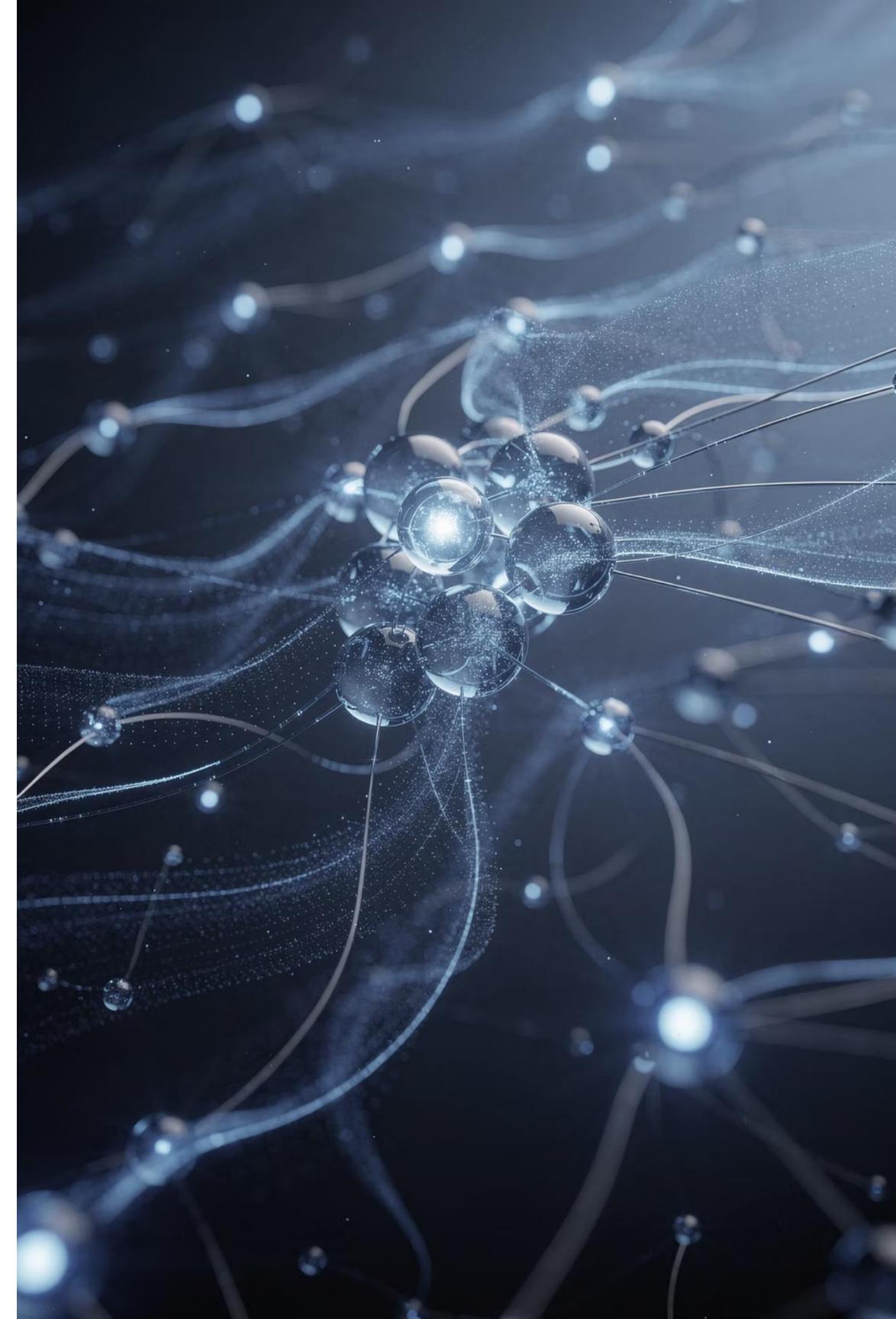


Observability & Evaluation in LLMs and Agentic AI Systems

Amir K. Saeed

Adjunct Faculty, Johns Hopkins University

AI Solutions Engineer, Arize AI



Why Observability Now?

LLM and agent systems differ greatly from traditional software:

- They are **complex, multi-step, and non-deterministic**, involving dozens of internal operations per query.
- Failures are insidious: **hallucinations, hidden tool errors, and misleading partial answers** can appear plausible.
- Without observability, teams resort to "debugging by vibes," an approach that doesn't scale.



Complex Systems



Multi-Step Pipelines



Subtle Failures



Common Pain Points

Engineering teams building LLM systems consistently encounter the same set of challenges. Understanding these pain points is the first step toward addressing them systematically.



Silent Hallucinations

Models generate plausible but factually incorrect information in production, eroding user trust and requiring extensive rework.



Latency Spikes

Unpredictable response times and throughput variations lead to poor user experience and potential customer churn.



Zero Visibility

No clear understanding of which prompts, agents, or configurations actually deliver results, slowing iteration cycles dramatically.



No Ground Truth

Open-ended tasks lack clear success criteria, making it nearly impossible to measure improvement objectively.

Tinkering vs. Engineering

The maturity journey from experimental prototype to production-grade system requires a fundamental shift in approach. This transition marks the difference between hobbyist experimentation and professional engineering practice.

Tinkering

- Prompt tweaks in isolation without context
- Manual spot checks on cherry-picked examples
- Anecdotal feedback from limited testing
- No systematic measurement or tracking
- Hope-driven development cycles

This approach works for demos but fails in production.

Engineering

- Systematic metrics collection and trace analysis
- LLM-as-a-judge evaluation at scale
- Structured experimentation with statistical rigor
- Automated monitoring and alerting
- Data-driven iteration loops

This is where reliability begins.

The Core Thesis

You cannot improve what you cannot see.

This fundamental principle applies universally across software engineering, but it takes on special significance in the world of LLMs and agentic AI. The path to reliable, production-grade systems follows a clear progression.



Observe

Instrument your systems to capture spans, traces, and metrics at every step

Measure

Apply evaluation frameworks and LLM judges to quantify quality across dimensions

Improve

Make data-driven optimizations to prompts, models, and system architecture

Observability + Evaluation = Reliable LLM & Agentic Systems

The LLM Ecosystem at a Glance

Before diving into observability specifics, it's essential to understand the broader ecosystem. The modern LLM stack comprises multiple layers, each serving distinct purposes in the journey from foundation models to production applications.



Each category represents a critical decision point in your architecture. The tools you choose here will shape your observability strategy.

Model Providers & Foundations

Foundation models form the bedrock of the LLM ecosystem. These are the large-scale models trained on massive corpora that provide the core language understanding and generation capabilities.

Key characteristics that differentiate providers:

- Model size and parameter count (affecting capability and cost)
- Modality support (text-only vs. multimodal)
- Reasoning capabilities and instruction-following
- API access patterns and rate limits
- Licensing terms (proprietary vs. open weights)

Your choice of provider impacts everything downstream—latency characteristics, cost structure, and the observability data you'll need to collect.

1 OpenAI

GPT-4, GPT-3.5

2 Anthropic

Claude family

3 Google

Gemini, PaLM

4 Meta

Llama models

5 Mistral

Open models

6 Cohere

Command, Embed

Orchestrators, Fine-Tuning, Repos

Beyond the foundation models themselves, a rich ecosystem of tools enables practical application development. These three categories form the connective tissue of modern LLM applications.

Agentic Orchestrators

Frameworks that enable multi-step reasoning, tool use, and autonomous agent behavior.

- LangChain
- LlamaIndex
- AutoGen
- CrewAI
- Semantic Kernel

Fine-Tuning Libraries

Tools for efficient model customization and adaptation to specific domains.

- PEFT (Parameter-Efficient Fine-Tuning)
- LoRA and QLoRA
- Hugging Face TRL
- Axolotl
- LitGPT

Model Repositories

Platforms for discovering, sharing, and deploying models and datasets.

- Hugging Face Hub
- ModelScope
- Ollama
- Together AI
- Replicate

Observability & Evaluation Tools

This is where our focus lies. The observability and evaluation layer provides the visibility needed to understand, debug, and improve LLM systems at scale. These tools transform black-box systems into instrumentable, measurable platforms.

Observability Platforms

Systems for capturing, visualizing, and analyzing traces, spans, and metrics from LLM applications.

- **Arize OSS Phoenix** – Open-source LLM observability
- OpenTelemetry + OpenInference – Standards-based tracing
- LangSmith – LangChain-native observability
- Weights & Biases – Experiment tracking
- TruLens – LLM app evaluation

Evaluation Frameworks

Tools for systematic assessment of LLM outputs across multiple quality dimensions.

- **Arize Ax** – LLM-as-a-judge experimentation platform
- PromptFoo – Prompt testing and evaluation
- LangChain Evaluators – Built-in evaluation tools
- OpenAI Evals – Evaluation framework
- RAGAS – RAG assessment suite

Throughout this presentation, we'll focus on **Phoenix** for observability and **Ax** for evaluation as concrete examples.

What is Generative AI?

Before we can effectively observe and evaluate these systems, we need a shared mental model of how they actually work. Generative AI refers to models that create new content—text, code, images—based on learned patterns.

1 Input Processing

User provides a prompt or query

2 Pattern Matching

Model identifies relevant learned patterns from training

3 Token Prediction

System predicts next most likely tokens based on probability distribution

4 Output Generation

Tokens are assembled into coherent text, code, or other content

 **Key Insight:** LLMs are fundamentally *next-token prediction engines* trained on massive text corpora. They learn statistical patterns, not facts or logic in the traditional sense.



What Generative AI Is *Not*

Clearing up misconceptions is crucial for setting appropriate expectations and building robust systems. Understanding limitations shapes how we approach observability and evaluation.

1 Not Deterministic

The same prompt can produce different outputs due to sampling and temperature settings. This non-determinism makes traditional testing approaches insufficient.

3 Not Conscious or Self-Aware

Despite human-like language, these are statistical models executing mathematical operations. There's no understanding, consciousness, or genuine reasoning.

2 Not Guaranteed Factual

Models don't "know" facts—they predict likely next tokens based on patterns. They can confidently generate plausible-sounding but completely false information.

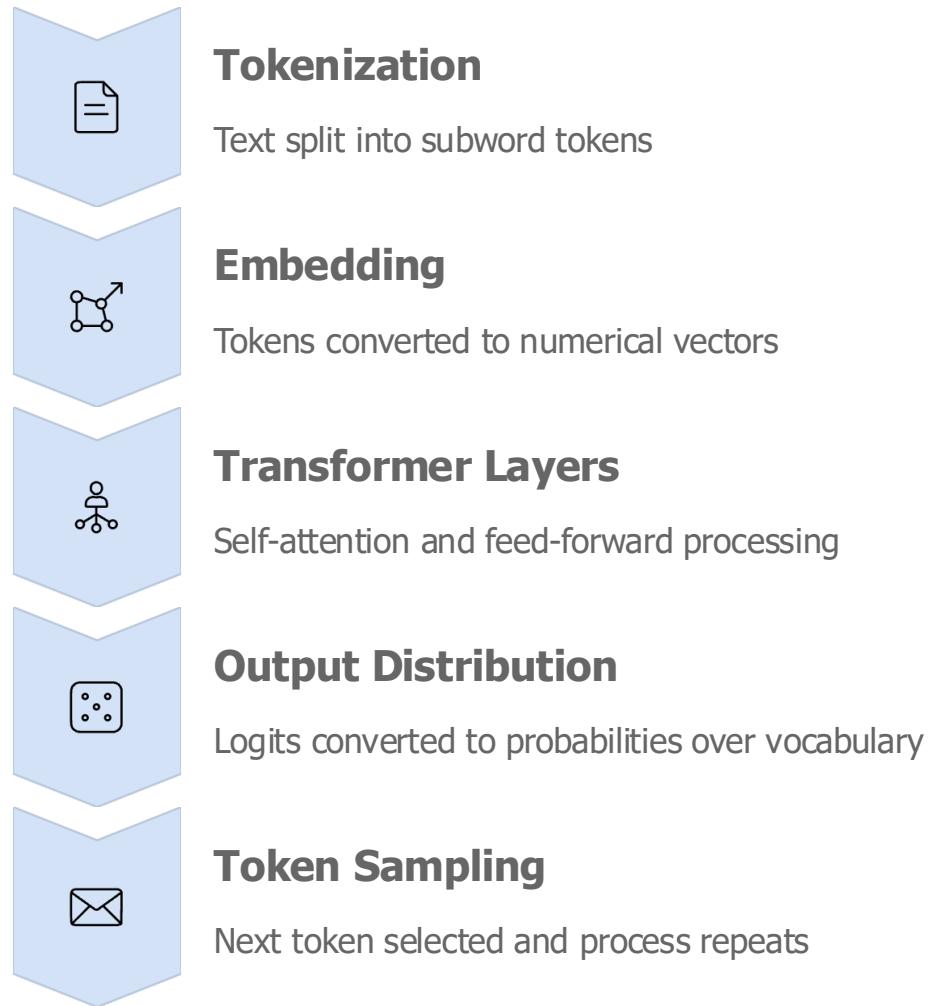
4 Not Always Calibrated

Confidence in output doesn't correlate with correctness. Models sound equally confident whether they're right or hallucinating completely.

Bottom line: LLMs are powerful pattern-completion engines that require careful observation and evaluation because their outputs cannot be trusted by default.

How LLMs Make Predictions

Understanding the prediction pipeline helps contextualize why certain failures occur and what signals we need to observe. Here's the simplified flow from input to output.



Each stage in this pipeline is a potential source of observable signals. Token counts affect costs. Attention patterns influence quality. Sampling parameters control creativity versus consistency.

For observability: We need to track not just final outputs, but intermediate states—especially token usage, latency at each stage, and the parameters that influenced generation.

Examples of LLMs in the Wild

LLMs power an increasingly diverse range of applications. Each use case brings unique observability requirements and evaluation challenges.



Code Generation

GitHub Copilot, Cursor, and similar tools assist developers by generating functions, fixing bugs, and explaining code. Evaluation focuses on syntactic correctness, functionality, and security.



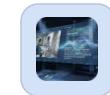
Q&A and Support

Chatbots handle customer inquiries, technical support, and information retrieval. Success metrics include answer relevance, resolution rate, and user satisfaction.



RAG & Summarization

Retrieval-augmented generation combines external knowledge with LLM capabilities for grounded responses. Critical to evaluate: retrieval quality, context usage, and factual accuracy.



Multimodal Applications

Vision-language models caption images, answer visual questions, and understand complex scenes. Evaluation must consider both visual grounding and textual quality.

How We Observed Traditional ML

Before discussing LLM-specific challenges, let's establish a baseline. Traditional machine learning had well-defined approaches to observability that worked for classification, regression, and other supervised tasks.

Standard Metrics

- **Accuracy:** Overall correctness across test set
- **Precision & Recall:** Class-specific performance measures
- **ROC-AUC:** Trade-offs at different thresholds
- **F1 Score:** Harmonic mean of precision and recall
- **Confusion Matrix:** Detailed breakdown of predictions

Explainability & Monitoring

- **Feature Importance:** Which inputs drive predictions
- **SHAP Values:** Contribution of each feature
- **Data Drift:** Changes in input distributions
- **Concept Drift:** Changes in underlying relationships
- **Model Performance Decay:** Tracking accuracy over time

These approaches worked well because traditional ML had clear ground truth labels, closed-form outputs (like class probabilities), and deterministic predictions for a given input.

Why LLMs Are Different

LLMs break many of the assumptions that made traditional ML observability straightforward. These differences demand new approaches to monitoring and evaluation.

1 Open-Ended Outputs

LLMs generate free-form text with infinite possible valid responses. There's rarely a single "correct" answer to compare against, making traditional accuracy metrics meaningless.

2 Non-Deterministic Sampling

Temperature and sampling parameters mean the same prompt produces different outputs each time. Reproducibility requires careful management of random seeds and generation parameters.

3 Context-Dependent Quality

Performance varies dramatically based on context length, instruction clarity, and examples provided. Small prompt changes can cause large output differences.

4 Compounding Errors

Multi-step pipelines amplify issues. A bad retrieval step poisons the generation step. A faulty tool call derails the entire agent. Errors cascade and interact in complex ways.

The challenge: We need observability systems that handle ambiguity, capture intermediate steps, and evaluate quality along multiple subjective dimensions.

Modes of Failure in LLM Systems

Understanding how LLM systems fail is essential for designing effective evaluation strategies. Each failure mode requires different detection approaches and mitigation strategies.

Hallucinations

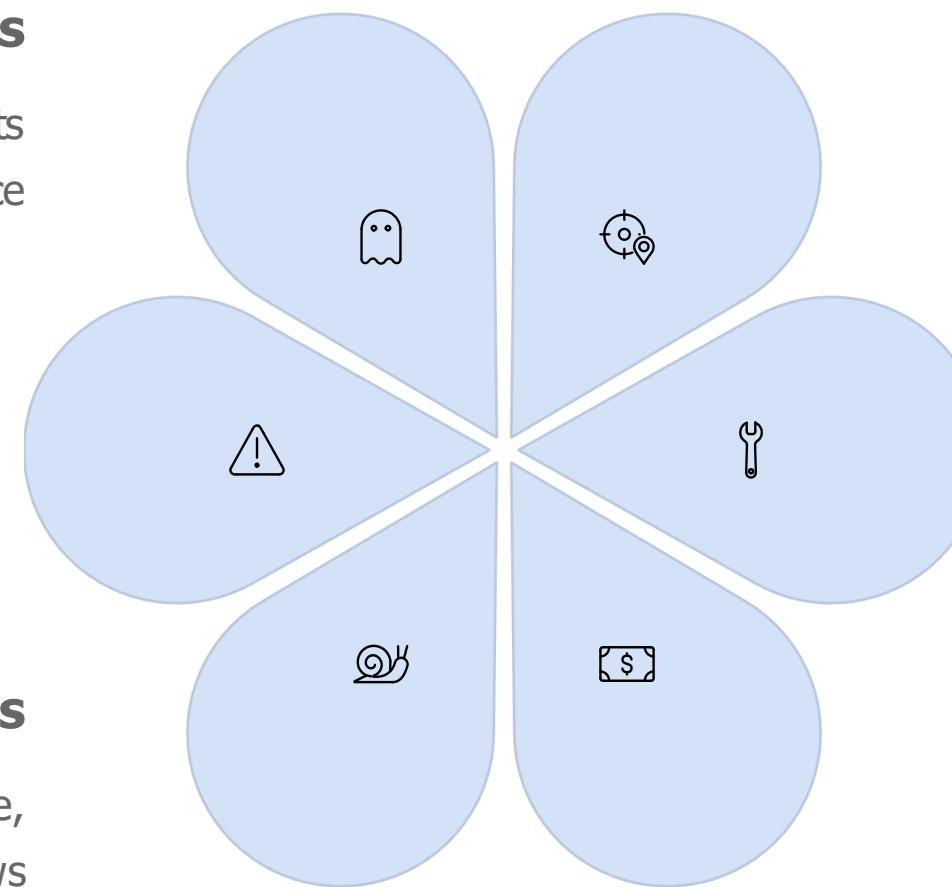
Plausible but factually incorrect statements presented with high confidence

Unsafe Outputs

Toxic, biased, or harmful content that violates safety guidelines

Latency Issues

Response times that exceed user tolerance, especially compounding in multi-step flows



Irrelevant Answers

Responses that fail to address the user's actual question or intent

Tool-Use Errors

Wrong tool selection, malformed parameters, or failure to use available tools

Cost Explosions

Inefficient token usage or excessive API calls leading to unsustainable costs

This taxonomy becomes our evaluation framework. Each failure mode maps to specific judge dimensions and numerical metrics we'll explore next.

Why Observability is Hard for LLMs

The challenges run deeper than just new failure modes. The fundamental nature of LLM systems makes traditional observability approaches insufficient.



No Obvious Ground Truth

For creative, open-ended, or subjective tasks, there's no gold standard to compare outputs against. What constitutes a "good" product description or "helpful" explanation varies by context and user.



Unstructured Text Outputs

Unlike classification models that output clean probabilities, LLMs produce paragraphs of text. Evaluating quality requires semantic understanding, not just string matching or statistical tests.



Multi-Dimensional Quality

A single response must be evaluated across numerous axes simultaneously: correctness, helpfulness, relevance, coherence, tone, safety, and more. No single metric captures overall quality.



Complex Blame Assignment

In multi-agent systems, when something goes wrong, pinpointing the root cause is non-trivial. Was it the retriever? The prompt? The model? The tool? The routing logic?

Two Pillars of Evaluation

Effective LLM evaluation requires combining quantitative metrics with qualitative assessment. Neither alone provides the complete picture.

Numerical Metrics

Objective, automatically collected measurements of system behavior and resource utilization.

- Latency and throughput
- Token counts and costs
- Error rates and retries
- Tool success rates
- Span durations

Strength: Precise, easy to aggregate, great for detecting performance degradation.

Limitation: Can't assess content quality or semantic correctness.

LLM-as-a-Judge

Using LLMs to evaluate LLM outputs along rubric-defined quality dimensions.

- Correctness and grounding
- Relevance and instruction-following
- Helpfulness and clarity
- Safety and toxicity
- Task completion

Strength: Can assess semantic quality at scale without human labeling.

Limitation: Subject to judge model biases and limitations.

The key: Use both pillars together. Numerical metrics catch operational issues; LLM judges catch quality issues. Combined, they provide comprehensive system visibility.

Numerical Metrics Overview

Let's break down the quantitative metrics you should track for any production LLM system. These form the operational heartbeat of your observability practice.

1 Performance Metrics

- Latency (P50, P95, P99)
- Time-to-first-token
- Time-to-last-token
- Throughput (requests/sec)

2 Efficiency Metrics

- Prompt token count
- Completion token count
- Cost per request
- Cost per user session

3 Reliability Metrics

- Error rate by type
- Retry count
- Fallback usage rate
- Timeout frequency

4 Pipeline Health

- Span depth and count
- Individual span durations
- Tool invocation success rate
- Retrieval quality scores

These metrics work together to provide a real-time view of system health. Performance metrics catch slowdowns. Efficiency metrics prevent cost overruns. Reliability metrics surface failures. Pipeline health metrics enable root cause analysis.

Performance & Efficiency Metrics

Let's dive deeper into the two most critical categories for operational excellence: performance and efficiency. These directly impact user experience and operating costs.

Performance: Latency Breakdown

Not all latency is equal. Breaking it down reveals optimization opportunities.

Time-to-First-Token (TTFT): The delay before streaming begins. Critical for perceived responsiveness. Affected by prompt length, model loading, and queue time.

Time-to-Last-Token: Total generation time. Affected by output length, model speed, and any post-processing. Important for batch workloads.

Per-Token Latency: Average time per generated token. Useful for comparing model efficiency and detecting performance degradation.

Efficiency: Token & Cost Management

Prompt Tokens: Input tokens including system prompts, context, and user query. Controllable through context pruning and prompt engineering.

Completion Tokens: Output tokens generated by model. Controllable through max_tokens limits and early stopping.

Cost Attribution: Track costs per user, per feature, per agent to identify expensive operations and optimize ROI.

350ms

Median TTFT

Target for responsive feel

2.5s

P95 Total

Acceptable max latency

1.2K

Avg Prompt

Typical input tokens

\$0.08

Cost/Request

Example blended rate

Reliability & Pipeline Metrics

Beyond speed and cost, you need visibility into *what's actually happening* inside your system. Reliability and pipeline metrics reveal the internal health of complex multi-step workflows.



Error & Retry Tracking

Model API Failures: Rate limits, timeouts, service errors from provider APIs. Track by error code to identify patterns.

Retry Behavior: How often does your system need to retry requests? High retry rates indicate capacity issues or aggressive rate limiting.

Fallback Usage: When primary models fail, how often do fallbacks engage? This reveals dependency on backup systems.



Tool Success Metrics

Tool Call Success Rate: Percentage of tool invocations that execute without errors. Low success indicates prompt issues or tool implementation bugs.

Parameter Validation: How often do agents provide malformed tool parameters? This reveals prompt engineering opportunities.

Tool Latency: Time spent in external tool calls. Slow tools bottleneck entire agent workflows.



Pipeline Analysis

Agent Steps per Query: How many internal steps does a typical request trigger? Excessive depth suggests over-complicated routing.

Span Duration Distribution: Which pipeline stages consume the most time? This identifies bottlenecks for optimization.

Trace Depth: How deeply nested are your operations? Deep traces are harder to debug and optimize.

Enter LLM-as-a-Judge

Numerical metrics tell you *how* your system is performing, but not *what quality* it's producing. LLM-as-a-judge bridges this gap by using AI to evaluate AI outputs at scale.

The core insight is simple but powerful: **LLMs are good at evaluating text along specific dimensions when given clear rubrics.** Instead of expensive, slow human evaluation, we can automate quality assessment using specialized judge models.

This approach gained traction with research like MT-Bench and deployments like Chatbot Arena, which demonstrated that LLM judges correlate well with human judgments across many dimensions.

Why LLM-as-a-Judge Works

- **Scalability:** Evaluate thousands of outputs in minutes, not weeks
- **Consistency:** Same rubric applied uniformly across all evaluations
- **Cost:** 10-100x cheaper than human annotation at scale
- **Iteration Speed:** Test prompt changes immediately with quantitative feedback

1 Prepare

Collect prompt, output, and context

2 Judge

Submit to evaluator LLM with rubric

3 Score

Receive structured rating + explanation

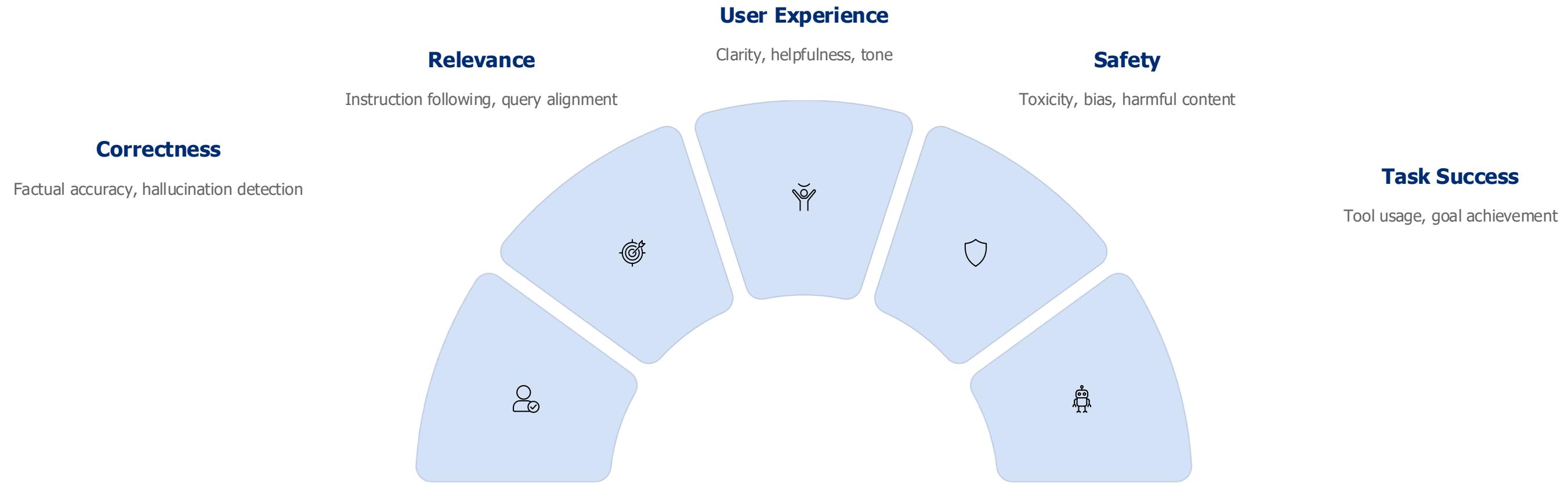
4 Aggregate

Track scores over time and by dimension

- Key limitation:** Judge models have their own biases and blindspots. Always validate with human spot-checks and be aware of issues like position bias or length bias in judge evaluations.

Categories of Judge Dimensions

LLM judges can evaluate outputs along numerous dimensions. Organizing these into categories helps structure your evaluation strategy and ensures comprehensive quality coverage.



Selecting Dimensions for Your Use Case

Not every application needs every dimension. A code generation tool prioritizes correctness and functionality over tone. A customer service chatbot prioritizes helpfulness, safety, and sentiment. A research assistant prioritizes grounding and citation accuracy.

Best practice: Start with 3-5 core dimensions most critical to your users, then expand as you mature your evaluation practice. Too many dimensions dilute focus and increase evaluation costs.

Example Judge Prompts

The power of LLM-as-a-judge lies in well-crafted evaluation prompts. Here are battle-tested templates you can adapt for your own systems.

1 Hallucination Judge

Given the following context and model response, evaluate factual correctness:
Context:
{context}Response: {response}Rate from 1-5 where:
1 = Major hallucinations or contradicts context
3 = Mostly accurate with minor issues
5 = Fully grounded in provided context
Provide rating and brief explanation.

2 Relevance Judge

Given the user query and model response:
Query: {query}Response: {response}
Rate from 1-5 how well the response addresses the query:
1 = Completely off-topic or misses key parts
3 = Partially addresses query, missing some aspects
5 = Fully addresses all parts of the query
Provide rating and brief explanation.

3 Tool Usage Judge

Evaluate the agent's tool usage:
Available tools: {tools}Task: {task}Actions taken: {actions}
Rate from 1-5:
1 = Wrong tools or failed to use needed tools
3 = Correct tools but suboptimal usage
5 = Optimal tool selection and usage
Explain your rating.

Pro tip: Include examples of each rating level in your judge prompts (few-shot evaluation) to improve consistency and calibration. Store prompts in version control alongside your application code.

What is Agency?

Agency refers to the capacity to act independently, make decisions, and influence outcomes based on goals and available information. It encompasses autonomy, intentionality, and the ability to navigate complex environments.

In human psychology, agency involves:

- Goal-directed behavior
- Decision-making under uncertainty
- Adaptation to environmental feedback
- Balancing competing objectives

The Milgram Experiment provides a famous example of how agency can be influenced or constrained. Participants believed they were shocking learners when instructed by authority figures, revealing how perceived authority and instructions shape autonomous action.

This connects to agentic AI: when we give LLMs tool access and autonomous decision-making capabilities, we're creating systems that exhibit agency—they act on our behalf based on instructions and perceived authority.



□ **Why this matters for AI:** Understanding human agency helps us design appropriate constraints, oversight mechanisms, and evaluation criteria for autonomous AI systems.

What is Agentic AI?

Agentic AI systems go beyond simple input-output models. They exhibit behaviors we associate with autonomous agents: planning, tool use, memory, and goal-oriented action.



Task Decomposition

Breaks complex requests into manageable sub-tasks, creating execution plans dynamically



Tool Selection

Decides which tools to invoke based on current context, available capabilities, and task requirements



State & Memory

Maintains working memory of actions taken, results observed, and relevant context across multiple steps



Autonomous Action

Acts independently toward goals, adapting strategy based on feedback and changing conditions

We've had agents in robotics, game AI, and planning systems for decades. **What's new:** LLMs serve as both the planning engine *and* the execution actuator, enabling more flexible and capable agents that can handle natural language, ambiguous instructions, and open-ended tasks.

Why Observability is Critical for Agents

Agentic systems amplify observability challenges. Each autonomous decision point is a potential failure point. Multi-step chains compound errors. Without visibility, debugging becomes nearly impossible.

1 Router Decision

Which branch to take?

2 Retrieval Step

What context to fetch?

3 Generation

What to output?

4 Tool Invocation

Which tool with what params?

5 Result Formatting

How to structure output?

Multi-Point Failure Reality

Each step in an agentic pipeline can fail in unique ways:

- Router makes wrong branching decision
- Retrieval returns irrelevant context
- Generator hallucinates despite good context
- Tool receives malformed parameters
- Formatter corrupts structured output

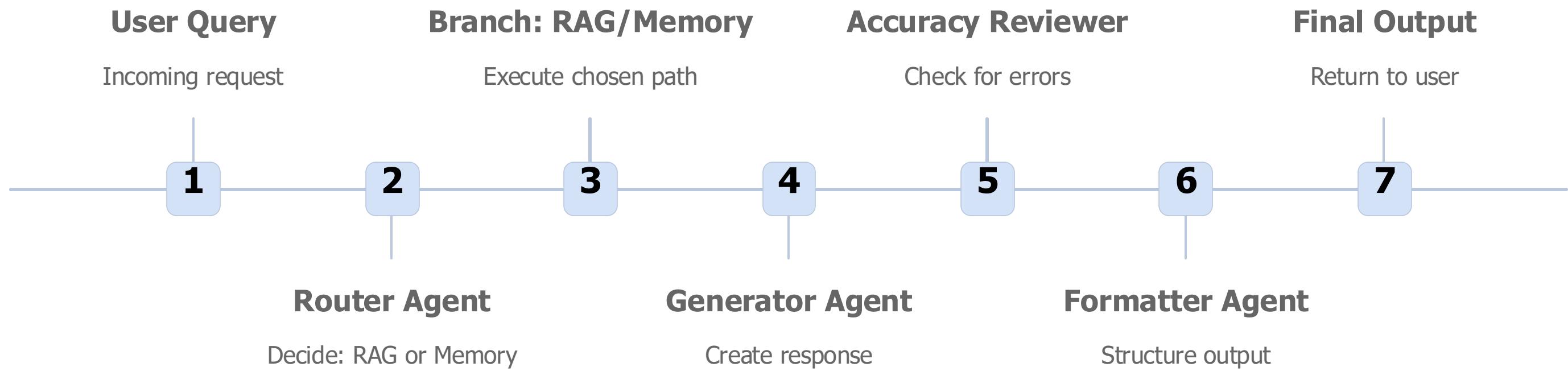
Without span-level observability, you can't answer:

- "Which step caused the bad output?"
- "Is this failure consistent or intermittent?"
- "What input patterns trigger this error?"

The solution: Instrument every agent decision, tool call, and generation with detailed spans. Make the implicit explicit. Turn black boxes into observable, debuggable systems.

Agent Pipeline Overview

Let's walk through a concrete example: a production agentic system that handles user queries with routing, retrieval, generation, review, and formatting. This architecture illustrates key observability challenges and solutions.



This pipeline contains multiple decision points, external calls, and quality gates. Each represents a span we need to observe, evaluate, and optimize.

Agent Roles Explained (Part 1)

Router → RAG/Memory Branch

Router Agent

The router makes a critical first decision: does this query require external knowledge retrieval (RAG) or can it be handled with the model's internal knowledge and conversation memory?

Decision factors:

- Query specificity and domain expertise required
- Temporal aspects (recent events need RAG)
- Proprietary/company-specific information needs
- User's previous context in conversation memory

Observable signals:

- Routing decision and confidence score
- Decision latency
- Correctness of routing choice (evaluated post-hoc)

RAG Branch

When RAG is selected, the system retrieves relevant documents or context from external sources before generation.

RAG steps:

1. Query reformulation for retrieval
2. Vector/keyword search execution
3. Document ranking and filtering
4. Context assembly and compression

Observable signals:

- Retrieval query and results count
- Relevance scores of retrieved docs
- Retrieval latency
- Context window utilization

Memory Branch: If memory is selected instead, the system uses conversation history and internal model knowledge without external retrieval.

Agent Roles Explained (Part 2)

Generator → Reviewer → Formatter

Generator Agent

Produces the main answer using assembled context (from RAG or memory). This is typically the most expensive and latency-sensitive operation.

- Takes user query + context as input
- Applies generation prompt/template
- Outputs candidate response

Key observables: tokens used, generation latency, model version, temperature/sampling params

Accuracy Reviewer Agent

Acts as an internal quality gate, checking the generated response for hallucinations, context alignment, and instruction adherence before presenting to user.

- Compares response against source context
- Flags potential hallucinations or errors
- May trigger regeneration if quality is poor

Key observables: review score, identified issues, whether regeneration triggered

Formatter Agent

Transforms the reviewed response into the desired output structure—could be TL;DR summary, bulleted list, structured JSON, or other format specified by user or application requirements.

- Applies formatting instructions
- Ensures output schema compliance
- Handles edge cases (truncation, invalid formatting)

Key observables: formatting success rate, output schema validity, length constraints met

Spans, Traces, Sessions Defined

Let's formalize the observability primitives that structure all modern LLM monitoring. These concepts come from distributed systems tracing but are adapted for LLM workloads.

1 Span

The atomic unit of work: a single LLM call, tool invocation, retrieval operation, or any discrete step in your pipeline. Contains timing, inputs, outputs, and metadata.

2 Trace

A directed acyclic graph (DAG) of spans representing a single end-to-end request. Shows parent-child relationships, captures the full execution path, and enables root cause analysis.

3 Session

A temporal sequence of traces for a single user or task. Represents a conversation, a multi-step workflow, or an extended interaction. Enables analysis of user journeys and multi-turn behavior.

Span Attributes

- Duration (start/end time)
- Parent span ID
- Input/output data
- Model version
- Token counts
- Status (success/error)
- Custom metadata

Trace Attributes

- Trace ID (unique)
- Root span
- Total duration
- Span count
- User/session ID
- Request metadata
- Error flags

Session Attributes

- Session ID
- User identifier
- Trace sequence
- Session duration
- Cumulative metrics
- Conversation state
- Outcome/conversion

Example Trace Visualization

Seeing a concrete trace structure makes the concepts tangible. Here's what an instrumented version of our example pipeline looks like when visualized

Trace Structure

```
Trace: user_query_abc123|— Router Span (45ms)|  └—  
Decision: RAG path|— RAG Retrieval Span (180ms)|  └—  
Query reformulation (15ms)|  └— Vector search (140ms)|  └—  
Context assembly (25ms)|— Generator Span (420ms)|  └— LLM  
call (GPT-4, 380 tokens)|— Reviewer Span (220ms)|  └—  
Accuracy check (score: 4.5/5)|└— Formatter Span (35ms)  
└— Output: JSON structureTotal duration: 900msTotal cost:  
$0.042
```

Insights from This Trace

Performance hotspot: Generator span consumes 47% of total latency. Could explore faster models or parallel processing.

Quality signal: Reviewer gave 4.5/5 accuracy score—high confidence in output quality.

Retrieval efficiency: Vector search took 140ms—within acceptable range but could be optimized with better indexing.

Decision audit trail: Router chose RAG path with clear reasoning, enabling later analysis of routing accuracy.

Cost attribution: Can track per-span costs to understand which operations drive expenses.

This level of visibility transforms debugging from guesswork into data-driven problem solving. When users report issues, you can immediately drill into the relevant trace and pinpoint failure modes.

How Spans Enable Optimization

Observability isn't just about debugging failures, it's about continuous improvement. Span-level visibility creates a tight feedback loop for optimization.

Diagnosis: Finding Problems



Bottleneck Identification

Aggregate span durations across thousands of traces to find consistently slow operations. Sort by P95 latency to prioritize optimization efforts.



Error Hotspots

Track error rates per span type and model. If retrieval spans fail 15% of the time, that's your signal to investigate data quality or infrastructure.



Quality Degradation

Attach judge scores to specific spans. Declining accuracy scores on generator spans reveal prompt drift or model issues.

The virtuous cycle: Instrument → Measure → Identify issues → Optimize → Validate → Repeat. Spans make this cycle data-driven and systematic.

Optimization: Taking Action



Model Swapping

Replace slow, expensive models in specific spans with faster, cheaper alternatives. Test impact on quality with A/B testing across traces.



Prompt Tuning

When judge scores drop for specific span types, iterate on prompts. Measure improvement quantitatively before deploying changes.



Routing Logic

Analyze router span decisions vs. downstream quality. If RAG path consistently outperforms memory path, adjust routing thresholds.

OpenTelemetry & OpenInference

Standards matter. Rather than building proprietary observability solutions, the industry has converged on open standards that enable vendor-neutral instrumentation and portability.

OpenTelemetry (OTel)

OpenTelemetry is the industry-standard framework for generating, collecting, and exporting telemetry data—traces, metrics, and logs.

Key benefits:

- Vendor-neutral: instrument once, export anywhere
- Language support: SDKs for Python, JS, Go, Java, etc.
- Automatic instrumentation: many frameworks auto-instrument
- Rich ecosystem: integrations with all major observability platforms

OTel provides the *mechanism* for collecting telemetry but doesn't define semantic conventions specific to LLM workloads.

OpenInference

OpenInference extends OpenTelemetry with semantic conventions designed specifically for LLM and AI workloads.

LLM-specific attributes:

- Prompts and completions
- Embeddings and vectors
- Tool calls and parameters
- Retrieval queries and results
- Judge scores and evaluations
- Model versions and configs

This standardization means different tools (Phoenix, LangSmith, etc.) can consume the same trace data and provide consistent views.

Implementation path: Instrument your application with OpenTelemetry + OpenInference → Export to your observability backend of choice (Phoenix, Jaeger, vendor platforms) → Gain immediate visibility without vendor lock-in.

Different Span Types

Not all spans are created equal. Different operations in your pipeline require different metadata, have different performance characteristics, and need different evaluation strategies.

Span Type	What We Capture
LLM Span	<ul style="list-style-type: none">Prompt text and templateCompletion textModel name and versionToken counts (prompt/completion/total)Generation parameters (temperature, max_tokens, etc.)Latency breakdown (TTFT, total)Cost estimate
Retrieval Span	<ul style="list-style-type: none">Query text (original and reformulated)Retrieved document IDs and scoresRetrieval method (vector, keyword, hybrid)LatencyRelevance quality scores
Tool Span	<ul style="list-style-type: none">Tool name and versionInput parameters (structured)Output/resultSuccess/failure statusTool execution time
Judge Span	<ul style="list-style-type: none">Dimension being evaluatedJudge model usedScore (numeric rating)Rationale/explanation

Why Span Types Matter

Understanding span types enables targeted optimization strategies. Each type has different levers for improvement and different metrics that matter.



LLM Spans

Optimization focus: Language quality, latency, cost

- Swap to cheaper/faster models where quality allows
- Optimize prompts to reduce token usage
- Adjust generation parameters for speed vs. quality tradeoff
- Use streaming for better perceived latency



Retrieval Spans

Optimization focus: Grounding quality, recall, precision

- Improve embedding models for better semantic matching
- Optimize chunk sizes and overlap strategies
- Tune retrieval thresholds and ranking algorithms
- Fix data quality issues in knowledge base



Tool Spans

Optimization focus: Reliability, parameter correctness

- Improve tool descriptions in prompts
- Add parameter validation and error handling
- Optimize tool implementation performance
- Consider tool alternatives or caching



Judge Spans

Optimization focus: Evaluation accuracy, cost, latency

- Use smaller, faster judge models where sufficient
- Batch evaluation requests to reduce overhead
- Cache judge results for duplicate scenarios
- Validate judge reliability with human spot-checks

The Power of Prompts

Prompts are the primary control surface for LLM behavior. Small changes in wording, structure, or examples can dramatically impact output quality. Observability reveals which prompts actually work.

Why Prompts Matter So Much

Unlike traditional software where logic is explicit, LLM behavior is shaped by natural language instructions. Prompts encode:

- **Task definition:** What you want the model to do
- **Output format:** Structure and style requirements
- **Constraints:** What to avoid or emphasize
- **Examples:** Few-shot demonstrations of desired behavior
- **Context:** Relevant information and background

A poorly crafted prompt can cause hallucinations, off-topic responses, format violations, or safety issues—even with powerful models.

The observability connection: Without systematic evaluation, prompt improvements are based on intuition. With span-level judge scores and metrics, you can measure impact quantitatively and iterate confidently.

Prompt Optimization Workflow

Systematic prompt improvement follows a structured process. This workflow transforms prompt engineering from art into engineering discipline.

1 Define Success Criteria

Establish rubric: what makes a "good" output? Define evaluation dimensions (correctness, relevance, tone, safety) and acceptance thresholds. Create diverse test dataset representing real usage.

2 Generate Candidates

Create multiple prompt variations exploring different approaches: instruction styles, example sets, constraint phrasings, output formats. Use prompt templating for systematic variation.

3 Run Experiments

Execute all prompt candidates against test dataset. Ensure consistent evaluation conditions: same model, temperature, and random seeds. Collect outputs systematically for analysis.

4 Evaluate & Score

Apply LLM-as-a-judge evaluation across all dimensions plus numerical metrics (latency, tokens, cost). Statistical significance testing to ensure improvements aren't noise. Analyze failure modes.

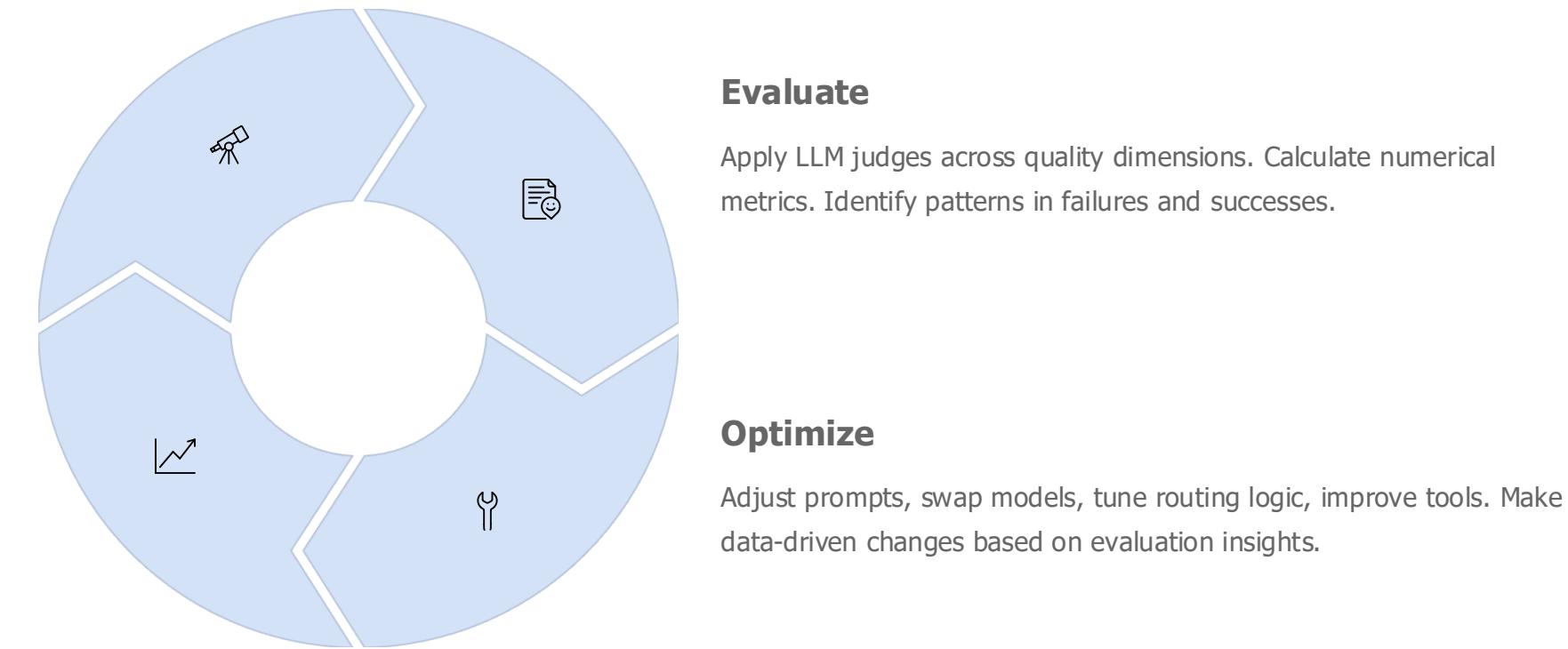
5 Deploy & Monitor

Deploy winning prompt variant to production. Continue monitoring with same evaluation framework. Watch for performance drift. Iterate when new issues emerge or use cases evolve.

- Best practice:** Version control your prompts. Store prompts as code, track changes in git, and maintain evaluation results alongside prompt versions. This enables rollback and historical analysis.

From Observability to Continuous Improvement

The true power of observability and evaluation emerges when they drive a continuous improvement flywheel. This is how production LLM systems mature from fragile prototypes to reliable services.



Key Success Factors

- Automated evaluation pipelines
- Consistent rubrics and metrics
- Representative test datasets
- Version control for all components
- Fast iteration cycles
- Clear ownership and accountability

Common Pitfalls to Avoid

- Optimizing for vanity metrics
- Insufficient test data diversity
- Over-fitting to specific examples
- Ignoring cost/latency tradeoffs
- Making changes without measurement
- Lack of rollback capabilities

Live Demo Roadmap

Now let's see these concepts in action. We'll walk through a concrete example demonstrating observability and evaluation tools working together to improve an agentic system.

Demo Flow

1 Agentic Pipeline

Show our multi-agent system handling queries with routing, RAG, generation, and review

2 Phoenix Traces

Explore trace visualization, span details, and performance metrics in Phoenix UI

3 Ax Evaluation

Run LLM-as-a-judge evaluation across multiple dimensions using Arize Ax

4 Prompt Iteration

Make a simple prompt change and demonstrate measurable improvement in quality scores

What to Watch For

- How quickly we can identify bottlenecks
- Span-level metadata revealing failure causes
- Judge scores providing quality signals
- Before/after comparison showing prompt impact
- End-to-end workflow from observation to optimization

The screenshot shows a GitHub repository card for 'AmirSaeed96/MORS_etf_2025_tutorial_demo'. The repository has a light blue background. At the top left is a small profile picture of a person with blue hair. To its right, the repository name 'AmirSaeed96/MORS_etf_2025_tutorial_demo...' is displayed, followed by a three-dot ellipsis. Below the name is a brief description: 'Codebase for observability tutorial given during the MORS ETF 2025 conference leveraging open source tools for observability.' Underneath this description are four small icons with zero counts: 'Contributors' (person icon), 'Issues' (circle icon), 'Stars' (star icon), and 'Forks' (fork icon). To the right of the repository name is the GitHub logo. Below the GitHub logo, the full repository name 'GitHub - AmirSaeed96/MORS_etf_2025_tutorial_demo: Codebase for observability' is shown, followed by another three-dot ellipsis. A detailed description follows: 'Codebase for observability tutorial given during the MORS ETF 2025 conference leveraging open source tools for observability. - AmirSaeed96/MORS_etf_2025_tutorial_demo'.

Key Takeaways & Next Steps

You can't improve what you can't see.

1 Observability First

Instrument your LLM systems from day one with spans, traces, and metrics. The cost of adding observability later is 10x higher than building it in from the start.

2 LLM-as-a-Judge is Powerful

Automatic evaluation at scale unlocks systematic improvement. Use judge models to assess quality dimensions that numerical metrics can't capture.

3 Agents Demand Visibility

Multi-step agentic systems amplify the need for span-level observability. Without it, debugging is nearly impossible.

4 Continuous Improvement Loop

Combine observability + evaluation to drive systematic optimization. Measure impact, iterate based on data, and build reliability over time.

Try This Yourself

Start Observing

- Install Phoenix (open-source, easy setup)
- Instrument your next LLM prototype
- Explore trace visualization
- Identify your first bottleneck

Start Evaluating

- Try Arize Ax or similar tools
- Define 3 core quality dimensions
- Write your first judge prompts
- Run experiments on prompt variants

Keep Learning

- Join the Arize community
- Explore OpenInference standards
- Share your findings and patterns
- Build observability culture in your team

Thank you! Questions?

Thank You!

Questions?

Connect & Learn More

- GitHub: [Link to repo.](#)
- Arize Phoenix: arize.com/phoenix
- Arize Ax: arize.com/ax
- OpenInference: openinference.io

Feedback & Follow-Up

- Email (JHU): asaeed7@jhu.edu
- Email (Arize): asaeed@arize.com



Appendix: Additional Resources

Tools & Platforms

Observability

- Arize Phoenix (OSS)
- OpenTelemetry + OpenInference
- LangSmith
- Weights & Biases
- TruLens
- MLflow

Evaluation

- Arize Ax
- PromptFoo
- RAGAS
- OpenAI Eval
- LangChain Evaluators
- DeepEval

Frameworks

- LangChain
- LlamaIndex
- AutoGen
- CrewAI
- Semantic Kernel
- Haystack

Appendix: Research & Reading

Foundational Papers

- **Attention Is All You Need** (Vaswani et al., 2017) - The transformer architecture
- **BERT** (Devlin et al., 2018) - Bidirectional transformer pretraining
- **GPT-3** (Brown et al., 2020) - Few-shot learning at scale
- **InstructGPT** (Ouyang et al., 2022) - Alignment via RLHF
- **Chain-of-Thought Prompting** (Wei et al., 2022) - Reasoning in LLMs
- **ReAct** (Yao et al., 2022) - Reasoning and acting in language models
- **Constitutional AI** (Bai et al., 2022) - Safe and helpful AI assistants

Evaluation & Observability

- **Judging LLM-as-a-Judge** (Zheng et al., 2023) - MT-Bench methodology
- **Chatbot Arena** - Large-scale human preference data
- **OpenTelemetry Semantic Conventions** - Standardized telemetry attributes

Appendix: Common Evaluation Dimensions

A reference guide for designing your evaluation strategy. Mix and match dimensions based on your use case.

Dimension	Description	Typical Scale
Correctness	Factual accuracy, absence of hallucinations	1-5 or binary
Relevance	Addresses user query, stays on topic	1-5
Completeness	Covers all aspects of request	1-5
Helpfulness	Provides actionable, useful information	1-5
Clarity	Easy to understand, well-structured	1-5
Coherence	Logical flow, consistent narrative	1-5
Conciseness	Says enough without being verbose	1-5
Tone	Appropriate style for audience	1-5 or categorical
Safety	Free of harmful, toxic content	Binary or 1-5
Bias	Lack of unfair stereotypes or prejudice	Binary or 1-5
Grounding	Uses provided context appropriately	1-5
Citation Quality	Proper attribution to sources	1-5
Format Compliance	Follows output structure requirements	Binary or 1-5
Instruction Following	Adheres to specific constraints	1-5

Appendix: Cost Optimization Strategies

Reducing Token Usage

Prompt Engineering

Eliminate unnecessary instructions, use concise language, leverage system prompts efficiently

Context Pruning

Dynamically filter retrieved documents, implement relevance thresholds, compress context intelligently

Output Length Control

Set appropriate max_tokens limits, use early stopping, optimize for required detail level

Model Selection

Task-Appropriate Models

Use smaller, faster models for simple tasks; reserve large models for complex reasoning

Cascade Architectures

Try cheap model first, escalate to expensive model only when needed

Specialized Models

Consider domain-specific or fine-tuned models that may be more efficient for your use case

Appendix: Latency Optimization Strategies

Generation Speed

1 Streaming

Use streaming APIs to improve perceived latency. Users see tokens as they're generated rather than waiting for completion.

2 Parallel Execution

When possible, run independent operations in parallel. Multiple retrieval queries or tool calls can execute concurrently.

3 Model Selection

Balance quality vs. speed. Smaller models with 10x faster inference may be acceptable for certain use cases.

4 Prompt Optimization

Shorter prompts reduce input processing time. Every token saved in the prompt speeds up TTFT.

Infrastructure & Caching

1 Embedding Cache

Cache document embeddings to avoid recomputing. Reduces retrieval latency significantly.

2 Response Cache

For repeated or similar queries, cache responses. Use semantic similarity to identify cacheable patterns.

3 Provider Selection

Different providers have different latency profiles. Benchmark and choose based on your latency requirements.

Appendix: Handling Hallucinations

Prevention Strategies

Prompt Engineering

- Explicit grounding instructions: "Only use information from the provided context"
- Request citations: "Quote the source for each claim"
- Uncertainty acknowledgment: "Say 'I don't know' when unsure"
- Conservative framing: "Avoid speculation"

Retrieval-Augmented Generation

- Provide relevant, verified context
- High-quality knowledge bases
- Freshness guarantees for time-sensitive info
- Clear source attribution

Model Selection

- Some models hallucinate less than others
- Fine-tuned models for specific domains
- Instruction-tuned models follow constraints better
- Benchmark on your task before deploying

Multi-Step Verification

- Reviewer agent checks claims
- Cross-reference with trusted sources
- Confidence scoring per statement
- Human-in-the-loop for critical outputs

Detection Strategies

LLM-as-a-Judge Hallucination Detection

Use evaluator models to compare outputs against source context. Flag statements that lack supporting evidence.

Uncertainty Quantification

Analyze token probabilities, use multiple model samples, detect inconsistencies across responses.

User Feedback Loop

Collect corrections, track user-reported inaccuracies, continuously improve based on real-world feedback.

Appendix: Prompt Engineering Best Practices

Structure & Clarity

1 Clear Instructions

Be explicit about what you want. Vague instructions yield unpredictable results.
Use imperative verbs and specific requirements.

2 Provide Context

Give the model necessary background, definitions, and constraints. Context shapes interpretation and execution.

3 Use Examples

Few-shot examples dramatically improve quality. Show don't just tell. Include edge cases in your examples.

4 Format Guidance

Specify output structure explicitly. Request JSON, markdown, or specific formatting. Provide templates when helpful.

Advanced Techniques

Chain-of-Thought

Ask model to show reasoning steps:

"Let's approach this step by step:1. First, identify...2. Then, analyze...3. Finally, conclude..."

Role Assignment

Frame the model's persona:

"You are an expert software architect with 15 years of experience in distributed systems..."

Constraints & Guardrails

Explicitly state limitations:

"Do not speculate. Do not include unverified information. Cite sources for all factual claims."

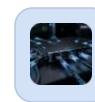
Appendix: Multi-Agent System Design Patterns

Common Patterns



Sequential Pipeline

Linear flow through specialized agents. Each agent performs one task and passes results to the next. Simple to understand and debug. Example: retrieval → generation → formatting.



Router Pattern

Initial agent classifies request and routes to specialized sub-agents. Enables handling diverse tasks with task-specific optimizations. Example: question type classifier → domain expert agents.



Collaborative Team

Multiple agents work together, each contributing expertise. Agents may critique each other's work. Higher quality but more complex. Example: researcher + writer + editor agents.



Hierarchical

Manager agent coordinates worker agents, breaking down tasks and synthesizing results. Scalable for complex workflows. Example: project manager agent → specialist agents → aggregation.

Appendix: Security & Safety Considerations

Prompt Injection & Attacks



Prompt Injection

Users may attempt to override your instructions with their own. Mitigation: clear role separation, input sanitization, output filtering, privilege boundaries.



Data Leakage

Models may expose training data or system prompts. Mitigation: monitor for unusual queries, implement rate limiting, filter sensitive output patterns.



Jailbreaking

Attempts to bypass safety constraints. Mitigation: defense-in-depth, multiple safety layers, continuous monitoring, human review for sensitive domains.

Safety Best Practices

- **Input Validation:** Sanitize user inputs, detect adversarial patterns, implement length limits
- **Output Filtering:** Scan for toxic content, PII, sensitive data before returning to users
- **Tool Safety:** Limit tool permissions, validate tool inputs, audit tool usage patterns
- **Rate Limiting:** Prevent abuse through request throttling and cost caps per user
- **Monitoring:** Track unusual patterns, flag potential attacks, maintain audit logs
- **Human Review:** Implement human-in-the-loop for high-stakes decisions

Appendix: Building Your Evaluation Dataset

Dataset Characteristics

Diversity

Your evaluation dataset should cover:

- Common use cases (80% of traffic)
- Edge cases and error conditions
- Different user personas and intents
- Varying complexity levels
- Positive and negative examples
- Seasonal or temporal variations

Quality

Ensure dataset quality through:

- Real production queries when possible
- Manual review and curation
- Removing PII and sensitive data
- Balanced representation
- Version control and updates
- Clear annotation guidelines

Dataset Size Guidelines

50-100

Minimum Viable

Small set for initial iteration

500+

Production Ready

Robust evaluation coverage

5K+

Enterprise Scale

Statistical significance across segments

Appendix: Debugging Common Issues

Symptom → Diagnosis → Solution

Symptom	Likely Causes	Debugging Steps
High latency	Large prompts, slow model, inefficient retrieval, synchronous operations	Check span durations, identify bottleneck, consider parallel execution or model swap
Hallucinations	Insufficient context, poor prompt, weak grounding instructions, model limitations	Review retrieval quality, strengthen prompt constraints, add reviewer agent, check judge scores
Format errors	Ambiguous instructions, no examples, output validation missing	Provide explicit format template, add few-shot examples, implement schema validation
High cost	Large model for simple tasks, inefficient prompts, no caching, excessive retries	Analyze token usage per span, consider model cascade, implement caching, reduce prompt length
Tool failures	Malformed parameters, wrong tool selection, tool implementation bugs	Check tool span error rates, validate parameter formats, improve tool descriptions in prompts
Inconsistent quality	High temperature, context variability, model non-determinism	Lower temperature, add more constraints, implement quality gates, use consistency checks