Building Agentic AI Applications with Databricks Agent Framework & MLflow 3

Technical Guide for the Multi-Country Pension Advisor

Repository: `pravinva/superannuation-agent-multi-country`
Architecture: ReAct Framework with Governance Layers
Deployment: Databricks Apps | Local Development

Table of Contents

Architecture Overview

The 8-Phase ReAct Pipeline

```
User Query
  ↓
┌─────────────────────────────────────────────┐
│ Phase 1: Data Retrieval (Observe)           │
│ - Fetch member profile from Unity Catalog   │
└─────────────────────────────────────────────┘
  ↓
┌─────────────────────────────────────────────┐
│ Phase 2: Privacy Anonymization (Action)     │
│ - Replace PII before LLM processing         │
└─────────────────────────────────────────────┘
  ↓
┌─────────────────────────────────────────────┐
│ Phase 3: Tool Planning (Reason)             │
│ - LLM decides which UC functions to call    │
```

```
└──────────────────────────────────────────────────────┘
  ↓
┌──────────────────────────────────────────────────────┐
│ Phase 4: Tool Execution (Act)                        │
│ - Execute UC functions with parameters               │
└──────────────────────────────────────────────────────┘
  ↓
┌──────────────────────────────────────────────────────┐
│ Phase 5: Response Synthesis (Reason)                 │
│ - LLM creates natural language response              │
└──────────────────────────────────────────────────────┘
  ↓
┌──────────────────────────────────────────────────────┐
│ Phase 6: Validation (Observe)                        │
│ - LLM Judge validates accuracy & citations           │
└──────────────────────────────────────────────────────┘
  ↓
┌──────────────────────────────────────────────────────┐
│ Phase 7: Name Restoration (Action)                   │
│ - Re-personalize with member's real name             │
└──────────────────────────────────────────────────────┘
  ↓
┌──────────────────────────────────────────────────────┐
│ Phase 8: Audit Logging (Action)                      │
│ - Log to Unity Catalog & MLflow                      │
└──────────────────────────────────────────────────────┘
  ↓
Response to User
```

Key Design Principles

1. **Separation of Concerns**
   - LLM handles reasoning and orchestration
   - Unity Catalog functions handle calculations
   - This prevents hallucinations in numerical results
2. **ReAct Loop Implementation**
   - **Reason:** Plan which tools to use
   - **Act:** Execute tools and observe results
   - **Iterate:** Continue until task is complete
3. **Governance by Design**
   - Privacy protection in Phase 2
   - Quality validation in Phase 6

- Complete audit trail in Phase 8

Core Components

Technology Stack

```
┌─────────────────────────────────────────┐
│ Databricks Agent Framework              │
│ - databricks-agents library             │
│ - ReAct loop orchestration              │
│ - Tool calling & multi-agent support    │
└─────────────────────────────────────────┘
     ↓
┌─────────────────────────────────────────┐
│ Foundation Models API                   │
│ - Claude Opus 4.1 (Main Agent)          │
│ - Claude Sonnet 4 (LLM Judge)           │
│ - Databricks Model Serving Endpoints    │
└─────────────────────────────────────────┘
     ↓
┌─────────────────────────────────────────┐
│ Unity Catalog                           │
│ - Functions (Tools) - 18 SQL functions  │
│ - Tables (Data) - Member profiles, audit│
│ - Governance - Lineage, access control  │
└─────────────────────────────────────────┘
     ↓
┌─────────────────────────────────────────┐
│ MLflow 3                                │
│ - Experiment tracking                   │
│ - Trace logging (LangChain integration) │
│ - Model registry                        │
└─────────────────────────────────────────┘
     ↓
┌─────────────────────────────────────────┐
│ Deployment Layer                        │
│ - Databricks Apps (Production)          │
│ - Streamlit UI                          │
│ - Local development (laptop)            │
└─────────────────────────────────────────┘
```

Setup & Prerequisites

1. Environment Setup

Install Dependencies:

```bash
# Core dependencies
pip install databricks-agents mlflow>=3.0.0 langchain-databricks

# UI dependencies (if building frontend)
pip install streamlit pandas

# Databricks SDK
pip install databricks-sdk
```

Project Structure:

```
superannuation-agent/
├── agent/
│   ├── __init__.py
│   ├── pension_agent.py        # Main agent implementation
│   ├── tools.py                # UC function wrappers
│   ├── privacy.py              # Anonymization logic
│   └── validation.py           # LLM Judge implementation
├── uc_functions/
│   ├── tax_calculators.sql     # Tax calculation functions
│   ├── benefit_checkers.sql    # Benefit eligibility
│   └── projections.sql         # Balance projections
├── config/
│   ├── agent_config.yaml       # Agent configuration
│   └── endpoints.yaml          # Model endpoint configs
├── ui/
│   └── app.py                  # Streamlit interface
├── tests/
│   └── test_agent.py
├── requirements.txt
├── databricks.yml              # Databricks App config
└── README.md
```

2. Unity Catalog Setup

Create Schema:

```sql
```

```
CREATE SCHEMA IF NOT EXISTS main.pension_advisory;

-- Member profiles table
CREATE TABLE main.pension_advisory.members (
    member_id STRING,
    name STRING,
    age INT,
    balance DECIMAL(18,2),
    employment_status STRING,
    country STRING,
    years_of_service INT
);

-- Audit log table
CREATE TABLE main.pension_advisory.audit_log (
    interaction_id STRING,
    timestamp TIMESTAMP,
    member_id STRING,
    query STRING,
    response STRING,
    cost_usd DECIMAL(10,6),
    validation_verdict STRING,
    tools_called ARRAY<STRING>
);
```

3. Configure Model Serving Endpoints

Via Databricks UI:

1. Navigate to **Serving → Foundation Model API**
2. Enable Claude Opus 4.1 and Claude Sonnet 4
3. Note endpoint URLs (format: `/serving-endpoints/databricks-claude-opus-4`)

Via API:

```python
from databricks.sdk import WorkspaceClient

w = WorkspaceClient()

# The endpoints are pre-configured for Foundation Models
main_endpoint = "databricks-claude-opus-4"
judge_endpoint = "databricks-claude-sonnet-4"
```

Building the Agent

Step 1: Define Unity Catalog Functions as Tools

Create UC Functions:

```sql
-- Example: Australia tax calculator
CREATE OR REPLACE FUNCTION main.pension_advisory.au_calculate_tax(
  withdrawal_amount DECIMAL(18,2),
  age INT,
  balance DECIMAL(18,2)
)
RETURNS STRUCT<
  tax_amount DECIMAL(18,2),
  net_amount DECIMAL(18,2),
  tax_rate DECIMAL(5,2),
  explanation STRING
>
LANGUAGE SQL
RETURN
  CASE
    WHEN age >= 60 THEN
      named_struct(
        'tax_amount', 0.0,
        'net_amount', withdrawal_amount,
        'tax_rate', 0.0,
        'explanation', 'Tax-free as age 60+'
      )
    WHEN age >= 55 AND age < 60 THEN
      named_struct(
        'tax_amount', withdrawal_amount * 0.15,
        'net_amount', withdrawal_amount * 0.85,
        'tax_rate', 15.0,
        'explanation', 'Taxed at 15% preservation age'
      )
    ELSE
      named_struct(
        'tax_amount', withdrawal_amount * 0.20,
        'net_amount', withdrawal_amount * 0.80,
        'tax_rate', 20.0,
        'explanation', 'Early access tax applies'
      )
  END;
```

Wrap Functions in Python:

python

```python
# agent/tools.py
from databricks.sdk import WorkspaceClient
from langchain.tools import tool
from typing import Dict, Any

w = WorkspaceClient()

@tool
def calculate_au_tax(withdrawal_amount: float, age: int, balance: float) -> Dict[str, Any]:
    """
    Calculate Australian superannuation withdrawal tax.

    Args:
        withdrawal_amount: Amount to withdraw in AUD
        age: Member's age
        balance: Current super balance in AUD

    Returns:
        Dictionary with tax_amount, net_amount, tax_rate, explanation
    """
    result = w.statement_execution.execute_statement(
        warehouse_id="your_warehouse_id",
        catalog="main",
        schema="pension_advisory",
        statement=f"""
            SELECT
                main.pension_advisory.au_calculate_tax(
                    {withdrawal_amount},
                    {age},
                    {balance}
                ) as result
        """
    ).result.data_array[0][0]

    return {
        "tax_amount": result.tax_amount,
        "net_amount": result.net_amount,
        "tax_rate": result.tax_rate,
        "explanation": result.explanation
    }

# Create tool list
pension_tools = [
    calculate_au_tax,
    calculate_us_401k_tax,
    calculate_uk_pension_tax,
```

```
    # ... 15 more functions
]
```

Step 2: Implement Privacy Anonymization

python

```python
# agent/privacy.py
import re
from typing import Tuple

def anonymize_member_data(member_data: dict, query: str) -> Tuple[dict, str, str]:
    """
    Replace member name with placeholder before LLM processing.

    Args:
        member_data: Dictionary with member profile
        query: User's query string

    Returns:
        Tuple of (anonymized_data, anonymized_query, original_name)
    """
    original_name = member_data.get("name", "Member")
    placeholder = "MEMBER_NAME_PLACEHOLDER"

    # Anonymize data
    anonymized_data = member_data.copy()
    anonymized_data["name"] = placeholder

    # Anonymize query if name appears
    anonymized_query = re.sub(
        rf'\b{re.escape(original_name)}\b',
        placeholder,
        query,
        flags=re.IGNORECASE
    )

    return anonymized_data, anonymized_query, original_name

def restore_member_name(response: str, original_name: str) -> str:
    """
    Restore member's real name in the response.
    """
    return response.replace("MEMBER_NAME_PLACEHOLDER", original_name)
```

Step 3: Build the Agent with ReAct Loop

python

```python
# agent/pension_agent.py
from langchain_databricks import ChatDatabricks
from langchain.agents import AgentExecutor, create_react_agent
from langchain.prompts import PromptTemplate
from typing import Dict, Any
import mlflow
from agent.tools import pension_tools
from agent.privacy import anonymize_member_data, restore_member_name

class PensionAgent:
    def __init__(
        self,
        main_endpoint: str = "databricks-claude-opus-4",
        judge_endpoint: str = "databricks-claude-sonnet-4",
        catalog: str = "main",
        schema: str = "pension_advisory"
    ):
        self.catalog = catalog
        self.schema = schema

        # Main LLM for reasoning
        self.llm = ChatDatabricks(
            endpoint=main_endpoint,
            temperature=0.1,
            max_tokens=2000
        )

        # Judge LLM for validation
        self.judge_llm = ChatDatabricks(
            endpoint=judge_endpoint,
            temperature=0
        )

        # Create agent with tools
        self.agent = self._create_agent()

    def _create_agent(self) -> AgentExecutor:
        """Create ReAct agent with Unity Catalog tools."""

        # ReAct prompt template
        prompt = PromptTemplate.from_template("""
You are a retirement planning advisor. Answer the following question using the available tools.

Tools available:
{tools}
```

```
Tool Names: {tool_names}

Member Context:
{member_context}

Question: {input}

Important:
- Always cite specific regulations (e.g., ATO rules, IRS code)
- Include appropriate disclaimers
- Use tools for ALL calculations - do not calculate in your head
- Be precise with numbers from tool outputs

Thought: {agent_scratchpad}
""")

        # Create ReAct agent
        agent = create_react_agent(
            llm=self.llm,
            tools=pension_tools,
            prompt=prompt
        )

        return AgentExecutor(
            agent=agent,
            tools=pension_tools,
            verbose=True,
            max_iterations=5,
            handle_parsing_errors=True
        )

    def process_query(
        self,
        member_id: str,
        query: str,
        validate: bool = True
    ) -> Dict[str, Any]:
        """
        Process a member query through the 8-phase pipeline.

        Args:
            member_id: Member's unique identifier
            query: Member's question
            validate: Whether to run LLM Judge validation

        Returns:
            Dictionary with response, cost, validation results
```

```python
"""

# Start MLflow run for tracking
with mlflow.start_run(run_name=f"query_{member_id}"):

    # PHASE 1: Data Retrieval
    member_data = self._retrieve_member_data(member_id)
    mlflow.log_param("member_id", member_id)
    mlflow.log_param("member_age", member_data["age"])
    mlflow.log_param("country", member_data["country"])

    # PHASE 2: Privacy Anonymization
    anon_data, anon_query, original_name = anonymize_member_data(
        member_data, query
    )
    mlflow.log_param("pii_anonymized", True)

    # PHASES 3-5: ReAct Loop (Reason → Act → Synthesize)
    # The agent handles tool planning, execution, and synthesis
    mlflow.log_param("query", query)

    response_data = self.agent.invoke({
        "input": anon_query,
        "member_context": self._format_member_context(anon_data)
    })

    response = response_data["output"]
    mlflow.log_text(response, "raw_response.txt")

    # Log tool calls
    intermediate_steps = response_data.get("intermediate_steps", [])
    tools_called = [step[0].tool for step in intermediate_steps]
    mlflow.log_param("tools_called", ",".join(tools_called))

    # PHASE 6: Validation
    validation_result = None
    if validate:
        validation_result = self._validate_response(
            query=query,
            response=response,
            tool_outputs=intermediate_steps
        )
        mlflow.log_dict(validation_result, "validation_result.json")

    # PHASE 7: Name Restoration
    final_response = restore_member_name(response, original_name)
    mlflow.log_text(final_response, "final_response.txt")
```

```python
        # Calculate costs
        cost = self._calculate_cost(response_data, validation_result)
        mlflow.log_metric("total_cost_usd", cost["total"])
        mlflow.log_metric("main_llm_cost", cost["main_llm"])
        mlflow.log_metric("judge_llm_cost", cost.get("judge_llm", 0))

        # PHASE 8: Audit Logging
        self._log_to_audit_table(
            member_id=member_id,
            query=query,
            response=final_response,
            cost=cost,
            validation=validation_result,
            tools_called=tools_called
        )

        return {
            "response": final_response,
            "cost": cost,
            "validation": validation_result,
            "tools_called": tools_called,
            "mlflow_run_id": mlflow.active_run().info.run_id
        }

    def _retrieve_member_data(self, member_id: str) -> Dict[str, Any]:
        """Fetch member profile from Unity Catalog."""
        from databricks.sdk import WorkspaceClient

        w = WorkspaceClient()
        result = w.statement_execution.execute_statement(
            warehouse_id="your_warehouse_id",
            catalog=self.catalog,
            schema=self.schema,
            statement=f"""
                SELECT * FROM {self.catalog}.{self.schema}.members
                WHERE member_id = '{member_id}'
            """
        ).result.data_array[0]

        return {
            "member_id": result[0],
            "name": result[1],
            "age": result[2],
            "balance": float(result[3]),
            "employment_status": result[4],
            "country": result[5],
```

```
            "years_of_service": result[6]
        }

    def _format_member_context(self, member_data: Dict) -> str:
        """Format member data for LLM context."""
        return f"""
Name: {member_data['name']}
Age: {member_data['age']}
Current Balance: ${member_data['balance']:,.2f}
Employment: {member_data['employment_status']}
Country: {member_data['country']}
Years of Service: {member_data['years_of_service']}
"""

    def _validate_response(
        self,
        query: str,
        response: str,
        tool_outputs: list
    ) -> Dict[str, Any]:
        """
        LLM Judge validation.

        Checks:
        1. Response accuracy vs. tool outputs
        2. Presence of citations
        3. Appropriate disclaimers
        4. No hallucinations
        """
        validation_prompt = f"""
You are a quality validator for retirement advice responses.

Original Query: {query}

Tool Outputs: {tool_outputs}

Response to Validate: {response}

Evaluate:
1. Accuracy: Do numbers in response match tool outputs?
2. Citations: Are regulations cited (e.g., ATO, IRS)?
3. Disclaimers: Is appropriate disclaimer included?
4. Hallucinations: Any made-up information?

Respond in JSON:
{{
    "is_valid": true/false,
```

```
    "confidence": 0.0-1.0,
    "violations": [],
    "reasoning": "explanation"
}}
"""

        validation_response = self.judge_llm.invoke(validation_prompt)

        import json
        return json.loads(validation_response.content)

    def _calculate_cost(
        self,
        response_data: Dict,
        validation_result: Dict = None
    ) -> Dict[str, float]:
        """
        Calculate costs based on Databricks Foundation Model API pricing.

        Claude Opus 4.1: $15 per 1M input tokens, $75 per 1M output
        Claude Sonnet 4: $3 per 1M input tokens, $15 per 1M output
        """
        # Simplified cost calculation
        # In production, extract actual token counts from response metadata

        main_llm_cost = 0.003  # Approximate per query
        judge_llm_cost = 0.0005 if validation_result else 0

        return {
            "main_llm": main_llm_cost,
            "judge_llm": judge_llm_cost,
            "total": main_llm_cost + judge_llm_cost
        }

    def _log_to_audit_table(
        self,
        member_id: str,
        query: str,
        response: str,
        cost: Dict,
        validation: Dict,
        tools_called: list
    ):
        """Log interaction to Unity Catalog audit table."""
        from databricks.sdk import WorkspaceClient
        import uuid
        from datetime import datetime
```

```
    w = WorkspaceClient()

    interaction_id = str(uuid.uuid4())
    timestamp = datetime.utcnow().isoformat()

    w.statement_execution.execute_statement(
        warehouse_id="your_warehouse_id",
        catalog=self.catalog,
        schema=self.schema,
        statement=f"""
            INSERT INTO {self.catalog}.{self.schema}.audit_log
            VALUES (
                '{interaction_id}',
                '{timestamp}',
                '{member_id}',
                '{query.replace("'", "''")}',
                '{response.replace("'", "''")}',
                {cost['total']},
                '{validation.get("is_valid", "unknown")}',
                array({','.join([f"'{t}'" for t in tools_called])})
            )
        """
    )
```

MLflow 3 Integration

Experiment Tracking

```python
# Set experiment
mlflow.set_experiment("/Users/your_username/pension_advisor_experiments")

# The process_query() method already logs:
# - Parameters (member_id, age, country, query)
# - Metrics (cost, validation confidence)
# - Artifacts (responses, validation results)
# - Tags (tools_called)
```

Trace Logging with LangChain

MLflow 3 has native LangChain integration:

```python
```

```
# Enable autologging
mlflow.langchain.autolog()

# All agent interactions are automatically traced:
# - LLM calls with prompts and responses
# - Tool invocations with inputs/outputs
# - Agent reasoning steps
# - Total duration and cost
```

Model Registry

```python
# Register the agent as a model
with mlflow.start_run():
    mlflow.langchain.log_model(
        lc_model=agent,
        artifact_path="pension_agent",
        registered_model_name="pension_advisor_prod"
    )

# Load model for inference
loaded_agent = mlflow.langchain.load_model("models:/pension_advisor_prod/latest")
```

Governance & Validation

LLM Judge Implementation

The validation layer (Phase 6) uses a separate LLM to independently verify response quality:

```python
```

```python
# agent/validation.py
from typing import Dict, List, Any
from langchain_databricks import ChatDatabricks
import json

class LLMJudge:
    def __init__(self, endpoint: str = "databricks-claude-sonnet-4"):
        self.llm = ChatDatabricks(endpoint=endpoint, temperature=0)

    def validate(
        self,
        query: str,
        response: str,
        tool_outputs: List[Any],
        citations_required: bool = True
    ) -> Dict[str, Any]:
        """
        Validate response quality.

        Returns:
            {
                "is_valid": bool,
                "confidence": float,
                "violations": List[str],
                "reasoning": str
            }
        """

        prompt = self._create_validation_prompt(
            query, response, tool_outputs, citations_required
        )

        result = self.llm.invoke(prompt)

        try:
            validation = json.loads(result.content)
            return validation
        except json.JSONDecodeError:
            return {
                "is_valid": False,
                "confidence": 0.0,
                "violations": ["Failed to parse validation response"],
                "reasoning": result.content
            }

    def _create_validation_prompt(
```

```
        self,
        query: str,
        response: str,
        tool_outputs: List[Any],
        citations_required: bool
    ) -> str:
        return f"""
You are an independent quality validator for financial advice responses.

Query: {query}

Tool Outputs:
{self._format_tool_outputs(tool_outputs)}

Response to Validate:
{response}

Validation Criteria:
1. ACCURACY: Numbers in response must match tool outputs exactly
2. CITATIONS: Must reference specific regulations (e.g., "ATO Section 307-5")
3. DISCLAIMERS: Must include appropriate risk disclaimers
4. HALLUCINATION: No information not derived from tools or known regulations
5. COMPLETENESS: Fully answers the query

Evaluate and respond in JSON format:
{{
    "is_valid": true or false,
    "confidence": 0.0 to 1.0,
    "violations": ["list of any issues found"],
    "reasoning": "detailed explanation of validation decision"
}}
"""

    def _format_tool_outputs(self, tool_outputs: List[Any]) -> str:
        formatted = []
        for i, (action, observation) in enumerate(tool_outputs):
            formatted.append(f"Tool {i+1}: {action.tool}")
            formatted.append(f"Input: {action.tool_input}")
            formatted.append(f"Output: {observation}")
            formatted.append("---")
        return "\n".join(formatted)
```

Audit Trail Queries

Query the audit log for compliance:

```
sql
```

```sql
-- All interactions for a member
SELECT * FROM main.pension_advisory.audit_log
WHERE member_id = 'M12345'
ORDER BY timestamp DESC;

-- Failed validations
SELECT * FROM main.pension_advisory.audit_log
WHERE validation_verdict = 'false'
ORDER BY timestamp DESC;

-- High-cost queries
SELECT * FROM main.pension_advisory.audit_log
WHERE cost_usd > 0.01
ORDER BY cost_usd DESC;

-- Tool usage statistics
SELECT
    explode(tools_called) as tool_name,
    COUNT(*) as usage_count
FROM main.pension_advisory.audit_log
GROUP BY tool_name
ORDER BY usage_count DESC;
```

Deployment Options

Option 1: Databricks Apps (Production)

1. Create `databricks.yml`:

```
yaml
```

```yaml
# databricks.yml
resources:
  apps:
    pension_advisor:
      name: pension-advisor-prod
      description: "Multi-Country Pension Advisor"

      # App configuration
      config:
        command:
          - "streamlit"
          - "run"
          - "ui/app.py"
          - "--server.port=8080"

      # Resources
      compute:
        size: MEDIUM
        auto_stop_mins: 30

      # Environment variables
      env:
        - name: DATABRICKS_HOST
          value: "{{workspace.host}}"
        - name: CATALOG
          value: "main"
        - name: SCHEMA
          value: "pension_advisory"
```

2. Deploy:

```bash
# Authenticate
databricks auth login --host https://your-workspace.databricks.com

# Deploy app
databricks bundle deploy

# Start app
databricks bundle run pension_advisor
```

3. Access: App will be available at: `https://your-workspace.databricks.com/apps/pension-advisor-prod`

Option 2: Local Development (Laptop)

1. Set up authentication:

```bash
# Option A: Use Databricks CLI
databricks auth login --host https://your-workspace.databricks.com

# Option B: Set environment variables
export DATABRICKS_HOST="https://your-workspace.databricks.com"
export DATABRICKS_TOKEN="your_personal_access_token"
```

2. Run locally:

```bash
# Install dependencies
pip install -r requirements.txt

# Run Streamlit app
streamlit run ui/app.py

# Or run agent directly in Python
python -c "
from agent.pension_agent import PensionAgent

agent = PensionAgent()
result = agent.process_query(
    member_id='M12345',
    query='When can I access my pension?'
)
print(result['response'])
"
```

3. Local development workflow:

```python
```

```python
# test_local.py
from agent.pension_agent import PensionAgent
import mlflow

# Set experiment for local testing
mlflow.set_experiment("/Users/your_name/pension_agent_local_dev")

# Initialize agent
agent = PensionAgent(
    main_endpoint="databricks-claude-opus-4",
    judge_endpoint="databricks-claude-sonnet-4"
)

# Test query
result = agent.process_query(
    member_id="TEST_001",
    query="What's my withdrawal tax?",
    validate=True
)

print(f"Response: {result['response']}")
print(f"Cost: ${result['cost']['total']:.4f}")
print(f"Valid: {result['validation']['is_valid']}")
print(f"MLflow Run: {result['mlflow_run_id']}")
```

Code Examples

Complete Streamlit UI

```python
```

```python
# ui/app.py
import streamlit as st
from agent.pension_agent import PensionAgent
import pandas as pd

st.set_page_config(
    page_title="Pension Advisor",
    page_icon="🏦",
    layout="wide"
)

# Initialize agent (cached)
@st.cache_resource
def get_agent():
    return PensionAgent()

agent = get_agent()

# Sidebar: Member selection
st.sidebar.header("Member Selection")

# Fetch members from UC
@st.cache_data
def load_members():
    from databricks.sdk import WorkspaceClient
    w = WorkspaceClient()
    result = w.statement_execution.execute_statement(
        warehouse_id="your_warehouse_id",
        catalog="main",
        schema="pension_advisory",
        statement="SELECT member_id, name, age, country FROM main.pension_advisory.members"
    )
    return pd.DataFrame(
        result.result.data_array,
        columns=["member_id", "name", "age", "country"]
    )

members_df = load_members()
selected_member = st.sidebar.selectbox(
    "Select Member",
    members_df["member_id"],
    format_func=lambda x: members_df[members_df["member_id"] == x]["name"].values[0]
)

# Display member context
st.sidebar.subheader("Member Context")
```

```python
member_info = members_df[members_df["member_id"] == selected_member].iloc[0]
st.sidebar.write(f"**Age:** {member_info['age']}")
st.sidebar.write(f"**Country:** {member_info['country']}")

# Main area: Query interface
st.title("🏦 Retirement Planning Advisor")
st.markdown("Ask questions about your pension, tax implications, and withdrawal options.")

# Sample questions
st.subheader("Sample Questions")
col1, col2 = st.columns(2)
with col1:
    if st.button("When can I access my pension?"):
        query = "When can I access my pension?"
    if st.button("What are my tax implications?"):
        query = "What are my tax implications if I withdraw now?"
with col2:
    if st.button("What's my balance projection?"):
        query = "What will my balance be at retirement?"
    if st.button("Am I eligible for benefits?"):
        query = "Am I eligible for government benefits?"

# Text input
query = st.text_input("Or ask your own question:", "")

# Process query
if query:
    with st.spinner("Processing your query..."):

        # Show pipeline progress
        progress_bar = st.progress(0)
        status_text = st.empty()

        # Phases (for UI display)
        phases = [
            "Retrieving member data...",
            "Protecting your privacy...",
            "Planning response...",
            "Calculating...",
            "Synthesizing answer...",
            "Validating accuracy...",
            "Finalizing...",
            "Logging for audit..."
        ]

        for i, phase in enumerate(phases):
            status_text.text(phase)
```

```
        progress_bar.progress((i + 1) / len(phases))

    # Execute query
    result = agent.process_query(
        member_id=selected_member,
        query=query,
        validate=True
    )

    progress_bar.progress(1.0)
    status_text.text("Complete!")

    # Display response
    st.subheader("Response")
    st.markdown(result["response"])

    # Validation verdict
    validation = result["validation"]
    if validation["is_valid"]:
        st.success(f"✓ Response validated (Confidence: {validation['confidence']:.0%})")
    else:
        st.warning(f"⚠ Validation issues: {', '.join(validation['violations'])}")

    # Expandable: Details
    with st.expander("View Details"):
        col1, col2, col3 = st.columns(3)
        with col1:
            st.metric("Cost", f"${result['cost']['total']:.4f}")
        with col2:
            st.metric("Tools Used", len(result["tools_called"]))
        with col3:
            st.metric("MLflow Run", result["mlflow_run_id"][:8] + "...")

        st.subheader("Tools Called")
        for tool in result["tools_called"]:
            st.write(f"- {tool}")

        st.subheader("Validation Details")
        st.json(validation)

# Governance tab
with st.expander("📊 Governance & Audit"):
    st.subheader("Recent Interactions")

    # Query audit log
    @st.cache_data
    def load_audit_log(member_id, limit=10):
```

```
    from databricks.sdk import WorkspaceClient
    w = WorkspaceClient()
    result = w.statement_execution.execute_statement(
        warehouse_id="your_warehouse_id",
        catalog="main",
        schema="pension_advisory",
        statement=f"""
            SELECT timestamp, query, cost_usd, validation_verdict
            FROM main.pension_advisory.audit_log
            WHERE member_id = '{member_id}'
            ORDER BY timestamp DESC
            LIMIT {limit}
        """
    )
    return pd.DataFrame(
        result.result.data_array,
        columns=["Timestamp", "Query", "Cost", "Valid"]
    )

audit_df = load_audit_log(selected_member)
st.dataframe(audit_df, use_container_width=True)
```

Testing & Evaluation

Unit Tests

```python
```

```python
# tests/test_agent.py
import pytest
from agent.pension_agent import PensionAgent

@pytest.fixture
def agent():
    return PensionAgent()

def test_retrieve_member_data(agent):
    """Test member data retrieval."""
    member_data = agent._retrieve_member_data("M12345")
    assert "name" in member_data
    assert "age" in member_data
    assert "balance" in member_data

def test_privacy_anonymization():
    """Test PII anonymization."""
    from agent.privacy import anonymize_member_data, restore_member_name

    member_data = {"name": "John Smith", "age": 58}
    query = "When can John Smith retire?"

    anon_data, anon_query, original = anonymize_member_data(member_data, query)

    assert anon_data["name"] == "MEMBER_NAME_PLACEHOLDER"
    assert "John Smith" not in anon_query
    assert original == "John Smith"

    # Test restoration
    response = "MEMBER_NAME_PLACEHOLDER can retire at 60"
    restored = restore_member_name(response, original)
    assert "John Smith" in restored

def test_validation(agent):
    """Test LLM Judge validation."""
    result = agent._validate_response(
        query="What's my tax?",
        response="Your tax is 15% based on ATO rules.",
        tool_outputs=[("calculate_tax", "tax_rate: 15%")]
    )

    assert "is_valid" in result
    assert "confidence" in result
    assert "violations" in result

def test_end_to_end(agent):
```

```
    """Test full query processing."""
    result = agent.process_query(
        member_id="TEST_001",
        query="When can I access my pension?",
        validate=True
    )

    assert "response" in result
    assert "cost" in result
    assert "validation" in result
    assert result["cost"]["total"] > 0
```

Evaluation with MLflow

```python
```

```python
# evaluation/evaluate_agent.py
import mlflow
import pandas as pd
from agent.pension_agent import PensionAgent

# Load evaluation dataset
eval_data = pd.read_csv("evaluation/test_queries.csv")
# Columns: member_id, query, expected_answer, evaluation_criteria

agent = PensionAgent()

# Start evaluation run
with mlflow.start_run(run_name="agent_evaluation"):

    results = []

    for _, row in eval_data.iterrows():
        # Process query
        result = agent.process_query(
            member_id=row["member_id"],
            query=row["query"],
            validate=True
        )

        # Log individual result
        results.append({
            "query": row["query"],
            "response": result["response"],
            "cost": result["cost"]["total"],
            "is_valid": result["validation"]["is_valid"],
            "confidence": result["validation"]["confidence"]
        })

    # Calculate metrics
    results_df = pd.DataFrame(results)

    avg_cost = results_df["cost"].mean()
    validation_pass_rate = results_df["is_valid"].mean()
    avg_confidence = results_df["confidence"].mean()

    # Log metrics
    mlflow.log_metric("avg_cost_per_query", avg_cost)
    mlflow.log_metric("validation_pass_rate", validation_pass_rate)
    mlflow.log_metric("avg_validation_confidence", avg_confidence)

    # Log results artifact
```

```
    mlflow.log_table(results_df, "evaluation_results.json")

    print(f"Average Cost: ${avg_cost:.4f}")
    print(f"Validation Pass Rate: {validation_pass_rate:.1%}")
    print(f"Average Confidence: {avg_confidence:.1%}")
```

Best Practices

1. Cost Management

```python
# Implement cost limits
class CostAwareAgent(PensionAgent):
    def __init__(self, max_cost_per_query: float = 0.01, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.max_cost_per_query = max_cost_per_query

    def process_query(self, *args, **kwargs):
        result = super().process_query(*args, **kwargs)

        if result["cost"]["total"] > self.max_cost_per_query:
            # Log alert
            mlflow.log_param("cost_exceeded", True)
            # Could trigger notification

        return result
```

2. Error Handling

```python
# Graceful error handling
try:
    result = agent.process_query(member_id, query)
except Exception as e:
    # Log error
    mlflow.log_param("error", str(e))

    # Fallback response
    result = {
        "response": "I'm having trouble processing your query. Please contact support.",
        "error": str(e),
        "cost": {"total": 0}
    }
```

3. Caching for Performance

```python
from functools import lru_cache

class CachedPensionAgent(PensionAgent):
    @lru_cache(maxsize=100)
    def _retrieve_member_data(self, member_id: str):
        """Cache member data for 1 hour."""
        return super()._retrieve_member_data(member_id)
```

4. A/B Testing

```python
# Test different prompts or models
with mlflow.start_run(run_name="variant_a"):
    mlflow.log_param("variant", "detailed_prompt")
    agent_a = PensionAgent(prompt_version="detailed")
    result_a = agent_a.process_query(member_id, query)

with mlflow.start_run(run_name="variant_b"):
    mlflow.log_param("variant", "concise_prompt")
    agent_b = PensionAgent(prompt_version="concise")
    result_b = agent_b.process_query(member_id, query)

# Compare in MLflow UI
```

Troubleshooting

Common Issues

1. Authentication Errors

```bash
# Verify authentication
databricks auth env
```

2. Unity Catalog Permissions

```sql
-- Grant necessary permissions
GRANT USAGE ON CATALOG main TO `your_user`;
GRANT USAGE ON SCHEMA main.pension_advisory TO `your_user`;
GRANT SELECT ON TABLE main.pension_advisory.members TO `your_user`;
GRANT EXECUTE ON FUNCTION main.pension_advisory.au_calculate_tax TO `your_user`;
```

3. Model Serving Endpoint Not Found

```python
# List available endpoints
from databricks.sdk import WorkspaceClient
w = WorkspaceClient()
endpoints = w.serving_endpoints.list()
for e in endpoints:
    print(f"{e.name}: {e.state.ready}")
```

4. MLflow Tracking URI

```python
# Set tracking URI explicitly
mlflow.set_tracking_uri("databricks")
```

Performance Optimization

1. Batch Processing

```python
def process_batch(member_ids: List[str], query: str):
    """Process same query for multiple members."""
    with mlflow.start_run(run_name="batch_processing"):
        results = []

        for member_id in member_ids:
            result = agent.process_query(member_id, query, validate=False)
            results.append(result)

        # Validate batch in parallel (if needed)
        # ...

        return results
```

2. Async Processing

```python
```

```python
import asyncio
from concurrent.futures import ThreadPoolExecutor

async def process_query_async(agent, member_id, query):
    """Async query processing."""
    loop = asyncio.get_event_loop()
    with ThreadPoolExecutor() as executor:
        result = await loop.run_in_executor(
            executor,
            agent.process_query,
            member_id,
            query
        )
    return result
```

Next Steps

1. **Customize for Your Use Case**
   - Adapt UC functions for your regulatory environment
   - Modify prompt templates
   - Add domain-specific tools
2. **Scale to Production**
   - Set up CI/CD with Databricks Asset Bundles
   - Implement monitoring and alerting
   - Configure auto-scaling
3. **Enhance Capabilities**
   - Add more sophisticated validation logic
   - Implement multi-agent collaboration
   - Integrate with external data sources
4. **Optimize Costs**
   - Implement caching strategies
   - Use smaller models for simple queries
   - Batch similar queries

Resources

- **Databricks Agent Framework**: https://docs.databricks.com/en/generative-ai/agent-framework/index.html
- **MLflow 3 Documentation**: https://mlflow.org/docs/latest/index.html
- **Unity Catalog Functions**: https://docs.databricks.com/en/sql/language-manual/sql-ref-functions-udf-udaf.html
- **Databricks Apps**: https://docs.databricks.com/en/dev-tools/databricks-apps/index.html
- **Foundation Model APIs**: https://docs.databricks.com/en/machine-learning/foundation-models/index.html

---

Summary

This architecture demonstrates:

✅ **ReAct Framework** - Reason → Act loop with governance
✅ **Separation of Concerns** - LLM orchestrates, UC functions calculate
✅ **Full Governance** - Privacy, validation, audit at every step
✅ **MLflow 3 Integration** - Complete observability and tracking
✅ **Production-Ready** - Deploy to Databricks Apps or run locally
✅ **Hallucination Mitigation** - Deterministic calculations prevent errors

The 8-phase pipeline ensures quality, compliance, and auditability while delivering instant, personalized retirement advice at scale.