Chapter 9. Validation and Measurement

It has never been easier to build products and applications, but effectively measuring these systems remains an enormous challenge. While teams are often under pressure to ship things quickly, taking the time to rigorously evaluate performance and assess quality pays long-term dividends and enables teams to ultimately move faster and with more confidence. Without rigorous evaluation and measurement, decisions about which changes to ship become much more difficult. Rigorous measurement and validation become essential, not only to optimize performance but also to build trust and ensure alignment with user expectations.

This chapter explores methodologies for evaluating agent-based systems, covering key principles, measurement techniques, and validation strategies. We explore the critical role of defining clear objectives, selecting appropriate metrics, and implementing robust testing frameworks to assess system performance under real-world conditions. Beyond mere functionality, the reliability of agent outputs—including accuracy, consistency, coherence, and responsiveness—requires systematic scrutiny, particularly given the probabilistic nature of foundation models that often power these systems.

Throughout this chapter, we follow a customer support agent handling a common ecommerce scenario: a customer reports a cracked coffee mug and requests a refund. We'll build on this case, exploring variations like multi-item orders, cancellations, or changes in addresses, to illustrate measurement, validation, and deployment.

Measuring Agentic Systems

Without rigorous measurement, it is impossible to ensure that the system meets its intended goals or handles the complexities of real-world environments. By defining clear objectives, establishing relevant metrics, and employing systematic evaluation processes, developers can guide the design and implementation of agent systems toward achieving high performance and user satisfaction.

Measurement Is the Keystone

Effective measurement begins with identifying clear, actionable metrics that align with the goals and requirements of the agent system. These metrics serve as the benchmarks for evaluating the agent's ability to perform tasks and meet user expectations. Success depends on defining specific, measurable objectives that reflect the desired outcomes for the system, such as enhancing user engagement or automating a complex process. By framing hero scenarios —representative examples of high-priority use cases—developers can ensure their metrics target the core functions that define the agent's success. In the absence of rigorous and ongoing measurement, it becomes impossible to know whether changes are truly improvements, to understand how agents perform in realistic and adversarial settings, or to guard against unexpected regressions.

Selecting the right metrics is equally crucial. Metrics should encompass a combination of quantitative indicators, such as accuracy, response time, robustness, scalability, precision, and recall, as well as qualitative measures like user satisfaction. For example, in a customer service agent, response time and accuracy might measure performance, while user feedback captures overall satisfaction. These metrics must reflect the real-world demands the system will face.

In the case of language-based agents, traditional exact-match metrics frequently fail to capture genuine utility, as correct answers can take many forms. As a result, modern practice relies increasingly on semantic similarity measures—such as embedding-based distance, BERTScore, BLEU (Bilingual Evaluation Understudy), or ROUGE (Recall-Oriented Understudy for Gisting Evaluation)—to evaluate whether agent outputs truly meet the intent of a given task, even if the wording diverges from a reference answer.

To realize the benefits of measurement, it is crucial to integrate evaluation mechanisms directly into the agent-development lifecycle. Rather than relegating evaluation to the end, successful teams automate as much as possible, triggering tests whenever new code is merged or models are updated. By maintaining a consistent source of truth for key metrics over time, it becomes possible to detect regressions early, preventing new bugs or degradations from reaching production. Automated evaluation, however, rarely tells the whole story. Particularly in novel or high-stakes domains, regular sampling and human-in-the-loop review of agent outputs can uncover subtle issues and provide a qualitative sense of progress or remaining challenges. The most effective teams treat evaluation as an iterative process, refining both their agents and their metrics in response to ongoing feedback and changing requirements.

Integrating Evaluation into the Development Lifecycle

Measurement must not be an afterthought, nor can it be left to informal methods such as simply "eyeballing" outputs or relying on gut instinct. In the absence of systematic evaluation, it is all too easy for even expert teams to fool themselves into believing their agentic systems are improving, when in fact progress is illusory or uneven. Leading teams integrate automated, offline evaluation into every stage of development. As new tools or workflows are added to an agent, corresponding test cases and evaluation examples should be added to a growing evaluation set. This disciplined approach ensures that progress is measured not just against a fixed benchmark, but across the expanding scope of the system's capabilities.

High-quality evaluation sets can act as a living specification for what the agent must handle, supporting reproducibility and regression detection as the system evolves. By tracking historical results on these evaluation sets, teams can identify when apparent improvements come at the cost of newly introduced errors or degradations elsewhere in the system. In contrast to ad hoc or manual review, this rigorous practice enforces a culture of accountability and provides a quantitative foundation for decision making. Ultimately, it is the careful curation and continual extension of evaluation sets—matched to both legacy and emerging features—that enables teams to maintain trust in their metrics and ensures that agentic systems are truly advancing toward their intended goals.

Creating and Scaling Evaluation Sets

The foundation of any measurement strategy is a high-quality evaluation set—one that reflects the diversity, ambiguity, and edge cases the system will face in the real world. Static, hand-curated test suites are insufficient for modern agentic systems: they risk overfitting, miss long-tail failure modes, and can't keep pace with evolving workflows and user behaviors.

A good evaluation set defines both the input state and the expected outcome, enabling automated validation of agent behavior. Consider this illustrative example from a customer support agent, which extends our cracked mug scenario, now with multiple items:

```
{
  "order": {
    "order_id": "A89268",
    "status": "Delivered",
    "total": 39.99,
    "items": [
      {"sku": "MUG-001", "name": "Ceramic Coffee Mug",
       "unit_price": 19.99},
      {"sku": "TSHIRT-S", "name": "T-Shirt-Small", "qty
       "unit_price": 20.00}
    ],
    "delivered_at": "2025-05-15"
  },
  "conversation": [
    {"role": "customer", "content": '''Hi, my coffee mu
        get a replacement or refund?'''},
    {"role": "assistant", "content": '''I'm very sorry
        please send us a quick photo of the damage so w
        refund?'''},
    {"role": "customer", "content": "Sure, here's the p
  ],
  "expected": {
    "final_state": {
      "tool_calls": [
        {"tool": "issue_refund", "params": {"order_id":
         "amount": 19.99}}
      ],
      "customer_msg_contains": ["been processed", "busi
    }
  }
}
```

This single example tests several things at once. It verifies whether the agent can reason correctly over multi-item orders, match conversational context to tool use, and produce human-friendly confirmations. Evaluation metrics such as tool recall, parameter accuracy, and phrase recall quantify these behaviors. If the agent instead refunded the entire order or failed to include appropriate language in its final message, those metrics would reflect the error—providing precise, actionable signals for improvement.

By formalizing evaluation examples in a structured format—including input state, conversation history, and expected final state—teams can automate scoring and aggregate metrics across a wide variety of scenarios. This format scales well. Once established, new examples can be added by hand, mined from production logs, or even generated using foundation models. Language models can be prompted to introduce ambiguity, inject rare idioms, or mutate working examples into edge cases. These model-generated samples can then be reviewed and refined by humans before inclusion in the test set.

To push the boundaries further, teams can apply targeted generation techniques such as adversarial prompting (e.g., "Find a user message that causes the agent to contradict itself"), counterfactual editing (e.g., "Change one word in the prompt and see if the agent fails"), or distributional interpolation (e.g., "Blend two intents to create a deliberately ambiguous request"). These strategies uncover subtle errors and probe the robustness of agent behavior.

In domains with access to real-world data, such as customer support logs or API call traces, domain-specific mining provides another rich source of evaluation material. Meanwhile, standard benchmarks like MMLU, BBH, and HELM can help contextualize performance relative to broader trends in the field, even as custom benchmarks remain essential for domain-specific agents.

Over time, a well-structured evaluation set becomes more than a test suite—it becomes a living specification of what the agent is expected to handle. It supports regression detection, enables continuous monitoring, and drives real progress by ensuring that agent behavior is improving not only on average, but in the places that matter most. This approach transforms evaluation from a static gatekeeping function into a dynamic, model-driven feedback loop that directly shapes the trajectory of system development.

For novel domains, teams should invest in custom benchmark creation, often pairing engineers with subject matter experts to define tasks, ground truth, and success criteria. This includes metadata for downstream analysis, such as failure type tagging or coverage tracking.

Regular evaluation against this continuously evolving evaluation corpus provides a scalable way to detect regressions, surface systemic weaknesses, and quantify improvements with statistical rigor.

This approach transforms evaluation from a static question-answer gate into a dynamic, model-driven feedback loop.

Component Evaluation

Unit testing is a fundamental practice in software development and is critical for validating the individual components of agent-based systems. Effective unit tests ensure that each part of the system functions as intended, contributing to the overall reliability and performance of the agent.

Evaluating Tools

Tools are the core functions that empower agents to act on their environment, retrieve or transform data, and interact with external systems. High-quality unit testing for tools begins with exhaustive enumeration of use cases, encompassing not only the typical "happy path" but also rare, adversarial, or malformed scenarios that could reveal brittle edges or hidden assumptions.

A mature agent development process defines a suite of automated tests for every tool. For instance, a data retrieval tool should be tested across different data formats, varied network conditions, and with both valid and intentionally corrupted data sources. Testing should explicitly validate not just the correctness of outputs but also latency, resource consumption, and error handling—ensuring that the tool degrades gracefully under load or failure.

Tool tests should assert that outputs are deterministic for identical inputs unless stochasticity is part of the tool's design (in which case, statistical properties must be checked). For tools with external dependencies, such as APIs or databases, developers should use mocks or simulators to reproduce edge cases that might be rare in production but catastrophic if mishandled. Regression tests are critical; every time a tool is modified, the full suite of tests must be rerun to verify that past capabilities have not broken.

Evaluating Planning

Planning modules transform high-level goals into actionable sequences of steps—often involving dynamic decision making, branching logic, and adaptation to environmental feedback. Unlike traditional scripts, agentic planning is often probabilistic or adaptive, requiring careful testing to avoid brittle or inconsistent behaviors. A planner might need to sequence tool calls, coordinate conditionals, or stop early depending on what it learns during execution. This makes validation both more subtle and more essential.

To assess planning quality, we begin with canonical workflows: common, well-understood user intents paired with known-good agent responses. For each scenario, we encode the starting environment, a conversation history, and the expected outcome in terms of tool usage and user communication. In the case of our customer support agent, for example, when a customer requests a refund for a damaged mug, the planner should determine that issuing a refund is the right action, not canceling the order or modifying an address. It should also include a confirmation message in natural language that reassures the customer that the issue has been resolved.

To evaluate these plans systematically, we run the agent end-to-end and extract its chosen actions. Specifically, we capture the list of tool invocations and their arguments from the agent's generated outputs. These are compared against the ground truth expectations for the scenario. From this comparison, we compute several automated metrics:

Tool recall

Did the planner include all expected tool invocations?

Tool precision

Did it avoid calling tools that were unnecessary?

Parameter accuracy

For each tool, did it supply the correct arguments—such as the specific order ID or refund amount?

These metrics provide fine-grained insight into the planner's behavior. A low recall score might indicate the planner failed to take an essential action, while low precision suggests it misunderstood the goal or misread the user's intent.

Parameter mismatches can highlight failures of contextual grounding—such as refunding the wrong item or issuing a refund for an order that was delivered successfully:

```
def tool_metrics(pred_tools: List[str], expected_calls:
    expected_names = [c.get("tool") for c in expected_c
    if not expected_names:
        return {"tool_recall": 1.0, "tool_precision": 1
    pred_set = set(pred_tools)
    exp_set = set(expected_names)
    tp = len(exp_set & pred_set)
    recall = tp / len(exp_set)
    precision = tp / len(pred_set) if pred_set else 0.0
    return {"tool_recall": recall, "tool_precision": pr

def param_accuracy(pred_calls: List[dict], expected_cal
    if not expected_calls:
        return 1.0
    matched = 0
    for exp in expected_calls:
        for pred in pred_calls:
            if pred.get("tool") == exp.get("tool")
                and pred.get("params") == exp.get("para
                matched += 1
                break
    return matched / len(expected_calls)
```

Because planning often depends on context, it is especially important to test edge cases. What if the order contains multiple items, and only one is defective? What if the user provides ambiguous input or contradicts themselves across messages? Tests should cover these situations to ensure the planner can navigate ambiguity and recover from intermediate failures.

Planning modules should also be evaluated for consistency. In deterministic scenarios, the same input should produce the same output; in probabilistic cases, the range of plans should still fall within acceptable bounds. Tests can check for reproducibility, sensitivity to small input changes, and graceful handling of unexpected conditions—such as missing fields in an order object or failed tool execution.

Over time, we maintain a growing corpus of planning scenarios that reflect the full range of what the agent must support—from simple, single-step flows to complex multiturn dialogues involving multiple interdependent actions. This corpus becomes the backbone of integration testing for planning. By continuously evaluating planning behavior as the system evolves, we detect regressions early and ensure that new capabilities do not introduce instability or drift.

Ultimately, planning evaluation tells us whether the agent knows what to do. It confirms that the agent not only understands user intent but can convert that intent into precise, coherent, and contextually grounded actions. As the bridge between perception and execution, planning must be scrutinized carefully—because everything downstream depends on it.

Evaluating Memory

Memory is essential for agents that need continuity and contextual awareness, whether for multiturn conversations, long-running workflows, or persistent user profiles. Testing memory modules is nontrivial, as it involves not only verifying raw storage and retrieval but also ensuring data integrity, relevance, and efficiency as the memory store grows.

Unit tests for memory should first verify that data written to memory is accurately stored and can be precisely retrieved, both immediately and after significant time has elapsed or other operations have intervened. This includes boundary cases such as maximum memory capacity, unusual data types, or rapid-fire read/write cycles. Tests should intentionally stress the system with malformed, duplicate, or ambiguous entries to ensure robustness:

```
def evaluate_memory_retrieval(
    retrieve_fn: Any,
    queries: List[str],
    expected_results: List[List[Any]],
    top_k: int = 1) -> Dict[str, float]:
    """
    Given a retrieval function `retrieve_fn(query, k)`
    k memory items, evaluate over multiple queries.
    Returns:
      - `retrieval_accuracy@k`: fraction of queries for
        expected item appears in the top-k.
    """
    hits = 0
    for query, expect in zip(queries, expected_results)
```

```
            results = retrieve_fn(query, top_k)
            # did we retrieve any expected item?
            if set(results) & set(expect):
                hits += 1
        accuracy = hits / len(queries) if queries else 1.0
        return {f"retrieval_accuracy@{top_k}": accuracy}
```

Beyond correctness, memory modules must be tested for relevance—ensuring that retrieval logic does not surface stale or irrelevant information. For instance, if the agent is asked for a user's recent preferences, the test must confirm that outdated or incorrect preferences are not returned due to data leakage or indexing errors. Tests should also check that irrelevant but similar data is not retrieved simply because of superficial similarity in phrasing or semantics.

Efficiency is a critical dimension, especially as memory size grows. Developers should benchmark retrieval times and resource usage under increasing memory loads, identifying any performance cliffs or bottlenecks. If vector search or semantic memory is used, tests should include scenarios with both "easy" and "hard" retrievals to catch subtle errors in embedding or indexing logic.

Finally, memory systems must be resilient to partial failures. Tests should simulate database unavailability, data corruption, or version migrations to ensure that the agent either recovers gracefully or fails in a controlled manner, with minimal user impact.

Evaluating Learning

Learning components are perhaps the most complex to unit test, given their stochastic nature and dependence on data. Nevertheless, rigorous testing is crucial to ensure that agents genuinely improve over time and do not simply overfit, regress, or "forget" previously mastered behaviors.

Testing learning begins with verification of the basic learning loop: does the agent correctly update its parameters, cache, or rules in response to labeled data, feedback, or reward signals? For agents employing supervised learning, unit tests should confirm that, when trained on a canonical dataset, the agent achieves expected accuracy and generalizes correctly to validation data. For reinforcement learning agents, tests should check that reward maximization leads to improved behavior over time, and that learning plateaus are detected and handled (e.g., through early stopping or dynamic exploration).

Generalization is paramount. Tests should evaluate how well the agent applies learned behaviors to novel, out-of-distribution scenarios. This includes “holdout” sets, synthetic examples, or adversarial test cases specifically constructed to challenge brittle heuristics or memorized responses.

Adaptability is also vital. Tests should simulate distribution shifts—such as new types of user inputs, previously unseen tool failures, or changing reward landscapes—and confirm that the agent can adapt without catastrophic forgetting or performance collapse. Where appropriate, learning modules should be tested across multiple paradigms (supervised, unsupervised, reinforcement), ensuring that cross-paradigm interactions do not introduce subtle bugs.

By rigorously testing these components—tools, planning, memory, and learning—developers can ensure that the foundational elements of the agent-based system operate reliably and effectively. This comprehensive approach to unit testing provides the confidence needed to build robust and scalable agents for real-world applications.

Holistic Evaluation

While unit tests validate the correctness of individual components in isolation, integration tests are designed to evaluate the agentic system as a whole, ensuring that all subsystems—tools, planning, memory, and learning—work together seamlessly in realistic settings. Integration testing exposes complex interactions, emergent behaviors, and end-to-end issues that cannot be predicted from unit testing alone. In agent-based systems, where the outputs of one module often become the inputs for another, integration tests are essential for surfacing problems that arise only during real-world use.

Performance in End-to-End Scenarios

The primary objective of integration testing is to validate the system’s ability to perform complete tasks from start to finish, under conditions that closely resemble actual usage. This involves constructing representative workflows or user journeys that exercise the full stack of the agentic system—perception, planning, tool invocation, and communication. For example, a customer support agent might be tested on multistep conversations that involve interpreting user requests, making decisions based on order data, calling business tools like `issue_refund` , and providing appropriate follow-up messages to the customer. These evaluations must ensure that the agent not only selects the right actions but also communicates clearly and stays aligned with user intent.

In our framework, this kind of evaluation is operationalized through an `evaluate_single_instance` function, which executes a complete test case and computes a set of metrics. The agent is given a structured input—including the order data and conversation history—and its outputs are compared against an expected final state. This includes checking which tools were called, with what parameters, and whether the final message includes required phrases. The results are summarized in metrics such as tool recall, tool precision, parameter accuracy, phrase recall, and an aggregate task success score. This makes it possible to assess the agent's full behavior—did it understand the situation, take the right actions, and explain them well? The following code is a helper function that executes an end-to-end integration test for a single scenario—invoking the agent on structured input and computing metrics for tool usage, parameter accuracy, phrase recall, and overall task success:

```
def evaluate_single_instance(raw: str, graph) -> Optior
    if not raw.strip():
        return None
    try:
        ex = json.loads(raw)
        order = ex["order"]
        messages = [to_lc_message(t) for t in ex["conve
        expected = ex["expected"]["final_state"]

        result = graph.invoke({"order": order, "message

        # Extract assistant's final message
        final_reply = ""
        for msg in reversed(result["messages"]):
            if isinstance(msg, AIMessage)
                and not msg.additional_kwargs.get("tool
                final_reply = msg.content or ""
                break
        # Collect predicted tool names and arguments
        pred_tools, pred_calls = [], []
        for m in result["messages"]:
            if isinstance(m, AIMessage):
                for tc in m.additional_kwargs.get("tool
                    name = tc.get("function", {}).get('
                    args = json.loads(tc["function"]["a
                        if "function" in tc else tc.get
                    pred_tools.append(name)
                    pred_calls.append({"tool": name, "p

        # Compute and return metrics
        tm = tool_metrics(pred_tools, expected.get("too
        return {
            "phrase_recall": phrase_recall(final_reply,
                expected.get("customer_msg_contains", [
            "tool_recall": tm["tool_recall"],
            "tool_precision": tm["tool_precision"],
            "param_accuracy": param_accuracy(pred_calls
                                             expected.g
            "task_success": task_success(final_reply, p
        }
    except Exception as e:
        print(f"[SKIPPED] example failed with error: {e
        return None
```

This approach enables scalable, repeatable measurement of end-to-end agent behavior across dozens or hundreds of diverse scenarios. A critical limitation is that automated tests are only as good as the evaluation sets and metrics they employ. If test cases are too narrow or unrepresentative, agents may appear to perform well in offline testing yet fail in production. Similarly, overreliance on a small set of metrics can lead to “metric overfitting,” where systems are tuned to excel on benchmarks at the expense of broader utility. This is particularly common with text-based agents, where optimizing for a single score (such as BLEU or exact match) may incentivize formulaic or unnatural outputs that miss the true intent behind user requests.

The best practice is to treat evaluation as a living process, not a static checklist. Teams should regularly expand and refine test sets to reflect new features, real user behavior, and emerging failure modes. Incorporating feedback—from internal reviewers or pilot users—helps reveal blind spots that automated pipelines miss. Iterative refinement of both evaluation methods and metrics ensures that agents are measured against what truly matters for success in the target environment.

By structuring each evaluation as a complete interaction—from input state to agent outputs—we can track how well the system performs in real-world tasks, detect regressions over time, and surface weaknesses in planning, grounding, or communication. These tests can also be extended to capture latency, throughput, and behavior under load—ensuring that the system remains robust and responsive under realistic operating conditions. And in failure cases, we can validate whether the agent degrades gracefully: does it attempt fallback strategies or escalate the issue appropriately? In this way, integration testing becomes a rigorous and essential safeguard for deploying agentic systems with confidence.

Consistency

Consistency testing for agent-based systems is particularly challenging because these systems often rely on foundation models that are inherently probabilistic and nondeterministic. Unlike traditional systems, where deterministic behavior ensures the same outputs for identical inputs, LLM-powered agents may produce varied responses due to their probabilistic nature. As a result, consistency testing focuses on ensuring that the agent's outputs align with its inputs, remain coherent over extended exchanges, and reliably address the user's intended questions or tasks.

In our running example of the customer support agent, consistency testing ensures that responses to a cracked coffee mug refund request (e.g., `order_id A89268`) remain aligned across probabilistic variations, such as always requesting a photo of the damage before invoking the `issue_refund` tool, even if the user's phrasing differs slightly. For extended interactions, like evolving from a refund to an order cancellation (as in `cancel_1_refund`, where the order is delivered), the agent must proceed without contradicting prior statements on order status.

One key goal of consistency testing is to validate that the agent's responses remain aligned with the given input across diverse scenarios. This involves assessing whether the agent provides relevant and accurate answers that directly address the user's queries. Automated tools can help detect cases where responses deviate from the expected alignment. Automated validation systems can cross-check outputs against the input context to flag inconsistencies for further review.

Longer interactions introduce additional complexity, as performance may degrade over time. Agents must maintain logical progression across multiturn conversations, avoiding scenarios where their responses contradict earlier statements or stray from the topic at hand. For example, a customer service bot must preserve context throughout an interaction, ensuring that its responses are consistent with the user's earlier inputs and the overall goal of the exchange. Testing in this area often requires extended simulated conversations to evaluate the system's ability to sustain consistent performance over time.

A subtle risk is that automated evaluations can miss rare but critical edge cases, especially those arising from novel inputs or system interactions. Agents may "pass" all standard tests but still behave unpredictably when confronted with situations outside the test set's distribution. For this reason, ongoing manual inspection and periodic refreshment of evaluation data are vital.

Both automated and human reviews play essential roles in addressing these challenges. Human reviewers can assess nuanced inconsistencies and provide feedback on how well the agent adheres to the intended purpose of its responses. This process is particularly important for evaluating edge cases or ambiguous inputs where automated systems may fall short. At the same time, scalable validation can be achieved through LLM-based evaluation techniques. By using the same or related models for consistency checking, agents can assess their own outputs against expectations. Providing these evaluation models with few-shot examples of what constitutes a consistent and relevant response enhances their reliability.

Actor-critic approaches offer another valuable tool for consistency testing. In this framework, the "actor" generates responses, while the "critic" evaluates them against predefined criteria for alignment and relevance. While effective, these methods alone may not suffice for complex or highly dynamic scenarios. The combination of actor-critic evaluations with LLM-based assessments and human feedback creates a more comprehensive framework for identifying and addressing inconsistencies.

Consistency testing ultimately ensures that agent-based systems deliver outputs that are aligned, logical, and purposeful, even in the face of nondeterministic behavior. By leveraging a mix of automated validation, human oversight, and advanced evaluation techniques like actor-critic frameworks and LLM-driven assessments, developers can build systems that inspire trust and perform reliably in both short and long interactions. This approach addresses the unique challenges posed by LLM-based agents, ensuring their outputs meet the high standards required for real-world applications.

Coherence

Coherence testing ensures that an agent's outputs remain logical, contextually relevant, and consistent across the span of an interaction. For agents managing multistep workflows or sustaining extended dialogues, coherence is what enables seamless, intuitive exchanges. The agent must retain and appropriately use context—such as user preferences or previous actions—so that its responses build naturally on what has come before. This is especially critical in multiturn conversations, where the agent should reference prior information without prompting the user to repeat themselves.

For instance, in the cracked mug scenario from our running customer support agent example, coherence requires the agent to reference the initial damage report and photo upload when confirming a refund, avoiding lapses such as overlooking the multi-item order details (e.g., only refunding the mug from `order_id A89268` while ignoring other items). In more complex cases, like a modification request following a refund (as in `modify_2`), the agent must maintain logical flow by confirming address changes without introducing contradictions in the conversation history.

Testing for coherence involves simulating extended interactions, verifying that the agent maintains a consistent understanding of state, and that its actions follow a logical, goal-directed sequence. Contradictions or lapses—such as conflicting recommendations or overlooked dependencies—are flagged as coherence failures. In customer service, for instance, coherence tests ensure that an agent's responses logically address user questions and maintain professional, unambiguous communication.

Ultimately, coherence testing is vital for preserving trust, usability, and the practical value of agentic systems in real-world applications. By rigorously evaluating for logical flow, context retention, and contradiction avoidance, developers ensure that agents operate reliably—even as tasks grow in complexity or session length.

Hallucination

Hallucination in AI systems occurs when an agent generates incorrect, nonsensical, or fabricated information. This challenge is particularly significant in systems designed for knowledge retrieval, decision making, or user interactions, where accuracy and reliability are paramount. Addressing hallucination requires rigorous testing and mitigation strategies to ensure the agent consistently produces responses grounded in reality.

To mitigate this, developers should ground outputs in verifiable data using techniques like retrieval-augmented generation (RAG), which cross-references trusted sources to enhance factual accuracy, as seen in legal AI tools that reduce hallucinations compared with general models.

At its core, mitigating hallucination begins with ensuring content accuracy. This involves verifying that the agent's outputs are based on factual data rather than fabrications. Systems must be rigorously tested to cross-check their responses against trusted sources of information. For instance, a medical diagnostic agent should base its recommendations on verified clinical guidelines, while a conversational agent providing historical facts must rely on validated databases. Regular audits of the system's knowledge base and decision-making processes are critical to maintaining this standard of accuracy.

Data dependence is another critical factor in addressing hallucination. The reliability of an agent's outputs is directly tied to the quality of its data sources. Systems that rely on outdated, incomplete, or poorly vetted data are more prone to generating erroneous information. Testing processes must ensure that the agent consistently draws from accurate, relevant, and up-to-date sources. For example, an AI summarizing news articles should rely on credible, well-regarded publications and avoid unverified sources.

Feedback mechanisms are essential for detecting and addressing hallucination. These systems monitor the agent's outputs, flagging inaccuracies for review and correction. Human-in-the-loop feedback loops can be particularly effective, enabling domain experts to refine the system's responses over time.

In dynamic applications, automated feedback mechanisms can identify discrepancies between the agent's predictions and actual outcomes, triggering updates to models or data sources to improve reliability.

Mitigations for hallucinations have evolved to emphasize hybrid human-AI feedback loops, where domain experts collaborate with AI systems in real-time oversight—such as in crisis self-rescue scenarios—to refine outputs, reduce cognitive load on users, and correct fabrications before they propagate. This approach integrates automated detection with human judgment, enhancing reliability in high-stakes applications like healthcare or legal advice. Additionally, cost-aware evaluations are gaining traction, focusing on balancing hallucination reduction with inference expenses; for instance, frameworks now quantify "hallucination cost" through metrics that weigh accuracy improvements against computational overhead, enabling more efficient deployments without sacrificing performance.

By prioritizing content accuracy, enforcing data dependence, leveraging feedback mechanisms, and rigorously testing for diverse scenarios, developers can minimize the risk of hallucination and build agents that deliver reliable, grounded, and trustworthy outputs. This disciplined approach ensures that the system operates as a reliable partner in its intended domain, meeting user expectations and adhering to high standards of accuracy and integrity.

Handling Unexpected Inputs

Real-world environments are unpredictable, and agents must be robust in the face of unanticipated, malformed, or even malicious inputs. Integration tests in this area intentionally supply inputs that fall outside the training or design assumptions—such as unexpected data formats, slang or typos in user language, or partial failures of external services. The goal is to ensure that the agent neither crashes nor produces harmful outputs, but instead responds gracefully: by clarifying, declining, or escalating as appropriate.

In the context of our ecommerce agent, unexpected inputs could include malformed order IDs (e.g., a typo in "A89268" during a cracked mug refund) or ambiguous requests blending intents (as in `cancel_4_refund`, where a cancellation is requested for a delivered order), requiring the agent to clarify or escalate rather than proceeding with erroneous tool calls like `issue_refund`. Systematic testing with adversarial variations from our evaluation sets, such as injecting slang or partial failures in photo uploads, ensures graceful handling without leaking sensitive order information.

Effective integration testing covers not only random “fuzzing” of inputs but also systematic exploration of edge cases informed by historical incidents or adversarial analysis. For safety-critical applications, it is important to verify that, even under stress, the agent does not leak sensitive information, violate policy, or cause downstream failures. By continuously extending and refining these tests as the agent evolves, developers can build systems that are robust, trustworthy, and ready for the complexities of the real world.

Preparing for Deployment

As an agentic system matures, transitioning from development to deployment requires disciplined readiness checks and quality gates to ensure reliability and trustworthiness in production. Production readiness is more than passing tests—it is a holistic assessment of whether the system can perform its intended function safely, consistently, and efficiently in a real-world environment.

Establishing clear deployment criteria is the first step. These often include meeting quantitative performance thresholds on relevant evaluation sets, demonstrating stability under stress and edge cases, and validating that all core workflows behave as intended. In practice, teams should use structured checklists to confirm that all components—tools, planning, memory, learning, and integrations—have been rigorously tested and reviewed. Key criteria may include passing end-to-end integration tests, meeting latency and uptime targets, and verifying the absence of critical or high-severity bugs.

For our running customer support agent, deployment criteria might include achieving at least 95% tool recall on refund and cancellation scenarios (e.g., correctly invoking `issue_refund` for damaged items like the cracked mug in `order_id A89268`), with automated gates blocking promotion if regressions appear in multiturn tests like address modifications (`modify_5`). This process, combined with pilot monitoring for real-world variations, enables confident rollout while enabling rapid rollback if issues arise in production.

A critical mechanism for enforcing these criteria is the use of gating mechanisms. Gates are automated or manual checks that prevent promotion to production unless all requirements are satisfied. This might involve blocking deployment if any regression is detected on the latest evaluation suite, or requiring explicit approval from technical and product leads after a successful pilot or beta phase. Gates can be configured to escalate issues for human review when automated results are ambiguous.

Equally important is establishing a reliable process for rolling out new versions, monitoring for regressions post-launch, and enabling rapid rollback if unexpected issues arise. This is where the foundation of robust, offline evaluation pays dividends, providing the confidence that the deployed system will perform as expected while minimizing risks to users and the business.

By rigorously preparing for deployment and establishing clear quality gates, teams create a culture of accountability and excellence, ensuring that only agentic systems meeting the highest standards reach users.

Conclusion

Measurement and validation form the backbone of developing robust and reliable agent-based systems, ensuring they are ready to perform effectively in real-world scenarios. By defining clear objectives and selecting relevant metrics, developers create a structured foundation for assessing an agent's performance. Thorough error analysis uncovers weaknesses and informs targeted improvements, while multitier evaluations provide a holistic view of the system's capabilities, from individual components to full-scale user interactions.

As illustrated through our running example of the ecommerce customer support agent—handling everything from a simple cracked mug refund (`order_id A89268`) to complex cancellations and modifications—these measurement and validation practices ensure robust performance across diverse scenarios. By iteratively refining metrics and evaluation sets based on such threaded cases, teams can deploy agents that not only meet objectives but also adapt to evolving user needs, ultimately fostering trust and efficiency in real-world applications.

This layered and methodical approach ensures that agent-based systems achieve their performance goals, deliver a seamless and satisfying user experience, and maintain reliability even in dynamic and complex environments. Comprehensive unit and integration tests safeguard the integrity of core functionalities and system-wide behaviors, enabling developers to address potential issues before deployment.

Ultimately, diligent measurement and validation empower teams to deploy agent systems with confidence, knowing they can withstand the challenges of real-world operation while meeting user needs. By prioritizing these practices, developers not only enhance the quality and reliability of their systems, but also pave the way for meaningful contributions to their intended applications across diverse industries and use cases.