databricks

The Big Book of AgentOps

A comprehensive guide to deploying quality agentic applications safely and reliably.

From MLOps to LLMOps to AgentOps

PAVITHRA RAO | JEANNE CHOO | KYRA WULFFERT | ALEX BAUR

JUL 10, 2026

Contents

databricks

Introduction

Welcome to The Big Book of AgentOps, your comprehensive guide to deploying quality agentic applications safely and reliably.

What You'll Learn

In addition to the technical solutions, we will also cover the principles, people and processes needed for a successful agentic application. In this book, we cover:

- **Background and context:** Understanding what agents are and how they differ from traditional ML systems
- **Principles:** Applying DevOps principles to agentic systems
- **Implementation:** Bringing together people, processes, and technology for successful AgentOps
- **Observability and Agent Evaluation:** Use MLflow to obtain a lens into how your agent behaves, what it costs, and trace and track all Agent operations.

Who This Book Is For

The intended audience includes:

- AI engineers building production agentic systems
- Data engineers working with agentic pipelines
- Software engineers integrating AI agents
- Product managers overseeing AI initiatives
- Platform engineers deploying agent infrastructure
- Anyone interested in the operational realities of deploying agentic systems at scale

databricks

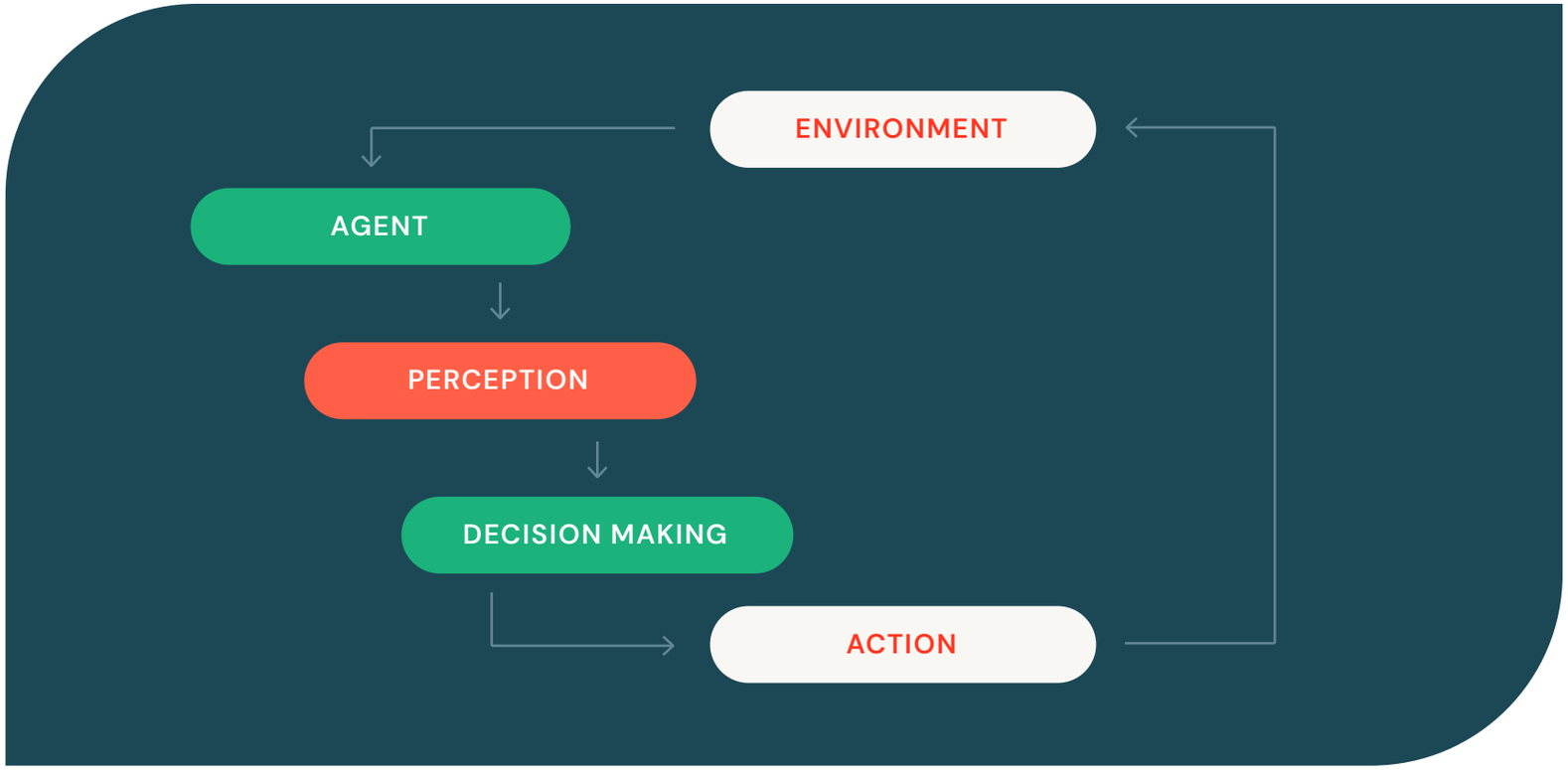
CHAPTER 1

Evolving Trends in the Industry

1.1 What Is an Agent?

An **agent** is an autonomous system that can perceive its environment, make decisions, and take actions to achieve specific goals. Unlike traditional software that follows predetermined paths, agents exhibit:

- **Autonomy:** Independent decision-making capabilities
- **Reactivity:** Ability to respond to environmental changes
- **Proactivity:** Goal-directed behavior
- **Social ability:** Interaction with other agents and humans



1.2 How Agents Differ From Other AI Systems

Agents differ from other AI systems in several ways. To understand these, it's important to understand the landscape of AI system architectures and their capabilities.

1.2.1 Core AI System Architectures

At a high level, modern LLM-centric AI systems can be viewed as falling into one of four core architectural patterns:

The following table provides a brief overview of the capabilities, limitations, and example use cases of each.

SYSTEM ARCHITECTURE	IMPLEMENTATION	CAPABILITIES	USE CASES	LIMITATIONS
LLM + prompt	Single model call with a prompt	Text generation, reasoning	Content creation	Static knowledge, no actions
Single tool-calling agent (e.g. RAG)	Fixed, scripted steps (retrieve → augment → generate)	Knowledge retrieval + generation	Document question-answer, over a knowledge base	Limited to retrieval
Multi tool-calling agent	LLM chooses which tool(s) to use at runtime	Dynamic reasoning + actions	Task automation	Complexity, reliability
Multi-agent	Router or coordinator orchestrates specialized sub-agents	Decomposition, parallelism, role specialization, escalation paths	Complex workflows and multi-domain support; supports triage across HR, IT, and Finance	Orchestration overhead, emergent loops, hard to debug and attribute

Note: **Agent harnesses** provide the runtime environment and scaffolding through which agents operate, managing capabilities such as context, tool access, execution loops, memory, and model interaction. The same agentic architecture may be implemented through different harnesses.

1.2.2 Operational Practices by System Architecture

As architectures grow in complexity, the operational capabilities required to maintain them must also mature. The following table shows how five critical operational capabilities evolve across the different architectural patterns:

- **Logging & tracing:** What you need to capture to understand system behavior
- **Evaluation gates:** Tests that must pass before deploying changes
- **Governance & policy:** Controls on who can change what
- **Deploy & rollback:** How you safely release changes
- **Monitoring & alerts:** What to watch in production

CONTROL	LLM + PROMPT	RAG	TOOL-CALLING AGENT	MULTI-AGENT
Logging & tracing	Single-step trace (prompt I/O, tokens, latency)	Multi-stage traces (query → retrieval → generation)	Per-tool spans (I/O, errors, retries)	Distributed traces (routing, handoffs, shared state)
Evaluation gates	Golden test set (20–100 examples)	RAG metrics (groundedness, citations)	Task success + side effects testing	Per-agent + end-to-end workflows
Governance & policy	Model/prompt change approval	Data source policies and index change control	Identity, tool permissions, runtime policies + guardrails	Cross-agent identity, permissions, delegation + runtime policies
Deploy & rollback	Optional canary, quick rollback	Canary for index/ prompt changes	Canary routes + trace replay	Orchestrator-level gates, per-agent versioning
Monitoring & alerts	Response drift, rate limits	Retrieval quality drift, index freshness	Tool failures/timeouts, quota alerts	Escalation rates, loop detection, per-role costs

1.3 From MLOps to AgentOps — How We Got to Today

As AI systems have evolved from predictive models to autonomous agents that take real-world actions, the operational practices — the tools, processes, and frameworks teams use to deploy and maintain these systems — have also had to evolve to accommodate new challenges and objectives. In the following section, we trace this evolution to see how we got to where we are today as an industry.

1.3.1 The Evolution of AgentOps

The evolution of AgentOps can be roughly divided into three epochs, with the focus of each shaped by the capabilities and requirements of the types of systems it was intended to support:

- **MLOps (2010s)**
  The set of processes and automation for managing data, code, and models to improve performance stability and long-term efficiency in ML systems. MLOps emerged as teams needed to move beyond experimental model training in notebooks to production-grade systems. It encompasses the entire ML lifecycle: data pipelines for feature engineering, model training and versioning, deployment infrastructure, monitoring for drift and performance degradation, and workflow orchestration to tie it all together.

  This requires unified governance for both data and models, scalable training infrastructure, continuous integration and deployment (CI/CD) pipelines for model deployment, low-latency model serving endpoints, and continuous monitoring to ensure models perform reliably over time.

- **LLMOps (2020s)**
  The practices and infrastructure required to tune, deploy, and manage large language models at scale. When models grew to billions of parameters, many teams faced new scale challenges: hosting models too large for single GPUs, managing distributed training across clusters, and controlling inference costs that could spiral out of control.

  LLMOps encompasses model optimization techniques like quantization and distillation, efficient tuning approaches using Low-Rank Adaptation (LoRA) and Quantized Low-Rank Adaptation (QLoRA), prompt versioning and management, and safety measures to prevent harmful or biased outputs. Databricks provides managed infrastructure for tuning foundation models, optimized serving endpoints that handle batching and caching, cost controls to prevent budget overruns, and guardrails to ensure outputs meet safety and compliance standards.

- **AgentOps (2024+)**
  The practice and tooling to build, evaluate, deploy, govern, monitor, and maintain autonomous AI systems that can reason, plan, and take actions. Unlike LLMs that operate in single request/response cycles, agents execute multi-step workflows using tools to interact with real systems, querying databases, calling APIs, modifying files, and executing code.

  This shift from single-step inference to active tool use introduced entirely new operational challenges: establishing permission boundaries for tools, tracing complex multi-step reasoning chains, preventing cascading failures when agents make mistakes, handling retry logic and error recovery, and orchestrating multiple agents without conflicts or infinite loops.

The core of AgentOps on Databricks is MLflow, which provides a unified layer for evaluating, observing, monitoring, and governing agent behavior. LLM-judge and rules-based scorers (via mlflow.genai.evaluate()) evaluate agent quality before deployment. MLflow Traces allow developers to observe and debug multi-step reasoning steps and tool calls, while also providing a feedback and monitoring layer that tracks quality and cost in production. Governance and cost control extend through the Unity AI Gateway (OSS Version MLflow AI Gateway), which routes every LLM call through a single control plane for traffic management, budget policies, and per-component cost attribution.

Finally, Agent Bricks, a managed platform that automatically builds, optimizes, and deploys AI agents — handling model selection, tuning with proprietary data, evaluation, monitoring, and continuous optimization — alongside code-first primitives for secure tool execution with permission controls and serving infrastructure that scales to zero when idle.

1.3.2 Core AgentOps Challenges

Building on lessons from MLOps and LLMOps, AgentOps must address five key operational challenges:

1. **Leveraging enterprise-specific context:** Integrating disparate knowledge sources such as unstructured PDFs, Confluence knowledge bases, blogs, forum and company jargon into an agent's context.
2. **Reliability:** Ensuring agents perform consistently across diverse inputs and edge cases.
3. **Observability:** Making every decision point and tool call traceable and explainable. This includes verifying that the correct tools are being called at the right times and that the LLM is appropriately balancing the flexibility provided by its probabilistic nature and ability to understand semantic intent with deterministic functionality to minimize hallucinations and errors.
4. **Safety:** Establishing boundaries to prevent harmful or unintended actions.
5. **Scalability:** Supporting multiple concurrent agents without resource conflicts.
6. **Maintainability:** Updating agent behaviors without breaking existing workflows.

1.4 AgentOps Anti-Patterns

Building effective agentic systems requires avoiding common architectural and operational mistakes that seem reasonable in theory but create problems in practice.

ANTI-PATTERN	PROBLEM	RISK	WHAT TO DO INSTEAD
Starting too broad	Undefined scope and unclear success criteria — e.g., "build a chatbot that answers every employee question"	Agents that perform poorly across all domains rather than excelling in one	Focus on a single, well-scoped use case.
Overcomplex architecture	Using supervisor agents and multi-agent systems when a simple sequential chain would suffice	Orchestration overhead, debugging complexity, and potential infinite loops between agents	Prefer sequential chains over complex multi-agent orchestration.
ReAct loops for single-tool tasks	Wrapping predetermined actions in a Reason and Acting loop	Added latency and cost with no benefit when the next action is never in doubt	Use direct function calls for deterministic actions.
Overcomplicated tools	Using LLM-powered tools like Text-to-SQL when a parameterized query would suffice	Increased latency, unpredictability, and security risk	Reserve LLM-based tool selection for complex mapping.
Unoptimized retrieval	Accepting the first RAG implementation without measuring or improving it	Poor agent outputs caused by irrelevant or missing retrieved context	Implement evaluation-driven retrieval optimization.
Ungoverned tools	Agents with unrestricted access to endpoints and broad credentials	Security, compliance, and reliability failures that can cascade system-wide	Implement a centralized, governed tool registry.
No rollback or versioning	Agent behavior emerges from prompts, tools, routing logic, and data sources — none of which are versioned	Inability to debug or recover from production failures	Version control every agent component.
No systematic evaluation	Relying on manual spot-checks rather than structured test suites	Failure modes that are invisible during development but surface at scale in production	Standardize on programmatic evaluation suites.
No human in the loop for high-stakes decisions	Agents that autonomously execute consequential actions without oversight	Unacceptable risk when agents misinterpret edge cases or encounter novel situations	Mandate human-in-the-loop for high-stakes actions.
No cost controls	Agents without per-request budgets, rate limits, or cost tracking	Runaway costs from expensive tool calls, excessive LLM iterations, or unbounded retrieval	Implement proactive budget monitoring and limits.

1.5 Next Steps

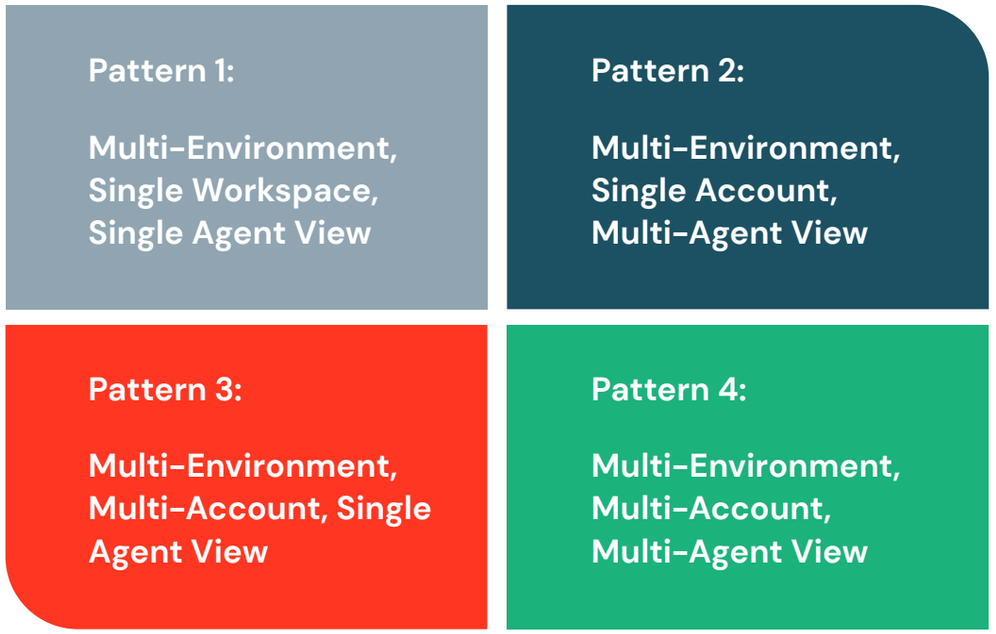
In the following chapters, we'll explore the foundational elements of AgentOps and how to build robust, production-ready agent deployments. We will begin with deployment architecture patterns for agentic systems, then dive into the AgentOps project pipeline and consider how classic DevOps principles should be applied in an AgentOps setting.

CHAPTER 2

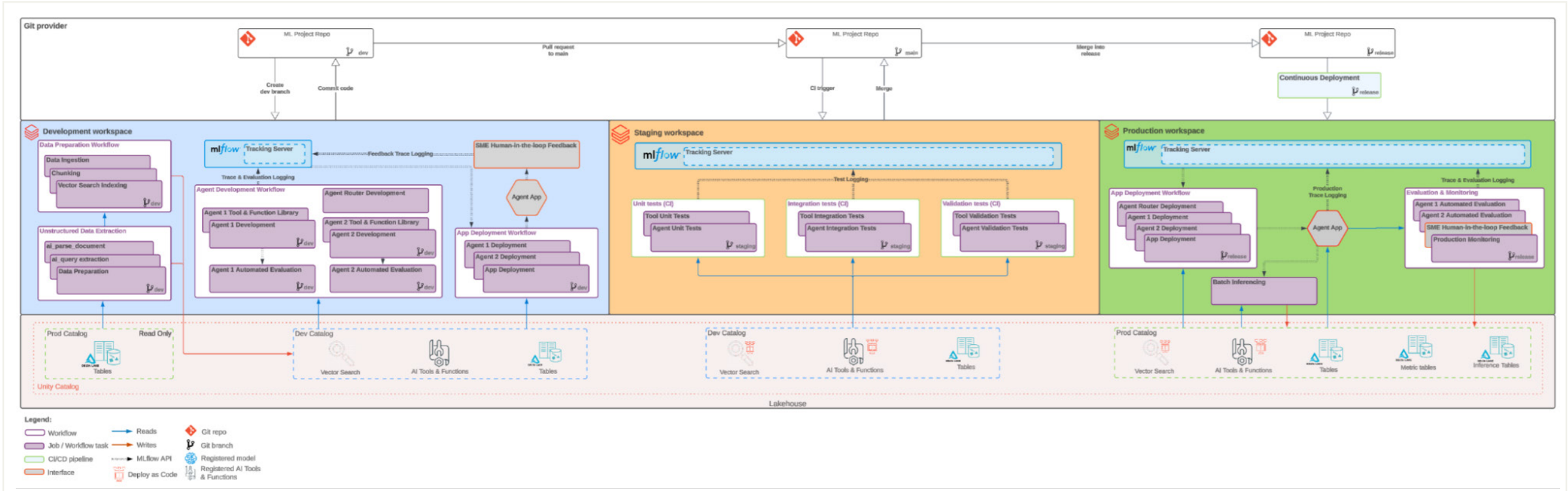
Deployment Architecture Patterns for Agentic Systems

In the previous chapter, we established the core AgentOps challenges: reliability, observability, safety, scalability, and maintainability. We now turn to concrete deployment patterns. The following sections analyze four deployment architecture patterns for agentic systems, progressing from simple to complex scenarios. Each one addresses different organizational needs around agent composition, workspace isolation, and multi-environment deployment strategies, combining these concepts in different ways.

The progression from pattern 1 to pattern 4 represents increasing complexity along two dimensions: account isolation (operational complexity and compliance strength) and agent specialization (coordination complexity and domain optimization). Organizations should choose the simplest architecture that meets their current needs, then evolve incrementally as requirements change.



2.1 Pattern 1: Multi-Environment, Single Workspace, Single Agent View



AgentOps Multi-Environment, Single Account, Single Agent View

USE CASE

Organizations starting with agentic AI or deploying a single, self-contained agent across development, staging, and production environments within a unified account structure.

ARCHITECTURE CHARACTERISTICS

Application architecture:

- Single monolithic agent deployed as a cohesive unit
- Agent serves a focused use case (e.g., customer support, document analysis)
- Simple workflow progression: Data preparation → Agent development → Automated evaluation → Deployment
- Deployment destinations limited to a single app per environment

databricks

Configuration management:

- Environment-specific configurations managed through declarative YAML/JSON files
- Single set of configuration parameters across the pipeline
- Streamlined configuration: one agent definition, three environment overlays (dev/staging/prod)
- Model serving endpoints configured per environment with consistent naming conventions

Infrastructure & resource management:

- Single Unity Catalog metastore shared across all environments
- Enable Unity AI Gateway to provide centralized model access, usage visibility, cost controls, and runtime policies across dev, staging, and production environments
- Shared lakehouse infrastructure with environment-specific catalog/schema isolation
- MLflow tracking server deployed in each workspace for experiment tracking
- Delta Sharing configured for data access across environments
- Common set of supporting resources: vector search, AI tools & functions, models and agents in Unity Catalog

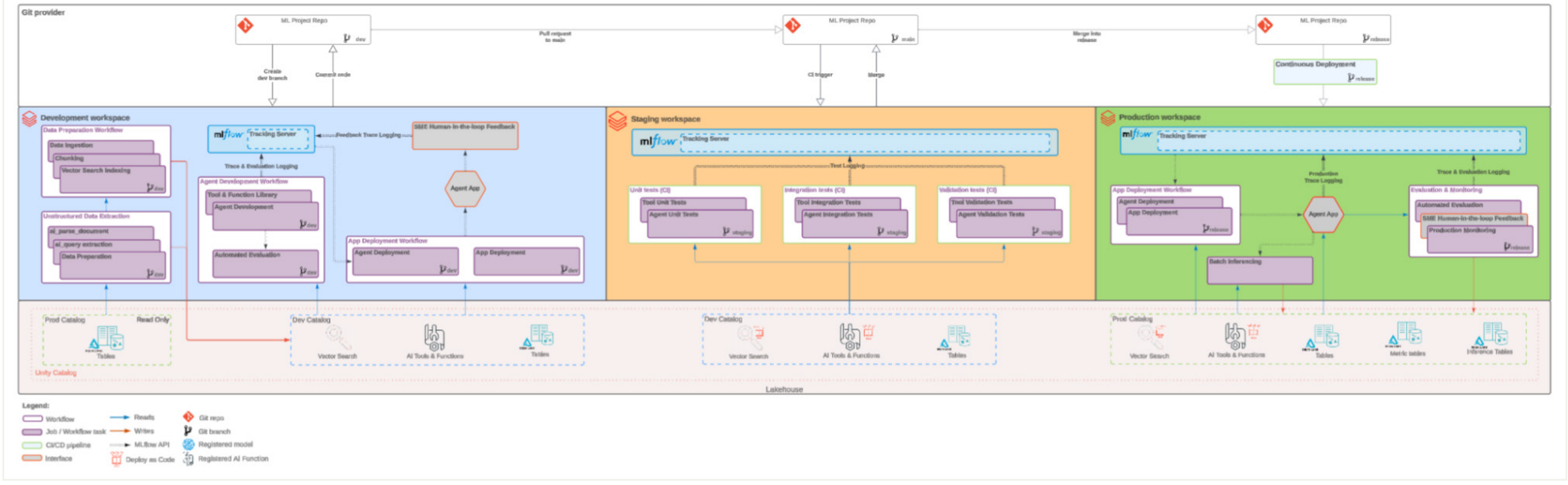
CI/CD pipeline:

- Git-driven workflow with single project repository (M1)
- Automated promotion path: dev → staging → production
- Environment-specific testing gates:
  – Unit tests (CI) in development
  – Integration tests (CI) in staging
  – Validation tests (CI) in production
- SME feedback loop in development
- Continuous deployment trigger from production back to development
- Batch inferencing capability in production for offline evaluation

COMPLEXITY LEVEL

Low — single agent, single account, straightforward promotion path.

2.2 Pattern 2: Multi-Environment, Single Account, Multi-Agent View



AgentOps Multi-Environment, Single Account, Multi-Agent View

USE CASE

Organizations building composite agentic systems with multiple specialized agents (orchestrator + sub-agents) within a single account, requiring agent composition and inter-agent communication patterns.

ARCHITECTURE CHARACTERISTICS

Application architecture:

- Multi-agent composition with clear separation of concerns:
  - Agent 1: Tool & function library (provides shared capabilities)
  - Agent 2: Tool & function library (alternative implementation or specialized tools)
  - Agent 3: Automated evaluation (validates agent performance)
- Microservices-style architecture where each agent is deployed independently as a Databricks App
- Supervisor agent acts as orchestrator for agent interactions
- Supports both parallel agent execution and sequential agent chaining

databricks

Configuration management:

- Multiple YAML/JSON configuration files
- Shared configuration schemas enforced through Pydantic models
- Environment-specific overlays for each agent:
  - Development: Rapid iteration with relaxed validation
  - Staging: Integration testing with production-like configs
  - Production: Hardened configurations with strict validation
- Agent interaction patterns defined in orchestrator configuration

Infrastructure & resource management:

- Single Unity Catalog metastore with agent-specific schemas
- Enable Unity AI Gateway to provide consistent access controls, policies, tracing, and spend management across models, agents, and tools
- Shared lakehouse with agent-specific table isolation patterns
- Each agent maintains its own deployment workflow
- Centralized MLflow tracking server aggregates metrics across agents
- Shared resource pools (vector search, models) with agent-specific indexes/versions

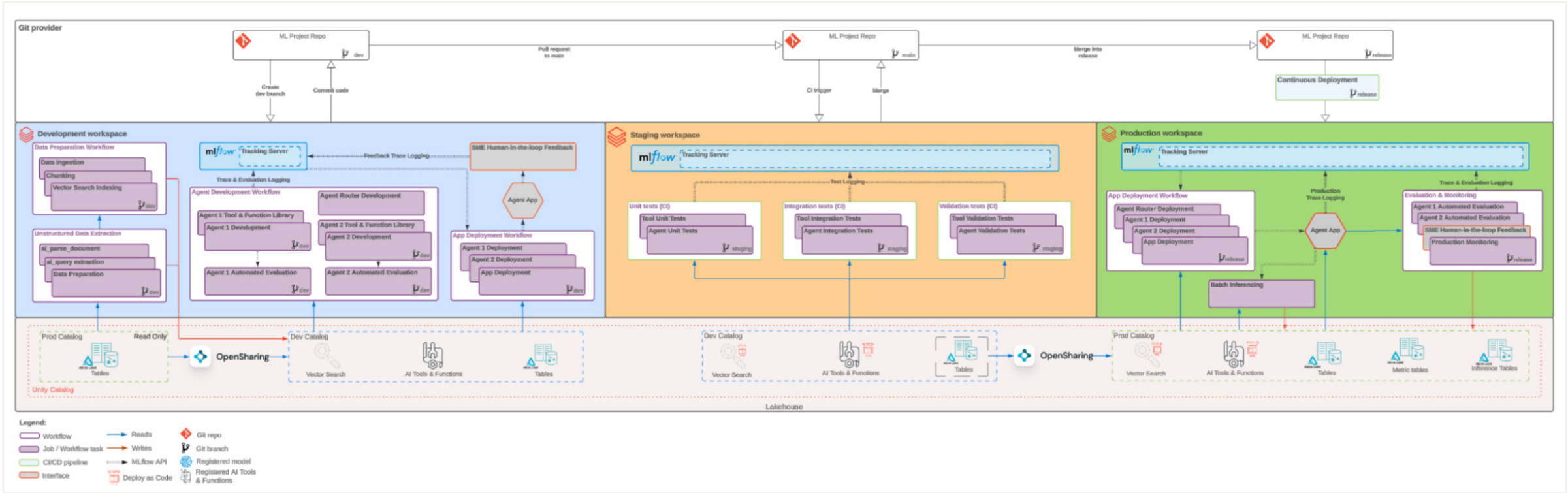
CI/CD pipeline:

- Parallel development workflows for each agent
- Independent deployment cadences per agent
- Cross-agent integration testing in staging environment
- Agent-specific validation gates:
  - Agent unit tests (development)
  - Agent integration tests (staging)
  - Agent validation tests (production)
- Coordinated promotion when agents have dependencies
- Batch inferencing supports multi-agent evaluation scenarios

COMPLEXITY LEVEL

Medium — multiple agents increase configuration complexity and require careful orchestration, but single account simplifies access control.

2.3 Pattern 3: Multi-Environment, Multi-Workspace, Single Agent View



AgentOps Multi-Environment, Multi-Account, Single Agent View

USE CASE

Enterprises requiring strict environment isolation with separate AWS accounts or Databricks workspaces per environment (dev/staging/prod), deploying a single agent across this isolated infrastructure.

ARCHITECTURE CHARACTERISTICS

Application architecture:

- Single agent deployed across completely isolated account boundaries
- Each environment operates as a sovereign deployment unit
- Agent code remains consistent; infrastructure varies significantly
- Production deployment includes additional monitoring and feedback mechanisms

Configuration management:

- Environment-specific configuration files with account-aware parameters
- Cross-account resource references managed through external IDs
- Databricks Automation Bundles critical for environment abstraction: databricks.yml contains account-specific targets
- Unity Catalog catalog and schema names parameterized per account
- Workspace URLs, service principals, and secrets injected at deployment time
- Example bundle configuration:

```yaml
targets:
  dev:
    workspace:
      host: https://dev-workspace.cloud.databricks.com
    resources:
      jobs:
        deployment_job:
          parameters:
            - catalog: dev_catalog
  staging:
    workspace:
      host: https://staging-workspace.cloud.databricks.com
    resources:
      jobs:
        deployment_job:
          parameters:
            - catalog: staging_catalog
```

Infrastructure & resource management:

- Multiple Unity Catalog metastores (one per account)
- Enable Unity AI Gateway to standardize runtime governance and cost attribution across teams and environments while preserving developer choice across models and providers
- Delta Sharing becomes critical for cross-account data access
- Separate lakehouse instances per environment
- Duplicated infrastructure: MLflow tracking servers, vector search indexes
- Network isolation requires PrivateLink or similar cross-account connectivity solutions
- Each account maintains independent:
  - Unity Catalog governance
  - Secret scopes
  - Cluster policies
  - Service principals

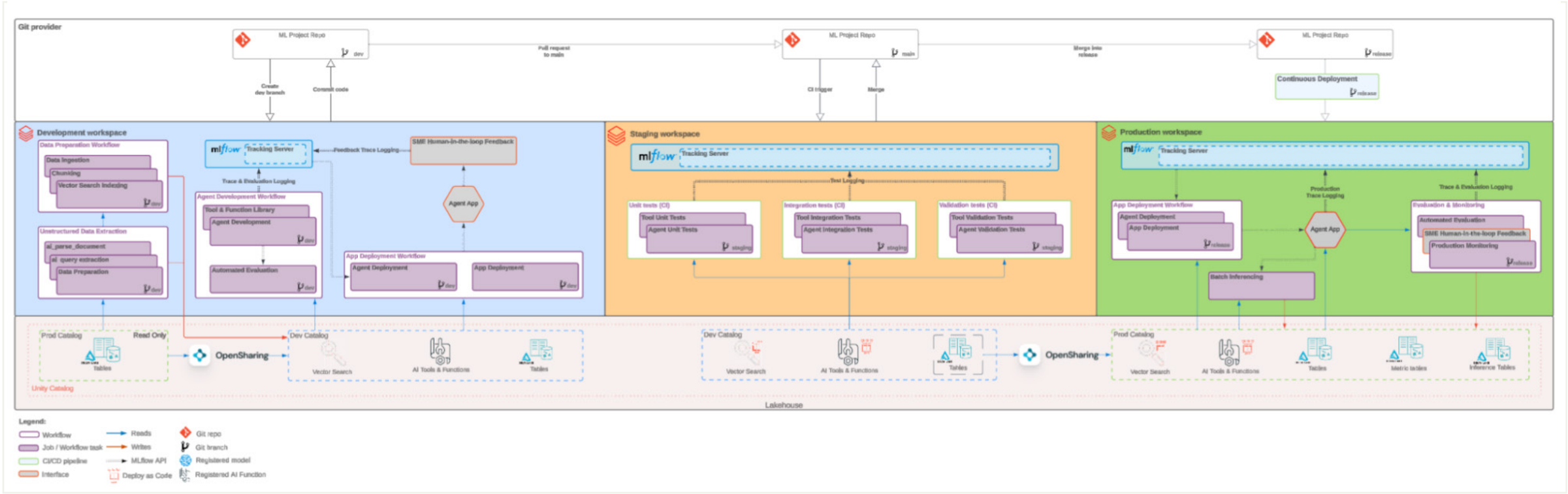
CI/CD pipeline:

- Git repo branches map to accounts/environments
- Cross-account deployment requires elevated permissions
- Promotion workflow includes account switching logic
- Testing strategy adapted for account boundaries:
  - Dev catalog isolation testing
  - Staging cross-account data sharing validation
  - Production monitoring with account-specific observability tools
- Continuous deployment must handle cross-account IAM and permissions

COMPLEXITY LEVEL

Medium-high — single agent simplifies application logic, but multi-account infrastructure significantly increases operational complexity.

2.4 Pattern 4: Multi-Environment, Multi-Account, Multi-Agent View



AgentOps Multi-Environment, Multi-Account, Multi-Agent View

USE CASE

Large enterprises with complex agentic systems requiring both agent composition and strict account isolation, supporting multiple teams, compliance requirements, and large-scale production deployments.

ARCHITECTURE CHARACTERISTICS

Application architecture:

- Full matrix complexity: multiple agents × multiple accounts × multiple environments
- Each agent can be independently versioned and deployed across account boundaries
- Production includes advanced patterns:
  - Post-deployment workflows for validation
  - Batch inferencing for large-scale evaluation
  - Evaluation & monitoring with SME feedback loops
  - Model serving endpoints supporting multi-agent routing

databricks

Configuration management:

- Hierarchical configuration strategy:
  – Base agent configurations (agent-specific YAML)
  – Environment overlays (dev/staging/prod JSON)
  – Account-specific overrides (cross-account resource mappings)
- Declarative Automation Bundles with complex targeting:

```
targets:
  dev_account_agent1:
    workspace: dev-workspace-url
    variables:
      catalog: dev_catalog
      agent_name: agent1
  staging_account_agent1:
    workspace: staging-workspace-url
    variables:
      catalog: staging_catalog
      agent_name: agent1
  # Repeated for agent2, agent3 across all environments
```

- Configuration validation enforced through:
  – Pydantic schemas for agent definitions
  – JSON Schema validation for environment configs
  – Pre-deployment validation jobs

Infrastructure & resource management:

- Multiple Unity Catalog metastores with complex sharing topology
- Enable Unity AI Gateway as a unified runtime control layer across models, agents, tools, MCP servers, and agent harnesses, with centralized permissions, policies, observability, routing, and cost controls
- Open Sharing configured bidirectionally between accounts
- Dedicated lakehouse infrastructure per account
- Resource isolation strategy:
  – Agent-specific compute clusters per account
  – Separate vector search indexes per agent per environment
  – MLflow tracking servers federated across accounts
- Cross-account networking:
  – PrivateLink configurations for prod ↔ staging
  – VPC peering or Transit Gateway for dev access
  – Centralized DNS for endpoint resolution

CI/CD pipeline:

- Git repository structured with multiple project directories (M1, M2, M3)
- Each agent has independent CI/CD workflows
- Multi-stage promotion with cross-account gates:
  – ***Development account:***
    - Data preparation workflows per agent
    - Structured data extraction with agent-specific schemas
    - Parallel agent development workflows
    - Agent unit tests (CI)
    - SME feedback loop per agent

– ***Staging account:***
  - Cross-agent integration tests (CI)
  - Agent integration tests (CI) per agent
  - Model serving endpoint integration validation
  - Performance regression testing

– ***Production account:***
  - Agent validation tests (CI) per agent
  - Post-deployment workflows for smoke testing
  - Batch inferencing for production traffic evaluation
  - Evaluation & monitoring with continuous feedback
  - SME feedback for production validation

- Continuous deployment orchestrates:
  – Cross-agent dependency resolution
  – Account promotion sequences
  – Rollback strategies per agent per account

Operational considerations:

- Observability spans multiple dimensions:
  – Per-agent metrics across accounts
  – Cross-agent interaction tracking
  – Account-level resource utilization
  – Environment-specific service level agreements (SLAs) and alerting
- Cost management requires sophisticated attribution:
  – Agent-level cost allocation
  – Environment-specific budgets
  – Cross-account data transfer costs
- Security and compliance:
  – Account-level audit logs
  – Agent-specific access policies
  – Cross-account data lineage tracking

COMPLEXITY LEVEL

High — full operational maturity required, suitable for organizations with dedicated MLOps/AgentOps teams and mature governance processes.

2.5 Choosing the Right Pattern

The following table summarizes our guidance on when to use each deployment architecture pattern. Again, we recommend that you choose the simplest pattern that meets your current needs and evolve incrementally as those needs change.

PATTERN	WHEN TO USE	TEAM MATURITY	PRIMARY CHALLENGES
Single Account, Single Agent	Starting with agentic AI, proofs of concept, single-purpose agents	Beginner	Initial setup, basic CI/CD
Single Account, Multi-Agent	Composite agents, agent orchestration, microservices approach	Intermediate	Agent coordination, shared resources
Multi-Account, Single Agent	Compliance requirements, environment isolation, enterprise governance	Intermediate/ advanced	Cross-account networking, identity and access management (IAM) complexity
Multi-Account, Multi-Agent	Enterprise scale, multiple teams, complex agentic systems	Advanced	Full operational overhead, requires dedicated AgentOps team

2.6 Common Pipeline Components Across All Patterns

Regardless of complexity, all deployment architecture patterns share these foundational elements:

SOURCE CONTROL

- Git-based version control with branch strategies aligned to environments
- Pull request workflows for code review
- Commit triggers for CI/CD automation

TESTING STRATEGY

- Unit tests in development (agent logic validation)
- Integration tests in staging and agent regression tests (cross-component validation)
- Validation tests in production (production traffic validation)

MLFLOW INTEGRATION

- Automated and human evaluation of agent quality
- End-to-end tracing of agent behavior and tool calls
- Production monitoring and feedback for continuous improvement

LAKEHOUSE FOUNDATION

- Unity Catalog for AI asset registry and policy definitions
- Delta Lake or Iceberg for reliable data storage
- Vector search for retrieval augmentation
- AI tools & functions for agent capabilities

RUNTIME GOVERNANCE & CONTROL

- Unity AI Gateway for centralized governance across models, agents, and tools
- Runtime permissions, policies, and guardrails
- Usage observability, cost attribution, budgets, and limits
- Model routing and provider management

DEPLOYMENT AUTOMATION

- Declarative Automation Bundles as infrastructure as code (IaC)
- Environment-specific parameter injection
- Automated rollback capabilities

FEEDBACK LOOPS

- SME validation
- Continuous monitoring and evaluation
- Metrics-driven improvements

The progression from pattern 1 to pattern 4 represents an organization's journey from initial agentic AI adoption to enterprise-scale AgentOps maturity. The next chapter describes the complete AgentOps project pipeline. This situates the selection of a deployment architecture as one task among many within a broader AgentOps project, spanning from use case identification through deployment and continuous improvement.

2.7 AgentOps Stacks

To streamline the development, curation, and deployment of the architectural patterns in this section, as well as keep current on best practices, we are continuously developing the AgentOps Stacks Project.

AgentOps Stacks provides pre-built scaffolding and deployment pathways to help streamline adoption of AgentOps in Databricks and remove complexity in the management of agentic system productionization. Backed by the guidance of this book, it is an excellent starting point for your AgentOps journey and will continue to grow with this exciting and refining paradigm.

CHAPTER 3

The Complete AgentOps Project Pipeline

Successfully deploying agent-based AI systems requires a systematic approach that spans from initial conception to continuous improvement. Previous chapters outlined important background context and architectural diagrams. This chapter provides a concise roadmap for AgentOps projects, guiding you through the entire development lifecycle.

While generative AI has achieved faster adoption rates than transformative technologies like the personal computer and the internet, a 2025 MIT study suggests that 95% of AI pilots fail. The key to bridging this gap is building the measurement capability to enable AI quality at scale. This creates the reliability needed to earn trust from both end users and leadership stakeholders.

This chapter walks through a typical agent development project flow, with cross-references to other parts of the book for detailed implementation guidance for each phase.

3.1 Phase 1: Project Conception and Team Formation

3.1.1 Business Use Case Identification

Objective: Define a focused, measurable business problem that agents can solve effectively.

Avoid common anti-patterns like overly broad use cases or problems better solved with deterministic workflows. Focus on specific, well-scoped problems with clear success metrics.

DETAILED COVERAGE:
Section 1.4, AgentOps Anti-Patterns

3.1.2 Cross-Functional Team Assembly

Objective: Build a collaborative team with the right mix of technical and domain expertise.

Assemble your core team, including leadership stakeholders, technical developers, and subject matter experts with expertise in the agent domain.

DETAILED COVERAGE:
Section 6.1, Stakeholder Map

3.1.3 Stakeholder Alignment and Governance

Objective: Create organizational alignment around key deliverables, success metrics, and project governance.

Establish governance frameworks, educate leadership on AI limitations, and plan communication strategies for regular stakeholder updates.

DETAILED COVERAGE:
Chapter 6, Stakeholder Management

3.2 Phase 2: Data Preprocessing and Indexing

3.2.1 Data and Agent Architecture Planning

Objective: Design data pipelines optimized for AI consumption rather than traditional analytics.

Focus on unstructured data processing, semantic search capabilities, and multimodal content handling. Key differences from traditional extract, transform, load (ETL) processes include more of an emphasis on vector indexes, embedding generation, tool-calling, and LLM-based information extraction.

DETAILED COVERAGE:
Chapter 2: Deployment Architecture Patterns for Agentic Systems

3.2.2 Production Data Infrastructure Setup

Objective: Implement robust, scalable data infrastructure with proper governance.

Establish ACID transactions, quality monitoring, failure handling, and compliance frameworks for production-ready data pipelines.

3.3 Phase 3: Agent Architecture Design

3.3.1 Architecture Pattern Selection

Objective: Choose the right level of complexity and framework for your use case.

Evaluate DIY/custom approaches vs. code-first frameworks vs. low-code platforms based on your project requirements. Avoid overengineering with unnecessary supervisor/agent patterns when deterministic chains suffice.

DETAILED COVERAGE:
Chapter 2: Deployment Architecture Patterns for Agentic Systems

3.3.2 Cross-Functional Evaluation Planning

Objective: Establish evaluation criteria through collaborative discovery between technical teams and domain experts.

Understanding what "good" looks like for complex, subjective AI outputs requires essential input from domain experts and end users. This necessitates close collaboration to define evaluation guidelines aligned with specific business objectives.

DETAILED COVERAGE:
Section 4.2, Principle 2: The Principle of Feedback
AND
Section 4.3, Principle 3: The Principle of Continuous Learning

3.3.3 Guardrails and Safety Implementation

Objective: Implement comprehensive safety measures and operational boundaries.

Establish input validation, output monitoring, tool access controls, and operational limits to ensure agents behave safely and reliably.

3.4 Phase 4: Agent Development and Evaluation Framework Implementation

3.4.1 The Agent Development Workflow

Objective: Treat development activities as first-class, alongside evaluation, with evaluation signals driving what gets built next.

DETAILED COVERAGE:
Section 4.2.2, Feedback Flow

3.4.2 The Testing Pyramid for AI Systems

Objective: Build a comprehensive evaluation strategy, from manual validation to automated monitoring.

Start with manual SME review to understand quality patterns, then scale through automated LLM judges and programmatic checks. Implement both offline testing and online production monitoring.

DETAILED COVERAGE:
Section 5.1.1, Development of a Golden Evaluation Dataset

3.4.3 Cost-Effective Evaluation Strategies

Objective: Scale evaluation while managing token costs and computational overhead.

Unlike traditional unit tests, LLM-based evaluations cost money. Implement token-efficient strategies, field-based vs. trace-based evaluation approaches, and smart sampling techniques.

DETAILED COVERAGE:
Section 5.2, Evaluations Are a Revenue Generator, Not a Cost Center

3.5 Phase 5: Feedback Collection and Monitoring

3.5.1 Multi-Source Feedback Integration

Objective: Establish mechanisms for comprehensive feedback collection from all stakeholders.

Integrate end-user feedback, SME validation, LLM judge assessments, and system metrics. Implement both structured pre-production reviews and real-time production collection mechanisms. Enable automation through LLM judge alignment with SME feedback and prompt optimization.

DETAILED COVERAGE:
Section 4.2, Principle 2: The Principle of Feedback

3.5.2 Production Telemetry and Observability

Objective: Implement comprehensive monitoring for both system and AI-specific metrics.

Monitor operational metrics (latency, cost), quality metrics (evaluation scores, user satisfaction), and business impact (ROI, efficiency gains). Ensure proper data protection and audit trails.

A recurring failure mode is discovering runaway spend only after the invoice arrives. Agent costs multiply rather than add: every sub-agent, retry, and guardrail check adds its own cost on top, so one user request can turn into four to six LLM calls behind the scenes. The Databricks Unity AI Gateway OSS Version MLflow AI Gateway addresses this by acting as a centralized control plane for every LLM call: routing all traffic through one point, autologging traces that capture token counts, latency, model, and cost per call, and attributing spend across the full span hierarchy (orchestrator → sub-agent → synthesis → guardrail). This lets teams pinpoint which component is driving cost before investing in the wrong optimization. Budget policies enforce limits directly: ALERT actions fire a webhook (e.g. a Slack notification) at a threshold, REJECT actions block requests outright over a hard cap. Cost management stops being reactive firefighting and becomes something you can actually tune over time.

→ For an end-to-end worked example, see Kyra Wulffert's **"Preventing Runaway Agent Costs with MLflow AI Gateway"**

3.6 Phase 6: Iterative Development and Optimization

3.6.1 Continuous Learning Feedback Loop Implementation

Objective: Establish systematic improvement cycles based on evaluation insights.

Implement iterative cycles of data collection, analysis, hypothesis formation, targeted improvements, validation, and deployment. Focus on prompt engineering, tool selection, architecture tuning, and cost optimization.

→ DETAILED COVERAGE:
Chapter 7, Principle 3: The Principle of Continuous Learning

3.6.2 Trust Building Through Transparency

Objective: Build stakeholder confidence through demonstrable quality improvements.

Use regular demonstrations, metric transparency, honest failure analysis, and user education to build trust. Transform evaluation data into reusable organizational assets.

→ DETAILED COVERAGE:
Chapter 7, Principle 3: The Principle of Continuous Learning

3.7 Phase 7: Scaling and Governance

3.7.1 Enterprise Scaling and Risk Management

Objective: Expand successful pilots to enterprise-wide deployment with proper governance.

Address cost management, quality consistency, organizational change, and technical infrastructure scaling. Establish AI governance boards and compliance monitoring.

Getting one agent to production is a milestone; running dozens across business units is a different discipline. As agents proliferate, organizations hit a governance problem: they can no longer answer basic questions about how many agents are running, who owns them, what data they touch, or what is driving costs. To manage this, an AI governance board is needed. Unlike traditional setups where compliance, IT and security operated separately from software engineering and business owners, an AI governance board only succeeds with every team prioritizing governance. An AI governance board needs to span engineering, security, legal/risk, and business owners. It sets centralized standards with federated execution, keeps a human in the loop for high-risk decisions, tracks evolving regulation, and carries C-suite sponsorship so governance enables teams rather than blocks them.

3.8 Conclusion: Delivering Strategic Business Value

While each project is different, the systematic pipeline outlined here provides a blueprint of how to transform AI initiatives from experimental pilots into strategic business assets that drive measurable outcomes.

This approach enables organizations to:

- Accelerate time-to-value through structured, repeatable processes.
- Minimize project risk by addressing common failure modes proactively.
- Scale AI capabilities across the enterprise with consistent quality.
- Generate measurable ROI through improved efficiency and quantified performance.
- Build organizational AI maturity that compounds over future projects.

Use this pipeline as your project roadmap, consulting the referenced chapters and sections for specific implementation guidance. Remember that successful AgentOps projects often iterate between phases as understanding deepens and requirements evolve.

The following chapters examine the continuing relevance of the core DevOps principles of flow, feedback, and continuous learning for AgentOps and provide relevant guidance across the project pipeline.

CHAPTER 4

DevOps Principles for Agent Operations

DevOps is incredibly useful and not a new idea. However, it does need to be updated to be relevant for AgentOps. The following chapters examine the continuing relevance of the core DevOps principles of flow, feedback, and continuous learning for AgentOps and provide relevant guidance across the project pipeline.

4.1 PRINCIPLE 1: The Principle of Flow

As outlined in *The DevOps Handbook*, teams must reduce the time required to deploy changes into production while simultaneously improving service reliability and quality, rather than trading off speed against reliability. This principle remains foundational for generative AI systems. However, generative AI introduces unique challenges:

- **Nondeterministic behavior**
  Unlike traditional software, where identical inputs generally produce identical outputs, LLMs can generate different responses to the same prompt, making traditional testing approaches insufficient.
- **Unstructured data interfaces**
  Testing systems that accept and output natural language, images, and audio requires different evaluation methodologies than testing APIs with structured data.
- **Context-dependent quality**
  The quality of AI outputs often depends heavily on context, user intent, and domain-specific knowledge that is difficult to encode in traditional test suites.
- **Expensive evaluation**
  Unlike traditional unit tests that run instantly and for free, LLM-based evaluations consume tokens and incur costs, requiring strategic approaches to test suite design.

The sections that follow address how to do that through the lens of the first DevOps principle: flow.

4.1.1 Implementing Efficient Flows

Development of a Golden Evaluation Dataset

Unit tests and integration tests for traditional software can usually be defined up front, because we have a clear idea of the behavior of the system we want to build, including its edge cases. Generative AI is different. Creating tests requires an additional preliminary step: extensively analyzing traces of previous requests and responses to identify common failure modes. Only when these failure modes have been identified and categorized can we define automated checks to identify the failure patterns.

THE DANGEROUS DEFAULT

Teams often fall into a dangerous default pattern where one tester runs five to six queries repeatedly in a chat interface and evaluates whether each response meets the acceptance criteria. This approach fails to capture error frequency metrics across a broad input distribution or classify failure types systematically.

THE SCIENTIFIC APPROACH

Teams should instead apply the scientific method iteratively:

1. Start by categorizing failure modes as quickly as possible using approximately 100 traces. These traces can be collected in a variety of ways:
   - If the application is already in production, sample production traces. Make sure these traces are sufficiently diverse and cover a range of input categories, conversation lengths, and types of tool calls.
   - If the application is still in development, work with an SME to define a set of inputs that they anticipate should cover the range of scenarios a given workflow or sub-agent should cover. Because LLM calls are nondeterministic, each input should be run three or four times to measure response consistency.
2. Create failure reports using tools like MLflow or Jupyter notebooks with visualizations.
3. Gradually calibrate automated LLM judge metrics based on insights from step 2 and consultation with domain experts. (We will cover a basic workflow in more detail in the following chapter, using a worked example of a customer email generator). The output will be a set of scorers (such as the MLflow GenAI scorers) that can scale up trace evaluation in development and production settings.
4. Set up scorers to run in both offline development mode and an online setting on production traces. For offline settings, this process involves passing an MLflow Dataset or Pandas DataFrame to the mlflow.genai.evaluate() function. For online settings, it involves collecting traces into an Online Analytical Processing (OLAP) system, such as Delta tables, and then running batch jobs to evaluate a sample of these production traces using the same MLflow scorers.
5. Surface aggregated metrics in dashboards to monitor trends over time. Periodically have product managers/business SMEs review and refine test suites based on new failure modes observed in production.

Note: It can be useful to start with failure reports provided by existing evaluation frameworks. For example, the Evaluation UI for an MLflow experiment allows users to filter for traces that fail a specific assessment. These traces, along with the LLM judge rationale that led to the failed assessment, can be surfaced in a notebook, SQL dashboard, or Databricks Apps UI.

4.1.2 Evaluations Are a Revenue Generator, Not a Cost Center

AI system evaluations are often viewed as a necessary evil, a compliance checkbox, or a drain on resources. Instead, they should be viewed as a strategic investment that fundamentally de-risks AI initiatives, accelerates the deployment of reliable systems, and ultimately drives tangible business value and a durable competitive advantage. This chapter has discussed fast, reliable deployment via the principle of flow, highlighting the importance of evaluation-driven agent development and creating quick feedback loops.
The next chapter covers the second DevOps principle, the principle of feedback, in more detail.

4.2 PRINCIPLE 2:

The Principle of Feedback

4.2.1 The Current State of AI System Monitoring

Given that generative AI applications blend software development and machine learning, GenAI developers need to be aware of different approaches to collecting feedback from the systems they build. For experienced software engineers, collecting comprehensive telemetry is second nature. An engineer developing a web server, for example, would typically monitor everything from page load times to database query latencies, disk usage, and network latency. These metrics are usually operational in nature and easily quantifiable. Traditional machine learning practitioners, meanwhile, tend to be more narrowly focused: They concentrate on metrics related to model performance, such as precision, recall, and root mean squared error (RMSE). GenAI developers need to handle both.

In addition, they must understand how to collect and incorporate new forms of feedback that arise regarding the natural language outputs of large language models. This includes subject matter expert feedback, LLM judge outputs and rules-based checks.

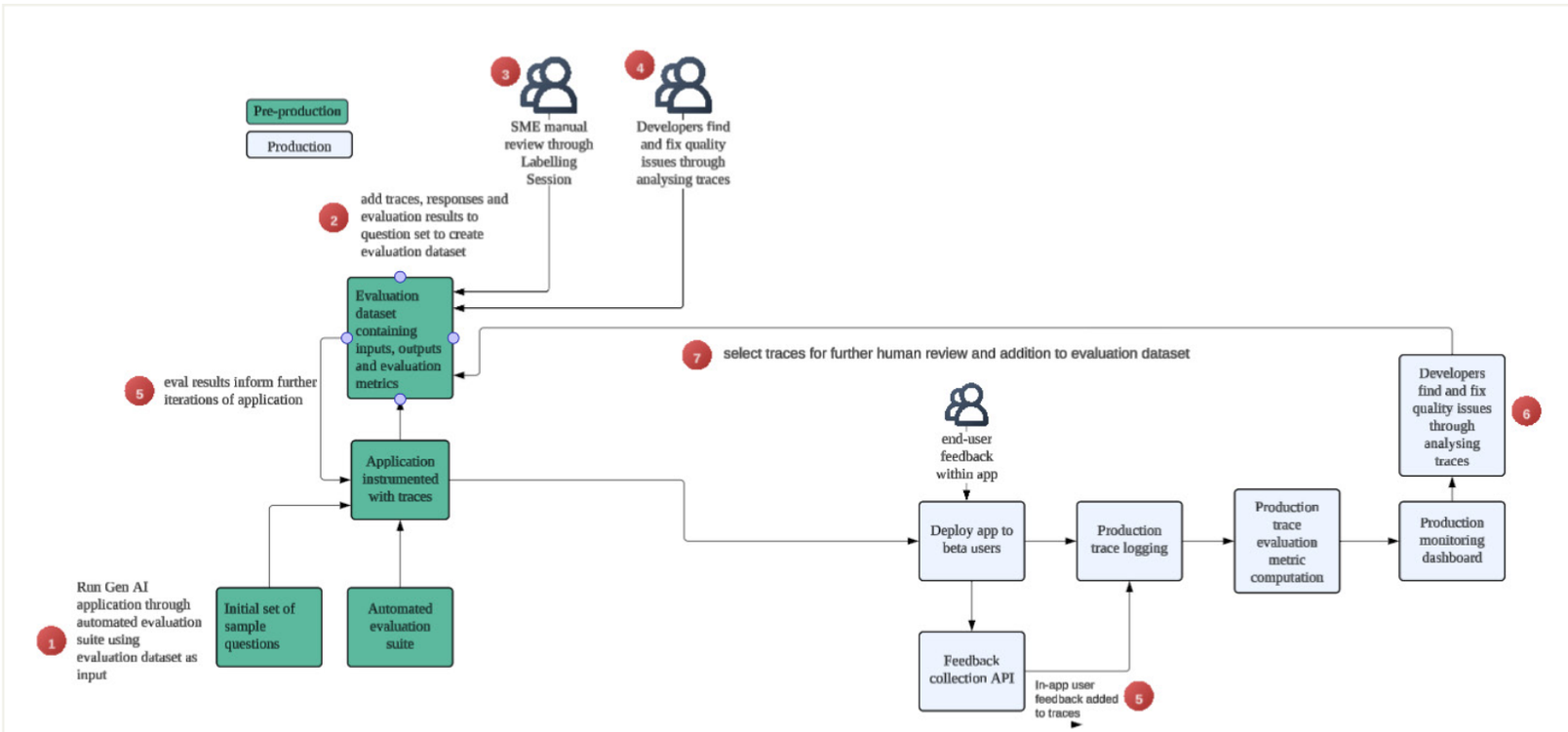
These forms of feedback must be evaluated for:

- **Consistency**
  SMEs may evaluate the same output differently based on context, timing, or individual interpretation. Establishing inter-rater reliability requires clear rubrics and regular calibration sessions.
- **Representativeness**
  End-user feedback suffers from selection bias. Users who provide ratings are often those with extreme experiences (very satisfied or very frustrated), not representative of typical usage.
- **Alignment**
  LLM judges must be continuously validated against human judgment through manual and automated approaches. A judge that agrees with itself consistently but diverges from SME assessments provides false confidence.
- **Actionability**
  Raw feedback (“this response is bad”) doesn’t tell you what to fix. Effective feedback must be categorized and structured to reveal patterns, such as whether issues stem from retrieval quality, reasoning errors, tone problems, or factual inaccuracies.

4.2.2 Feedback Flow

The step-by-step workflow of collecting and acting upon feedback for a generative AI application is not very different from that of a software developer identifying issues in development and production, then fixing bugs and writing tests to guard against future regressions.

The following diagram shows a feedback collection workflow for developing a generative AI application with both offline and online components.



The feedback flow

The workflow can be divided into three phases:

PRE-PRODUCTION:

- SMEs manually review and label data in labeling sessions.
- Developers analyze traces to identify and fix quality issues.
- An evaluation dataset is built containing inputs, outputs, and metrics.
- Automated evaluation suites test the application.

PRODUCTION:

- The application is deployed with tracing enabled (first to beta, then full production).
- A real-time monitoring dashboard tracks performance metrics.
- Users provide feedback directly in the app via a feedback collection API.
- Production traces are continuously evaluated.

CONTINUOUS IMPROVEMENT:

- The system creates a feedback loop where evaluation results inform further application iterations. Offline evaluation uses the dataset used during development, while online feedback comes from production users and automated monitoring.
- Selected traces and human reviews are added back to the evaluation dataset to improve future iterations.
- Judge alignment is executed after each iteration of SME feedback to ensure LLM judges are as closely tied to SME feedback as possible
- The aligned judges are used as scorers for prompt optimization to directly drive improvements to the agent. In MLflow, this can be automated by running optimize_prompts() immediately after an align() job completes.

The workflow enables a SME driven development loop, where SME feedback directly triggers workflows that improve monitoring through aligned judges, and directly improve agent quality. This approach balances scalability with quality as human expertise guides automated agent optimization. Initial cross-functional alignment is critical to ensure there is mutual agreement on agent scope and expected outputs, then from that point automation drives continuous improvements. Developers can choose to inject manual gates in the process as well, such as re-running an evaluation job on a separate dataset to ensure aligned judges generalize to new dataset, but in principle this approach enables pure automation of improvements.

Databricks surfaces these through MLflow Tracing (span-level capture of agent execution), MLflow Assessments (the UI for reviewing traces and attaching judgments), and the MLflow Feedback API (mlflow.log_feedback()) for programmatically recording human and automated feedback against a trace

4.2.3 Best Practices for Collecting Feedback

The following practices help teams build effective feedback systems for generative AI applications:

Start with boolean feedback

Use MLflow's boolean feedback type for simple thumbs up/down collection. Once you analyze patterns with MLflow's search APIs, expand to numeric ratings or structured feedback types.

Link feedback to fresh traces

Collect feedback immediately after trace generation when the interaction context is available. MLflow's direct trace-feedback linkage ensures you always have the full execution context.

Use source attribution properly

Set meaningful source_id values in AssessmentSource objects for tracking feedback providers. MLflow preserves complete audit trails with timestamps and source attribution.

Combine programmatic and UI collection

Use MLflow's API for autoated collection and the UI for manual review. Both methods integrate seamlessly, allowing different teams to contribute feedback through their preferred interface.

Use consistent naming conventions

Standarize feedback names like 'user_satisfaction' or 'quality_rating' across traces. This enables MLflow's search and aggregation features to provide meaningful insights across your application.

→ Feedback Best Practices

4.2.4 Worked Example: Customer Email Generator

Consider an AI application that helps account managers with customer communications. The system generates different types of emails, including meeting follow-ups and cold outreach messages, using context from previous customer interactions. To assess the application's quality, several sources of feedback are needed:

1. A select group of account managers (SMEs) verify whether the emails accurately reflect known context about customer meetings to guide the development process.
2. Account managers (end users) indicate whether the emails were helpful and saved them communication time.
3. LLM judges evaluate the emails' relevance, tone, and professionalism.
4. Automated rule-based checks ensure the emails follow company guidelines.
5. As more data is collected, SMEs continue to provide feedback which automatically updates LLM judges to be more accurate leading to agent performance improvement

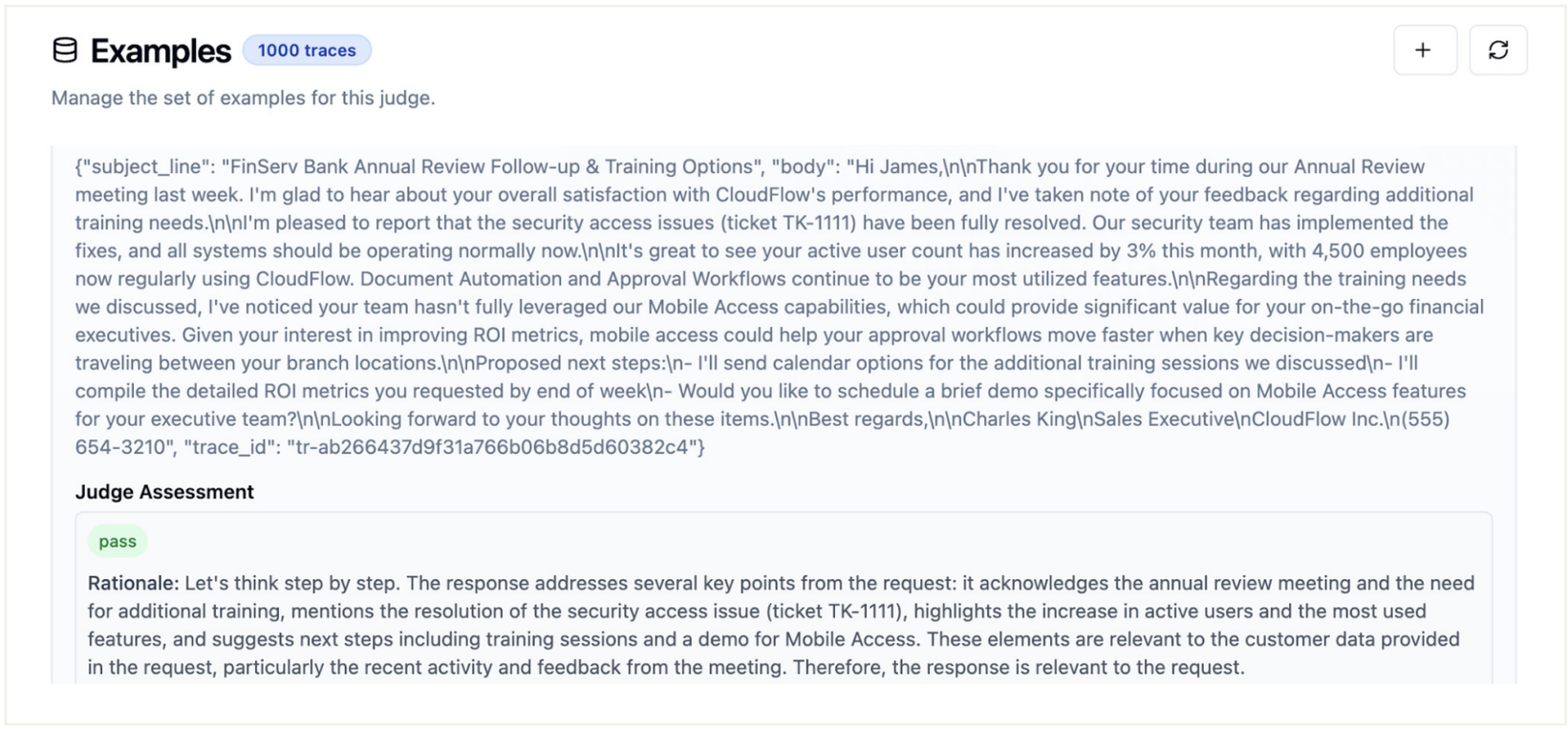
Each feedback source helps improve the system's ability to generate appropriate, contextual customer communications. This can occur through manual improvements such as adding new tooling or functionality, as well as automating LLM judge alignment to SME feedback and using those aligned judges to directly improve the agent through prompt optimization

FEEDBACK IS ITERATIVE

Collecting feedback by itself is not enough to ensure a quality application. In contrast to more traditional system metrics (such as web page response times) that are informative as is, feedback for generative AI applications usually needs to be categorized and analyzed before it can be incorporated into application improvements. GenAI engineers will typically need to perform these steps on traces generated by the application. This is important for calibrating LLM judges and developing a taxonomy of errors that may occur in production.

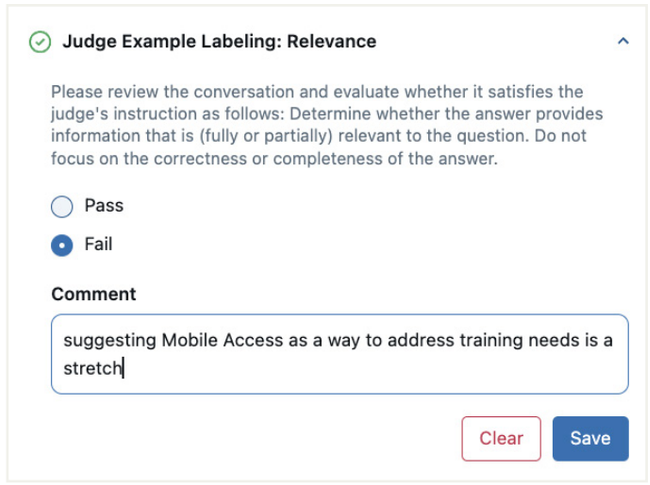
For example, when we start application development, we may set up a generic "relevance" LLM assessor that:

- Provides a binary pass/fail assessment on whether the generated email was relevant given the provided context
- Provides a rationale for its decision



LLM relevance judge output

However, an actual SME (account manager) might disagree with this feedback and provide a correction through the labeling UI.



SME feedback

Given this feedback, we might decide that the original relevance metric is too general. Instead, we need evaluation criteria that assess whether proposed solutions actually fit the customer's specific context:

- **Solution appropriateness:** Are the suggested solutions genuinely suited to addressing the customer's stated needs, or are they tangential product pushes?
- **Context alignment:** Does the response demonstrate understanding of the customer's actual situation and constraints?

This refinement leads to the creation of a custom evaluation judge that reflects the enterprise's actual quality standards, rather than generic relevance.

ANALYZE FEEDBACK MANUALLY FIRST, THEN AUTOMATE THE PROCESS

The pattern of starting with manual SME review, discovering evaluation criteria, then automating those criteria through LLM judges or code checks is fundamental to building reliable generative AI applications. This approach acknowledges that:

- **Requirements emerge through exploration.**
  Quality criteria are often unknown up front. SMEs require empirical evidence to provide a credible description of how to evaluate an agent.
- **Domain expertise is essential.**
  SMEs understand nuances that engineers may miss.
- **Automation scales human judgment.**
  Once initial criteria are clear, LLM judges can apply them consistently at scale, and developers can leverage automation to continue to align judges based on new feedback provided by domain experts.

This iterative feedback process highlights the importance of translating business requirements into technical specifications. A common failure mode is for development teams to build in isolation, attempting to perfect the system before showing it to SMEs, only to discover fundamental misalignments too late. Early and continuous SME involvement prevents wasted effort by surfacing domain-specific requirements, edge cases, and quality expectations before they become expensive to fix.

There has been debate about whether this translation should be done by product managers, domain experts, or engineers who become more "product-minded." Regardless of how responsibilities are allocated, the key is creating a collaborative environment where technical and business teams jointly discover what's possible with generative AI while maintaining high quality standards.

This approach reflects the challenges identified by Shreya Shankar and Hamel Husain in their AI Evals course, particularly their "Three Gulfs" framework. Related research by Shankar et al. on "Who Validates the Validators?" demonstrates how evaluation criteria emerge through the iterative process of aligning automated evaluations with human judgment.

4.2.5 Summary

Feedback helps guide the AgentOps process via evaluation loops that evolve from SME feedback. Organizations then need to institutionalize what they've learned via the feedback process to improve future efforts. The next chapter discusses applying the DevOps principle of continuous learning to enable institution-wide transformation and compound insights across projects.

4.3 PRINCIPLE 3:

The Principle of Continuous Learning

Generative AI is a field that changes fast. What was hard to accomplish yesterday might become easy tomorrow. For instance, as of early 2026, post-training methods have greatly improved a model's ability to handle instructions and tool use, which has enabled agentic applications that were not feasible a year ago. This rapid rate of change makes it imperative for both business and technical stakeholders to continuously update their mental models of what products it's possible for them to build.

Organizations can enable institution-wide continual learning by encoding hard-won knowledge into assets different teams can reliably reuse to increase their chances of success. Examples of such assets include:

- **Standardized frameworks**
- **Reusable reference architectures**
- **Industry- or enterprise-specific agentic design patterns**

4.3.1 Context

Typically, an organization does not want to build just one successful generative AI application. Ideally, they are able to repeatedly produce high-quality applications that can accelerate business outcomes across multiple departments. We've spoken with larger organizations that estimate up to hundreds of potential use cases. These customers typically come to us asking not only how to ensure the quality of a single proof of concept or minimum viable product (MVP), but how to build reliable best practices that will allow them to scale how the organization applies generative AI to improve business practices.

Broadly speaking, scaling GenAI best practices across an organization can be achieved by:

- Institutionalizing technical best practices through standardized frameworks, reference architectures, and documentation
- Investing in continuously refining how business and technical stakeholders work together to uncover how best to use generative AI to accomplish their goals

4.3.2 Standardized Frameworks

Standardized development frameworks allow us to achieve consistent, reliable, and efficient delivery at scale. Rather than the organization creating a series of one-off custom projects, an efficient framework aims to make software development and deployment function like a well-oiled assembly line, ultimately enabling faster time to market with higher quality and lower risk.

An example of a standardized framework that can be used for generative AI applications is Declarative Automation Bundles. A Databricks Asset Bundle is a deployment unit that packages together all the related resources needed for a complete data/ML workflow or application. Specifically, it's a:

- **Resource collection:** It contains notebooks, jobs, pipelines, models, libraries, and configuration files that work together as a cohesive unit.
- **Infrastructure-as-code package:** YAML configuration files specify not just the code paths, but also the compute resources, permissions, schedules, and environment settings needed to run an application.
- **Deployment artifact:** It can be versioned, promoted across environments (dev → staging → prod), and deployed atomically as a single unit.
- **Dependency container: I**t handles all the interconnections between different Databricks resources — how jobs trigger each other, which notebooks depend on which libraries, etc.

THE DEVELOPMENT LIFECYCLE USING DECLARATIVE AUTOMATION BUNDLES

At a high level, the development lifecycle consists of the following steps:

1. Create a skeleton project folder structure by initializing a default bundle template or a custom bundle template.
2. Update the bundle configuration files by specifying settings such as workspace details, artifact names, file locations, job details, and pipeline details. These can be customized according to different development, staging, and production deployment targets.
3. Add source code to relevant project folders.
4. Verify that definitions in the bundle configuration files are valid by running the Databricks CLI tool databricks bundle validate.
5. Deploy the jobs, agents, vector databases, and other artifacts that make up the generative AI application by running databricks bundle run.

THE BENEFITS OF STANDARDIZED FRAMEWORKS

A framework like Declarative Automation Bundles provides a templatized end-to-end definition of a deployable project. This approach provides several benefits:

1. The templatized structure enables standardization across development teams, rather than each team developing its own standards.
2. Because project and resource definitions are captured as YAML files, they can be managed by version control systems.
3. Projects can be deployed using external CI/CD tools such as GitHub Actions and Azure DevOps.

4.3.3 Reference Architectures

Reference architectures provide teams with proven blueprints containing established design patterns and best practices for building agentic applications. For example, a common pattern in regulated industries is to require all requests and responses to an agent endpoint to pass through a guardrail step that filters out personally identifiable information from LLM outputs.

We covered deployment reference architectures in detail in Part 1. We mention them again here in the context of organizational learning because they are valuable tools for scaling architectural knowledge and best practices across an organization. Reference architectures create a shared understanding across teams about how systems should be built and why. They also capture a significant portion of the architectural decisions (commonly on the order of 70–80%) up front, leaving teams to focus on the 20–30% that are specific to their unique use cases.

4.3.4 Industry- and Enterprise-Specific Agentic Design Patterns

Reference architectures are useful for outlining how different components of a technology stack integrate with each other. But it's also useful to have shared blueprints that drill down specifically into how agents themselves should be designed.

COLLABORATIVE METRIC DEVELOPMENT AND LLM-AS-JUDGE CALIBRATION

A critical challenge in AgentOps is establishing effective evaluation criteria. As we've noted a few times, unlike in traditional software and machine learning, where success metrics are often clear-cut, evaluating agentic AI requires synthesizing knowledge scattered across different departments, from technical teams who understand system capabilities, to business leaders who own outcomes, to subject matter experts who define quality in domain-specific contexts.

Without this unified view, teams may struggle with ambiguous success criteria and waste effort measuring against disparate, unwritten standards. It isn't possible to calibrate an automated judge if the success metric itself is not clear.

BUILDING SHARED UNDERSTANDING THROUGH CROSS-FUNCTIONAL COLLABORATION

When a technical lead, a domain expert, and a product manager review the same customer service response, they often disagree on what makes it "good" or "bad" — and that disagreement is exactly what you need to surface and resolve.

Start by gathering concrete examples of your agent's outputs, ideally covering the range of scenarios you expect in production. Bring together representatives from each stakeholder group and work through these examples systematically. The technical team might focus on whether the agent used tools correctly, while the domain expert cares about accuracy and the product manager evaluates the user experience.

These sessions reveal the gaps between what different groups consider success. A response that's technically correct might still frustrate users if it's too verbose. An answer that delights users might concern compliance teams if it makes claims beyond the agent's knowledge base.

The goal isn't to eliminate these different perspectives, but to make them explicit and find ways to measure all the dimensions that matter. This usually means developing multiple evaluation criteria rather than trying to find a single "quality score."

Once you've established shared criteria through human review, you can scale the evaluation using LLM judges trained on your team's collective understanding. These judges won't replace human oversight entirely, but they can handle the bulk of routine evaluation while flagging edge cases for human review.

4.3.5 Business Stakeholder Alignment

Although the consensus is that generative AI has the potential to automate and enhance workflows across industries, the details of how we should tailor agent design to the particularities of any given industry remain vague. Teams often have to learn as they develop. This hard-won knowledge should be shared as design patterns that can be adopted across an enterprise, so different teams don't have to reinvent the wheel. Let's consider a couple of examples.

EXAMPLE 1: INSURANCE CLAIMS PROCESSING

There are a few domain-specific aspects of insurance claims processing that influence how agents need to be built:

- **Claims processing is an event-driven process.** When a claim is first received, the event may trigger a document review process. If the review process identifies missing documents, then another event is triggered: A webhook is called that sends a customer a follow-up email asking for the missing documents.

- **Claims workflows need the ability to be paused and resumed.** The event-driven workflow outlined above implies that, at certain crucial junctures, the agent graph needs to be paused and then resumed at a later point in time. In this case, the processing workflow must be paused when the email is sent until the next event occurs (an email reply arriving from the customer).

  In terms of agent design, engineers need to plan for a state store that can create a checkpoint with details on the paused step. This checkpoint will be referenced when the workflow is resumed. The agent subgraph encoding the agentic workflow will also need to be designed such that it can resume processing at specific points rather than needing to be rerun from the beginning.

- **Agent state depends on both local and global events.** To add another layer of complexity, there may be parallel events (such as registering claims with a third-party regulatory authority) that need to be tracked and taken into account when the workflow is resumed in order for the agent to have the most up-to-date context.

These requirements — an event-driven architecture, pause-and-resume capability, and a state store — are something that will likely be shared across different claims departments. Developers building agentic solutions for each department would benefit from shared design elements, such as a common schema outline for the state store or a common JSON payload structure used to trigger a resume event.

EXAMPLE 2: FINANCIAL SERVICES COMPLIANCE

Within the financial services industry, it is common for regulatory bodies to periodically release new regulations and reporting requirements that compliance departments must follow. Monitoring for new regulatory requirements and designing appropriate compliance checks used to be a manual process done by humans. However, organizations are increasingly interested in having this process automated by an agent.

As with the example of insurance claims processing, there are shared elements of agent design that should be documented and reused by agents implemented for various compliance and audit-related use cases. Examples here might include mechanisms for monitoring regulatory updates, interpreting policy changes, routing ambiguous cases for human review, and maintaining audit trails.

4.3.6 Summary

The Principle of Continuous Learning treats every production interaction as an opportunity to improve both the agent and the organization that builds it. Realizing it at scale depends on standardization: shared frameworks such as Databricks Asset Bundles give teams a consistent development lifecycle, while reference architectures and industry-specific design patterns turn hard-won lessons into reusable starting points.

Continuous learning is not purely technical — collaborative metric development and LLM-judge calibration keep evaluation aligned with business intent, and educating business stakeholders builds the shared understanding needed to act on feedback. As the insurance claims and financial services compliance examples show, the teams that compound these practices — codifying patterns, aligning judges with domain experts, and reusing components across use cases — are the ones that move from one-off agents to a reliable, ever-improving portfolio of agentic systems.

CHAPTER 5

Prioritizing High-Leverage Activities

5.1 What Should I Focus on First?

Our customers are never short of ideas for how generative AI can improve their processes. But they do frequently ask, "What should I focus on first when developing my agent?"

This question is not surprising. Agentic workflows are an order of magnitude more complex than the linear RAG chains that were the main focus of 2024. While a basic RAG pipeline may include a predictable set of steps, such as retrieval and generation, agentic workflows tend to be less well scoped. They usually involve open-ended reasoning, dynamic tool selection, and automation of intricate multi-step business workflows.

In insurance claims processing, for example, the agent may need to go through multiple rounds of asking for evidence for a claim, verifying that evidence, and asking the customer for more information. Likewise, a telco customer support agent investigating an unexpectedly high bill may need to first identify the cause of the anomaly, then wait for a human to approve a refund.

To address this complexity, we've identified a repeatable sequence of planning activities that we have found to be helpful for mapping out the problem space and aligning stakeholders. Focus on these key areas at the start of a project, and the project's chances of success increase greatly.

These activities can be implemented as part of a design sprint, a time-boxed collaborative workshop that brings together cross-functional teams to rapidly prototype and validate solutions. Originally popularized by Google Ventures, design sprints compress months of traditional planning into an intensive, focused period where teams can quickly align on design decisions, identify key data sources, and plan evaluation suites before committing significant development resources to the complex technical implementation.

5.2 Key Planning Activities to Prioritize

Let's start by going over our recommended planning sequence:

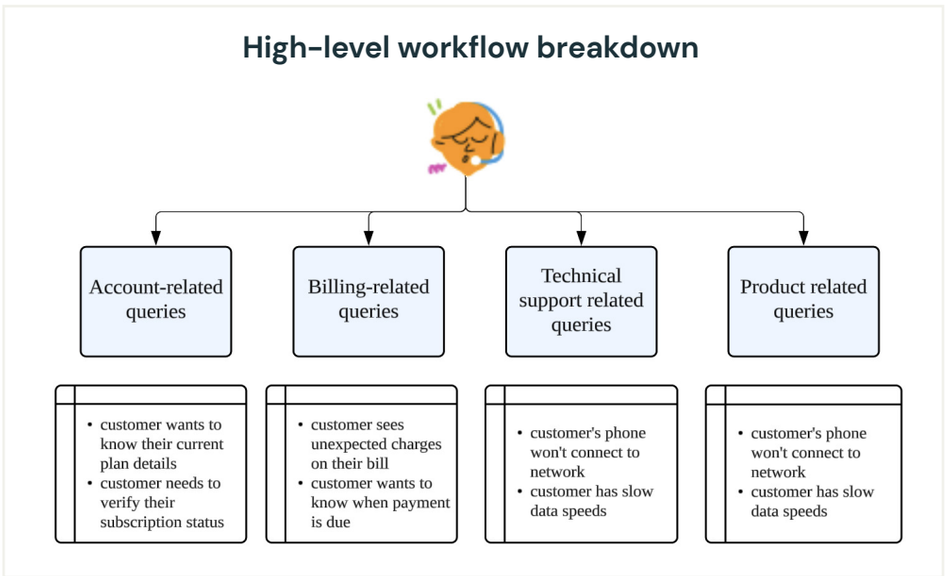
1. Map out in detail the current workflow as it is executed by humans.
2. Translate the business workflow mapped in step 1 into a technical architecture covering:
    - Which aspects of the workflow should be handled by tool-calling sub-agents versus branching but linear workflows
    - The tools available to each agent
    - The required structured and unstructured data sources
    - How each sub-agent or sub-workflow should be evaluated for quality
3. List the observability requirements of different stakeholders, from C-level executives to business users to developers.
4. Design tracing and logging modules addressing the needs listed in step 3. Tracing and logging can use sensible defaults such as those offered by `mlflow.autolog()` to begin with, then be customized further later.
5. Map out the access controls each agent and final end user requires.
6. Identify components that other projects can reuse and build abstractions accordingly.

To make these steps more concrete, in the next section we will walk through a worked example of a customer support agent designed to handle customer queries sent to a telecommunications company.

5.3 Worked Example: Telecommunications Customer Support Agent

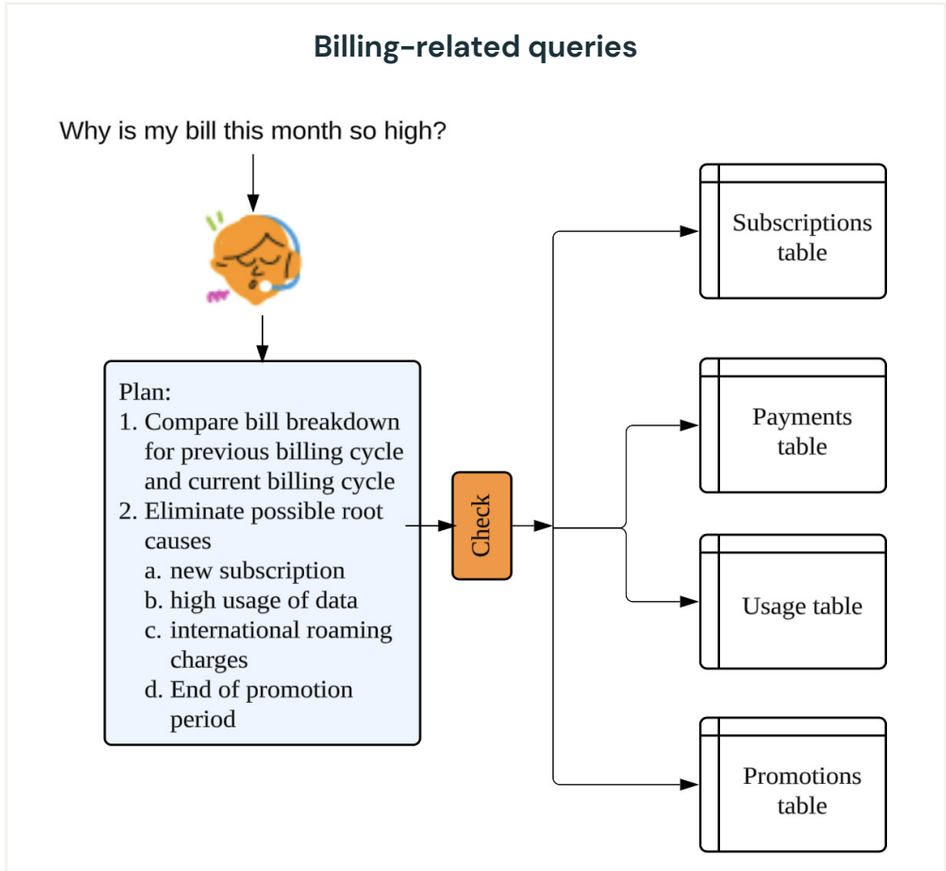
5.3.1 Mapping current human workflow

In our example, support agents primarily respond to inbound customer calls, emails, live chats, and support tickets regarding service disruptions, billing questions, plan changes, and technical problems. They must quickly diagnose issues ranging from simple password resets to complex network connectivity problems. The following diagram shows the broad categories of queries an agent might receive, as well as some example questions. Note that some questions might span multiple categories, and multiple categories might also be addressed within a single customer call.



Customer query categories and example questions

Addressing a customer query is not always straightforward. As shown in the following workflow, when handling a question related to an unexpectedly high bill, an agent may have to iteratively work through a set of scenarios to identify a root cause. This workflow involves a combination of planning, taking action, and evaluating the action's output before deciding on a next step, making it suitable to be handled by an agentic solution rather than a linear LLM workflow.



Billing-specific workflow breakdown

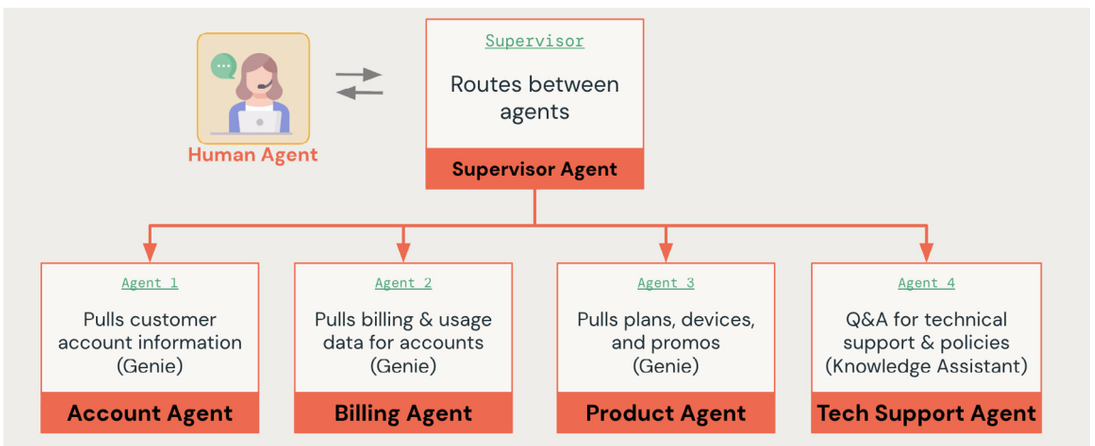
5.3.2 Translating Business Process into a Technical Architecture

From mapping out how human customer support agents manage their responsibilities, we understand that they commonly need to address queries around four broad areas: accounts, billing, products, and technical support.

Although a human can easily mentally classify and handle customer queries across all these areas, agents are different. We want to ensure that each agent has a defined set of responsibilities and toolset. By designing for modularity, we can better construct prompts, tools, and evaluations.

Therefore, we'll create one sub-agent per query category, with a supervisor agent responsible for routing questions between them. We do not expect sub-agents to communicate with each other, which simplifies our setup and reduces potential sources of indeterminism.

Here's a high-level overview of our multi-agent architecture:



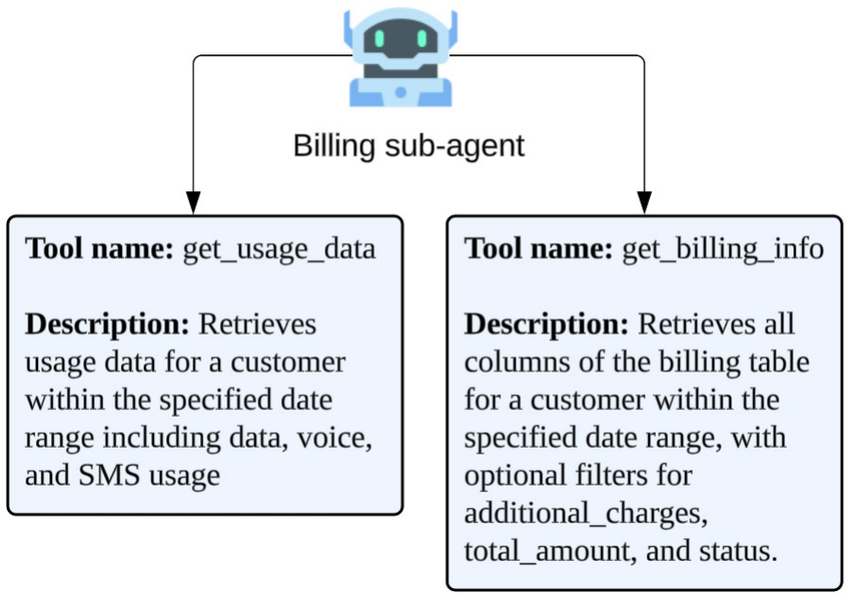
Example multi-agent architecture

The following table breaks down the roles and responsibilities of each agent.

PATTERN	WHEN TO USE	TEAM MATURITY
Supervisor agent	Workflow orchestration	Orchestrates the workflow, routes queries to specialized agents, generates responses, conducts query intent classification, performs sentiment analysis, and extracts attributes for routing decisions
Account agent	Customer profile management	Handles customer profile and subscription queries
Billing agent	Financial services	Processes billing, payment, and usage-related queries
Product agent	Product information	Manages queries about plans, devices, and promotions
Tech support agent	Technical assistance	Provides troubleshooting and technical support

IDENTIFYING AVAILABLE TOOLS

We should also map out the tools we'll provide to each agent. We'll use the billing agent as an example, as it will be responsible for our sample workflow. Here's what the billing sub-agent setup looks like:



Billing agent tools

IDENTIFYING AVAILABLE DATA SOURCES

Let's assume we have the following structured data sources available:

DATA SOURCE	SCHEMA SUMMARY
`telco_customer_support_dev.bronze.customers`	Customer profile data including customer_id, segment, location, registration_date, status, contact preferences, and scoring metrics (loyalty_tier, churn_risk_score, customer_value_score, satisfaction_score)
`telco_customer_support_dev.bronze.subscriptions`	Subscription records linking customers to plans/devices with subscription_id, customer_id, plan_id, device_id, promo_id, dates, contract terms, charges, and status
`telco_customer_support_dev.bronze.plans`	Service plan catalog containing plan_id, name, type, pricing, data limits, feature flags (unlimited calls/texts), contract requirements, and descriptions
`telco_customer_support_dev.bronze.devices`	Device catalog with device_id, name, manufacturer, type, pricing (retail/installment), specifications (storage, 5G compatibility), colors, and availability status
`telco_customer_support_dev.bronze.promotions`	Promotional offers including promo_id, name, discount details (type/value), date ranges, descriptions, and active status
`telco_customer_support_dev.bronze.billing`	Billing records with billing_id, customer/subscription references, dates, charge breakdowns (base, additional, tax), payment information, and status
`telco_customer_support_dev.bronze.usage`	Usage metrics containing usage_id, subscription_id, daily usage data (data_usage_mb, voice_minutes, sms_count), and billing cycle information

And the following unstructured data sources:

TABLE	UNSTRUCTURED CONTENT
`telco_customer_support_dev.bronze.knowledge_base`	Full content stored as Markdown text, including FAQs, policies, guides, and procedures
`telco_customer_support_dev.bronze.support_tickets`	Free-text issue descriptions from customers
`telco_customer_support_dev.bronze.support_tickets`	Free-text resolution details from support agents

EVALUATING AGENT QUALITY

The business workflows that enterprises want to automate with agents can be complex, with branches, conditional fields, and human-in-the-loop-checks needed. To manage this complexity, a common approach is to start by building a "thin slice" — a simplified version of the workflow that is just complex enough to visualize how data will flow through it — rather than attempting to build an agentic workflow that covers the entire business process in one go.

Then, for each substep within the workflow, the expected inputs and outputs are defined. What makes a "high quality" response is easier to define if outputs are structured JSON or binary true/false judgments. Structuring outputs in this way simplifies evaluation because established metrics such as F1 scores can be used. As far as possible, structured responses should be prioritized over free-text responses.

Because we also have unstructured fields, such as customer call transcripts, live-chat logs, and technician notes, developing automated LLM judges aligned with the business domain is important. We covered this in Chapter 4.3.4.

During development, the go-to person for judging "quality" should be someone who can understand both the business use case and the technical aspects. We've seen various personas take on this role, including product managers, scrum masters, data scientists attached to the business department, and technically savvy business users. How these expected inputs and outputs become concrete, runnable checks — both the deterministic tests we can define up front and the judge-based evaluations we derive from real traces — is the subject of Designing Test Suites.

5.3.3 Customizing Observability Reports and Dashboards Based on Personas

In Section 4.2, we discussed how developers can use MLflow Tracing, MLflow Assessments, and the MLflow Feedback API to collect data on an agentic system's quality. In this section, we'll expand on the concept of observability, exploring how traces and feedback can be used to provide different stakeholders with a view on how the agent is functioning in a way that addresses what they care about.

Here are some examples of stakeholders, their priorities, and the resulting interfaces, built on a foundation of data collected from MLflow traces and the Feedback API.

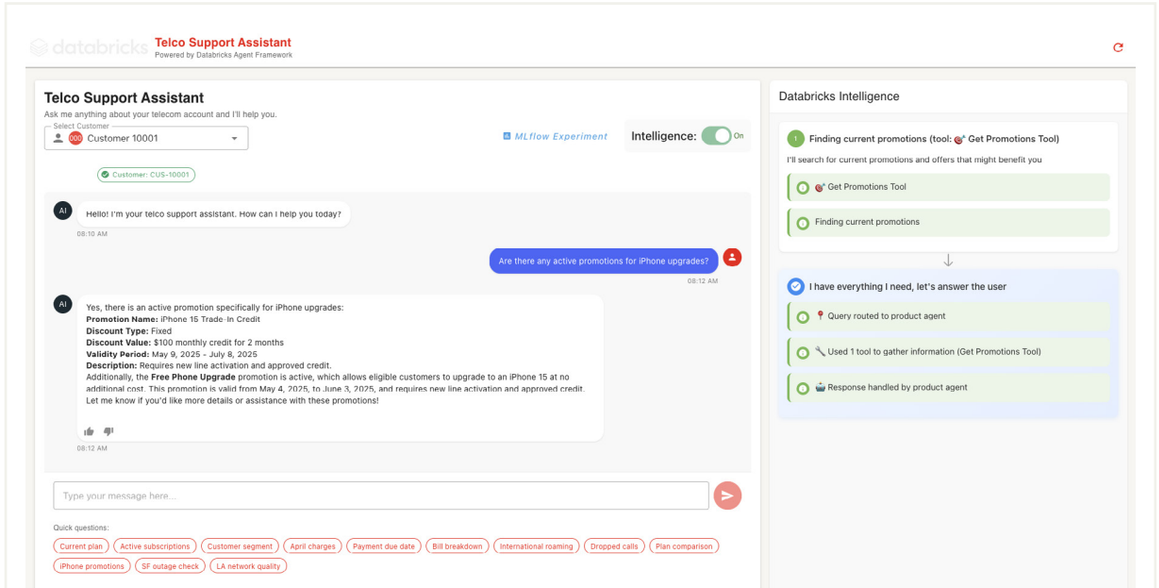
PERSONA #1: BUSINESS SME FROM THE CUSTOMER SUPPORT DEPARTMENT

Priorities:

- Transparency into agent reasoning, tool calling, and decision making
- Ability to provide feedback to troubleshoot problematic steps in the workflow
- High-level visibility into agent performance without inspecting individual traces

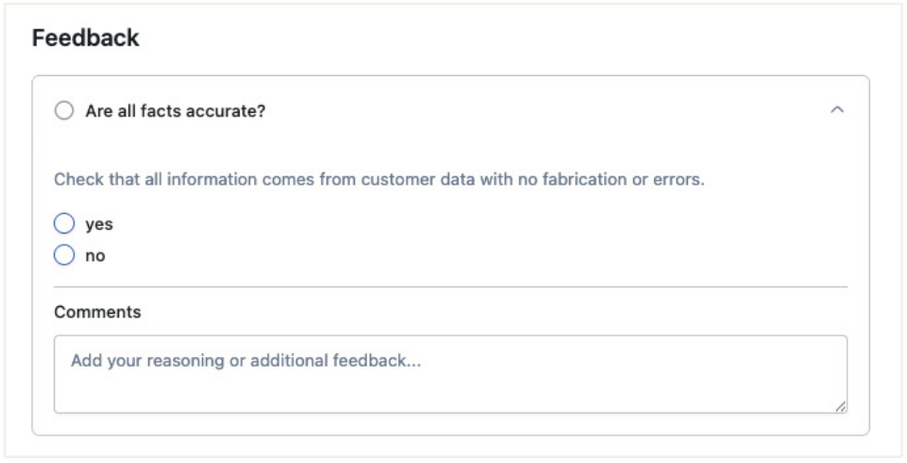
Business SME and developer user interface:

- In addition to the conversation UI, include a side panel showing an MLflow trace outlining agent reasoning, tool calling, and response generation steps.

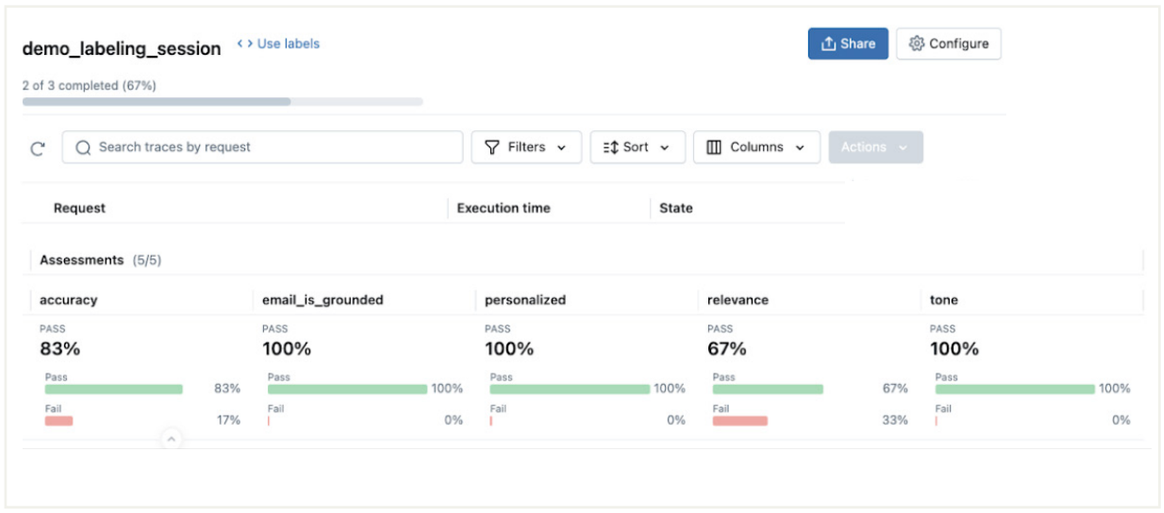


Business SME view

- Provide an interface for giving feedback on response quality. Text inputs are recorded as MLflow assessments on the backend and attached to traces. Assessments and traces are then viewable by a developer from the MLflow Experiments page in a Databricks workspace.

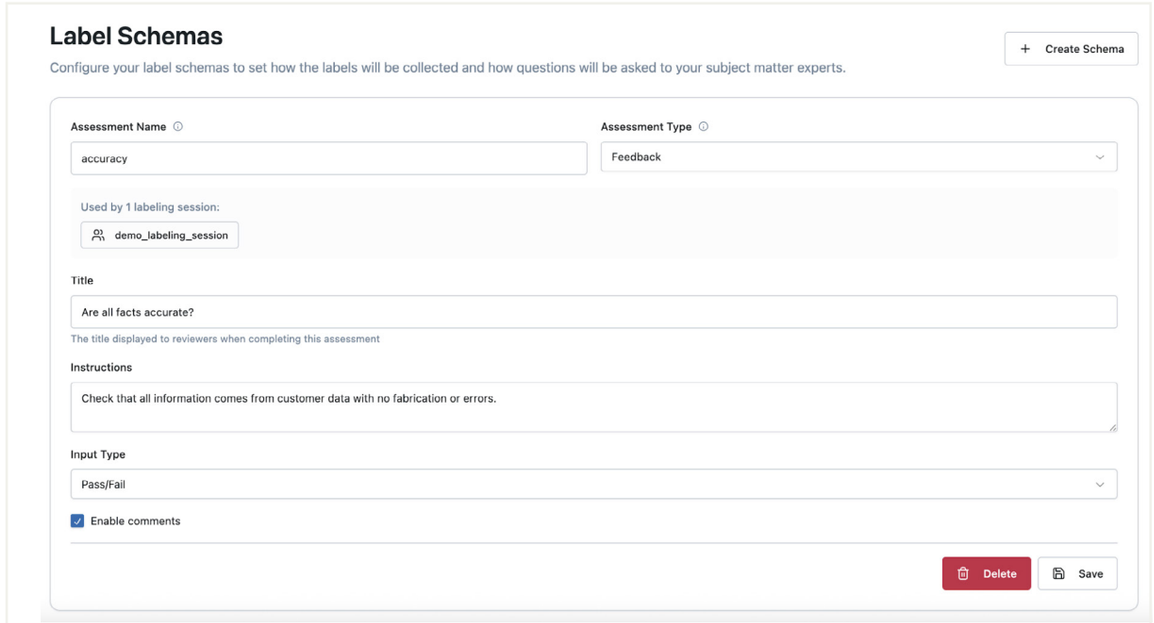


UI labeling session



MLflow experiment reflecting feedback from an SME labeling session

- Provide an interface for customizing MLflow label schemas to a specific use case.



Label schema configuration interface

- Provide a summary report, refreshed periodically, detailing key metrics.

Summary report

Raw MLflow traces are granular and information-dense. Because traces can be synced to a Delta table, developers can process raw traces to produce summaries and reports that are digestible by a non-technical audience.

Additionally, although business SMEs may not be provided user access to a Databricks workspace and the MLflow Experiments UI, hosting a feedback UI on Databricks Apps and making the app accessible via Databricks One (a Databricks interface specifically for business users) can enable SMEs to be productively involved and stay informed throughout the AI agent development process.

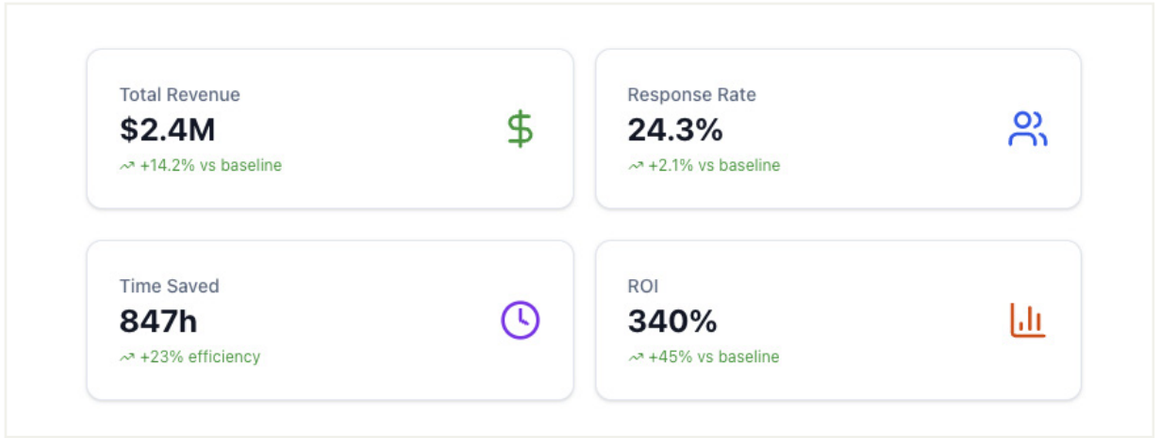
PERSONA #2: EXECUTIVE SPONSOR

Priorities:

- Return on investment, often tracked through metrics like automation rate, productivity gains, cost reductions, or revenue impact, and rapid time-to-value
- Agent response quality
- Risks and compliance

Executive sponsor view:

- Provide an ROI dashboard based on aggregated metrics from Delta tables.



ROI dashboard

PERSONA #3: CUSTOMER

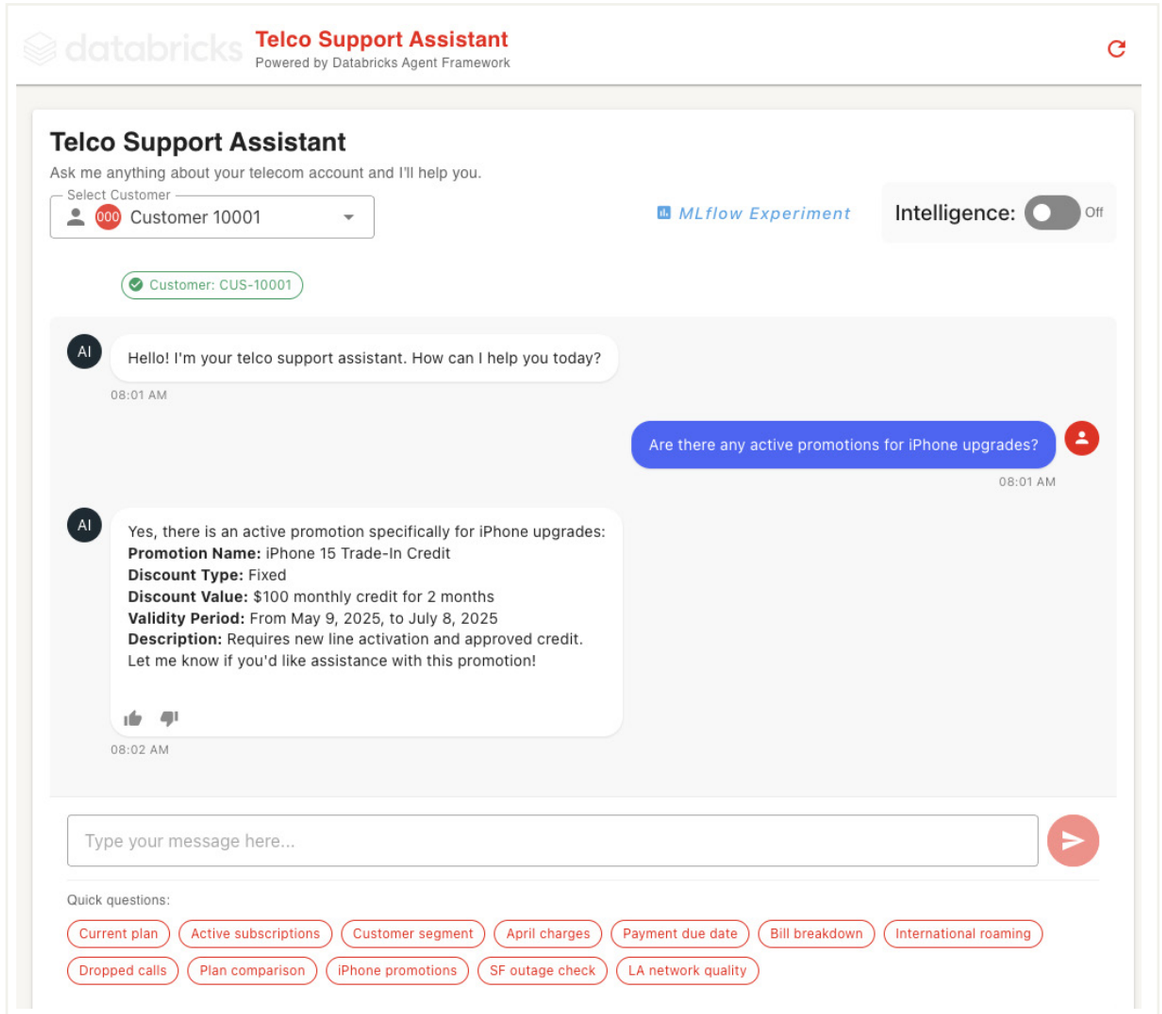
Priorities:

- Fast resolution of issue/questions
- High response reliability
- Ability to escalate issues to a human agent if needed
- Strong data privacy and security guarantees

Executive sponsor view:

- Databricks Apps, a managed app-hosting solution for building and serving interactive apps — used here to stand up a customer-facing UI, from a quick prototype through to a production deployment.
- UI might include:
  - Preset cards for commonly asked questions
  - Cited sources for reliability
  - Ability to provide quick feedback on response quality through like/dislike buttons

These feedback actions can be tied to the specific customer trace through the `mlflow.log_feedback()` API endpoint hosted on a FastAPI backend.



End user view

5.3.4 Designing Test Suites

Note that test-suite design is distinct from the tracing and logging modules of planning step 4 (covered in 5.3.3, Observability): here we address planning step 2's question of how each sub-agent and sub-workflow should be evaluated for quality

As we design our multi-agent workflow, certain aspects of the design will make it clear what evaluations we may need. For example, our billing agent will need a tool to query a table for customer spend, so we should create a test that ensures this tool works correctly. This is an evaluation that can be identified up front without analyzing any traces.

At the same time, certain failure modes only become evident after a bottom-up analysis of LLM inputs or outputs. For example, an LLM step to generate SQL that queries a database table may mix up different product acronyms. To identify these kinds of failure modes, we have to conduct an analysis.

Here are some potential ways to get started with top-down and bottom-up evaluation suite design to set up a project for success. Generally speaking, you'll need to use a combination of the two approaches.

TOP-DOWN APPROACH

One option is to generate various customer personas and scenarios and have an LLM create sample queries. In the case of our billing agent, for example, these might look like:

```
"billing": {
    "contexts": [
        QueryContext("billing", True, True, "concerned
customer", "bill inquiry"),
        QueryContext("billing", True, True, "budget-conscious
customer", "payment planning"),
        QueryContext("billing", True, True, "traveling
customer", "roaming charges"),
            ],
    "base_scenarios": [
        "customer sees unexpected charges on their bill",
        "customer wants to know when payment is due",
        "customer needs breakdown of current month charges",
        "customer is questioning data usage amounts",
    ]
},
```

BOTTOM-UP APPROACH

A more bottom-up approach, where we conduct error analysis based on sample traces, then align LLM judge outputs with SME feedback to create automated tests, is outlined in detail in Chapter 6. In this chapter, we move beyond a purely developer-focused way of constructing evaluations and explore how to also gather analyses and feedback from non-developer stakeholders.

5.3.5 Access Control

Planning step 5 asks us to map out the access controls each agent and end user requires. In an agentic system this is more involved than in a traditional application, because permissions apply at two distinct levels: what each agent (and its tools) is allowed to do, and what the end user on whose behalf the agent is acting is allowed to see. Getting this wrong is not only a security risk since this could expose sensitive data to unauthorized viewers.

AGENT- AND TOOL-LEVEL PERMISSIONS

Each sub-agent should operate with the least privilege it needs. Returning to our telco example:

- The billing sub-agent needs read access to customer spend and invoice tables, but no write access to account settings or plan configurations.
- The account-management sub-agent may need write access to update contact preferences, gated behind an explicit confirmation step.
- The technical-support sub-agent needs read access to network and device diagnostics, but not to billing data.
- The escalation path needs permission to open a human-review ticket, but not to resolve or close it autonomously.

Scoping tools this narrowly contains the blast radius of a misbehaving or compromised agent: even if the LLM is manipulated into calling a tool inappropriately, the tool itself cannot act outside its granted scope.

END-USER-LEVEL PERMISSIONS

Agent-level permissions are necessary but not sufficient. Because a single deployed agent serves many customers, the identity and permission scope of the end user must flow through every asset accessed so a user only ever sees information they are authorized for. Two common patterns:

- Identity passthrough (on-behalf-of-user): the agent queries using the end user's identity, so a user's existing row- and column-level access policies apply automatically.
- Explicit parameterization: where passthrough isn't available, the authenticated customer ID is injected as a trusted, non-LLM-controlled filter on every query — never taken from the model's output, which could be manipulated.

ENFORCING CONTROLS CENTRALLY

On Databricks, both levels are expressed through Unity Catalog: grants on the tables, functions, MCP servers and tools each agent can access will define the agent-level permissions scope. This is defined inside the Databricks Apps configuration file that maps what Unity Catalog assets are accessible by a Databricks Apps Service Principal while row- and column-level security is enforced with on-behalf-of-user identity passthrough end-user scope.

Centralizing these controls in the governance layer — rather than in agent or prompt logic — keeps permissions auditable, consistent across agents, and resistant to prompt-injection attempts to talk the agent out of them. Every tool call and data access is also captured in the audit trail saved to a Unity Catalog System Table, which feeds the observability and compliance reporting discussed in 5.3.3.

5.3.6 Future Planning

The organizations we work with often have long use case backlogs. They need a system for reliably developing and deploying not a few but dozens or sometimes hundreds of generative AI use cases, distributed across multiple development teams spread across different departments.

In such a scenario, identifying components that other projects and teams can reuse and building appropriate abstractions is vital. Shared abstractions and reusable helper modules standardize how applications are developed, save time by eliminating duplicate work, and ensure that all teams conform to best practices, especially around security, data privacy, and evaluations.

Here are a few examples of shared abstractions that teams can use. These examples are also available in the Databricks Field Solutions GitHub repository.

EXAMPLE 1: BASE AGENT CLASS

In this example, a base agent class sets out some standard interfaces that all agents should implement, including:

- A uniform way of specifying Unity Catalog assets through a UCConfig class
- A common way of easily initializing an agent through specifying a name as a string, which then calls the relevant configuration file

Although not used in the following example, this class could also include class methods for initializing machine-to-machine authentication through the Databricks Workspace Client:

```
class BaseAgent(ResponsesAgent, abc.ABC):
    """Base agent class all agents inherit from."""

    _config_cache: dict[str, AgentConfig] = {}

    def __init__(
        self,
        agent_type: str,
        llm_endpoint: Optional[str] = None,
        tools: Optional[list[dict]] = None,  # UC function tools
        vector_search_tools: Optional[
            dict[str, Any]
        ] = None,  # Map of tool name -> VectorSearchRetrieverTool (not UC
function)
        system_prompt: Optional[str] = None,
        config_dir: Optional[Path | str] = None,
        inject_tool_args: Optional[list[str]] = None,
        disable_tools: Optional[list[str]] = None,
        uc_config: Optional[UCConfig] = None,
    ):
        """Initialize base agent from config file.

        Args:
            agent_type: Type of agent (used for config loading)
            llm_endpoint: Optional override for LLM endpoint
            tools: Optional list of UC function tools
            vector_search_tools: Optional dict mapping tool names to
VectorSearchRetrieverTool objects
            system_prompt: Optional override for system prompt
            config_dir: Optional directory for config files
            inject_tool_args: Optional list of additional tool arguments to be
injected into tool calls from custom_inputs.
            disable_tools: Optional list of tool names to disable. Can be simple
names or full UC function names.
            uc_config: Optional UC configuration for Unity Catalog resources
        """
```

EXAMPLE 2: BASE CLASS FOR EVALUATION SCORERS

MLflow provides many options for creating evaluation scorers. Teams new to agent development may not be familiar with how to select the right evaluation APIs for their use case. The make_judge API in MLflow is a unified interface for all types of judge-based evaluation. Instead of spending development cycles trialing different evaluation scorers, teams can get up and running quickly with an opinionated wrapper built on top of make_judge, like the following:

```
from abc import ABC, abstractmethod
from typing import Any, Optional

from mlflow.entities import Feedback, Trace
from mlflow.genai.judges import custom_prompt_judge, meets_guidelines
from mlflow.genai.scorers import scorer
from mlflow.genai.scorers.builtin_scorers import BuiltInScorer


class BaseScorer(ABC):
    def __init__(self, name, sample_rate):
        self.name = name
        self.sample_rate = sample_rate

    @abstractmethod
    def get_make_judge_scorer(self):
        """Implementation of LLM judge evaluation wrapping MLflow make_
judge"""
        pass
```

EXAMPLE 3: MLFLOW CLI TOOL FOR COMMON TASKS

Throughout development and evaluation, AI engineers across teams will use MLflow for a common set of tasks, such as creating labeling sessions and generating evaluation reports.

One way to increase productivity is to provide a CLI wrapper for these common tasks. The benefit is twofold: Developers can invoke standardized tools rather than writing their own scripts, and these tools can also be provided to AI coding agents (such as Claude) to further speed up development.

Here's an example of a Claude code prompt that leverages helper Python scripts to set up a labeling session for business SMEs:

```
# Label Schema Management

Manage labeling schemas for your MLflow Review App. This command helps you
view, create, modify, and delete schemas interactively.

## Usage

```bash
/label-schemas [action]
```
```

/label-schemas shows all current labeling schemas, with details about the schema type, options, and instructions.

When available, references experiment summaries from experiments/[experiment_id]_summary.md to suggest schemas tailored to your specific agent's capabilities and use cases.

We could also add descriptions to each schema.

```
/label-schemas add [description]
```

Examples:

- `/label-schemas` add quality rating from 1 to 5
- `//label-schemas` add helpfulness categorical with options: Very Helpful, Helpful, Not Helpful
- `/label-schemas` add feedback text field for comments

5.4 Summary

Effectively executing the planning activities and building the technical components described in this chapter requires clear roles, defined responsibilities, and collaboration patterns suited to the unique nature of agent development. The next chapter examines how organizations are evolving their teams and roles for AgentOps.

CHAPTER 6

Stakeholder Management

AgentOps projects change how organizations plan, deliver, and operate software. Because agentic systems are nondeterministic and evolve quickly, success depends on proactive stakeholder management, including clear ownership, short feedback loops, transparent metrics, and explicit decision records.

Agents introduce new metrics and operational tasks, like reconciling SME feedback with automated judges. Even "standard" reliability metrics behave differently; for instance, a reasoning LLM can produce highly variable latency by query. So while the metric names are unchanged, the way you evaluate them should be.

This chapter provides a practical stakeholder playbook aligned to the project lifecycle outlined in Chapter 3. We'll cover stakeholder mapping, a lightweight responsibility assignment (RACI) matrix, communication cadences, lifecycle touchpoints, and success metrics.

Note that this level of management might not be necessary for every single use case, but it is critical in complex, enterprise use cases that require a high bar for quality and involve many cross-functional stakeholders.

6.1 Stakeholder Map

The first step in stakeholder management is to identify the minimum viable set of stakeholders and the value they expect. The following is the full superset of potential stakeholders that could be involved in a particular use case. Bear in mind that in smaller organizations several of these roles may be held by the same person, and not all of them will be relevant for all use cases.

STAKEHOLDER	PRIMARY CONCERNS	SUCCESS SIGNALS
Executive sponsor	Strategic fit, risk, ROI, reputation	Clear ROI narrative, governance in place, predictable spend
Product manager	Problem–solution fit, adoption, user satisfaction	Task success rate rising, active users, fewer escalations
AI engineer	Output quality, latency, cost, reliability	Stable prompts/tools, improving evaluation scores, low incident rate
Data engineer	Data availability, quality, governance	Reliable pipelines, documented lineage, privacy preserved
Platform engineer	Scalability, SLAs, cost control, provider management	SLOs met, budget adherence, safe rollout procedures
Software engineer	Integration UX, APIs, error handling	Clean contracts, graceful fallbacks, low UI regressions
SME (domain expert)	Domain accuracy, compliance, usefulness	High SME agreement, fewer critical misinterpretations
Legal/compliance	Regulatory exposure, auditability	Guardrails, audit trails, documented approvals
Security/privacy	Data leakage, supply-chain risk, access control	Passing red-team tests, no secret exposure, auditable access controls
Finance/FinOps	Cost predictability, unit economics	Cost per interaction trending down, budgets met

6.2 Lightweight RACI Matrix for Key Decisions

Next, identify who is responsible for key decisions. Use a pragmatic responsibility assignment matrix to avoid ambiguity: R = Responsible (does the work), A = Accountable (final decision), C = Consulted, I = Informed. Keep it short and revisit it monthly.

DECISION	R	A	C	I
Use case selection and scope	Product manager	Executive sponsor	SME, AI engineer	All
Project resourcing	Executive sponsor	Product manager	All	All
Data access/PII handling	Data engineer	w/compliance	Security, product manager	Executive sponsor
Architecture & development	AI engineer	Product manager	Security, platform engineer, data engineer	Legal/compliance
Evaluation & guardrails	AI engineer, SME	Product manager	Legal/compliance, executive sponsor	Platform engineer
Budget/FinOps guardrails	Platform engineer	Finance/FinOps	Product manager, executive sponsor	AI engineer
Deployment & rollout plan	Platform engineer	Product manager	Software engineer	All
Incident response & escalation	Platform engineer	Executive sponsor	AI engineer, product manager	Security
Prompt/tooling change management	AI engineer	Product manager	SME, platform engineer	Legal/compliance
Governance reviews/audits	Legal/compliance	Executive sponsor	Platform engineer, product manager	All

6.3 Communication Cadence

Like for most projects, the focus and cadence of communication for GenAI initiatives should be dictated by the stage of the product development process.

In the early stages, the focus should primarily be on team formation, cross-functional metrics definition, and SME data labeling and feedback to enable fast iteration. It is a best practice to share early and often. Because SME feedback can directly impact application performance, waiting for perfection before sharing with internal SMEs can significantly slow iteration.

Feedback may address general response quality or tone or be more concrete, such as flagging missing documentation that should be included in responses. Both forms of feedback are valuable for informing agent development and building labeled datasets that can be used for evaluation.

In later stages of the project, the focus should shift toward production concerns, such as operations, cost optimization, production monitoring, and roadmap management.

The following touchpoints are recommended.

6.3.1 Early-Stage (Development, Pre-Production)

INITIAL PROJECT KICKOFF (30–45 MIN)

- **Objective:** Align on agent objectives, identify cross-functional stakeholders, define agent metrics
- **Objective:** Product manager
- **Artifacts:** Project plan, defined model metrics, stakeholder map, RACI chart

BIWEEKLY OR MONTHLY PRODUCT DEMO (30–45 MIN)

- **Objective:** Demonstrate working prototypes, review evaluation metrics, collect SME feedback
- **Objective:** Product manager
- **Artifacts:** Functioning prototype, top regressions, prioritized roadmap

TWICE-WEEKLY BUILD REVIEW (15–20 MIN, OR ASYNCHRONOUS)

- **Objective:** Plan/confirm experimental changes, track evaluation runs and acceptance thresholds
- **Objective:** AI engineer with product manager
- **Artifacts:** Experiment log, evaluation dashboard snapshot, change requests

SME REVIEW SESSIONS (60–90 MIN, AS NEEDED)

- **Objective:** Refine rubrics, review edge cases, validate domain correctness, discuss open-ended feedback
- **Objective:** Product manager with SME
- **Artifacts:** Golden set updates, judge calibration notes, acceptance criteria (all tracked in MLflow)

OPS DRY RUN (60 MIN, PRIOR TO FIRST RELEASE)

- **Objective:** Create and validate runbooks, harden metrics, align on deployment pipeline plan for beta testing and production
- **Objective:** Platform engineer
- **Artifacts:** Runbook check, alert test results, rollback plan, CI/CD architecture

6.3.2 Late-Stage (Production, Scaling)

DAILY OPS TRIAGE (15 MIN)

- **Objective:** Scan for incidents, cost spikes, degraded evaluations, provider issues
- **Owner:** Platform engineer
- **Artifacts:** Open incidents, SLO dashboard, mitigation status

WEEKLY PRODUCT/BUSINESS REVIEW (30–45 MIN)

- **Objective:** Track adoption and ROI, prioritize fixes/features against metric movements
- **Owner:** Product manager
- **Artifacts:** Adoption funnel, task success trend, cost per interaction, top regressions, updated roadmap

MONTHLY GOVERNANCE REVIEW (45–60 MIN)

- **Objective:** Review risks, policy changes, and audit items; confirm compliance posture
- **Owner:** Legal/compliance with executive sponsor
- **Artifacts:** Decision log summary, change summaries, audit trail exports

AS-NEEDED INCIDENT CHANNEL

- **Objective:** Coordinate P1/P2 incidents and external comms
- **Owner:** Platform engineer
- **Artifacts:** Incident timeline, stakeholder updates, root cause analysis template and publication

QUARTERLY ROADMAP/FINOPS REVIEW (45–60 MIN)

- **Objective:** Evaluate unit economics and model/provider mix, review ops plan, analyze market trends
- **Owner:** Product manager with finance and platform engineers.
- **Artifacts:** Cost per interaction trends, vendor mix report, savings opportunities backlog, changes to ops architecture

6.4 Success Metrics and Dashboards

Quality metrics

Defining quality metrics is often the most challenging aspect of communicating project updates to stakeholders. Ensuring that metrics are cross-functionally defined and clearly communicated at the start of a project is critical. Manual reviews of agents by stakeholders are important, but given the complex nature of agent evaluation, this can become an anti-pattern if subjective feedback from an influential stakeholder disproportionately drives the project roadmap.

Operational metrics

Operational metrics, such as response latency and number of successful responses, typically mirror those used in traditional software systems, but they can have a much higher variance given the rapid pace of innovation, model releases, and changes in pricing. Continuous monitoring for spikes, regressions, and optimization opportunities is essential in this ever-evolving landscape.

The right mix of metrics depends on the use case and whether the agent is customer-facing or used in internal applications. Typical metrics to track include:

- **Quality:** Task success rate, SME agreement rate, harmful output rate, judge score trends, hallucination rate
- **Operations:** P95 latency, error rate, incident mean time to resolve (MTTR), prompt/tool change failure rate
- **Cost:** Cost per interaction, token/input-output mix, cache hit rate, vendor model mix
- **Adoption/business:** Active users, retention, time saved, Customer Satisfaction Score (CSAT)/Net Promoter Score (NPS), revenue or cost avoidance

Recommended views to provide include:

- **Executive view:** ROI narrative, risk trend, budget vs. actual spend
- **Product/engineering view:** Experiment throughput, regression alerts, guardrail trips
- **Ops view:** SLO compliance, incident heatmap, provider health

This closes the AgentOps lifecycle covered in this book: from establishing operational challenges and deployment patterns, through the project pipeline and DevOps principles of flow, feedback, and continuous learning, to prioritizing high-leverage work and managing the stakeholders who depend on these systems. Reliable, observable, safe, scalable, maintainable agentic systems that reflect enterprise-specific context are the goal throughout — the practices in this book are how teams get there.