Deep Dive: Integrating AI/LLMs/MLOps 2

This document provides a practical, interactive guide to integrating AI and Large Language Models (LLMs) into your data platform, all within the framework of Machine Learning Operations (MLOps). Building on the conceptual overview, we'll demonstrate concrete examples using your local environment, focusing on real-world scenarios like RAG (Retrieval Augmented Generation) data preparation and basic LLM interaction.

1. Prerequisites

Before starting, ensure your **Advanced Track** local environment is fully operational:

- All services from docker-compose.yml are up and healthy (docker compose up --build -d).
- You have access to the UI for FastAPI (http://localhost:8000/docs), Spark History Server (http://localhost:18080), MinIO (http://localhost:9001), and Grafana (http://localhost:3000).
- A **Gemini API Key** is required for the LLM interaction example. You can obtain one from Google AI Studio. Set it as an environment variable in your docker-compose.yml for the fastapi_ingestor service:
  ```
  # ... inside fastapi_ingestor service definition ...
  environment:
    KAFKA_BROKER: kafka:29092
    KAFKA_TOPIC_FINANCIAL: raw_financial_transactions
    KAFKA_TOPIC_INSURANCE: raw_insurance_claims
    LLM_API_KEY: your_gemini_api_key_here # <--- ADD THIS LINE
  # ...
  ```

  Remember to replace your_gemini_api_key_here with your actual key. After modifying, you'll need to docker compose up -d --no-deps --build fastapi_ingestor to apply the change.

2. Interactive How-Tos

How-To 1: Preparing a Knowledge Base for RAG using Spark

Scenario: You have a collection of internal documents (e.g., product manuals, customer support FAQs) in raw text format. You want to prepare these documents to serve as a knowledge base for a RAG system, allowing an LLM to answer questions using your specific enterprise data.
Goal: Process raw text documents, chunk them into smaller, manageable pieces, and store them in a Delta Lake table, ready for indexing or embedding generation.
Steps:

1. **Create Sample Raw Documents:**

- Navigate to your data/minio/raw-data-bucket/ directory.
- Create a new subdirectory, e.g., llm_raw_knowledge/.
- Inside llm_raw_knowledge/, create a file named policy_docs.json with the following content (JSON Lines format):

```
# data/minio/raw-data-bucket/llm_raw_knowledge/policy_docs.json
{"doc_id": "policy_001", "content": "Our refund policy states that customers can return items within 30 days of purchase for a full refund, provided the item is in its original condition and accompanied by a valid receipt. After 30 days, only store credit will be issued. Sale items are final sale and cannot be returned or exchanged. For online purchases, the return window begins on the day of delivery. Shipping fees are non-refundable."}
{"doc_id": "policy_002", "content": "Warranty coverage for electronic devices extends for 12 months from the date of original purchase. This warranty covers defects in materials and workmanship. It does not cover damage caused by accident, misuse, unauthorized modification, or normal wear and tear. To initiate a warranty claim, please contact our support team with your proof of purchase and a description of the issue. A repair or replacement will be provided at our discretion."}
{"doc_id": "faq_001", "content": "How do I reset my password? To reset your password, visit our login page and click on the 'Forgot Password' link. Enter your registered email address, and we will send you a password reset link. Follow the instructions in the email to create a new password. If you do not receive the email, please check your spam folder."}
{"doc_id": "faq_002", "content": "What payment methods do you accept? We accept major credit cards (Visa, MasterCard, American Express), PayPal, and Google Pay. We do not accept cash on delivery or personal checks."}
```

2. **Inspect the llm_data_prep.py Spark Script:**
   - This script, located in pyspark_jobs/, is designed to read raw text (JSON Lines), perform simple chunking (splitting by double newline or a heuristic), and write to a Delta Lake table.
   - **Open conceptual_code/pyspark_jobs/highlights/llm_data_prep.py** (from the consolidated code document). Pay attention to the df_chunked and df_final transformations, especially how chunk_id is created and text_content is extracted.
3. **Run the Spark Job to Prepare the Knowledge Base:**
   - Open a new terminal.
   - Execute the following command to submit the Spark job to process your policy_docs.json:

```
docker exec -it spark spark-submit \
  --packages io.delta:delta-core_2.12:2.4.0 \
  --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
  --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog
\
  --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
  --conf spark.hadoop.fs.s3a.access.key=minioadmin \
  --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
```

```
--conf spark.hadoop.fs.s3a.path.style.access=true \
/opt/bitnami/spark/jobs/llm_data_prep.py \
s3a://raw-data-bucket/llm_raw_knowledge/policy_docs.json \
s3a://curated-data-bucket/llm_knowledge_base
```

- Monitor the terminal for Spark job logs indicating completion.

4. **Verify the Prepared Knowledge Base in MinIO:**
   - Open your web browser and go to the MinIO Console: http://localhost:9001.
   - Log in with minioadmin/minioadmin.
   - Navigate to the curated-data-bucket.
   - You should now see a new directory: llm_knowledge_base/. Click into it.
   - Observe the .parquet files and _delta_log directory, indicating your processed chunks are stored as a Delta Lake table.
5. **Query the Prepared Knowledge Base with Spark SQL:**
   - Open another terminal.
   - Connect to Spark SQL and query the newly created Delta table:

```
docker exec -it spark spark-sql \
  --packages io.delta:delta-core_2.12:2.4.0 \
  --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
  --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog
\
  --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
  --conf spark.hadoop.fs.s3a.access.key=minioadmin \
  --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
  --conf spark.hadoop.fs.s3a.path.style.access=true \
  -e "SELECT chunk_id, source_document_id, text_content FROM
delta.\`s3a://curated-data-bucket/llm_knowledge_base\` LIMIT 10;"
```

   - **Observe:** The output shows individual chunk_ids, source_document_id (e.g., policy_001), and the text_content (the processed chunks of your original documents). This structured format is ideal for later embedding and retrieval in a RAG system.

How-To 2: Real-time LLM Interaction via FastAPI (with OpenTelemetry Monitoring)

Scenario: You want to expose a simple API endpoint that allows applications to send a natural language query, which your FastAPI service then forwards to an LLM (e.g., Gemini) to get a response. You also want to monitor the LLM interaction's performance.
Goal: Demonstrate FastAPI as an LLM gateway, and observe metrics related to LLM calls in Grafana.
Steps:

1. **Ensure FastAPI is configured with LLM_API_KEY:**
   - Double-check that you've added LLM_API_KEY: your_gemini_api_key_here to the

fastapi_ingestor service's environment in your docker-compose.yml.
    - If you just added it, run: docker compose up -d --no-deps --build fastapi_ingestor to apply the change.
    - Confirm httpx is in fastapi_app/requirements.txt and the main.py is using the main_advanced.py content (which includes the /llm-query/ endpoint and OpenTelemetry instrumentation).
2. **Access FastAPI Docs (Swagger UI):**
    - Open your browser to http://localhost:8000/docs.
    - Scroll down to the "AI/LLM" section. You should see the /llm-query/ endpoint.
    - Click "Try it out" and then "Execute" to send a sample query. You can modify the text_input and context if you wish.
3. **Send Queries to the LLM Endpoint via curl:**
    - Open a terminal and send a few queries:

```
curl -X POST "http://localhost:8000/llm-query/" \
    -H "Content-Type: application/json" \
    -d '{
        "text_input": "What is data lineage and why is it important?",
        "context": "Focus on its role in enterprise data platforms."
      }'

curl -X POST "http://localhost:8000/llm-query/" \
    -H "Content-Type: application/json" \
    -d '{
        "text_input": "Summarize the key benefits of using Apache Kafka for real-time data ingestion."
      }'

curl -X POST "http://localhost:8000/llm-query/" \
    -H "Content-Type: application/json" \
    -d '{
        "text_input": "How can I check the health of my MinIO service?",
        "context": "Assume a Docker Compose setup."
      }'
```

    - **Observe:** The responses will come back as JSON, containing generated_text from the LLM (if your API key is valid).
4. **Monitor LLM Interactions in Grafana:**
    - Open your browser to the Grafana UI: http://localhost:3000.
    - Log in with admin/admin.
    - Go to the "Explore" view (compass icon on the left).
    - Select your Prometheus data source.
    - **Query LLM Request Latency:**
      histogram_quantile(0.95, sum by(le, model_name, success) (rate(llm_request_duration_bucket[1m])))

        - This query shows the 95th percentile latency of your LLM requests, segmented by model and success status.
    - **Query LLM Request Rate:** rate(llm_request_duration_count{model_name="gemini-2.0-flash"}[1m])
        - This shows the rate of requests to the LLM API.
    - **Query LLM Error Count (if you send a bad query or API key fails):** llm_api_errors_total{model_name="gemini-2.0-flash"}
        - Try sending a query to a non-existent LLM endpoint or with an invalid API key (by temporarily removing it from docker-compose.yml and restarting fastapi_ingestor). Then query this metric.
    - **Observe:** As you send curl requests, these metrics in Grafana will update, demonstrating real-time monitoring of your LLM gateway service.

How-To 3: Orchestrating an ML/LLM Pipeline (Conceptual with Airflow)

Scenario: You want to automate a pipeline that regularly updates your RAG knowledge base and then potentially triggers a process to re-embed the new chunks or refresh an LLM cache.
Goal: Outline an Airflow DAG that orchestrates this MLOps workflow.
Steps (Conceptual Airflow DAG):

1. **Inspect an Airflow DAG for an ML/LLM pipeline:**
    - This DAG orchestrates the preparation of new data for your RAG system.
    - In your airflow_dags/ directory, create rag_pipeline_dag.py.

```
# airflow_dags/rag_pipeline_dag.py
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from airflow.utils.dates import days_ago
from datetime import timedelta
import os

# Define common Spark submit arguments
SPARK_COMMON_CONF = (
   "--packages io.delta:delta-core_2.12:2.4.0 "
   "--conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension "
   "--conf spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog "
   "--conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 "
   "--conf spark.hadoop.fs.s3a.access.key=minioadmin "
   "--conf spark.hadoop.fs.s3a.secret.key=minioadmin "
   "--conf spark.hadoop.fs.s3a.path.style.access=true "
)

def trigger_embedding_service_mock(**kwargs):
```

```
    """
    Simulates calling an external embedding service or a Lambda/FastAPI endpoint
    that would take the new chunks and generate/update embeddings.
    In a real scenario, this would be an HTTP call or Kafka message.
    """
    ti = kwargs['ti']
    # This would pass information about the newly processed Delta Lake table
    processed_data_path = kwargs['dag_run'].conf.get('processed_data_path',
's3a://curated-data-bucket/llm_knowledge_base')
    print(f"Triggering embedding service for new data at: {processed_data_path}")
    print("This would involve: 1. Loading new chunks. 2. Generating embeddings. 3. Storing in
vector database.")
    # Example: make an HTTP call to a dedicated embedding service
    # requests.post("http://embedding_service:5000/embed_documents", json={"path":
processed_data_path})

with DAG(
    dag_id='rag_knowledge_base_update_pipeline',
    start_date=days_ago(1),
    schedule_interval=timedelta(days=1), # Daily update
    catchup=False,
    tags=['mlops', 'llm', 'rag', 'data_prep'],
    default_args={
        'owner': 'airflow',
        'depends_on_past': False,
        'email_on_failure': False,
        'email_on_retry': False,
        'retries': 1,
        'retry_delay': timedelta(minutes=5),
    },
    doc_md="""
    ### RAG Knowledge Base Update Pipeline
    This DAG orchestrates the process of updating the RAG knowledge base:
    1.  **`ingest_new_raw_docs`**: Simulates ingestion of new raw documents.
    2.  **`prepare_llm_data`**: Processes raw documents into chunks in Delta Lake.
    3.  **`trigger_embedding_service`**: Conceptually triggers an external service to embed
new chunks.
    """
) as dag:
    # Task 1: Simulate Ingestion of New Raw Documents (e.g., from an SFTP or API)
    ingest_new_raw_docs = BashOperator(
        task_id='ingest_new_raw_docs',
        bash_command='echo "Simulating ingestion of new raw documents into
raw-data-bucket/llm_raw_knowledge/new_docs.json" && '
                     'echo \'{"doc_id": "policy_003", "content": "Our updated privacy policy
emphasizes data encryption."}\' > /tmp/new_doc.json && '
                     'docker exec minio mc cp /tmp/new_doc.json
local/raw-data-bucket/llm_raw_knowledge/new_doc_{{ ds_nodash }}.json && '
```

```
            'rm /tmp/new_doc.json',
        doc_md="""
        #### Ingest New Raw Documents
        Simulates new raw documents arriving in the MinIO raw bucket.
        """
    )

    # Task 2: Prepare LLM Data using Spark (chunking, cleaning)
    prepare_llm_data = BashOperator(
        task_id='prepare_llm_data',
        bash_command=f"docker exec -it spark spark-submit {SPARK_COMMON_CONF} "
                     f"/opt/bitnami/spark/jobs/llm_data_prep.py "
                     f"s3a://raw-data-bucket/llm_raw_knowledge/new_doc_{{ ds_nodash }}.json " #
Process the newly ingested file
                     f"s3a://curated-data-bucket/llm_knowledge_base", # Appending to existing
knowledge base
        doc_md="""
        #### Prepare LLM Data
        Runs the Spark job to preprocess and chunk the raw documents,
        updating the `llm_knowledge_base` Delta table.
        """
    )

    # Task 3: Trigger Embedding Service
    trigger_embedding_service = PythonOperator(
        task_id='trigger_embedding_service',
        python_callable=trigger_embedding_service_mock,
        op_kwargs={'processed_data_path': 's3a://curated-data-bucket/llm_knowledge_base'}, #
Pass path as config
        provide_context=True,
        doc_md="""
        #### Trigger Embedding Service
        Conceptually triggers an external service (e.g., a dedicated microservice
        or a cloud function) to generate embeddings for the new data chunks
        and update the vector database used by the RAG system.
        """
    )

    # Define the task dependencies
    ingest_new_raw_docs >> prepare_llm_data >> trigger_embedding_service
```

Steps to Exercise:

1. **Place the DAG file:** Save the rag_pipeline_dag.py content into your airflow_dags/ directory.
2. **Access Airflow UI:** Go to http://localhost:8080.
3. **Find and Enable the DAG:** Locate rag_knowledge_base_update_pipeline, and toggle it to "On".

4. **Trigger the DAG:** Click the "Play" button (trigger DAG).
5. **Monitor Execution:**
    - Observe the DAG in "Graph View" and "Gantt Chart View".
    - Check the logs of each task.
        - ingest_new_raw_docs: You should see it creating a new dummy JSON file in MinIO.
        - prepare_llm_data: You should see Spark processing this new dummy file and appending it to your llm_knowledge_base Delta Lake table.
        - trigger_embedding_service: Its logs will show the conceptual message about triggering the embedding service.
    - Verify in MinIO Console (http://localhost:9001) that new files appear in llm_knowledge_base/ for the new doc_id.

Verification:

- **Airflow UI:** The rag_knowledge_base_update_pipeline DAG successfully executes all tasks, demonstrating automated data preparation and conceptual triggering of downstream ML/LLM services.
- **MinIO:** New parquet files are added to the llm_knowledge_base Delta table, confirming that the new raw document was processed and integrated.
- **Logs:** The task logs confirm that each step of the MLOps pipeline (simulated ingestion, Spark processing, and conceptual embedding trigger) was executed.

This concludes the deep dive into Integrating AI/LLMs/MLOps.