**## 📁 File Breakdown**

**### 1. \*\*app.py\*\* (Frontend) - 196 lines**

**\*\*Purpose\*\***: Streamlit chat interface

**\*\*Key Functions\*\***:

- `initialize\_state()`: Connects to Azure, loads data factories

- Sidebar: Data factory selection, connection status

- Chat interface: Message handling, real-time status updates

- `st.status()`: Shows what agent is doing in real-time

**\*\*User Experience\*\***:

```

User types → Shows in chat → Status widget appears →

Agent works (with live updates) → Status collapses →

Response displays

```

---

**### 2. \*\*agent.py\*\* (AI Brain) - 260 lines**

**\*\*Purpose\*\***: Autonomous AI agent using LangChain

**\*\*Key Components\*\***:

**#### a. \*\*LLM Configuration\*\***

```python

llm = AzureChatOpenAI(...)  # Azure OpenAI o4-mini model

```

**#### b. \*\*System Prompt\*\* (Lines 41-115)**

- Defines agent behavior

- Sets ADF-only scope

- Provides workflow instructions

- Explains tool usage

**#### c. \*\*StreamlitCallbackHandler\*\* (Lines 125-165)**

- Provides real-time status updates

- Maps tool calls to user-friendly messages

- Updates UI as agent works

**#### d. \*\*ChatAgent Class\*\* (Lines 169-241)**

```python

class ChatAgent:

    def \_\_init\_\_(self):

        self.memory = ConversationBufferMemory(...)  # Chat history

        agent = create\_openai\_tools\_agent(...)      # ReAct agent

        self.agent\_executor = AgentExecutor(        # Executes agent loop

            max\_iterations=5,                        # Max 5 steps

            early\_stopping\_method="generate"         # Stop when done

        )

```

**\*\*Agent Loop\*\***:

```

1. Receive query

2. Reason: What should I do?

3. Act: Call appropriate tool

4. Observe: What did I get?

5. Repeat: Until goal achieved or 5 iterations

6. Return: Final answer

```

---

**### 3. \*\*azure\_tools.py\*\* (API Integration) - 200 lines**

**\*\*Purpose\*\***: Wraps Azure Data Factory REST APIs as LangChain tools

**\*\*Authentication\*\***:

```python

credential = ClientSecretCredential(

    tenant\_id, client\_id, client\_secret

)

adf\_client = DataFactoryManagementClient(credential, subscription\_id)

```

**\*\*8 Tools\*\*** (All decorated with `@tool`):

| # | Tool Name | REST API | Purpose |

|---|-----------|----------|---------|

| 1 | `list\_all\_data\_factories\_in\_subscription` | GET /factories | List all ADFs |

| 2 | `list\_pipelines` | GET /pipelines | Get pipeline names |

| 3 | `get\_pipeline\_runs` | POST /queryPipelineRuns | Get run history |

| 4 | `get\_run\_activity\_logs` | POST /queryActivityruns | Get error details |

| 5 | `get\_pipeline\_definition` | GET /pipelines/{name} | Fetch JSON definition |

| 6 | `update\_pipeline` | PUT /pipelines/{name} | Apply changes |

| 7 | `create\_pipeline\_run` | POST /createRun | Trigger execution |

| 8 | `get\_pipeline\_run` | GET /pipelineruns/{id} | Check status |

---

* **list\_all\_data\_factories\_in\_subscription** – Retrieves all Azure Data Factories in the current subscription.
* **list\_pipelines** – Fetches the list of all pipelines within a specific Data Factory.
* **get\_pipeline\_runs** – Retrieves the execution history and run details of pipelines.
* **get\_run\_activity\_logs** – Fetches activity-level logs and error details for a specific pipeline run.
* **get\_pipeline\_definition** – Returns the JSON structure or definition of a given pipeline.
* **update\_pipeline** – Updates or modifies an existing pipeline’s configuration.
* **create\_pipeline\_run** – Triggers or starts the execution of a specified pipeline.
* **get\_pipeline\_run** – Retrieves the current status and metadata of a particular pipeline run.

how the agent decides which tool to call:

The Decision Process

1. OpenAI Function Calling (Tool Selection)

When you send a query, the agent uses OpenAI's function calling capability. Here's the flow:

User Query → LLM analyzes → Matches to tool descriptions → Selects best tool(s) → Returns function call

2. How LLM Sees Your Tools

Each tool in azure\_tools.py has a detailed description that the LLM reads:

@tool

def list\_pipelines(resource\_group\_name: str, data\_factory\_name: str) -> str:

"""

Lists all pipelines in the specified Azure Data Factory.

Args:

resource\_group\_name: The name of the resource group

data\_factory\_name: The name of the data factory

Returns:

A formatted string with pipeline names

"""

The LLM sees:

- Function name: list\_pipelines

- Description: "Lists all pipelines in the specified Azure Data Factory"

- Parameters needed: resource\_group\_name, data\_factory\_name

- What it returns: "A formatted string with pipeline names"

3. Real Example - Query to Tool Mapping

User asks: "Show me all pipelines in my data factory"

LLM's reasoning (behind the scenes):

1. User wants to see pipelines

2. I have a tool called "list\_pipelines"

3. Description says: "Lists all pipelines..."

4. This matches the user's intent perfectly

5. Parameters needed: resource\_group\_name, data\_factory\_name

6. I have these from session state

7. DECISION: Call list\_pipelines(rg\_name, adf\_name)

User asks: "Why did errorpipeline1 fail?"

LLM's reasoning:

1. User wants to know why a specific pipeline failed

2. Need to find failed runs first

→ Call get\_pipeline\_runs() with filter for failures

3. Get the run\_id from failed run

4. Need error details

→ Call get\_run\_activity\_logs(pipeline\_name, run\_id)

5. Analyze error message

6. DECISION: Make 2 tool calls sequentially

4. The Technical Implementation

In agent.py:130-180, here's how it works:

# System prompt tells the agent its capabilities

system\_message = """You are an Azure Data Factory assistant...

Available tools:

1. list\_pipelines - Lists all pipelines

2. get\_pipeline\_runs - Gets run history with filters

3. get\_run\_activity\_logs - Gets detailed error logs

...

Instructions:

- For listing queries, use list\_pipelines

- For error diagnosis, use get\_pipeline\_runs then get\_run\_activity\_logs

- For fixing, use get\_pipeline\_definition then update\_pipeline

"""

# Create agent with OpenAI Tools

agent = create\_openai\_tools\_agent(

llm=llm, # Azure OpenAI model (o4-mini)

tools=tools, # List of 8 tools

prompt=prompt # System instructions above

)

5. Function Calling Under the Hood

When LLM receives your query, Azure OpenAI API returns:

{

"role": "assistant",

"content": null,

"tool\_calls": [

{

"id": "call\_123",

"type": "function",

"function": {

"name": "list\_pipelines",

"arguments": "{\"resource\_group\_name\":\"rg-analytics\",\"data\_factory\_name\":\"adf-prod\"}"

}

}

]

}

LangChain's AgentExecutor then:

1. Parses the function call

2. Executes list\_pipelines("rg-analytics", "adf-prod")

3. Gets result from Azure API

4. Sends result back to LLM

5. LLM generates human-readable response

6. Multi-Step Reasoning (ReAct Pattern)

For complex queries, agent makes multiple decisions:

Query: "Fix the broken pipeline"

STEP 1:

Thought: "Need to find which pipeline is broken"

Action: get\_pipeline\_runs(status\_filter="Failed")

Observation: "errorpipeline1 failed at 2:30 PM, run\_id=abc123"

STEP 2:

Thought: "Need to know WHY it failed"

Action: get\_run\_activity\_logs("errorpipeline1", "abc123")

Observation: "CopyActivity1 failed: Source file 'movies.csv' not found"

STEP 3:

Thought: "Need current pipeline configuration"

Action: get\_pipeline\_definition("errorpipeline1")

Observation: { "activities": [...], "source": "movies.csv" }

STEP 4:

Thought: "File path is wrong, should be 'data/movies.csv'"

Action: update\_pipeline("errorpipeline1", {...updated definition...})

Observation: "Pipeline updated successfully"

STEP 5:

Thought: "Now trigger the pipeline to test fix"

Action: create\_pipeline\_run("errorpipeline1")

Observation: "Run started with ID xyz789"

Final Answer: "I fixed the pipeline by correcting the source file path..."

7. How Agent Learns Tool Usage

The system prompt in agent.py:133-177 teaches the agent:

When user asks to fix a pipeline, follow these steps:

1. Use get\_pipeline\_runs to find recent failed runs

2. Use get\_run\_activity\_logs to analyze the error

3. Use get\_pipeline\_definition to get current configuration

4. Determine if the issue can be fixed programmatically

5. If yes, use update\_pipeline with the corrected definition

6. Use create\_pipeline\_run to trigger the pipeline again

This is like training instructions that guide tool selection.

8. Key Factors in Tool Selection

The LLM considers:

1. Semantic matching: Query keywords match tool description

- "list" → list\_pipelines

- "failed" → get\_pipeline\_runs with filter

- "fix" → update\_pipeline

2. Parameter availability: Does it have required inputs?

- Has resource\_group\_name? ✅

- Has data\_factory\_name? ✅

- Then it CAN call the tool

3. Context from memory: Previous conversation matters

- User: "Show pipeline X"

- User: "Now fix it" ← LLM remembers X from context

4. System instructions: Explicit guidance in prompt

- "For diagnostics, use get\_run\_activity\_logs"

- LLM follows these rules

9. Visual Decision Tree

User Query

↓

Does it ask to LIST something?

YES → list\_pipelines or list\_all\_data\_factories

NO ↓

Does it ask about ERRORS or FAILURES?

YES → get\_pipeline\_runs (filter=Failed) → get\_run\_activity\_logs

NO ↓

Does it ask to FIX or UPDATE?

YES → get\_pipeline\_definition → update\_pipeline

NO ↓

Does it ask to RUN or TRIGGER?

YES → create\_pipeline\_run

NO ↓

Does it ask about STATUS?

YES → get\_pipeline\_run

NO ↓

Out of scope → Politely decline

10. Why This Works

OpenAI models are trained on:

- Function calling patterns

- API documentation structure

- Natural language to code mapping

Result: The model has learned to:

- Parse user intent

- Match it to available functions

- Extract required parameters

- Call functions in logical order

---

Summary

Agent decides which tool to call by:

1. ✅ Reading tool descriptions and parameters

2. ✅ Analyzing user query semantics

3. ✅ Following system prompt instructions

4. ✅ Using OpenAI function calling capability

5. ✅ Applying ReAct reasoning (Think → Act → Observe → Repeat)

Q: How did you add memory?

A: Used ConversationBufferMemory with StreamlitChatMessageHistory. Stores all messages in Streamlit session state. Agent sees chat

history via MessagesPlaceholder in prompt template, allowing context-aware responses like "show me that pipeline again."

Q: How do you limit scope to ADF only?

A: System prompt explicitly instructs: "You ONLY answer Azure Data Factory questions." Includes examples of what to reject (weather,

general programming) with pre-written rejection response. LLM follows these instructions because system prompts have highest priority in

OpenAI models.

The Decision Process

1. OpenAI Function Calling (Tool Selection)

When you send a query, the agent uses OpenAI's function calling capability. Here's the flow:

User Query → LLM analyzes → Matches to tool descriptions → Selects best tool(s) → Returns function call

2. How LLM Sees Your Tools

Each tool in azure\_tools.py has a detailed description that the LLM reads:

@tool

def list\_pipelines(resource\_group\_name: str, data\_factory\_name: str) -> str:

"""

Lists all pipelines in the specified Azure Data Factory.

Args:

resource\_group\_name: The name of the resource group

data\_factory\_name: The name of the data factory

Returns:

A formatted string with pipeline names

"""

The LLM sees:

- Function name: list\_pipelines

- Description: "Lists all pipelines in the specified Azure Data Factory"

- Parameters needed: resource\_group\_name, data\_factory\_name

- What it returns: "A formatted string with pipeline names"

3. Real Example - Query to Tool Mapping

User asks: "Show me all pipelines in my data factory"

LLM's reasoning (behind the scenes):

1. User wants to see pipelines

2. I have a tool called "list\_pipelines"

3. Description says: "Lists all pipelines..."

4. This matches the user's intent perfectly

5. Parameters needed: resource\_group\_name, data\_factory\_name

6. I have these from session state

7. DECISION: Call list\_pipelines(rg\_name, adf\_name)

User asks: "Why did errorpipeline1 fail?"

LLM's reasoning:

1. User wants to know why a specific pipeline failed

2. Need to find failed runs first

→ Call get\_pipeline\_runs() with filter for failures

3. Get the run\_id from failed run

4. Need error details

→ Call get\_run\_activity\_logs(pipeline\_name, run\_id)

5. Analyze error message

6. DECISION: Make 2 tool calls sequentially

4. The Technical Implementation

In agent.py:130-180, here's how it works:

# System prompt tells the agent its capabilities

system\_message = """You are an Azure Data Factory assistant...

Available tools:

1. list\_pipelines - Lists all pipelines

2. get\_pipeline\_runs - Gets run history with filters

3. get\_run\_activity\_logs - Gets detailed error logs

...

Instructions:

- For listing queries, use list\_pipelines

- For error diagnosis, use get\_pipeline\_runs then get\_run\_activity\_logs

- For fixing, use get\_pipeline\_definition then update\_pipeline

"""

# Create agent with OpenAI Tools

agent = create\_openai\_tools\_agent(

llm=llm, # Azure OpenAI model (o4-mini)

tools=tools, # List of 8 tools

prompt=prompt # System instructions above

)

5. Function Calling Under the Hood

When LLM receives your query, Azure OpenAI API returns:

{

"role": "assistant",

"content": null,

"tool\_calls": [

{

"id": "call\_123",

"type": "function",

"function": {

"name": "list\_pipelines",

"arguments": "{\"resource\_group\_name\":\"rg-analytics\",\"data\_factory\_name\":\"adf-prod\"}"

}

}

]

}

LangChain's AgentExecutor then:

1. Parses the function call

2. Executes list\_pipelines("rg-analytics", "adf-prod")

3. Gets result from Azure API

4. Sends result back to LLM

5. LLM generates human-readable response

6. Multi-Step Reasoning (ReAct Pattern)

For complex queries, agent makes multiple decisions:

Query: "Fix the broken pipeline"

STEP 1:

Thought: "Need to find which pipeline is broken"

Action: get\_pipeline\_runs(status\_filter="Failed")

Observation: "errorpipeline1 failed at 2:30 PM, run\_id=abc123"

STEP 2:

Thought: "Need to know WHY it failed"

Action: get\_run\_activity\_logs("errorpipeline1", "abc123")

Observation: "CopyActivity1 failed: Source file 'movies.csv' not found"

STEP 3:

Thought: "Need current pipeline configuration"

Action: get\_pipeline\_definition("errorpipeline1")

Observation: { "activities": [...], "source": "movies.csv" }

STEP 4:

Thought: "File path is wrong, should be 'data/movies.csv'"

Action: update\_pipeline("errorpipeline1", {...updated definition...})

Observation: "Pipeline updated successfully"

STEP 5:

Thought: "Now trigger the pipeline to test fix"

Action: create\_pipeline\_run("errorpipeline1")

Observation: "Run started with ID xyz789"

Final Answer: "I fixed the pipeline by correcting the source file path..."

7. How Agent Learns Tool Usage

The system prompt in agent.py:133-177 teaches the agent:

When user asks to fix a pipeline, follow these steps:

1. Use get\_pipeline\_runs to find recent failed runs

2. Use get\_run\_activity\_logs to analyze the error

3. Use get\_pipeline\_definition to get current configuration

4. Determine if the issue can be fixed programmatically

5. If yes, use update\_pipeline with the corrected definition

6. Use create\_pipeline\_run to trigger the pipeline again

This is like training instructions that guide tool selection.

8. Key Factors in Tool Selection

The LLM considers:

1. Semantic matching: Query keywords match tool description

- "list" → list\_pipelines

- "failed" → get\_pipeline\_runs with filter

- "fix" → update\_pipeline

2. Parameter availability: Does it have required inputs?

- Has resource\_group\_name? ✅

- Has data\_factory\_name? ✅

- Then it CAN call the tool

3. Context from memory: Previous conversation matters

- User: "Show pipeline X"

- User: "Now fix it" ← LLM remembers X from context

4. System instructions: Explicit guidance in prompt

- "For diagnostics, use get\_run\_activity\_logs"

- LLM follows these rules

9. Visual Decision Tree

User Query

↓

Does it ask to LIST something?

YES → list\_pipelines or list\_all\_data\_factories

NO ↓

Does it ask about ERRORS or FAILURES?

YES → get\_pipeline\_runs (filter=Failed) → get\_run\_activity\_logs

NO ↓

Does it ask to FIX or UPDATE?

YES → get\_pipeline\_definition → update\_pipeline

NO ↓

Does it ask to RUN or TRIGGER?

YES → create\_pipeline\_run

NO ↓

Does it ask about STATUS?

YES → get\_pipeline\_run

NO ↓

Out of scope → Politely decline

10. Why This Works

OpenAI models are trained on:

- Function calling patterns

- API documentation structure

- Natural language to code mapping

Result: The model has learned to:

- Parse user intent

- Match it to available functions

- Extract required parameters

- Call functions in logical order

---

Summary

Agent decides which tool to call by:

1. ✅ Reading tool descriptions and parameters

2. ✅ Analyzing user query semantics

3. ✅ Following system prompt instructions

4. ✅ Using OpenAI function calling capability

5. ✅ Applying ReAct reasoning (Think → Act → Observe → Repeat)

It's like a junior engineer who:

- Reads API documentation (tool descriptions)

- Understands what you need (query analysis)

- Knows which API to call (function calling)

- Can chain multiple API calls (ReAct pattern)

No hardcoded rules - the LLM figures it out intelligently based on context! 🧠

> what we have done to add meomery and limit the question answer to the adf factory only if we ask something othe than that it wont give

the answer,how we have done it?

● Great question! Let me explain the two key implementations:

1. Memory Implementation (Chat History)

● Read(agent.py)

⎿ Read 25 lines

● Perfect! Let me explain exactly how we implemented both features:

1. Memory Implementation (Remember Chat History)

Location: agent.py:90-94

self.memory = ConversationBufferMemory(

memory\_key="chat\_history",

return\_messages=True,

chat\_memory=StreamlitChatMessageHistory(key=session\_key),

)

How it works:

1. ConversationBufferMemory: Stores ALL chat messages in memory (no limit)

- Keeps full conversation history

- No token counting (that's why we switched from ConversationSummaryBufferMemory)

2. StreamlitChatMessageHistory: Persists messages in Streamlit session

- Stores in st.session\_state["langchain\_messages"]

- Survives across interactions (until browser closes)

- Each user gets isolated session

3. memory\_key="chat\_history": Injects history into prompt

- Look at agent.py:135 - prompt template has MessagesPlaceholder(variable\_name="chat\_history")

- Agent sees previous Q&A when making decisions

Example Flow:

User: "List all pipelines"

Agent: [Shows 3 pipelines: pipeline1, pipeline2, errorpipeline1]

Memory stores: User message + Agent response

User: "Show me details of the first one" ← Doesn't specify name!

Agent: Checks memory → Sees "first one" = pipeline1 → Fetches details

Location: agent.py:135 (Prompt Template)

prompt = ChatPromptTemplate.from\_messages([

("system", system\_prompt\_template),

MessagesPlaceholder(variable\_name="chat\_history"), ← Memory injected here

("human", "{input}"),

MessagesPlaceholder(variable\_name="agent\_scratchpad"),

])

What agent sees:

System: You are an Azure Data Factory assistant...

[CHAT HISTORY FROM MEMORY]

User: List all pipelines

Assistant: Here are your pipelines: ...

[END CHAT HISTORY]

Human: Show me details of the first one

---

2. ADF-Only Scope (Reject Other Questions)

Location: agent.py:41-53 (System Prompt)

system\_prompt\_template = """You are an expert AI assistant specialized EXCLUSIVELY in Azure Data Factory (ADF).

\*\*IMPORTANT SCOPE RESTRICTIONS:\*\*

- You ONLY answer questions related to Azure Data Factory, pipelines, data integration, ETL/ELT processes, and related Azure services.

- If a user asks questions outside of Azure Data Factory scope (e.g., "What's the latest ChatGPT version?", "Tell me about Python

programming", "What's the weather?"), you MUST respond with:

"I apologize, but I'm specifically designed to assist with Azure Data Factory queries only. I can help you with:

- Pipeline management and monitoring

- Error diagnosis and troubleshooting

- Pipeline run analysis

- Activity logs and debugging

- Pipeline fixes and optimizations

Please ask me anything related to Azure Data Factory!"

How it works:

1. System prompt is FIRST message: LLM sees this instruction before EVERY query

2. Explicit examples: Shows exactly what to reject (weather, Python, ChatGPT)

3. Pre-written response: Agent must use this exact response for off-topic queries

4. Enforced by LLM: The AI model follows these instructions inherently

Example Test:

User: "What's the weather today?"

LLM reads system prompt:

- "You ONLY answer questions related to Azure Data Factory"

- "If user asks about weather... you MUST respond with: I apologize..."

LLM generates:

"I apologize, but I'm specifically designed to assist with Azure Data Factory queries only..."

Why this works:

- OpenAI models are TRAINED to follow system instructions

- System prompt has highest priority in the message hierarchy

- Agent has NO TOOLS for non-ADF tasks (can't execute even if it wanted to)

---

Visual Summary

Memory Flow:

┌─────────────────────────────────────────┐

│ User: "List pipelines" │

└───────────────┬─────────────────────────┘

↓

┌─────────────────────────────────────────┐

│ Agent: "pipeline1, pipeline2, ..." │

└───────────────┬─────────────────────────┘

↓

┌─────────────────────────────────────────┐

│ STORED IN MEMORY (session\_state) │

│ - User message │

│ - Agent response │

└───────────────┬─────────────────────────┘

↓

┌─────────────────────────────────────────┐

│ User: "Show details of first one" │ ← Doesn't say "pipeline1"

└───────────────┬─────────────────────────┘

↓

┌─────────────────────────────────────────┐

│ Agent reads MEMORY: │

│ - Previous Q: "List pipelines" │

│ - Previous A: "pipeline1, pipeline2..." │

│ - Current Q: "details of first one" │

│ → Understands "first one" = pipeline1 │

└───────────────┬─────────────────────────┘

↓

┌─────────────────────────────────────────┐

│ Calls: get\_pipeline\_definition("pipeline1") │

└─────────────────────────────────────────┘

Scope Restriction Flow:

┌─────────────────────────────────────────┐

│ User: "What's the weather?" │

└───────────────┬─────────────────────────┘

↓

┌─────────────────────────────────────────┐

│ LLM reads SYSTEM PROMPT: │

│ "You ONLY answer ADF questions" │

│ "If user asks about weather... respond: │

│ I apologize, but..." │

└───────────────┬─────────────────────────┘

↓

┌─────────────────────────────────────────┐

│ LLM generates REJECTION response │

│ (follows system instruction) │

└───────────────┬─────────────────────────┘

↓

┌─────────────────────────────────────────┐

│ User sees: "I apologize, but I'm │

│ specifically designed for ADF only..." │

└─────────────────────────────────────────┘

---

Quick Answer for SME:

Q: How did you add memory?

A: Used ConversationBufferMemory with StreamlitChatMessageHistory. Stores all messages in Streamlit session state. Agent sees chat

history via MessagesPlaceholder in prompt template, allowing context-aware responses like "show me that pipeline again."

Q: How do you limit scope to ADF only?

A: System prompt explicitly instructs: "You ONLY answer Azure Data Factory questions." Includes examples of what to reject (weather,

general programming) with pre-written rejection response. LLM follows these instructions because system prompts have highest priority in

OpenAI models.

Code locations:

- Memory: agent.py:90-94 + agent.py:135

- Scope: agent.py:41-53

Simple, effective, and reliable! 🎯

> whats all happening in agent.py file?

● Let me break down everything happening in the agent.py file step by step:

● Read(agent.py)

⎿ Read 313 lines

● Perfect! Let me break down everything happening in agent.py in a clear, structured way:

---

Complete Breakdown of agent.py (312 lines)

Section 1: Imports & Setup (Lines 1-21)

What's happening:

import os, json, streamlit as st

from langchain\_openai import AzureChatOpenAI

from langchain.agents import AgentExecutor, create\_openai\_tools\_agent

from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder

from langchain.memory import ConversationBufferMemory

from langchain\_community.chat\_message\_histories import StreamlitChatMessageHistory

from langchain.callbacks.base import BaseCallbackHandler

from azure\_tools import [8 tools imported]

Purpose: Loads all required libraries and imports the 8 Azure Data Factory tools from azure\_tools.py

Why important: Sets up LangChain framework for agent orchestration and Streamlit for UI callbacks

---

Section 2: LLM Configuration (Lines 23-28)

llm = AzureChatOpenAI(

azure\_endpoint=os.getenv("AZURE\_OPENAI\_ENDPOINT"),

api\_key=os.getenv("AZURE\_OPENAI\_API\_KEY"),

azure\_deployment=os.getenv("AZURE\_OPENAI\_DEPLOYMENT\_NAME"), # o4-mini-6

api\_version=os.getenv("AZURE\_OPENAI\_API\_VERSION"),

)

What's happening:

- Creates connection to Azure OpenAI service

- Loads credentials from .env file

- Uses o4-mini-2025-04-16 model deployment

Why important: This is the "brain" - the LLM that understands natural language and decides which tools to call

For SME: "We connect to Azure OpenAI using service credentials stored in environment variables for security"

---

Section 3: Tools List (Lines 30-39)

tools = [

list\_pipelines,

get\_pipeline\_runs,

get\_run\_activity\_logs,

list\_all\_data\_factories\_in\_subscription,

get\_pipeline\_definition,

update\_pipeline,

create\_pipeline\_run,

get\_pipeline\_run,

]

What's happening: Creates a list of 8 available tools the agent can use

Why important: Agent uses OpenAI function calling to select from these tools based on user query

For SME: "These are the 8 Azure Data Factory operations our agent can perform - like giving it a toolkit"

---

Section 4: System Prompt (Lines 41-114)

This is the MOST IMPORTANT section - it's the agent's "instruction manual"

Part A: Scope Restrictions (Lines 43-53)

\*\*IMPORTANT SCOPE RESTRICTIONS:\*\*

- You ONLY answer questions related to Azure Data Factory

- If user asks about weather, ChatGPT, etc. → Reject with standard message

What's happening: Tells agent to ONLY handle ADF questions

Why important: Prevents agent from answering off-topic queries (security + focus)

For SME: "This is how we enforce ADF-only scope - through explicit LLM instructions"

---

Part B: Capabilities List (Lines 55-65)

\*\*YOUR CAPABILITIES:\*\*

1. List and inspect pipelines

2. Retrieve pipeline run history

3. Analyze activity logs

...

What's happening: Lists what agent can do (matches the 8 tools)

Why important: Helps LLM understand its role and available actions

---

Part C: Auto-Fix Workflow (Lines 67-80)

\*\*AUTOMATIC PIPELINE FIXING WORKFLOW:\*\*

1. Get Recent Runs → get\_pipeline\_runs

2. Get Activity Logs → get\_run\_activity\_logs

3. Analyze error

4. If fixable → Get definition → Explain → Suggest fix

5. If not fixable → Explain manual steps

What's happening: Step-by-step instructions for fixing pipelines

Why important: Guides agent's reasoning process for complex multi-step tasks

For SME: "This is the workflow that makes the agent autonomous - it knows the exact steps to diagnose and fix issues"

---

Part D: Tool Usage Instructions (Lines 82-100)

\*\*IMPORTANT FOR update\_pipeline TOOL:\*\*

Must provide complete pipeline\_definition with:

- activities (required)

- parameters, variables, annotations (optional)

Example call with escaped JSON {{ }}

What's happening: Detailed instructions on how to call update\_pipeline correctly

Why important: Prevents validation errors (we had this bug earlier - fixed by adding these instructions)

---

Part E: Response Style (Lines 102-113)

- Be clear and concise

- Use structured formatting

- Confirm context before actions

- Explain each step when fixing

- Ask for missing information

What's happening: Guidelines for how agent should communicate with user

Why important: Ensures professional, helpful responses

---

Section 5: Prompt Template (Lines 116-123)

prompt = ChatPromptTemplate.from\_messages([

("system", system\_prompt\_template), # Instructions

MessagesPlaceholder(variable\_name="chat\_history"), # Memory

("human", "{input}"), # User query

MessagesPlaceholder(variable\_name="agent\_scratchpad"), # Tool results

])

What's happening: Defines the structure of messages sent to LLM

Visual representation:

┌────────────────────────────────────┐

│ System: You are an ADF assistant...│ ← Instructions

├────────────────────────────────────┤

│ [CHAT HISTORY] │ ← Memory

│ User: List pipelines │

│ AI: Here are your pipelines... │

├────────────────────────────────────┤

│ Human: Show me errorpipeline1 │ ← Current query

├────────────────────────────────────┤

│ [AGENT SCRATCHPAD] │ ← Tool call results

│ Calling get\_pipeline\_definition... │

│ Result: {...} │

└────────────────────────────────────┘

Why important: This structure enables memory (chat history) and tool use (scratchpad)

For SME: "The prompt template combines system instructions, conversation history, current query, and tool results into a single context

for the AI"

---

Section 6: StreamlitCallbackHandler Class (Lines 126-168)

Purpose: Show real-time status updates in the UI

class StreamlitCallbackHandler(BaseCallbackHandler):

def \_\_init\_\_(self, status\_container):

self.status\_container = status\_container # Streamlit status widget

def on\_tool\_start(self, serialized, input\_str, \*\*kwargs):

"""Called when agent starts using a tool"""

tool\_name = serialized.get("name")

# Map tool names to friendly messages

tool\_messages = {

"list\_pipelines": "📋 Fetching list of pipelines...",

"get\_pipeline\_runs": "🔄 Retrieving pipeline run history...",

...

}

self.status\_container.update(label=message) # Update UI

def on\_tool\_end(self, output, \*\*kwargs):

"""Called when tool finishes"""

pass

def on\_tool\_error(self, error, \*\*kwargs):

"""Called when tool fails"""

self.status\_container.update(label=f"❌ Error: {error}")

def on\_llm\_start(self, serialized, prompts, \*\*kwargs):

"""Called when LLM starts thinking"""

self.status\_container.update(label="🤖 AI analyzing...")

def on\_agent\_action(self, action, \*\*kwargs):

"""Called when agent decides on action"""

self.status\_container.update(label="💭 Processing...")

What's happening:

- Monitors agent's activities (tool calls, LLM thinking, errors)

- Updates Streamlit UI in real-time with friendly messages

Why important: Provides verbose feedback so user sees what's happening behind the scenes

For SME: "This callback handler intercepts agent events and displays real-time status updates like '📋 Fetching pipelines...' in the UI"

Example flow:

User asks: "Why did pipeline X fail?"

Status updates shown:

1. "🤖 AI analyzing your request..."

2. "🔄 Retrieving pipeline run history..."

3. "📊 Analyzing activity logs..."

4. "✅ Complete" (status collapses)

---

Section 7: ChatAgent Class (Lines 170-244)

This is the CORE agent class that orchestrates everything

Part A: Initialization (Lines 171-192)

class ChatAgent:

def \_\_init\_\_(self, session\_key="langchain\_messages"):

# Step 1: Set up memory

self.memory = ConversationBufferMemory(

memory\_key="chat\_history",

return\_messages=True,

chat\_memory=StreamlitChatMessageHistory(key=session\_key),

)

# Step 2: Create agent

agent = create\_openai\_tools\_agent(llm, tools, prompt)

# Step 3: Create executor

self.agent\_executor = AgentExecutor(

agent=agent,

tools=tools,

memory=self.memory,

verbose=True, # Print reasoning to console

handle\_parsing\_errors=True, # Graceful error recovery

max\_iterations=5, # Max reasoning loops

return\_intermediate\_steps=False, # Don't return tool results

early\_stopping\_method="generate", # Stop early if confident

)

What's happening:

1. Memory setup: Stores chat history in Streamlit session

2. Agent creation: Combines LLM + tools + prompt template

3. Executor creation: Wraps agent with execution controls

Why important: This initialization creates the autonomous agent with memory, tools, and reasoning capabilities

For SME: "When we initialize ChatAgent, we give it memory to remember conversations, 8 tools to use, and constraints like max 5 reasoning

steps for performance"

Configuration choices explained:

- max\_iterations=5: Prevents infinite reasoning loops (optimized from 10)

- early\_stopping\_method="generate": Agent stops as soon as it has an answer

- handle\_parsing\_errors=True: If LLM generates invalid tool call, agent recovers

- verbose=True: Prints ReAct reasoning to terminal for debugging

---

Part B: get\_agent\_response Method (Lines 194-244)

This is called every time user sends a message

def get\_agent\_response(self, user\_query, resource\_group, data\_factory, status\_container):

# Step 1: Build contextual query

contextual\_query = f"""

Current context:

- Resource Group: '{resource\_group}'

- Data Factory: '{data\_factory}'

User's question: {user\_query}

"""

What's happening: Adds resource group and data factory context to user query

Why important: Agent always knows which ADF it's working with

Example:

User types: "List pipelines"

Agent sees:

Context:

- Resource Group: 'rg-analytics'

- Data Factory: 'adf-production'

User's question: List pipelines

---

# Step 2: Create callback handler for status updates

callback\_handler = StreamlitCallbackHandler(status\_container)

# Step 3: Execute agent

response = self.agent\_executor.invoke(

{"input": contextual\_query},

config={"callbacks": [callback\_handler]}

)

What's happening:

1. Creates callback handler to show real-time updates

2. Invokes agent with query

3. Agent starts ReAct loop (Think → Act → Observe → Repeat)

Behind the scenes during invoke():

1. LLM reads: System prompt + Chat history + User query

2. LLM thinks: "User wants to list pipelines"

3. LLM decides: Call list\_pipelines(rg, adf)

4. Callback: on\_tool\_start → Shows "📋 Fetching pipelines..."

5. Tool executes: Calls Azure API

6. Tool returns: ["pipeline1", "pipeline2", "errorpipeline1"]

7. LLM sees result

8. LLM generates: "Here are your 3 pipelines: ..."

9. Returns response to user

---

# Step 4: Extract response with error handling

if not isinstance(response, dict):

return "Unexpected response type. Please try again."

agent\_output = response.get("output")

if agent\_output is None:

agent\_output = response.get("result") or response.get("text")

if agent\_output is None or not isinstance(agent\_output, str):

return "I received incomplete response. Please rephrase."

if not agent\_output.strip():

return "I couldn't generate a response. Try different wording."

return agent\_output

What's happening: Robust error handling to extract the agent's final answer

Why important: Different LangChain versions use different response keys ("output" vs "result" vs "text")

For SME: "We have multiple fallbacks to extract the agent's response, making the system resilient to LangChain version changes"

---

except Exception as e:

import traceback

error\_msg = str(e)

traceback\_str = traceback.format\_exc()

print(f"Agent error: {error\_msg}")

print(f"Traceback: {traceback\_str}")

return f"I encountered an error: {error\_msg}\n\nPlease try rephrasing..."

What's happening: Catches any exceptions, logs them, shows friendly error to user

Why important: Graceful degradation - user sees helpful message instead of crash

---

Section 8: Helper Functions (Lines 247-311)

Function 1: get\_error\_analysis (Lines 247-265)

def get\_error\_analysis(error\_message: str) -> str:

"""Asks LLM to analyze an error and provide human-readable explanation"""

analysis\_prompt = f"""

As an expert ADF developer, analyze this error:

1. What it means in plain language

2. The likely root cause

3. Steps to fix

4. Preventive measures

Error: {error\_message}

"""

response = llm.invoke(analysis\_prompt)

return response.content

What's happening: Standalone function to explain ADF errors in simple terms

When used: NOT currently used in main workflow (legacy function)

For SME: "This function can convert cryptic Azure errors into plain English explanations - not currently integrated but available for

future use"

---

Function 2: get\_pipeline\_fix\_json (Lines 268-301)

def get\_pipeline\_fix\_json(pipeline\_definition: str, error\_message: str, activity\_name: str) -> str:

"""Asks LLM to fix a pipeline by returning modified JSON"""

analysis\_prompt = f"""

You are an ADF automated debugging agent.

Given:

- Pipeline JSON: {pipeline\_definition}

- Failed activity: {activity\_name}

- Error: {error\_message}

Output one of:

1. Fixed JSON (if programmatically fixable)

2. {{"manual\_intervention\_required": "explanation"}} (if needs manual fix)

Pipeline: {pipeline\_definition}

Failed Activity: {activity\_name}

Error: {error\_message}

"""

response = llm.invoke(analysis\_prompt)

return response.content

What's happening: Standalone function to generate fixed pipeline JSON

When used: NOT currently used (legacy function, agent does fixing via tools now)

Why kept: Could be useful for batch fixing or alternative workflows

For SME: "This function asks the AI to generate a fixed pipeline JSON given an error - not currently used but demonstrates our

multi-approach capabilities"

---

Function 3: check\_openai\_connection (Lines 304-311)

def check\_openai\_connection():

"""Quick test to verify OpenAI credentials work"""

try:

llm.invoke("test")

return True

except Exception as e:

print(f"OpenAI connection check failed: {e}")

return False

What's happening: Sends test message to verify Azure OpenAI connection

When used: Called in app.py on startup to validate credentials before showing UI

Why important: Fails fast if credentials are wrong instead of confusing user later

For SME: "We test the Azure OpenAI connection at startup to catch credential issues early"

---

Complete Data Flow Diagram

┌─────────────────────────────────────────────────────┐

│ User types: "Why did errorpipeline1 fail?" │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ app.py: Captures query, passes to ChatAgent │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ agent.py: ChatAgent.get\_agent\_response() │

│ - Adds context (RG, ADF) │

│ - Creates callback handler │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ AgentExecutor.invoke() starts ReAct loop │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ LLM reads: │

│ - System prompt (instructions) │

│ - Chat history (memory) │

│ - User query with context │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ LLM thinks (ReAct reasoning): │

│ "Need to find failed run for errorpipeline1" │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ LLM decides: Call get\_pipeline\_runs() │

│ Function calling: {"name": "get\_pipeline\_runs", │

│ "args": {...}} │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ StreamlitCallbackHandler.on\_tool\_start() │

│ Updates UI: "🔄 Retrieving pipeline run history..." │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ Tool executes: azure\_tools.get\_pipeline\_runs() │

│ → Calls Azure REST API │

│ → Returns: [run data with status="Failed"] │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ LLM observes result: "Found failed run abc123" │

│ LLM thinks: "Now need error details" │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ LLM decides: Call get\_run\_activity\_logs() │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ StreamlitCallbackHandler.on\_tool\_start() │

│ Updates UI: "📊 Analyzing activity logs..." │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ Tool executes: azure\_tools.get\_run\_activity\_logs() │

│ → Calls Azure REST API │

│ → Returns: "CopyActivity1 failed: File not found" │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ LLM observes error: "movies.csv not found" │

│ LLM thinks: "I have enough info to answer" │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ LLM generates final answer: │

│ "Pipeline failed because source file │

│ 'movies.csv' was not found in the specified │

│ location. The CopyActivity1 couldn't read..." │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ AgentExecutor returns response dict │

│ {"output": "Pipeline failed because..."} │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ get\_agent\_response() extracts "output" key │

│ Returns string to app.py │

└──────────────────────┬──────────────────────────────┘

↓

┌─────────────────────────────────────────────────────┐

│ app.py displays response to user │

│ Status widget collapses │

└─────────────────────────────────────────────────────┘

---

Quick Reference for SME Questions

Q: What makes this agent "agentic"?

A: Lines 67-80 define autonomous workflow. Agent reasons through multi-step tasks (ReAct pattern in AgentExecutor). It decides which

tools to call, in what order, without hardcoded logic.

Q: How does memory work?

A: Lines 174-178 + Line 119. ConversationBufferMemory stores all messages in Streamlit session. MessagesPlaceholder injects history into

prompt. Agent sees past conversations.

Q: How do you limit scope to ADF only?

A: Lines 43-53. System prompt explicitly restricts scope with examples of what to reject and exact rejection message to use.  
  
 Q: How does real-time status work?

A: Lines 126-168. StreamlitCallbackHandler intercepts agent events (tool start, LLM start, errors) and updates Streamlit status widget

with friendly messages.

Q: What prevents infinite loops?

A: Line 186: max\_iterations=5 limits reasoning steps. Line 188: early\_stopping\_method="generate" stops as soon as agent has answer.

Q: How does agent decide which tool to call?

A: Lines 179-189. create\_openai\_tools\_agent gives LLM access to tool descriptions. OpenAI function calling matches user intent to tool

based on descriptions. System prompt (lines 67-80) guides decision-making.

Q: What's the difference between agent and llm variables?

A:

- llm (line 23): Raw Azure OpenAI connection - just generates text

- agent (line 179): LLM + tools + prompt template - can use tools

- agent\_executor (line 180): Agent + memory + execution controls - full system

---  
  
  
Your Project Follows BOTH! ✅

Your project is an AUTONOMOUS AGENT that uses the ReAct PATTERN.

Let me explain the relationship:

---

Understanding the Terms

1. ReAct Pattern = HOW the agent thinks

ReAct = Reasoning + Acting

It's a METHOD of decision-making:

┌─────────────────────┐

│ THOUGHT (Reasoning) │ "What do I need to know?"

└──────────┬──────────┘

↓

┌─────────────────────┐

│ ACTION (Acting) │ Call a tool

└──────────┬──────────┘

↓

┌─────────────────────┐

│ OBSERVATION │ See the result

└──────────┬──────────┘

↓

(Loop until done)

2. Autonomous Agent = WHAT your system is

Autonomous = Self-governing, makes its own decisions

Your agent is autonomous because it:

- Decides WHICH tools to call (not hardcoded)

- Decides WHEN to stop (has enough info)

- Decides HOW to chain multiple tools

- Does NOT need step-by-step instructions

---

The Relationship

┌──────────────────────────────────────────────────┐

│ YOUR PROJECT: AUTONOMOUS AGENT │

│ │

│ ┌────────────────────────────────────────────┐ │

│ │ USES: ReAct Pattern │ │

│ │ │ │

│ │ This is the reasoning method that │ │

│ │ ENABLES the agent to be autonomous │ │

│ └────────────────────────────────────────────┘ │

│ │

│ Other characteristics: │

│ • Memory (chat history) │

│ • 8 tools available │

│ • Function calling (OpenAI) │

│ • Max 5 iterations │

│ • Early stopping │

└──────────────────────────────────────────────────┘

---

Proof Your Agent is Autonomous

Looking at your code in agent.py:180-189:

self.agent\_executor = AgentExecutor(

agent=agent, # ← This is an AUTONOMOUS agent

tools=tools, # ← Has 8 tools to choose from

memory=self.memory, # ← Remembers context

verbose=True,

handle\_parsing\_errors=True,

max\_iterations=5, # ← Can loop multiple times

return\_intermediate\_steps=False,

early\_stopping\_method="generate", # ← Decides when to stop

)

Key evidence:

- max\_iterations=5 → Agent can make multiple decisions in a loop

- early\_stopping\_method="generate" → Agent decides when it has enough info

- tools=[8 tools] → Agent selects from options (not hardcoded which tool to use)

---

Proof Your Agent Uses ReAct Pattern

From your data flow table (the one you selected):

| Step | Component | Action | What's Happening |

|-------|------------------------|--------------|-----------------------------------------|

| 4 | OpenAI | Reasoning | "Need to find failed runs" ← THOUGHT |

| 5 | OpenAI | Decision | Call get\_pipeline\_runs() ← ACTION |

| 6-7 | azure\_tools.py + Azure | Execute tool | Tool executes |

| 8 | OpenAI | Observes | "Found failed run abc123" ← OBSERVATION |

| 9 | OpenAI | Reasoning | "Need error details" ← THOUGHT (Loop!) |

| 10 | OpenAI | Decision | Call get\_run\_activity\_logs() ← ACTION |

| 11-12 | azure\_tools.py + Azure | Execute tool | Tool executes |

| 13 | OpenAI | Observes | "Error is missing file" ← OBSERVATION |

| 14 | OpenAI | Generates | Final answer ← DONE |

This is classic ReAct:

- Thought (step 4) → Act (step 5) → Observe (step 8) → Thought (step 9) → Act (step 10) → Observe (step 13) → Generate

---

Comparison with Non-Autonomous Systems

Hardcoded System (NOT Autonomous):

# User: "Why did pipeline fail?"

def diagnose\_pipeline():

runs = get\_pipeline\_runs() # Step 1 always

logs = get\_activity\_logs() # Step 2 always

return analyze(logs) # Step 3 always

# Same steps EVERY TIME, no decision-making

Your Autonomous Agent:

# User: "Why did pipeline fail?"

agent decides:

→ Hmm, need to find failures first

→ Call get\_pipeline\_runs()

→ Sees failed run

→ Decides: Need more details

→ Call get\_activity\_logs()

→ Sees error message

→ Decides: I have enough info now

→ Generate answer

# DIFFERENT steps based on context and results

---

Examples Showing Autonomy

Example 1: Simple Query (1 tool call)

User: "List all pipelines"

Agent reasoning:

- Thought: "User wants pipeline list"

- Action: Call list\_pipelines()

- Observation: Got list

- Decision: Done! (Only 1 iteration)

Example 2: Complex Query (3 tool calls)

User: "Fix the broken pipeline"

Agent reasoning:

- Thought: "Need to find broken pipeline"

- Action: get\_pipeline\_runs(status="Failed")

- Observation: Found errorpipeline1

- Thought: "Need to know WHY it failed"

- Action: get\_run\_activity\_logs()

- Observation: movies.csv not found

- Thought: "Need current pipeline config"

- Action: get\_pipeline\_definition()

- Observation: Got JSON config

- Decision: Generate fix suggestion

The agent autonomously decided to make 3 calls. A hardcoded system would always make the same number of calls.

---

For Your SME Review

Question: "Is this an autonomous agent?"

Answer:

"Yes, it's a fully autonomous agent. It doesn't follow a hardcoded script. Instead, it uses the ReAct pattern to reason through each

query step-by-step. Based on what it learns from each API call, it autonomously decides what to do next - whether to call another tool,

which tool to call, or if it has enough information to answer. This is enabled by LangChain's AgentExecutor with OpenAI function

calling."

Question: "What is the ReAct pattern?"

Answer:

"ReAct stands for Reasoning plus Acting. It's a pattern where the AI thinks about what it needs, takes an action (calls a tool), observes

the result, and repeats this loop until it has enough information. For example, when asked 'Why did the pipeline fail?', it reasons 'I

need to find failed runs', calls get\_pipeline\_runs, observes the failed run ID, reasons 'Now I need error details', calls

get\_activity\_logs, and generates the final answer. This thinking-acting loop is what makes the agent autonomous."

Question: "So it's both?"

Answer:

"Exactly! The agent IS autonomous, and it USES the ReAct pattern to achieve that autonomy. Think of it this way: 'Autonomous' describes

what the system is (self-governing), and 'ReAct' describes how it works (the thinking method). They're complementary, not mutually

exclusive."