

AI Observability

Monitor, trace, and debug LLM applications in production — cost, latency, quality, and errors.

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/) · [Evals](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/)

Related: [LangChain & LlamaIndex](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/) · [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: When a traditional API misbehaves, logs of inputs, outputs, and error codes are usually enough to diagnose it. LLM systems are harder to operate: outputs are probabilistic, cost varies with token usage, latency is variable, and quality can degrade without producing any error.

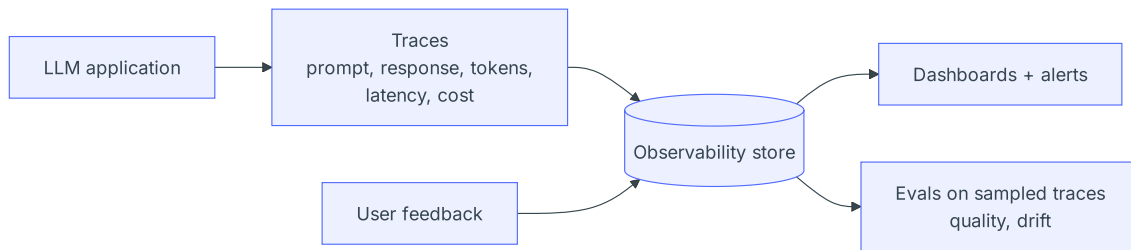
Solution: AI observability is the practice of **systematically recording what an LLM system does** — every call, its inputs and outputs, token usage, latency, and quality scores — so failures can be debugged, regressions detected, and cost and speed optimized.

Traditional API monitoring:

- Response time
 - Error rate
 - HTTP status codes
- text

AI observability adds:

- Which model was used
- Input + output tokens
- Full prompt + completion
- Quality score
- Hallucination detection
- Retrieval trace (for RAG)
- Per-user cost



On this page

Basic - [What to Monitor](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#what-to-monitor) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#what-to-monitor) - [Manual Logging](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#manual-logging) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#manual-logging) - [Cost Tracking](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#cost-tracking) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#cost-tracking)

Intermediate - [LangSmith](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#langsmith) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#langsmith) - [Langfuse](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#langfuse) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#langfuse) - [OpenTelemetry for LLMs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#opentelemetry-for-llms) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#opentelemetry-for-llms)

Advanced - [RAG Tracing](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#rag-tracing) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#rag-tracing) - [Drift Detection](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#drift-detection) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#drift-detection) - [Production Alert Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#production-alert-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/#production-alert-patterns)

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

What to Monitor

The four signals for LLM observability:



1. Latency

- Time to first token (TTFT) – how long until streaming starts
- Total completion time
- P50, P95, P99 – watch P99 for SLA breaches
- Latency by model, by prompt length, by user

2. Cost

- Input tokens, output tokens, total cost per call
- Cost per user, per feature, per day
- Cost trends – catching prompt bloat early

3. Quality

- LLM-as-judge scores on sample of traffic
- User feedback (thumbs up/down)
- Hallucination rate
- Answer relevance, faithfulness (for RAG)

4. Errors

- Rate limit errors (429)
- Context length exceeded
- Timeout / connection errors
- Unexpected empty or truncated responses
- JSON parse failures (for structured output)

Manual Logging

The minimum viable observability — log every LLM call to a structured store.

```
import time
import uuid
import logging
import json
from dataclasses import dataclass, asdict
from datetime import datetime, timezone
from pathlib import Path
import anthropic

client = anthropic.Anthropic()

@dataclass
class LLMCallLog:
    call_id: str
    timestamp: str
    model: str
    feature: str # which feature/pipeline
    made this call
```

```

    user_id:          str
    input_tokens:     int
    output_tokens:    int
    latency_ms:       float
    cost_usd:         float
    success:          bool
    error:            str
    stop_reason:      str
    prompt_preview:   str          # first 200 chars of
prompt
    output_preview:   str          # first 200 chars of
output

# Prices change often, so load them from config rather than
hardcoding them.
# pricing.json maps model ID -> USD per 1M tokens: {"
<model-id>": {"input": <usd>, "output": <usd>}, ...}
# Current prices:
https://platform.claude.com/docs/en/about-claude/pricing
and https://developers.openai.com/api/docs/pricing
COST_PER_1M = json.loads(Path("pricing.json").read_text())

def compute_cost(model: str, input_tokens: int,
output_tokens: int) -> float:
    if model not in COST_PER_1M:
        return 0.0
    p = COST_PER_1M[model]
    return (input_tokens * p["input"] + output_tokens *
p["output"]) / 1_000_000

def tracked_call(feature: str, user_id: str = "system",
**kwargs) -> str:
    """Wrapper around the Anthropic client that logs every
call."""
    call_id = str(uuid.uuid4())[:8]
    start = time.perf_counter()

    prompt_preview = str(kwargs.get("messages", ""))[:200]

    try:
        response = client.messages.create(**kwargs)
        latency = (time.perf_counter() - start) * 1000

```

```

        output = next(b.text for b in response.content if
b.type == "text") if response.content else ""

        log = LLMCallLog(
            call_id      = call_id,
            timestamp    =
datetime.now(timezone.utc).isoformat(),
            model        = response.model,
            feature      = feature,
            user_id      = user_id,
            input_tokens = response.usage.input_tokens,
            output_tokens = response.usage.output_tokens,
            latency_ms   = latency,
            cost_usd     = compute_cost(response.model,

response.usage.input_tokens,

response.usage.output_tokens),
            success      = True,
            error        = "",
            stop_reason  = response.stop_reason,
            prompt_preview = prompt_preview,
            output_preview = output[:200],
        )
        _save_log(log)
        return output

    except Exception as e:
        latency = (time.perf_counter() - start) * 1000
        log = LLMCallLog(
            call_id=call_id,
timestamp=datetime.now(timezone.utc).isoformat(),
            model=kwargs.get("model", "unknown"),
feature=feature,
            user_id=user_id, input_tokens=0,
output_tokens=0,
            latency_ms=latency, cost_usd=0.0,
success=False,
            error=str(e), stop_reason="error",
            prompt_preview=prompt_preview,
output_preview=""
        )

```

```

        _save_log(log)
        raise

def _save_log(log: LLMCallLog):
    # Write to JSONL file (append)
    with open("llm_calls.jsonl", "a") as f:
        f.write(json.dumps(asdict(log)) + "\n")

# Usage
result = tracked_call(
    feature="pipeline-debugger",
    user_id="alice",
    model="claude-haiku-4-5-20251001",
    max_tokens=512,
    messages=[{"role": "user", "content": "What is the
medallion architecture?"}]
)

```

Cost Tracking

```

import pandas as pd
from pathlib import Path

def analyze_costs(log_file: str = "llm_calls.jsonl") -> dict:
    """Read logs and compute cost breakdown."""
    rows = [json.loads(l) for l in
Path(log_file).read_text().splitlines() if l]
    df = pd.DataFrame(rows)
    df["date"] = pd.to_datetime(df["timestamp"]).dt.date

    return {
        "total_cost_usd":      df["cost_usd"].sum(),
        "cost_by_model":      df.groupby("model")
["cost_usd"].sum().to_dict(),
        "cost_by_feature":    df.groupby("feature")
["cost_usd"].sum().to_dict(),
        "cost_by_day":        df.groupby("date")
["cost_usd"].sum().to_dict(),
        "avg_input_tokens":   df["input_tokens"].mean(),
    }

```

```

        "avg_output_tokens": df["output_tokens"].mean(),
        "p95_latency_ms":
df["latency_ms"].quantile(0.95),
        "error_rate":      (~df["success"]).mean(),
        "calls_today":      df[df["date"] ==
pd.Timestamp.today().date()].shape[0],
    }

report = analyze_costs()
print(f"Total cost: ${report['total_cost_usd']:.4f}")
print(f"By model: {report['cost_by_model']}")
print(f"P95 latency: {report['p95_latency_ms']:.0f}ms")
print(f"Error rate: {report['error_rate']:.1%}")

```

LangSmith

Hosted tracing and evaluation platform from the LangChain team. It traces LangChain and LangGraph apps automatically, and other code through the `@traceable` decorator.

```

import os
# Set these in the environment (or a secrets manager), not
in code.
# The older LANGCHAIN_TRACING_V2 / LANGCHAIN_API_KEY /
LANGCHAIN_PROJECT names still work.
os.environ["LANGSMITH_TRACING"] = "true"
os.environ["LANGSMITH_API_KEY"] = "<your-api-key>"
os.environ["LANGSMITH_PROJECT"] = "my-de-app"
# os.environ["LANGSMITH_ENDPOINT"] =
"https://eu.api.smith.langchain.com" # EU region or self-
hosted only

# All LangChain calls are now traced automatically
from langchain_anthropic import ChatAnthropic
from langchain_core.prompts import ChatPromptTemplate

llm = ChatAnthropic(model="claude-sonnet-5-5")
prompt = ChatPromptTemplate.from_template("Answer:
{question}")
chain = prompt | llm

```

```
# This call appears in the LangSmith UI with full trace
result = chain.invoke({"question": "What is Kafka?"})
# Visit https://smith.langchain.com → project "my-de-app"
to see the trace
```

```
# Manual tracing (non-LangChain code)
from langsmith import traceable, Client

client = Client()

@traceable(name="rag-pipeline", project_name="my-de-app")
def rag_answer(question: str) -> str:
    chunks = retrieve(question)
    context = "\n".join(c["text"] for c in chunks)
    response = anthropic_client.messages.create(
        model="claude-sonnet-5-5",
        max_tokens=512,
        messages=[{"role": "user", "content": f"Context:
{context}\nQ: {question}"}]
    )
    return next(b.text for b in response.content if b.type
== "text")

# This creates a trace with nested spans for retrieve +
generate
answer = rag_answer("What is the orders table schema?")
```

```
# Add human feedback to a traced run
from langsmith import Client

client = Client()

def on_user_feedback(run_id: str, score: int, comment: str
= ""):
    """Call this when a user gives thumbs up/down."""
    client.create_feedback(
        run_id=run_id,
        key="user_rating",
        score=score,          # 1=positive, 0=negative
```

```
        comment=comment,  
    )
```

Langfuse

Open-source LLM observability, available as a hosted service or self-hosted.

SDK version note: these examples use the Langfuse Python SDK v4, which is built on OpenTelemetry (`from langfuse import observe, get_client`). The v2 API (`langfuse.decorators`, `langfuse.trace()`) is different and remains available with `pip install "langfuse<3"` . Pin the major version you use and follow the matching docs.

```
pip install langfuse  
# Self-host: see the deployment docs at langfuse.com/docs
```

```
from langfuse import Langfuse, observe, get_client,  
propagate_attributes  
  
# Keys and host can also come from LANGFUSE_PUBLIC_KEY,  
LANGFUSE_SECRET_KEY, LANGFUSE_BASE_URL  
langfuse = Langfuse(  
    public_key = "pk-lf-...",  
    secret_key = "sk-lf-...",  
    base_url   = "https://cloud.langfuse.com",    # or your  
self-hosted URL  
)
```

```
# — Manual tracing
```

```
# propagate_attributes sets user, session and tags on every  
observation inside the block  
with propagate_attributes(user_id="alice",  
session_id="session-123", tags=["production", "rag"]):  
    with langfuse.start_as_current_observation(  
        name="rag-pipeline", as_type="span", input=  
{"question": question}  
    ) as root:
```

```
    # Retrieval
```

```

        with langfuse.start_as_current_observation(
            name="retrieval", as_type="retriever", input=
{"query": question}
        ) as retrieval:
            chunks = retrieve(question)
            retrieval.update(output={"chunks": len(chunks),
"top_score": chunks[0]["score"]})

# Generation
with langfuse.start_as_current_observation(
    name="answer-generation", as_type="generation",
    model="claude-sonnet-5-5",
    input={"question": question, "context_chunks":
len(chunks)},
) as generation:
    answer = generate(question, chunks)
    generation.update(output={"answer": answer},
        usage_details={"input": 450,
"output": 120})

    root.update(output={"answer": answer})
    trace_id = langfuse.get_current_trace_id()

langfuse.flush() # short-lived scripts must flush before
exit

# — Decorator-based (cleaner)


---


@observe(name="retrieval")
def retrieve_with_trace(question: str) -> list:
    return retrieve(question)

@observe(name="generation", as_type="generation")
def generate_with_trace(question: str, chunks: list) ->
str:
    get_client().update_current_generation(model="claude-
sonnet-5-5")
    return generate(question, chunks)

@observe(name="rag-pipeline")
def rag_pipeline(question: str) -> str:
    with propagate_attributes(user_id="alice", tags=

```

```
["rag"]):
    chunks = retrieve_with_trace(question)
    return generate_with_trace(question, chunks)
```

```
# Add scores for quality evaluation
langfuse.create_score(
    trace_id = trace_id,
    name      = "faithfulness",
    value     = 0.92,
    comment  = "All claims supported by context",
)
langfuse.create_score(trace_id=trace_id,
name="user_rating", value=1)    # thumbs up

# Query traces via the SDK
from datetime import datetime, timezone
traces = langfuse.api.trace.list(
    tags          = ["production"],
    from_timestamp = datetime(2026, 9, 1,
tzinfo=timezone.utc),
    limit         = 100,
)
```

OpenTelemetry for LLMs

Vendor-neutral tracing standard. Emit spans that work with Jaeger, Grafana Tempo, Datadog, etc.

```
pip install opentelemetry-api opentelemetry-sdk
opentelemetry-exporter-otlp
```

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import
BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.http.trace_exporter
import OTLPSpanExporter
from opentelemetry.trace import SpanKind
import anthropic
```

```

# Set up tracing
provider = TracerProvider()
provider.add_span_processor(

BatchSpanProcessor(OTLPSpanExporter(endpoint="http://localhost:4318/v1/traces"))
)
trace.set_tracer_provider(provider)
tracer = trace.get_tracer("llm-app")

client = anthropic.Anthropic()

def traced_llm_call(prompt: str, model: str = "claude-
haiku-4-5-20251001") -> str:
    # Attribute names follow the OpenTelemetry GenAI
    semantic conventions (still in development)
    with tracer.start_as_current_span(f"chat {model}",
kind=SpanKind.CLIENT) as span:
        span.set_attribute("gen_ai.operation.name", "chat")
        span.set_attribute("gen_ai.provider.name",
"anthropic")
        span.set_attribute("gen_ai.request.model", model)
        span.set_attribute("gen_ai.request.max_tokens",
512)

        response = client.messages.create(
            model=model, max_tokens=512,
            messages=[{"role": "user", "content": prompt}]
        )

        span.set_attribute("gen_ai.response.model",
response.model)
        span.set_attribute("gen_ai.usage.input_tokens",
response.usage.input_tokens)
        span.set_attribute("gen_ai.usage.output_tokens",
response.usage.output_tokens)

        span.set_attribute("gen_ai.response.finish_reasons",
[response.stop_reason])

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

```

RAG Tracing

Trace each stage of the RAG pipeline to identify where quality degrades.

```
from langfuse import observe, get_client,
propagate_attributes

langfuse = get_client()

# Each decorated function becomes a nested span inside the
calling function's trace
@observe(name="query-analysis")
def analyze(question: str) -> str:
    return classify_query(question)          # factual /
conversational / analytical

@observe(name="retrieval", as_type="retriever")
def retrieve_traced(question: str) -> list[dict]:
    chunks = retrieve(question, k=5)
    langfuse.update_current_span(
        output={
            "chunks_retrieved": len(chunks),
            "top_score": chunks[0]["score"] if chunks else
0,
            "avg_score": sum(c["score"] for c in chunks) /
max(len(chunks), 1),
        }
    )
    return chunks

@observe(name="reranking")
def rerank_traced(question: str, chunks: list[dict]) ->
list[dict]:
    return rerank(question, chunks, top_n=3)

@observe(name="generation", as_type="generation")
def generate_traced(question: str, chunks: list[dict]) ->
str:
    return generate(question, chunks)

@observe(name="rag-full-pipeline")
```

```
def rag_pipeline(question: str, user_id: str) -> dict:
    with propagate_attributes(user_id=user_id):
        query_type = analyze(question)
        chunks = retrieve_traced(question)
        reranked = rerank_traced(question, chunks)
        answer = generate_traced(question, reranked)

        # Quality check, attached to the trace as a score
        faithfulness = judge_faithfulness(question,
                                           "\n".join(c["text"]
                                                    for c in reranked),
                                           answer)

        langfuse.score_current_trace(name="faithfulness",
                                     value=faithfulness.score)

    return {"answer": answer, "sources": [c["source"] for c
in reranked]}
```

Drift Detection

Quality degrades silently over time — prompts become stale, data distributions shift, model versions change.

```
import pandas as pd
from scipy import stats

class QualityDriftMonitor:
    def __init__(self, baseline_window: int = 7,
alert_window: int = 1):
        self.baseline_window = baseline_window # days of
"normal" data
        self.alert_window = alert_window # days to
compare against baseline

    def detect_drift(self, scores: list[dict]) -> dict:
        """
        scores: [{"date": str, "faithfulness": float,
"relevance": float}]
        """
        df = pd.DataFrame(scores)
```

```

df["date"] = pd.to_datetime(df["date"])
df = df.sort_values("date")

cutoff = df["date"].max() -
pd.Timedelta(days=self.alert_window)
baseline_df = df[(df["date"] <= cutoff) &
(df["date"] > cutoff -
pd.Timedelta(days=self.baseline_window))]
recent_df = df[df["date"] > cutoff]

alerts = []
for metric in ["faithfulness", "relevance"]:
    if metric not in df.columns:
        continue
    baseline_vals =
baseline_df[metric].dropna().values
    recent_vals =
recent_df[metric].dropna().values
    if len(baseline_vals) < 10 or len(recent_vals)
< 3:
        continue

    # KS test for distribution shift
    stat, p_value = stats.ks_2samp(baseline_vals,
recent_vals)
    baseline_mean = baseline_vals.mean()
    recent_mean = recent_vals.mean()
    pct_change = (recent_mean - baseline_mean) /
baseline_mean * 100

    if p_value < 0.05 and pct_change < -5:
        alerts.append({
            "metric": metric,
            "baseline_mean":
round(float(baseline_mean), 3),
            "recent_mean":
round(float(recent_mean), 3),
            "pct_change":
round(float(pct_change), 1),
            "p_value": round(float(p_value),
4),
        })

```

```
        return {"alerts": alerts, "drift_detected":  
len(alerts) > 0}
```

Production Alert Patterns

```
# Alert rules as code

ALERT_RULES = [  
    {  
        "name":      "high_error_rate",  
        "condition": lambda metrics: metrics["error_rate"]  
> 0.05,  
        "message":   "LLM error rate exceeds 5% – check  
rate limits and API status",  
        "severity":  "critical",  
    },  
    {  
        "name":      "cost_spike",  
        "condition": lambda metrics:  
metrics["daily_cost_usd"] > metrics["avg_daily_cost_7d"] *  
2,  
        "message":   "Daily LLM cost is 2x the 7-day  
average – check for prompt bloat or traffic spike",  
        "severity":  "warning",  
    },  
    {  
        "name":      "latency_degradation",  
        "condition": lambda metrics:  
metrics["p95_latency_ms"] > 10_000,  
        "message":   "P95 latency exceeds 10s – LLM API may  
be degraded",  
        "severity":  "warning",  
    },  
    {  
        "name":      "quality_drop",  
        "condition": lambda metrics:  
metrics["avg_faithfulness_1h"] < 0.7,  
        "message":   "Average faithfulness score dropped  
below 0.7 in the last hour",
```

```

        "severity": "critical",
    },
    {
        "name": "empty_responses",
        "condition": lambda metrics:
metrics["empty_response_rate"] > 0.01,
        "message": "More than 1% of responses are empty –
check max_tokens settings",
        "severity": "warning",
    },
]

def check_alerts(metrics: dict) -> list[dict]:
    return [
        {"name": rule["name"], "severity":
rule["severity"], "message": rule["message"]}
        for rule in ALERT_RULES
        if rule["condition"](metrics)
    ]

def send_alert(alert: dict, webhook_url: str):
    import requests
    emoji = ":red_circle:" if alert["severity"] ==
"critical" else ":warning:"
    requests.post(webhook_url, json={
        "text": f"{emoji} *{alert['name']}*:
{alert['message']}"
    })

```

Common Pitfalls

Pitfall	Symptom	Fix
Logging only errors	Quality problems are invisible; "no errors" while answers get worse	Trace every call (inputs, outputs, tokens, latency) and sample outputs for quality scoring
No cost attribution	A surprise bill with no idea which feature or customer caused it	Tag every call with feature, user/tenant, prompt version, and model; aggregate cost by tag

Pitfall	Symptom	Fix
Hardcoded price tables	Cost dashboards drift from the invoice	Keep prices in config with a date; reconcile against the provider's usage and cost reports
Averages only	p99 latency spikes and timeouts hidden by a healthy mean	Track p50/p95/p99 latency, time-to-first-token for streaming, and error rate by type
Logging full prompts with PII	Sensitive data copied into a third-party tool	Redact or hash PII before logging; set retention limits; self-host if required
Traces with no link to user feedback	Can't find the answers users disliked	Attach feedback and eval scores to trace IDs
Tracing that blocks the request path	Added latency or failures when the tracing backend is down	Asynchronous, batched exporters; tracing failures must never fail the request
No alerting on quality	A prompt or model change silently degrades answers	Scheduled evals on sampled production traffic with thresholds and alerts
Ignoring <code>stop_reason</code> and refusals	Truncated or refused outputs look like normal successes	Record <code>stop_reason</code> ; alert on spikes in <code>max_tokens</code> or <code>refusal</code>

Cheat Sheet

What to record per LLM call

Field	Why
<code>trace_id</code> , <code>span_id</code> , <code>parent</code>	Reconstruct multi-step chains and agent runs
<code>model</code> , prompt version, parameters	Explain behavior changes; compare versions
<code>input_tokens</code> , <code>output_tokens</code> , cache read/write tokens	Cost and caching efficiency
Latency, time to first token	User experience and SLAs

Field	Why
<code>stop_reason</code> , error type, retries	Reliability
Feature, user/tenant, environment	Cost attribution and debugging
Request ID from the provider	Support tickets with the provider
Feedback and eval scores	Quality over time

Dashboards and alerts

Metric	Alert when
Error rate (4xx / 5xx / timeouts)	Above baseline for 5–10 minutes
p95 latency	Above the SLA
Cost per day / per feature	Above budget or a sudden jump (e.g. 2× the 7-day average)
Tokens per request	Sudden increase (prompt bloat, runaway agents)
Cache hit rate	Drops (a silent cache invalidation)
Quality score on sampled traffic	Below threshold or a significant drop from baseline
<code>refusal</code> / <code>max_tokens</code> stop rate	Spike

Tools: LangSmith (LangChain ecosystem) · Langfuse (open source, self-hostable) · Arize Phoenix (open source, OpenTelemetry) · MLflow Tracing · Datadog / Grafana with the OpenTelemetry GenAI semantic conventions · your own warehouse table of call logs

Interview Questions

Q: What metrics would you monitor for an LLM-powered data assistant in production? A:

Four categories: (1) Infrastructure — latency (P50/P95/P99), error rate, throughput; (2) Cost — input/output tokens per call, cost per feature, daily total; (3) Quality — LLM-as-judge scores (faithfulness, relevance) on a 10% sample, user feedback signals; (4) RAG-specific — retrieval precision, context score, answer groundedness. Alert on error rate >5%, cost 2x baseline, quality score drops, and latency P95 >10s.

Q: How do you detect when an LLM pipeline degrades without users reporting it? A: (1)

Run automated quality evals on a sample of real traffic using LLM-as-judge — score faithfulness and relevance; (2) track score distributions over time and alert on statistically significant drops (KS test); (3) monitor cost-per-call — unexpected increases often mean prompt bloat; (4) log all inputs/outputs and do random manual spot-checks; (5) track thumbs-up/down or implicit signals (follow-up questions often indicate a bad answer).

Further Reading

- [OpenTelemetry semantic conventions for GenAI](https://opentelemetry.io/docs/specs/semconv/gen-ai/) (https://opentelemetry.io/docs/specs/semconv/gen-ai/)
 - [Langfuse documentation](https://langfuse.com/docs) (https://langfuse.com/docs)
 - [LangSmith documentation](https://docs.smith.langchain.com/) (https://docs.smith.langchain.com/)
 - [Arize Phoenix](https://docs.arize.com/phoenix) (https://docs.arize.com/phoenix)
 - [MLflow Tracing](https://mlflow.org/docs/latest/genai/tracing/) (https://mlflow.org/docs/latest/genai/tracing/)
 - [Anthropic Usage and Cost API](https://platform.claude.com/docs/en/manage-claude/usage-cost-api) (https://platform.claude.com/docs/en/manage-claude/usage-cost-api) — reconcile your own cost tracking with billing data
-

Previous: [Fine-Tuning](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/fine-tuning/) • **Next:** [Local LLMs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/local-llms/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 28, 2026