Building Single-Agent Applications on Databricks

1. Foundations of AI Agents and Tools on Databricks

Modern AI applications require agents that can interact with data, perform analytical tasks, and make informed decisions based on available information. By understanding these foundational concepts, you'll be
prepared to build robust, scalable AI solutions that combine governance, security, and analytical power.

AI agents represent a revolutionary shift from traditional AI systems that simply provide information based on
user prompts. Instead, agents use available tools to help them make more accurate and informed decisions,
acting autonomously within their environment to achieve user-defined goals.

A. Understanding AI Agents

A1. What Are AI Agents?

An AI agent is an intelligent software system that can perceive its environment, make decisions, and take actions to achieve specific goals. Unlike traditional AI systems that require continuous inputs from users,
AI agents are autonomous systems that can:

- Reason about complex problems and situations
- Plan sequences of actions to achieve objectives
- Adapt their behavior based on new information
- Interact with external systems and data sources
- Learn from experience to improve future performance

What makes AI agents exciting is their adaptability. They use tools that dynamically pull up-to-date datasets to
inform decisions and processes, making them ideal for complex and unpredictable tasks. While

humans set the goals,
AI agents determine the best way to achieve those goals.

In the context of data analytics and business intelligence, AI agents serve as intelligent intermediaries between
users and data systems, capable of understanding natural language queries and executing complex analytical tasks.

A2. Evolution of AI Agents:

AI agents have evolved significantly since their inception:

i. 1960s - Rule-Based Systems:

- Basic chatbots with predetermined logic trees
- Rigid, rule-based programming
- Limited to simple, scripted responses

ii. 1990s - Statistical Learning:

- More autonomous systems processing information
- Simple decision-making capabilities
- Foundation for consumer-grade AI devices

iii. 2000s - Machine Learning Integration:

- Consumer devices like robot vacuums and digital assistants (Siri, Alexa)
- Statistical machine learning models and neural networks
- Enhanced decision-making and analysis capabilities

iv. 2020s - Large Language Models:

- Breakthrough with deep reinforcement learning and transformer-based large language models (LLMs)
- Multimodal interfaces and advanced reasoning
- Dynamic interaction with complex environments
- Tool-calling capabilities for enhanced functionality

A3. Key Principles of AI Agents:

AI agents operate on three fundamental principles that distinguish them from traditional software:

i. Perception:
The first step for agents to understand the context in which they're operating. For language models, this includes:

- User inputs and queries via text, photos, or audio
- Environmental data from sensors or APIs
- Historical context and conversation memory

ii. Decision-Making:
The agent processes collected information through algorithms and determines proper actions according to user goals:

- Analyzing requirements and constraints
- Determining necessary steps and tool usage
- Planning optimal execution sequences

iii. Action:
Finally, an agent takes concrete steps to achieve objectives:

- Executing database queries and API calls
- Processing and transforming data
- Generating reports and recommendations
- Making decisions that affect real-world outcomes

A4. Core Components of AI Agents:

Modern AI agents typically consist of several key components working together:

- Large Language Model (LLM) Brain: The central reasoning engine that processes natural language,
  understands context, and makes decisions about what actions to take.
- Memory System: Stores conversation history, context, and learned information to maintain coherent interactions
  over time.
- Planning Module: Breaks down complex requests into smaller, manageable tasks and determines the optimal sequence
  of actions.
- Tool Interface: Connects the agent to external systems, databases, APIs, and functions that extend its
  capabilities beyond text generation.

- Execution Engine: Manages the actual execution of planned actions and handles responses from external tools and
  systems.

> Example agent pattern: The LLM acts as the brain to plan and execute tasks within its environment based on the
> user's request. Tools can be stored securely within Unity Catalog while agent memory can be used with Delta Lake
> and Lakebase.

A5. Types of AI Agents by Complexity:

AI agents differ based on their complexity and application. Understanding these types helps in selecting the
right approach for specific use cases:

Agent Type	Description	Example
i. Simple Reflex Agents	Make decisions based on current conditions only	Robot vacuum that cleans only when it senses dirt
ii. Model-Based Reflex Agents	Account for current state and use world models to guide actions	Smart thermostat adjusting based on time, weather, and preferences
iii. Goal-Based Agents	Plan specific strategies to achieve desired goals	Navigation systems like Google Maps considering traffic and routes
iv. Utility-Based Agents	Evaluate multiple ways to achieve goals for optimal efficiency	AI trading bots adjusting investment strategies
v. Learning Agents	Learn from past actions and adapt to future situations	Recommendation systems that improve based on user behavior

A6. Can All LLMs Use Tools?

No, not all LLMs have the tool-calling capability. On Databricks, tool usage by LLMs is enabled through specific
frameworks and integrations, such as Databricks Assistant or custom agent frameworks that allow LLMs to interact
with external systems, databases, or APIs. This capability is not inherent to all LLMs; it requires additional
engineering, orchestration, and security controls to ensure safe and effective tool usage.

For example, Databricks Assistant is designed to use tools to answer questions and perform actions within the
Databricks environment, but this is a feature of the platform, not a universal capability of all LLMs.

> For a complete list of Foundation Model APIs that can perform tool-calling, please read more here.

B. Understanding Agent Tools:

B1. What Are Agent Tools?

Agent tools are specialized functions or capabilities that extend an AI agent's ability to interact with external
systems and perform specific tasks. Think of tools as the "hands" of an AI agent, while the LLM provides the
"brain" for reasoning and decision-making—tools enable the agent to actually manipulate data, call APIs, perform
calculations, and interact with the real world.

Tools transform agents from purely conversational systems into actionable, productive assistants. Some examples include:

- Execute database queries and retrieve specific information
- Perform complex calculations and statistical analysis
- Interact with external APIs and web services
- Process and transform data in various formats
- Generate reports, visualizations, and summaries
- Make real-time decisions based on current data

B2. How Tools Differ from Traditional AI Components

It's important to understand how agent tools relate to other AI technologies. Here are examples to help
distinguish between tools and machine learning models, chatbots, and traditional APIs:

i. Tools vs. Machine Learning Models:

- ML Models: Provide intelligence (prediction, generation, reasoning) used by agents

- Agent Tools: Executable capabilities an agent can call to take action or retrieve information — some tools may
  call ML models, others may call APIs, databases, or run business logic
- Example: A sentiment model scores a customer message; the agent uses a tool (e.g., `escalate_ticket`) to act
  based on that score

ii. Tools vs. Chatbots:

- Chatbots: Provide conversational responses within a bounded scope (scripts, retrieval, predefined flows)
- Agent Tools: Allow an agent to go beyond responding — the agent can decide to execute actions
  (e.g., search a database, send an email, write to a record)
- Key Point: Chatbots converse; agents use tools to do things in the real world

iii. Tools vs. Traditional APIs:

- Traditional APIs: Require manual programming to choose and call functions
- Agent Tools: Can be dynamically selected and orchestrated by AI reasoning based on context and goals
- Intelligence: Tools expose metadata and descriptions so the agent understands when and how to use them

B3. Tool Selection and Orchestration

One of the key capabilities of modern AI agents is intelligent tool selection. When presented with a user request,
the agent must:

1. Analyze the Request: Understand what the user is trying to accomplish
2. Identify Required Tools: Determine which tools are needed to fulfill the request
3. Plan Execution Order: Decide the sequence in which tools should be called
4. Execute and Coordinate: Call tools with appropriate parameters and handle responses
5. Synthesize Results: Combine outputs from multiple tools into a coherent response
6. Learn and Adapt: Improve tool selection based on success patterns

This orchestration capability allows agents to handle complex, multi-step workflows automatically, making them
ideal for scenarios requiring dynamic problem-solving approaches.

C. Unity Catalog and Agent Tool Governance

It's important to understand the tooling ecosystem on Databricks, so you can decide which tool use case is best
for you. Currently, there are three options for creating agent tools:

1. Unity Catalog function tools: This is the primary focus of this course. Your tools are defined as UC UDFs and
   are managed in UC as a central registry for your agent's tools. This allows for built-in security and compliance
   features while granting easier discoverability and reuse.
2. Agent-code tools: These are tools defined directly in an agent's code. This is best for calling REST APIs,
   running arbitrary code, or running low-latency tools. However, this approach lacks built-in governance and
   discoverability that UC brings to the table.
3. Model Context Protocol (MCP) tools: These are tools that follow the MCP standard for tool interoperability.
   Databricks-managed MCP servers are currently available and you can check the release status here.

C1. Why Unity Catalog for Agent Tools?

Now that we have an understanding of what the basics of what makes an agent an agent, let's turn our attention
to understanding how Databricks enables tool calling by first looking at where tools are stored and how they're
governed on the platform via Unity Catalog.

Traditional tool calling lacks comprehensive governance. With Unity Catalog, users can build tools for retrieving
structured and unstructured data and test those tools in the AI Playground. When connecting external tools
(such as Slack, Google Calendar, or any API service) via Unity Catalog connections, the management of
credentials and authentication is governed by Unity Catalog policies. This means you can ensure secure,
auditable access and apply organization-wide governance for integrations with external services.

Unity Catalog provides the foundation for agent tool management with enterprise-grade capabilities:

i. Centralized Governance:

- Unified object model and three-level namespace for data and AI assets, including functions, across all
  UC-enabled workspaces.
- Built-in auditing and lineage, with system tables to simplify access and analysis.
- Consistent metadata and discoverability via Catalog Explorer and search.
- Governed tools: functions registered in UC can be used as agent tools, enabling reuse and control.

ii. Security and Access Control:

- Fine-grained permissions (ANSI GRANTs), including EXECUTE on functions/tools.
- Centralized identity integration (SCIM, account-level identities) for consistent access across workspaces.
- Secure, isolated execution for Python UDFs; governed access to external connections and locations.
- Python UDFs require Unity Catalog and either a serverless/pro SQL warehouse, or UC-enabled clusters.
- Role-based hierarchical privileges aligned to catalogs → schemas → objects (tables, views, volumes, models,
  functions).

iii. Discoverability and Documentation:

- Searchable catalog with rich metadata (function and parameter comments), lineage, and browse capabilities.
- Recommended docstrings for functions (purpose, parameters, return value, examples, change log) to aid tool
  calling.
- Platform support for AI-powered documentation to accelerate discovery for governed assets.

iv. Scalability and Performance:

- UC-governed tools run via Databricks compute; agent tool execution uses serverless generic compute
  (Spark Connect serverless). Some integrations can execute UC functions via SQL Warehouses (`uc_function`).

- Scaling and concurrency controls on SQL Warehouses; autoscaling on clusters to match workload demand.

v. External Tool Support:

- When connecting external tools (such as Slack, Google Calendar, or any API service) via Unity Catalog
  connections, the management of credentials and authentication is governed by Unity Catalog policies.
  This means you can ensure secure, auditable access and apply organization-wide governance for integrations
  with external services.

C2. Tool Registration and Management

Unity Catalog provides multiple approaches for registering and managing SQL-based agent tools. For tracing tool
usage with your agent, Databricks leverages its managed MLflow agent tooling features like automatic signature
and tracing and ResponseAgent interface. This course will concentrate on the fundamentals of tool calling
by keeping the tool logic simple and direct (e.g. we will not go into an agent's ability to perform vector
search) so you can focus on how to use the Databricks platform for developing agents.

i. SQL-Based Registration:
Using `CREATE OR REPLACE FUNCTION` statements with comprehensive metadata that can be used with LLMs:

- Clear parameter definitions with types and descriptions
- Function-level documentation and usage guidance
- Deterministic behavior specifications
- Built-in validation and error handling

ii. Programmatic Registration:
Using the `DatabricksFunctionClient()` for automated tool management:

- Programmatic creation and updates
- Integration with CI/CD pipelines
- Batch operations and bulk management
- Automated testing and validation workflows

iii. Documentation Best Practices:
SQL functions should include rich metadata to help AI agents understand their purpose:

- Comprehensive function comments explaining business logic
- Parameter descriptions with expected data types and ranges
- Return value specifications and example outputs
- Usage examples and common patterns
- Mark functions `DETERMINISTIC` where appropriate

Both approaches ensure that SQL functions are properly documented, versioned, and accessible to AI agents while
maintaining governance and security standards.

> Note: MLflow is foundational for building, monitoring, and deploying agent-based applications on Databricks, providing
> robust tracing, versioning, evaluation, and production deployment, especially when working with agent tooling.

C3. Other Tools and Common Patterns

While we will focus on agent tools in UC, it's important to point out that there are other tools that exist
outside those discussed in this course.

i. Model Context Protocol (MCP):
The main benefit of MCP is standardization. You can create a tool once and use it with any agent—whether it's
one you've built or a third-party agent. Similarly, you can use tools developed by others, either from your team
or from outside your organization.

You can read more about MCP on Databricks here.
You can also read the official MCP documentation here.

ii. Mosaic AI Vector Search:
Mosaic AI Vector Search is a vector search solution that is built into the Databricks Data Intelligence Platform and integrated with its governance and productivity tools. Vector search is a type of search optimized for retrieving embeddings.

You can read more about vector search here.

iii. Common Tool Patterns:

Tool Pattern	Description
Structured data retrieval tools	Query SQL tables, databases, and structured data sources.
Unstructured data retrieval tools	Search document collections and perform retrieval-augmented generation.
Code interpreter tools	Allow agents to run Python code for calculations, data analysis, and dynamic processing.
External connection tools	Connect to external services and APIs such as Slack.
AI Playground prototyping	Use the AI Playground to quickly add Unity Catalog tools to agents and prototype behavior.

Conclusion:

You now have a comprehensive foundation in the concepts and principles underlying AI agents and UC function
tools on Databricks. This lecture has covered the evolution of AI agents from simple rule-based systems to
today's sophisticated, tool-enabled systems that are transforming industries worldwide.

Key takeaways from this lecture include:

- AI agents are autonomous systems that combine perception, decision-making, and action capabilities to solve
  complex problems.
- Agent tools extend AI capabilities by providing interfaces to external systems, data sources, and specialized functions.
- Unity Catalog provides the governance, security, and management framework needed for enterprise-grade agent
  tool deployment.

Now that you have an understanding of what an agent is and how tools are a core component of agentic behavior,
let's dive a little deeper into UC tools on Databricks.

Unity Catalog Functions as Agent Tools on Databricks

2.

Imagine an AI agent that can instantly answer questions like "What's the average home price in the Mission district?" or "Calculate the customer lifetime value for our top clients" by automatically discovering and executing the right data operations and business logic.

This is the power of Unity Catalog Functions as Agent Tools.

Building on the foundational concepts of AI agents and tools, this session focuses on one of the most practical implementations: using both Unity Catalog SQL and Python Functions as intelligent, discoverable tools that AI agents can automatically select and execute based on natural language queries.

This lecture establishes the technical foundation and best practices needed for the hands-on demonstrations that follow, where you'll build and test both SQL and Python Unity Catalog Functions as Agent Tools using AI Playground.

A. Understanding Unity Catalog Functions as Agent Tools:

A1. What Are Unity Catalog Functions as Agent Tools?:

Before getting started, it's important to keep in mind what UC functions are. Unity Catalog tools are really just Unity Catalog user-defined functions (UDFs) under the hood.

When you define a Unity Catalog tool, you're registering a function in Unity Catalog. To learn more about Unity Catalog UDFs, see this documentation.

What's a UDF? User-defined functions (UDFs) in Unity Catalog extend SQL and Python capabilities within Databricks. They allow custom functions to be defined, used, and securely shared and governed across computing environments.

Unity Catalog Functions as Agent Tools are Unity Catalog functions written in either SQL or Python that can be dynamically discovered, selected, and executed by AI agents to perform data operations and business logic. Unlike traditional functions that require manual programming to call, Unity Catalog Functions as Agent Tools are designed to be:

- Self-describing through comprehensive metadata and documentation

- Contextually appropriate for specific business or analytical tasks
- Governable through Unity Catalog's security and access control mechanisms

A2. SQL vs Python Agent Tools: Key Differences and Use Cases:

Aspect	(i) SQL Agent Tools	(ii) Python Agent Tools
Optimized for	Data querying and analytical operations	Custom logic and complex computations
Execution	`CREATE OR REPLACE FUNCTION` statements	`CREATE OR REPLACE FUNCTION` or `DatabricksFunctionClient()`
Capabilities	SQL syntax and built-in functions	External APIs, libraries, complex algorithms
Execution Mode	Serverless only	Serverless and local
Requirements	None beyond SQL	Explicit type hints, Google-style docstrings
Ideal for	Data retrieval, aggregations, filtering, analytical calculations	Business logic, external integrations, complex algorithms, data transformations

(iii) Combining Tools:
The most powerful agent architectures use both SQL and Python tools together, where SQL handles data access and analysis while Python functions manage business logic and external integrations.

A3. Agent Tools vs Traditional Functions:

Understanding the distinction between agent tools and traditional functions is crucial for effective implementation:

Feature	(i) Traditional Functions	(ii) Unity Catalog Functions as Agent Tools
Design Purpose	Direct programmatic use by developers	Dynamic discovery and use by AI agents
Documentation	Limited or minimal	Rich metadata and comprehensive documentation required

Feature	(i) Traditional Functions	(ii) Unity Catalog Functions as Agent Tools
Parameter Usage	Called explicitly with known parameters	Parameters and usage inferred from natural language queries
Primary Focus	Computational efficiency and performance	Clarity, interpretability, and agent usability
Context	Technical implementation details	Business context and usage examples included

B. Registration Methods

B2. Function Registration Methods:

Unity Catalog provides different approaches for registering SQL and Python functions as agent tools. Since UC-registered functions are governed by UC permissions, this differentiates registration when compared to session-scoped/notebook UDFs.

SQL Function Registration:

(i) Using `CREATE OR REPLACE FUNCTION` statements:

- Immediate registration and availability
- Full control over function definition and metadata
- Integration with existing SQL development workflows
- Support for complex SQL logic and business rules
- No support for custom environment or dependencies

Python Function Registration:

(i) Using the `DatabricksFunctionClient()`:

- `create_python_function()` API accepts Python callables directly
- Automatic extraction of type hints and docstring metadata
- Integration with Unity Catalog's three-level namespace
- Support for function versioning and replacement
- Supports serverless (production) and local (dev) modes, though local mode does not support SQL-based functions

(ii) Using `CREATE OR REPLACE FUNCTION` statements:

- Similar to SQL tool creation, you can also create a Python function using python logic but SQL syntax for registration (see an example here).
- Can define custom dependencies using the `ENVIRONMENT` clause (read more here).

B3. (Optional) Execution Environment Considerations:

To read more about technical considerations (serverless vs local mode) for UC Python functions, please see this documentation.

C. Tool Registration Validation with UI

Once your functions have been registered to Unity Catalog, you can validate the metadata information the LLM will consume for context and query filtering. Below are two examples of what to expect when navigating the tool locations within UC.

(Note: The original text mentioned "two examples" but did not provide the actual image/UI examples. In practice, you would navigate to your function in the Catalog Explorer to inspect its details, including comments, parameters, and return type.)

D. Integration with AI Playground for Prototyping

D1. AI Playground Integration:

AI Playground provides a no-code interface for testing and prototyping both SQL and Python Unity Catalog Functions as Agent Tools. In the AI Playground, you have automatic UC-permission-level access to tools as well as state-of-the-art LLMs like Claude and GPT models. The AI Playground should be used for prototyping queries, LLMs, and tool usage before building agent code. Below is an example of how the AI Playground looks when sending a prompt that invokes tool usage from the LLM.

(Note: The original text mentioned "an example" but did not provide the actual image. The AI Playground interface allows you to select which tools are available to the model and see the model's decision-making process, including which tools it calls and the results.)

Conclusion:

Now that you understand how UC Tools can be built, registered, visually inspected, and tested on Databricks, you are ready for the follow-along demonstration that will cover these concepts in practice.

Authoring Single AI Agents with Databricks Mosaic AI Agent Framework

The Databricks Mosaic AI Agent Framework provides a unified platform for authoring, deploying, and monitoring AI agents. This framework supports multiple popular agent development libraries including LangChain, LangGraph, DSPy, and OpenAI, while providing native integration with Databricks services like Vector Search, Model Serving, and MLflow.

The framework emphasizes production readiness through features like automatic tracing, comprehensive evaluation capabilities, and seamless deployment to Mosaic AI Model Serving. Whether you're building simple chat agents or complex multi-agent systems, Databricks provides the tools and infrastructure needed for enterprise-scale AI applications.

A. Introduction to Mosaic AI Agent Framework

The Databricks Mosaic AI Agent Framework represents a comprehensive solution for building, deploying, and managing AI agents at enterprise scale. This framework addresses the complete agent lifecycle from development to production monitoring.

An Agent's Lifecycle:

An agent's lifecycle can be summarized as follows:

(i) Prepare data and create tools:

This phase includes AI-related ETL using Notebooks, SQL queries, and the Lakeflow suite. Typically, this is where the AI engineer embeds and indexes unstructured data with Vector Search.

Once the data is prepared, the engineer moves on to creating tools in either SQL or Python syntax and registers those tools in Unity Catalog (https://docs.databricks.com/aws/en/data-governance/unity-catalog/) for comprehensive governance.

(ii) Rapid prototyping with quality checks:

In this phase, typically rapid tests are performed with AI Playground's no-code interface for rapid prototyping agents. Here you can specify a system prompt, select different models and compare them side-by-side and vibe check the results.

The AI engineer then evaluates the content with evaluation tools with MLflow 3 (https://docs.databricks.com/aws/en/mlflow3/genai/eval-monitor/), which is designed to help you identify quality issues and the root cause of those issues.

Once rapid prototyping has been completed, you can export the code from the playground and leverage `mlflow.genai.evaluate()` (read more here: https://docs.databricks.com/aws/en/mlflow3/genai/eval-monitor/concepts/eval-harness).

(iii) Evaluate and collect feedback:

Test the agent against evaluation datasets using methods such as LLM judges, stakeholder labeling, and synthetic data. Stakeholder/domain expert feedback is gathered, typically through review apps or direct tracing of interactions.

(iv) Label data and feedback:

Interactions and outputs are labeled to create high-quality benchmarks for testing future agent iterations. This creates an evaluation set that serves as ground truth for quality assessment.

Labeling sessions provide a structured way to gather feedback from domain experts on the behavior of your GenAI applications. You can read more about labeling sessions here: https://docs.databricks.com/aws/en/mlflow3/genai/human-feedback/concepts/labeling-sessions.

(v) Iterative improvement:

- Use feedback and benchmark results to identify and fix root causes of quality issues.
- Evaluate multiple versions/configurations to achieve the desired balance of accuracy, safety, cost, and latency.

(vi) Deploy to production:

- The agent moves from development to a scalable, production-ready environment (often REST API via Model Serving). Here, agents are governed for access and compliance, leveraging components such as Unity Catalog for unified governance.

(vii) Monitor quality and performance:

- Post-deployment, agents are continuously monitored using the same evaluation and tracing tools as in development. Logs, traces, user feedback, and automated judges provide ongoing quality signals; new data from production interactions is incorporated into evaluation sets for future improvements.
- AI Gateway-enhanced (https://docs.databricks.com/aws/en/ai-gateway/) inference tables are automatically enabled for deployed agents, which provides access to detailed request log metadata when using the Mosaic AI Agent Framework with popular agent authoring libraries like DSPy and LangChain.
- MLflow tracing (https://docs.databricks.com/aws/en/mlflow3/genai/tracing/) can be leveraged for end-to-end observability.
- Production monitoring for GenAI lets you automatically run MLflow scorers on traces from production GenAI apps here: https://docs.databricks.com/aws/en/mlflow3/genai/eval-monitor/production-monitoring.

A1. Framework Architecture

Assuming you have completed initial testing of your UC/external tools both in Notebooks/SQL Editor and the AI Playground, you are now ready to take a look at agent frameworks. To aid with this, Databricks offers a suite of tooling to author, deploy, and monitor high-quality agentic and RAG applications with the Mosaic AI Agent Framework. This framework is built around several key components:

- MLflow 3 Integration: Provides experiment tracking, model logging, and lifecycle management
- ResponsesAgent Interface: A production-ready interface compatible with OpenAI Responses schema
- Agent Authoring Libraries: Support for LangChain, LangGraph, DSPy, and OpenAI
- Databricks AI Bridge: Integration packages that connect agents to Databricks AI features
- Agent Governance: Tools and agents are registered and governed in UC; inference logs/traces via AI Gateway and MLflow tracing.
- Mosaic AI Model Serving: Scalable deployment infrastructure for production agents
- Evaluation & Monitoring: Built-in tools for agent quality assessment and performance tracking

A2. Requirements and Setup

To develop agents using the Databricks framework, you will need to be aware of some technical platform requirements and packages.

Core Requirements:

- `databricks-agents` 1.2.0 or above

- `mlflow` 3.1.3 or above
- Python 3.10 or above
- Serverless compute or Databricks Runtime 13.3 LTS or above

Installation Command:

```
%pip install -U -qqqq databricks-agents mlflow
```

AI Bridge Integration Packages: The Databricks AI Bridge library provides a shared layer of APIs to interact with Databricks AI features, such as Databricks AI/BI Genie and Vector Search. You can see the latest release notes and versions on PyPi.

- `databricks-openai` for OpenAI integration
- `databricks-langchain` for LangChain/LangGraph integration
- `databricks-dspy` for DSPy integration
- `databricks-ai-bridge` for pure Python agents (without dedicated integration packages)

B. ResponsesAgent: The Production Interface

Databricks recommends the MLflow ResponsesAgent interface as the primary method for creating production-grade agents. This interface provides compatibility with the OpenAI Responses schema while adding Databricks-specific enhancements.

Note: If you are familiar with ChatAgent, ResponsesAgent is meant to replace this interface for new agents.

B1. ResponsesAgent Benefits

The ResponsesAgent interface offers significant advantages over traditional agent interfaces, while also supporting wrapping existing agents in supporting frameworks.

(i) Advanced Agent Capabilities:

- Multi-agent support for complex workflows
- Streaming output with real-time response chunks
- Comprehensive tool-calling message history
- Tool-calling confirmation support
- Long-running tool execution support

(ii) Streamlined Development & Deployment:

- Framework-agnostic: Wrap any existing agent for Databricks compatibility
- Typed authoring interfaces with IDE autocomplete support
- Automatic signature inference during model logging
- If you are not use the recommended ResponsesAgent interface, you must either manually define your signature or use MLflow's Model Signature inference capabilities to automatically generate the agent's signature based on input examples. Read more here: https://docs.databricks.com/aws/en/generative-ai/agent-framework/log-agent#infer-model-signature-during-logging.
- Automatic tracing with aggregated streamed responses via `predict` and `predict_stream`
- ResponsesAgent requires implementing a `predict` method that returns ResponsesAgentResponse to handle non-streaming requests. On the other hand, for streaming agents, you can implement a `predict_stream` method. This goes beyond the scope of this lecture, but you can read more here: https://docs.databricks.com/aws/en/generative-ai/agent-framework/author-agent?language=Streaming+with+code+re-use.
- AI Gateway-enhanced inference tables for detailed logging

B2. ResponsesAgent Schema Structure

The ResponsesAgent uses a structured schema for inputs and outputs:

(i) Input Format (ResponsesAgentRequest):

```
{
"input": [
{
"role": "user",
"content": "What did the data scientist say when their Spark job finally completed?"
}
]
}
```

(ii) Output Format (ResponsesAgentResponse):

```
ResponsesAgentResponse(
output=[
{
"type": "message",
"id": str(uuid.uuid4()),
"content": [{"type": "output_text", "text": "Well, that really sparked joy!"}],
"role": "assistant",
}
]
)
```

B3. Wrapping Existing Agents

If you already have an agent built with LangChain, LangGraph, or similar frameworks, you don't need to rewrite it. Instead, create a wrapper class that inherits from `mlflow.pyfunc.ResponsesAgent`:

Basic Wrapper Pattern:

```
from uuid import uuid4
from mlflow.pyfunc import ResponsesAgent
from mlflow.types.responses import ResponsesAgentRequest, ResponsesAgentResponse

class MyWrappedAgent(ResponsesAgent):
    def init(self, agent):
      # Reference your existing agent (LangChain/LangGraph/OpenAI, etc.)
      self.agent = agent

    def prep_msgs_for_llm(self, messages: list[dict]) -> list[dict]:
        # Implement conversion from ResponsesAgentRequest messages to your agent's expe
        return messages

    def predict(self, request: ResponsesAgentRequest) -> ResponsesAgentResponse:
        # Convert incoming messages to your agent's format
        messages = self.prep_msgs_for_llm([i.model_dump() for i in request.input])

        # Call your existing agent (non-streaming)
        agent_response = self.agent.invoke(messages)

        # Ensure string output; convert if necessary
        if not isinstance(agent_response, str):
            agent_response = str(agent_response)

        # Convert to ResponsesAgent format
        output_item = self.create_text_output_item(text=agent_response, id=str(uuid4())
        return ResponsesAgentResponse(output=[output_item])
```

C. Supported Agent Authoring Frameworks

Databricks supports multiple popular frameworks for agent development, each with specific strengths and use cases.

C1. LangChain Integration

LangChain is a comprehensive framework for building LLM applications with extensive integrations and capabilities.

Key Features on Databricks:

- Use Databricks-served models as LLMs or embeddings
- Integration with Mosaic AI Vector Search for vector storage
- MLflow experiment tracking and performance monitoring
- MLflow Tracing for development and production observability
- PySpark DataFrame loader for seamless data integration
- Spark DataFrame Agent and Databricks SQL Agent for natural language querying

Example Usage:

```
from databricks_langchain import ChatDatabricks

chat_model = ChatDatabricks(
endpoint="databricks-gpt-5-1",
temperature=0.1,
max_tokens=250,
)
chat_model.invoke("How to use Databricks?")
```

Note: Keep in mind that LangChain on Databricks (https://docs.databricks.com/aws/en/large-language-models/langchain) for LLM development are experimental features and the API definitions might change over time.

C2. DSPy Framework

DSPy is a framework for programmatically defining and optimizing generative AI agents with automated prompt engineering capabilities.

Core DSPy Components:

- Modules: Components handling specific text transformations (replacing hand-written prompts)
- Signatures: Natural language descriptions of input/output behavior ("question -> answer")
- Compiler: Optimization tool that improves pipelines by adjusting modules for performance metrics
- Program: Connected modules forming a pipeline for complex tasks

DSPy Advantages:

- Automated prompt optimization
- Systematic approach to agent improvement
- Built-in performance optimization capabilities
- Programmatic rather than manual prompt engineering

C3. OpenAI Integration

Databricks provides native support for OpenAI-style agents while leveraging Databricks-hosted models.

Integration Benefits:

- Use familiar OpenAI API patterns
- Leverage Databricks Foundation Model APIs
- Seamless migration from OpenAI to Databricks models
- Support for both streaming and non-streaming responses
- Tool-calling capabilities with Databricks models

C4. LangGraph for Complex Workflows

LangGraph extends LangChain with graph-based agent orchestration for more complex, stateful workflows.

LangGraph Capabilities:

- Graph-based agent workflows
- State management across agent interactions
- Complex decision trees and conditional logic
- Multi-step reasoning and tool coordination

D. Streaming and Real-Time Responses

Streaming capabilities allow agents to provide real-time responses in chunks, improving user experience and enabling interactive applications. The idea is to wait for a complete response before sending the result to a user. With MLflow, you can not only view the agent's response, but also these chunks and thought processes, which can lead to insight as to why it chose or did not choose to use a particular tool.

The Knowledge Assistant with Agent Bricks (https://docs.databricks.com/aws/en/generative-ai/agent-bricks/knowledge-assistant) has full streaming support.

D1. Streaming Implementation

To implement streaming with ResponsesAgent, follow this pattern:

- Emit Delta Events: Send multiple `output_text.delta` events with the same `item_id`
- Finish with Done Event: Send a final `response.output_item.done` event containing complete output

Streaming Benefits:

- Real-time user feedback
- Improved perceived performance
- Better user engagement for long-running operations
- Automatic MLflow tracing integration
- Aggregated responses in AI Gateway inference tables

D2. Error Handling in Streaming

Mosaic AI propagates streaming errors through the last token under `databricks_output.error`:

```
{
"delta": "...",
"databricks_output": {
"trace": {...},
"error": {
"error_code": "BAD_REQUEST",
"message": "TimeoutException: Tool XYZ failed to execute."
}
}
}
```

Note: Client applications must handle and surface these errors appropriately.

E. (Optional) Advanced Features and Customization

The Databricks Agent Framework provides several advanced features for sophisticated agent implementations. The topics given below go beyond the scope of this course.

E1. Custom Inputs and Outputs

Some scenarios require additional agent inputs (like `client_type`, `session_id`) or outputs (like retrieval source links) that shouldn't be included in chat history.

Custom Fields Support:

- `custom_inputs` : Additional input parameters beyond standard messages. You can read more about custom inputs and output here: https://docs.databricks.com/aws/en/generative-ai/agent-framework/author-agent#advanced-features.
- `custom_outputs` : Additional output data not part of conversation flow
- Access via `request.custom_inputs` in agent code
- JSON configuration in AI Playground and review apps

Important Limitation: The Agent Evaluation review app does not support rendering traces for agents with additional input fields.

You can read more about advanced features in this regard here: https://docs.databricks.com/aws/en/generative-ai/agent-framework/author-agent#advanced-features. This goes beyond the scope of this course.

E2. Retriever Integration and Schema

AI agents commonly use retrievers for unstructured data from vector search indices. Databricks provides specialized support for retriever tracing and evaluation. You can read more about retriever integration here: https://docs.databricks.com/aws/en/generative-ai/agent-framework/unstructured-retrieval-tools#set-retriever-schema-to-ensure-mlflow-compatibility.

Retriever Benefits:

- Automatic source document links in AI Playground UI
- Automatic retrieval groundedness and relevance evaluation
- Integration with Databricks AI Bridge retriever tools

Custom Retriever Schema:

```
import mlflow

mlflow.models.set_retriever_schema(
name="mlflow_docs_vector_search",
primary_key="document_id", # Document ID field
text_column="chunk_text", # Content field
doc_uri="doc_uri", # Document URI field
other_columns=["title"], # Additional metadata
)
```

This goes beyond the scope of this course. You can read more about retriever tools here: https://docs.databricks.com/aws/en/generative-ai/agent-framework/unstructured-retrieval-tools.

E3. Multi-Agent Systems

Databricks supports complex multi-agent systems where multiple specialized agents collaborate to solve problems.

This goes beyond the scope of this course. To learn more about multi-agent systems managed by Databricks, you can read documentation on Genie (https://docs.databricks.com/aws/en/generative-ai/agent-framework/multi-agent-genie) and Agent Bricks (https://docs.databricks.com/aws/en/generative-ai/agent-bricks/multi-agent-supervisor).

You can read more about Genie with multi-agents systems here: https://docs.databricks.com/aws/en/generative-ai/agent-framework/multi-agent-genie.

E4. Stateful Agents

Stateful agents can maintain memory across conversation threads and provide conversation checkpointing capabilities.

This goes beyond the scope of this course. To learn more about stateful agents, please see this documentation: https://docs.databricks.com/aws/en/generative-ai/agent-framework/stateful-agents.

Conclusion

The Databricks Mosaic AI Agent Framework provides a comprehensive platform for building production-ready AI agents. Key takeaways include understanding popular framework strengths (e.g., LangChain), develop and understanding for best practices when building an agent's lifecycle, and production considerations.

Building Agents on Databricks with MLflow

MLflow enhances GenAI applications with end-to-end tracking, observability and evaluations, all within one integrated platform. Therefore, it supports the whole lifecycle of AI agents: from development, evaluation, deployment and all the way up to production monitoring. This lecture explores how MLflow's comprehensive capabilities make it an essential component in the agent development lifecycle, from initial prototyping through production deployment and ongoing observability.

A. The Agent Development Challenge:

Building production-ready AI agents presents unique challenges that traditional machine learning workflows don't fully address. Understanding these challenges helps us appreciate why MLflow on Databricks has made Databricks a comprehensive platform for the whole agent lifecycle.

A1. Complexity of Agent Systems:

AI agents are fundamentally different from traditional ML models in several key ways:

- Multi-step reasoning: Agents perform complex, multi-turn interactions that involve planning, tool usage, and decision-making across multiple steps
- Dynamic behavior: Unlike static models, agents can exhibit different behaviors based on context, available tools, and conversation history
- Tool integration: Agents must seamlessly integrate with external systems, APIs, and data sources to accomplish tasks
- Conversational context: Maintaining state and context across multi-turn conversations adds complexity to deployment and monitoring

These characteristics create unique requirements for development, testing, and production monitoring that traditional ML platforms weren't originally designed to handle.

A2. Observability Requirements:

Agent observability goes far beyond traditional model monitoring:

- Execution tracing: Understanding the step-by-step reasoning process, including which tools were called and why
- Performance analysis: Tracking latency, token usage, and costs across complex multi-step workflows
- Quality assessment: Evaluating not just final outputs but intermediate reasoning steps and tool usage patterns
- Error diagnosis: Identifying where failures occur in multi-step processes and understanding their root causes

Without proper observability, debugging agent behavior becomes nearly impossible, especially in production environments.

A3. Governance and Reproducibility Challenges:

Enterprise deployment of agents requires robust governance capabilities:

- Version management: Tracking changes to agent logic, prompts, tools, and configurations
- Reproducibility: Ensuring consistent behavior across development, staging, and production environments
- Access control: Managing who can deploy, modify, or access different agent versions
- Audit trails: Maintaining complete records of agent behavior for compliance and debugging
- AI Guardrails: Allows users to configure and enforce data compliance at the model-serving-endpoint level (read more here)

These requirements necessitate a platform that can handle the full agent lifecycle with enterprise-grade governance features.

B. MLflow on Databricks:

Here we will break down MLflow a little more to help understand how it addresses the issues raised in the previous section.

B1. Why is Tracing Needed?:

To understand why we need tracing (and what it actually is), we need to understand traditional machine learning inference.

Traditional ML inference (request/response):

A typical inference flow for machine learning is given by the following high-level steps:

(i) The client sends an input request to a request handler on the serving endpoint.
(ii) The handler forwards the request to the model for inference.
(iii) The model's output is returned through the handler to the client.

In this basic scenario, the transparent components are often just the input and output.

For robust operations in the era of agents, we may also want insight into processes like server-side transparency, latency/cost metrics, and API logging (even though these may not be needed for ML workloads).

These capabilities are standard with Databricks Model Serving and AI Gateway for production telemetry and governance. Databricks hosts managed MLflow, where setting your tracking URI to

databricks logs traces to your workspace with built-in security, reliability, search, and UI.

Additionally, deploying with Mosaic AI Agent Framework automatically integrates real-time tracing and can enable the Review App and monitoring for production traffic.

Agents need more insight:

- Agents perform multiple intermediate steps (for example: retrieval, tool use, LLM calls), and you need to see each step, its inputs/outputs, and per-step latency/token usage to debug and improve quality.
- MLflow Tracing captures these as traces and spans automatically for supported libraries (OpenAI SDK, LangChain/LangGraph, DSPy, etc.) and provides a UI and APIs to analyze them across development and production.
- A single operation in a tracing system is called a span. It records when the operation started and ended, along with the metadata, inputs, and outputs per unit of work.

MLflow spans follow the OpenTelemetry standard, which requires any extra information (like token counts) to be stored in key-value attributes on the span, not as custom fields.

B2. Tracing for Agents:

Now that we understand the struggle with developing agentic systems and why agents need more insights per unit of work, we're ready to define what a trace is.

A Trace in the context of GenAI applications is a collection of spans arranged in a DAG-like structure, where each span represents a single operation. These single operations can be something like a function call or a database query.

As an example, suppose you are working on developing an agent that is exposed to 3 UC tools. Suppose also that you are noticing slow execution times, but you're not sure what the issue is. The MLflow interface in Databricks can help you troubleshoot this scenario.

For example, you can:

- View the specific Foundation Model API that was used for each reasoning step.
- View the system prompts (if any) used for the agent.
- Identify if tools have been called, the order they were called in, and their inputs/outputs.
- The agent's reasoning at each step.
- The latency to identify which tool took the longest to run. This can be useful to, for example, help build optimized SQL queries.
- Token usage is exposed per span and trace (aggregated).

B3. Hierarchical Span Structure:

MLflow organizes trace data using a hierarchical span structure that mirrors agent execution, starting with a single root span that represents the overall request or workflow, with nested child spans for each sub-step.

- Parent spans: High-level operations like "process user request"
- Child spans: Detailed steps like "call retrieval tool" or "generate response"
- Span relationships: Clear parent-child relationships that show execution flow, which should mimic your application's execution plan.
- Span types: Categorization of spans (TOOL, CHAT_MODEL, RETRIEVER) for better organization

B4. Custom Tracing and Tagging:

MLflow provides flexible APIs for custom tracing needs as well (which we will see in our demonstration).

- Custom Tracing: The `@mlflow.trace` decorator lets you turn any function into a traced span with almost no added work. When applied, it provides a lightweight but powerful way to instrument your code:
  - MLflow automatically infers parent-child relationships between traced functions, ensuring full compatibility with auto-tracing integrations.
  - Any exceptions raised within the function are captured and logged as span events.
  - The function's name, inputs, outputs, and execution duration are recorded without additional configuration.
  - It works seamlessly alongside auto-tracing features like `mlflow.openai.autolog`.
  - Accepts the following arguments:
    - `name` : parameter to override the span name from the default (the name of decorated function).
    - `span_type` : parameter to set the type of span. Set either one of built-in Span Types or a string.
    - `attributes` : parameter to add custom attributes to the span.

Function Type Considerations:

To view a complete list of function types and supported dependencies when using the `@mlflow.trace` decorator, please see this documentation.

- Tagging: Tags are flexible key-value pairs that you can update throughout the trace's lifecycle, while metadata is immutable and set once at trace creation.

> Note: This course will only be concerned with a subset of the Span object schema. You can read more about the Span object schema here.

C. Models in Unity Catalog for Agent Governance:

MLflow's integration with Unity Catalog enables enterprise-grade governance for agent deployments by registering agents as Unity Catalog models, so they can be managed with the same rigor as other business-critical assets.

Once you have your data source and tools registered to Unity Catalog (or external tools), you can register your agent code to UC by first packaging the agent with MLflow and using the model's URI.

Centralized governance via the Model Registry in UC:

Registering agents as UC models provides a centralized, cross-workspace catalog of agent assets:

- Version management: MLflow logs a point-in-time snapshot of agent code, configuration, and declared resources; each UC model version is an immutable snapshot you can reference and deploy.
- Lineage tracking: When you log inputs (for example with `mlflow.log_input`), UC shows lineage between models and upstream datasets; lineage is also captured for feature store training flows.
- Access control: Fine-grained UC privileges govern who can create, read, or modify models and who can execute functions, query tables, use connections, and access other resources your agent depends on.
- Cross-workspace sharing: Models in UC are discoverable and governable across workspaces attached to the same metastore.
- Governed tags: Tags can be applied to registered models and model versions; governed tags (public preview) enforce standardized keys/values and assignment permissions for consistent classification and control. See the docs here.

Reproducible deployments:

Using UC + MLflow ensures that agent deployments are reproducible and observable:

- Immutable versions: Registered model versions are immutable snapshots; update metadata if needed, but changing code/dependencies requires a new version.
- Dependency capture: MLflow captures environment dependencies (for example via pip/conda) to enable consistent loading and serving.

- Managed serving: Deploy UC-registered agents to Model Serving endpoints with built-in scaling, tracing, and review apps for feedback and monitoring.

Conclusion:

MLflow has evolved into a comprehensive platform that addresses all aspects of AI agent development, from initial experimentation through production deployment and monitoring. Its combination of experiment tracking, tracing, model registry, and evaluation capabilities makes it uniquely suited to handle the complexity of modern AI agents.

The platform's integration with Unity Catalog and the broader Databricks ecosystem provides the governance, security, and scalability needed for enterprise agent deployments. As AI agents become increasingly important in business applications, MLflow's role as the foundational platform for agent development will continue to grow.

Single Agents with Agent Bricks

Agent Bricks provides a high-level abstraction designed to help technical users quickly build and optimize production-ready, domain-specific AI agents, focusing on automatic evaluation and optimization, including Agent Learning on Human Feedback (ALHF), to maximize quality while balancing cost considerations.

Unlike traditional agent development approaches that require extensive manual configuration and optimization, Agent Bricks streamlines the implementation process so users can focus on the problem, data, and metrics instead of low-level technical details. The platform supports four distinct agent types, each optimized for specific use cases and deployment patterns.

A. Introduction to Agent Bricks:

Agent Bricks provides a simple, powerful approach to building domain-specific agent systems. The platform abstracts away much of the complexity traditionally associated with agent development while maintaining the flexibility needed for enterprise applications.

The core philosophy of Agent Bricks is to enable users to focus on defining their business problems and providing relevant data, while the platform handles the technical complexities of agent optimization, evaluation, and deployment. This approach significantly reduces the time-to-value for AI agent implementations in enterprise environments.

A1. Supported Agent Types and Use Cases:

Agent Bricks supports four primary agent types, each designed for specific enterprise use cases and operational patterns. Understanding these distinctions is crucial for selecting the appropriate agent type for your specific requirements.

The Four Agent Types:

- Information Extraction (IE): Automated extraction of structured data from unstructured sources such as documents, PDFs, emails, and images
- Custom LLM (CLLM): Domain-specific language models fine-tuned and optimized for particular tasks and datasets
- Knowledge Assistant (KA): Interactive agents that provide question-answering capabilities over knowledge bases using retrieval-augmented generation. That is, KA is a single agent where tool-calling capabilities are restricted to RAG applications.
- Multi-Agent Supervisor (MAS): Coordination systems that manage and orchestrate multiple specialized agents to complete complex, multi-step tasks. For example, we can equip an MAS with a set of tools and no additional agents and have it act as a single agent with a toolkit.

Genie Agent:

Users can create and use a Genie Agent to use natural language to query databases or other structured data, making data analysis more accessible. Genie agents can be orchestrated with an MAS or can be standalone single agents.

A2. Operational Categories:

Agents are organized into two operational models based on their intended use patterns:

- Automated Bricks (Information Extraction and Custom LLM): Optimized for high-scale, batch processing scenarios with minimal human intervention. These agents prioritize cost-performance optimization and throughput.
- Interactive Bricks (Knowledge Assistant, Multi-Agent Supervisor, and Genie): Designed for human-in-the-loop experiences and real-time interaction scenarios. These agents focus on conversational interfaces and dynamic response generation.

B. Agent Bricks Development Lifecycle:

The Agent Bricks development process follows a structured, iterative approach designed to optimize agent performance through continuous improvement and feedback incorporation. This lifecycle ensures that agents not only meet initial requirements but continue to improve through real-world usage and feedback.

B1. Core Three-Step Development Cycle:

The Agent Bricks development lifecycle consists of three primary phases that form the foundation of agent development, followed by continuous iteration for ongoing improvement.

Step 1: Specify Your Problem:

At a high level, the user begins by building an agent that is specific to their use case. For example, with the MAS, you might want a managed agent that allows for only tool calling with a Genie Agent. After setting up proper permissions, MLflow is leveraged for tracking metrics and logging.

In this initial phase, you define the scope and requirements of your AI agent:

- Clearly define the required task and expected outcomes with your team
- Select the appropriate agent type from the four available options: Information Extraction, Custom LLM, Knowledge Assistant, or Multi-Agent Supervisor
- Depending on your use case, you next need to provide your UC-managed datasets (Delta tables, UC Volumes), equip tools, and attach other agents
- Establish success criteria and quality metrics for evaluation

Step 2: Optimize on Your Enterprise Data:

Agent Bricks automatically builds and optimizes the best agent system based on quality versus cost tradeoffs:

- The system automatically creates evaluation benchmarks related to your specific task (such as Accuracy, Product Relevance, Customer Churn prediction, etc.)
- Optimization involves intelligent selection and composition of multiple techniques:
    - Advanced prompt optimization using proven methodologies
    - Selective fine-tuning based on task requirements and data availability
    - Optimal tool selection and configuration
    - Implementation of Custom LLM Judges for quality assessment
    - Reward Model filtering for response quality improvement
    - Reinforcement Learning from Human Feedback (RLHF) when beneficial

Step 3: Continuous Improvement:

The final step establishes a feedback loop for ongoing optimization:

- Deploy the optimized agent to production environment
- Continuously measure agent quality through automated and human evaluation
- Systematically identify issues and improvement opportunities through monitoring
- Apply natural language feedback to improve system performance
- Leverage Agent Learning on Human Feedback (ALHF) for iterative enhancement

B2. Evaluation and Monitoring Framework:

Agent Bricks provides comprehensive evaluation and monitoring capabilities built directly into the platform, ensuring continuous visibility into agent performance and quality metrics.

Automatic MLflow Integration:

Every agent deployed through Agent Bricks automatically includes comprehensive tracking capabilities:

- Request Tracking: Complete logging of all incoming requests with timestamps and user context
- Response Monitoring: Detailed capture of outgoing responses including confidence scores and reasoning paths
- Inter-Agent Communication: Full tracing of communication between agents in multi-agent systems
- Performance Metrics: Automatic collection of latency, throughput, and resource utilization data

Quality Assessment Mechanisms:

The platform implements multiple layers of quality evaluation:

- Automatic Benchmark Creation: Task-specific metrics tailored to your use case requirements
- LLM Judge Evaluation: Automated quality scoring using specialized language models trained for evaluation tasks
- Human Feedback Integration: Structured collection and integration of expert feedback through review applications
- Production Performance Monitoring: Real-time tracking of agent performance in live environments
- Comparative Analysis: Benchmarking against baseline models and previous agent versions

C. Integration with Other Services:

Agent Bricks is tightly integrated with Mosaic AI Model Serving, Vector Search, Unity Catalog, Genie, MLflow 3, and Databricks Apps, creating a unified platform for building, governing, evaluating, and deploying AI agents end-to-end.

This integration means users can rapidly prototype, iterate, and deploy agent systems using their own enterprise data, while maintaining best-in-class governance, security, and scalability.

How Agent Bricks Works Alongside the Databricks Stack:

- Mosaic AI Model Serving: Agents can be deployed as scalable REST APIs with automatic load balancing and monitoring. This serving platform also provides secure authentication and natively integrates with MLflow 3, enabling real-time tracing and quality evaluation.
- Vector Search: Agents tap into Databricks Vector Search to efficiently retrieve relevant unstructured information, supporting both retrieval-augmented generation (RAG) and advanced use cases like semantic search across documents and tables.
- Unity Catalog: Ensures unified governance across all data, models, agents, and tools. Agent logic, data lineage, and tool access are controlled to meet regulatory and compliance needs, while supporting integration with enterprise security requirements.
- Genie & Genie Spaces: Enable agents to interact directly with structured data (e.g., text-to-SQL queries) and orchestrate multiple tools, expanding agent capabilities (e.g. multi-agent or tool-calling architectures).
- MLflow 3: Provides robust experiment tracking, versioning, tracing, and evaluation for agents. Real-time traces and automated quality measurement (with research-backed LLM judges) allow rapid debugging and improvement cycles.
- Databricks Apps: Offer user interfaces such as built-in chat apps, feedback collection portals, and production dashboards. These UIs allow stakeholders to interact with agents, submit feedback, and ensure agents meet business needs.