Evaluation in the Age of Intelligent Systems

From Software Engineering to Agentic AI

Filippo Lentoni

Sr Applied Scientist, Amazon
Columbia University

Agenda

Why evaluation gets harder (and why agents force the issue)

- Modern AI systems fail in more ways than "wrong output"
- Each layer adds uncertainty and expands what must be evaluated
- Agents force evaluation to become **continuous** (not a pre-launch checkbox)

Core thesis: evaluation complexity compounds

Uncertainty shifts over time:

Inputs $\rightarrow$ **Learned parameters** $\rightarrow$ **Token trajectories** $\rightarrow$ **System behavior & feedback loops**

Each step **builds on** the previous layer: you keep the old evaluation problems and add new ones.

What are we evaluating, really?

Definition

Evaluation = measuring the gap between intended behavior and observed behavior under uncertainty.

- Object: output, prediction, completion, trajectory, or workflow outcome
- Constraints: latency, cost, safety, format, coverage, adoption
- Evidence: tests, datasets, human labels, traces, dashboards

1) Traditional Software Engineering

- Deterministic program: same input $\Rightarrow$ same output
- Correctness is (mostly) binary and spec-driven
- Failures are reproducible: bugs, regressions, integration issues

Software evaluation toolbox

- Unit tests, integration tests, end-to-end tests
- Contracts, invariants, assertions
- Static analysis, type checks, fuzzing

Mental model

If it fails, the code is wrong (or the spec is).

2) ML Systems: what changes?

- Behavior is learned from data (not explicitly coded)
- Correctness is statistical: performance depends on the data distribution
- Generalization matters: in-sample vs out-of-sample

Core shift

From *"is it correct?"* to *"how often is it correct, and under what distribution?"*

ML evaluation primitives

- Train/validation/test splits; cross-validation
- Metrics: accuracy, precision/recall, calibration, AUC, error analysis
- Failure modes: overfitting, leakage, distribution shift

Mental model

The system is approximately correct (with uncertainty bounds).

What Can LLMs Do?

Core Capabilities:

- **Summarization** – Condense documents
- **Generation** – Create new content
- **Classification** – Categorize inputs
- **Translation** – Cross-language conversion
- **Extraction** – Structured data from text

Evaluation Challenge:

- Output is **free-form text**
- Multiple valid answers exist
- "Correctness" is often subjective
- Traditional accuracy doesn't apply

Key Insight

Unlike classification (one correct label), LLM outputs require **semantic** evaluation.

LLMs: A Distributional View

- Output is a **stochastic token trajectory**, not a single label
- Multiple completions can be acceptable for the same prompt
- Decoding parameters (temperature, top-p) change behavior

Generative Process

$$P(y_1, y_2, \ldots, y_n \mid x) = \prod_{t=1}^{n} P(y_t \mid y_{<t}, x)$$

where $y_{<t}$ are all tokens before position t.

Control Point	**Description**
Model Choice	Claude, GPT, Llama, Nova, Gemini, etc.
Reasoning Mode	Enable/disable extended thinking (CoT)
Parameters	Temperature, top-p, max tokens
Prompt	**Most impactful lever**
– Static	System instructions, persona, rules
– Dynamic	User input, retrieved context

Key Insight

Prompt engineering is often more impactful than model selection for task performance.

Prompt Engineering Strategies

Core Techniques:

1. **Clear Instructions**
   Explicit output format, constraints
2. **Chain-of-Thought (CoT)**
   "Think step by step..."
3. **Few-Shot Examples**
   Provide input/output pairs
4. **Role/Persona**
   "You are an expert in..."

Advanced Techniques:

1. **Self-Consistency**
   Sample multiple, take majority
2. **Decomposition**
   Break complex tasks into steps
3. **Guardrails**
   Define boundaries, edge cases
4. **Abstention Prompting**
   Allow "I don't know" (next slide)

Evaluation Implication

Each prompt version is a "model" that must be evaluated systematically.

Abstention: Trading Coverage for Accuracy

Problem: LLMs have a bias toward producing answers even when uncertain.

Abstention Prompting ("Unable to Classify")

Explicitly instruct the model to decline when information is insufficient:

> *"If the input lacks sufficient information about [X, Y, Z], respond with 'Unable to determine' rather than guessing."*

Without Abstention:

- High coverage
- Lower precision
- Hidden errors

With Abstention:

- Lower coverage
- Higher precision
- Explicit uncertainty

When to Use

Use abstention when the **cost of a wrong answer** exceeds the **cost of no answer**.

Avoiding Overfitting in Prompt Engineering

Key Insight: Prompts can overfit to evaluation data, just like ML models.

The Problem

- Developer iterates prompt on a fixed set of examples
- Prompt becomes highly tuned to those specific cases
- Performance degrades on unseen production data

Best Practices

1. **Train/Test Split**: Calibrate prompt on one set, evaluate on held-out set
2. **Stratified Sampling**: Ensure test set covers diverse input types
3. **Version Control**: Track all prompt iterations and their metrics
4. **Production Monitoring**: Compare offline metrics to online performance

Analogy to ML

Prompt = Model, Examples in Prompt = Training Data, Evaluation Set = Test Set

LLM Evaluation Toolkit

1. **Rubric-based Human Evaluation**
   - Dimensions: correctness, completeness, safety, format, helpfulness
   - Gold standard but expensive and slow
2. **Task-Specific Metrics**
   - Extraction: accuracy, F1 on entities
   - Classification: precision, recall, confusion matrix
   - Code: execution success, test pass rate
   - Structured output: schema validity
3. **LLM-as-a-Judge**
   - Use a (often stronger) LLM to score outputs
   - Scalable but requires calibration against human labels
   - Must monitor for judge drift over time

Key Principle

Match evaluation method to **task risk** and **output structure**.

Offline Evaluation Workflow

Step-by-Step Process:

1. **Create Evaluation Dataset**
   - Diverse inputs: typical cases, edge cases, adversarial examples
   - Ground truth labels from domain experts (SMEs)
2. **Run Model/Prompt on Held-Out Test Set**
3. **Compute Metrics** (task-appropriate)
4. **If metrics meet threshold** $\rightarrow$ Deploy
5. **If not** $\rightarrow$ Iterate on prompt, model, or architecture

Critical

Test on **unseen data** to avoid prompt overfitting.

Case Study: LLM for Hierarchical Classification

Task: Multi-level decision tree classification (e.g., incident triage, document routing)

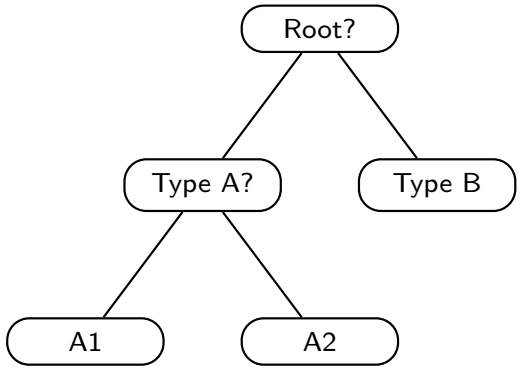
Approach:

- Prompt-guided LLM navigates hierarchy top-to-leaf
- Chain-of-thought reasoning at each node
- Abstention when evidence insufficient

Why GenAI over Traditional ML?

- No labeled training data required
- Native explainability (reasoning trace)
- Faster iteration (days vs. weeks)
- Handles semantic nuance

Decision Tree Example:



Key Insight

This "LLM app" borrows heavily from **ML-style evaluation rigor**.

Classification Metrics for LLM Applications

When LLMs perform **structured tasks** (classification, extraction), use standard ML metrics:

Core Metrics:

- **Accuracy**: $\frac{TP+TN}{Total}$
- **Precision**: $\frac{TP}{TP+FP}$
- **Recall**: $\frac{TP}{TP+FN}$
- **F1 Score**: Harmonic mean

When Costs are Asymmetric:

- If FN is costly $\rightarrow$ optimize **Recall**
- If FP is costly $\rightarrow$ optimize **Precision**
- Use confusion matrix for detailed analysis

The Coverage-Accuracy Trade-off

$$\text{Coverage} = \frac{\text{Cases Classified}}{\text{Total Cases}}$$

With abstention: lower coverage $\leftrightarrow$ higher precision on classified cases.

Designing User Acceptance Testing (UAT)

Goal: Validate model performance before production deployment.

UAT Design Principles

1. **Stratified Sampling**: Ensure representation across categories
   - Especially important when class distribution is skewed
   - Sample proportionally or oversample rare classes
2. **SME Validation**: Domain experts establish ground truth
3. **Holdout Data**: Test on data not used for prompt development
4. **Slice Analysis**: Report metrics by subgroup, not just aggregate

Common Pitfall

Aggregate metrics can hide poor performance on important subgroups. Always analyze slices.

Example: Evaluation Metrics Report

Metric	Value	Interpretation
Coverage	77%	Cases auto-classified
Precision	93%	Correct among positives
Recall	100%	All true positives found
False Negative Rate	0%	No missed critical cases

Coverage Gap Analysis:

- 10% – Requires human review (by design)
- 9% – Prompt engineering opportunity
- 4% – Input data quality issues

Reporting Best Practice

Always report **why** coverage isn't 100% – separates model limitations from design choices.

Explainability-Driven Feedback Loop

Advantage of LLMs: Native chain-of-thought reasoning provides explainability.

Benefits of Explainability

1. **Validate reasoning**: Even if output is wrong, was the logic sound?
2. **Debug failures**: Identify if error is model vs. data quality issue
3. **Build trust**: Users/auditors can verify AI reasoning
4. **Improve prompts**: Reasoning traces reveal what to fix

Feedback Loop:

$$\text{Error} \xrightarrow{\text{Review Reasoning}} \text{Root Cause} \xrightarrow{\text{Fix}} \text{Prompt Update} \xrightarrow{\text{Re-evaluate}}$$

Contrast with Traditional ML

Black-box models require post-hoc explanations (SHAP, LIME). LLMs can explain natively.

Post-Deployment: Continuous Monitoring

Offline evaluation is necessary but not sufficient.

Online Monitoring Activities

1. **Log all predictions**: Input, output, reasoning, latency, cost
2. **Periodic Audits**: Weekly/monthly sampling of production outputs
3. **Ground Truth Creation**: SMEs label sampled cases
4. **Metric Tracking**: Monitor for drift from offline performance
5. **Feedback Integration**: User signals (thumbs up/down, corrections)

Continuous Improvement Cycle

$$\text{Production} \rightarrow \text{Sample} \rightarrow \text{Label} \rightarrow \text{Analyze} \rightarrow \text{Update Prompt} \rightarrow \text{Re-deploy}$$

Labeled production errors should feed back into the offline evaluation dataset.

Self-Service Experimentation Platforms

Goal: Enable domain experts to iterate on prompts without ML engineering support.

Key Platform Capabilities

1. **Quick Test Mode**: Test prompts on individual examples instantly
2. **Batch Evaluation**: Run prompts against full evaluation sets
3. **Experiment Tracking**: Version prompts, record metrics per iteration
4. **Multi-Model Support**: Compare Claude, GPT, Llama, etc.
5. **Human-in-the-Loop**: UI for SME review and labeling
6. **Copilot Mode**: AI-assisted prompt refinement suggestions

Impact

Self-service platforms can reduce ML engineer involvement by 50-75% for prompt-based applications.

LLM Evaluation: Key Takeaways

1. **Free-form output** requires semantic evaluation, not exact match
2. **Traditional metrics** (BLEU, ROUGE, METEOR) have limitations; use task-specific metrics when possible
3. **Prompt = Model**: Treat prompt versions like model versions; evaluate systematically
4. **Avoid overfitting**: Test on held-out data, use stratified sampling
5. **Abstention** trades coverage for accuracy – use when error cost is high
6. **Explainability** enables debugging and builds trust
7. **Continuous monitoring**: Offline metrics $\neq$ online performance; audit regularly
8. **Close the loop**: Production errors $\rightarrow$ evaluation dataset $\rightarrow$ better prompts

Core Principle

Good LLM evaluation is **workflow engineering**: metrics + audits + iteration tooling.

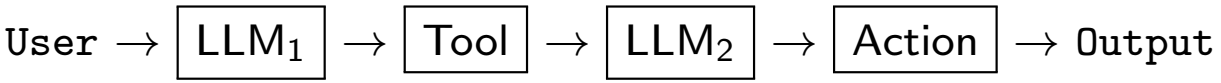
From LLMs to Agents

LLM Application:

- Single model invocation
- Direct input → output
- Stateless (typically)
- Limited to text generation

Agent:

- Orchestrated system
- Multiple LLM calls
- Uses tools & external systems
- Multi-step reasoning
- Maintains state across turns



Evaluation Implication

Increased complexity = more failure modes = more evaluation dimensions required.

Anatomy of an LLM Agent

Definition

An **agent** is a system that uses an LLM as its reasoning engine to:

1. Interpret user intent
2. Plan a sequence of actions
3. Execute actions using tools
4. Observe results and iterate

Core Components:

- **Planner/Orchestrator**: Decides what to do next (LLM-based)
- **Tools**: External capabilities (APIs, databases, code execution)
- **Memory**: Context persistence across interactions
- **Guardrails**: Safety and policy constraints

Agent Loop (Simplified)

`Observe` $\rightarrow$ `Think` $\rightarrow$ `Act` $\rightarrow$ `Observe` $\rightarrow$ ...

Examples of Agentic Applications

Agent Type	Description	Key Evaluation Focus
Data Analysis Agent	Query databases, generate reports, create visualizations	Query correctness, data grounding
Explainability Agent	Analyze code/models, explain inputs/outputs	Reasoning accuracy
Document Intelligence	Extract, summarize, answer questions from documents	Extraction completeness, citation accuracy
Coding Agent	Generate, review, debug, and execute code	Functional correctness, security
Orchestrator Agent	Coordinate multiple sub-agents	Action sequencing, delegation accuracy
Research Agent	Search, synthesize, and summarize information	Source quality, hallucination rate

What We Control in Agent Systems

Control Point	Description	Optimization Goal
Agent Architecture	Number of agents, interaction patterns (sequential, parallel, hierarchical)	Right-size complexity
Model per Agent	Which LLM for each sub-task	Cost/performance balance
Agent Prompts	System instructions per agent	Task accuracy
Tools / MCP	Available actions, APIs, data sources	Capability coverage
Orchestration Logic	When to call which agent/tool	Efficiency, correctness

Design Principle

Find the **simplest and cheapest** combination of agents that solves your problem. More agents $\neq$ better.

Why Agent Evaluation is Challenging

Agents introduce new failure modes beyond single LLM calls:

1. **Multi-step dependencies**: Error in step 1 cascades to step N
2. **Tool selection**: Agent may choose wrong tool or wrong parameters
3. **State management**: Context may be lost or corrupted across turns
4. **Non-determinism**: Same query can take different paths
5. **Partial success**: Agent may complete 4/5 steps correctly
6. **Side effects**: Actions may change external state (databases, APIs)

Key Insight

Single accuracy metric is insufficient. Need **multi-dimensional evaluation** covering the full agent lifecycle.

AI Metrics Taxonomy for Agents

Hierarchical Relationship:

$$\text{Adoption} \xleftarrow{\text{depends on}} \text{Coverage} \xleftarrow{\text{within scope}} \begin{cases} \text{Accuracy} \\ \text{Latency} \\ \text{Cost} \end{cases}$$

Interpretation

- **Adoption** is the *outcome* metric (downstream of quality)
- **Coverage** defines what the agent *can* answer
- Within coverage, optimize for **Accuracy**, **Latency**, and **Cost**

Warning

Never interpret adoption in isolation—declining adoption signals coverage or quality issues.

Agent Metrics: Definitions

Metric	Definition	How Measured
Adoption	User engagement with the agent	Unique users, queries/user, return rate, session length
Coverage	Percentage of in-scope queries the agent can handle	% handled vs. refused or failed
Latency	Time from query to complete response	Time to first token, total time, # of turns
Accuracy	Correctness, completeness, and safety of responses	Via Accuracy Bridge taxonomy
Cost	Resource consumption per query	$/query, tokens used, API calls, compute

Trade-offs

Improving accuracy may increase latency and cost. Optimizing cost may reduce accuracy. Evaluation must consider all dimensions.

Coverage: Defining Agent Scope

Coverage = the spectrum of queries the agent is designed to handle.

Expected Behaviors

- **In-scope query**: Agent should provide accurate, complete response
- **Out-of-scope query**: Agent should *explicitly refuse*
  - "I'm not able to answer this question."
  - "This is outside my capabilities. Please contact [X]."

Coverage Analysis:

1. Log all user queries (in-scope and out-of-scope)
2. Analyze out-of-scope queries to distinguish:
   - True misuse (user asking wrong agent)
   - Capability gap (should be in scope, but isn't)
3. Use gaps to inform roadmap and expand evaluation dataset

Key Insight

Refusing gracefully is a **correct behavior**—test for it explicitly.

Latency: Breaking Down Response Time

Latency can be decomposed into multiple components:

Component	Definition
Time to First Token (TTFT)	Time until agent starts responding
Total Response Time	Time from query to complete response
Per-Tool Latency	Time spent in each tool invocation
Number of Turns	Conversation rounds to reach answer
Planning Time	Time spent in reasoning/orchestration

Context-Dependent Thresholds

Acceptable latency depends on use case:

- Interactive chat: < 5 seconds expected
- Complex analysis: Minutes may be acceptable
- Background task: Hours may be fine

Define latency thresholds **per task type** in your evaluation criteria.

Accuracy: Beyond a Single Number

Problem: "The agent was 85% accurate" doesn't enable improvement.

The Accuracy Bridge

A standardized taxonomy that decomposes inaccuracies into:

1. **Failure Type**: What went wrong (user-visible outcome)
2. **Root Cause**: Why it went wrong (technical origin)
3. **Example**: Concrete illustration

Benefits:

- **Actionable**: Links failure type to fix (prompting? tools? retrieval?)
- **Comparable**: Same taxonomy across different agents
- **Trackable**: Monitor if fixes reduce specific failure types over time
- **Reportable**: Aggregated view for stakeholders

The Accuracy Bridge: Failure Taxonomy (1/2)

Failure Type	Description	Root Causes	Example
Hallucination (Factually Wrong)	Information that is incorrect or not grounded in source data	Prompting, Retrieval, Tool error, Model limitation, Data quality	Agent returns incorrect value for a metric
Incomplete	Response omits critical information required to answer	Prompting, Orchestration, Retrieval, Model limitation	Agent explains variance but omits primary driver
Extra Information	Includes unrequested, irrelevant information	Prompting, Model limitation	Agent adds unrelated suggestions to a factual query
Wrong Format	Content correct, but format doesn't match request	Prompting, Output schema, Model limitation	Returns paragraph when table was requested

The Accuracy Bridge: Failure Taxonomy (2/2)

Failure Type	Description	Root Causes	Example
Wrong Action	Executes incorrect, unintended, or unsafe state-changing action	Orchestration logic, Prompting, Tool config, Missing guardrails	Agent updates wrong record in database
Throttling / Timeout	Fails to complete due to rate or time limits	Infrastructure limits, Retry logic, Token constraints, Tool latency	Agent stops mid-reasoning, returns error
Unsafe / Policy Violation	Response or action violates safety, compliance, or policy	Prompting, Missing guardrails, Tool misuse, Model limitation	Agent exposes restricted data or performs unauthorized action

Using the Accuracy Bridge

When labeling an inaccurate response:

1. Assign exactly one **failure type**
2. Identify the most likely **root cause**
3. This enables targeted fixes and trend analysis

Root Cause → Remediation Mapping

Root Cause	Typical Remediation
Prompting	Refine system prompt, add examples, improve instructions
Retrieval (RAG)	Improve chunking, embedding model, retrieval strategy
Tool Configuration	Fix tool parameters, add validation, improve tool descriptions
Orchestration Logic	Adjust agent routing, improve planning prompts
Missing Guardrails	Add input/output validation, safety filters
Model Limitation	Try different model, add reasoning steps, decompose task
Data Quality	Fix upstream data, add data validation
Infrastructure	Increase limits, add retries, optimize for latency

Feedback Loop

Accuracy Bridge → Root Cause Analysis → Targeted Fix → Re-evaluate

Two Evaluation Modes: Offline and Online

Offline Evaluation
(Pre-Deployment)

- Run agent against curated dataset
- Ground truth available
- Controlled environment
- Gate for deployment
- Catch issues *before* users see them

Online Evaluation
(Post-Deployment)

- Monitor production traffic
- Ground truth often unavailable
- Real user behavior
- Continuous monitoring
- Catch issues *in the wild*

Both Are Required

- Offline: Necessary but not sufficient (can't anticipate all real queries)
- Online: Catches drift, edge cases, and unexpected user behavior

Offline Evaluation Workflow

Step-by-Step Process:

1. **Create Evaluation Dataset**
   - Representative of expected production queries
   - Include edge cases and adversarial examples
   - Include out-of-scope queries (refusal = correct)
   - Ground truth labels from domain experts (SMEs)
2. **Run Agent Against Dataset**
   - Execute full agent pipeline (tools, reasoning, output)
   - Capture all traces: prompts, tool calls, intermediate steps
3. **Evaluate and Label**
   - Compute metrics (coverage, accuracy, latency, cost)
   - Label failures using Accuracy Bridge
4. **Decision Gate**
   - Meets quality bar? $\rightarrow$ Deploy
   - Below bar? $\rightarrow$ Iterate (prompts, tools, architecture)

Building a High-Quality Evaluation Dataset

Best Practices:

Do:

- ✓ Define expected output for each input
- ✓ Include diverse query types
- ✓ Add edge cases and ambiguous inputs
- ✓ Include multi-turn conversations
- ✓ Test refusal behavior (out-of-scope)
- ✓ Use SMEs for ground truth
- ✓ Consider cost of mistakes per category

Don't:

- × Only happy-path scenarios
- × Homogeneous query types
- × Ground truth without SME review
- × Static dataset (never updated)
- × Ignore tool invocation correctness

Synthetic Data Generation

Use an LLM to generate diverse test queries based on your agent's scope. Human review is still required for ground truth labels.

Versioning: Experiments and Datasets

Critical Practice: Version *both* the agent and the evaluation dataset.

What to Version

- **Agent Configuration**
  - Model(s) used, prompts, tool configurations
  - Orchestration logic, parameters
- **Evaluation Dataset**
  - Input queries, expected outputs, labels
  - Dataset version ID, creation date, author
- **Results**
  - Metrics per experiment run
  - Mapping: Agent version $\times$ Dataset version $\rightarrow$ Metrics

Why It Matters

Enables: reproducibility, regression detection, A/B comparison, audit trails.

Online Evaluation Workflow

Post-Deployment Monitoring:

1. **Automatic Telemetry Capture**
   - Log all interactions: queries, outputs, tool calls, reasoning
   - Capture latency per step, total cost, tokens used
   - Collect user feedback (thumbs up/down, explicit ratings)
2. **Trace Review** (choose based on volume)
   - *Low volume*: Review all traces manually
   - *Medium volume*: Use LLM-as-a-Judge to flag, then human review
   - *High volume*: Sample (stratified), then LLM-as-a-Judge + human review
3. **Label Using Accuracy Bridge**
   - Enables root cause analysis
   - Feeds directly into reporting
4. **Export to Offline Dataset**
   - Labeled production traces become new test cases

Leveraging User Feedback

Types of User Signals:

Signal	What It Captures	Limitation
Thumbs Up/Down	Binary satisfaction	No detail on *why*
Star Rating	Graded satisfaction	Still lacks specifics
Free-text Feedback	Detailed user input	Unstructured, hard to aggregate
Explicit Corrections	What should have been said	High signal, rare
Implicit Signals	Retry, abandon, escalate	Requires inference

Best Practice

- Use **thumbs-down** as a prioritization signal for manual review
- Don't rely solely on feedback—users don't always report errors
- Combine with automated flagging (LLM-as-a-Judge)

LLM-as-a-Judge: Scaling Evaluation

Concept: Use an LLM to evaluate another LLM or agent's outputs.

Two Operating Modes

Mode	Offline (with ground truth)	Online (no ground truth)
Input	Query + Agent Output + Expected Output	Query + Agent Output + Trace
Task	Score against reference	Flag likely issues
Use Case	Automate scoring at scale	Prioritize traces for review

Critical Requirement

LLM-as-a-Judge must be **calibrated** against human-labeled data. Human labeling remains the source of truth.

Designing an LLM-as-a-Judge

Key Design Decisions:

1. **Judge Model Selection**
   - Often use a stronger/different model than the agent
   - Trade-off: cost vs. reliability
2. **Judge Prompt Engineering**
   - Define evaluation criteria explicitly
   - Provide rubric with examples
   - Request structured output (scores + reasoning)
3. **Calibration Process**
   - Run judge on human-labeled gold set
   - Measure agreement rate (Cohen's Kappa, accuracy)
   - Set threshold: only deploy if agreement $> X\%$
4. **Ongoing Monitoring**
   - Periodically re-validate against new human labels
   - Watch for judge drift

LLM-as-a-Judge: Pitfalls and Mitigations

Pitfall	Mitigation
Self-preference bias	Judge favors outputs similar to its own style. Use different model family for judge.
Position bias	In pairwise comparison, judge prefers first/second option. Randomize order, average scores.
Verbosity bias	Longer responses rated higher regardless of quality. Explicitly instruct to ignore length.
Lack of calibration	Judge disagrees with humans. Mandatory calibration on gold set before deployment.
Domain blindness	Judge lacks domain expertise. Include domain context in prompt, or use domain-specific judge.

Golden Rule

Never fully automate evaluation with LLM-as-a-Judge without ongoing human validation.

Sampling Strategies for High-Volume Agents

Problem: Can't manually review every trace when volume is high.

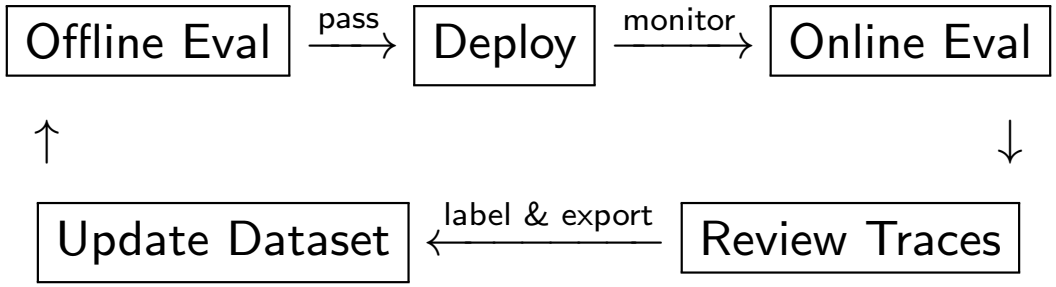
Sampling Approaches

1. **Random Sampling**
   - Simple, unbiased estimate of overall quality
   - May miss rare but important failure modes
2. **Stratified Sampling**
   - Sample proportionally across categories (query type, user segment)
   - Ensures representation of all strata
3. **Importance Sampling**
   - Oversample high-risk or high-value interactions
   - Weight results to recover population estimates
4. **Signal-Based Sampling**
   - Prioritize: thumbs-down, long latency, errors, LLM-as-a-Judge flags

Combine strategies: e.g., random baseline + signal-based enrichment.

Closing the Feedback Loop

Continuous Improvement Cycle



Key Activities:

1. Label production failures using Accuracy Bridge
2. Export labeled traces to offline evaluation dataset (new version)
3. Test new agent versions against real failure modes
4. Use for **non-regression testing**: ensure fixes don't break other cases

Result

Over time, evaluation dataset reflects real production challenges → more robust pre-deployment testing.

Advanced: Automated Feedback Analysis

Concept: Use an agent to analyze evaluation data and suggest improvements.

Feedback Analysis Agent

Inputs:

- LLM-as-a-Judge scores
- Human labels (Accuracy Bridge categories)
- User feedback (thumbs up/down, comments)
- Agent traces and codebase structure

Outputs:

- Pattern detection across failures (common root causes)
- Suggested prompt refinements
- Identified capability gaps
- Prioritized improvement opportunities

Example Platforms: Langfuse, LangSmith, Weights & Biases Weave, custom solutions.

Agent Observability: What to Capture

Comprehensive Tracing:

Component	What to Log
Input	User query, session ID, timestamp, user metadata
Planning	Reasoning steps, selected tools, agent routing decisions
Tool Calls	Tool name, parameters, response, latency, errors
LLM Invocations	Prompt, completion, model, tokens, latency, cost
Output	Final response, confidence signals, citations
Feedback	User rating, explicit feedback, implicit signals
Metadata	Agent version, experiment ID, environment

Design Principle

Log at a granularity that enables **root cause analysis**. If you can't trace a failure to a specific step, you can't fix it.

Standardized Reporting

Example Status Report Structure:

"As of [date], Agent X can answer **X%** *of in-scope questions (Coverage).* **X users** *submitted* **X queries** *this period, ±X% vs. prior (Adoption). Average latency was* **X seconds** *over* **X turns** *(Latency). Total cost was* **$X** *(Cost). Of* **X reviewed interactions**, **Y** *were inaccurate (Accuracy):*

- *X cases: hallucination (root cause: retrieval)*
- *X cases: wrong action (root cause: tool config)*
- *X cases: timeout (root cause: infrastructure)*

Corrective actions: ..."

Benefits of Standardization

- Comparable across agents
- Directly derived from Accuracy Bridge labels
- Enables trend tracking over time

Common Risks in Agent Evaluation

Risk	Description	Mitigation
Under-labeling	Not enough traces reviewed	Set review targets, allocate dedicated time
Eval Set Gaming	Eval dataset doesn't represent production	Follow best practices, continuously update from production
Judge Misuse	LLM-as-a-Judge without calibration	Mandatory calibration, periodic re-validation
Metric Fixation	Optimizing one metric at expense of others	Track all five dimensions, set balanced thresholds
Drift Blindness	Not detecting degradation over time	Automated monitoring, scheduled audits

Agent Evaluation Tooling Landscape

Open Source & Commercial Platforms:

Tool	Key Capabilities
Langfuse	Tracing, scoring, prompt management, analytics
LangSmith	Tracing, evaluation, dataset management, hub
Weights & Biases	Experiment tracking, Weave for LLM tracing
Arize Phoenix	Observability, evaluation, embedding analysis
Braintrust	Eval framework, scoring, CI/CD integration
Ragas	RAG-specific evaluation metrics
DeepEval	Unit testing framework for LLMs
OpenAI Evals	Evaluation framework and benchmarks

Build vs. Buy

Evaluate based on: integration needs, customization requirements, scale, and team expertise.

Agent Evaluation: Key Takeaways

1. **Agents are complex systems** with more failure modes than single LLM calls
2. **Multi-dimensional metrics**: Adoption, Coverage, Latency, Accuracy, Cost
3. **Accuracy Bridge**: Standardized failure taxonomy enables actionable insights
   - Failure Type $\rightarrow$ Root Cause $\rightarrow$ Targeted Fix
4. **Offline + Online**: Both required for comprehensive evaluation
5. **LLM-as-a-Judge**: Scales evaluation but must be calibrated; humans remain source of truth
6. **Close the loop**: Production failures $\rightarrow$ evaluation dataset $\rightarrow$ better agents
7. **Observability is foundational**: Can't improve what you can't trace
8. **Version everything**: Agent configs, datasets, and results

LLM vs. Agent Evaluation: Summary Comparison

Dimension	LLM Application	Agent
Complexity	Single model call	Multi-step, multi-component
Failure Modes	Output quality	+ Tool selection, orchestration, state
Key Metrics	Task-specific (precision, recall, BLEU, etc.)	Adoption, Coverage, Latency, Accuracy, Cost
Accuracy Analysis	Binary or rubric-based	Accuracy Bridge taxonomy
Ground Truth	Often available offline	Harder to define for multi-step tasks
Observability	Input/output logging	Full trace: planning, tools, intermediate steps
Iteration Unit	Prompt version	Agent config (prompts + tools + orchestration)

Unifying Principle

Both require: systematic evaluation, continuous monitoring, feedback loops, and human oversight.

Questions?

Key Resources

- Langfuse Documentation: `https://langfuse.com/docs`
- LangChain Evaluation Guide: `https://python.langchain.com/docs/guides/evaluation`
- "Judging LLM-as-a-Judge" (Zheng et al., 2023)
- RAGAS: `https://docs.ragas.io`

Thank you!