Chapter 11. Improvement Loops

In any sufficiently complex multiagent system, failure is not an anomaly—it's an inevitability. These systems operate in dynamic, real-world environments, interacting with diverse users, unpredictable inputs, and rapidly changing external data sources. Even the most well-designed systems will encounter edge cases, ambiguous instructions, and emergent behaviors that the original design didn't anticipate. But the real test of a system isn't whether it fails—it's how well it learns from those failures and improves over time. This chapter focuses on building feedback-driven improvement loops that enable agent systems to not only recover from failure but to evolve and refine themselves continuously.

Continuous improvement is not a single mechanism but an interconnected cycle of using feedback pipelines to aid in diagnosing issues, running experiments, and learning. First, failures must be observed, understood, and categorized through feedback pipelines that surface actionable insights. These pipelines combine automated analysis at scale with human-in-the-loop review to extract meaningful conclusions from raw telemetry data and real-world user interactions. Next, proposed improvements must be validated in controlled environments through experimentation frameworks like shadow deployments, A/B testing, and Bayesian Bandits. These techniques provide structured pathways for rolling out changes incrementally, minimizing risk while maximizing impact. Finally, improvements must be embedded into the system through continuous learning mechanisms, whether through immediate in-context adjustments or periodic offline retraining. To understand this cycle of continuous improvement, it's helpful to draw an analogy from reinforcement learning, where agents learn optimal behaviors through iterative interactions with their environment. See Figure 11-1.

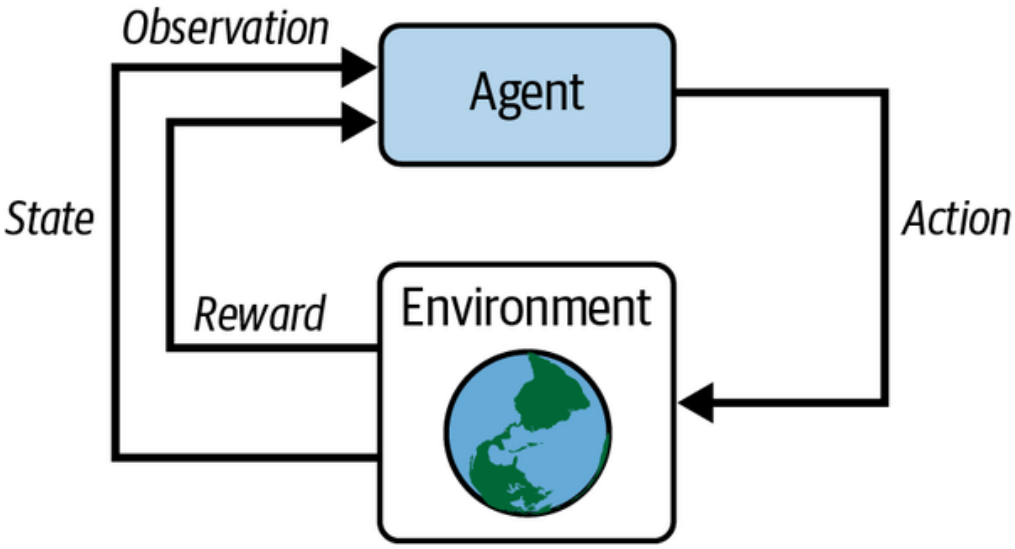

Figure 11-1. The interaction between an agent and its environment in a reinforcement learning system, showing how the agent receives observations, takes actions, and receives rewards and new observations from the environment.

Many teams rely on pretrained foundation models without directly training their agents—and often lack structured improvement loops altogether. This chapter explores how to close that gap by implementing feedback-driven mechanisms that enable agents to adapt and refine over time based on real-world interactions with their environment. Fine-tuning, as we discussed in Chapter 7, is an effective way to close this loop, but in this chapter, we'll discuss a wider range of techniques beyond fine-tuning.

However, improvement is not purely a technical challenge—it's also an organizational one. Effective improvement loops require alignment across engineering, data science, product management, and UX teams. They require systems for documenting insights, prioritizing improvements, and safeguarding against unintended consequences. Most importantly, they require a culture of curiosity and iteration—one that sees every failure as a valuable source of information and every success as a foundation for further refinement.

This chapter breaks down continuous improvement into three core sections. The first section explores the architecture of feedback pipelines, detailing how to collect, analyze, and prioritize insights from both automated tools and human reviewers. Next, I'll delve into experimentation frameworks, explaining how techniques like shadow deployments and A/B testing can validate proposed changes in low-risk environments. Then I'll cover continuous learning, showing how systems can adapt dynamically through in-context strategies and periodic offline updates. Table 11-1 provides an overview of what we'll cover.

Table 11-1. Improvement loop methodologies

Technique	Purpose	Strengths	Limitations	W
Feedback pipelines	Observe, analyze, and prioritize issues from interactions to generate actionable insights	Scalable data handling; blends automation and human oversight; proactive risk detection; basis for improvement cycles	Depends on data quality; may overlook highly novel issues without escalation	Fo dia fai spe pat bu im ba sui hig co sy
Experimentation	Validate changes in controlled settings, measure impact, and reduce risk predeployment	Data-driven; minimizes risks; enables variant comparisons; adapts to real conditions	Needs ample data for significance; resource-heavy; unsuitable for ultra-high-risk without gates	Fo im ide inc rol co or en ne fe
Continuous learning	Embed dynamic adaptations based on interactions and evolving needs	Real-time adaptability; addresses user changes; enhances resilience; supports personalization	Overfitting/regression risks; computationally costly; requires robust monitoring	Fo to pe or sy iss ra ch en or im ad

In the end, building a system that improves itself isn't just about fixing what's broken—it's about designing a workflow where every failure, insight, and

experiment becomes fuel for growth. This chapter provides the tools, strategies, and mindset required to ensure that agent systems adapt to changing circumstances.

Feedback Pipelines

Automated feedback pipelines are essential for handling the immense volume and complexity of data generated by multiagent systems operating at scale. These pipelines serve as the first line of analysis, continuously monitoring interactions, detecting failure patterns, and clustering issues to surface actionable insights. By leveraging optimization frameworks like DSPy (Declarative Self-Improving Language Programs), Microsoft's Trace, and Automatic Prompt Optimization (APO), alongside observability tools, these systems can operate with fine-grained visibility into agent behavior, tool usage, and decision-making pathways while enabling automated refinements.

The core function of automated feedback pipelines is to systematically identify recurring issues across agent workflows. For example, repeated failures in skill selection might indicate a misalignment between user intent and the agent's reasoning process, while consistent errors in tool execution might reveal ambiguities in how tool parameters are being generated. Automated systems excel at pattern recognition across vast datasets, clustering similar failure cases together to make trends apparent and actionable. Instead of relying on engineers to comb through raw logs and traces, automated pipelines distill these patterns into digestible insights, flagging high-impact issues for immediate attention.

Figure 11-2 illustrates a typical automated prompt optimization loop, as employed by frameworks like DSPy and APO. In this process, an initial prompt is fed into a target model, which generates outputs evaluated against a dataset by an evaluation model. The resulting scores inform an optimization model, which iteratively refines and proposes new prompts to improve performance. This approach enables continuous, data-driven enhancements without manual intervention, making it a cornerstone of scalable feedback pipelines in agentic workflows.

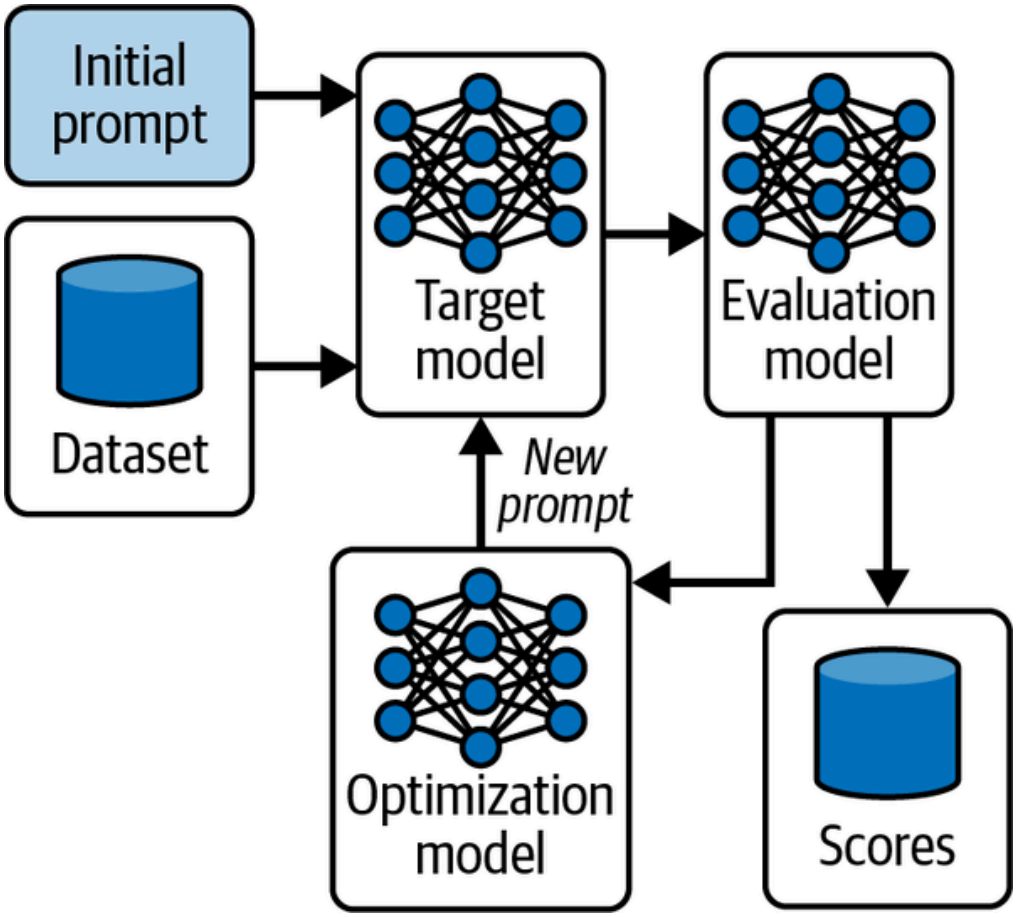

Figure 11-2. Automated prompt optimization pipeline, showing the flow from initial prompt through target and evaluation models, with an optimization model generating refined prompts based on dataset-driven scores.

Automated feedback pipelines, powered by tools like DSPy, Trace, and APO, transform raw observational data into iterative improvements, ensuring that multiagent systems remain robust and adaptive. We'll now discuss several of these approaches in more depth. DSPy is an open source Python framework developed by researchers at Stanford NLP for automatically optimizing and improving systems using foundation models. Unlike traditional prompt engineering, which relies on manual trial and error, DSPy treats language model (LM) pipelines as modular, declarative programs that can be systematically refined using data. Developers define "signatures" (input/output specifications for tasks), compose them into modules (e.g., chain of thought or ReAct for reasoning and tool use), and apply optimizers (like BootstrapFewshot or MIPROv2) to automatically generate better prompts and few-shot examples and even fine-tune model behaviors based on a dataset of examples and a metric (e.g., exact match or semantic similarity). This data-driven approach enables self-improving loops, where insights from failure patterns are backpropagated to enhance prompts, tools, or reasoning strategies —ideal for proactive optimization in agentic systems. DSPy integrates with popular LM APIs (e.g., OpenAI, Anthropic) and supports multistage compilation for complex workflows.

Complementing DSPy, Microsoft's Trace is an open source framework for generative optimization of AI systems. It enables end-to-end training and refinement of AI agents using general feedback signals (e.g., scores, natural language critiques, or pairwise preferences) rather than requiring gradients or differentiable objectives. By treating optimization as a generative process, Trace uses a foundation model to propose and evaluate improvements iteratively, making it suitable for black box systems where traditional methods fall short. This is particularly useful for refining agent behaviors in dynamic, multistep environments, such as incorporating feedback from clustered errors to evolve reasoning strategies or tool invocations over time.

To illustrate the concepts in this section, we'll use a running example of a Security Operations Center (SOC) analyst agent built with LangGraph. This agent handles cybersecurity tasks like investigating threats, analyzing logs, and triaging incidents. Its core components include a system prompt guiding the agent's methodology, tools for actions like querying logs or isolating hosts, and a workflow that invokes a foundation model (e.g., GPT-5) bound to those tools. Here's a simplified excerpt of the agent's system prompt and a tool definition:

```
You are an experienced Security Operations Center (SOC)
in cybersecurity incident response.
Your expertise covers:
- Threat intelligence analysis and IOC research
- Security log analysis and correlation across multiple
- Incident triage and classification (true positive/fal
- Malware analysis and threat hunting
- Network security monitoring and anomaly detection
- Incident containment and response coordination
- SIEM/SOAR platform operations

Your investigation methodology:
  1) Analyze security alerts and gather initial indicat
  2) Use lookup_threat_intel to research IPs, hashes, U
  3) Use query_logs to search relevant log sources for
  4) Use triage_incident to classify findings as true/f
  5) Use isolate_host when containment is needed to pre
  6) Follow up with send_analyst_response to document f

Always prioritize rapid threat containment and accurate
```

Our agent has several tools defined here:

```
@tool
def lookup_threat_intel(indicator: str, type: str, **kv
    """Look up threat intelligence for IP addresses, fil
    URLs, and domains."""
    print(f'''[TOOL] lookup_threat_intel(indicator={indi
        type={type}, kwargs={kwargs})''')
    log_to_loki("tool.lookup_threat_intel", f"indicator=
    return "threat_intel_retrieved"

@tool
def query_logs(query: str, log_index: str, **kwargs) ->
    """Search and analyze security logs across authentic
    firewall, and DNS systems."""
    print(f"[TOOL] query_logs(query={query}, log_index={
          kwargs={kwargs})")
    log_to_loki("tool.query_logs", f"query={query}, log_
    return "log_query_executed"

@tool
def triage_incident(incident_id: str, decision: str, re
    """Classify security incidents as true positive, fal
    for further investigation."""
    print(f'''[TOOL] triage_incident(incident_id={incide
     decision={decision}, reason={reason},
     kwargs={kwargs})''')
    log_to_loki("tool.triage_incident", f"incident_id={i
                decision={decision}")
    return "incident_triaged"

@tool
def isolate_host(host_id: str, reason: str, **kwargs) -
    """Isolate compromised hosts to prevent lateral move
    and contain security incidents."""
    print(f"[TOOL] isolate_host(host_id={host_id}, reasc
          kwargs={kwargs})")
    log_to_loki("tool.isolate_host", f"host_id={host_id}
    return "host_isolated"

@tool
def send_analyst_response(incident_id: str = None, mess
    """Send security analysis, incident updates, or recc
    stakeholders."""
    print(f"[TOOL] send_analyst_response → {message}")
```

```
    log_to_loki("tool.send_analyst_response", f"incident
                message={message}")
    return "analyst_response_sent"

TOOLS = [
    lookup_threat_intel, query_logs, triage_incident, is
    send_analyst_response
]
```

In a real deployment, this agent processes alerts like “Suspicious login attempt from IP 203.0.113.45.” Over time, as threats evolve (e.g., new attack vectors emerge), user queries shift, or external data sources change, the agent may encounter failures—such as misinterpreting queries, selecting suboptimal tools, or generating inaccurate triages. This is where feedback pipelines come in: they detect these issues, analyze root causes, and drive refinements. For instance, “drift” might occur if the agent’s prompt assumes outdated threat patterns (e.g., focusing on IP-based logins when attackers shift to credential stuffing), leading to repeated false negatives. Human engineers can fix this by refining prompts to include updated examples or adding validation steps in tools.

Automated feedback pipelines are essential for handling the immense volume and complexity of data generated by multiagent systems operating at scale. These pipelines serve as the first line of analysis, continuously monitoring interactions, detecting failure patterns, and clustering issues to surface actionable insights. By leveraging observability tools like Trace, DSPy, and similar frameworks, these systems can operate with fine-grained visibility into agent behavior, tool usage, and decision-making pathways.

One of the most powerful capabilities of modern feedback tools is their ability to back-propagate text-based feedback directly into the system’s prompts, skill parameters, and reasoning strategies. For example, if analysis reveals that certain task instructions frequently lead to ambiguous outputs, the pipeline can suggest refinements to the relevant prompts—tightening wording, adjusting constraints, or reordering steps in the reasoning process. Similarly, if tool invocations repeatedly fail due to malformed parameters, automated systems can recommend adjustments to how those parameters are constructed, including introducing validation steps or dynamic fallbacks.

Beyond reactive improvements, automated pipelines also support proactive optimization. By continually analyzing incoming data, they can surface areas

of latent risk before they manifest as critical failures. For example, early detection of drift in user query patterns can trigger prompt adjustments to ensure agents remain aligned with evolving user expectations. These proactive insights enable teams to address potential issues before they cascade into larger problems.

However, automated pipelines are not infallible. While they excel at identifying patterns and proposing changes, they cannot fully account for contextual nuances or prioritize improvements based on broader strategic goals. This is where human oversight becomes crucial—engineers must review, validate, and, when necessary, override the recommendations made by these systems. Automated pipelines, therefore, serve not as replacements for human insight but as powerful amplifiers, enabling engineers to focus their expertise where it matters most.

In essence, automated feedback pipelines create a scalable, self-improving loop: they observe, cluster, analyze, and propose improvements across prompts, tools, and reasoning flows. By efficiently managing failure data and generating actionable insights, these systems form the foundation of a robust feedback-driven development cycle, empowering multiagent systems to adapt and evolve continuously in response to real-world demands.

Automated Issue Detection and Root Cause Analysis

As agentic systems grow in complexity, manual monitoring and debugging quickly become unscalable. Automated issue detection and root cause analysis (RCA) are essential for identifying and diagnosing problems at speed and scale.

In our SOC agent example, imagine the system processes hundreds of alerts daily. Automated detection could flag a spike in failed `query_logs` calls where the query parameter is malformed (e.g., due to the agent generating overly complex SQL-like queries that the backend can't parse). Using tools like Trace, the pipeline logs each invocation, clusters similar errors (e.g., "invalid query syntax"), and correlates them with upstream reasoning steps in the agent's prompt.

Automated issue detection leverages a combination of rule-based triggers, anomaly detection algorithms, and statistical clustering to sift through massive volumes of logs and events. These systems can flag certain patterns:

- Repeated failures in a particular skill or tool
- Sudden spikes in error rates or response times
- Anomalies in user engagement or satisfaction metrics
- Divergent behavior across agent versions or deployment environments

Modern feedback pipelines often employ ML or statistical techniques to detect subtle trends that might otherwise go unnoticed—such as gradual drift in agent decision patterns, or correlations between specific user inputs and downstream failures.

Once an issue is detected, RCA seeks to answer not just *what* failed, but *why*. RCA is more than postmortem debugging; it is an ongoing, iterative inquiry into the relationships between user intent, agent reasoning, system architecture, and the external environment. Effective RCA typically follows several steps:

Workflow tracing

Reconstruct the end-to-end chain of agent decisions, tool invocations, and user interactions leading up to the failure.

Fault localization

Isolate the precise component—such as a misinterpreted prompt, an inappropriate skill selection, or a tool with restrictive parameter logic—responsible for the breakdown.

Pattern recognition

Identify whether the failure is an isolated incident or part of a recurring trend, potentially linked to specific user cohorts, data inputs, or system states.

Impact assessment

Evaluate the frequency and severity of the issue to prioritize response.

Critically, RCA in agentic systems often reveals that failures are not purely technical—they may stem from ambiguous task definitions, gaps in training data, or evolving user expectations that the system was not designed to handle. In some cases, RCA uncovers organizational blind spots, such as success metrics that incentivize the wrong behaviors or workflows that no longer match user needs.

Actionable RCA does more than assign blame; it surfaces opportunities for meaningful system improvement—whether through prompt or tool refinement, skill orchestration changes, or even rethinking the way user needs are represented and communicated.

A robust feedback pipeline, anchored by automated issue detection and RCA, shifts teams from endless triage to a disciplined, insight-driven process where every failure is mined for learning. It is the first step in turning telemetry into transformation—laying the groundwork for all subsequent cycles of experimentation and continuous learning in agentic systems.

Human-in-the-Loop Review

While automated systems excel at flagging anomalies and surfacing recurring patterns in multiagent workflows, there remain many situations where automated analysis alone is insufficient. Some issues—particularly those involving ambiguous user intent, ethical nuances, conflicting goals, or novel edge cases—require human intuition, domain expertise, and contextual judgment. Human-in-the-loop (HITL) review serves as a critical complement to automated detection and RCA, ensuring that feedback pipelines remain effective, comprehensive, and aligned with broader organizational goals.

For the SOC agent, HITL might escalate cases where automated RCA flags ambiguous triages (e.g., a “suspicious login” that could be a false positive from a virtual private network or a real breach). A security engineer reviews the trace, validates the prompt’s interpretation, and decides on fixes like adding ethical guidelines to the prompt (e.g., “Avoid isolating hosts without confirming impact on critical operations”).

Figure 11-3 depicts an HITL review workflow, where input data is processed by an agent to produce generated output candidates. These candidates undergo review by a human evaluator, who provides manual feedback to refine or approve them, resulting in human-approved outputs delivered to end users. System feedback from the review process loops back to enhance the agent’s performance, ensuring alignment with complex requirements that automation alone cannot handle. This structure highlights the integration of human judgment to address ambiguities and high-stakes decisions, as seen in the SOC agent’s escalation for nuanced threat assessments.

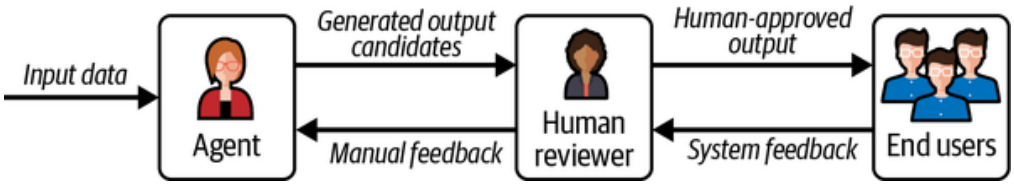

Figure 11-3. HITL review workflow, where input data flows through an agent that generates output candidates, to human review with manual feedback, culminating in approved outputs for end users supported by system feedback loops.

HITL review is not just a safety net for automation; it is a structured escalation process that brings human judgment to bear on the most complex, ambiguous, or high-impact system issues. Automated pipelines flag incidents that exceed predefined thresholds, exhibit unexplained patterns, or present unresolved conflicts—these are then routed for human evaluation. Escalation criteria may include:

- Persistent errors with no clear technical explanation
- Anomalies in workflows with regulatory or ethical implications
- Failures in high-value or mission-critical tasks
- Conflicting recommendations or diagnoses from automated tools

To find the right balance between human and AI decision making—ensuring humans focus on high-value interventions without being overwhelmed—escalation should prioritize cases with the least model certainty or the most consequential outcomes. For low-certainty cases, integrate confidence scores directly into the agent's outputs: many foundation models (e.g., GPT-5) can output a self-assessed certainty score (0–1) alongside responses by including instructions like "End your response with: certainty: [0–1 score based on confidence in accuracy]." Thresholds can be set (e.g., escalate if certainty < 0.7), or entropy measures used on probabilistic outputs (e.g., high entropy in classification logits indicates ambiguity). Variance across multiple runs (e.g., ensemble 3–5 inferences and escalate if outputs diverge > 20%) or external evaluators (e.g., a secondary foundation model critic scoring coherence) can further quantify uncertainty. In the SOC agent, low-certainty triages (e.g., a threat classification with score < 0.8) could auto-escalate for review, filtering out routine high-confidence cases.

For high-consequence cases, assess impact based on domain-specific severity: in the SOC agent, flag incidents with "high" severity ratings (e.g., potential data breaches) or those affecting critical assets (e.g., admin accounts). Combine this with risk scoring—e.g., multiply uncertainty by consequence (escalate if score > threshold)—to prioritize. Tools like DSPy can optimize these thresholds offline using historical data, simulating escalation rates to

balance load (e.g., aim for < 10% of cases escalated to avoid human fatigue). This hybrid approach ensures AI handles the bulk of routine decisions while humans intervene where judgment is most needed, fostering scalable, trustworthy systems. By defining clear escalation triggers, teams prevent automated systems from making inappropriate or myopic interventions and ensure that nuanced cases receive the attention they deserve.

When a case is escalated, a multidisciplinary review team—often including engineers, product managers, data scientists, and UX experts—systematically analyzes the flagged issue. The review process typically involves the following:

Contextual analysis

: Reproducing the failure or anomaly in a controlled environment to understand the sequence of events and decision points.

Trace inspection

: Examining logs, traces, and decision chains to clarify how the agent interpreted user intent and selected actions.

Impact assessment

: Evaluating the scope and severity of the issue, considering both technical correctness and UX.

Resolution design

: Recommending targeted interventions—ranging from prompt refinement to workflow redesign, new skill development, or even changes to user-facing features. In the SOC example, if drift causes over-isolation of hosts, humans might fix it by updating the `isolate_host` tool to include a confirmation step.

Effective HITL review protocols emphasize documentation and reproducibility. Decisions are logged, rationales are captured, and outcomes are tracked to ensure that future incidents can be resolved more efficiently and that systemic issues are identified over time.

HITL review often benefits from diverse perspectives beyond pure engineering. Product managers can clarify whether the observed failure reflects a deeper misalignment with user needs. Data scientists may recognize

patterns or edge cases invisible to others. UX researchers can surface friction points in user interactions that automated metrics might miss. This collaborative approach ensures that improvements are not just technically correct but are also meaningful and valuable for end users.

The ultimate value of HITL review lies in its contribution to organizational learning. Each reviewed case becomes a data point in an evolving knowledge base—a reference for training new team members, informing system design, and refining feedback loops. Lessons learned are fed back into prompt and tool refinement, skill development, and system documentation, reducing the recurrence of similar failures in the future.

By balancing automation with human oversight, HITL review ensures that multiagent systems remain both scalable and trustworthy. It transforms feedback pipelines from mere error correction mechanisms into engines of insight, resilience, and continuous improvement.

Prompt and Tool Refinement

Once feedback pipelines and HITL reviews have surfaced actionable insights, the next step is to implement targeted improvements. In agentic systems, the most direct and impactful levers for system refinement are the design of prompts (the instructions and context provided to language models) and the construction and invocation of external tools (functions, APIs, and actions the agent can use), so refining the prompt can be a very efficient way to improve the overall performance.

Prompt refinement

Prompts are the bridge between user intent and agent action. Subtle changes in prompt wording, structure, or context can dramatically affect an agent's interpretation, reasoning, and outputs. Feedback loops commonly reveal issues such as:

- Ambiguous instructions leading to inconsistent or irrelevant responses
- Overly broad prompts causing hallucination or off-task outputs
- Rigid, narrow prompts failing to generalize to real-world variability
- Lack of clarity around task boundaries, escalation, or error handling

Refinement begins with analysis: reviewing misfires, tracing agent reasoning, and isolating which part of the prompt contributed to undesired outcomes. Improvements might include:

Rewriting for clarity

Making instructions more explicit, reducing ambiguity, and specifying expected response formats

Adding exemplars

Providing positive and negative examples in the prompt to anchor agent reasoning

Decomposing tasks

Splitting complex multistep instructions into smaller, sequential prompts or intermediate reasoning stages

Context expansion

Incorporating additional context, constraints, or relevant background to guide the agent more effectively

DSPy excels at automating prompt refinement by compiling optimized prompts from a set of examples. For the SOC agent, we can use DSPy to refine the internal prompts of a ReAct module, improving how the agent handles alerts by better aligning reasoning and tool calls with expected responses. This is particularly useful for addressing issues like suboptimal tool selection or inconsistent outputs identified in feedback. Here's an example DSPy code snippet that optimizes a ReAct module for SOC incident handling using a small set of synthetic test cases (expand to 100+ annotated examples in practice for better results):

```
import dspy
dspy.configure(lm=dspy.OpenAI(model="gpt-4o-mini"))

def lookup_threat_intel(indicator: str) -> str:
    """Mock: Look up threat intelligence for an indicat
    return f"Mock intel for {indicator}: potentially ma

def query_logs(query: str) -> str:
    """Mock: Search and analyze security logs."""
    return f"Mock logs for '{query}': suspicious activi
```

```
# Handful of synthetic test cases (alert -> expected re
# In practice, derive from real logs or annotate failur
# aim for 100+ for better optimization
trainset = [
    dspy.Example(alert='''Suspicious login attempt from
                 admin account.''',
                 response='''Lookup threat intel for IP
                     triage as true positive, isolate h
                     .with_inputs('alert'),
    dspy.Example(alert="Unusual file download from URL
                 response='''Lookup threat intel for UR
                     for endpoint activity, triage as t
                     host.''').with_inputs('alert'),
    dspy.Example(alert="High network traffic to domain
                 response='''Lookup threat intel for do
                     network and firewall, triage as fa
                     benign.''').with_inputs('alert'),
    dspy.Example(alert='''Alert: Potential phishing ema
                 hash abc123.''',
                 response='''Lookup threat intel for ha
                     and endpoint, triage as true posit
                     response.''').with_inputs('alert')
    dspy.Example(alert='''Anomaly in user behavior: mul
                 new device.''',
                 response='''Query logs for authenticat
                     for device IP, triage as true posi
                     attack.''').with_inputs('alert'),
]

# Define ReAct module for SOC incident handling
react = dspy.ReAct("alert -> response", tools=[lookup_t

# Optimizer with a simple metric
# (exact match for illustration;
# use a more nuanced metric like
# semantic similarity in production)
tp = dspy.MIPROv2(metric=dspy.evaluate.answer_exact_mat
                  num_threads=24)
optimized_react = tp.compile(react, trainset=trainset)
```

This code optimizes the ReAct module's prompts (e.g., for reasoning steps and tool invocation) to better match the provided examples, effectively refining the agent's behavior without manual prompt tweaking. The resulting

`optimized_react` can be integrated into the SOC agent's workflow, leading to more reliable handling of diverse alerts and reducing issues like hallucinations or off-task outputs.

In advanced feedback systems, prompt adjustments can even be automated in response to observed failure patterns, though all changes should be validated—preferably in both offline testing and live shadow deployments—to prevent regressions or unintended side effects.

Tool Refinement

In modern agentic architectures, prompts alone rarely suffice. Agents increasingly rely on a suite of external tools—APIs, code functions, database queries, or custom skills—to retrieve information, perform transactions, or take concrete actions. Feedback pipelines frequently surface issues such as:

- Incorrect or suboptimal tool selection for a given user task
- Parameter mismatches or malformed inputs to tool calls
- Gaps in the toolset—tasks the agent cannot accomplish due to missing or incomplete tools
- Tool chaining failures, where the output of one step is not properly formatted for the next

Tool refinement is a multilevel process:

Refining internal logic

Optimizing prompts or models within tools to better process and classify data

Expanding capabilities

Enhancing tools to cover broader scenarios by incorporating optimized reasoning

Integration improvements

Ensuring tools output reliable, actionable results for the agent's needs

DSPy supports tool refinement by optimizing how tools are selected and chained within agent modules. Extending the previous example, suppose feedback reveals a gap in the toolset for incident triage (e.g., the agent often

skips classification steps, leading to suboptimal decisions). We can add a new mock tool for triage, update the ReAct module to include it, expand the trainset with examples emphasizing proper tool chaining, and reoptimize. This improves tool selection heuristics and integration, making the agent more robust to real-world variability. Here's the extended DSPy code:

```
import dspy

dspy.configure(lm=dspy.LM("openai/gpt-4o-mini"))

# Define a DSPy signature for the threat classificatior
class ThreatClassifier(dspy.Signature):
    """Classify the threat level of a given indicator (e
    'benign', 'suspicious', or 'malicious'."""
    indicator: str = dspy.InputField(desc="The indicator
    IP address, URL, or file hash.")
    threat_level: str = dspy.OutputField(desc="The class
    'benign', 'suspicious', or 'malicious'.")

# A DSPy module using ChainOfThought for reasoned class
class ThreatClassificationModule(dspy.Module):
    def __init__(self):
        super().__init__()
        self.classify = dspy.ChainOfThought(ThreatClassi

    def forward(self, indicator):
        return self.classify(indicator=indicator)

# Synthetic/hand-annotated dataset for optimization (ir
# examples from real SOC logs)
# Each example includes an indicator and the ground-tru
trainset = [
    dspy.Example(indicator="203.0.113.45",
        threat_level="suspicious").with_inputs('indicatc
    dspy.Example(indicator="example.com/malware.exe",
        threat_level="malicious").with_inputs('indicator
    dspy.Example(indicator="benign-site.net",
        threat_level="benign").with_inputs('indicator'),
    dspy.Example(indicator="abc123def456",
        threat_level="malicious").with_inputs('indicator
    dspy.Example(indicator="192.168.1.1",
        threat_level="benign").with_inputs('indicator'),
    dspy.Example(indicator="obfuscated.url/with?params",
        threat_level="suspicious").with_inputs('indicatc
    # Edge case: obfuscated URL
```

```
        dspy.Example(indicator="new-attack-vector-hash789",
        threat_level="malicious").with_inputs('indicator'),
    ]

    # Metric for evaluation (exact match on threat level
    # use semantic match or custom scorer for production)
    def threat_match_metric(example, pred, trace=None):
        return example.threat_level.lower() == pred.threat_l

    # Optimize the module (this refines the internal prompt
    # handling of diverse cases)
    optimizer = dspy.BootstrapFewshotWithRandomSearch(metri
        max_bootstrapped_demos=4, max_labeled_demos=4)
    optimized_module = optimizer.compile(ThreatClassificati
                                         trainset=trainset)

    # Example usage in the tool: After optimization, use ir
    def classify_threat(indicator: str) -> str:
        """Classify threat level using the optimized DSPy mo
        prediction = optimized_module(indicator=indicator)
        return prediction.threat_level
```

This refinement enhances the tool's ability to accurately classify threat levels from real API data, handling a wider range of responses—including no-results cases, partial matches, or emerging threats—by optimizing the foundation model's interpretation prompt.

Each prompt or tool refinement should be documented with a clear rationale —what problem was observed, what change was made, and how its effectiveness will be measured. This discipline ensures improvements are traceable and repeatable, and provides future teams with a knowledge base of what works and why.

Refinements should be validated iteratively, using both offline evaluation (with held-out logs or synthetic cases) and controlled live experiments (e.g., shadow deployments, A/B tests). Monitoring post-deployment performance is critical: even seemingly minor prompt tweaks can have system-wide effects, especially in complex or highly agentic environments.

Over time, the accumulated effect of systematic prompting and tool refinement is substantial. Agents become more reliable, less brittle, and better aligned with user needs. Feedback-driven refinement also reveals higher-level patterns—common sources of misunderstanding or recurring gaps in

capability—that can inform architectural improvements and future agent design.

Prompt and tool refinement are the hands-on instruments of progress in agentic systems. By connecting insight to action, and iterating thoughtfully, teams can ensure that every failure or friction point becomes an opportunity for more robust, responsive, and capable AI.

Aggregating and Prioritizing Improvements

As agentic systems grow in complexity and scale, so does the stream of actionable insights generated by feedback pipelines and human review. Teams quickly discover that not every bug, misfire, or enhancement can—or should—be addressed immediately. Without a system for aggregating and prioritizing improvements, teams risk being overwhelmed by noise, chasing low-impact fixes, or missing systemic problems in favor of surface-level symptoms.

The first step is aggregation: consolidating insights from multiple sources into a unified, accessible view. Feedback may originate from automated monitoring systems, RCA reports, user complaints, HITL reviews, or direct engineer observation. Aggregation platforms (such as centralized dashboards, observability tools, or structured issue trackers) help transform scattered data into a coherent improvement backlog. Key practices include:

Deduplication

Clustering similar issues together (e.g., recurring prompt failures or repeated tool invocation errors) to avoid fragmented effort

Tagging and categorization

Labeling issues by root cause, affected workflows, user impact, or system component for easier sorting and filtering

Linking to context

Attaching supporting logs, traces, user reports, and RCA documentation to each improvement for efficient triage and action

With a unified backlog in hand, the next challenge is prioritization. Not all improvements are created equal—some have outsized impact on system

reliability, user satisfaction, or business outcomes. Effective prioritization requires balancing several dimensions:

Frequency

How often does this issue occur? Frequent but minor issues can add up to significant user friction or operational overhead.

Severity/Impact

What is the business or user impact? Issues causing critical failures, security risks, or major dissatisfaction should rise to the top.

Feasibility

How difficult is the fix? Quick wins (low effort, high impact) are often prioritized, while complex improvements may require careful scoping or sequencing.

Strategic alignment

Does the improvement align with current product goals, upcoming features, or compliance requirements? Sometimes, a fix is essential not for its frequency but for its role in enabling a major initiative or regulatory milestone.

Recurrence and risk

Are similar failures likely to recur if not addressed? Systemic issues—those rooted in architecture, training data, or agent reasoning—should be flagged for deeper attention.

Prioritization frameworks—ranging from simple impact/effort matrices to more formal Agile or Kanban systems—can help teams reach consensus and adjust plans as system dynamics evolve. It's essential to treat the improvement backlog as a living artifact, not a static to-do list. Regular review cycles, "bug triage" meetings, and cross-team syncs ensure that priorities are continuously reevaluated in light of new incidents, shifting user needs, or strategic pivots. As improvements are implemented and validated, lessons learned should be fed back into the aggregation process—closing the loop and ensuring that recurring patterns inform future prevention.

The discipline of aggregation and prioritization turns the raw firehose of feedback into a clear, actionable roadmap. By focusing limited resources on the most impactful, feasible, and strategically aligned changes, teams can accelerate system evolution, build user trust, and prevent the accumulation of “technical debt” that can otherwise slow progress. In agentic systems, where the pace of change is rapid and the stakes are high, this process is not a luxury—it’s a necessity.

Experimentation

Experimentation is the engine of safe progress in multiagent systems. It serves as the bridge between insight and deployment, enabling teams to validate changes, measure their real-world effects, and mitigate risk before rolling out updates broadly. Given the complexity and interconnectedness of agentic architectures, even minor adjustments—such as tweaking a prompt, updating tool parameters, or refining orchestration logic—can produce far-reaching and sometimes unpredictable consequences. Without rigorous experimentation frameworks, teams risk introducing regressions, undermining reliability, or drifting away from user and business objectives.

A well-designed experimentation process provides a structured, incremental pathway for change. Rather than leaping straight from idea to production, changes are introduced and evaluated in controlled environments that closely mimic real-world conditions. This often begins with staging or release candidate (RC) environments—standard best practices where updates are tested in isolated, production-like setups to catch issues early without impacting live users. From there, teams can layer on advanced deployment techniques such as shadow deployments, canary rollouts (gradual exposure to a subset of traffic), rolling updates (incremental instance-by-instance upgrades), or blue/green deployments (switching between two identical environments). This approach not only uncovers unintended side effects early but also enables direct comparisons between alternative configurations, paving the way for data-driven decision making.

Shadow Deployments

Imagine rehearsing a play backstage while the live show runs uninterrupted—that’s shadow deployments in action. Here, your updated agent (e.g., with refined reasoning for query ambiguity) shadows the production version,

processing identical inputs in parallel. But only the live system's outputs reach users; those of the shadow are logged for scrutiny, shielding everyone from mishaps.

Shadow deployments are a powerful approach for validating system changes under real-world conditions—without exposing users to risk. This side-by-side comparison enables teams to observe, measure, and diagnose the behavior of new or updated agent logic under authentic operational loads. Shadow deployments are especially valuable for high-impact or high-risk changes—such as updates to planning workflows, integrations with external systems, or significant prompt modifications—where failures could have serious consequences if released unchecked. Key benefits include:

Realistic validation

Shadow systems experience the full spectrum of real user behavior, surfacing discrepancies and emergent issues that often elude controlled test environments.

Safe exploration

Engineers can experiment with bold improvements or architectural changes, confident that any errors, regressions, or performance degradations will not reach production.

Edge-case discovery

Rare or unpredictable scenarios—such as malformed user inputs, ambiguous instructions, or integration quirks—can be detected and analyzed before deployment.

Integration with complementary strategies

Shadow deployments can be aided by blue-green deployments (maintaining two identical environments and switching traffic only after validation) to enable seamless, zero-downtime rollouts after shadow testing, or canary deployments (gradually routing a small percentage of live traffic to the new version) to support incremental real-time validation in production, reducing overall risk and facilitating smoother transitions from testing to full deployment.

Quality instrumentation is key: compare traces, metrics (accuracy, latency), and outputs rigorously. Triangulate discrepancies to separate breakthroughs

from bugs.

Challenges arise in HITL-dependent agents (e.g., those querying users for approvals)—shadows can't interact without exposure risks. Simulate responses via historical replays or synthetics, or hybridize with staging or A/B testing for interactive flows.

In essence, shadows build confidence quietly, validating in the wild minus the stakes.

A/B Testing

If shadows are about observing from the sidelines, A/B testing thrusts variants into the spotlight—splitting live traffic between control (A) and treatment (B) versions for head-to-head showdowns. Users interact with one or the other, yielding quantifiable wins on metrics like task success in a collaborative agent swarm or reduced hallucinations in responses. This shines for measurable tweaks, such as running an A/B on prompt variants to optimize user satisfaction in real-time chats, where shadows might miss subtle engagement shifts. As seen in Figure 11-4, a common setup for A/B testing involves randomly assigning users to different agent variants to enable direct, real-world comparisons.

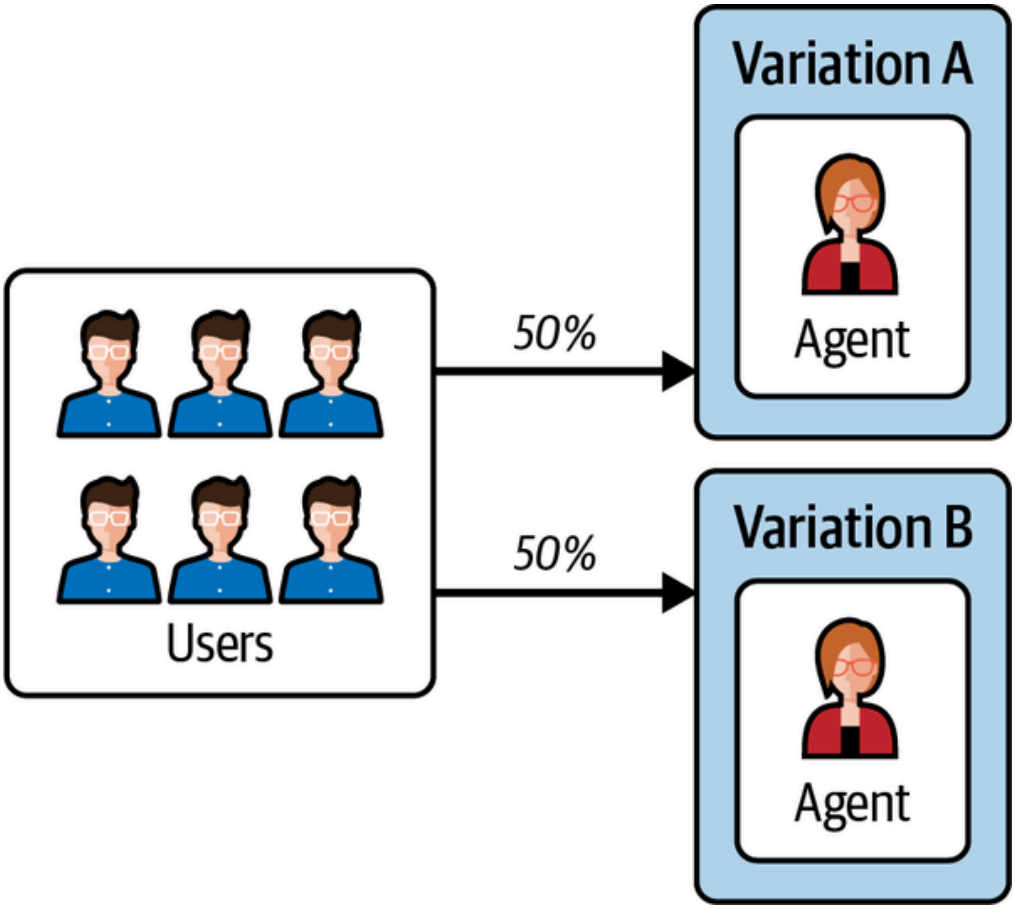

Figure 11-4. A/B testing for agent configurations, where users are split evenly (50/50) between two variations of an agent (A and B) to evaluate performance differences based on live interactions.

This balanced allocation ensures fair exposure and reliable metrics, enabling teams to confidently identify which variant performs better in practical scenarios. Strengths of A/B testing include:

Real-world relevance

Results reflect genuine user behavior and input diversity, providing strong evidence of whether changes generalize beyond isolated test cases.

Direct comparison

Teams can quickly determine which version delivers superior outcomes, under actual operational conditions.

Statistical rigor

Properly designed A/B tests ensure that observed differences are meaningful—not the result of random variation or biased sampling.

To maximize the value of A/B testing, teams should:

- Define clear, actionable metrics that align with the objectives of the proposed change.
- Ensure sufficient sample size to achieve statistical significance, minimizing the risk of false positives or negatives.
- Prevent cross-contamination (e.g., users switching between versions in a single session) to preserve result integrity.
- Monitor both short- and long-term effects, as some changes may yield quick gains but introduce longer-term issues.

Qualitative review remains important: a decrease in completion rate for version B, for example, may reflect deeper, more thoughtful engagement—rather than outright failure.

However, A/B testing can be more difficult when agents store long-term interaction states, such as chat histories or persistent user contexts, as users might experience inconsistencies if they are reassigned to different versions across sessions. To mitigate this, teams can implement "sticky" user assignments (ensuring users remain in the same variant over time), conduct tests at the session level rather than the user level, or isolate state management to prevent cross-version contamination—potentially by duplicating or versioning state stores for each test group.

Modern experimentation platforms (e.g., LaunchDarkly, Optimizely, or custom dashboards) automate much of the traffic allocation, metric collection, and analysis, freeing teams to focus on interpreting results and acting on insights.

Bayesian Bandits

What if your A/B test could learn on the fly, shifting users toward winning variants mid-experiment instead of rigidly splitting traffic? That's the power of adaptive experimentation, where Bayesian Bandits stand out as a smart upgrade for multiagent systems—dynamically balancing exploration (trying new ideas) with exploitation (sticking to what works) to accelerate improvements in unpredictable environments.

Picture a casino slot machine with multiple "arms" (levers), each offering unknown odds of payout. In Bayesian Bandits, each arm represents a system variant—like alternative prompts for an agent's query handling or different orchestration strategies in a multiagent swarm. As interactions unfold, the

algorithm observes rewards (e.g., successful task resolutions, lower latency, or higher user ratings) and uses Bayesian updates to refine its beliefs about each arm's performance. Over time, it funnels more traffic to promising arms while sparingly testing others, ensuring you don't miss hidden gems.

For a concrete agentic example, suppose you're optimizing an SOC (Security Operations Center) multiagent system, testing three reasoning chains for resolving ambiguous threat queries. The bandit starts evenly, but as data rolls in, say, one chain improves threat classification accuracy by 15%. It reallocates 70% of queries there, still probing the others for shifts in user behavior. This is especially potent in multiagent setups, where interactions can be computationally expensive or reveal emergent behaviors only under load. In fact, frameworks like Knowledge-Aware Bayesian Bandits (KABB) extend this to coordinate expert agents dynamically, using semantic insights to select subsets for tasks like knowledge-intensive queries. Some of the key advantages of Bayesian Bandits include:

Responsiveness

The system learns and shifts traffic allocations in near real time, reducing opportunity costs and accelerating improvements.

Efficiency

Rather than "wasting" equal traffic on suboptimal variants, the majority of users experience the best-performing configuration as soon as it is identified.

Scalability

Well-designed Bayesian Bandit systems can scale to very large numbers of parameters, enabling a much more rapid exploration of the action space than configurating and reviewing a series of fixed experiments.

However, adaptive experimentation also requires:

Metric mastery

Rewards must reflect true system goals (e.g., user satisfaction, task success) to avoid optimizing for misleading proxies.

Thoughtful initialization

Neutral priors and regularization help avoid biasing the system prematurely toward any variant.

Vigilant oversight

Teams must watch for pathological feedback loops or exploitation of short-term trends at the expense of long-term objectives.

Bayesian Bandits shine in fluid, data-rich agentic worlds—think real-time personalization in recommendation agents or adaptive workflows in autonomous teams—delivering faster, smarter evolution than traditional methods.

Continuous Learning

We now move to continuous learning, where the agentic system is designed to adapt, improve, and optimize performance over time based on real-world interactions, feedback, and evolving user needs. Unlike static models or prescripted workflows, continuously learning agents are designed to ingest new data, refine their behavior, and update reasoning strategies dynamically. This process blends automated adaptation with carefully managed oversight to prevent unintended consequences, such as overfitting to short-term trends or introducing regressions.

Continuous learning encompasses two core mechanisms: in-context learning and online learning. These enable improvements at varying scales—from real-time tweaks within a session to incremental updates across workflows. As discussed in Chapter 7, these build on foundational nonparametric techniques (e.g., exemplar retrieval, Reflexion) and can incorporate parametric methods like fine-tuning where appropriate. Here, we emphasize integrating them into improvement loops, using live production data (e.g., user interactions, telemetry, and failure logs) to tighten the cycle: feedback pipelines surface issues, experimentation validates fixes, and continuous learning embeds them for immediate or ongoing impact.

In-Context Learning

In-context learning offers the most immediate and flexible means of adaptation in foundation model–based systems. Rather than relying on model fine-tuning or architectural changes, in-context learning empowers agents to

modify their behavior dynamically within a single session. By embedding examples, intermediate reasoning steps, or contextual signals directly into prompts, agents can be "taught" new behaviors on the fly—adapting at runtime rather than depending solely on static, pretrained weights.

Consider an agent assisting users with code debugging. If the agent consistently struggles with a particular type of error, an engineer can revise the prompt to include an illustrative example that demonstrates the correct solution. This change takes effect instantly, scoped only to the current session, enabling the agent to improve its responses without requiring broader system retraining. Additionally, agents can leverage user feedback in real time—such as corrections or clarifications—to further refine their reasoning and adapt their next steps within the same interaction. Key strengths of in-context learning include:

User-specific adaptation

: Tailoring responses to individual user preferences or recurring issues, providing a personalized experience

Real-time feedback incorporation

: Dynamically adjusting behavior in response to user clarifications or follow-up instructions, enhancing responsiveness

Guided reasoning

: Integrating explicit reasoning steps or intermediate outputs to steer the agent toward more reliable or interpretable conclusions

A critical enabler of effective in-context learning is robust context management. Because foundation models have finite context windows, systems must carefully curate which information to include in prompts, how to structure it, and when to remove or compress outdated details. Techniques such as rolling context windows, semantic compression, and vector-based memory retrieval help ensure that the most relevant information remains accessible throughout an interaction.

However, in-context learning comes with inherent limitations. Changes made within a session are ephemeral—once the session ends, any learned adaptations are lost. To preserve valuable insights for future use, successful in-

context strategies should be promoted to more permanent mechanisms, such as prompt engineering, workflow updates, or full model retraining.

In practice, in-context learning often serves as a first line of adaptation—enabling rapid, low-risk testing of improvements in live interactions. It acts as a testing ground for new reasoning strategies or prompt structures before these approaches are codified into broader workflows or incorporated system-wide. This makes in-context learning especially useful for handling session-specific failures, rapidly iterating on small refinements, or addressing highly dynamic and unpredictable user inputs where traditional approaches may fall short.

The strengths of in-context learning lie in its instant adaptability, minimal risk, and ability to deliver session-level personalization and real-time refinement. However, these adaptations are inherently transient; any changes made are limited in scope and do not persist once the session concludes. As such, while in-context learning is ideal for rapid prototyping of refinements and responding to evolving or unpredictable user needs, valuable insights derived from these interactions must eventually be formalized through prompt engineering, workflow updates, or model retraining to achieve lasting improvement.

When thoughtfully integrated into a continuous learning pipeline, in-context learning provides not only a powerful mechanism for immediate improvement but also a vital foundation for scalable, longer-term system optimization.

Offline Retraining

Offline retraining represents a structured, periodic approach to embedding lasting improvements in agent systems, drawing on accumulated data from feedback pipelines and experiments. Unlike in-context adaptations, which are session-bound, offline retraining involves collecting batches of interaction data—such as user queries, agent outputs, and labeled outcomes—and using them to update prompts and tools or even fine-tune underlying models in a controlled, nonproduction environment. This method is particularly suited for addressing systemic issues identified over time, such as recurring misalignments in reasoning or tool usage, without disrupting live operations.

In the SOC agent example, suppose feedback reveals a pattern of false positives in threat triages due to evolving attack vectors. Teams can aggregate historical logs and annotations into a dataset, then use frameworks like DSPy

to optimize prompts or fine-tune a lightweight adapter on the base foundation model (as discussed in Chapter 7). The process typically follows these steps:

Data curation

Gather and label examples from production traces, ensuring diversity and balance to avoid bias.

Model updates

Apply techniques like few-shot optimization or full fine-tuning on held-out data, focusing on metrics like accuracy or latency.

Validation

Test the retrained components offline against benchmarks, then via shadow deployments before rollout.

Key strengths include:

Durability

Changes persist across sessions and users, providing long-term alignment with shifting environments.

Scalability

Batched updates are efficient for high-volume systems, enabling teams to incorporate large datasets without real-time overhead.

Risk mitigation

Offline nature enables thorough testing, reducing the chance of regressions.

However, offline retraining requires careful management to prevent overfitting to historical data or ignoring emerging trends. Limitations include computational costs (though mitigated by efficient methods like LoRA) and the need for periodic scheduling to keep models fresh. It's best used for foundational refinements, such as updating the SOC agent's prompt with new threat examples or retraining tool classifiers on recent logs.

When integrated with feedback and experimentation, offline retraining closes the improvement loop by translating insights into enduring enhancements. For

teams relying on pretrained models, it offers a bridge to customization without constant online adjustments, ensuring agents evolve robustly over time.

Conclusion

Continuous improvement is not merely a feature of multiagent systems—it is a fundamental requirement for their long-term success. As these systems grow more complex, interact with diverse users, and operate across ever-changing environments, their ability to adapt, learn, and refine themselves becomes essential for maintaining reliability, performance, and alignment with user needs. This chapter has explored the key pillars of continuous improvement: feedback pipelines, experimentation, and continuous learning, each playing a distinct yet interconnected role in driving iterative progress.

Feedback pipelines serve as the diagnostic engine of the improvement cycle, capturing data from live interactions, identifying recurring failure patterns, and surfacing actionable insights through both automated and human-driven processes. From root cause analysis to aggregating and prioritizing improvements, these pipelines create a systematic foundation for identifying *what* needs to change and *why*.

Experimentation frameworks provide the controlled environments necessary to validate improvements before full deployment. Techniques like shadow deployments, A/B testing, and Bayesian Bandits enable teams to minimize risk, measure impact, and ensure that every change contributes positively to the overall system.

Finally, continuous learning ensures that improvements extend beyond isolated patches, embedding adaptability directly into the system's behavior, focusing on in-context learning, which provides instant, session-level refinements.

Crucially, none of these components operate in isolation. Feedback loops feed into experimentation workflows, which in turn guide fine-tuning retraining. Automated pipelines accelerate insight generation, while human oversight ensures that changes are aligned with strategic goals. Documentation serves as the connective tissue across these processes, preserving organizational memory and enabling cross-team collaboration.

Continuous improvement is not a linear process—it's an ongoing cycle of observation, adjustment, validation, and deployment. As multiagent systems become more deeply integrated into critical workflows, the importance of robust feedback mechanisms, well-designed experiments, and adaptive learning processes will only grow. Organizations that invest in these capabilities will not only reduce failures and improve reliability—they will also unlock the ability to anticipate user needs, respond to emerging trends, and deliver meaningful innovation at scale.

In the end, continuous improvement is as much about systems design as it is about organizational culture. It requires a mindset that views every failure not as a setback but as a signal—an opportunity to learn, iterate, and evolve. By building systems that can observe themselves, learn from their behavior, and adapt with intention, teams can create agent ecosystems that don't just function—they thrive.