Highlighting Apache Airflow: Workflow Orchestration

Apache Airflow serves as the central nervous system of your enterprise data platform, enabling you to programmatically author, schedule, and monitor complex data pipelines. It transforms disparate tasks across various systems into managed workflows, crucial for ensuring data freshness and operational efficiency.
This guide will demonstrate basic and advanced use cases of Apache Airflow, leveraging your **Advanced Track** local environment setup.
Reference: This guide builds upon the concepts and setup described in **Section 4.2. Core Technology Deep Dive** of the **Core Handbook** and the **Progressive Path Setup Guide Deep-Dive Addendum**.

Basic Use Case: Scheduling a Batch ETL Spark Job

Objective: To demonstrate how Airflow can schedule and trigger a simple batch ETL (Extract, Transform, Load) Spark job that processes data from the raw data zone in MinIO and writes to a curated zone.
Role in Platform: Automate routine data processing tasks.
Setup/Configuration (Local Environment - Advanced Track):

1. **Ensure all Advanced Track services are running:** docker compose up --build -d from your project root.
2. **Verify Airflow is accessible:** Go to http://localhost:8080 and log in with admin/admin.
3. **Prepare a simple Spark job:** You should have a conceptual pyspark_jobs/batch_transformations.py script that reads from a source Delta Lake path in MinIO and writes to a target curated Delta Lake path.
4. **Create a simple Airflow DAG:** In your airflow_dags/ directory, create a DAG file (e.g., simple_batch_etl_dag.py). This DAG will use a BashOperator to execute a spark-submit command within the spark container.
   Example airflow_dags/simple_batch_etl_dag.py (conceptual):
   ```
   from airflow import DAG
   from airflow.operators.bash import BashOperator
   from datetime import datetime, timedelta

   with DAG(
       dag_id='simple_batch_etl_spark_job',
       start_date=datetime(2023, 1, 1),
       schedule_interval=timedelta(days=1), # Run daily
       catchup=False,
       tags=['spark', 'etl'],
       default_args={
   ```

```
        'owner': 'airflow',
        'depends_on_past': False,
        'email_on_failure': False,
        'email_on_retry': False,
        'retries': 1,
        'retry_delay': timedelta(minutes=5),
    }
) as dag:
    # Task to submit the Spark batch transformation job
    # This command runs inside the Airflow worker container, which then execs into the
'spark' container
    # Ensure 'spark' container has the 'pyspark_jobs' mounted and dependencies
installed
    submit_spark_job = BashOperator(
        task_id='run_batch_transformation_spark_job',
        bash_command="""
            docker exec 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/batch_transformations.py \
                s3a://raw-data-bucket/financial_data_delta \
                s3a://curated-data-bucket/financial_data_curated
        """,
    )
```

Steps to Exercise:

1. **Place DAG:** Ensure simple_batch_etl_dag.py is in your airflow_dags/ folder. Airflow will automatically detect it.
2. **Unpause DAG:** In the Airflow UI (http://localhost:8080), find simple_batch_etl_spark_job and toggle it to "On" (unpause).
3. **Trigger DAG:** Manually trigger a run by clicking the "Play" icon.
4. **Monitor:** Go to the "Graph View" or "Gantt Chart" to observe task execution. Click on the run_batch_transformation_spark_job task and then "Log" to see the spark-submit output.

Verification:

- **Airflow UI:** The DAG run shows "success" (green).
- **MinIO Console:** Navigate to http://localhost:9001, then curated-data-bucket. You

should see new Delta Lake files (.parquet, _delta_log) in financial_data_curated path, indicating successful Spark processing.
- **Spark History Server (Optional):** Check http://localhost:18080 for details of the completed Spark job.

Advanced Use Case 1: Data-Driven Dependencies & SLA Management

Objective: To trigger a DAG only when new data files arrive in the raw S3 (MinIO) bucket and define a Service Level Agreement (SLA) for its completion. This ensures pipelines are data-activated and critical deadlines are met.
Role in Platform: Build reactive and reliable data pipelines.
Setup/Configuration:

1. **Ensure Basic Use Case setup is complete.**
2. **Prepare a Sensor DAG:** Create a new DAG (e.g., data_arrival_sensor_dag.py) that uses an S3KeySensor (or a custom sensor if needed for Delta Lake completion signals).
3. **Define SLA:** Add sla parameter to DAG or tasks.
   Example airflow_dags/data_arrival_sensor_dag.py (conceptual):
```
from airflow import DAG
from airflow.providers.amazon.aws.sensors.s3 import S3KeySensor # Requires
apache-airflow-providers-amazon
from airflow.operators.bash import BashOperator
from datetime import datetime, timedelta

with DAG(
   dag_id='data_arrival_sensor_with_sla',
   start_date=datetime(2023, 1, 1),
   schedule_interval=None, # Triggered manually or by external system
   catchup=False,
   tags=['s3', 'sensor', 'sla'],
   default_args={
      'owner': 'airflow',
      'depends_on_past': False,
      'email_on_failure': False,
      'email_on_retry': False,
      'retries': 0,
      'retry_delay': timedelta(minutes=1),
      'sla': timedelta(minutes=10) # SLA: Task must complete within 10 minutes of start
   }
) as dag:
   # Sensor to wait for a specific file pattern in MinIO
   # Note: S3KeySensor by default uses boto3, ensure minio is configured as S3
endpoint
```

```
    wait_for_financial_data = S3KeySensor(
        task_id='wait_for_new_financial_data_file',
        bucket_name='raw-data-bucket',
        # Key should be a pattern that indicates a new partition/file is ready
        # e.g., for daily partitions: 'financial_data_delta/daily_load_{{ ds }}/_SUCCESS'
        # For streaming, you might look for a new parquet file in the latest micro-batch
directory
        # For a simpler test, just look for any new parquet file in the path
        prefix='financial_data_delta/', # Just checks if files exist under this prefix
        wildcard_match=True, # Allows prefix/wildcard matching
        poke_interval=5, # Check every 5 seconds
        timeout=60 * 60, # Timeout after 1 hour if file not found
        # Ensure your MinIO setup is configured for S3 compatible endpoints for boto3
        # In a real environment, you'd specify aws_conn_id
    )

    # Once data arrives, trigger the transformation
    run_transformation = BashOperator(
        task_id='transform_financial_data_after_arrival',
        bash_command="""
            echo "New financial data detected! Starting transformation..."
            docker exec 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/batch_transformations.py \
                s3a://raw-data-bucket/financial_data_delta \
                s3a://curated-data-bucket/financial_data_curated_sensor_triggered
        """,
        sla=timedelta(minutes=5) # This task must complete within 5 minutes of starting
    )

    wait_for_financial_data >> run_transformation
```

Note: For S3KeySensor to work with MinIO, you might need to ensure apache-airflow-providers-amazon is installed in your Airflow Docker image and configure a dummy AWS connection in Airflow that points s3.amazonaws.com to your MinIO endpoint using extra_args, or modify the sensor to use a custom S3 client. For

local testing, a BashOperator with curl or mc commands polling MinIO might be simpler.
Steps to Exercise:

1. **Place DAG:** Put data_arrival_sensor_dag.py in airflow_dags/.
2. **Unpause DAG:** In Airflow UI, unpause data_arrival_sensor_with_sla. It will start a DAG run, and the wait_for_new_financial_data_file task will go into up_for_reschedule (poking) state.
3. **Generate Data to Trigger Sensor:**
   - Ensure your simulate_data.py script is running and sending financial data.
   - The Spark streaming job for financial data (streaming_consumer.py) should be running, which writes to s3a://raw-data-bucket/financial_data_delta.
   - The S3KeySensor will detect the new files appearing in this path.
4. **Monitor SLA:** In Airflow UI, observe the run_transformation task. If it takes longer than 5 minutes to complete after starting, Airflow will mark an SLA Miss.

Verification:

- **Airflow UI:** The wait_for_new_financial_data_file sensor task eventually succeeds (turns green). The run_transformation task executes and also succeeds.
- **MinIO Console:** New curated data appears in curated-data-bucket/financial_data_curated_sensor_triggered.
- **Airflow SLA:** If the run_transformation task exceeds 5 minutes, an "SLA Miss" notification will appear in the Airflow UI, demonstrating SLA management.

Advanced Use Case 2: Dynamic DAG Generation & Data Backfilling

Objective: To demonstrate generating multiple, similar DAGs dynamically from a configuration, allowing for easier management of many data pipelines, and then performing a historical backfill for one of these dynamically generated DAGs.
Role in Platform: Manage pipeline sprawl and handle historical data reprocessing.
Setup/Configuration:

1. **Define a configuration for data sources:** Create a file (e.g., config/data_sources.json) to define different financial/insurance data sources, each needing a similar ETL pipeline.
2. **Create a dynamic DAG factory:** Write a Python script in airflow_dags/ that reads this configuration and generates multiple DAG objects based on a template.
   Example config/data_sources.json:
   ```
   [
    {
     "source_name": "financial_transactions_source_a",
     "kafka_topic": "raw_financial_transactions_a",
     "raw_delta_path": "s3a://raw-data-bucket/financial_a_delta",
     "curated_delta_path": "s3a://curated-data-bucket/financial_a_curated"
    },
    {
     "source_name": "financial_transactions_source_b",
   ```

```
   "kafka_topic": "raw_financial_transactions_b",
   "raw_delta_path": "s3a://raw-data-bucket/financial_b_delta",
   "curated_delta_path": "s3a://curated-data-bucket/financial_b_curated"
 },
 {
   "source_name": "insurance_claims_source_c",
   "kafka_topic": "raw_insurance_claims_c",
   "raw_delta_path": "s3a://raw-data-bucket/insurance_c_delta",
   "curated_delta_path": "s3a://curated-data-bucket/insurance_c_curated"
 }
]
```

Example airflow_dags/dynamic_pipeline_generator.py (conceptual):

```
import os
import json
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime, timedelta

# Load configuration from a JSON file (assuming 'config' directory at project root)
CONFIG_FILE_PATH = os.path.join(os.environ.get("AIRFLOW_HOME", "/opt/airflow"),
"src/config/data_sources.json")

def create_etl_dag(source_config):
   """Creates a templated ETL DAG based on source configuration."""
   source_name = source_config['source_name']
   kafka_topic = source_config['kafka_topic']
   raw_delta_path = source_config['raw_delta_path']
   curated_delta_path = source_config['curated_delta_path']

   with DAG(
      dag_id=f'dynamic_etl_pipeline_{source_name}',
      start_date=datetime(2023, 1, 1),
      schedule_interval=timedelta(days=1),
      catchup=False,
      tags=['dynamic', 'spark', source_name],
      default_args={
         'owner': 'airflow',
         'depends_on_past': False,
         'email_on_failure': False,
         'email_on_retry': False,
         'retries': 1,
         'retry_delay': timedelta(minutes=5),
```

```
    }
  ) as dag:
    # Task to submit Spark streaming consumer (reads from Kafka to Raw Delta)
    run_streaming_consumer = BashOperator(
      task_id=f'run_{source_name}_streaming_consumer',
      bash_command=f"""
        docker exec spark spark-submit \
          --packages
org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0,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/streaming_consumer.py \
          {kafka_topic} kafka:29092 {raw_delta_path}
      """,
    )

    # Task to submit Spark batch transformation (Raw to Curated)
    run_batch_transformation = BashOperator(
      task_id=f'run_{source_name}_batch_transformation',
      bash_command=f"""
        docker exec 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/batch_transformations.py \
          {raw_delta_path} {curated_delta_path}
      """,
    )

    run_streaming_consumer >> run_batch_transformation

  return dag
```

```
# --- Main execution to generate DAGs ---
if os.path.exists(CONFIG_FILE_PATH):
    with open(CONFIG_FILE_PATH, 'r') as f:
        data_sources = json.load(f)
    for source_config in data_sources:
        globals()[f'dynamic_etl_pipeline_{source_config["source_name"]}'] =
create_etl_dag(source_config)
else:
    print(f"Config file not found: {CONFIG_FILE_PATH}. No dynamic DAGs will be
created.")
```

Note: You would need to ensure your docker-compose.yml mounts src/config into the Airflow container's AIRFLOW_HOME or a path accessible by the DAGs for dynamic_pipeline_generator.py to read it.

Steps to Exercise:

1. **Create Config:** Place data_sources.json in a config/ directory at your project root.
2. **Place DAG Generator:** Put dynamic_pipeline_generator.py in airflow_dags/.
3. **Observe Dynamic DAGs:** In Airflow UI, refresh the page. You should now see multiple DAGs appear (e.g., dynamic_etl_pipeline_financial_transactions_source_a, etc.).
4. **Perform Backfill:**
    - Select one of the dynamically generated DAGs (e.g., dynamic_etl_pipeline_financial_transactions_source_a).
    - From the DAGs list, click the "DAGs" dropdown, then "Trigger DAG w/ config" to select it.
    - In the DAG details page, click the "Graph View" tab.
    - Click the "Action" dropdown and select "Clear/Mark success". Choose a date range (e.g., last 3 days) and enable "Past" and "Future" if necessary. Select "Task Instances" and click "Clear". This will reset the state for those past runs.
    - Alternatively, from the command line (from your Airflow container or a machine with Airflow CLI installed and configured to connect to your Airflow DB):
      ```
      docker exec -it airflow-scheduler airflow dags backfill \
        -s 2023-01-01 -e 2023-01-03 \
        dynamic_etl_pipeline_financial_transactions_source_a
      ```
    - **Simulate Historical Data:** For backfill to process data, you would need historical data present in Kafka or MinIO corresponding to the backfill dates. This often involves re-ingesting or copying historical data for the period.

Verification:

- **Airflow UI:** Multiple, similarly structured DAGs are visible. After the backfill, you will see multiple historical DAG runs for the selected DAG, indicating successful reprocessing of past data periods.
- **MinIO Console:** Observe new or updated data in the raw_delta_path and curated_delta_path specified in the data_sources.json for the backfilled DAG,

demonstrating historical data processing.

Advanced Use Case 3: Cross-Platform Orchestration & External System Integration

Objective: To demonstrate Airflow's capability to orchestrate tasks involving external systems beyond Spark, such as triggering an OpenMetadata metadata ingestion and interacting with PostgreSQL for data validation or lookups.
Role in Platform: Create comprehensive data governance workflows and integrate heterogeneous systems.
Setup/Configuration:

1. **Ensure OpenMetadata is configured and running** (Advanced Track setup).
2. **Ensure PostgreSQL is running** (Advanced Track setup).
3. **Prepare OpenMetadata Ingestion Script:** You should have a Python script (e.g., openmetadata_ingestion_scripts/ingest_s3_metadata.py) that uses the OpenMetadata Python client to ingest metadata from MinIO/S3.
4. **Create an Integration DAG:** A DAG that includes tasks to:
   - Run a Spark job (as before).
   - Call the OpenMetadata ingestion script (e.g., using BashOperator or PythonOperator).
   - Perform a database validation using PostgresOperator or a PythonOperator connecting to PostgreSQL.

Example airflow_dags/full_pipeline_with_governance_dag.py (conceptual):
```
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from airflow.providers.postgres.operators.postgres import PostgresOperator # Requires apache-airflow-providers-postgres
from datetime import datetime, timedelta

# Assume this script exists in openmetadata_ingestion_scripts/
# And AIRFLOW_HOME/openmetadata_ingestion_scripts is mounted
OM_INGESTION_SCRIPT = "/opt/airflow/openmetadata_ingestion_scripts/ingest_s3_metadata.py"

def _validate_record_count(**kwargs):
    """Python callable to perform a data quality check on PostgreSQL."""
    from sqlalchemy import create_engine, text
    # This assumes your Airflow environment can connect to Postgres
    # In docker-compose, this is typically 'postgres' service name
    pg_conn_str = "postgresql+psycopg2://user:password@postgres:5432/main_db"
    engine = create_engine(pg_conn_str)
    with engine.connect() as connection:
        result = connection.execute(text("SELECT COUNT(*) FROM financial_transactions;")).scalar()
```

```
        print(f"Current record count in PostgreSQL: {result}")
        if result < 100: # Example: Check for minimum records
            raise ValueError(f"Record count is too low: {result}")
    print("Record count validation successful.")

with DAG(
    dag_id='full_pipeline_with_governance',
    start_date=datetime(2023, 1, 1),
    schedule_interval=timedelta(days=1),
    catchup=False,
    tags=['governance', 'openmetadata', 'postgres'],
    default_args={
        'owner': 'airflow',
        'depends_on_past': False,
        'email_on_failure': False,
        'email_on_retry': False,
        'retries': 1,
        'retry_delay': timedelta(minutes=5),
    }
) as dag:
    # 1. Ingest raw data (example: assuming a FastAPI call or S3 sensor)
    # For simplicity, let's use a dummy task, or chain from a Spark job if it produces new data
    start_ingestion = BashOperator(
        task_id='start_data_ingestion',
        bash_command='echo "Simulating data ingestion..."',
    )

    # 2. Run Spark Transformation (example, could be financial or insurance)
    run_spark_transformation = BashOperator(
        task_id='run_spark_financial_transformation',
        bash_command="""
            docker exec 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/batch_transformations.py \
                s3a://raw-data-bucket/financial_data_delta \
                s3a://curated-data-bucket/financial_data_curated_full_pipeline
        """,
    )

    # 3. Validate data in PostgreSQL (e.g., lookup table updates, audit counts)
    validate_postgres_data = PythonOperator(
```

```
        task_id='validate_financial_data_in_postgres',
        python_callable=_validate_record_count,
        provide_context=True,
    )

    # 4. Ingest new metadata into OpenMetadata
    ingest_openmetadata = BashOperator(
        task_id='ingest_openmetadata_for_financial_data',
        # This assumes your OpenMetadata ingestion script can be run this way
        # It should connect to your OM server and source MinIO/S3 metadata
        bash_command=f"docker exec openmetadata python {OM_INGESTION_SCRIPT}
--source minio --entity financial_data_curated_full_pipeline",
        # This is a highly conceptual command. In reality, the script would be more complex
        # and might run in its own container or use the OpenMetadata ingestion client in Airflow
worker
    )

    start_ingestion >> run_spark_transformation >> validate_postgres_data >>
ingest_openmetadata
```

Note: The ingest_openmetadata Bash command is highly conceptual. In a real setup, OpenMetadata ingestion often runs via Python scripts with the OpenMetadata SDK, which would need to be accessible and configured within the Airflow worker environment or a separate container.

Steps to Exercise:

1. **Place DAG:** Put full_pipeline_with_governance_dag.py in airflow_dags/.
2. **Ensure OM_INGESTION_SCRIPT is valid/dummy placeholder:** Verify the path and command for ingest_openmetadata is correct for your conceptual script.
3. **Unpause and Trigger DAG:** In Airflow UI, unpause and trigger full_pipeline_with_governance.
4. **Monitor:** Observe DAG run in Airflow UI, check task logs.

Verification:

- **Airflow UI:** The DAG run completes successfully, with all tasks (start_data_ingestion, run_spark_financial_transformation, validate_financial_data_in_postgres, ingest_openmetadata_for_financial_data) turning green.
- **MinIO Console:** Confirm new data in curated-data-bucket/financial_data_curated_full_pipeline.
- **PostgreSQL:** Run a query to confirm _validate_record_count was able to connect and query.
- **OpenMetadata UI:** After the ingest_openmetadata task completes, navigate to http://localhost:8585. Search for your financial_data_curated_full_pipeline dataset. You should see its metadata updated or created, demonstrating that Airflow successfully triggered the metadata ingestion.