Enterprise MLOps Agentic Architecture: Integrating Google ADK, Chainlit, and FastMCP

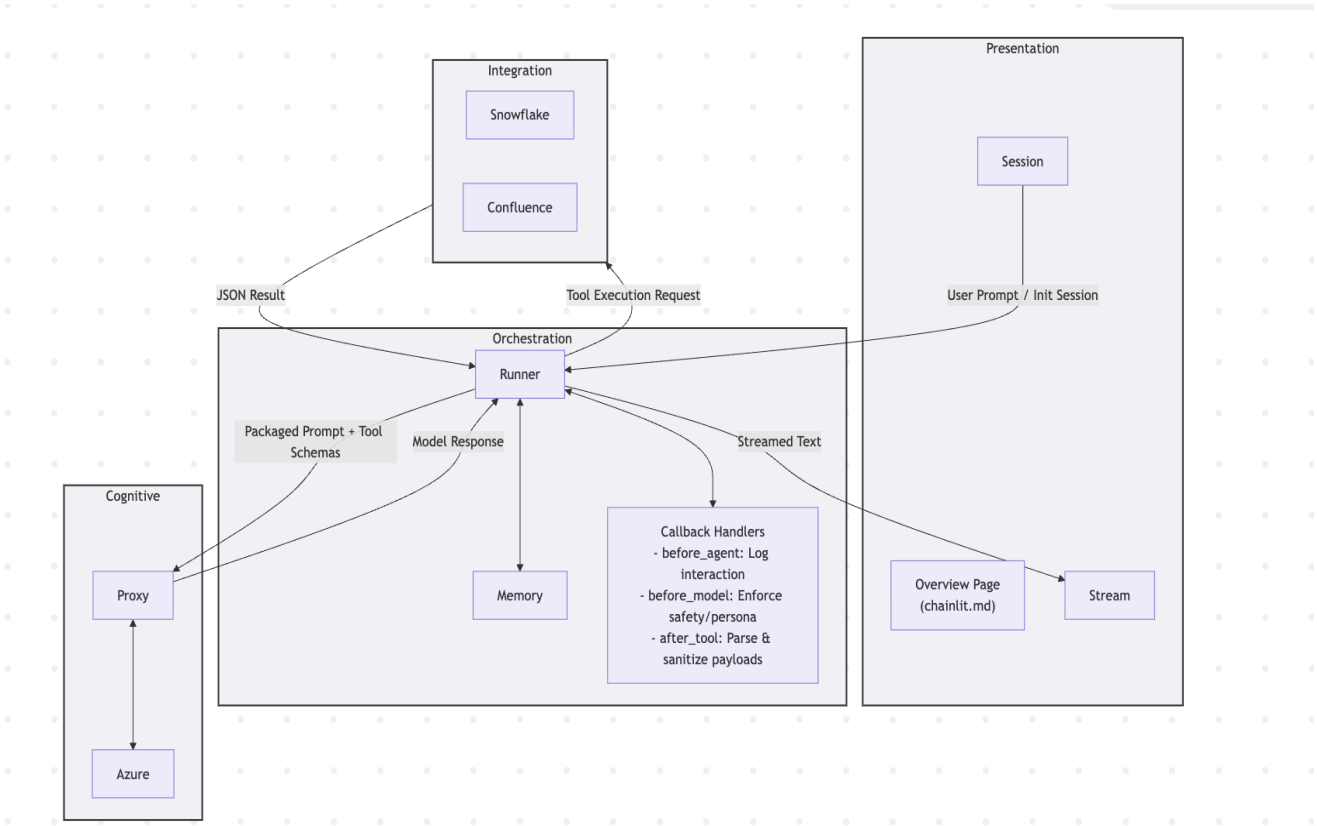
The rapid evolution of artificial intelligence has precipitated a paradigm shift in how specialized teams, such as data science units, retrieve information, enforce best practices, and interact with complex metadata. For a data science organization—specifically referencing the operational needs of a modern enterprise—the centralization of Machine Learning Operations (MLOps) guidelines, deployment protocols, and model metadata is paramount for operational efficiency. When institutional knowledge is siloed across static Confluence pages and discrete Snowflake tables, the velocity of model development is inherently bottlenecked by manual information retrieval and repetitive inquiries directed at senior personnel.

The deployment of an autonomous, highly interactive agentic system acts as a force multiplier, effectively cloning the domain expertise of lead architects. This system must dynamically ingest continuous updates to Confluence documentation and provide real-time responses regarding feature registries and machine learning metadata stored in Snowflake. This analysis presents an exhaustive, expert-level blueprint for constructing a resilient, multi-session chat application. The architecture utilizes Python's Chainlit for the frontend interface, Google's Agent Development Kit (ADK) for cognitive orchestration, FastMCP for standardized tool integration, and Azure OpenAI's GPT-5.2 model interfaced through LiteLLM. The resulting system is fully containerized via Docker and orchestrated through Kubernetes, establishing a production-grade, highly available MLOps assistant.

Architectural Blueprint and Agentic Diagram

Designing a robust agentic system requires decoupling the presentation, orchestration, cognitive, and integration layers. This modularity ensures that the underlying Large Language Model (LLM) or data sources can be swapped or scaled without rewriting the core business logic. The architecture relies heavily on the Model Context Protocol (MCP) to provide a standardized, open-source connection between the AI agent and external data systems.[1]

The following structural mapping outlines the data flow, memory management, tool routing, and callback execution within the proposed system. This diagrammatic representation serves as the foundational blueprint for reproduction and deployment.



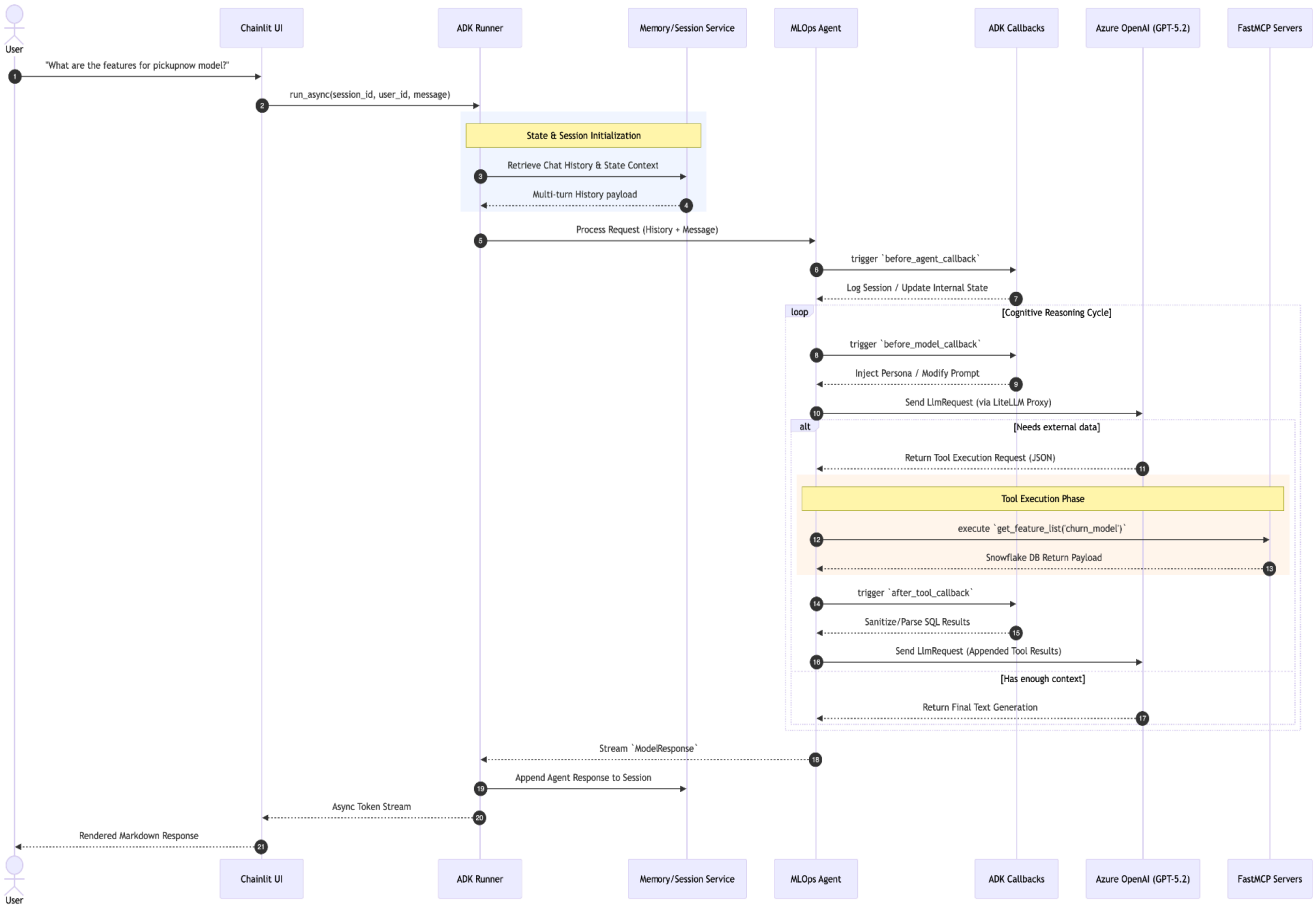
Code snippet

```
flowchart TD
  subgraph Presentation
    direction LR
    UI["Overview Page<br/>(chainlit.md)"]
    Session
    Stream
  end

  subgraph Orchestration
    direction TB
    Runner
    Memory
    Callbacks["Callback Handlers<br/>- before_agent: Log interaction<br/>- before_model: Enforce safety/persona<br/>- after_tool: Parse & sanitize payloads"]
    Runner <--> Memory
    Runner <--> Callbacks
  end
```

```
  subgraph Cognitive
    direction LR
    Proxy
    Azure
    Proxy <--> Azure
  end

  subgraph Integration
    direction LR
    Snowflake
    Confluence
  end

  %% Data Flow Links
  Session -- "User Prompt / Init Session" --> Runner
  Runner -- "Packaged Prompt + Tool Schemas" --> Proxy
  Proxy -- "Model Response" --> Runner
  Runner -- "Streamed Text" --> Stream
  Runner -- "Tool Execution Request" --> Integration
  Integration -- "JSON Result" --> Runner

  classDef layer fill:#f4f4f9,stroke:#333,stroke-width:2px;
  class Presentation,Orchestration,Cognitive,Integration layer;
```

Detailed Agent, Tools, and Memory Interaction Sequence

To fully understand how the underlying state, session memory, and callbacks interact with the external tools, the following sequence diagram maps the complete lifecycle of a single user query. It highlights how ADK intercepts logic at various stages (callbacks) and how FastMCP tools seamlessly feed structured context back into the agent's reasoning loop.



Code snippet

```
sequenceDiagram
    autonumber
    actor User
    participant UI as Chainlit UI
    participant Runner as ADK Runner
    participant Mem as Memory/Session Service
    participant Agent as MLOps Agent
    participant CB as ADK Callbacks
    participant LLM as Azure OpenAI (GPT-5.2)
    participant Tools as FastMCP Servers

    User->>UI: "What are the features for the churn model?"
    UI->>Runner: run_async(session_id, user_id, message)
```

```
  rect rgb(240, 248, 255)
    Note over Runner, Mem: State & Session Initialization
    Runner->>Mem: Retrieve Chat History & State Context
    Mem-->>Runner: Multi-turn History payload
  end

  Runner->>Agent: Process Request (History + Message)

  Agent->>CB: trigger `before_agent_callback`
  CB-->>Agent: Log Session / Update Internal State

  loop Cognitive Reasoning Cycle
    Agent->>CB: trigger `before_model_callback`
    CB-->>Agent: Inject Persona / Modify Prompt

    Agent->>LLM: Send LlmRequest (via LiteLLM Proxy)

    alt Needs external data
      LLM-->>Agent: Return Tool Execution Request (JSON)

      rect rgb(255, 245, 238)
        Note over Agent, Tools: Tool Execution Phase
        Agent->>Tools: execute `get_feature_list('churn_model')`
        Tools-->>Agent: Snowflake DB Return Payload
      end

      Agent->>CB: trigger `after_tool_callback`
      CB-->>Agent: Sanitize/Parse SQL Results

      Agent->>LLM: Send LlmRequest (Appended Tool Results)
    else Has enough context
      LLM-->>Agent: Return Final Text Generation
    end
  end

  Agent-->>Runner: Stream `ModelResponse`
  Runner->>Mem: Append Agent Response to Session
  Runner-->>UI: Async Token Stream
  UI-->>User: Rendered Markdown Response
```

Agentic Flow and Trajectory

When a data scientist queries the system (e.g., "What are the required features for the churn

prediction model, and what is our standard deployment protocol for it?"), the trajectory unfolds systematically. The Chainlit interface captures the input and routes it to the ADK Runner.[2] The ADK Agent retrieves the multi-turn session history via the memory service and triggers the before_agent_callback to securely log the interaction.[3]

The agent packages the user input, system instructions, and FastMCP tool schemas, sending them via LiteLLM to Azure's GPT-5.2.[4] The model recognizes the necessity for external enterprise data and halts text generation, returning a structured tool call request. The ADK framework intercepts this object, triggering the appropriate FastMCP endpoints. It simultaneously fetches feature metadata from the Snowflake instance and deployment protocols from the Confluence workspace.[5] The external tools return strictly typed JSON payloads, which the ADK agent feeds back into the LLM context window. Finally, the synthesized, context-aware response is streamed asynchronously back to the Chainlit frontend, providing a seamless, highly contextual answer.

Foundation: LLM Integration via LiteLLM and Azure OpenAI

The Google Agent Development Kit is intrinsically model-agnostic. While it is heavily optimized for the Google ecosystem (specifically Gemini models via Vertex AI), its architecture explicitly supports third-party providers through dedicated wrapper classes.[7] Integrating an advanced enterprise model like Azure OpenAI's GPT-5.2 requires the implementation of the LiteLlm model connector. This connector acts as an abstraction and translation layer, conforming Azure's proprietary REST API structure to ADK's expected internal data schemas.[8]

Configuration and Environment Management

To successfully route ADK requests to a secured, Azure-hosted GPT-5.2 deployment, the system must utilize LiteLLM's explicit routing capabilities. This requires defining specific environment variables at the operating system or container level to authenticate and direct the payload accurately.

Environment Variable	Required Format / Description
AZURE_API_KEY	The secure, cryptographic access token generated for the specific Azure OpenAI resource.[9]
AZURE_API_BASE	The base URL endpoint assigned to the Azure instance (e.g.,

	https://enterprise-ds-resource.openai.azure.com).[9]
AZURE_API_VERSION	The specific API version identifier required by Azure for compatibility (e.g., 2024-02-15-preview).[9]

The ADK agent definition initializes the model by passing an instantiated LiteLlm object. The model string parameter must follow the specific LiteLLM convention designed for Azure deployments: azure/<deployment-name>.[9]

Tool Call Propagation Insights

A critical technical nuance when bridging ADK, LiteLLM, and a highly sophisticated reasoning model like GPT-5.2 is the precise handling of function calls. If an agent determines it needs to execute a tool, the LLM responds not with conversational text, but with a strictly formatted structured execution request. In complex multi-framework integrations, standard text-completion wrappers occasionally fail to propagate the underlying tool_calls array back into the ADK framework's standardized ModelResponse object.[4]

If the agent consistently fails to trigger the FastMCP tools despite explicit prompting, the integration layer must ensure that the LiteLlm wrapper is correctly mapping the structured output. ADK relies on the exact formatting of the tool call payload to halt the standard reasoning loop and transfer execution authority down to the local Python functions.[4] Because GPT-5.2 possesses advanced, multi-step planning capabilities, ensuring absolute fidelity in the schema translation between ADK's internal definitions and Azure's API expectations is non-negotiable for achieving autonomous behavior.

Building the Integration Layer: FastMCP Tool Implementation

The Model Context Protocol (MCP) revolutionizes agentic tool utilization by establishing a standardized, open-source protocol for how LLMs interface with external enterprise data sources.[10] Rather than writing bespoke, fragile API connectors for every new data source, developers utilize FastMCP—a highly Pythonic framework that employs simple decorator patterns to convert standard Python functions into discoverable, highly structured tools complete with JSON schemas and automatic input validation.[11]

This approach is highly strategic for robust enterprise environments. By isolating tool logic into standalone FastMCP servers, the MLOps team ensures that the data retrieval mechanisms are completely decoupled from the ADK agent logic. If the orchestration framework is updated or changed in the future, the FastMCP servers remain fully operational and natively compatible

with any MCP-compliant client.[12]

The Snowflake Data Connector

For the enterprise data science team, real-time programmatic access to model metadata and feature engineering registries is critical. The Snowflake MCP implementation utilizes the official Snowflake Python Connector to securely execute parameterized queries against the designated MLOps database.[13]

The FastMCP server for Snowflake exposes distinct tools tailored to the required operational tables. To adhere to best practices and ensure reproducibility, the tool logic is compartmentalized.

Python

```python
# mcp_snowflake.py
import os
from fastmcp import FastMCP
import snowflake.connector

# Initialize the MCP Server for Snowflake operations
mcp_snowflake = FastMCP(name="Snowflake_ML_Metadata_Server")

def get_snowflake_connection():
    """Establishes a secure connection using environment variables."""
    return snowflake.connector.connect(
        user=os.environ.get("SNOWFLAKE_USER"),
        password=os.environ.get("SNOWFLAKE_PASSWORD"),
        account=os.environ.get("SNOWFLAKE_ACCOUNT"),
        warehouse=os.environ.get("SNOWFLAKE_WAREHOUSE"),
        database=os.environ.get("SNOWFLAKE_DATABASE"),
        schema=os.environ.get("SNOWFLAKE_SCHEMA")
    )

@mcp_snowflake.tool
def get_ml_metadata(model_name: str) -> str:
    """
    Retrieves execution metadata, performance metrics, and deployment status
    for a specific machine learning model from the ML_METADATA table.

    Args:
```

```
    model_name: The exact string identifier of the model (e.g., 'churn_v2').
    """
    conn = get_snowflake_connection()
    try:
        cursor = conn.cursor()
        # Parameterized query to prevent SQL injection vulnerabilities
        query = "SELECT METADATA_JSON FROM ML_METADATA WHERE MODEL_NAME = %s LIMIT 1"
        cursor.execute(query, (model_name,))
        result = cursor.fetchone()

        if result:
            return f"Metadata for {model_name}: {result}"
        return f"No metadata found in Snowflake for model: {model_name}"
    finally:
        conn.close()

@mcp_snowflake.tool
def get_feature_list(model_name: str) -> str:
    """
    Retrieves the authoritative list of active features utilized by a specific model
    from the FEATURE_LIST table.
    """
    conn = get_snowflake_connection()
    try:
        cursor = conn.cursor()
        query = "SELECT FEATURE_COLUMNS FROM FEATURE_LIST WHERE MODEL_NAME = %s LIMIT 1"
        cursor.execute(query, (model_name,))
        result = cursor.fetchone()

        if result:
            return f"Active features for {model_name}: {result}"
        return f"No feature list found in Snowflake for model: {model_name}"
    finally:
        conn.close()

if __name__ == "__main__":
    # Runs the server using standard input/output for local ADK consumption
    mcp_snowflake.run()
```

The underlying implementation relies on establishing a secure connection pattern. The Python function enforces strict type hinting (model_name: str), which FastMCP automatically parses into an operational JSON schema for GPT-5.2 to interpret.[14] The framework validates all parameters sent by GPT-5.2 before the SQL execution occurs, preventing runtime compilation

failures and reducing unnecessary token waste.

The Confluence Knowledge Connector

To accurately replicate the lead engineer's knowledge regarding code structuring, deployment pipelines, and evolving MLOps best practices, the agent must seamlessly access the team's live Confluence documentation. The Confluence FastMCP server bridges this gap by interacting directly with the Atlassian REST API.[15]

A deeper insight into this process reveals a common pitfall in agentic document retrieval: Large Language Models often struggle to process dense, heavily formatted HTML or XML returns efficiently. Therefore, the FastMCP Python function must do more than simply execute the API call; it must sanitize and parse the page content, stripping extraneous markup and returning pure, contextually dense text to the ADK agent.[16] By returning clean, markdown-formatted text representing the Confluence page, GPT-5.2 can efficiently synthesize the deployment practices without becoming disoriented by irrelevant web data.

Python

```python
# mcp_confluence.py
import os
import requests
from bs4 import BeautifulSoup
from fastmcp import FastMCP

mcp_confluence = FastMCP(name="Confluence_MLOps_Knowledge_Server")

CONFLUENCE_DOMAIN = os.environ.get("CONFLUENCE_DOMAIN")
CONFLUENCE_EMAIL = os.environ.get("CONFLUENCE_EMAIL")
CONFLUENCE_API_TOKEN = os.environ.get("CONFLUENCE_API_TOKEN")

@mcp_confluence.tool
def search_mlops_guidelines(query: str) -> str:
    """
    Searches the internal MLOps Confluence space for documentation, deployment
    guides, and coding structure best practices based on a user query.

    Args:
        query: The search term or natural language question.
    """
    # Construct CQL (Confluence Query Language) payload
```

```
    url = f"https://{CONFLUENCE_DOMAIN}/wiki/rest/api/content/search"
    cql = f'space="MLOPS" AND text~"{query}"'

    response = requests.get(
        url,
        params={"cql": cql, "limit": 3, "expand": "body.storage"},
        auth=(CONFLUENCE_EMAIL, CONFLUENCE_API_TOKEN)
    )

    if response.status_code!= 200:
        return f"Error retrieving documentation: HTTP {response.status_code}"

    data = response.json()
    if not data.get("results"):
        return "No relevant MLOps guidelines found in Confluence for this query."

    aggregated_content =
    for result in data["results"]:
        title = result.get("title", "Untitled Page")
        raw_html = result.get("body", {}).get("storage", {}).get("value", "")

        # Sanitize HTML to pure text for LLM consumption
        soup = BeautifulSoup(raw_html, "html.parser")
        clean_text = soup.get_text(separator="\n", strip=True)

        aggregated_content.append(f"--- Document Title: {title} ---\n{clean_text[:1500]}")

    return "\n\n".join(aggregated_content)

if __name__ == "__main__":
    mcp_confluence.run()
```

Orchestrating Cognitive Flow: Google ADK Configuration

With the LLM configured and the disparate FastMCP tools exposed, the Google Agent Development Kit serves as the central nervous system. It manages the interaction lifecycle, preserves conversational memory, and dictates the overarching workflow.[17]

Defining the Agent and Memory Services

The root agent is defined using the Agent class, which binds the LiteLLM model representation,

the definitive system instructions, and the FastMCP tool endpoints.[2] The system instruction (the prompt) defines the persona and operational boundaries of the MLOps assistant. It explicitly informs the agent that it acts as the lead AI architect clone for the enterprise data science team, instructing it to leverage the Snowflake tools exclusively for metadata queries and the Confluence tools for structural and deployment inquiries.

Furthermore, a hallmark of a high-quality chat interface like Gemini is its ability to contextually remember previous statements within a single, continuous conversation thread.[18] ADK addresses this through its integrated Session Services.[19] When a new interaction begins, the system initializes a session via InMemorySessionService.create_session(), generating a unique session ID.[20] As the data scientist interacts with the bot, the ADK Runner automatically appends both user inputs and agent responses to this session store.[21] When a subsequent query is submitted, the Runner retrieves the entire session payload and prepends it to the context window sent to Azure OpenAI.[22]

Python

```python
# agent_orchestrator.py
import os
import asyncio
from google.adk.agents import Agent
from google.adk.models.lite_llm import LiteLlm
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.adk.agents.callback_context import CallbackContext
from google.adk.models import LlmRequest

# Initialize the Azure OpenAI model via LiteLLM wrapper
llm_model = LiteLlm(model="azure/gpt-5.2-deployment")

# Instantiate Session Memory
session_service = InMemorySessionService()

# --------------------------------------------------------------------------
# Callback Implementations
# --------------------------------------------------------------------------
async def log_interaction_callback(callback_context: CallbackContext):
    """Executes before the agent begins processing a new user request."""
    session_id = callback_context._invocation_context.session.id
    print(f" Initiating reasoning loop for Session: {session_id}")
```

```
async def enforce_persona_callback(callback_context: CallbackContext, llm_request: LlmRequest):
    """Intercepts the payload before it hits Azure to ensure strict compliance."""
    for content in reversed(llm_request.contents):
        if content.role == "user" and content.parts:
            # Dynamically inject situational awareness
            content.parts.text += "\n\n(System Note: Answer purely as the Lead MLOps Architect. Rely strictly on provided tool data.)"
    return None # Return None to proceed with modified payload

# ------------------------------------------------------------------------
# Agent Definition
# ------------------------------------------------------------------------
mlops_architect_agent = Agent(
    name="Enterprise_MLOps_Architect_Clone",
    model=llm_model,
    description="An autonomous agent designed to clone MLOps architectural expertise.",
    instruction="""You are the Lead MLOps Architect for the enterprise data science team.
    Your primary function is to assist data scientists with structuring code, deploying models,
    and retrieving ML metadata. You must ALWAYS use the Confluence tool for best practices
    and the Snowflake tools for feature lists and model metadata. Never hallucinate protocols.""",
    tools=["mcp_snowflake.py", "mcp_confluence.py"], # ADK will parse MCP stdio servers
    before_agent_callback=log_interaction_callback,
    before_model_callback=enforce_persona_callback
)

# Initialize the Runner to bridge the Agent with Memory
adk_runner = Runner(
    agent=mlops_architect_agent,
    app_name="MLOps_Assistant",
    session_service=session_service
)
```

Intercepting Execution with Callbacks

ADK's callback mechanism provides unprecedented control over the agent's internal execution loop. Callbacks are user-defined functions that intercept the pipeline at specific lifecycle moments—such as before a model is queried or immediately after a tool executes.[23]

For an enterprise MLOps assistant, callbacks serve several advanced architectural purposes. The before_model_callback demonstrated above allows developers to programmatically alter the LlmRequest payload before it reaches the Azure endpoint.[24] If the user is querying specific deployment metrics, a callback could seamlessly inject the current system timestamp or Kubernetes infrastructure health status into the prompt behind the scenes, ensuring the LLM's

response is grounded in absolute real-time reality. Additionally, an after_agent_callback can be utilized to trigger asynchronous background tasks that summarize the interaction and commit it to long-term storage, such as Vertex AI Memory Bank, allowing the agent to remember user preferences across entirely different operational sessions.[3]

Presentation Layer: Chainlit for Multi-Session Interaction

While ADK handles the cognitive heavy lifting, the user experience ultimately dictates the adoption rate of any internal tool. Chainlit is utilized to construct a fluid, async-first, multi-session frontend that visually and functionally mirrors commercial AI chat applications.[25]

Building the Overview Page

The technical requirements specify an overview page dictating exactly what the bot can accomplish. Chainlit supports this out-of-the-box via a chainlit.md file placed in the root directory. This document automatically renders as a rich, markdown-formatted landing dashboard when no active chat is currently engaged.[26]

Welcome to the Enterprise MLOps Architect Assistant

This autonomous agent acts as a digital clone of the Lead AI Architect, designed to assist the data science team with model deployment, metadata retrieval, and coding standards.

Core Capabilities:

- **Query ML Metadata:** Connects directly to our Snowflake ML_METADATA tables to retrieve model performance and status.
- **Extract Feature Registries:** Pulls real-time columns from the Snowflake FEATURE_LIST to ensure alignment across environments.
- **Search Deployment Guidelines:** Seamlessly queries the live MLOps Confluence space to provide the most up-to-date deployment protocols and code structuring best practices.

Select 'New Chat' to initiate a multi-turn session.

Handling the Chat Lifecycle and Real-Time Streaming

Chainlit operates on event-driven asynchronous decorators that neatly map to the ADK operational lifecycle. When a user navigates to the application and initiates a chat, the @cl.on_chat_start decorator fires.[27] This function performs three critical actions: it generates a unique UUID representing the user's isolated session [27], it instantiates the connection to the

ADK Runner, and it persists the session metadata using cl.user_session.set(). This guarantees that concurrent requests from different data scientists do not cross-contaminate memory states.[28]

Python

```
# app.py
import uuid
import chainlit as cl
from google.genai import types
from agent_orchestrator import adk_runner, session_service

@cl.on_chat_start
async def start_chat():
    """Initializes a new multi-turn session mirroring Gemini's UX."""
    # Generate a unique session identifier for the new chat thread
    session_id = str(uuid.uuid4())
    user_id = "enterprise_ds_user" # In production, derive from Chainlit Auth

    # Register the session with ADK's Memory Service
    session_service.create_session(
        app_name="MLOps_Assistant",
        user_id=user_id,
        session_id=session_id
    )

    # Store identifiers in the localized Chainlit user session
    cl.user_session.set("session_id", session_id)
    cl.user_session.set("user_id", user_id)

    await cl.Message(
        content="System initialized. I am connected to Snowflake and Confluence. How can I assist with your MLOps workflow today?"
    ).send()

@cl.on_message
async def main_loop(message: cl.Message):
    """Intercepts user queries and streams the ADK/LiteLLM response."""
    session_id = cl.user_session.get("session_id")
    user_id = cl.user_session.get("user_id")
```

```
    # Create an empty Chainlit message object to handle the streaming response
    ui_message = cl.Message(content="")
    await ui_message.send()

    # Format the input for ADK consumption
    adk_content = types.Content(role='user', parts=[types.Part(text=message.content)])

    # Execute the ADK Runner asynchronously
    async for event in adk_runner.run_async(
        user_id=user_id,
        session_id=session_id,
        new_message=adk_content
    ):
        # Filter for token generation events to stream to the UI
        if event.is_model_response() and event.content and event.content.parts:
            token = event.content.parts.text
            # Dynamically update the UI to prevent latent application states
            await ui_message.stream_token(token)

    await ui_message.update()
```

When a user submits a prompt, the @cl.on_message decorator intercepts the text.[29] The application invokes the ADK runner.run_async() method. Because LLM generation is inherently latent (especially when executing multiple external database queries mid-generation), Chainlit utilizes real-time streaming. As the run_async generator yields tokens from Azure OpenAI, Chainlit's ui_message.stream_token() dynamically updates the UI, preventing the application from appearing unresponsive and mimicking the immediate feedback loop of premium commercial tools.[30]

Robustness and Reliability: ADK Evaluation Framework

Deploying an autonomous agent into an enterprise environment without rigorous, programmatic testing is a severe operational risk. Because LLMs are inherently non-deterministic, traditional unit testing (asserting that input X always exactly equals output Y) is entirely insufficient. To mitigate this, the Google ADK includes a sophisticated, purpose-built evaluation framework designed to simultaneously score both the final linguistic output and the internal logical trajectory of the agent.[31]

The Philosophy of Agentic Evaluation

Agent evaluation operates on a three-tier pyramid: component-level unit tests, trajectory-level integration tests, and end-to-end human reviews.[32] The ADK evaluation engine primarily addresses the critical second tier.

When evaluating a multi-tool agent, evaluating only the final response is dangerous. An LLM might generate a factually correct answer by relying on its internal pre-training data (a "lucky guess") while completely failing to execute the required FastMCP tool to query Snowflake. If the underlying data in Snowflake subsequently changes the following week, the agent will confidently output incorrect, hallucinated data.[33] Therefore, the evaluation framework must assess the *process*—verifying that the agent selected the correct Confluence tool, passed the correct CQL arguments, and interpreted the return payload accurately.[34]

Establishing the Golden Dataset

The foundation of the evaluation is the Golden Dataset.[31] This is a curated JSON collection of perfect, idealized interactions. Using the ADK Web UI development server (adk web), developers can manually interact with the agent locally, testing specific edge cases (e.g., asking for a feature list for a model that does not exist to ensure it handles the error gracefully). Once the agent correctly navigates the logic and produces the desired trace, the session is exported and saved as a .evalset.json file.[31] This file serves as the definitive answer key against which future iterations of the agent will be graded.

JSON

```
// eval_dataset.json (Golden Dataset Snapshot)
{
 "scenarios": [
  {
   "user_input": "Get the features for churn_v2",
   "expected_tool_calls": ["get_feature_list"],
   "expected_tool_arguments": {"model_name": "churn_v2"},
   "expected_response_context": "The agent must list the features exactly as returned by the Snowflake DB."
  }
 ]
}
```

Configuring Evaluation Metrics

ADK evaluations are driven by a test_config.json file that defines the acceptable mathematical

thresholds for passing the test suite.[31] The framework employs an "LLM-as-a-judge" paradigm, utilizing a highly capable secondary model to qualitatively score the agent's performance across various dimensions.[35]

Relevant evaluation metrics critical for the MLOps assistant include:

Evaluation Metric	Primary Purpose	Application Context
tool_trajectory_avg_score	Verifies exact matching of the tool call sequence.[35]	Ensures the agent consistently calls the FastMCP Snowflake tool before attempting to answer metadata queries, rather than guessing.
final_response_match_v2	Judges semantic similarity between the generated response and the golden dataset.[35]	Prevents test failures if the LLM phrases the deployment guidelines slightly differently than the reference answer, provided the semantic meaning remains identical.
hallucinations_v1	Assesses groundedness against context.[35]	Critical for ensuring the agent does not invent MLOps protocols; every procedural claim must trace back to the Confluence return payload.

For advanced Continuous Integration (CI/CD) pipelines, testing static conversation paths is inadequate because real users hold dynamic, multi-turn conversations. ADK addresses this vulnerability through "User Simulation".[36] Instead of hardcoding static user inputs, developers define a ConversationScenario (e.g., "Ask the agent for the churn model metadata, and then ask it to format the feature list as a table"). An automated LLM acts as the user, dynamically generating successive prompts to achieve this goal, thereby stress-testing the agent's ability to maintain context over long session histories.[36] These evaluations can be run systematically via the adk eval CLI command or integrated directly into Pytest workflows.[35]

Infrastructure: Containerization and Kubernetes

Deployment

To transition the application from a local development environment to a resilient, highly available production state for the team, the entire software stack must be structured properly, containerized via Docker, and orchestrated via Kubernetes (K8s).

Repository and Project Structure

A clean, modular repository ensures that code orchestration, frontend configuration, and tool logic are naturally isolated. The optimal directory layout for this architecture is as follows:

mlops-assistant/

├── app.py # Chainlit frontend and UI routing

├── agent_orchestrator.py # ADK Agent, Runner, Callbacks, and Session definitions

├── mcp_snowflake.py # FastMCP server for Snowflake metadata/features

├── mcp_confluence.py # FastMCP server for Confluence guidelines

├── chainlit.md # Chainlit overview landing page

├──.env # Environment variables (not in version control)

├── requirements.txt # Python dependencies

├── Dockerfile # Containerization instructions

├── kubernetes/ # K8s manifests

│ ├── deployment.yaml

│ └── service.yaml

└── tests/ # ADK Evaluation Framework

├── eval_dataset.json # Golden dataset for evaluation

└── test_config.json # ADK test configuration thresholds

Requirements (requirements.txt)

To successfully orchestrate the disparate frameworks, the application requires specific dependencies handling the frontend (Chainlit), cognitive orchestration (ADK and LiteLLM), tooling (FastMCP), and API clients (Snowflake and HTTP parsers).

chainlit==1.1.4

google-adk==1.25.0

fastmcp==0.2.0

litellm==1.44.0

snowflake-connector-python==3.12.0

beautifulsoup4==4.12.3

requests==2.32.3

Docker Image Construction

The application comprises multiple moving parts: the Chainlit web server, the ADK orchestration logic, and the FastMCP tool endpoints. The deployment strategy requires an optimized Dockerfile that encapsulates all dependencies.

Dockerfile

```
# Dockerfile
FROM python:3.11-slim

# Set environment variables to prevent Python from buffering stdout/stderr
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

WORKDIR /app

# Install system dependencies required for compilation
RUN apt-get update && apt-get install -y gcc build-essential && rm -rf /var/lib/apt/lists/*

# Install specific Python framework dependencies
COPY requirements.txt.
RUN pip install --no-cache-dir -r requirements.txt

# Copy application source code
COPY. /app

# Expose the Chainlit default port
```

```
EXPOSE 8000

# Execute Chainlit in headless mode bound to all network interfaces
CMD ["chainlit", "run", "app.py", "-h", "--host", "0.0.0.0", "--port", "8000"]
```

A critical deployment detail involves Chainlit's execution command. In a local environment, running chainlit run app.py often automatically attempts to open a browser window. In a Dockerized, headless production environment, the command must execute with the -h flag to prevent server-side browser execution, and bind to 0.0.0.0 to ensure it listens to the correct container network interfaces.[37]

Kubernetes Orchestration Strategy

Deploying to a cluster requires the creation of structured Kubernetes manifests defining Deployments, Services, and Secrets.[38]

1. **Secret Management:** Hardcoding sensitive variables like AZURE_API_KEY, SNOWFLAKE_PASSWORD, or Atlassian API tokens within the codebase is a severe security vulnerability. These must be provisioned as Kubernetes Secrets and securely injected into the container at runtime as environment variables, allowing LiteLLM and FastMCP to authenticate securely.[39]
2. **Deployment Configuration:** The K8s Deployment YAML defines the desired state, specifying resource requests and limits to ensure the Python application has sufficient memory to handle concurrent ADK sessions, heavy JSON payloads returned from Snowflake, and intensive string parsing from Confluence HTML.
3. **Service and Ingress (WebSocket Affinity):** A Kubernetes Service exposes the application pods to network traffic. Because Chainlit relies heavily on WebSockets for real-time text streaming, the deployment infrastructure (whether an Ingress controller or an external Load Balancer) must be explicitly configured to support WebSocket HTTP upgrades.

Furthermore, if the architecture scales horizontally to multiple pods to support heavy usage by the data science team, configuring "Sticky Sessions" (Session Affinity) on the Load Balancer is absolutely vital. Without sticky sessions, sequential WebSocket packets from a single user's chat session might be routed to different container instances, instantly breaking the connection and destroying the real-time chat experience.[37] To mitigate internal load balancer inconsistencies, the Chainlit .chainlit/config.toml file should strictly enforce WebSocket transport by defining transports = ["websocket"].[37]

YAML

```yaml
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mlops-assistant-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: mlops-assistant
  template:
    metadata:
      labels:
        app: mlops-assistant
    spec:
      containers:
      - name: mlops-assistant
        image: your-registry/mlops-assistant:latest
        ports:
        - containerPort: 8000
        envFrom:
        - secretRef:
            name: mlops-assistant-secrets # Contains Azure, Snowflake, Atlassian keys
---
apiVersion: v1
kind: Service
metadata:
  name: mlops-assistant-service
  annotations:
    # Ensure the ingress/load balancer is configured for WebSocket affinity
    # e.g., ingress.kubernetes.io/affinity: "cookie"
spec:
  selector:
    app: mlops-assistant
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8000
  type: LoadBalancer
```

By structurally integrating the Google Agent Development Kit, the system gains a resilient cognitive loop, robust multi-turn session memory, and precise callback control. Routing this logic through LiteLLM ensures seamless compatibility with advanced models like Azure's

GPT-5.2, while the implementation of FastMCP normalizes the interface with critical business data in Snowflake and Confluence. When coupled with the fluid user interface provided by Chainlit, and hardened by ADK's rigorous evaluation frameworks and Kubernetes orchestration, the resulting architecture delivers an autonomous, scalable knowledge assistant capable of drastically optimizing the workflow of the data science unit.

Works cited

1. Snowflake-managed MCP server, accessed February 23, 2026, https://docs.snowflake.com/en/user-guide/snowflake-cortex/cortex-agents-mcp
2. Develop an Agent Development Kit agent | Vertex AI Agent Builder | Google Cloud Documentation, accessed February 23, 2026, https://docs.cloud.google.com/agent-builder/agent-engine/develop/adk
3. Memory - Agent Development Kit (ADK) - Google, accessed February 23, 2026, https://google.github.io/adk-docs/sessions/memory/
4. Implementing a custom LiteLlm for Google agent development kit that supports tool calls, accessed February 23, 2026, https://stackoverflow.com/questions/79767829/implementing-a-custom-litellm-for-google-agent-development-kit-that-supports-too
5. Building a Snowflake MCP Server. A Comprehensive Guide | by Vikrambalauae Aj - Medium, accessed February 23, 2026, https://medium.com/@vikrambalaaj/building-a-snowflake-mcp-server-9aa9eb27744d
6. Building Better AI Integrations with Model Context Protocol: A Confluence Case Study, accessed February 23, 2026, https://blog.gopenai.com/building-better-ai-integrations-with-model-context-protocol-a-confluence-case-study-ef502364369e
7. AI Models for ADK agents - Agent Development Kit (ADK) - Google, accessed February 23, 2026, https://google.github.io/adk-docs/agents/models/
8. How to Use Third-Party LLMs with Google Agent Development Kit (ADK): A Dark Joke Agent Tutorial | by Ravindu Guruge | Medium, accessed February 23, 2026, https://ravinduguruge.medium.com/how-to-use-third-party-llms-with-google-agent-development-kit-adk-a-dark-joke-agent-tutorial-31241f42f1c4
9. Azure OpenAI - LiteLLM, accessed February 23, 2026, https://docs.litellm.ai/docs/providers/azure/
10. Creating Your First MCP Server: A Hello World Guide | by Gianpiero Andrenacci | AI Bistrot, accessed February 23, 2026, https://medium.com/data-bistrot/creating-your-first-mcp-server-a-hello-world-guide-96ac93db363e
11. Build an MCP Server with Python in 20 Minutes, accessed February 23, 2026, https://www.youtube.com/watch?v=Ywy9x8gM410
12. Getting Started with Managed Snowflake MCP Server, accessed February 23, 2026, https://www.snowflake.com/en/developers/guides/getting-started-with-snowflake-mcp-server/

13. MCP Server for Snowflake including Cortex AI, object management, SQL orchestration, semantic view consumption, and more - GitHub, accessed February 23, 2026, https://github.com/Snowflake-Labs/mcp
14. Tools - FastMCP, accessed February 23, 2026, https://gofastmcp.com/servers/tools
15. MahithChigurupati/Confluence-MCP-Server - GitHub, accessed February 23, 2026, https://github.com/MahithChigurupati/Confluence-MCP-Server
16. MCP tool for vector search in Confluence, Jira and local files | by Oleksii Shnepov | Medium, accessed February 23, 2026, https://medium.com/@shnax0210/mcp-tool-for-vector-search-in-confluence-and-jira-6beeade658ba
17. google/adk-python: An open-source, code-first Python toolkit for building, evaluating, and deploying sophisticated AI agents with flexibility and control. - GitHub, accessed February 23, 2026, https://github.com/google/adk-python
18. Manage sessions and history | Gemini CLI, accessed February 23, 2026, https://geminicli.com/docs/cli/tutorials/session-management/
19. Agent Development Kit documentation - Google, accessed February 23, 2026, https://google.github.io/adk-docs/api-reference/python/
20. Google's Agent Development Kit (ADK): A Guide With Demo Project - DataCamp, accessed February 23, 2026, https://www.datacamp.com/tutorial/agent-development-kit-adk
21. BerriAI/litellm: Python SDK, Proxy Server (AI Gateway) to call 100+ LLM APIs in OpenAI (or native) format, with cost tracking, guardrails, loadbalancing and logging. [Bedrock, Azure, OpenAI, VertexAI, Cohere, Anthropic, Sagemaker, HuggingFace, VLLM, NVIDIA NIM] - GitHub, accessed February 23, 2026, https://github.com/BerriAI/litellm
22. How to Build Persistent Memory Agents in ADK (Step-by-Step Demo) | Vertex AI memory Bank, accessed February 23, 2026, https://m.youtube.com/watch?v=5onUg-YJBZA
23. Callbacks: Observe, Customize, and Control Agent Behavior - Agent Development Kit (ADK), accessed February 23, 2026, https://google.github.io/adk-docs/callbacks/
24. Master ADK Callbacks: DOs and DON'Ts | by minherz | Google Cloud - Community | Feb, 2026, accessed February 23, 2026, https://medium.com/google-cloud/master-adk-callbacks-dos-and-donts-adedd2386983
25. Building a Simple Chatbot with LangGraph and Chainlit: A Step-by-Step Tutorial, accessed February 23, 2026, https://dev.to/jamesbmour/building-a-simple-chatbot-with-langgraph-and-chainlit-a-step-by-step-tutorial-4k6h
26. Welcome page is not shown on the main page anymore. · Issue #1073 · Chainlit/chainlit · GitHub, accessed February 23, 2026, https://github.com/Chainlit/chainlit/issues/1073
27. My First Time Using Chainlit: Building a Simple and Interactive UI for my ChatBot | by Andres Felipe Tellez Yepes | Medium, accessed February 23, 2026,

https://medium.com/@andres.tellez/my-first-time-using-chainlit-building-a-simple-and-interactive-ui-for-my-chatbot-e69efa139655
28. User Session - Chainlit, accessed February 23, 2026, https://docs.chainlit.io/concepts/user-session
29. Chainlit: A Guide With Practical Examples - DataCamp, accessed February 23, 2026, https://www.datacamp.com/tutorial/chainlit
30. How to get chat history working with Chainlit? : r/LangChain - Reddit, accessed February 23, 2026, https://www.reddit.com/r/LangChain/comments/1g01njm/how_to_get_chat_history_working_with_chainlit/
31. Evaluating Agents with ADK - Google Codelabs, accessed February 23, 2026, https://codelabs.developers.google.com/adk-eval/instructions
32. Agent Factory Recap: A Deep Dive into Agent Evaluation, Practical Tooling, and Multi-Agent Systems | Google Cloud Blog, accessed February 23, 2026, https://cloud.google.com/blog/topics/developers-practitioners/agent-factory-recap-a-deep-dive-into-agent-evaluation-practical-tooling-and-multi-agent-systems
33. A methodical approach to agent evaluation | Google Cloud Blog, accessed February 23, 2026, https://cloud.google.com/blog/topics/developers-practitioners/a-methodical-approach-to-agent-evaluation
34. Getting Started with Cortex Agent Evaluations - Snowflake, accessed February 23, 2026, https://www.snowflake.com/en/developers/guides/getting-started-with-cortex-agent-evaluations/
35. Why Evaluate Agents - Agent Development Kit (ADK) - Google, accessed February 23, 2026, https://google.github.io/adk-docs/evaluate/
36. Announcing User Simulation in ADK Evaluation - Google Developers Blog, accessed February 23, 2026, https://developers.googleblog.com/announcing-user-simulation-in-adk-evaluation/
37. Overview - Chainlit, accessed February 23, 2026, https://docs.chainlit.io/deploy/overview
38. Deploy ADK agents to Google Kubernetes Engine (GKE), accessed February 23, 2026, https://codelabs.developers.google.com/codelabs/production-ready-ai-with-gc/5-deploying-agents/deploy-adk-agents-to-gke
39. mcp-jira-confluence - PyPI, accessed February 23, 2026, https://pypi.org/project/mcp-jira-confluence/