suite(fn_v2, eval_cases)
    delta = r2["avg_score"] - r1["avg_score"]
    print(f"v1: {r1['avg_score']:.3f} v2:
{r2['avg_score']:.3f} delta: {delta:+.3f}")
    if delta < -0.05:
        print("REGRESSION DETECTED - v2 is significantly
worse than v1")
        return False
    return True

# Save results for trending
def save_eval_run(summary: dict, path: str =
"./eval_results/"):
    Path(path).mkdir(exist_ok=True)
    filename = f"{path}/eval_{summary['timestamp']
[:10]}.json"
    Path(filename).write_text(json.dumps(summary,
indent=2))

```

Human Evaluation

```
# Simple annotation interface (run in a Jupyter notebook or CLI)

def annotate_responses(responses: list[dict]) -> list[dict]:
    """
    responses = [{"question": str, "answer": str,
"context": str}]
    Returns responses with human scores added.
    """
    annotated = []
    for i, r in enumerate(responses):
        print(f"\n{'='*60}")
        print(f"[{i+1}/{len(responses)}] Question:
{r['question']}")
        print(f"\nContext:\n{r['context'][:500]}...")
        print(f"\nAnswer:\n{r['answer']}")
        print()

        while True:
            rating = input("Rate the answer (1=bad, 2=ok,
3=good, s=skip): ").strip()
            if rating in ["1", "2", "3", "s"]:
                break
            print("Invalid input. Enter 1, 2, 3, or s.")

        if rating != "s":
            note = input("Note (optional, press Enter to
skip): ").strip()
            annotated.append({
                **r,
                "human_score": int(rating),
                "human_note": note
            })

        print(f"\nAnnotated {len(annotated)}/{len(responses)}
responses")
    return annotated

# Calibrate LLM-as-judge against human scores
```

```
def calibrate_judge(human_annotations: list[dict],
judge_fn) -> float:
    """How well does the LLM judge correlate with human
    scores?"""
    agreements = []
    for ann in human_annotations:
        judge_result = judge_fn(ann["question"],
ann["answer"])
        # Normalize judge score to 1-3 scale
        judge_scaled = 1 + judge_result.score * 2
        # Within 1 point = agreement
        agreement = abs(judge_scaled - ann["human_score"])
        <= 1
        agreements.append(agreement)
    correlation = sum(agreements) / len(agreements)
    print(f"Judge-human agreement: {correlation:.0%}")
    return correlation
```

Eval-Driven Development

The workflow for improving LLM applications systematically.

1. Baseline: Run eval suite, measure current scores
2. Identify: Find the lowest-scoring cases – these are your bugs
3. Hypothesize: Why is it failing? Wrong retrieval? Bad prompt? Missing docs?
4. Fix: Update prompt, chunking, retrieval, or knowledge base
5. Re-eval: Run eval suite again – did the score improve?
6. Commit: Only ship if overall score didn't regress

```
# Automated eval-driven CI pipeline

def ci_eval_gate(rag_fn, eval_dataset: list[dict],
min_pass_rate: float = 0.80,
min_avg_score: float = 0.75) -> bool:
    """Returns True if the system meets quality gates."""
```

```

results = run_eval_suite(rag_fn, eval_dataset)

print(f"Pass rate: {results['pass_rate']:.0%} (min:
{min_pass_rate:.0%})")
print(f"Avg score: {results['avg_score']:.3f} (min:
{min_avg_score:.3f})")

gate_passed = (
    results["pass_rate"] >= min_pass_rate and
    results["avg_score"] >= min_avg_score
)

# Print failures for debugging
if not gate_passed:
    print("\nFailed cases:")
    for r in results["details"]:
        if not r["passed"]:
            print(f"  Q: {r['question']}")
            print(f"  Score: {r['metrics']
['overall']:.3f}")
            print(f"  A: {r['answer'][:200]}\n")

    return gate_passed

# In CI:
if not ci_eval_gate(my_rag_pipeline, eval_dataset):
    raise SystemExit("Eval gate failed – not deploying")

```

Common Pitfalls

Pitfall	Symptom	Fix
No eval set until something breaks	Every change is a gamble; regressions found by users	Start with 20–50 real examples on day one and grow the set from production failures
Synthetic-only test data	High scores, poor real-world performance	Sample real (anonymized) inputs; add synthetic cases only for coverage gaps

Pitfall	Symptom	Fix
An LLM judge with a vague rubric ("rate 1–10")	Noisy, inflated scores that don't track quality	Specific, binary or low-cardinality criteria; ask for reasoning before the verdict; validate the judge against human labels
Judging with the same model and prompt you're testing	The judge shares the system's blind spots	A different or stronger judge model, and a separate grading prompt
Overfitting prompts to the eval set	Scores go up, production quality doesn't	Keep a held-out test split you don't look at while iterating
Reporting one aggregate number	Improvements in one area hide regressions in another	Slice results by category (question type, customer, difficulty)
Single run per case	Random variation mistaken for improvement	Run several trials; look at variance and confidence intervals
Evaluating only the final answer in RAG or agents	Can't tell whether retrieval or generation failed	Score each stage: retrieval recall, faithfulness, tool-call correctness, final answer
Evals that nobody runs	Stale scores	Run in CI on every prompt, model, or retrieval change; block merges on regressions

Cheat Sheet

Choose the grader

Output type	Best grader
Classification / extraction with known answers	Exact match, F1, accuracy per field
JSON, SQL, code	Code checks: parses, schema validates, SQL runs, tests pass, result matches
Grounded answers (RAG)	Faithfulness / groundedness judge + retrieval recall@k

Output type	Best grader
Open-ended text (summaries, explanations)	LLM judge with a specific rubric, calibrated on human labels
Safety / tone / policy	LLM judge with explicit criteria + human spot checks
Agents	Task success on the end state + trajectory review (tools, steps, cost)

LLM-judge prompt pattern

You are grading an answer about our data platform. 

```
<question>{question}</question>
<reference>{reference_answer}</reference>
<answer>{answer}</answer>
```

Criteria:

1. Correct: agrees with the reference on every fact that matters.
2. Grounded: makes no claims absent from the reference.
3. Complete: covers everything the question asks.

Think through each criterion, then return JSON:

```
{"correct": true/false, "grounded": true/false, "complete": true/false, "reason": "<one sentence>"}
```

Eval dataset record: `id` · `input` · `expected` (answer, facts, or rubric) · `category` · `source` (production, synthetic, bug report) · `difficulty`

Workflow: collect cases → define graders → baseline score → change one thing → re-run → compare per slice → ship only if better without regressions → add new production failures to the set

Interview Questions

Q: Why can't you test LLM applications with ordinary unit tests alone? A: Outputs are non-deterministic and often open-ended, so there's usually no single correct string to assert. Unit tests still help for deterministic parts — parsing, validation, tool code, output

format — but quality has to be *measured* over a representative set of inputs with scoring methods that tolerate valid variation (rubrics, LLM judges, semantic checks), and tracked as a rate over time.

Q: What is LLM-as-a-judge, and how do you make it reliable? A: Using an LLM to grade another model's outputs against criteria. To make it reliable: use specific, preferably binary criteria rather than 1–10 scales; give reference answers where possible; ask for reasoning before the verdict; use structured output; check position bias when comparing two answers (swap the order); and — most importantly — validate the judge against a set of human-labelled examples and measure agreement before trusting it.

Q: How would you evaluate a RAG system? A: Separate the stages. For retrieval: a labelled set of questions with their relevant chunks, measuring recall@k and MRR. For generation: faithfulness (are claims supported by the retrieved context?), answer relevance, and correctness against reference answers. End to end: success rate on realistic questions, including ones that *shouldn't* be answered. Frameworks like RAGAS automate parts of this, and the metrics tell you whether to fix retrieval or the prompt.

Q: How do you build an eval dataset when you don't have one? A: Start small and real: collect 20–50 actual user questions or inputs (from logs, tickets, or stakeholders), write expected answers or grading criteria with domain experts, and cover important categories and known edge cases. Add synthetic cases to fill gaps, and keep adding every production failure as a new case. Version the dataset, and keep a held-out split for final checks.

Q: What is eval-driven development? A: Writing the evals before changing the system — like test-driven development for LLM apps. Define what "good" means for the task, measure the baseline, then iterate on prompts, retrieval, or models, and accept a change only when the eval scores improve without regressions in any important slice. It turns prompt engineering from guesswork into measurable engineering.

Further Reading

- [Anthropic: Define success criteria and build evaluations](https://platform.claude.com/docs/en/test-and-evaluate/develop-tests) (https://platform.claude.com/docs/en/test-and-evaluate/develop-tests)
 - [OpenAI Evals](https://developers.openai.com/api/docs/guides/evals) (https://developers.openai.com/api/docs/guides/evals)
 - [RAGAS](https://docs.ragas.io/) (https://docs.ragas.io/) • [DeepEval](https://deepeval.com/docs/getting-started) (https://deepeval.com/docs/getting-started) • [promptfoo](https://www.promptfoo.dev/docs/intro/) (https://www.promptfoo.dev/docs/intro/) — eval frameworks
 - [Hamel Husain: Your AI product needs evals](https://hamel.dev/blog/posts/evals/) (https://hamel.dev/blog/posts/evals/) — a practical guide to building evals
 - *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena* — Zheng et al., 2023 (judge biases and reliability)
-

Previous: [LangChain & LlamaIndex](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/) · **Next:** [MLflow](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/mlflow/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/mlflow/) · **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 28, 2026