Deep Dive: Cloud Component Comparison (AWS, Azure, GCP)

This document provides a high-level comparison of the cloud-native service equivalents for the core components of your enterprise-ready data platform across the three major cloud providers: Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Understanding these equivalences is fundamental for strategic decision-making regarding cloud migration, multi-cloud strategies, or leveraging managed services.

The choice of cloud provider often depends on existing infrastructure, organizational expertise, specific service features, pricing models, and compliance requirements. This comparison highlights the primary services that align with the functionalities demonstrated in your local environment.

1. Comparative Overview of Cloud Components

The table below maps your local open-source components to their most common and functionally equivalent managed services in AWS, Azure, and GCP. Note that while services often provide similar capabilities, their underlying architectures, pricing, and specific feature sets can vary significantly.

2. Key Considerations for Cloud Migration

When considering migrating your local data platform to one of these cloud environments, several factors come into play:

a. Managed Services vs. Self-Managed

Managed Services (PaaS/SaaS): Most cloud equivalents listed are managed services, reducing operational overhead (patching, scaling, backups). This is a significant advantage over self-managing open-source components on VMs.

Cost: While managed services abstract complexity, their operational costs can sometimes be higher than finely tuned self-managed solutions on raw VMs, especially at extreme scale. However, the total cost of ownership often favors managed services due to reduced labor.

b. Data Ingestion & Storage Strategy

Data Lake vs. Data Warehouse: All three clouds advocate a "lakehouse" architecture. Data typically lands in object storage (S3, Blob Storage, Cloud Storage) first, then processed into structured formats in data warehouses (Snowflake, Redshift, Synapse, BigQuery).

Streaming Ingestion: Cloud-native streaming services (Kinesis, Event Hubs, Pub/Sub) offer high throughput and seamless integration with other cloud services. Snowpipe is Snowflake's specific solution for continuous file ingestion.

c. Compute Paradigm

Serverless Compute (Lambda, Azure Functions, Cloud Functions/Run): Ideal for event-driven, short-lived, or bursty tasks (e.g., lightweight data transformations, API endpoints, triggering pipelines).

Managed Cluster Compute (EMR, Databricks, Dataproc): Best for large-scale, long-running, or complex batch/streaming jobs that require distributed processing frameworks like Spark.

Data Warehouse Compute (Redshift, Synapse, BigQuery, Snowflake): Optimized for analytical queries and transformations on structured and semi-structured data within the warehouse itself. Snowpark extends this to allow Python/Java/Scala code execution directly on Snowflake compute.

d. Observability & Governance

Integrated Monitoring: Each cloud provider has its own comprehensive monitoring suite (CloudWatch, Azure Monitor, Cloud Monitoring) that natively integrates with their services.

Unified Logging: Centralized logging services (CloudWatch Logs, Azure Monitor Logs, Cloud Logging) collect logs from all services, crucial for troubleshooting.

Data Catalogs: Services like AWS Glue Data Catalog, Azure Purview, and GCP Data Catalog provide metadata management, lineage, and discovery features, some with deeper native integrations than others. OpenMetadata often provides a vendor-agnostic layer on top.

e. Ecosystem and Integration

Deep Integrations: Choosing a single cloud provider often simplifies integrations as services within that ecosystem are designed to work seamlessly together.

Vendor Lock-in: Relying heavily on proprietary cloud services can lead to vendor lock-in, making future migration to another cloud more challenging. Your local open-source setup provides flexibility in this regard.

By understanding these cloud equivalents and the underlying considerations, you can strategically plan the evolution of your enterprise data platform to a scalable, production-ready cloud environment.

This concludes the deep dive into cloud component comparisons.

Deep Dive: Integrating AI/LLMs/MLOps

This document explores how your enterprise-ready data platform extends its capabilities to support Artificial Intelligence (AI) and Large Language Models (LLMs), all managed under the umbrella of robust Machine Learning Operations (MLOps) principles. While traditional ML focuses on structured data and predictive models, AI/LLMs introduce new dimensions related to unstructured text, natural language understanding, and generative capabilities. MLOps provides the essential operational framework to reliably develop, deploy, and monitor all types of AI and ML solutions.

1. Core Concepts & Platform Alignment for AI/LLMs/MLOps

Your existing data platform, designed for enterprise-grade data management, is inherently well-suited to integrate AI/LLM workloads.

Data Pipelines for AI/LLMs:

Ingestion (FastAPI, Kafka, Delta Lake): Just as with structured data, raw text, documents, or conversation logs are ingested into the platform. This raw data forms the basis for Retrieval Augmented Generation (RAG) systems or fine-tuning datasets for LLMs.

Feature Engineering (Apache Spark): For traditional ML, Spark processes structured features. For LLMs, Spark can perform text preprocessing, chunking, embedding generation, or preparing structured prompts/responses for fine-tuning.

Data Quality & Governance (OpenMetadata): Critical for both traditional ML and LLMs. Ensuring the quality of text data used for RAG or fine-tuning, and maintaining lineage for prompt templates, models, and generated outputs.

Model Management (MinIO/Delta Lake, conceptual Model Registry):

Trained traditional ML models are versioned in your data lakehouse (MinIO/Delta Lake).

LLMs, whether open-source models or proprietary ones accessed via API, also require versioning of their configurations, fine-tuned weights (if applicable), and prompt templates. MinIO/Delta Lake can store these assets.

Orchestration (Apache Airflow):

Airflow remains the central orchestrator for complex pipelines:

Data preparation for model training/fine-tuning.

Automated training/fine-tuning jobs.

Batch inference jobs.

Model evaluation and validation workflows.

Deployment triggers for new model versions or prompt templates.

Observability (OpenTelemetry, Grafana Alloy, Grafana):

Monitoring extends beyond system health to model performance. For LLMs, this includes metrics like:

Latency: Time to generate responses.

Token Usage: Input/output token counts for cost management.

Quality Metrics: Hallucination rates, relevance, coherence (often requires human-in-the-loop or proxy metrics).

Abuse/Safety: Monitoring for inappropriate content generation.

Traces can follow LLM calls, and logs can include prompts and responses (with necessary PII masking).

2. Interactive How-Tos: Integrating AI/LLMs/MLOps

Let's explore some practical examples of how these integrations can take shape within your platform.

Basic Use Case: Data Preparation for LLM Fine-tuning/RAG

Objective: To demonstrate how Apache Spark can preprocess unstructured text data (e.g., articles, customer support transcripts) from your raw data lake into a structured format suitable for LLM fine-tuning or Retrieval Augmented Generation (RAG) systems.

Role in Platform: Transform raw, unstructured data into a consumable format for LLMs, enabling the creation of custom knowledge bases or training datasets.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d. This includes spark and minio.

Simulate Raw Text Data: Create a dummy text file raw\_docs.json that looks like ingested unstructured data.

In your data/minio/raw-data-bucket/ directory, create articles/raw\_docs.json.

# data/minio/raw-data-bucket/articles/raw\_docs.json  
{"id": "doc1", "text": "Apache Spark is a unified analytics engine for large-scale data processing. It provides high-level APIs in Java, Scala, Python and R. Spark also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Structured Streaming for incremental computation and stream processing."}  
{"id": "doc2", "text": "Large Language Models (LLMs) are deep learning models trained on vast amounts of text data. They are capable of understanding, generating, and manipulating human language. Popular LLMs include GPT-3, LLaMA, and Gemini. Applications range from chatbots and content generation to code completion and translation."}  
{"id": "doc3", "text": "Retrieval Augmented Generation (RAG) is an AI framework for improving the specificity and factual accuracy of generative AI models with information retrieved from external knowledge bases. RAG models combine a retrieval component with a generation component. Instead of generating responses solely based on their training data, RAG models first retrieve relevant documents or passages from a given knowledge base and then use this retrieved information to inform their generated response."}  
{"id": "doc4", "text": "MLOps (Machine Learning Operations) is a set of practices that aims to deploy and maintain machine learning models in production reliably and efficiently. The MLOps lifecycle includes data preparation, model training, deployment, monitoring, and governance. Key tools often include orchestrators like Airflow and observability platforms like Grafana."}

Create a PySpark script for LLM data preparation: In pyspark\_jobs/, create llm\_data\_prep.py.  
# pyspark\_jobs/llm\_data\_prep.py  
import sys  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import col, lit, current\_timestamp, split, explode, monotonically\_increasing\_id, concat\_ws  
from pyspark.sql.types import StructType, StringType, IntegerType  
  
def create\_spark\_session(app\_name):  
 """Helper function to create a SparkSession with Delta Lake packages."""  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "io.delta:delta-core\_2.12:2.4.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
# Simple text chunking function (for RAG)  
# In a real scenario, this would be more sophisticated (e.g., using NLTK, spaCy, or Hugging Face tokenizers)  
# This UDF is illustrative. For large-scale text, pure Spark functions are preferred where possible.  
# from pyspark.sql.functions import udf  
# @udf(returnType=ArrayType(StringType()))  
# def chunk\_text\_udf(text: str, chunk\_size: int = 1000):  
# if not text:  
# return []  
# words = text.split(" ")  
# chunks = []  
# current\_chunk = []  
# for word in words:  
# current\_chunk.append(word)  
# if len(" ".join(current\_chunk)) >= chunk\_size:  
# chunks.append(" ".join(current\_chunk))  
# current\_chunk = []  
# if current\_chunk:  
# chunks.append(" ".join(current\_chunk))  
# return chunks  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if len(sys.argv) != 3:  
 print("Usage: llm\_data\_prep.py <input\_raw\_text\_path> <output\_delta\_path>")  
 sys.exit(-1)  
  
 input\_raw\_text\_path = sys.argv[1]  
 output\_delta\_path = sys.argv[2]  
  
 spark = create\_spark\_session("LLMDataPreparation")  
 spark.sparkContext.setLogLevel("WARN")  
  
 print(f"Reading raw text data from: {input\_raw\_text\_path}")  
 # Read raw JSON lines from MinIO/S3  
 df\_raw = spark.read.json(input\_raw\_text\_path)  
 df\_raw.printSchema()  
 df\_raw.show(5, truncate=False)  
  
 print("Performing text preprocessing and chunking (conceptual)...")  
 # Example: Simple chunking by paragraph (splitting by double newline for illustration)  
 # In a real application, you'd use a more robust text splitter, potentially preserving context.  
 df\_chunked = df\_raw.withColumn("paragraphs", split(col("text"), "\\n\\n")) \  
 .withColumn("chunk", explode(col("paragraphs"))) \  
 .filter(col("chunk") != "") # Remove empty chunks  
  
 # Assign a unique ID to each chunk and add metadata  
 df\_final = df\_chunked.withColumn("chunk\_id", concat\_ws("\_", col("id"), monotonically\_increasing\_id())) \  
 .select(  
 col("chunk\_id"),  
 col("id").alias("source\_document\_id"),  
 col("chunk").alias("text\_content"),  
 current\_timestamp().alias("processed\_at")  
 )  
  
 print("Schema of prepared data for LLM:")  
 df\_final.printSchema()  
 df\_final.show(5, truncate=False)  
  
 # Write the prepared data to a new Delta Lake table  
 print(f"Writing prepared LLM data to: {output\_delta\_path}")  
 df\_final.write.format("delta") \  
 .mode("overwrite") \  
 .option("overwriteSchema", "true") \  
 .save(output\_delta\_path)  
 print("LLM data preparation job completed.")  
  
 spark.stop()

Steps to Exercise:

Place raw\_docs.json: Ensure the raw\_docs.json file is in data/minio/raw-data-bucket/articles/.

Submit the Spark LLM Data Prep Job:  
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/articles/raw\_docs.json \  
 s3a://curated-data-bucket/llm\_prepared\_docs

Monitor Spark Job: Observe the console output for Spark logs, confirming data reading, processing, and writing.

Verify Prepared Data in MinIO: Access the MinIO Console (http://localhost:9001). Navigate to curated-data-bucket/llm\_prepared\_docs/. You should see new Delta Lake files.

Query Prepared Data via Spark SQL:  
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 \* FROM delta.\`s3a://curated-data-bucket/llm\_prepared\_docs\` LIMIT 10;"

Verification:

Spark Job Completion: The job runs successfully, indicated by console output.

MinIO Contents: The llm\_prepared\_docs Delta Lake table is created with new .parquet files and \_delta\_log.

Spark SQL Query: The query results show the text\_content column, demonstrating that raw text has been transformed into a structured format with unique chunk\_ids, ready for use in RAG or fine-tuning datasets.

Advanced Use Case 1: Integrating LLM Inference into an Application (Conceptual)

Objective: To conceptually demonstrate how a FastAPI service could integrate with an external LLM API (e.g., Gemini API) to enrich incoming data or generate responses. This shows how your platform's API layer can become a gateway for AI capabilities.

Role in Platform: Enable real-time AI-powered features, such as intelligent routing of support tickets, sentiment analysis on customer feedback, or dynamic content generation.

Setup/Configuration:

Ensure FastAPI is running: Your fastapi\_ingestor service should be active.

Conceptual LLM API Key: Assume an API key for the LLM service exists (e.g., in environment variables).

Modify fastapi\_app/app/main.py: Add a new endpoint that calls a placeholder LLM API.  
# fastapi\_app/app/main.py (conceptual additions for LLM integration)  
# ... existing imports  
import httpx # For making async HTTP requests  
# ... existing OpenTelemetry setup, Kafka producer, Pydantic models, etc.  
  
# --- LLM Integration Configuration ---  
LLM\_API\_URL = os.getenv("LLM\_API\_URL", "https://generativelanguage.googleapis.com/v1beta/models/gemini-2.0-flash:generateContent")  
LLM\_API\_KEY = os.getenv("LLM\_API\_KEY", "") # IMPORTANT: In production, use AWS Secrets Manager or similar!  
  
# --- New Pydantic Models for LLM Interaction ---  
class LLMQueryRequest(BaseModel):  
 text\_input: str = Field(..., example="Explain the concept of data lineage in simple terms.")  
 context: Optional[str] = Field(None, example="Consider a financial data pipeline.")  
  
class LLMResponse(BaseModel):  
 original\_input: str  
 generated\_text: str  
 tokens\_used: Optional[int] = None  
 processing\_time\_ms: Optional[float] = None  
 error: Optional[str] = None  
  
  
# --- New FastAPI Endpoint for LLM Interaction ---  
@app.post("/llm-query/", response\_model=LLMResponse, tags=["AI/LLM"])  
async def llm\_query(request: LLMQueryRequest):  
 start\_time = time.time()  
 prompt\_text = request.text\_input  
 if request.context:  
 prompt\_text = f"Context: {request.context}\n\nQuery: {request.text\_input}"  
  
 try:  
 # Prepare chat history for the LLM API call  
 chatHistory = []  
 chatHistory.push({ "role": "user", "parts": [{ "text": prompt\_text }] })  
  
 # Prepare the payload for the LLM API  
 payload = { "contents": chatHistory }  
 # If you want structured response, add generationConfig  
 # payload["generationConfig"] = {  
 # "responseMimeType": "application/json",  
 # "responseSchema": {  
 # "type": "OBJECT",  
 # "properties": {  
 # "summary": { "type": "STRING" },  
 # "keywords": { "type": "ARRAY", "items": { "type": "STRING" } }  
 # }  
 # }  
 # }  
  
 # Make the async HTTP request to the LLM API  
 async with httpx.AsyncClient() as client:  
 response = await client.post(  
 f"{LLM\_API\_URL}?key={LLM\_API\_KEY}",  
 json=payload,  
 timeout=60 # Adjust timeout as needed for LLM response  
 )  
 response.raise\_for\_status() # Raise an exception for bad status codes  
  
 llm\_result = response.json()  
 generated\_text = "No response from LLM."  
 tokens\_used = None # LLM API might provide token usage  
  
 if llm\_result.get("candidates") and len(llm\_result["candidates"]) > 0:  
 first\_candidate = llm\_result["candidates"][0]  
 if first\_candidate.get("content") and first\_candidate["content"].get("parts"):  
 # Extract the text from the LLM's response  
 generated\_text = first\_candidate["content"]["parts"][0]["text"]  
  
 # Conceptual token usage (replace with actual API response parsing)  
 # if llm\_result.get("usage\_metadata"):  
 # tokens\_used = llm\_result["usage\_metadata"].get("total\_token\_count")  
  
 processing\_time\_ms = (time.time() - start\_time) \* 1000  
  
 # Log the LLM interaction (with PII masking if necessary)  
 print(f"LLM Query: '{request.text\_input[:50]}...', Response: '{generated\_text[:50]}...', Time: {processing\_time\_ms:.2f}ms")  
  
 return LLMResponse(  
 original\_input=request.text\_input,  
 generated\_text=generated\_text,  
 tokens\_used=tokens\_used,  
 processing\_time\_ms=processing\_time\_ms  
 )  
  
 except httpx.HTTPStatusError as e:  
 error\_msg = f"LLM API HTTP Error: {e.response.status\_code} - {e.response.text}"  
 print(error\_msg)  
 raise HTTPException(status\_code=e.response.status\_code, detail=error\_msg)  
 except httpx.RequestError as e:  
 error\_msg = f"LLM API Request Error: {e}"  
 print(error\_msg)  
 raise HTTPException(status\_code=500, detail=error\_msg)  
 except json.JSONDecodeError:  
 error\_msg = "Invalid JSON response from LLM API."  
 print(error\_msg)  
 raise HTTPException(status\_code=500, detail=error\_msg)  
 except Exception as e:  
 error\_msg = f"An unexpected error during LLM query: {e}"  
 print(error\_msg)  
 raise HTTPException(status\_code=500, detail=error\_msg)  
  
  
Note: You'll need to add httpx to fastapi\_app/requirements.txt and rebuild the FastAPI image.

Steps to Exercise:

Add httpx to fastapi\_app/requirements.txt:  
# fastapi\_app/requirements.txt  
...  
httpx

Rebuild FastAPI image and restart container:  
docker compose build fastapi\_ingestor  
docker compose restart fastapi\_ingestor

Send a query to the new LLM endpoint:  
curl -X POST -H "Content-Type: application/json" \  
 -d '{  
 "text\_input": "What is the capital of France?",  
 "context": "Answer briefly."  
 }' \  
 http://localhost:8000/llm-query/  
  
If you have a real Gemini API key, ensure LLM\_API\_KEY is set as an environment variable for the fastapi\_ingestor service in docker-compose.yml.  
If not, the request will likely fail, but the internal logic of parsing and attempting the call will be demonstrated.

Verification:

HTTP Response: You will receive a JSON response from the FastAPI endpoint. If the LLM API call was successful (e.g., if you provided a valid API key), generated\_text will contain the LLM's response. If not, it will show an error message.

FastAPI Logs: The fastapi\_ingestor logs will show the "LLM Query..." message, confirming the endpoint was hit and the conceptual interaction occurred.

Advanced Use Case 2: MLOps for LLMs - Monitoring & Evaluation (Conceptual)

Objective: To discuss and conceptually outline how LLM-specific metrics and evaluation results can be collected using OpenTelemetry and visualized in Grafana, crucial for understanding LLM performance and cost in production.

Role in Platform: Extend observability to cover LLM-specific KPIs, enabling proactive monitoring for issues like high latency, excessive token usage, or potential "hallucinations."

Setup/Configuration:

Ensure OpenTelemetry is configured for FastAPI: Your fastapi\_app/app/main.py should have the OpenTelemetry setup (as detailed in "Highlighting OpenTelemetry" document), with OTLPMetricExporter sending to grafana-alloy.

Ensure Grafana Alloy and Grafana are running.

Steps to Exercise (Conceptual Discussion):

Define Custom LLM Metrics (within main.py where LLM call is made):

Latency: llm.request.duration\_ms (Histogram with attributes like model\_name, endpoint, success).

Token Usage: llm.input.tokens\_total, llm.output.tokens\_total (Counters with attributes like model\_name).

Cost: llm.estimated.cost\_usd (Counter, if you can estimate cost per token).

Error Rate: llm.api.errors\_total (Counter with error\_type, model\_name).

Conceptual Quality Metrics (Proxy): llm.response.length\_chars, llm.response.keywords\_count (Gauges, or derived metrics from text post-processing).

# fastapi\_app/app/main.py (conceptual additions for LLM metrics)  
# ... existing OpenTelemetry setup ...  
# From Advanced Use Case 1 in OpenTelemetry:  
# meter = metrics.get\_meter("fastapi.ingestion.app")  
  
# Create LLM-specific metrics  
llm\_request\_duration = meter.create\_histogram(  
 "llm.request.duration",  
 description="Duration of LLM API requests",  
 unit="ms",  
 boundaries=[50, 100, 200, 500, 1000, 2000, 5000, 10000] # Adjust for expected LLM latencies  
)  
llm\_input\_tokens = meter.create\_counter(  
 "llm.input.tokens\_total",  
 description="Total input tokens sent to LLM",  
 unit="1"  
)  
llm\_output\_tokens = meter.create\_counter(  
 "llm.output.tokens\_total",  
 description="Total output tokens received from LLM",  
 unit="1"  
)  
llm\_error\_count = meter.create\_counter(  
 "llm.api.errors\_total",  
 description="Total errors from LLM API calls",  
 unit="1"  
)  
  
# --- Inside @app.post("/llm-query/") endpoint ---  
# ...  
 # After successful LLM call:  
 llm\_request\_duration.record(processing\_time\_ms, {"model\_name": "gemini-2.0-flash", "success": True})  
 # if actual\_input\_tokens:  
 # llm\_input\_tokens.add(actual\_input\_tokens, {"model\_name": "gemini-2.0-flash"})  
 # if actual\_output\_tokens:  
 # llm\_output\_tokens.add(actual\_output\_tokens, {"model\_name": "gemini-2.0-flash"})  
# ...  
# In error handling block (e.g., in except):  
# llm\_error\_count.add(1, {"error\_type": "http\_error" if isinstance(e, httpx.HTTPStatusError) else "other", "model\_name": "gemini-2.0-flash"})  
# llm\_request\_duration.record(processing\_time\_ms, {"model\_name": "gemini-2.0-flash", "success": False, "error\_type": "timeout"}) # Example

Visualize in Grafana:

After adding metrics and generating traffic, access http://localhost:3000.

Create a new dashboard or panel.

PromQL Queries:

LLM Request Rate: rate(llm\_request\_duration\_count{model\_name="gemini-2.0-flash"}[1m])

LLM P99 Latency: histogram\_quantile(0.99, sum by(le, model\_name) (rate(llm\_request\_duration\_bucket[1m])))

Total Input/Output Tokens: llm\_input\_tokens\_total{model\_name="gemini-2.0-flash"} (and for output)

Error Rate: rate(llm\_api\_errors\_total{model\_name="gemini-2.0-flash"}[5m])

Discuss LLM Evaluation & Monitoring for Drift:

Data Drift: Use OpenMetadata's data profiling on the llm\_prepared\_docs Delta table. Monitor for changes in text length distribution, vocabulary, or topic distribution of incoming text data.

Model Drift (LLM Output): This is harder.

Proxy Metrics: Monitor metrics derived from LLM output (e.g., llm.response.length\_chars). Sudden changes could indicate drift.

Human Evaluation: For critical LLM applications, a sample of responses might require periodic human review.

Embeddings: Generate embeddings for LLM inputs/outputs and monitor the distribution of these embeddings over time (e.g., using dimensionality reduction and clustering in Grafana for anomaly detection).

Logging Prompts/Responses: Crucial for post-hoc analysis and debugging. Ensure sensitive data is masked. OpenTelemetry LoggingInstrumentor can help link these logs to traces.

Verification (Conceptual):

Grafana Dashboards: Successfully visualize LLM-specific operational metrics like request rate, latency, and token usage.

Architectural Understanding: Demonstrate a clear understanding of how to set up monitoring for LLM applications, extending beyond traditional infrastructure metrics to capture critical AI-specific KPIs.

Advanced Use Case 3: Versioning and Deploying Models/Prompts (Conceptual MLOps Workflow)

Objective: To illustrate how MLOps principles (versioning, automated pipelines, controlled deployments) apply to both traditional ML models and LLM prompts/configurations within your platform.

Role in Platform: Ensure reproducibility, auditability, and reliable deployment of all AI/ML artifacts, from data to models to prompts.

Setup/Configuration (Conceptual Discussion):

Existing CI/CD Pipeline: Refer to .github/workflows/release.yml for the overall CI/CD structure.

MinIO/Delta Lake for Artifact Storage: MinIO acts as the artifact repository.

Airflow for Orchestration: Refer to airflow\_dags/ for pipeline orchestration.

Steps to Exercise (Conceptual/Discussion):

Model Versioning in Data Lakehouse:

ML Models: After training (model\_training\_job.py), the model is saved to a specific versioned path in MinIO/Delta Lake (e.g., s3a://models-bucket/financial\_fraud\_model/v1.0.0/).

LLM Prompts/Configs: Store prompt templates, RAG knowledge base configurations, or LLM fine-tuning datasets in versioned paths within MinIO/Delta Lake as well (e.g., s3a://llm-artifacts-bucket/prompts/sentiment\_analysis\_v1.0/prompt\_template.txt). This ensures traceability of the exact prompt used with a given model.

OpenMetadata Role: OpenMetadata can catalog these model and prompt artifacts, linking them to their source data and lineage.

Automated Training/Fine-tuning Pipelines (Airflow):

Airflow DAGs: Create Airflow DAGs that:

Trigger feature\_engineering\_job.py.

Trigger model\_training\_job.py (or a fine-tuning job for LLMs).

Run model evaluation (e.g., on a test dataset) and store metrics/artifacts.

Conditionally push the new model/prompt version to the artifact store.

Conceptually: Trigger a deployment pipeline if metrics are satisfactory.

Example Trigger: An Airflow DAG could be scheduled daily, or triggered by a data quality anomaly on features.

Deployment of Inference Services:

CI/CD: The existing CI/CD (GitHub Actions) can be extended.

Traditional ML: The FastAPI inference service (or a dedicated Spark Streaming inference job) would be deployed/updated to point to the new model version in MinIO. This could involve updating an environment variable for the FastAPI service or a Spark job parameter.

LLM Integration: If the FastAPI service directly integrates with an LLM API, prompt changes might be deployed by updating the FastAPI application code itself (e.g., new prompt templates in a config file). Or, if prompts are externalized to a data store, the deployment ensures the service picks up the latest version.

Blue/Green or Canary Deployments: In a production cloud environment, you'd apply blue/green or canary deployments for inference services to minimize downtime and risk during model/prompt rollouts.

Rollback Strategy:

Because models/prompts are versioned in MinIO/Delta Lake, a rollback is as simple as re-deploying the inference service to point to a previous, known-good version of the model or prompt. This makes ML deployments robust.

Verification (Conceptual):

MLOps Lifecycle Understanding: Demonstrates a comprehensive understanding of how data platform components support a robust MLOps lifecycle, including versioning of models and prompts, automated pipelines, and controlled deployments, which are essential for reliable AI/ML initiatives in an enterprise.

This concludes the deep dive into Integrating AI/LLMs/MLOps within your data platform.

Highlighting Apache Spark: Distributed Processing Engine

Apache Spark is a powerful, unified analytics engine for large-scale data processing. It excels at both batch processing and real-time streaming analytics, making it a cornerstone of modern data platforms. With its rich APIs in Python (PySpark), Scala, Java, and R, Spark allows engineers to perform complex transformations, aggregations, and machine learning tasks on massive datasets.

This guide will demonstrate basic and advanced use cases of Apache Spark, 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: Streaming ETL from Kafka to Delta Lake

Objective: To demonstrate how Spark Structured Streaming can consume real-time data from Kafka topics, apply a basic ETL process (e.g., parsing, schema enforcement), and write the results to a Delta Lake table in MinIO.

Role in Platform: Power the real-time ingestion pipeline, transforming raw event streams into structured, queryable data in the data lakehouse.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root.

Verify Spark is accessible: Check Docker logs for the spark container to ensure it's healthy. Access Spark History Server at http://localhost:18080.

Ensure Kafka topics are initialized: Confirm raw\_financial\_transactions and raw\_insurance\_claims topics exist (from onboard.sh or manual commands).

simulate\_data.py is running: Ensure data is continuously being sent to your FastAPI and then published to Kafka.

Spark Consumer Script: You will use the pyspark\_jobs/streaming\_consumer.py script. This script defines the logic for consuming from Kafka and writing to Delta Lake.  
Example pyspark\_jobs/streaming\_consumer.py (conceptual, as referenced in previous docs):  
# pyspark\_jobs/streaming\_consumer.py  
import sys  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import col, from\_json  
from pyspark.sql.types import StructType, StringType, FloatType, TimestampType, MapType  
  
def create\_spark\_session(app\_name):  
 """Helper function to create a SparkSession with Delta Lake and Kafka packages."""  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10\_2.12:3.5.0,io.delta:delta-core\_2.12:2.4.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if len(sys.argv) != 4:  
 print("Usage: streaming\_consumer.py <kafka\_topic> <kafka\_broker> <delta\_output\_path>")  
 sys.exit(-1)  
  
 kafka\_topic = sys.argv[1]  
 kafka\_broker = sys.argv[2]  
 delta\_output\_path = sys.argv[3]  
  
 spark = create\_spark\_session(f"KafkaToDeltaStream\_{kafka\_topic}")  
 spark.sparkContext.setLogLevel("WARN") # Reduce verbosity of Spark logs  
  
 # Define schema for the incoming Kafka message value (financial/insurance data)  
 # This schema should match the data structure produced by FastAPI  
 # For simplicity, using a generic schema, in reality you'd have specific ones  
 # for financial\_transactions and insurance\_claims  
 data\_schema = StructType() \  
 .add("transaction\_id", StringType(), True) \  
 .add("timestamp", StringType(), True) \  
 .add("account\_id", StringType(), True) \  
 .add("amount", FloatType(), True) \  
 .add("currency", StringType(), True) \  
 .add("transaction\_type", StringType(), True) \  
 .add("merchant\_id", StringType(), True) \  
 .add("category", StringType(), True) \  
 .add("claim\_id", StringType(), True) \  
 .add("policy\_number", StringType(), True) \  
 .add("claim\_amount", FloatType(), True) \  
 .add("claim\_type", StringType(), True) \  
 .add("claim\_status", StringType(), True) \  
 .add("customer\_id", StringType(), True) \  
 .add("incident\_date", StringType(), True)  
  
 # Read from Kafka as a streaming DataFrame  
 kafka\_df = (spark.readStream  
 .format("kafka")  
 .option("kafka.bootstrap.servers", kafka\_broker)  
 .option("subscribe", kafka\_topic)  
 .option("startingOffsets", "latest") # Start consuming new messages  
 .load())  
  
 # Parse the value column (which contains the JSON message)  
 # Add metadata for debugging (topic, offset, timestamp)  
 parsed\_df = kafka\_df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING) as json\_value",  
 "topic", "partition", "offset", "timestamp") \  
 .select(from\_json(col("json\_value"), data\_schema).alias("data"),  
 col("topic"), col("partition"), col("offset"), col("timestamp").alias("kafka\_timestamp")) \  
 .select("data.\*", "topic", "partition", "offset", "kafka\_timestamp")  
  
 # Define checkpoint location for fault tolerance and exactly-once processing  
 checkpoint\_location = f"{delta\_output\_path}/\_checkpoints"  
  
 # Write the processed data to Delta Lake  
 query = (parsed\_df.writeStream  
 .format("delta")  
 .outputMode("append") # Append new data to the Delta table  
 .option("checkpointLocation", checkpoint\_location) # Required for streaming writes  
 .start(delta\_output\_path))  
  
 print(f"Spark Structured Streaming job for topic '{kafka\_topic}' started, writing to: {delta\_output\_path}")  
 print(f"Checkpoint location: {checkpoint\_location}")  
  
 query.awaitTermination() # Keep the job running until terminated  
  
 spark.stop()

Steps to Exercise:

Ensure data generation: Verify python3 simulate\_data.py is running in the background.

Submit Financial Streaming Job: In a new terminal, submit the Spark job for financial data.  
docker exec -it 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 \  
 raw\_financial\_transactions kafka:29092 s3a://raw-data-bucket/financial\_data\_delta

Submit Insurance Streaming Job: In another new terminal, submit the Spark job for insurance data.  
docker exec -it 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 \  
 raw\_insurance\_claims kafka:29092 s3a://raw-data-bucket/insurance\_data\_delta  
  
Observe the console output for both jobs; they will show Spark logging and progress.

Verification:

MinIO Console (http://localhost:9001): Navigate to raw-data-bucket. You should see two growing directories: financial\_data\_delta and insurance\_data\_delta. Each will contain .parquet files (the actual data) and a \_delta\_log directory (Delta Lake transaction log), confirming continuous data ingestion.

Spark History Server (http://localhost:18080): You should see two active streaming applications. Click on them to inspect their progress, input/output rates, and micro-batch statistics.

Grafana (http://localhost:3000): On the "Kafka Overview" or "Health Dashboard," observe that the consumer lag for both raw\_financial\_transactions and raw\_insurance\_claims topics remains low and stable, indicating that Spark is efficiently consuming messages as they arrive.

Advanced Use Case 1: Complex Batch Transformation & Data Quality

Objective: To demonstrate Spark's capability for complex batch transformations, including data cleansing, enrichment (e.g., joining with a reference table in PostgreSQL), and applying data quality rules before writing to a curated zone.

Role in Platform: Refine raw data into high-quality, consumable datasets for analytics and machine learning.

Setup/Configuration:

Ensure Basic Use Case is running: raw-data-bucket/financial\_data\_delta is populated.

PostgreSQL reference data: Assume your PostgreSQL main\_db has a merchant\_lookup table with merchant\_id and merchant\_category (can be populated manually or via a setup script).

Spark Batch Transformation Script: Create pyspark\_jobs/batch\_transformations.py to handle these steps.  
Example pyspark\_jobs/batch\_transformations.py (conceptual):  
# pyspark\_jobs/batch\_transformations.py  
import sys  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import col, when, trim, lower, coalesce, lit, current\_timestamp  
from pyspark.sql.types import StringType, FloatType, TimestampType, LongType  
from delta.tables import DeltaTable  
  
def create\_spark\_session(app\_name):  
 """Helper function to create a SparkSession with Delta Lake and PostgreSQL packages."""  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "io.delta:delta-core\_2.12:2.4.0,org.postgresql:postgresql:42.6.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
def run\_transformation(spark, raw\_path, curated\_path, pg\_host, pg\_port, pg\_db, pg\_user, pg\_password):  
 """Performs batch transformation, enrichment, and data quality checks."""  
 print(f"Reading raw data from: {raw\_path}")  
 df\_raw = spark.read.format("delta").load(raw\_path)  
  
 # 1. Data Cleansing  
 df\_cleaned = df\_raw.withColumn("account\_id", trim(col("account\_id"))) \  
 .withColumn("currency", lower(col("currency")))  
  
 # 2. Data Enrichment (Joining with PostgreSQL lookup table)  
 # Assuming a merchant\_lookup table in PostgreSQL  
 print(f"Connecting to PostgreSQL at {pg\_host}:{pg\_port}/{pg\_db} for merchant lookup...")  
 df\_merchant\_lookup = spark.read \  
 .format("jdbc") \  
 .option("url", f"jdbc:postgresql://{pg\_host}:{pg\_port}/{pg\_db}") \  
 .option("dbtable", "merchant\_lookup") # Assuming 'merchant\_lookup' table  
 .option("user", pg\_user) \  
 .option("password", pg\_password) \  
 .load()  
  
 df\_enriched = df\_cleaned.join(  
 df\_merchant\_lookup,  
 df\_cleaned["merchant\_id"] == df\_merchant\_lookup["merchant\_id"],  
 "left" # Use left join to keep all financial transactions  
 ).select(df\_cleaned["\*"], coalesce(df\_merchant\_lookup["category"], lit("UNKNOWN")).alias("enriched\_category")) # Example enrichment  
  
 # 3. Data Quality Checks (Simple example: flagging invalid amounts)  
 df\_quality\_checked = df\_enriched.withColumn(  
 "is\_amount\_valid",  
 when(col("amount").isNull() | (col("amount") <= 0), False).otherwise(True)  
 ).withColumn("processing\_timestamp", current\_timestamp()) # Add processing timestamp  
  
 # Define schema for the curated table (conceptual)  
 curated\_schema = StructType() \  
 .add("transaction\_id", StringType()) \  
 .add("timestamp", StringType()) \  
 .add("account\_id", StringType()) \  
 .add("amount", FloatType()) \  
 .add("currency", StringType()) \  
 .add("transaction\_type", StringType()) \  
 .add("merchant\_id", StringType(), True) \  
 .add("category", StringType(), True) \  
 .add("enriched\_category", StringType(), True) \  
 .add("is\_amount\_valid", StringType()) \  
 .add("processing\_timestamp", TimestampType())  
  
 # Select columns in the order of curated\_schema and cast if necessary  
 df\_final = df\_quality\_checked.select([col(c.name).cast(c.dataType) for c in curated\_schema.fields])  
  
  
 print(f"Writing curated data to: {curated\_path}")  
 # Write to Curated Delta Lake (using DeltaTable for upserts if needed, otherwise standard write)  
 if DeltaTable.isDeltaTable(spark, curated\_path):  
 # Example: Simple overwrite for batch, or merge for SCD type operations  
 df\_final.write.format("delta").mode("overwrite").option("overwriteSchema", "true").save(curated\_path)  
 else:  
 df\_final.write.format("delta").mode("overwrite").option("overwriteSchema", "true").save(curated\_path) # Changed to overwrite for simple batch runs  
  
 print("Batch transformation complete.")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if len(sys.argv) != 3: # raw\_path, curated\_path  
 print("Usage: batch\_transformations.py <raw\_delta\_path> <curated\_delta\_path>")  
 sys.exit(-1)  
  
 raw\_path = sys.argv[1]  
 curated\_path = sys.argv[2]  
  
 spark = create\_spark\_session("BatchETLTransformation")  
 spark.sparkContext.setLogLevel("WARN")  
  
 # Get PostgreSQL connection details from environment variables (set in docker-compose.yml for 'spark' service)  
 PG\_HOST = "postgres" # Service name in docker-compose  
 PG\_PORT = "5432"  
 PG\_DB = "main\_db"  
 PG\_USER = "user"  
 PG\_PASSWORD = "password"  
  
 # Create a dummy merchant\_lookup table in PostgreSQL if it doesn't exist  
 # This would typically be part of your database migration  
 try:  
 conn\_str = f"jdbc:postgresql://{PG\_HOST}:{PG\_PORT}/{PG\_DB}"  
 df\_dummy = spark.createDataFrame([(1, "Electronics"), (2, "Groceries"), (3, "Entertainment")], ["merchant\_id", "category"])  
 df\_dummy.write.format("jdbc") \  
 .option("url", conn\_str) \  
 .option("dbtable", "merchant\_lookup") \  
 .option("user", PG\_USER) \  
 .option("password", PG\_PASSWORD) \  
 .mode("overwrite") \  
 .save()  
 print("Dummy 'merchant\_lookup' table ensured in PostgreSQL.")  
 except Exception as e:  
 print(f"Could not ensure dummy merchant\_lookup table: {e}. Assuming it exists or will be created.")  
  
 run\_transformation(spark, raw\_path, curated\_path, PG\_HOST, PG\_PORT, PG\_DB, PG\_USER, PG\_PASSWORD)  
 spark.stop()

Steps to Exercise:

Stop streaming jobs: Stop the previously submitted Spark streaming jobs (Ctrl+C in their terminals, or docker compose stop spark then docker compose start spark). This is important so the batch job can have a consistent snapshot of the raw data.

Submit Batch Job: In a new terminal, submit the batch\_transformations.py job.  
docker exec -it spark spark-submit \  
 --packages io.delta:delta-core\_2.12:2.4.0,org.postgresql:postgresql:42.6.0 \  
 --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\_batch

Monitor: Observe the console output of the spark-submit command.

Verification:

MinIO Console (http://localhost:9001): Navigate to curated-data-bucket. You should see a new financial\_data\_curated\_batch directory containing .parquet files and \_delta\_log.

Spark History Server (http://localhost:18080): The completed batch job should appear. Inspect its details, including the data read and written.

Data Content (Conceptual Query): If you can query the Delta table (e.g., using spark-sql from inside the spark container), verify the enriched\_category and is\_amount\_valid columns are correctly populated.  
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 transaction\_id, amount, currency, enriched\_category, is\_amount\_valid FROM delta.\`s3a://curated-data-bucket/financial\_data\_curated\_batch\` LIMIT 10;"

Advanced Use Case 2: Machine Learning Integration & Feature Engineering

Objective: To demonstrate how Spark can be used for feature engineering on curated data and conceptually apply a machine learning model, highlighting the analytical capabilities of the platform.

Role in Platform: Prepare and serve high-quality features for ML models, and perform large-scale inference.

Setup/Configuration:

Ensure financial\_data\_curated\_batch is populated: From Advanced Use Case 1.

ML Script: Create pyspark\_jobs/ml\_model\_inference.py (as provided in the previous Data Platform Usage Guide). This script will read curated data, perform basic feature engineering (e.g., using VectorAssembler), and conceptually apply an ML model.  
Example pyspark\_jobs/ml\_model\_inference.py (as used before):  
# pyspark\_jobs/ml\_model\_inference.py  
import sys  
from pyspark.sql import SparkSession  
from pyspark.ml.feature import VectorAssembler  
from pyspark.ml.classification import LogisticRegression # Example ML library  
from pyspark.sql.functions import col  
  
def create\_spark\_session(app\_name):  
 """Helper function to create a SparkSession with Delta Lake and MLlib packages."""  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "io.delta:delta-core\_2.12:2.4.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if len(sys.argv) != 2:  
 print("Usage: ml\_model\_inference.py <curated\_delta\_path>")  
 sys.exit(-1)  
  
 curated\_path = sys.argv[1]  
 spark = create\_spark\_session("ML\_Inference\_Example")  
 spark.sparkContext.setLogLevel("WARN")  
  
 print(f"Reading curated data from: {curated\_path}")  
 try:  
 df = spark.read.format("delta").load(curated\_path)  
 df.printSchema()  
 df.show(5, truncate=False)  
  
 # --- Feature Engineering ---  
 # For demonstration, let's create a 'feature' vector from 'amount' and 'is\_amount\_valid'  
 # In a real scenario, this would involve more complex feature selection and transformation  
 feature\_columns = ["amount"]  
 if "is\_amount\_valid" in df.columns: # Conditionally add if exists from previous step  
 feature\_columns.append(col("is\_amount\_valid").cast("double").alias("is\_amount\_valid\_numeric"))  
 # If "is\_amount\_valid" is boolean, cast to double for VectorAssembler  
  
 assembler = VectorAssembler(inputCols=feature\_columns, outputCol="features")  
 feature\_df = assembler.transform(df)  
 print("Schema after feature engineering:")  
 feature\_df.printSchema()  
  
 # --- Conceptual ML Model Application ---  
 # This part is conceptual as we don't have a trained model.  
 # In a real scenario, you would load a pre-trained model:  
 # from pyspark.ml.classification import LogisticRegressionModel  
 # model = LogisticRegressionModel.load("path/to/your/trained\_model")  
 # predictions = model.transform(feature\_df)  
 # predictions.show()  
  
 # For demonstration, we'll just print some summary statistics of the features  
 print("Summary of features for ML:")  
 feature\_df.select("features").show(5, truncate=False)  
 # You could save this prepared feature set for later training/inference  
 # feature\_df.write.format("delta").mode("overwrite").save("s3a://ml-features-bucket/financial\_features")  
  
 print(f"Successfully read data from {curated\_path} and conceptually prepared for ML.")  
 print("In a real scenario, an ML model would now be applied or trained on these features.")  
  
 except Exception as e:  
 print(f"Error reading curated data for ML or during feature engineering: {e}")  
 print("Ensure batch\_transformations.py has run and populated the curated-data-bucket path.")  
  
 spark.stop()

Steps to Exercise:

Submit ML Script: In a new terminal, submit the ml\_model\_inference.py job.  
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/ml\_model\_inference.py \  
 s3a://curated-data-bucket/financial\_data\_curated\_batch

Monitor: Observe the console output for the spark-submit command.

Verification:

Console Output: The script should successfully read from the curated Delta Lake and print the schema with the new features column (a vector of your chosen numerical features). It should also print a summary of the features. This demonstrates Spark's role in preparing data for ML.

Spark History Server (http://localhost:18080): A new completed job for "ML\_Inference\_Example" will appear.

Advanced Use Case 3: Data Lineage Tracking and Schema Enforcement (via Spline & Delta Lake)

Objective: To explicitly demonstrate how Spark, combined with Delta Lake, enforces schema, and how Spline automatically captures and visualizes the lineage of Spark transformations.

Role in Platform: Ensure data quality, provide a single source of truth for schema, and enable full transparency of data flow for governance and debugging.

Setup/Configuration:

Ensure Spline and OpenMetadata are running (Advanced Track setup).

Ensure streaming\_consumer.py is running (Basic Use Case).

Introduce Schema Drift in Producer: Modify simulate\_data.py to temporarily introduce a schema change that violates the expected schema for financial data. For example, change amount from a float to a string for a few messages. Then, revert it back quickly to avoid excessive errors for other jobs.

Original (in simulate\_data.py): "amount": round(random.uniform(1.0, 10000.0), 2),

Temporary change: "amount": "invalid\_amount\_string",

Revert back to original immediately after testing this step.

Ensure Spark Streaming job is running with mergeSchema disabled for this test: This is to explicitly show a failure if schema enforcement is strict. If mergeSchema is always on, it will adapt. For this demo, let's assume streaming\_consumer.py uses outputMode("append") without mergeSchema for a moment, or you are running a specific test job without it.

Steps to Exercise:

Trigger Schema Violation (Temporary):

Modify simulate\_data.py to send a few messages with amount as a string instead of a float.

Run simulate\_data.py for a very short period (e.g., 5-10 seconds) with this modification.

Immediately revert simulate\_data.py back to its correct schema for amount (float)! This is crucial to avoid continuous errors.

Observe Spark Job Behavior:

Watch the logs of your financial\_transactions Spark streaming job.

Expected (if mergeSchema is OFF): Spark will likely throw an error (e.g., AnalysisException: Cannot write unknown type string into float type column ...) and the job might fail or restart, indicating strict schema enforcement.

Expected (if mergeSchema is ON): Spark will likely append data, potentially creating a new column for the string if it detects a new type, or handling nulls if it's a type coercion issue. This demonstrates the flexibility of mergeSchema.

Re-enable/Restart Spark: Ensure your Spark streaming job is running correctly again (e.g., docker compose restart spark if it crashed, or re-submit with correct mergeSchema options if you were toggling them).

Access Spline UI: http://localhost:8081.

View Lineage:

Locate the Spark job that processes raw\_financial\_transactions to financial\_data\_delta.

Click on the job to view its lineage graph.

Utilize: Observe the input (Kafka topic), the Spark transformation node, and the output (Delta Lake table). Spline will capture the schema of both input and output, and detail the operations performed (e.g., from\_json, select, write). This visualizes the schema flow through the pipeline.

Access OpenMetadata UI: http://localhost:8585.

Verify Schema & Lineage in Catalog:

Search for your financial\_data\_delta table.

Go to its Schema tab. Verify the current schema matches what Spark is writing.

Go to its Lineage tab. OpenMetadata should pull the lineage from Spline, providing an end-to-end view of the data's journey, including column-level lineage if Spline captured it.

Utilize: This unified view in OpenMetadata demonstrates how governance teams can audit data flow, understand schema evolution, and pinpoint potential data quality issues by tracing data back to its source transformation.

This use case strongly emphasizes how Spark, Delta Lake, and Spline/OpenMetadata collaborate to provide robust data quality, schema management, and transparent data lineage within your platform.

Highlighting MongoDB: Flexible NoSQL Document Database

MongoDB is a popular open-source NoSQL document database. Unlike traditional relational databases, MongoDB stores data in flexible, JSON-like documents, making it ideal for semi-structured data, rapidly evolving schemas, and scenarios where data models are dynamic or complex. In your data platform, MongoDB can serve as a specialized storage layer for specific application use cases, event logs, or data that doesn't fit neatly into a rigid relational schema.

This guide will demonstrate basic and advanced use cases of MongoDB, leveraging your Advanced Track local environment setup and its integration with Python and Apache Spark.

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: Storing Flexible Semi-Structured Data

Objective: To demonstrate how to store and retrieve semi-structured data into MongoDB, highlighting its ability to accept documents without a predefined, rigid schema.

Role in Platform: Provide a highly flexible and scalable storage solution for data that doesn't conform to strict relational models, such as diverse log events, user profiles with varying attributes, or metadata with dynamic properties.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root. This includes the mongodb service.

Verify MongoDB is accessible: Check Docker logs for the mongodb container (docker compose logs mongodb).

Install MongoDB Shell (mongosh): If not already installed on your host, you might want it for direct interaction. Alternatively, you can use docker exec to access the shell inside the container.

Install mongosh locally: MongoDB Shell Installation

Or execute inside container: docker exec -it mongodb mongosh --authenticationDatabase admin -u root -p password

Steps to Exercise:

Connect to MongoDB:  
mongosh "mongodb://localhost:27017" --authenticationDatabase admin -u root -p password  
  
(Replace localhost with mongodb if running from another container within the Docker network, e.g., from fastapi\_ingestor container via docker exec).

Insert a Simple Document:

Switch to a database (it will be created if it doesn't exist):  
use my\_data\_platform\_db

Insert a document into a collection (also created if it doesn't exist):  
db.financial\_events.insertOne({  
 "event\_id": "EVT-001",  
 "type": "login\_attempt",  
 "user\_id": "USR-XYZ",  
 "timestamp": ISODate("2024-06-14T10:30:00Z"),  
 "ip\_address": "192.168.1.100",  
 "status": "success"  
})

Insert a Document with Different Fields:

Insert another document into the same financial\_events collection, but with different or additional fields:  
db.financial\_events.insertOne({  
 "event\_id": "EVT-002",  
 "type": "failed\_login",  
 "user\_id": "USR-ABC",  
 "timestamp": ISODate("2024-06-14T10:31:00Z"),  
 "ip\_address": "192.168.1.101",  
 "reason": "incorrect\_password",  
 "attempts": 3  
})

Query the Documents:  
db.financial\_events.find().pretty()

Verification:

MongoDB Shell Output: The insertOne commands return success acknowledgments. The find().pretty() command displays both documents, showcasing that MongoDB accepted records with different structures in the same collection without any schema definition errors.

Docker Logs: The mongodb container logs should show successful insert operations.

Advanced Use Case 1: Dynamic Schema and Flexible Data Models

Objective: To explicitly demonstrate MongoDB's schemaless nature by inserting and querying documents with highly varying structures within the same collection, which is challenging for relational databases.

Role in Platform: Accommodate rapidly changing data formats, store diverse IoT data, social media feeds, or complex nested documents where a rigid schema is impractical or constantly evolving.

Setup/Configuration:

Ensure MongoDB is running (as per Basic Use Case).

Steps to Exercise:

Connect to MongoDB Shell:  
mongosh "mongodb://localhost:27017" --authenticationDatabase admin -u root -p password  
use dynamic\_insurance\_data

Insert a "Claim" Document (with nested object, array):  
db.claims.insertOne({  
 "claim\_id": "C-001",  
 "policy\_id": "P-1001",  
 "claim\_date": ISODate("2024-05-15T00:00:00Z"),  
 "status": "pending",  
 "details": {  
 "type": "auto",  
 "incident\_date": ISODate("2024-05-10T14:00:00Z"),  
 "vehicle\_vin": "ABC123XYZ456",  
 "damages": ["front\_bumper", "headlight"]  
 },  
 "attachments": [  
 {"filename": "photo1.jpg", "type": "image"},  
 {"filename": "report.pdf", "type": "document"}  
 ]  
})

Insert a "Customer Profile" Document (with a different structure):  
db.customer\_profiles.insertOne({  
 "customer\_id": "CUST-001",  
 "name": "Alice Smith",  
 "contact": {  
 "email": "alice@example.com",  
 "phone": "555-1234"  
 },  
 "addresses": [  
 {"street": "123 Main St", "city": "Anytown", "zip": "12345", "type": "billing"},  
 {"street": "456 Oak Ave", "city": "Anytown", "zip": "12345", "type": "shipping"}  
 ],  
 "preferences": {  
 "newsletter": true,  
 "product\_updates": false  
 }  
})  
  
Note: We used a different collection (customer\_profiles) here, but the principle of flexible schema applies within a single collection as well, by simply inserting documents with varying fields.

Query Documents with Specific Fields:  
db.claims.find({"details.type": "auto"}).pretty()  
db.customer\_profiles.find({"addresses.city": "Anytown"}).pretty()  
  
This demonstrates querying into nested structures and arrays.

Verification:

MongoDB Shell Output: The documents are successfully inserted and retrieved despite their different, complex structures. Queries on nested fields work as expected. This proves MongoDB's flexibility with dynamic schemas and its ability to handle varied JSON-like data.

Advanced Use Case 2: Integration with Spark for Complex Document Processing

Objective: To demonstrate how Apache Spark can efficiently read and process semi-structured data directly from MongoDB collections, perform transformations (e.g., flatten nested fields, filter), and then write the processed data to a structured format like Delta Lake.

Role in Platform: Enable powerful distributed analytics and ETL on data stored in MongoDB, bridging the gap between flexible document storage and structured data lake analysis.

Setup/Configuration:

Ensure MongoDB and Spark are running.

Install Spark-MongoDB Connector: Ensure your Spark Docker image (or the spark-submit command) includes the MongoDB Spark Connector package. (e.g., org.mongodb.spark:mongo-spark-connector\_2.12:10.0.0).

Populate MongoDB: Ensure my\_data\_platform\_db.financial\_events (from Basic Use Case) or dynamic\_insurance\_data.claims (from Adv Use Case 1) has some data.

Spark Processing Script: Create a Python script pyspark\_jobs/mongo\_processor.py.  
Example pyspark\_jobs/mongo\_processor.py (conceptual):  
# pyspark\_jobs/mongo\_processor.py  
import sys  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import col, explode, current\_timestamp  
  
def create\_spark\_session(app\_name):  
 """Helper function to create a SparkSession with Delta Lake and MongoDB packages."""  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "io.delta:delta-core\_2.12:2.4.0,org.mongodb.spark:mongo-spark-connector\_2.12:10.0.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if len(sys.argv) != 4:  
 print("Usage: mongo\_processor.py <mongo\_db\_name> <mongo\_collection\_name> <delta\_output\_path>")  
 sys.exit(-1)  
  
 mongo\_db\_name = sys.argv[1]  
 mongo\_collection\_name = sys.argv[2]  
 delta\_output\_path = sys.argv[3]  
  
 spark = create\_spark\_session(f"MongoDBToDelta\_{mongo\_collection\_name}")  
 spark.sparkContext.setLogLevel("WARN")  
  
 # MongoDB connection URI (use 'mongodb' service name in Docker Compose)  
 mongo\_uri = "mongodb://root:password@mongodb:27017/"  
  
 # Read data from MongoDB  
 print(f"Reading data from MongoDB: db={mongo\_db\_name}, collection={mongo\_collection\_name}")  
 df\_mongo = (spark.read.format("mongodb")  
 .option("spark.mongodb.input.uri", f"{mongo\_uri}{mongo\_db\_name}.{mongo\_collection\_name}")  
 .load())  
  
 print("Schema of data read from MongoDB:")  
 df\_mongo.printSchema()  
 df\_mongo.show(5, truncate=False)  
  
 # --- Example Transformation: Flattening nested 'details' and 'attachments' for 'claims' ---  
 # Assuming the 'claims' collection with nested structures  
 if mongo\_collection\_name == "claims":  
 print("Applying transformations for claims data...")  
 # Select relevant fields and flatten nested 'details' object  
 df\_transformed = df\_mongo.select(  
 col("claim\_id"),  
 col("policy\_id"),  
 col("claim\_date"),  
 col("status"),  
 col("details.type").alias("claim\_type"),  
 col("details.incident\_date").alias("incident\_date"),  
 col("details.vehicle\_vin").alias("vehicle\_vin"),  
 explode(col("details.damages")).alias("damage\_item"), # Explode damages array  
 current\_timestamp().alias("processing\_timestamp")  
 )  
 print("Schema after flattening:")  
 df\_transformed.printSchema()  
 df\_transformed.show(5, truncate=False)  
  
 elif mongo\_collection\_name == "financial\_events":  
 print("Applying transformations for financial events data...")  
 df\_transformed = df\_mongo.select(  
 col("event\_id"),  
 col("type"),  
 col("user\_id"),  
 col("timestamp"),  
 col("ip\_address"),  
 col("status"),  
 col("reason").alias("failure\_reason"), # Handle optional field  
 col("attempts").alias("login\_attempts"), # Handle optional field  
 current\_timestamp().alias("processing\_timestamp")  
 )  
 print("Schema after selecting/renaming:")  
 df\_transformed.printSchema()  
 df\_transformed.show(5, truncate=False)  
 else:  
 print("No specific transformation defined for this collection. Writing as is.")  
 df\_transformed = df\_mongo.withColumn("processing\_timestamp", current\_timestamp())  
  
  
 # Write the processed data to Delta Lake in MinIO  
 print(f"Writing transformed data to Delta Lake: {delta\_output\_path}")  
 df\_transformed.write.format("delta").mode("overwrite").option("overwriteSchema", "true").save(delta\_output\_path)  
 print("Transformation and write to Delta Lake complete.")  
  
 spark.stop()

Steps to Exercise:

Populate MongoDB: Ensure you have inserted sample data into my\_data\_platform\_db.financial\_events (or dynamic\_insurance\_data.claims) as per Basic/Advanced Use Case 1.

Submit Spark Job: In a new terminal, submit the mongo\_processor.py job.

For financial\_events:  
docker exec -it spark spark-submit \  
 --packages io.delta:delta-core\_2.12:2.4.0,org.mongodb.spark:mongo-spark-connector\_2.12:10.0.0 \  
 --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/mongo\_processor.py \  
 my\_data\_platform\_db financial\_events s3a://curated-data-bucket/mongo\_financial\_events\_curated

For claims (if populated):  
docker exec -it spark spark-submit \  
 --packages io.delta:delta-core\_2.12:2.4.0,org.mongodb.spark:mongo-spark-connector\_2.12:10.0.0 \  
 --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/mongo\_processor.py \  
 dynamic\_insurance\_data claims s3a://curated-data-bucket/mongo\_insurance\_claims\_curated

Monitor: Observe the console output of the spark-submit command, looking for schema inference and transformation details.

Verification:

Spark Logs: The logs will show Spark successfully connecting to MongoDB, reading the documents, applying the transformations, and writing to the specified Delta Lake path.

MinIO Console (http://localhost:9001): Navigate to curated-data-bucket. You should see new directories (mongo\_financial\_events\_curated or mongo\_insurance\_claims\_curated) containing .parquet files and a \_delta\_log, confirming the data has been processed and stored.

Query Processed Data (Spark-SQL):  
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 \* FROM delta.\`s3a://curated-data-bucket/mongo\_financial\_events\_curated\` LIMIT 5;"  
  
(Or for claims: mongo\_insurance\_claims\_curated). The query results will show the flattened and transformed data, ready for further analysis.

Advanced Use Case 3: Scalability and High Availability (Conceptual)

Objective: To conceptually demonstrate MongoDB's native support for horizontal scalability through sharding and high availability through replica sets, which are critical for production deployments handling large data volumes and high query loads.

Role in Platform: Ensure the MongoDB layer can scale to accommodate growing data volumes and query throughput, and provide continuous availability even in the face of node failures.

Setup/Configuration (Conceptual Discussion):

Replica Sets:

In a production environment, MongoDB is typically deployed as a replica set, which is a group of mongod processes that maintain the same data set. This provides redundancy and high availability.

Your docker-compose.yml likely runs a single MongoDB instance for local development simplicity. To run a replica set locally, you'd need multiple mongodb services and an explicit replicaSet configuration in their command.  
Example docker-compose.yml snippet (conceptual, for a 3-node replica set):  
# services:  
# mongo1:  
# image: mongo:latest  
# command: mongod --replSet rs0 --bind\_ip 0.0.0.0  
# volumes:  
# - mongo\_data1:/data/db  
# ports:  
# - "27017:27017"  
# mongo2:  
# image: mongo:latest  
# command: mongod --replSet rs0 --bind\_ip 0.0.0.0  
# volumes:  
# - mongo\_data2:/data/db  
# mongo3:  
# image: mongo:latest  
# command: mongod --replSet rs0 --bind\_ip 0.0.0.0  
# volumes:  
# - mongo\_data3:/data/db  
# # ... then initiate the replica set via mongosh on one node:  
# # rs.initiate({ \_id: "rs0", members: [{ \_id: 0, host: "mongo1:27017" }, { \_id: 1, host: "mongo2:27017" }, { \_id: 2, host: "mongo3:27017" }]})

Sharding:

For horizontal scalability beyond what a single replica set can offer (e.g., handling petabytes of data or millions of operations per second), MongoDB supports sharding. Sharding distributes data across multiple shard clusters, each acting as an independent replica set.

This setup involves: config servers (metadata), mongos router (query routing), and shard replica sets. This is significantly more complex to simulate locally in Docker Compose and is typically a production-only architecture.

Steps to Exercise (Conceptual Discussion):

Discuss Replica Set Benefits:

Automatic Failover: If the primary node fails, an election occurs, and a new primary is chosen, ensuring continuous operation.

Data Redundancy: Multiple copies of data protect against data loss.

Read Scalability: Reads can be distributed across secondary nodes, offloading the primary.

Discuss Sharding Benefits:

Horizontal Scalability: Distribute data and load across many servers, allowing the cluster to grow almost indefinitely.

High Throughput: Handle massive query volumes by parallelizing operations across shards.

Large Data Sets: Store datasets that exceed the capacity of a single server.

Explain Local vs. Production: Emphasize that while your docker-compose.yml provides a single mongodb instance for development convenience, a production deployment would always involve a replica set (at least three nodes) and potentially sharding for extreme scale.

Verification (Conceptual):

Understanding the architectural patterns of MongoDB replica sets and sharding demonstrates a grasp of how the database layer contributes to the overall scalability and high availability of the data platform in a production environment. This directly translates to an AWS Engineer's understanding of services like Amazon DocumentDB, which manages these underlying complexities.

This concludes the guide for MongoDB.

Highlighting OpenTelemetry: Standardized Telemetry Collection

OpenTelemetry is a vendor-neutral set of open-source tools, APIs, and SDKs that standardize the collection and export of telemetry data – metrics, logs, and traces – from your software applications. In your data platform, OpenTelemetry is pivotal for achieving deep, consistent observability across all services, regardless of their underlying technology. It enables unified monitoring, tracing, and logging, essential for understanding system behavior and troubleshooting issues.

This guide will demonstrate basic and advanced use cases of OpenTelemetry, leveraging your Advanced Track local environment setup and its integration with Grafana Alloy and Grafana.

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, specifically emphasizing OpenTelemetry's role in the Observability section and the "Highlighting Grafana Alloy" document.

Basic Use Case: Instrumenting an Application for Metrics Collection

Objective: To demonstrate how to instrument a Python application (FastAPI) to emit application-specific metrics using OpenTelemetry, and how these metrics are then collected by Grafana Alloy and visualized in Grafana.

Role in Platform: Enable collection of custom, application-level metrics (e.g., request counts, latency, business-specific events) from services, providing granular insights beyond basic infrastructure metrics.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root. This includes fastapi\_ingestor, grafana-alloy, and grafana.

Install OpenTelemetry Python SDK and Exporters: Your fastapi\_app/requirements.txt should include necessary OpenTelemetry packages.  
# fastapi\_app/requirements.txt  
fastapi  
uvicorn  
python-dotenv  
kafka-python  
pydantic  
# OpenTelemetry packages  
opentelemetry-api  
opentelemetry-sdk  
opentelemetry-exporter-otlp  
opentelemetry-instrumentation-fastapi  
opentelemetry-instrumentation-requests  
opentelemetry-sdk-extension-aws  
opentelemetry-distro  
opentelemetry-instrumentation-logging # For logs  
opentelemetry-sdk-metrics # For custom metrics

Instrument FastAPI application: Modify fastapi\_app/app/main.py to initialize OpenTelemetry and configure it to export metrics to Grafana Alloy.  
Example fastapi\_app/app/main.py (conceptual additions):  
# fastapi\_app/app/main.py  
import os  
import json  
from datetime import datetime  
from typing import Optional  
  
from fastapi import FastAPI, HTTPException, status  
from pydantic import BaseModel, Field  
from kafka import KafkaProducer  
  
# --- OpenTelemetry Imports and Setup ---  
from opentelemetry import metrics  
from opentelemetry import trace  
from opentelemetry.sdk.resources import Resource  
from opentelemetry.sdk.trace import TracerProvider  
from opentelemetry.sdk.metrics import MeterProvider  
from opentelemetry.sdk.metrics.export import (  
 ConsoleMetricExporter,  
 PeriodicExportingMetricReader,  
)  
from opentelemetry.exporter.otlp.proto.http.trace\_exporter import OTLPSpanExporter  
from opentelemetry.exporter.otlp.proto.http.metric\_exporter import OTLPMetricExporter  
from opentelemetry.instrumentation.fastapi import FastAPIInstrumentor  
from opentelemetry.instrumentation.logging import LoggingInstrumentor # For logs  
  
  
# Resource for identifying the service  
resource = Resource.create({  
 "service.name": "fastapi-ingestor",  
 "service.version": "1.0.0",  
 "env.type": "local-dev"  
})  
  
# Configure OTLP Exporter endpoint (Grafana Alloy)  
# This should match the otelcol.receiver.otlp config in alloy-config.river  
OTEL\_EXPORTER\_OTLP\_ENDPOINT = os.getenv("OTEL\_EXPORTER\_OTLP\_ENDPOINT", "http://grafana-alloy:4318")  
  
# Metrics Provider (for custom metrics)  
metric\_reader = PeriodicExportingMetricReader(  
 OTLPMetricExporter(endpoint=f"{OTEL\_EXPORTER\_OTLP\_ENDPOINT}/v1/metrics")  
)  
meter\_provider = MeterProvider(resource=resource, metric\_readers=[metric\_reader])  
metrics.set\_meter\_provider(meter\_provider)  
meter = metrics.get\_meter("fastapi.ingestion.app")  
  
# Create counters for business metrics  
financial\_tx\_counter = meter.create\_counter(  
 "financial.transactions.ingested",  
 description="Number of financial transactions ingested",  
 unit="1"  
)  
insurance\_claim\_counter = meter.create\_counter(  
 "insurance.claims.ingested",  
 description="Number of insurance claims ingested",  
 unit="1"  
)  
  
# Tracer Provider (for distributed tracing)  
trace\_exporter = OTLPSpanExporter(endpoint=f"{OTEL\_EXPORTER\_OTLP\_ENDPOINT}/v1/traces")  
trace.set\_tracer\_provider(TracerProvider(resource=resource))  
trace.get\_tracer\_provider().add\_span\_processor(  
 BatchSpanProcessor(trace\_exporter)  
)  
  
# Instrument logging to include trace/span IDs  
LoggingInstrumentor().instrument(set\_logging\_format=True)  
  
# --- Pydantic Models and FastAPI App Init (as before) ---  
class FinancialTransaction(BaseModel):  
 transaction\_id: str = Field(..., example="FT-20231026-001")  
 timestamp: datetime = Field(..., example="2023-10-26T14:30:00Z")  
 account\_id: str = Field(..., example="ACC-001")  
 amount: float = Field(..., gt=0, example=150.75)  
 currency: str = Field(..., max\_length=3, example="USD")  
 transaction\_type: str = Field(..., example="debit")  
 merchant\_id: Optional[str] = Field(None, example="MER-XYZ")  
 category: Optional[str] = Field(None, example="groceries")  
  
class InsuranceClaim(BaseModel):  
 claim\_id: str = Field(..., example="IC-20231026-001")  
 timestamp: datetime = Field(..., example="2023-10-26T15:00:00Z")  
 policy\_number: str = Field(..., example="POL-987654")  
 claim\_amount: float = Field(..., gt=0, example=1000.00)  
 claim\_type: str = Field(..., example="auto")  
 claim\_status: str = Field(..., example="submitted")  
 customer\_id: str = Field(..., example="CUST-ABC")  
 incident\_date: datetime = Field(..., example="2023-09-15T08:00:00Z")  
  
app = FastAPI(  
 title="Financial/Insurance Data Ingestor API",  
 description="API for ingesting various financial and insurance data into the data platform.",  
 version="1.0.0",  
)  
  
# Instrument FastAPI application with OpenTelemetry  
FastAPIInstrumentor.instrument\_app(app)  
  
# --- Kafka Producer Setup (as before) ---  
KAFKA\_BROKER = os.getenv("KAFKA\_BROKER", "kafka:29092")  
KAFKA\_TOPIC\_FINANCIAL = os.getenv("KAFKA\_TOPIC\_FINANCIAL", "raw\_financial\_transactions")  
KAFKA\_TOPIC\_INSURANCE = os.getenv("KAFKA\_TOPIC\_INSURANCE", "raw\_insurance\_claims")  
  
try:  
 producer = KafkaProducer(  
 bootstrap\_servers=[KAFKA\_BROKER],  
 value\_serializer=lambda v: json.dumps(v).encode('utf-8'),  
 retries=5,  
 linger\_ms=100,  
 batch\_size=16384  
 )  
 print(f"Kafka Producer initialized for broker: {KAFKA\_BROKER}")  
except Exception as e:  
 print(f"Error initializing Kafka Producer: {e}")  
 producer = None  
  
# --- API Endpoints ---  
@app.get("/health", tags=["Monitoring"])  
async def health\_check():  
 return {"status": "healthy", "message": "Welcome to Financial/Insurance Data Ingestor API!"}  
  
@app.post("/ingest-financial-transaction/", status\_code=status.HTTP\_200\_OK, tags=["Ingestion"])  
async def ingest\_financial\_transaction(transaction: FinancialTransaction):  
 try:  
 if producer:  
 producer.send(KAFKA\_TOPIC\_FINANCIAL, transaction.dict())  
 print(f"Financial transaction ingested and sent to Kafka topic '{KAFKA\_TOPIC\_FINANCIAL}': {transaction.transaction\_id}")  
 else:  
 print("Kafka producer not available. Skipping send.")  
  
 # Increment custom metric  
 financial\_tx\_counter.add(1, {"transaction.type": transaction.transaction\_type, "currency": transaction.currency})  
  
 return {"message": "Financial transaction ingested successfully", "transaction\_id": transaction.transaction\_id}  
 except Exception as e:  
 raise HTTPException(status\_code=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR, detail=f"Failed to ingest transaction: {e}")  
  
@app.post("/ingest-insurance-claim/", status\_code=status.HTTP\_200\_OK, tags=["Ingestion"])  
async def ingest\_insurance\_claim(claim: InsuranceClaim):  
 try:  
 if producer:  
 producer.send(KAFKA\_TOPIC\_INSURANCE, claim.dict())  
 print(f"Insurance claim ingested and sent to Kafka topic '{KAFKA\_TOPIC\_INSURANCE}': {claim.claim\_id}")  
 else:  
 print("Kafka producer not available. Skipping send.")  
  
 # Increment custom metric  
 insurance\_claim\_counter.add(1, {"claim.type": claim.claim\_type, "claim.status": claim.claim\_status})  
  
 return {"message": "Insurance claim ingested successfully", "claim\_id": claim.claim\_id}  
 except Exception as e:  
 raise HTTPException(status\_code=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR, detail=f"Failed to ingest claim: {e}")  
  
Note: You'll need to add opentelemetry-sdk-metrics to your fastapi\_app/requirements.txt to run this. Also, for BatchSpanProcessor you'd need to from opentelemetry.sdk.trace.export import BatchSpanProcessor.

Configure Grafana Alloy: Ensure observability/alloy-config.river has an otelcol.receiver.otlp component to receive metrics, traces, and logs from FastAPI, forwarding them to Grafana (acting as a Prometheus remote write endpoint).  
Example observability/alloy-config.river (relevant snippet for OTLP receiver):  
# observability/alloy-config.river  
# ...  
prometheus.remote\_write "default" {  
 url = "http://grafana:9090/api/prom/push"  
}  
  
otelcol.receiver.otlp "default" {  
 http { } # Listen for OTLP HTTP  
 grpc { } # Listen for OTLP gRPC  
 output {  
 metrics = [prometheus.remote\_write.default.receiver]  
 # Traces and logs would go to other exporters/receivers if configured  
 traces = [] # For this basic case, we might not forward traces/logs yet  
 logs = []  
 }  
}  
# ...

Steps to Exercise:

Rebuild and Restart FastAPI and Grafana Alloy:  
docker compose up --build -d fastapi\_ingestor grafana-alloy

Generate data: Run python3 simulate\_data.py. This will send requests to the FastAPI ingestor.

Access Grafana: Go to http://localhost:3000.

Query Custom Metrics:

Open the "Explore" view and select your Prometheus data source.

Enter PromQL queries for the new custom metrics:

financial\_transactions\_ingested\_total

insurance\_claims\_ingested\_total

You can also filter by attributes:

financial\_transactions\_ingested\_total{transaction\_type="purchase"}

insurance\_claims\_ingested\_total{claim\_type="auto"}

Verification:

Grafana: The custom metrics (financial\_transactions\_ingested\_total, insurance\_claims\_ingested\_total) will appear in Grafana, and their values will increase as simulate\_data.py sends data. The attributes (e.g., transaction\_type, claim\_type) will also be visible as labels, demonstrating successful OpenTelemetry instrumentation for metrics.

Advanced Use Case 1: Distributed Tracing for End-to-End Latency

Objective: To demonstrate how OpenTelemetry automatically propagates trace context across service boundaries (e.g., FastAPI calling Kafka, and conceptually, Kafka triggering a Spark job), allowing for end-to-end tracing and bottleneck identification.

Role in Platform: Provide deep visibility into the entire data flow path, helping to pinpoint latency bottlenecks and error origins across microservices and distributed processing stages.

Setup/Configuration:

Ensure Basic Use Case setup is complete (FastAPI is instrumented for traces as shown in the basic setup's main.py).

Ensure Grafana Alloy is configured to forward traces:

You'll need a trace backend. For local setup, we can conceptually demonstrate by configuring Alloy to forward to a dummy endpoint or a simple local Jaeger instance (if you have one). If you don't have a Jaeger instance, we'll confirm trace export from FastAPI.

Example observability/alloy-config.river (additions for traces):# ...  
# 1. Define an OTLP receiver for traces  
otelcol.receiver.otlp "default" { # This receiver is already configured in basic setup  
 http { }  
 grpc { }  
 output {  
 metrics = [prometheus.remote\_write.default.receiver]  
 traces = [otelcol.exporter.otlp.jaeger\_mock.input] # Forward traces to a specific exporter  
 logs = []  
 }  
}  
  
# 2. Define an OTLP exporter for traces (to a conceptual Jaeger/Tempo endpoint or just logs)  
# For a real Jaeger, uncomment this in docker-compose.yml and add here:  
# otelcol.exporter.otlp "jaeger\_mock" {  
# client {  
# endpoint = "http://jaeger-all-in-one:4318" # Conceptual Jaeger/Tempo endpoint  
# }  
# }  
# OR, for a simple local demo without Jaeger UI, you can send to console  
otelcol.exporter.logging "trace\_logger" {  
 log\_level = "debug"  
 output {  
 traces = [otelcol.exporter.logging.trace\_logger.input]  
 }  
}  
# Then change otelcol.receiver.otlp -> traces = [otelcol.exporter.logging.trace\_logger.input]  
# ...  
Note: To fully visualize traces, you'd need a Jaeger or Tempo UI. For this local demo, we'll primarily observe that traces are being sent by FastAPI and received/forwarded by Grafana Alloy.

Steps to Exercise:

Ensure FastAPI is configured to export traces (as shown in basic setup).

Ensure Grafana Alloy is configured to receive and forward traces.

Restart affected services: docker compose up --build -d fastapi\_ingestor grafana-alloy.

Generate API calls: Run python3 simulate\_data.py.

Observe Grafana Alloy Logs:  
docker compose logs -f grafana-alloy  
  
You should see messages indicating that Grafana Alloy is receiving OTLP trace data (spans) from fastapi-ingestor.  
If you configured otelcol.exporter.logging for traces, you'll see detailed JSON representations of the spans in Alloy's logs.

Conceptual Trace Visualization (if Jaeger is integrated):

If you had Jaeger running (http://localhost:16686), you would search for traces related to fastapi-ingestor service.

Expected: You would see individual traces, each representing an API request, with spans showing the duration of the HTTP request, Kafka message production, and potentially downstream Spark processing (if Spark was also instrumented).

Verification:

Grafana Alloy Logs: Logs confirm that Grafana Alloy is successfully receiving and forwarding trace data from the instrumented FastAPI application. This demonstrates OpenTelemetry's capability to collect distributed traces, which are critical for debugging end-to-end performance and errors in complex data pipelines.

Advanced Use Case 2: Structured Logging and Contextualization

Objective: To demonstrate how OpenTelemetry integrates with standard logging frameworks to automatically inject trace and span IDs into log messages, enabling easier correlation of logs with specific requests and traces.

Role in Platform: Enhance debuggability by linking application logs to distributed traces, providing rich context for troubleshooting complex data flow issues.

Setup/Configuration:

Ensure Basic Use Case setup is complete (FastAPI has LoggingInstrumentor().instrument(set\_logging\_format=True)).

Ensure Grafana Alloy is configured to receive logs: Add an otelcol.exporter.logging for logs in Alloy's config, or if you have a Loki instance, configure otelcol.exporter.loki.  
Example fastapi\_app/app/main.py (logging setup, from basic setup):  
# ...  
from opentelemetry.instrumentation.logging import LoggingInstrumentor  
# ...  
LoggingInstrumentor().instrument(set\_logging\_format=True)  
# ...  
  
Example observability/alloy-config.river (additions for logs):  
# ...  
otelcol.receiver.otlp "default" {  
 http { }  
 grpc { }  
 output {  
 metrics = [prometheus.remote\_write.default.receiver]  
 traces = [] # Keep traces as before, or comment out if not using  
 logs = [otelcol.exporter.logging.log\_printer.input] # Forward logs to a logging exporter  
 }  
}  
  
# Exporter to print logs to Alloy's stdout  
otelcol.exporter.logging "log\_printer" {  
 log\_level = "debug"  
}  
# ...

Steps to Exercise:

Rebuild and Restart services: docker compose up --build -d fastapi\_ingestor grafana-alloy.

Generate API calls: Run python3 simulate\_data.py.

Observe FastAPI Logs (Docker Compose):  
docker compose logs -f fastapi\_ingestor  
  
Look for log messages emitted by your FastAPI application.

Observe Grafana Alloy Logs:  
docker compose logs -f grafana-alloy  
  
You should see the logs forwarded by FastAPI appearing in Grafana Alloy's stdout (because of log\_printer exporter).

Verification:

Logs Content: Log messages from FastAPI (in both its own container logs and Grafana Alloy's logs if forwarded) will contain additional fields like trace\_id and span\_id. These IDs will correspond to the traces generated for the respective API requests. This demonstrates how OpenTelemetry enriches logs with tracing context, making it easier to connect log events to specific request flows in a distributed system.

Advanced Use Case 3: Custom Metric Types and Attributes for Business Monitoring

Objective: To demonstrate how to define and record custom OpenTelemetry metrics (beyond basic counters, e.g., gauges or histograms) with rich attributes, allowing for deep business-level monitoring of your data platform.

Role in Platform: Collect domain-specific business metrics (e.g., "number of fraudulent transactions detected", "insurance claim processing duration", "data quality validation success rate"), providing insights directly relevant to business value and data quality.

Setup/Configuration:

Ensure Basic Use Case setup is complete (FastAPI is instrumented and sending basic counters).

Modify fastapi\_app/app/main.py: Add a histogram metric to track the processing duration of ingestion requests or a gauge to track queue sizes. Add more specific attributes to existing counters.  
Example fastapi\_app/app/main.py (further conceptual additions to existing meter):  
# ... existing OpenTelemetry setup ...  
  
# Add a Histogram to track ingestion latency  
ingestion\_latency\_histogram = meter.create\_histogram(  
 "ingestion.request.duration",  
 description="Duration of data ingestion requests",  
 unit="ms",  
 # Explicitly define buckets for better granularity in Grafana  
 # Recommended to use power-of-2 values for buckets  
 boundaries=[0.01, 0.05, 0.1, 0.2, 0.5, 1.0, 2.5, 5.0, 10.0, 20.0, 50.0, 100.0, 200.0, 500.0, 1000.0, 2000.0, 5000.0, 10000.0]  
)  
  
# --- inside ingest\_financial\_transaction endpoint ---  
# ...  
 import time  
 start\_time = time.time()  
 # ... existing Kafka send logic ...  
 end\_time = time.time()  
 duration\_ms = (end\_time - start\_time) \* 1000  
 ingestion\_latency\_histogram.record(duration\_ms, {"endpoint": "/ingest-financial-transaction", "status": "success"})  
# ... similar for insurance claims ...  
  
Note: Actual FastAPIInstrumentor already captures HTTP request durations. This example shows adding a custom duration metric for specific internal logic if needed.

Steps to Exercise:

Rebuild and Restart FastAPI: docker compose up --build -d fastapi\_ingestor.

Generate API calls: Run python3 simulate\_data.py.

Access Grafana: Go to http://localhost:3000.

Query Custom Metrics with Attributes:

Open the "Explore" view and select your Prometheus data source.

For the custom counters, use sum by (transaction\_type, currency) (financial\_transactions\_ingested\_total).

For the histogram, query rate(ingestion\_request\_duration\_bucket{endpoint="/ingest-financial-transaction"}[1m]) or histogram\_quantile(0.99, sum by(le, endpoint) (rate(ingestion\_request\_duration\_bucket[1m]))).

Observe: The graphs will show the totals segmented by the attributes you defined (e.g., total financial transactions by transaction\_type and currency). The histogram will provide insights into the distribution of your ingestion latency.

Verification:

Grafana Dashboards: The custom metrics appear with their associated attributes as labels. You can create panels that display these metrics aggregated or filtered by the attributes, demonstrating OpenTelemetry's ability to capture rich, multi-dimensional business and operational data. This is crucial for building comprehensive dashboards that provide insights into business process performance and data quality.

This concludes the guide for OpenTelemetry.

Data Platform Local Environment Walkthrough & QA Test Suite

This document provides a step-by-step learning resource and a practical QA test suite to set up and validate a robust local data platform environment. By following these steps and executing the test cases, you will gain hands-on experience with key technologies and develop a deeper understanding of the platform's functionality and operational aspects. This version introduces additional complexity to demonstrate the system's flexibility in handling different data types and its scalability.

This walkthrough leverages the architectural principles, setup guides, and best practices detailed in the "Building Enterprise-Ready Data Platforms: Core Handbook" and its associated Deep-Dive Addendums.

1. General Setup: Laying the Foundation

These steps are foundational and apply to all testing activities.

1.1. Prerequisites Installation

Action: Ensure your local machine has the necessary software installed.

Install Docker Desktop (or Docker Engine if on Linux).

Install Git.

Install Python 3.x with pip.

Verify docker-compose is installed (usually included with Docker Desktop, or install separately if not).

1.2. Project Repository Setup

Action: Clone the project mono-repo, which contains all necessary code and configuration files.

Navigate to your desired development directory in your terminal.

Execute: git clone <your-repo-url>/data-ingestion-platform

Change into the cloned directory: cd data-ingestion-platform

2. Local Environment Setup: The Progressive Path

This section guides you through building the local data platform incrementally, mirroring the "Progressive Complexity Path" outlined in the Core Handbook. Each track builds upon the previous one.

Reference: For detailed docker-compose.yml configurations and specific instructions, refer to the Progressive Path Setup Guide Deep-Dive Addendum.

2.1. Starter Track Setup: Minimal Single-Machine Setup

Purpose: Understand foundational data ingestion and structured storage.

Components: FastAPI (Ingestor), PostgreSQL, MinIO (S3 compatible data lake).

Setup Steps:

Configure docker-compose.yml:

Open the docker-compose.yml file in the project root.

Uncomment the services for fastapi\_ingestor, postgres, and minio.

Comment out all other services (Kafka, Spark, Airflow, etc.) to keep the setup minimal.

Ensure the data/postgres and data/minio directories exist in your project root for persistent volumes (Docker will create them if they don't).

Bring Up Services:

Execute the onboard.sh script (from Progressive Path Setup Guide Deep-Dive Addendum) or manually run: docker compose up --build -d

Initial Verification:

Access FastAPI health check: http://localhost:8000/health (Expected: HTTP 200 OK with a success message).

Access MinIO Console: http://localhost:9001 (Login with minioadmin/minioadmin). Expected: Console loads successfully.

Connect to PostgreSQL: Use a client (e.g., psql) to connect to localhost:5432 with user user, password password, database main\_db. Expected: Connection successful, basic tables are present (if migrations run automatically).

Check Docker logs for all services: docker compose logs -f (Expected: No critical errors, services show healthy startup messages).

2.2. Intermediate Track Setup: Adding Streaming Capabilities

Purpose: Introduce real-time data streams and distributed transformations.

Components (in addition to Starter): Apache Kafka, Apache Spark.

Setup Steps:

Configure docker-compose.yml:

Open docker-compose.yml.

Uncomment (or keep uncommented) fastapi\_ingestor, postgres, minio.

Uncomment the services for zookeeper, kafka, and spark (and optionally spark-history-server).

Comment out other Advanced Track services.

Review fastapi\_ingestor's environment variables to ensure it publishes to multiple Kafka topics (e.g., KAFKA\_BROKER: kafka:29092, KAFKA\_TOPIC\_FINANCIAL, KAFKA\_TOPIC\_INSURANCE).

Verify spark service is configured to connect to Kafka and MinIO.

Ensure data/spark-events exists for Spark history server logs.

Note on Spark Scalability (Local Simulation): In a local Docker Compose environment, "more Spark nodes" is simulated by allocating more resources (cores, memory) to the single spark service or by running multiple spark-worker services if configured. The conceptual docker-compose.yml and Spark job submissions will imply this parallelism.

Bring Up Services:

Run the onboard.sh script again or docker compose up --build -d. The onboard.sh script should now be updated to initialize both raw\_financial\_transactions and raw\_insurance\_claims Kafka topics.

Initial Verification:

Verify Starter Track components are running and healthy.

Check Kafka topic creation: docker exec -it kafka kafka-topics --bootstrap-server localhost:9092 --list (Expected: Both raw\_financial\_transactions and raw\_insurance\_claims topics are listed).

Check Spark History Server: http://localhost:18080 (if enabled). Expected: Spark History Server UI is accessible.

Generate External Data:

Run the simulate\_data.py script (from Progressive Path Setup Guide Deep-Dive Addendum). This script is designed to continuously send mock financial transactions and insurance claims to the FastAPI Ingestor, which in turn publishes them to their respective Kafka topics.

python3 simulate\_data.py

Trigger Spark Jobs:

Manually submit two separate Spark streaming jobs from the spark container: one to consume from the financial Kafka topic and write to a financial Delta Lake path, and another for insurance data.

For Financial Data: docker exec -it 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 pyspark\_jobs/streaming\_consumer.py raw\_financial\_transactions kafka:29092 s3a://raw-data-bucket/financial\_data\_delta

For Insurance Data: docker exec -it 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 pyspark\_jobs/streaming\_consumer.py raw\_insurance\_claims kafka:29092 s3a://raw-data-bucket/insurance\_data\_delta

2.3. Advanced Track Setup: The Full Production-Ready Stack

Purpose: Integrate orchestration, observability, lineage, and metadata management for a comprehensive environment.

Components (in addition to Intermediate): Apache Airflow, OpenTelemetry & Grafana Alloy, Grafana, Spline, OpenMetadata, MongoDB, cAdvisor.

Setup Steps:

Configure docker-compose.yml:

Open docker-compose.yml.

Uncomment ALL services including airflow-init, airflow-webserver, airflow-scheduler, airflow-worker, mongodb, openmetadata, grafana, grafana-alloy, cAdvisor, spline.

Ensure all environment variables for inter-service communication are correctly set.

Ensure necessary data/ subdirectories for persistent volumes exist.

Mount airflow\_dags and observability directories as volumes for Airflow DAGs and Grafana configurations. These airflow\_dags should now include separate DAGs for financial and insurance data pipelines to demonstrate complexity.

Bring Up Services:

Run the onboard.sh script or docker compose up --build -d. The airflow-init service will set up Airflow's database and load DAGs.

Initial Verification:

Verify Intermediate Track components are running and healthy.

Access Airflow UI: http://localhost:8080 (login admin/admin). Expected: Airflow UI accessible, multiple DAGs (e.g., for financial and insurance data) are listed (though not necessarily running).

Access Grafana UI: http://localhost:3000 (initially anonymous or configure admin user). Expected: Grafana UI accessible.

Access OpenMetadata UI: http://localhost:8585. Expected: OpenMetadata UI accessible.

Verify Spline UI: http://localhost:8081. Expected: Spline UI accessible.

Check Docker logs for all new services (e.g., grafana-alloy, cAdvisor). Expected: Services start without critical errors.

3. QA Test Suite for Data Platform Proficiency

This section provides a detailed, comprehensive test suite designed for a QA tester to validate the functionality and operational aspects of the data platform, demonstrating proficiency relevant to both Lead Data Engineer and AWS Engineer roles.

Reference: This test suite draws heavily from the Testing & Observability Patterns Deep-Dive Addendum, DR & Runbooks Deep-Dive Addendum, and Cloud Migration + Terraform Snippets Deep-Dive Addendum.

3.1. Starter Track Test Cases (FastAPI, PostgreSQL, MinIO)

Relevant Roles: Lead Data Engineer, AWS Engineer (simulating API Gateway/Lambda, RDS, S3)

3.2. Intermediate Track Test Cases (Kafka, Spark)

Relevant Roles: Lead Data Engineer, AWS Engineer (simulating MSK, Glue/EMR)

3.3. Advanced Track Test Cases (Airflow, Observability, Lineage, Metadata)

Relevant Roles: Lead Data Engineer, AWS Engineer (simulating MWAA, Managed Grafana/ADOT, Glue Data Catalog)

3.4. AWS Engineer Focus Test Cases (Cloud Concepts, IaC, CI/CD, DR)

Relevant Roles: AWS Engineer, Lead Data Engineer

These test cases are primarily conceptual demonstrations and discussions of how the local environment prepares for or mirrors AWS best practices. Actual execution would require an AWS account and provisioned resources.

4. Conclusion

This comprehensive QA test suite, integrated with the progressive setup guide, provides a robust framework for validating an enterprise-ready data platform. By actively performing these tests and understanding the underlying concepts, you will gain practical, demonstrable proficiency in roles such as Lead Data Engineer and AWS Engineer, solidifying your knowledge across local development, data quality, orchestration, cloud migration, and operational excellence.

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):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root.

Verify Airflow is accessible: Go to http://localhost:8080 and log in with admin/admin.

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.

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:

Place DAG: Ensure simple\_batch\_etl\_dag.py is in your airflow\_dags/ folder. Airflow will automatically detect it.

Unpause DAG: In the Airflow UI (http://localhost:8080), find simple\_batch\_etl\_spark\_job and toggle it to "On" (unpause).

Trigger DAG: Manually trigger a run by clicking the "Play" icon.

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:

Ensure Basic Use Case setup is complete.

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).

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:

Place DAG: Put data\_arrival\_sensor\_dag.py in airflow\_dags/.

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.

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.

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:

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.

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:

Create Config: Place data\_sources.json in a config/ directory at your project root.

Place DAG Generator: Put dynamic\_pipeline\_generator.py in airflow\_dags/.

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.).

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:

Ensure OpenMetadata is configured and running (Advanced Track setup).

Ensure PostgreSQL is running (Advanced Track setup).

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.

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:

Place DAG: Put full\_pipeline\_with\_governance\_dag.py in airflow\_dags/.

Ensure OM\_INGESTION\_SCRIPT is valid/dummy placeholder: Verify the path and command for ingest\_openmetadata is correct for your conceptual script.

Unpause and Trigger DAG: In Airflow UI, unpause and trigger full\_pipeline\_with\_governance.

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.

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:

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."}

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.

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.

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.

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:

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).

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.

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).

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):

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:

Place the DAG file: Save the rag\_pipeline\_dag.py content into your airflow\_dags/ directory.

Access Airflow UI: Go to http://localhost:8080.

Find and Enable the DAG: Locate rag\_knowledge\_base\_update\_pipeline, and toggle it to "On".

Trigger the DAG: Click the "Play" button (trigger DAG).

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.

Highlighting AWS SAM CLI: Local Serverless Development

The AWS Serverless Application Model Command Line Interface (AWS SAM CLI) is an essential tool for developing, testing, and debugging serverless applications locally before deploying them to AWS. It allows you to simulate the AWS Lambda and API Gateway environments on your local machine, significantly accelerating the development feedback loop and reducing cloud development costs.

This guide will demonstrate basic and advanced use cases of AWS SAM CLI, leveraging your Advanced Track local environment setup, particularly how it integrates with Docker and can simulate cloud services.

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, as well as the conceptual mention in Cloud Migration + Terraform Snippets Deep-Dive Addendum (Section 9.3).

Basic Use Case: Local Lambda Function Development & Testing

Objective: To demonstrate how to define a simple AWS Lambda function and its API Gateway endpoint using SAM, and then invoke it locally to test its functionality.

Role in Platform: Enable rapid iteration and testing of lightweight, event-driven ETL functions or API endpoints that will eventually run as AWS Lambdas, complementing Spark's distributed processing.

Setup/Configuration (Local Environment - Advanced Track):

Ensure Docker is running: SAM CLI uses Docker to run Lambda functions locally.

Install AWS SAM CLI: Follow the official AWS documentation for installation on your operating system.

Create a new SAM project directory:  
mkdir lambda\_sam\_etl  
cd lambda\_sam\_etl  
sam init --runtime python3.9 --app-template hello-world --name MyLocalETLLambda  
  
Choose the Hello World Example template, Python 3.9 runtime. This creates a template.yaml (SAM template) and hello\_world directory with app.py.

Modify hello\_world/app.py for a simple ETL concept:  
# lambda\_sam\_etl/hello\_world/app.py  
import json  
  
def lambda\_handler(event, context):  
 """  
 A simple Lambda function to simulate a lightweight ETL step.  
 It takes an input event (e.g., a mock transaction) and adds a processing timestamp.  
 """  
 print("Received event:", json.dumps(event, indent=2))  
  
 try:  
 # Assume event body is JSON string for API Gateway proxy integration  
 if 'body' in event and isinstance(event['body'], str):  
 input\_data = json.loads(event['body'])  
 else:  
 input\_data = event # Direct invocation  
  
 # Simulate a lightweight transformation: add a processing timestamp  
 if isinstance(input\_data, dict):  
 input\_data['processed\_timestamp'] = json.dumps(context.get\_remaining\_time\_in\_millis() / 1000.0) # Dummy timestamp  
 message = "Data processed successfully (local)."  
 else:  
 message = "Invalid input data format."  
 input\_data = {} # Ensure input\_data is a dict even if invalid  
  
 return {  
 "statusCode": 200,  
 "body": json.dumps({  
 "message": message,  
 "processed\_data": input\_data  
 }),  
 "headers": {  
 "Content-Type": "application/json"  
 }  
 }  
 except json.JSONDecodeError:  
 print("Error: Invalid JSON in event body.")  
 return {  
 "statusCode": 400,  
 "body": json.dumps({"message": "Invalid JSON input"}),  
 "headers": {  
 "Content-Type": "application/json"  
 }  
 }  
 except Exception as e:  
 print(f"An unexpected error occurred: {e}")  
 return {  
 "statusCode": 500,  
 "body": json.dumps({"message": f"Internal server error: {e}"}),  
 "headers": {  
 "Content-Type": "application/json"  
 }  
 }

Steps to Exercise:

Build the SAM application:  
sam build  
  
This command compiles your Lambda code and dependencies into a .aws-sam/build directory, ready for local execution.

Start the local API Gateway:  
sam local start-api  
  
This command starts a local web server that emulates API Gateway. It will print the local endpoint URL (e.g., http://127.0.0.1:3000).

Send a request to the local API:  
Open a new terminal and use curl to send a POST request.  
curl -X POST -H "Content-Type: application/json" \  
 -d '{"transaction\_id": "LOC-001", "amount": 100.0, "currency": "USD"}' \  
 http://127.0.0.1:3000/hello  
  
(Replace /hello with the path shown by sam local start-api if different).

Verification:

Console Output (sam local start-api terminal): You will see logs from your Lambda function (print statements) showing the received event and processing.

curl Output: The curl command should return a JSON response similar to:  
{"message": "Data processed successfully (local).", "processed\_data": {"transaction\_id": "LOC-001", "amount": 100.0, "currency": "USD", "processed\_timestamp": "..."}}  
  
This confirms your Lambda executed locally and performed the simulated transformation.

Advanced Use Case 1: Integrating with LocalStack for Mock AWS Services

Objective: To test a Lambda function that interacts with other AWS services (like S3 for data storage) by integrating SAM CLI with LocalStack, providing a full local cloud simulation.

Role in Platform: Crucial for testing Lambda-based micro-ETL or data transformation jobs that read from or write to S3, without incurring cloud costs or requiring live AWS credentials.

Setup/Configuration:

Ensure LocalStack is running: From your main data platform docker-compose.yml, ensure localstack service is uncommented and running (or run it separately).

Example docker-compose.yml snippet for LocalStack:  
# In your main docker-compose.yml  
localstack:  
 image: localstack/localstack:latest  
 ports:  
 - "4566:4566" # Standard LocalStack port  
 - "4510-4559:4510-4559" # For services on dynamic ports (e.g., S3)  
 environment:  
 # Set services to start (e.g., 's3' or 's3,lambda,sqs')  
 SERVICES: s3,lambda,apigateway  
 DEBUG: 1  
 # Add this to map localstack to host.docker.internal for SAM  
 HOSTNAME\_EXTERNAL: localhost  
 healthcheck:  
 test: ["CMD", "curl", "-f", "http://localhost:4566/health"]  
 interval: 10s  
 timeout: 5s  
 retries: 5

Install boto3: In your Lambda's requirements.txt (e.g., lambda\_sam\_etl/hello\_world/requirements.txt), add boto3. Run pip install -r requirements.txt locally, then sam build.

Modify hello\_world/app.py to interact with S3:  
# lambda\_sam\_etl/hello\_world/app.py (updated)  
import json  
import os  
import boto3  
  
# Configure S3 client to point to LocalStack  
# This environment variable will be passed during `sam local invoke`  
S3\_ENDPOINT\_URL = os.environ.get("S3\_ENDPOINT\_URL", None)  
s3\_client = boto3.client('s3', endpoint\_url=S3\_ENDPOINT\_URL) if S3\_ENDPOINT\_URL else boto3.client('s3')  
  
def lambda\_handler(event, context):  
 print("Received event:", json.dumps(event, indent=2))  
 bucket\_name = "my-local-data-bucket"  
 object\_key = f"processed-data/{context.aws\_request\_id}.json"  
  
 try:  
 input\_data = {}  
 if 'body' in event and isinstance(event['body'], str):  
 input\_data = json.loads(event['body'])  
 else:  
 input\_data = event  
  
 input\_data['processed\_timestamp'] = json.dumps(context.get\_remaining\_time\_in\_millis() / 1000.0)  
  
 # Create bucket if it doesn't exist  
 try:  
 s3\_client.head\_bucket(Bucket=bucket\_name)  
 except s3\_client.exceptions.ClientError as e:  
 if e.response['Error']['Code'] == '404':  
 print(f"Bucket {bucket\_name} does not exist, creating...")  
 s3\_client.create\_bucket(Bucket=bucket\_name)  
 print(f"Bucket {bucket\_name} created.")  
 else:  
 raise # Re-raise other errors  
  
 # Write processed data to S3  
 s3\_client.put\_object(  
 Bucket=bucket\_name,  
 Key=object\_key,  
 Body=json.dumps(input\_data)  
 )  
 print(f"Object written to s3://{bucket\_name}/{object\_key}")  
  
 return {  
 "statusCode": 200,  
 "body": json.dumps({  
 "message": f"Data processed and stored in S3://{bucket\_name}/{object\_key}",  
 "processed\_data": input\_data  
 }),  
 "headers": {  
 "Content-Type": "application/json"  
 }  
 }  
 except Exception as e:  
 print(f"An unexpected error occurred: {e}")  
 return {  
 "statusCode": 500,  
 "body": json.dumps({"message": f"Internal server error: {e}"}),  
 "headers": {  
 "Content-Type": "application/json"  
 }  
 }

Steps to Exercise:

Ensure LocalStack is running and healthy.

Build SAM app again: sam build

Invoke Lambda locally, pointing to LocalStack:  
sam local invoke MyLocalETLLambda \  
 --event-str '{"body": "{\"transaction\_id\": \"S3-TEST-001\", \"amount\": 250.0}"}' \  
 --env-vars env.json

Create an env.json file in lambda\_sam\_etl/ to configure the S3 endpoint for LocalStack:  
{  
 "MyLocalETLLambda": {  
 "S3\_ENDPOINT\_URL": "http://host.docker.internal:4566"  
 }  
}  
  
Note: host.docker.internal allows the Lambda container (run by SAM) to connect to LocalStack running directly on your host machine (or in a separate Docker network). If LocalStack is in the same Docker network as SAM's build container, you might use the LocalStack service name (e.g., http://localstack:4566). host.docker.internal is more general for this local setup.

Verification:

SAM CLI Output: The sam local invoke command will show the Lambda's logs, including the "Object written to s3://..." message.

LocalStack Logs: Check the logs of your LocalStack container (docker compose logs localstack). You should see S3 PutObject and CreateBucket calls.

LocalStack S3: Use the AWS CLI configured for LocalStack (or LocalStack's UI if available):  
aws --endpoint-url=http://localhost:4566 s3 ls s3://my-local-data-bucket/  
aws --endpoint-url=http://localhost:4566 s3 cp s3://my-local-data-bucket/processed-data/<object\_key>.json -  
  
You should see the bucket and the JSON object created by your Lambda.

Advanced Use Case 2: Event-Driven Processing (Simulating Kafka Triggers)

Objective: To demonstrate how to test a Lambda function that is designed to be triggered by a Kafka event, allowing local debugging of consumer logic for streaming data.

Role in Platform: Develop and test serverless functions that act as lightweight consumers of Kafka topics, performing real-time transformations or triggering subsequent actions, complementing or replacing parts of Spark streaming for simpler tasks.

Setup/Configuration:

Ensure Kafka is running: From your main data platform docker-compose.yml, ensure kafka and zookeeper services are running.

Update hello\_world/app.py for Kafka event processing:  
# lambda\_sam\_etl/hello\_world/app.py (updated for Kafka event)  
import json  
import base64  
  
def lambda\_handler(event, context):  
 """  
 Lambda function to process a mock Kafka event.  
 Assumes the event structure from a Kafka trigger (e.g., MSK as an event source).  
 """  
 print("Received Kafka event:", json.dumps(event, indent=2))  
  
 try:  
 records = event.get('records', {}).get('example.com:RawFinancialTransactions', []) # Adjust topic name as needed  
 processed\_messages = []  
  
 for record in records:  
 # Kafka event value is base64 encoded  
 decoded\_value = base64.b64decode(record['value']).decode('utf-8')  
 message\_data = json.loads(decoded\_value)  
  
 # Add a processing timestamp or perform a simple transformation  
 message\_data['kafka\_processed\_at'] = json.dumps(context.get\_remaining\_time\_in\_millis() / 1000.0) # Dummy timestamp  
 message\_data['source\_topic'] = record['topic']  
 message\_data['kafka\_offset'] = record['offset']  
  
 processed\_messages.append(message\_data)  
 print(f"Processed message from topic {record['topic']}: {message\_data}")  
  
 return {  
 "statusCode": 200,  
 "body": json.dumps({  
 "message": f"Successfully processed {len(processed\_messages)} Kafka messages.",  
 "processed\_records": processed\_messages  
 }),  
 "headers": {  
 "Content-Type": "application/json"  
 }  
 }  
  
 except Exception as e:  
 print(f"Error processing Kafka event: {e}")  
 return {  
 "statusCode": 500,  
 "body": json.dumps({"message": f"Internal server error: {e}"}),  
 "headers": {  
 "Content-Type": "application/json"  
 }  
 }

Create a mock Kafka event file: In your lambda\_sam\_etl/ directory, create kafka\_event.json. This mimics the structure of an event that Lambda receives from an MSK (Managed Streaming for Kafka) trigger.  
{  
 "eventSource": "aws:kafka",  
 "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/MyMSKCluster/a1b2c3d4-5678-90ab-cdef-111111111111-2",  
 "records": {  
 "example.com:RawFinancialTransactions": [  
 {  
 "topic": "raw\_financial\_transactions",  
 "partition": 0,  
 "offset": 100,  
 "timestamp": "2024-06-13T10:00:00.000Z",  
 "timestampType": "CREATE\_TIME",  
 "value": "eyJ0cmFuc2FjdGlvbl9pZCI6ICJLRlQtMDAxIiwgImFtb3VudCI6IDE1MC4wLCAiY3VycmVuY3kiOiAiVVNEIn0=",  
 "key": "base64encodedkey"  
 },  
 {  
 "topic": "raw\_financial\_transactions",  
 "partition": 0,  
 "offset": 101,  
 "timestamp": "2024-06-13T10:01:00.000Z",  
 "timestampType": "CREATE\_TIME",  
 "value": "eyJ0cmFuc2FjdGlvbl9pZCI6ICJLRlQtMDAyIiwgImFtb3VudCI6IDIwMC4wLCAiY3VycmVuY3kiOiAiRVVSIn0=",  
 "key": "base64encodedkey"  
 }  
 ],  
 "example.com:RawInsuranceClaims": [  
 {  
 "topic": "raw\_insurance\_claims",  
 "partition": 1,  
 "offset": 50,  
 "timestamp": "2024-06-13T10:05:00.000Z",  
 "timestampType": "CREATE\_TIME",  
 "value": "eyJjbGFpbV9pZCI6ICJMQUNfMDAxIiwgInBvbGljeV9udW1iZXIiOiAiUE9MLTExMSIsICJjbGFpbV9hbW91bnQiOiAxMDAwLjAsICJjbGFpbV90eXBlIjogImF1dG8ifQ==",  
 "key": "base64encodedkey"  
 }  
 ]  
 }  
}  
  
Note: The value fields are base64 encoded JSON strings for {"transaction\_id": "KFT-001", "amount": 150.0, "currency": "USD"} for financial and {"claim\_id": "LAC\_001", "policy\_number": "POL-111", "claim\_amount": 1000.0, "claim\_type": "auto"} for insurance.

Steps to Exercise:

Build SAM app: sam build

Invoke Lambda locally with Kafka event:  
sam local invoke MyLocalETLLambda --event kafka\_event.json

Verification:

SAM CLI Output: The console will show the Lambda's logs, indicating that it successfully parsed and processed the Kafka messages from both financial and insurance topics. The processed\_records in the response body will contain the transformed data. This demonstrates how you can test Kafka-triggered Lambdas locally with mock event payloads.

Advanced Use Case 3: Layered Lambda Functions for Shared Dependencies

Objective: To demonstrate how to create and use Lambda Layers to share common code and dependencies across multiple Lambda functions, reducing deployment package sizes and promoting code reuse.

Role in Platform: Efficiently manage libraries for multiple serverless ETL functions, ensuring consistency and simplified maintenance, especially for common data models or utility functions.

Setup/Configuration:

Define a Lambda Layer in template.yaml: Add a Resources section for the layer.  
# lambda\_sam\_etl/template.yaml (updated)  
AWSTemplateFormatVersion: '2010-09-09'  
Transform: AWS::Serverless-2016-10-31  
Description: >  
 MyLocalETLLambda  
  
 Sample SAM Template for MyLocalETLLambda.  
  
Globals:  
 Function:  
 Timeout: 30  
 MemorySize: 128  
  
Resources:  
 MyLocalETLLambda:  
 Type: AWS::Serverless::Function  
 Properties:  
 CodeUri: hello\_world/  
 Handler: app.lambda\_handler  
 Runtime: python3.9  
 Architectures:  
 - x86\_64  
 Events:  
 HelloWorld:  
 Type: Api  
 Properties:  
 Path: /hello  
 Method: post  
 Layers:  
 - !Ref CommonUtilsLayer # Reference the layer here  
  
 # Define the Lambda Layer  
 CommonUtilsLayer:  
 Type: AWS::Serverless::LayerVersion  
 Properties:  
 LayerContentUri: common\_layer/  
 CompatibleRuntimes:  
 - python3.9  
 RetentionPolicy: Retain # Keep layer even if stack is deleted

Create the shared common code: Create a new directory lambda\_sam\_etl/common\_layer/python/ (SAM expects python/ subfolder) and add a utility file, e.g., my\_utils.py.  
mkdir -p lambda\_sam\_etl/common\_layer/python  
  
Example lambda\_sam\_etl/common\_layer/python/my\_utils.py:  
# lambda\_sam\_etl/common\_layer/python/my\_utils.py  
def format\_timestamp(timestamp\_str):  
 """Formats a timestamp string for consistency."""  
 from datetime import datetime  
 try:  
 dt\_obj = datetime.fromisoformat(timestamp\_str.replace('Z', '+00:00'))  
 return dt\_obj.strftime("%Y-%m-%d %H:%M:%S UTC")  
 except ValueError:  
 return "Invalid Timestamp"  
  
def calculate\_tax(amount, rate=0.05):  
 """Calculates a simple tax."""  
 return round(amount \* rate, 2)

Modify hello\_world/app.py to use the layer:  
# lambda\_sam\_etl/hello\_world/app.py (updated to use layer)  
import json  
import base64  
import os  
import boto3  
from my\_utils import format\_timestamp, calculate\_tax # Import from the layer  
  
S3\_ENDPOINT\_URL = os.environ.get("S3\_ENDPOINT\_URL", None)  
s3\_client = boto3.client('s3', endpoint\_url=S3\_ENDPOINT\_URL) if S3\_ENDPOINT\_URL else boto3.client('s3')  
  
def lambda\_handler(event, context):  
 print("Received event:", json.dumps(event, indent=2))  
  
 try:  
 # Assume event body is JSON string for API Gateway proxy integration  
 if 'body' in event and isinstance(event['body'], str):  
 input\_data = json.loads(event['body'])  
 else:  
 input\_data = event # Direct invocation  
  
 # Use functions from the layer  
 original\_timestamp = input\_data.get('timestamp', 'N/A')  
 input\_data['formatted\_timestamp\_from\_layer'] = format\_timestamp(original\_timestamp)  
  
 original\_amount = input\_data.get('amount', 0.0)  
 input\_data['calculated\_tax\_from\_layer'] = calculate\_tax(original\_amount)  
  
 input\_data['processed\_timestamp'] = json.dumps(context.get\_remaining\_time\_in\_millis() / 1000.0)  
  
 # ... (S3 writing logic from Advanced Use Case 1 if desired, or remove for simple test)  
  
 return {  
 "statusCode": 200,  
 "body": json.dumps({  
 "message": "Data processed with layer functions.",  
 "processed\_data": input\_data  
 }),  
 "headers": {  
 "Content-Type": "application/json"  
 }  
 }  
 except Exception as e:  
 print(f"An unexpected error occurred: {e}")  
 return {  
 "statusCode": 500,  
 "body": json.dumps({"message": f"Internal server error: {e}"}),  
 "headers": {  
 "Content-Type": "application/json"  
 }  
 }

Steps to Exercise:

Build SAM app (layers will be built):  
sam build

Start local API Gateway:  
sam local start-api

Send a request:  
curl -X POST -H "Content-Type: application/json" \  
 -d '{"transaction\_id": "LAYER-001", "timestamp": "2024-06-13T15:00:00Z", "amount": 200.0}' \  
 http://127.0.0.1:3000/hello

Verification:

Console Output: The sam local start-api terminal logs and curl response will show the processed\_data including formatted\_timestamp\_from\_layer and calculated\_tax\_from\_layer fields, populated by functions from your shared layer. This confirms the Lambda successfully accessed and used code from the defined layer.

This concludes the guide for AWS SAM CLI.

Highlighting FastAPI: High-Performance Data Ingestion API

FastAPI is a modern, fast (high-performance) web framework for building APIs with Python 3.7+ based on standard Python type hints. In your data platform, it serves as the crucial ingestion layer, providing robust and well-documented endpoints for receiving disparate data (e.g., financial transactions, insurance claims) from external sources before it enters your streaming pipeline.

This guide will demonstrate basic and advanced use cases of FastAPI, leveraging your Advanced Track local environment setup and its integration with Kafka and observability tools.

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: Receiving and Acknowledging Data Ingestion

Objective: To demonstrate FastAPI's core function: providing a well-defined HTTP endpoint to receive data and return a success acknowledgment.

Role in Platform: Act as the secure, high-performance gateway for all incoming data, translating external requests into internal data platform events.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root.

Verify FastAPI is accessible: Check Docker logs for the fastapi\_ingestor container (docker compose logs fastapi\_ingestor). Its health check endpoint should be reachable at http://localhost:8000/health.

Review FastAPI application code: Your fastapi\_app/app/main.py should define the API endpoints and Pydantic models for request bodies.  
Example fastapi\_app/app/main.py (conceptual, core parts):  
# fastapi\_app/app/main.py  
from fastapi import FastAPI, HTTPException, status  
from pydantic import BaseModel, Field  
from typing import Optional  
from datetime import datetime  
import os  
import json  
from kafka import KafkaProducer # Assumed to be installed in FastAPI container  
  
# --- Pydantic Models for Data Contracts ---  
class FinancialTransaction(BaseModel):  
 transaction\_id: str = Field(..., example="FT-20231026-001")  
 timestamp: datetime = Field(..., example="2023-10-26T14:30:00Z")  
 account\_id: str = Field(..., example="ACC-001")  
 amount: float = Field(..., gt=0, example=150.75) # Amount must be greater than 0  
 currency: str = Field(..., max\_length=3, example="USD")  
 transaction\_type: str = Field(..., example="debit")  
 merchant\_id: Optional[str] = Field(None, example="MER-XYZ")  
 category: Optional[str] = Field(None, example="groceries")  
  
class InsuranceClaim(BaseModel):  
 claim\_id: str = Field(..., example="IC-20231026-001")  
 timestamp: datetime = Field(..., example="2023-10-26T15:00:00Z")  
 policy\_number: str = Field(..., example="POL-987654")  
 claim\_amount: float = Field(..., gt=0, example=1000.00)  
 claim\_type: str = Field(..., example="auto")  
 claim\_status: str = Field(..., example="submitted")  
 customer\_id: str = Field(..., example="CUST-ABC")  
 incident\_date: datetime = Field(..., example="2023-09-15T08:00:00Z")  
  
# --- FastAPI Application Initialization ---  
app = FastAPI(  
 title="Financial/Insurance Data Ingestor API",  
 description="API for ingesting various financial and insurance data into the data platform.",  
 version="1.0.0",  
)  
  
# --- Kafka Producer Setup ---  
KAFKA\_BROKER = os.getenv("KAFKA\_BROKER", "kafka:29092") # Default to service name for Docker Compose  
KAFKA\_TOPIC\_FINANCIAL = os.getenv("KAFKA\_TOPIC\_FINANCIAL", "raw\_financial\_transactions")  
KAFKA\_TOPIC\_INSURANCE = os.getenv("KAFKA\_TOPIC\_INSURANCE", "raw\_insurance\_claims")  
  
# Initialize Kafka producer globally (or use a dependency injection)  
try:  
 producer = KafkaProducer(  
 bootstrap\_servers=[KAFKA\_BROKER],  
 value\_serializer=lambda v: json.dumps(v).encode('utf-8'),  
 retries=5, # Number of retries on failed sends  
 linger\_ms=100, # Batch messages for 100ms  
 batch\_size=16384 # Batch size in bytes  
 )  
 print(f"Kafka Producer initialized for broker: {KAFKA\_BROKER}")  
except Exception as e:  
 print(f"Error initializing Kafka Producer: {e}")  
 producer = None # Handle case where producer fails to initialize  
  
# --- API Endpoints ---  
@app.get("/health", tags=["Monitoring"])  
async def health\_check():  
 """Health check endpoint."""  
 return {"status": "healthy", "message": "Welcome to Financial/Insurance Data Ingestor API!"}  
  
@app.post("/ingest-financial-transaction/", status\_code=status.HTTP\_200\_OK, tags=["Ingestion"])  
async def ingest\_financial\_transaction(transaction: FinancialTransaction):  
 """  
 Ingests a financial transaction record.  
 This endpoint uses Pydantic for automatic request validation.  
 """  
 try:  
 if producer:  
 producer.send(KAFKA\_TOPIC\_FINANCIAL, transaction.dict()).get(timeout=10) # Send synchronously for basic verification  
 print(f"Financial transaction ingested and sent to Kafka topic '{KAFKA\_TOPIC\_FINANCIAL}': {transaction.transaction\_id}")  
 else:  
 print("Kafka producer not available. Skipping send.")  
 return {"message": "Financial transaction ingested successfully", "transaction\_id": transaction.transaction\_id}  
 except Exception as e:  
 raise HTTPException(status\_code=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR, detail=f"Failed to ingest transaction: {e}")  
  
@app.post("/ingest-insurance-claim/", status\_code=status.HTTP\_200\_OK, tags=["Ingestion"])  
async def ingest\_insurance\_claim(claim: InsuranceClaim):  
 """  
 Ingests an insurance claim record.  
 This endpoint uses Pydantic for automatic request validation.  
 """  
 try:  
 if producer:  
 producer.send(KAFKA\_TOPIC\_INSURANCE, claim.dict()).get(timeout=10) # Send synchronously for basic verification  
 print(f"Insurance claim ingested and sent to Kafka topic '{KAFKA\_TOPIC\_INSURANCE}': {claim.claim\_id}")  
 else:  
 print("Kafka producer not available. Skipping send.")  
 return {"message": "Insurance claim ingested successfully", "claim\_id": claim.claim\_id}  
 except Exception as e:  
 raise HTTPException(status\_code=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR, detail=f"Failed to ingest claim: {e}")

Steps to Exercise:

Send a valid financial transaction:  
Open a terminal and use curl to send a POST request.  
curl -X POST -H "Content-Type: application/json" \  
 -d '{  
 "transaction\_id": "FT-001",  
 "timestamp": "2024-06-14T10:00:00Z",  
 "account\_id": "ACC-FIN-001",  
 "amount": 100.50,  
 "currency": "USD",  
 "transaction\_type": "purchase"  
 }' \  
 http://localhost:8000/ingest-financial-transaction/

Send a valid insurance claim:  
curl -X POST -H "Content-Type: application/json" \  
 -d '{  
 "claim\_id": "IC-001",  
 "timestamp": "2024-06-14T11:00:00Z",  
 "policy\_number": "POL-12345",  
 "claim\_amount": 5000.00,  
 "claim\_type": "auto",  
 "claim\_status": "submitted",  
 "customer\_id": "CUST-001",  
 "incident\_date": "2024-05-01T09:00:00Z"  
 }' \  
 http://localhost:8000/ingest-insurance-claim/

Observe FastAPI logs: In a separate terminal, watch the logs of the fastapi\_ingestor container:  
docker compose logs -f fastapi\_ingestor

Verification:

HTTP Response: Both curl commands should return HTTP 200 OK with a JSON response confirming successful ingestion (e.g., {"message": "Financial transaction ingested successfully", "transaction\_id": "FT-001"}).

FastAPI Logs: The fastapi\_ingestor logs will show messages like "Financial transaction ingested and sent to Kafka topic 'raw\_financial\_transactions': FT-001", confirming receipt and forwarding.

Advanced Use Case 1: Robust Input Validation and Error Handling

Objective: To demonstrate FastAPI's automatic data validation using Pydantic models, and its ability to return clear, structured error responses for invalid input, preventing bad data from entering the pipeline.

Role in Platform: Act as the first line of defense for data quality, ensuring that incoming data adheres to defined schemas and business rules before further processing.

Setup/Configuration:

Basic Use Case completed: Ensure FastAPI is running and you have the FinancialTransaction and InsuranceClaim Pydantic models defined as shown above.

Steps to Exercise:

Send an invalid financial transaction (missing required field):  
curl -X POST -H "Content-Type: application/json" \  
 -d '{  
 "timestamp": "2024-06-14T10:00:00Z",  
 "account\_id": "ACC-FIN-001",  
 "amount": 100.50,  
 "currency": "USD",  
 "transaction\_type": "purchase"  
 }' \  
 http://localhost:8000/ingest-financial-transaction/  
  
(transaction\_id is missing)

Send an invalid financial transaction (wrong data type):  
curl -X POST -H "Content-Type: application/json" \  
 -d '{  
 "transaction\_id": "FT-002",  
 "timestamp": "2024-06-14T10:00:00Z",  
 "account\_id": "ACC-FIN-001",  
 "amount": "not\_a\_number",  
 "currency": "USD",  
 "transaction\_type": "purchase"  
 }' \  
 http://localhost:8000/ingest-financial-transaction/  
  
(amount is a string, but expects float)

Send an invalid insurance claim (invalid claim\_amount - less than or equal to 0):  
curl -X POST -H "Content-Type: application/json" \  
 -d '{  
 "claim\_id": "IC-002",  
 "timestamp": "2024-06-14T11:00:00Z",  
 "policy\_number": "POL-12346",  
 "claim\_amount": -50.00,  
 "claim\_type": "health",  
 "claim\_status": "submitted",  
 "customer\_id": "CUST-002",  
 "incident\_date": "2024-05-01T09:00:00Z"  
 }' \  
 http://localhost:8000/ingest-insurance-claim/  
  
(claim\_amount violates gt=0 constraint)

Verification:

HTTP Response: All curl commands should return HTTP 422 Unprocessable Entity.

Response Body: The response body will contain a structured JSON error message detailing the validation failures (e.g., "field required", "value is not a valid float", "ensure this value is greater than 0"). This provides clear feedback to API clients.

FastAPI Logs: The fastapi\_ingestor logs will show INFO:uvicorn.access:XXX "POST /ingest-... 422 Unprocessable Entity" entries, confirming the validation errors.

Advanced Use Case 2: Asynchronous Kafka Publishing and Idempotency

Objective: To demonstrate how FastAPI can asynchronously publish messages to Kafka and how the ingestion endpoint can conceptually handle idempotency to prevent duplicate processing from client retries.

Role in Platform: Maximize ingestion throughput by not waiting for Kafka acknowledgment (for high-volume fire-and-forget scenarios) and ensure data integrity by gracefully handling re-submission of the same data.

Setup/Configuration:

FastAPI with Kafka Producer: Ensure fastapi\_app/app/main.py has the KafkaProducer setup.

Modify producer.send(): For asynchronous behavior, remove .get(timeout=...) to make it non-blocking. For idempotency, we'll conceptualize using a unique ID.  
Example fastapi\_app/app/main.py (modification to ingest\_financial\_transaction):  
# ... inside ingest\_financial\_transaction endpoint ...  
 if producer:  
 # Send asynchronously and add a callback for logging success/failure  
 # This makes the API endpoint respond faster  
 future = producer.send(KAFKA\_TOPIC\_FINANCIAL, transaction.dict())  
 future.add\_callback(lambda record\_metadata: print(  
 f"Successfully sent transaction {transaction.transaction\_id} to topic "  
 f"{record\_metadata.topic} partition {record\_metadata.partition} "  
 f"offset {record\_metadata.offset}"  
 ))  
 future.add\_errback(lambda exc: print(  
 f"Failed to send transaction {transaction.transaction\_id}: {exc}"  
 ))  
 print(f"Financial transaction ingestion request acknowledged for {transaction.transaction\_id}. Publishing to Kafka asynchronously.")  
 else:  
 print("Kafka producer not available. Skipping send.")  
 # Conceptual Idempotency check:  
 # In a real system, you'd store transaction\_id in a fast lookup (e.g., Redis)  
 # and reject if already seen within a short window.  
 # For this demo, we'll just acknowledge the request as if it's new.  
 return {"message": "Financial transaction accepted for processing", "transaction\_id": transaction.transaction\_id}  
# ... similar for insurance claims ...

Steps to Exercise:

Restart FastAPI container: docker compose restart fastapi\_ingestor to apply code changes.

Generate data with simulate\_data.py: Let it run for a while.

Observe the fastapi\_ingestor logs. You'll see Acknowledging... Publishing to Kafka asynchronously messages immediately, followed by Successfully sent... callbacks a moment later. This shows the non-blocking send.

Simulate a retry/duplicate submission:

Pick a transaction\_id from a previously successful request (e.g., FT-001).

Send the exact same request again using curl.

Expected Response: Even though it's a duplicate, the API will still return HTTP 200 OK and Financial transaction accepted for processing.

Conceptual Idempotency Handling: While the API endpoint itself accepts it, the downstream Spark job (or Lambda) consuming from Kafka must be designed to handle idempotency. This is typically done by using the transaction\_id as a primary key for upserts into Delta Lake, ensuring that re-processing the same message doesn't create duplicate records but merely updates the existing one.

Verification:

FastAPI Response Time: The FastAPI endpoint will return responses faster compared to blocking on producer.send().get().

Kafka Logs: The Kafka consumer (Spark job or console consumer) will still receive the duplicate message, but the data platform's later stages (e.g., Delta Lake MERGE INTO operation based on transaction\_id) are responsible for ensuring logical idempotency, resulting in only one unique record for FT-001 in the curated zone.

Advanced Use Case 3: API Documentation (Swagger UI) & Observability Integration

Objective: To demonstrate FastAPI's automatic generation of interactive API documentation (Swagger UI/OpenAPI) and its integration with OpenTelemetry/Prometheus for detailed API observability metrics.

Role in Platform: Provide self-service documentation for API consumers (developers, data scientists) and ensure comprehensive monitoring of the ingestion layer for operational teams.

Setup/Configuration:

Ensure fastapi\_app/app/main.py is configured with title, description, version for FastAPI (as shown in Basic Use Case).

Install Prometheus Instrumentator: Ensure prometheus\_fastapi\_instrumentator is in fastapi\_app/requirements.txt and imported/instrumented in main.py (as hinted in "Highlighting cAdvisor" document's Advanced Use Case 2).  
# fastapi\_app/app/main.py (additions for observability)  
from prometheus\_fastapi\_instrumentator import Instrumentator  
# ...  
app = FastAPI(  
 title="Financial/Insurance Data Ingestor API",  
 description="API for ingesting various financial and insurance data into the data platform.",  
 version="1.0.0",  
)  
Instrumentator().instrument(app).expose(app) # Exposes metrics on /metrics endpoint by default

Ensure Grafana Alloy is configured to scrape FastAPI metrics: Check observability/alloy-config.river for a scrape configuration for fastapi\_ingestor on port 8000 at path /metrics.

Ensure Grafana is running and accessible.

Steps to Exercise:

Access Interactive API Documentation (Swagger UI):

Open your web browser and navigate to http://localhost:8000/docs.

Explore: You will see a comprehensive, interactive documentation of your FastAPI endpoints, including request schemas, example values, and response codes, generated automatically from your Pydantic models and endpoint decorators. You can even try out API calls directly from this UI.

Access OpenAPI Specification:

Navigate to http://localhost:8000/openapi.json.

Explore: This provides the raw OpenAPI (Swagger) specification in JSON format, which can be used by various tools for client code generation, API testing, or API management platforms.

Generate API Traffic:

Run python3 simulate\_data.py to continuously send requests to the FastAPI endpoints.

Observe API Metrics in Grafana:

Go to http://localhost:3000.

Navigate to your "Health Dashboard" or create a new panel.

Query (PromQL):

http\_requests\_total{job="fastapi\_ingestor"}: To see total requests.

http\_request\_duration\_seconds\_bucket{job="fastapi\_ingestor", le="0.5"}: To see requests completing within 500ms (for latency).

rate(http\_requests\_total{job="fastapi\_ingestor", method="POST"}[1m]): To see Requests Per Second for POST methods.

Observe: The graphs will show real-time metrics for your FastAPI ingestion API, demonstrating its throughput, latency, and overall health.

Verification:

Swagger UI: The interactive documentation is fully functional and reflects your API's endpoints and schemas.

OpenAPI Spec: The raw OpenAPI JSON is accessible and well-formed.

Grafana: Metrics from FastAPI are flowing correctly into Grafana, providing granular insights into the API's performance, allowing operators to monitor its health and identify issues quickly.

This concludes the guide for FastAPI.

Highlighting Apache Kafka: Distributed Streaming Platform

Apache Kafka is the backbone of real-time data ingestion and streaming applications within your enterprise data platform. It acts as a highly scalable, fault-tolerant, and durable message broker, decoupling data producers from consumers and enabling a wide array of real-time use cases.

This guide will demonstrate basic and advanced use cases of Apache Kafka, 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: High-Throughput Decoupled Data Ingestion

Objective: To demonstrate Kafka's fundamental role in receiving high volumes of disparate data (financial transactions and insurance claims) from the FastAPI ingestor and holding it durably for asynchronous consumption.

Role in Platform: Serve as the central streaming hub, ensuring data resilience and decoupling producers from consumers, preventing backpressure issues.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root.

Verify Kafka is accessible: Check Docker logs for the kafka and zookeeper containers to ensure they are healthy.

Ensure Kafka topics are initialized: As per onboard.sh or manual commands, confirm raw\_financial\_transactions and raw\_insurance\_claims topics exist.

Steps to Exercise:

Start Data Generator:  
Open a new terminal session in your project root and execute the simulate\_data.py script. This script sends mock financial and insurance data to your FastAPI endpoints, which then publish to Kafka.  
python3 simulate\_data.py  
  
Let this run in the background.

Verify Data Ingestion (Kafka Console Consumers):  
Open two separate new terminal sessions.

Financial Data Consumer:  
docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw\_financial\_transactions --from-beginning

Insurance Data Consumer:  
docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw\_insurance\_claims --from-beginning

Verification:

Console Output: Both kafka-console-consumer terminals should continuously display JSON messages, confirming that FastAPI is successfully publishing data to the respective Kafka topics and Kafka is receiving and storing them.

FastAPI Logs: Check the fastapi\_ingestor container logs (docker compose logs fastapi\_ingestor) for messages indicating successful publishing to Kafka.

Advanced Use Case 1: Real-time Fan-out to Multiple Consumers

Objective: To demonstrate Kafka's ability to deliver the same data stream to multiple, independent consumer groups without affecting each other's progress, enabling diverse real-time applications.

Role in Platform: Enable real-time dashboards, fraud detection systems, and other concurrent consumers from a single, canonical data stream.

Setup/Configuration:

Basic Use Case completed: Ensure data is actively flowing into both raw\_financial\_transactions and raw\_insurance\_claims Kafka topics (i.e., simulate\_data.py is running).

Spark Streaming Consumers are active: Ensure your Spark streaming jobs are consuming from these topics (as set up in the Data Platform Usage Guide, Section 3).

Steps to Exercise:

Start a new "Dashboard" Consumer (Financial Data):  
Open a new terminal and start a Kafka console consumer for financial transactions, but assign it a different consumer group ID than your Spark job's consumer group.  
docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw\_financial\_transactions --group real-time-dashboard-financial --from-beginning

Start a new "Audit" Consumer (Insurance Data):  
Open another new terminal and start a Kafka console consumer for insurance claims, with a unique group ID.  
docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw\_insurance\_claims --group audit-log-insurance --from-beginning

Observe all Consumers: Keep an eye on the output of these new consumers, as well as the logs/metrics of your existing Spark streaming jobs (e.g., via Grafana's Kafka consumer lag panel).

Verification:

All Consumers Receiving Data: Both new console consumers should be actively receiving financial and insurance transaction messages, respectively.

Independent Progress: Crucially, the Kafka consumer lag for your Spark jobs (visible in Grafana) should remain low and unaffected by the introduction of these new console consumers. This demonstrates that each consumer group maintains its own offset and consumes independently from the same topic.

Data Consistency: All consumers receive the exact same sequence of messages within a partition, showcasing Kafka's immutable log.

Advanced Use Case 2: Data Retention, Replay, and Disaster Recovery Preparedness

Objective: To demonstrate Kafka's capability to retain messages for a configurable period, allowing for data replay from arbitrary points, which is crucial for reprocessing, debugging, and disaster recovery scenarios.

Role in Platform: Provide a durable, replayable source of truth for streaming data, reducing data loss RPO (Recovery Point Objective).

Setup/Configuration:

Basic Use Case completed: Ensure simulate\_data.py is running and ingesting data into Kafka.

Identify a Kafka Topic: For this example, we'll use raw\_financial\_transactions.

Note Current Offset: Before stopping Spark, check a recent offset.

Steps to Exercise:

Simulate Spark Job Failure/Redeployment:  
Pause your financial Spark streaming job container (or stop the corresponding Airflow DAG run) to simulate a failure or a need for a full reprocessing.  
docker compose pause spark # If one Spark service handles both. If separate, pause only the relevant part.  
  
Observe in Grafana: The Kafka consumer lag for raw\_financial\_transactions will start to increase significantly.

Let Data Accumulate: Allow simulate\_data.py to continue running for a few minutes while Spark is paused, letting messages accumulate in Kafka.

Simulate New/Restarted Spark Job (Replay from Earliest):  
Unpause Spark (if paused) or submit a new Spark job instance for financial data, configured to read from the earliest offset (or a specific historical offset if you noted one earlier).  
docker compose unpause spark # If you paused it  
# Or, if restarting a new instance/job, ensure its startingOffsets is set to 'earliest'  
# Example for new consumer reading from earliest for replay  
docker exec -it 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 \  
 raw\_financial\_transactions kafka:29092 s3a://raw-data-bucket/financial\_data\_delta\_reprocessed \  
 --startingOffsets earliest # <-- Key for replay

Verification:

Grafana: Observe the Kafka consumer lag for raw\_financial\_transactions rapidly decreasing as the (restarted/new) Spark job consumes the accumulated backlog and older messages.

MinIO: If you wrote to a new path (financial\_data\_delta\_reprocessed), you'll see new data accumulating, including historical data that was reprocessed. This validates Kafka's durable storage and replay capabilities.

Advanced Use Case 3: Partitioning Strategy & Scalability

Objective: To understand how Kafka partitioning directly influences throughput and consumer parallelism, demonstrating a core scaling mechanism.

Role in Platform: Optimize data distribution for maximum throughput and enable highly parallel consumption.

Setup/Configuration:

Initial Setup: Your raw\_financial\_transactions topic likely has 3 partitions (as initialized by onboard.sh).

Prepare Scalable Consumer (Conceptual): Your Spark streaming job is inherently capable of parallel consumption across partitions.

Steps to Exercise:

Monitor Baseline:

Ensure simulate\_data.py is running at a high rate (e.g., DELAY\_SECONDS = 0.01 in simulate\_data.py).

Monitor the raw\_financial\_transactions Kafka topic's consumer lag in Grafana. Observe the Spark job's CPU utilization in Grafana. This is your baseline performance with the current partition count.

Increase Topic Partitions (Dynamically):  
While it's generally best practice to set partition counts at topic creation, Kafka allows increasing them dynamically (though not decreasing).  
docker exec -it kafka kafka-topics --bootstrap-server localhost:29092 --alter --topic raw\_financial\_transactions --partitions 6  
  
Note: Increasing partitions only benefits new messages. Existing messages remain in their original partitions. For full benefit, you might re-ingest data or create a new topic with more partitions.

Adjust Spark Parallelism (Conceptual/Manual):  
If your Spark job is constrained by partitions (i.e., less Spark executors/cores than partitions), it won't fully utilize the new partitions. Conceptually, you would:

Stop the current Spark job.

Increase the number of Spark executors or cores allocated to your financial data processing job (in docker-compose.yml or the spark-submit command).

Restart the Spark job.

Observe Performance under Increased Partitions/Parallelism:

Continue running simulate\_data.py at a high rate.

Monitor Grafana.

Verification:

Kafka Topic Description: docker exec -it kafka kafka-topics --bootstrap-server localhost:29092 --describe --topic raw\_financial\_transactions (Expected: Output shows 6 partitions).

Grafana Metrics: With sufficient producer load and increased Spark parallelism, you should observe:

Potentially higher overall throughput for the raw\_financial\_transactions topic.

Better distribution of load across Spark executors (if you could monitor individual executor metrics).

The Kafka consumer lag remaining stable or reducing faster under heavy load, indicating that the increased parallelism is effectively consuming the data.

This demonstrates how strategic Kafka partitioning, combined with corresponding consumer parallelism, is a key lever for scaling your streaming data pipelines.

Deep-Dive Addendum: DR & Runbooks

This addendum focuses on Disaster Recovery (DR) strategies and the creation of detailed runbooks, critical components for ensuring the resilience and rapid recovery of your enterprise data platform. These practices minimize downtime and data loss in the face of unforeseen incidents.

6.1. RPO and RTO in Context

Understanding and defining your Recovery Point Objective (RPO) and Recovery Time Objective (RTO) are fundamental to a robust disaster recovery strategy.

RPO (Recovery Point Objective): The maximum tolerable amount of data (measured in time) that can be lost after a disaster. It determines the frequency of your data backups or replication.

Example: If your RPO is 1 hour, you can afford to lose at most 1 hour of data. This means your backups or replication must occur at least hourly.

Implications for Data Platform:

Streaming Data (Kafka/MSK): For critical real-time data, RPO should ideally be near zero (minutes or seconds). This is achieved through continuous replication and durable message storage.

Batch Data (Delta Lake/S3): RPO can be longer (hours or days) depending on criticality, often achieved through frequent snapshots or replication of the object storage.

Databases (PostgreSQL/MongoDB/RDS/DocumentDB): RPO depends on the business impact of data loss, typically ranging from seconds (through transaction logs/replication) to hours (through daily backups).

RTO (Recovery Time Objective): The maximum tolerable duration of time within which a business process must be restored after a disaster to avoid unacceptable consequences.

Example: If your RTO is 4 hours, your entire data platform (or critical components) must be fully operational within 4 hours of a disaster being declared.

Implications for Data Platform:

Automation: Achieving low RTO requires significant automation in infrastructure provisioning (IaC), application deployment (CI/CD), and data recovery processes.

Testing: Regular DR drills and testing are essential to validate that the RTO can indeed be met.

Compute Provisioning: Rapid provisioning of compute resources (e.g., EC2, EMR, Glue) for data processing in a DR scenario.

Database Spin-up: Quick restoration or failover of databases.

6.2. Backup & Restore Verification

A backup strategy is only as good as its restore capabilities. Regular verification is non-negotiable.

Strategy Components:

Database Backups:

PostgreSQL/RDS: Automated snapshots, point-in-time recovery (PITR), or logical backups (pg\_dump).

MongoDB/DocumentDB: Automated backups, replica set snapshots, or oplog-based recovery.

Object Storage (MinIO/S3): Versioning, cross-region replication (for S3), or daily/hourly snapshots of critical data lake paths.

Kafka/MSK: While Kafka itself stores messages durably for a configured retention period, for long-term archival or disaster recovery, external backup mechanisms (e.g., Kafka Connect to S3, or replication to another cluster) might be needed. Consumer offsets also need to be managed.

Application Code & Configuration: Version-controlled in Git, automatically deployed via CI/CD. Docker images should be stored in registries (e.g., ECR, Docker Hub).

Verification Procedures:

Automated Validation: Implement automated jobs (e.g., Airflow DAGs) that periodically:

Restore a recent backup to a temporary, isolated environment.

Run data quality checks (using Great Expectations) on the restored data.

Perform smoke tests on critical data pipelines using the restored data.

Compare a sample of restored data with the source data (if feasible).

Manual Spot Checks: Regularly restore a small, critical dataset manually to ensure the process is well-documented and human-executable.

DR Drills: Conduct full-scale DR drills at least annually, simulating a disaster scenario (e.g., region outage) and executing the entire recovery playbook to validate RTO and RPO. Document any discrepancies and refine the plan.

6.3. Runbook Templates for Critical Systems

Runbooks are step-by-step guides for executing operational procedures, especially crucial during incidents or disaster recovery. They ensure consistency, reduce cognitive load under stress, and enable efficient handovers.

Purpose of a Runbook:

Standardization: Provides a consistent approach to common operational tasks and incident response.

Efficiency: Reduces the time to resolve incidents by providing clear, pre-defined steps.

Knowledge Transfer: Documents tribal knowledge, making operations less reliant on specific individuals.

Reduced Stress: Offers a calm, structured guide during high-pressure situations.

Core Runbook Sections (Template):

# Runbook: [Descriptive Title, e.g., "Restore Financial Transactions Delta Lake Table"]  
  
## 1. Overview  
  
\* \*\*Purpose:\*\* [Briefly explain the goal of this runbook, e.g., "To restore the financial\_transactions Delta Lake table from a daily backup due to data corruption or accidental deletion."]  
\* \*\*Affected System(s):\*\* [List systems, e.g., "financial\_transactions Delta Lake table (Raw and Curated zones), downstream BI reports, data analysts."]  
\* \*\*Trigger Condition:\*\* [When should this runbook be used? e.g., "Detection of data integrity issues in financial\_transactions table; accidental `DROP TABLE` or `DELETE`."]  
\* \*\*RTO/RPO Impact:\*\* [How does following this runbook affect RTO/RPO? e.g., "Expected RTO: 2 hours. Expected RPO: 24 hours (based on last daily snapshot)."]  
\* \*\*Severity:\*\* [e.g., "High (P1-P2) - Immediate business impact, data unavailability."]  
\* \*\*Owner:\*\* [Team/Individual responsible for this system, e.g., "Data Platform Team - Core Data Engineering"]  
  
## 2. Pre-requisites & Preparation  
  
\* \*\*Access:\*\*  
 \* AWS Console access (Administrator/PowerUser IAM Role).  
 \* AWS CLI configured with appropriate credentials.  
 \* Access to Terraform state for `terraform\_infra/environments/prod`.  
 \* SSH access to EMR/EC2 instances (if applicable).  
 \* Grafana/CloudWatch access for monitoring.  
\* \*\*Tools:\*\*  
 \* `terraform` CLI installed.  
 \* `aws cli` installed.  
 \* `kubectl` or `ecs-cli` (if containerized).  
 \* Python environment with `boto3`, `pyspark`, `delta-spark` installed.  
\* \*\*Information Needed:\*\*  
 \* Last known good S3 bucket path/prefix for the Delta table backup.  
 \* AWS Region.  
 \* Specific `transaction\_id` or timestamp of corrupted data (if known).  
 \* Relevant Jira/Incident ticket number.  
  
## 3. Execution Steps  
  
\*\*IMPORTANT:\*\* Follow steps sequentially. Document all commands and outputs in the incident ticket.  
  
### 3.1. Incident Declaration & Communication  
  
1. \*\*Declare Incident:\*\* Notify relevant stakeholders via [Communication Channel, e.g., Slack channel #data-incidents, PagerDuty].  
2. \*\*Create Incident Ticket:\*\* Log a new incident in Jira/ServiceNow with title "[INCIDENT] - [Brief Title]".  
3. \*\*Establish Bridge/War Room:\*\* Create a dedicated communication channel (e.g., Zoom/Slack Huddle) for the incident team.  
  
### 3.2. Isolate & Confirm Scope  
  
1. \*\*Stop Downstream Consumers:\*\* Temporarily halt all Spark jobs and other consumers reading from the `financial\_transactions` curated Delta table.  
 \* \*Command (Airflow):\* Pause relevant DAGs (`data\_transformation\_dag`).  
 \* \*Command (EMR/Glue):\* Terminate active Spark jobs or stop Glue triggers.  
2. \*\*Verify Data Corruption/Loss:\*\*  
 \* Connect to Spark environment (EMR/Glue/local).  
 \* Attempt to read the corrupted Delta table:  
 ```python  
 # PySpark snippet to read the table  
 spark.read.format("delta").load("s3://your-curated-bucket/financial\_transactions/").show()  
 ```  
 \* Confirm corruption/missing data (e.g., `SELECT COUNT(\*)` vs. expected, check for nulls, garbled data).  
3. \*\*Identify Last Known Good State:\*\*  
 \* Review S3 bucket versions/backups for `financial\_transactions` Delta table.  
 \* Consult data quality reports or audit logs for the last successful data load.  
  
### 3.3. Recovery Procedure  
  
1. \*\*Navigate to Backup Location:\*\*  
 \* Identify the S3 path of the last healthy backup or a specific version to restore.  
 \* Example: `s3://your-backup-bucket/daily\_snapshots/financial\_transactions\_2023-10-25/`  
2. \*\*Restore Data:\*\*  
 \* \*\*Option A: Delta Lake Time Travel (if applicable & sufficient):\*\* If the corruption is recent and within your Delta Lake retention, simply query a previous version.  
 ```python  
 # PySpark: Read a specific version  
 df = spark.read.format("delta").option("versionAsOf", <version\_number>).load("s3://your-curated-bucket/financial\_transactions/")  
 # Then, if needed, overwrite the current table with the restored data  
 df.write.format("delta").mode("overwrite").save("s3://your-curated-bucket/financial\_transactions/")  
 ```  
 \* \*\*Option B: Copy from Backup to Active Path:\*\* If time travel is not feasible or the backup is external.  
 \* \*\*Important:\*\* Consider renaming/moving the current corrupted table first (e.g., `s3 mv s3://current-table/ s3://corrupted-backup/`).  
 \* Copy the good data:  
 ```bash  
 # AWS CLI: Copy entire directory  
 aws s3 cp s3://your-backup-bucket/daily\_snapshots/financial\_transactions\_2023-10-25/ s3://your-curated-bucket/financial\_transactions/ --recursive  
 ```  
3. \*\*Run Post-Restore Data Quality Checks:\*\*  
 \* Execute targeted Great Expectations suites or custom validation scripts on the restored `financial\_transactions` table.  
 \* Verify key metrics: record count, sum of amounts, absence of nulls in critical columns.  
4. \*\*Resync Dependent Systems:\*\*  
 \* If any downstream systems consume directly from this Delta table, trigger a refresh or full reload for them.  
 \* Restart all downstream Spark jobs/Airflow DAGs that were paused in step 3.1.  
 \* Monitor their catch-up process.  
  
### 3.4. Verification & Handover  
  
1. \*\*Monitor System Health:\*\*  
 \* Verify all relevant Grafana dashboards (e.g., Delta Lake health, Spark job health, API throughput) show normal operation.  
 \* Check for new errors or warnings in CloudWatch/Grafana logs.  
2. \*\*Confirm Data Flow:\*\*  
 \* Send a few test transactions through the FastAPI API.  
 \* Verify they flow through Kafka and Spark and appear correctly in the Delta table.  
3. \*\*Communicate Resolution:\*\*  
 \* Update incident ticket with resolution details.  
 \* Notify stakeholders of service restoration.  
4. \*\*Clean Up:\*\* Remove any temporary files or resources used during recovery.  
  
## 4. Post-Incident Analysis  
  
\* \*\*Schedule Post-Mortem (within 24-48 hours):\*\* Conduct a blameless post-mortem meeting to identify root causes, contributing factors, and action items. (See Section 5.6.3 for template).  
\* \*\*Update Documentation:\*\* Update this runbook, relevant architectural diagrams, and data quality checks based on lessons learned.  
\* \*\*Improve Monitoring/Alerting:\*\* Adjust SLIs/SLOs or add new alerts to prevent recurrence or improve detection.  
\* \*\*Automate More:\*\* Identify manual steps in the recovery process that can be automated in future iterations.  
  
## Appendix G: Disaster Recovery (DR) Runbook Examples  
  
This appendix provides more detailed examples of common DR runbooks.  
  
### DR Runbook Example: Kafka Cluster Failover (Conceptual)  
  
\* \*\*Purpose:\*\* To failover to a secondary Kafka cluster (e.g., in a different AWS region or a mirrored cluster) in case of a primary cluster outage.  
\* \*\*Key Steps:\*\*  
 1. \*\*Confirm Primary Cluster Outage:\*\* Verify using Kafka tools (`kafka-topics.sh --describe`) and monitoring dashboards.  
 2. \*\*Update DNS/Client Configurations:\*\* Change application configurations (FastAPI, Spark) to point to the secondary Kafka cluster's bootstrap servers.  
 3. \*\*Verify New Message Production:\*\* Ensure FastAPI (or other producers) is successfully publishing to the secondary cluster.  
 4. \*\*Verify Consumer Offset Migration/Reset:\*\* Decide if consumer offsets need to be migrated or if consumers should start from the earliest available offset on the secondary cluster (this implies data loss up to that point).  
 5. \*\*Restart Consumers:\*\* Restart Spark streaming jobs and other Kafka consumers to connect to the secondary cluster.  
 6. \*\*Monitor Lag and Data Flow:\*\* Ensure consumers are catching up and data is flowing as expected.  
 7. \*\*Failback (Optional):\*\* Plan and execute a controlled failback to the primary cluster once it's restored and stable.  
  
### DR Runbook Example: Critical Database Restoration (Conceptual)  
  
\* \*\*Purpose:\*\* To restore a critical database (e.g., PostgreSQL for application metadata or Airflow) from a backup.  
\* \*\*Key Steps:\*\*  
 1. \*\*Identify Recovery Point:\*\* Determine the desired point in time for restoration based on RPO and known good state.  
 2. \*\*Spin Up New Database Instance:\*\* Provision a new RDS instance or Docker container for PostgreSQL. \*Do NOT restore over the corrupted production instance directly.\*  
 3. \*\*Perform Restore:\*\* Use AWS RDS snapshot restore, PITR, or `pg\_restore` from a logical backup file.  
 4. \*\*Validate Data Integrity:\*\* Run schema checks and sample data queries to confirm successful restoration.  
 5. \*\*Update Application Configuration:\*\* Modify application (FastAPI, Airflow) database connection strings to point to the newly restored instance.  
 6. \*\*Restart Applications:\*\* Restart services dependent on the database.  
 7. \*\*Monitor:\*\* Check application logs and database metrics for healthy operation.  
  
### DR Runbook Example: Airflow Metastore Database Recovery (Conceptual)  
  
\* \*\*Purpose:\*\* Recover the Airflow metastore database (PostgreSQL in this case) from corruption or loss. This is critical as Airflow DAGs and their states are stored here.  
\* \*\*Key Steps:\*\*  
 1. \*\*Stop Airflow Components:\*\* Halt all Airflow scheduler, webserver, and worker instances.  
 2. \*\*Restore Metastore Database:\*\* Follow the "Critical Database Restoration" runbook (above) specifically for the Airflow metastore.  
 3. \*\*Verify Database Integrity:\*\* Run `airflow db check` or manually inspect tables for data.  
 4. \*\*Start Airflow Metastore First:\*\* Bring up \*only\* the PostgreSQL service for Airflow.  
 5. \*\*Run `airflow db upgrade`:\*\* If necessary, run this command to ensure schema is up-to-date with Airflow version.  
 6. \*\*Start Airflow Scheduler and Webserver:\*\* Bring up the core Airflow components.  
 7. \*\*Inspect DAGs:\*\* Verify all DAGs are visible, and their last run states are consistent with the restored point. DAGs might need to be re-run or marked as successful if the state is lost.  
 8. \*\*Start Airflow Workers:\*\* Bring up the worker processes.  
 9. \*\*Monitor:\*\* Check Airflow UI and logs for any issues.

Deep-Dive Addendum: Testing & Observability Patterns

This addendum provides comprehensive details on testing approaches and observability patterns essential for building reliable, high-quality, and maintainable enterprise data platforms. It covers various testing levels and the critical components for gaining actionable insights into system health and performance.

5.4. Comprehensive Testing Approaches

Robust testing is vital to ensure the reliability, accuracy, and performance of data pipelines.

Unit Tests:

Purpose: Verify the correctness of individual, isolated components or functions.

Application: FastAPI endpoint logic, PySpark transformation functions (e.g., specific UDFs, data cleansing functions), and any custom Python utilities.

Tools: pytest for Python code.

Sample Snippet (fastapi\_app/tests/unit/test\_api.py):

# fastapi\_app/tests/unit/test\_api.py  
import pytest  
from fastapi.testclient import TestClient  
# Assuming your FastAPI app is structured like app.main.app  
from fastapi\_app.app.main import app  
from datetime import datetime  
  
client = TestClient(app)  
  
def test\_read\_main():  
 response = client.get("/")  
 assert response.status\_code == 200  
 assert response.json() == {"message": "Welcome to Financial/Insurance Data Ingestor API!"}  
  
def test\_ingest\_financial\_transaction\_invalid\_data():  
 response = client.post("/ingest-financial-transaction/", json={  
 "transaction\_id": "FT-001",  
 "timestamp": "invalid-date", # Invalid timestamp  
 "account\_id": "ACC-XYZ",  
 "amount": "not-a-number", # Invalid amount  
 "currency": "USD",  
 "transaction\_type": "debit"  
 })  
 assert response.status\_code == 422 # Unprocessable Entity due to validation error  
 assert "validation error" in response.text

Integration Tests:

Purpose: Verify that different components of the pipeline work together as expected.

Application: FastAPI to Kafka, Kafka to Spark (Streaming), Spark transformations.

Tools: docker-compose.test.yml, pytest, Testcontainers (for robust service orchestration in tests), Kafka client libraries, MinIO SDK.

Conceptual docker-compose.test.yml for Integration Tests:

This file defines a stripped-down set of services specifically for integration testing, focusing on inter-service communication.

# docker-compose.test.yml (for integration testing)  
version: '3.8'  
services:  
 zookeeper:  
 image: confluentinc/cp-zookeeper:7.4.0  
 environment:  
 ZOOKEEPER\_CLIENT\_PORT: 2181  
 healthcheck:  
 test: ["CMD", "sh", "-c", "nc -z localhost 2181"]  
 interval: 10s  
 timeout: 5s  
 retries: 5  
  
 kafka:  
 image: confluentinc/cp-kafka:7.4.0  
 depends\_on:  
 zookeeper:  
 condition: service\_healthy  
 ports:  
 - "9092:9092"  
 environment:  
 KAFKA\_BROKER\_ID: 1  
 KAFKA\_ZOOKEEPER\_CONNECT: 'zookeeper:2181'  
 KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT\_HOST://localhost:9092  
 KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT\_HOST:PLAINTEXT  
 KAFKA\_INTER\_BROKER\_LISTENER\_NAME: PLAINTEXT  
 KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1  
 healthcheck:  
 test: ["CMD", "sh", "-c", "kafka-topics --bootstrap-server localhost:9092 --list"]  
 interval: 10s  
 timeout: 5s  
 retries: 5  
  
 minio:  
 image: minio/minio:latest  
 ports:  
 - "9000:9000"  
 environment:  
 MINIO\_ROOT\_USER: test\_user  
 MINIO\_ROOT\_PASSWORD: test\_password  
 command: server /data --console-address ":9000"  
 healthcheck:  
 test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]  
 interval: 30s  
 timeout: 20s  
 retries: 3  
  
 fastapi\_ingestor:  
 build: ./fastapi\_app  
 environment:  
 KAFKA\_BROKER: kafka:29092  
 KAFKA\_TOPIC: raw\_data\_test  
 depends\_on:  
 kafka:  
 condition: service\_healthy  
 healthcheck:  
 test: ["CMD", "curl", "-f", "http://localhost:8000/health || exit 1"]  
 interval: 5s  
 timeout: 3s  
 retries: 5  
  
 # Spark service for integration testing (can be a standalone driver in test, or a small cluster)  
 spark-test-runner:  
 image: bitnami/spark:3.5.0  
 depends\_on:  
 kafka:  
 condition: service\_healthy  
 minio:  
 condition: service\_healthy  
 environment:  
 SPARK\_MASTER\_URL: "local[\*]" # Run Spark in local mode for test  
 KAFKA\_BROKER: kafka:29092  
 MINIO\_HOST: minio  
 MINIO\_ACCESS\_KEY: test\_user  
 MINIO\_SECRET\_KEY: test\_password  
 volumes:  
 - ./pyspark\_jobs:/opt/bitnami/spark/data/pyspark\_jobs # Mount jobs  
 - ./data/test\_spark\_output:/tmp/spark\_output # Output dir for tests  
 # No exposed ports unless needed for Spark UI inspection during debug  
 command: ["tail", "-f", "/dev/null"] # Keep container running

Conceptual Integration Test (fastapi\_app/tests/integration/test\_data\_flow.py):

This example uses docker-compose command directly, but Testcontainers provides a more Pythonic way to manage test lifecycle.

# fastapi\_app/tests/integration/test\_data\_flow.py  
import pytest  
import requests  
import subprocess  
import time  
from kafka import KafkaConsumer  
import json  
import os  
from datetime import datetime  
from minio import Minio # Assuming minio client library is installed  
  
# Define the path to your test compose file  
COMPOSE\_FILE = os.path.join(os.path.dirname(\_\_file\_\_), '../../docker-compose.test.yml')  
  
@pytest.fixture(scope="module")  
def docker\_services(request):  
 """Starts and stops docker-compose services for integration tests."""  
 print(f"\nStarting Docker services from: {COMPOSE\_FILE}")  
 # Ensure services are down first  
 subprocess.run(["docker", "compose", "-f", COMPOSE\_FILE, "down", "-v"], check=True)  
 subprocess.run(["docker", "compose", "-f", COMPOSE\_FILE, "up", "--build", "-d"], check=True)  
  
 # Wait for FastAPI to be healthy  
 api\_url = "http://localhost:8000"  
 for \_ in range(30): # Wait up to 30 seconds  
 try:  
 response = requests.get(f"{api\_url}/health")  
 if response.status\_code == 200:  
 print("FastAPI is healthy.")  
 break  
 except requests.exceptions.ConnectionError:  
 pass  
 time.sleep(1)  
 else:  
 pytest.fail("FastAPI did not become healthy in time.")  
  
 # Wait for Kafka to be healthy  
 kafka\_broker = "localhost:9092"  
 print(f"Waiting for Kafka at {kafka\_broker}...")  
 # More robust check could involve kafka-topics --list or similar  
 time.sleep(10) # Give Kafka some time to initialize  
  
 # Wait for MinIO to be healthy and create test bucket  
 minio\_client = Minio("localhost:9000", access\_key="test\_user", secret\_key="test\_password", secure=False)  
 bucket\_name = "raw-data-bucket-test"  
 if not minio\_client.bucket\_exists(bucket\_name):  
 minio\_client.make\_bucket(bucket\_name)  
 print(f"MinIO healthy and bucket '{bucket\_name}' ready.")  
  
 yield # Tests run here  
  
 print("Stopping Docker services.")  
 subprocess.run(["docker", "compose", "-f", COMPOSE\_FILE, "down", "-v"], check=True)  
  
def test\_end\_to\_end\_financial\_transaction\_flow(docker\_services):  
 """Tests ingestion via FastAPI, consumption via Kafka, and processing to Delta Lake."""  
 api\_url = "http://localhost:8000"  
 kafka\_broker = "localhost:9092"  
 kafka\_topic = "raw\_data\_test" # As defined in docker-compose.test.yml  
 minio\_host = "localhost:9000"  
 minio\_access\_key = "test\_user"  
 minio\_secret\_key = "test\_password"  
 minio\_bucket = "raw-data-bucket-test"  
 spark\_output\_dir = "/tmp/spark\_output/financial\_data\_delta" # Matches volume in spark-test-runner  
  
 # 1. Send data via FastAPI  
 transaction\_data = {  
 "transaction\_id": "INT-001",  
 "timestamp": datetime.now().isoformat(),  
 "account\_id": "ACC-INT-001",  
 "amount": 123.45,  
 "currency": "USD",  
 "transaction\_type": "deposit"  
 }  
 response = requests.post(f"{api\_url}/ingest-financial-transaction/", json=transaction\_data)  
 assert response.status\_code == 200  
 assert response.json()["message"] == "Financial transaction ingested successfully"  
  
 # 2. Consume data from Kafka and verify (optional, for explicit check)  
 consumer = KafkaConsumer(  
 kafka\_topic,  
 bootstrap\_servers=[kafka\_broker],  
 auto\_offset\_reset='earliest',  
 enable\_auto\_commit=False,  
 group\_id='test-consumer-group',  
 value\_deserializer=lambda x: json.loads(x.decode('utf-8'))  
 )  
 consumed\_message = None  
 start\_time = time.time()  
 for msg in consumer:  
 consumed\_message = msg.value  
 print(f"Consumed: {consumed\_message}")  
 if consumed\_message.get("transaction\_id") == transaction\_data["transaction\_id"]:  
 break  
 if time.time() - start\_time > 10: # Timeout after 10 seconds  
 break  
 consumer.close()  
 assert consumed\_message is not None, "Did not consume message from Kafka"  
 assert consumed\_message["transaction\_id"] == transaction\_data["transaction\_id"]  
  
 # 3. Trigger Spark job to process from Kafka to Delta Lake  
 # Create a simplified Spark job script for testing that reads from Kafka  
 # and writes to Delta Lake in MinIO.  
 # Example: pyspark\_jobs/streaming\_consumer\_test.py  
 # This script needs to be mounted into spark-test-runner  
 # For this test, we'll assume a simple job that writes raw Kafka messages to Delta Lake.  
 spark\_submit\_command = [  
 "docker", "exec", "spark-test-runner", "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=test\_user",  
 "--conf", "spark.hadoop.fs.s3a.secret.key=test\_password",  
 "--conf", "spark.hadoop.fs.s3a.path.style.access=true",  
 "pyspark\_jobs/streaming\_consumer\_test.py", # This script will read from Kafka and write to MinIO  
 kafka\_topic,  
 "kafka:29092", # Kafka broker for Spark  
 f"s3a://{minio\_bucket}/{spark\_output\_dir.replace('/tmp/spark\_output/', '')}" # S3a path  
 ]  
 print(f"Running Spark job: {' '.join(spark\_submit\_command)}")  
 spark\_process = subprocess.run(spark\_submit\_command, capture\_output=True, text=True, check=True)  
 print(spark\_process.stdout)  
 print(spark\_process.stderr)  
 time.sleep(15) # Give Spark time to consume and write  
  
 # 4. Verify data in Delta Lake (MinIO)  
 minio\_client = Minio(minio\_host, access\_key=minio\_access\_key, secret\_key=minio\_secret\_key, secure=False)  
 # List objects in the Delta Lake path to confirm data written  
 found\_delta\_files = False  
 for obj in minio\_client.list\_objects(minio\_bucket, prefix=f"{spark\_output\_dir.replace('/tmp/spark\_output/', '')}/", recursive=True):  
 if "\_delta\_log" in obj.object\_name or ".parquet" in obj.object\_name:  
 found\_delta\_files = True  
 break  
 assert found\_delta\_files, "No Delta Lake files found in MinIO after Spark job execution."  
 # Optional: Read data back from Delta Lake using a local SparkSession (if `pyspark` is installed locally)  
 # from pyspark.sql import SparkSession  
 # spark\_read = (SparkSession.builder.appName("DeltaReadTest")  
 # .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 # .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 # .config("spark.hadoop.fs.s3a.endpoint", f"http://{minio\_host}")  
 # .config("spark.hadoop.fs.s3a.access.key", minio\_access\_key)  
 # .config("spark.hadoop.fs.s3a.secret.key", minio\_secret\_key)  
 # .config("spark.hadoop.fs.s3a.path.style.access", "true")  
 # .getOrCreate())  
 #  
 # delta\_df = spark\_read.read.format("delta").load(f"s3a://{minio\_bucket}/{spark\_output\_dir.replace('/tmp/spark\_output/', '')}")  
 # delta\_df.show()  
 # assert delta\_df.count() >= 1 # At least one row should be there  
 # assert delta\_df.filter(delta\_df.value.contains(transaction\_data["transaction\_id"])).count() == 1  
 # spark\_read.stop()

Note for streaming\_consumer\_test.py:

You'd need a simple PySpark script like this in pyspark\_jobs/:

# pyspark\_jobs/streaming\_consumer\_test.py  
import sys  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import col, from\_json  
from pyspark.sql.types import StructType, StringType, FloatType, TimestampType, MapType  
  
def create\_spark\_session(app\_name):  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10\_2.12:3.5.0,io.delta:delta-core\_2.12:2.4.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if len(sys.argv) != 4:  
 print("Usage: streaming\_consumer\_test.py <kafka\_topic> <kafka\_broker> <delta\_output\_path>")  
 sys.exit(-1)  
  
 kafka\_topic = sys.argv[1]  
 kafka\_broker = sys.argv[2]  
 delta\_output\_path = sys.argv[3]  
  
 spark = create\_spark\_session("KafkaToDeltaTest")  
  
 # Define schema for the incoming Kafka message value (adjust as per your FastAPI data)  
 schema = StructType() \  
 .add("transaction\_id", StringType()) \  
 .add("timestamp", StringType()) \  
 .add("account\_id", StringType()) \  
 .add("amount", FloatType()) \  
 .add("currency", StringType()) \  
 .add("transaction\_type", StringType()) \  
 .add("merchant\_id", StringType(), True) \  
 .add("category", StringType(), True)  
  
 # Read from Kafka  
 kafka\_df = (spark.readStream  
 .format("kafka")  
 .option("kafka.bootstrap.servers", kafka\_broker)  
 .option("subscribe", kafka\_topic)  
 .option("startingOffsets", "earliest")  
 .load())  
  
 # Parse the value column from Kafka  
 parsed\_df = kafka\_df.selectExpr("CAST(value AS STRING) as json\_value") \  
 .select(from\_json(col("json\_value"), schema).alias("data")) \  
 .select("data.\*")  
  
 # Write to Delta Lake  
 query = (parsed\_df.writeStream  
 .format("delta")  
 .outputMode("append")  
 .option("checkpointLocation", f"{delta\_output\_path}/\_checkpoints")  
 .start(delta\_output\_path))  
  
 query.awaitTermination(30) # Run for 30 seconds to capture test data  
 query.stop()  
 spark.stop()

Data Quality Tests:

Purpose: Ensure accuracy, completeness, consistency, validity, and timeliness of data.

Application: Integrate data quality checks within Spark jobs or as separate validation steps.

Tools: Great Expectations, Pydantic (for schema validation), custom validation logic.

Conceptual Pact Contract Testing Snippet:

Pact is a "consumer-driven contract" testing tool. This would typically be a separate test suite (pyspark\_jobs/tests/contract/financial\_transaction\_consumer\_pact.py).

# pyspark\_jobs/tests/contract/financial\_transaction\_consumer\_pact.py  
import pytest  
from pact import Consumer, Provider  
from pyspark.sql import SparkSession  
from pyspark.sql.types import StructType, StringType, FloatType, TimestampType  
import json  
from datetime import datetime  
from pyspark.sql.functions import current\_timestamp  
  
# Define Pact mock server details  
PACT\_MOCK\_HOST = 'localhost'  
PACT\_MOCK\_PORT = 1234  
PACT\_DIR = './pacts' # Directory where pact files will be written  
  
# Define the consumer and provider for this contract  
consumer = Consumer('FinancialTransactionSparkConsumer')  
provider = Provider('FastAPIIngestor')  
  
@pytest.fixture(scope='module')  
def pact\_spark\_session():  
 """Fixture for a local SparkSession to be used in contract tests."""  
 spark = (SparkSession.builder  
 .appName("PactSparkConsumer")  
 .master("local[\*]")  
 .getOrCreate())  
 yield spark  
 spark.stop()  
  
@pytest.fixture(scope='module')  
def pact():  
 """Starts and stops the Pact mock service."""  
 pact\_instance = consumer.has\_pact\_with(  
 provider,  
 host\_name=PACT\_MOCK\_HOST,  
 port=PACT\_MOCK\_PORT,  
 pact\_dir=PACT\_DIR  
 )  
 print(f"\nStarting Pact mock service on {PACT\_MOCK\_HOST}:{PACT\_MOCK\_PORT}")  
 pact\_instance.start\_service()  
 yield pact\_instance  
 print("Stopping Pact mock service")  
 pact\_instance.stop\_service()  
  
def test\_spark\_can\_process\_financial\_transaction\_from\_kafka(pact, pact\_spark\_session):  
 """  
 Verifies that the Spark consumer can correctly process a financial transaction  
 message from Kafka, based on the contract with the FastAPI Ingestor.  
 """  
 # Define the expected message structure from the producer (FastAPI)  
 expected\_message\_body = {  
 "transaction\_id": "TRANS-12345",  
 "timestamp": "2023-10-26T14:30:00.000Z",  
 "account\_id": "ACC-FIN-001",  
 "amount": 500.75,  
 "currency": "USD",  
 "transaction\_type": "credit",  
 "merchant\_id": "MER-ABC",  
 "category": "utilities"  
 }  
  
 # Define the interaction for the Kafka message  
 (pact  
 .given('a financial transaction is published to Kafka')  
 .upon\_receiving('a Kafka message with financial transaction data')  
 .with\_message(  
 'application/json', # Mime type of the message  
 json.dumps(expected\_message\_body) # The expected message content  
 ))  
  
 with pact:  
 # Simulate receiving the message as if from Kafka  
 # In a real Spark job, this would be the actual Kafka consumer logic  
 # For a contract test, we feed the expected message directly to the Spark logic  
 # Convert the expected message body to a Spark DataFrame  
 schema = StructType() \  
 .add("transaction\_id", StringType()) \  
 .add("timestamp", StringType()) \  
 .add("account\_id", StringType()) \  
 .add("amount", FloatType()) \  
 .add("currency", StringType()) \  
 .add("transaction\_type", StringType()) \  
 .add("merchant\_id", StringType(), True) \  
 .add("category", StringType(), True)  
  
 # Create a DataFrame from the single expected message  
 df\_from\_kafka = pact\_spark\_session.createDataFrame([expected\_message\_body], schema=schema)  
  
 # Apply a dummy transformation that resembles your actual Spark job logic  
 # This ensures your Spark code can parse and work with the contract-defined schema  
 processed\_df = df\_from\_kafka.withColumn("processed\_at", current\_timestamp())  
  
 # Collect and assert the processed data  
 collected\_data = processed\_df.collect()  
 assert len(collected\_data) == 1  
 assert collected\_data[0]['transaction\_id'] == expected\_message\_body['transaction\_id']  
 assert collected\_data[0]['amount'] == expected\_message\_body['amount']  
 assert 'processed\_at' in collected\_data[0]

Performance and Load Testing:

Purpose: Assess the system's performance under expected and peak load conditions, identify bottlenecks, and ensure it meets non-functional requirements (e.g., latency, throughput).

Application: Use tools to simulate high volumes of data being sent to the FastAPI endpoint and monitor Kafka, Spark, and database performance using Grafana dashboards.

Tools: Locust (for API load testing), JMeter, Spark UI, Grafana.

5.6. Observability: From Configuration to Practice

Effective observability moves beyond collecting data to enabling actionable insights and proactive problem-solving.

5.6.1. Defining SLIs and SLOs

SLI (Service Level Indicator): A quantitative measure of some aspect of the level of service that is provided.

SLO (Service Level Objective): A target value or range for an SLI that defines the desired level of service.

5.6.2. Alert Fatigue Mitigation

Contextual Alerts: Use alert annotations to provide immediate context, links to runbooks, and suggested remediation steps.

Annotation Templates: Standardize alert messages to include:

summary: What happened? (e.g., "High Kafka consumer lag detected")

description: Why is this important? (e.g., "Spark job is falling behind, data freshness impacted")

remediation: What are the first 3 steps to take? (e.g., "1. Check Spark job logs. 2. Verify Spark cluster resources. 3. Scale up Spark executors.")

dashboard\_link: Link to the relevant Grafana dashboard.

runbook\_link: Link to the detailed runbook in your repository (e.g., /runbooks/kafka\_consumer\_lag.md).

Muting Strategy: Define clear policies for muting alerts during planned maintenance, backfills, or specific development activities. Automate muting where possible (e.g., via Airflow operators triggering alert suppression during maintenance windows).

Escalation Policies: Use PagerDuty, Opsgenie, or similar tools for structured escalation paths and on-call rotations.

5.6.3. Sample Incident Review Template (“Post-Mortem Lite”)

A brief, structured review process for every significant alert or incident to foster continuous learning and prevent recurrence.

# Incident Review Template (Post-Mortem Lite)  
  
\*\*Incident Title:\*\* [Brief, descriptive title, e.g., "High Kafka Consumer Lag on Raw Financial Data Topic"]  
\*\*Date/Time of Incident:\*\* [YYYY-MM-DD HH:MM UTC] - [YYYY-MM-DD HH:MM UTC]  
\*\*Detected By:\*\* [Alert Name (e.g., KafkaConsumerLagHigh), or Manual Observation]  
  
\*\*Impact:\*\*  
\* What broke? [e.g., "Spark Structured Streaming job for financial data"]  
\* Who was affected? [e.g., "Downstream BI reports reliant on real-time financial data, data analysts"]  
\* What was the business impact? [e.g., "Delayed revenue reporting by 2 hours, potential for stale insights"]  
\* SLO Violation(s): [List violated SLOs, e.g., "Kafka Consumer Lag SLO ($<10,000$ msgs) violated for 30 minutes"]  
  
\*\*Initial Root Cause (Hypothesis):\*\*  
\* [e.g., "Under-provisioned Spark executor memory causing excessive garbage collection and slow processing."]  
  
\*\*Mitigation Steps Taken:\*\*  
\* [e.g., "Increased Spark job executor memory from 6GB to 12GB."]  
\* [e.g., "Restarted Spark Structured Streaming job."]  
  
\*\*Resolution:\*\*  
\* [e.g., "Consumer lag caught up within 15 minutes after increasing memory."]  
  
\*\*Lessons Learned:\*\*  
\* \*\*System:\*\* [e.g., "Our Spark resource allocation was insufficient for peak ingestion rates."]  
\* \*\*Process:\*\* [e.g., "Our alert threshold for consumer lag was too high, delaying detection."]  
\* \*\*Tools:\*\* [e.g., "Grafana dashboards need to be updated to show Spark GC metrics more prominently."]  
  
\*\*Action Items (with Owners & Due Dates):\*\*  
\* \*\*[Action 1]:\*\* Increase default Spark executor memory in `docker-compose.yml` for local dev.  
 \* \*\*Owner:\*\* [Data Engineer A]  
 \* \*\*Due Date:\*\* [YYYY-MM-DD]  
\* \*\*[Action 2]:\*\* Update Kafka Consumer Lag alert threshold in Grafana Alloy config.  
 \* \*\*Owner:\*\* [Data Engineer B]  
 \* \*\*Due Date:\*\* [YYYY-MM-DD]  
\* \*\*[Action 3]:\*\* Create a new runbook for "Spark Job Resource Exhaustion" with specific debugging steps.  
 \* \*\*Owner:\*\* [Data Engineer C]  
 \* \*\*Due Date:\*\* [YYYY-MM-DD]  
\* \*\*[Action 4]:\*\* Review historical Kafka ingestion patterns to better predict peak loads.  
 \* \*\*Owner:\*\* [Data Analyst D]  
 \* \*\*Due Date:\*\* [YYYY-MM-DD]  
  
\*\*Link to relevant dashboards/logs:\*\*  
\* Grafana Dashboard: [URL]  
\* Spark UI Logs: [URL]  
\* Kafka Logs: [URL]

5.7. Common Gotchas & Debug Playbooks

Practical troubleshooting steps for common issues. Each point implies a conceptual "debug flowchart" or checklist for triage.

Kafka "Stuck" Consumers:

Symptoms: High Kafka consumer lag (messages piling up), Spark Structured Streaming job not processing, KafkaConsumerLagHigh alert.

Triage Flow:

Check Spark Job Status: Is the Spark Structured Streaming job consuming from Kafka actually running? (http://localhost:8080 for Spark UI). Look at "Running Applications" and "Completed Applications." Is your job listed? Check its current status, stages, and tasks.

Review Spark Logs: Examine executor logs and driver logs for specific errors (deserialization, processing exceptions, OutOfMemoryError), continuous restarts, or backpressure warnings.

Inspect Kafka Offsets: Use kafka-consumer-groups.sh to get current offsets and confirm lag directly.  
# Conceptual command to inspect Kafka consumer group offsets  
docker exec -it kafka kafka-consumer-groups.sh --bootstrap-server kafka:29092 --describe --group <your\_consumer\_group\_name>

Verify Kafka Broker Health: Check Kafka and Zookeeper container logs for any errors (e.g., disk full, network issues).

Grafana Consumer Lag Panel: Monitor a pre-built Grafana dashboard (see "Health-Check Dashboard" in Section 8.1, not in this addendum but detailed in the full guide) showing consumer lag metrics, often providing historical context.

Action: If Spark job is failing, debug code logic. If Spark is too slow, scale up Spark executors/cores or optimize transformations. If Kafka is unhealthy, investigate broker issues. Refer to runbooks/kafka\_consumer\_lag.md.

Delta Lake Writes Failing under Schema Drift:

Symptoms: Spark writes to Delta Lake fail with schema mismatch errors, AnalysisException: Cannot resolve '...' given input columns, Schema is not compatible.

Triage Flow:

Identify Schema Change: Compare incoming DataFrame schema with the existing Delta table schema. The error message usually highlights the problematic column or type.

Review Error Message: Understand if a column was added, removed, renamed, or its type changed.

Decide on Schema Evolution Strategy:

mergeSchema (Recommended for evolution): Allows adding new columns or reordering existing ones without breaking the write.  
# PySpark: Enable schema merging for writes  
df.write.format("delta") \  
 .mode("append") \  
 .option("mergeSchema", "true") \  
 .save("/path/to/delta\_table")

overwriteSchema (Use with EXTREME CAUTION): Overwrites the entire table schema. This is destructive and can lead to data loss or make historical data unreadable if not managed carefully.  
# PySpark: Overwrite schema (use with EXTREME CAUTION)  
df.write.format("delta") \  
 .mode("overwrite") \  
 .option("overwriteSchema", "true") \  
 .save("/path/to/delta\_table")

Action: Apply mergeSchema for non-breaking changes. For breaking changes, plan a migration (e.g., creating a new table version, backfilling, or data re-processing).

Docker Networking Pitfalls on M1/Mac vs. Windows:

Symptoms: Containers cannot communicate with each other or with services on the host machine (e.g., fastapi\_ingestor cannot reach kafka), Connection Refused, Name or service not known.

Triage Flow:

Check docker-compose.yml:

Service Names: Ensure containers reference each other by their service name within the Docker network (e.g., kafka:29092, not localhost:9092).

Port Mappings: Verify correct ports mappings (e.g., 9092:9092) for external host access. Remember that internal and external ports can differ.

depends\_on: Use condition: service\_healthy to ensure dependencies are fully ready before a dependent service tries to connect.

host.docker.internal (Mac/Windows Specific): If a container needs to connect to a service running directly on the host machine (e.g., a locally run Python script acting as a mock API), use host.docker.internal as the hostname.  
# Example: A custom script inside container needs to connect to host-bound service  
my\_container:  
 environment:  
 HOST\_API\_URL: http://host.docker.internal:8080

Firewall Rules: On Windows, explicitly check and configure your firewall rules to allow inbound connections to the exposed Docker ports. Docker Desktop generally manages this for macOS, but custom firewall settings can interfere.

Network Inspection: Use docker inspect <container\_id> or docker network inspect <network\_name> to view container IP addresses and network configurations, which can help diagnose routing issues.

Action: Correct hostnames/IPs in environment variables, verify port mappings, adjust host firewall rules.

7.4. Sample Benchmarking Harness & Observed Data

To truly understand performance, theoretical sizing must be combined with empirical measurements. This section outlines a conceptual benchmarking harness and provides illustrative observed data, directly contributing to testing and observability of the platform.

Benchmarking Harness Components:

Load Generator (Locust): Simulates concurrent users sending financial/insurance data to the FastAPI ingestion API.

FastAPI Ingestor: Receives data and publishes it to Kafka.

Kafka Cluster: Buffers the incoming data stream.

Spark Structured Streaming Job: Consumes from Kafka, performs basic transformations (e.g., parsing, schema enforcement), and writes to the Raw Delta Lake zone in MinIO.

Metrics Collector (Grafana Alloy): Collects metrics from FastAPI, Kafka, Spark, and cAdvisor.

Monitoring (Grafana): Visualizes end-to-end latency, throughput, and resource utilization.

Conceptual Benchmarking Steps:

Setup Environment: Bring up the full Advanced Track Docker Compose environment.

Run Load Generator: Start Locust to simulate X users sending Y requests per second to FastAPI.

Monitor Metrics: Observe Grafana dashboards for key metrics:

FastAPI request rate (RPS) and latency.

Kafka producer throughput (messages/sec, MB/sec).

Kafka consumer throughput and lag (messages/sec, messages in backlog).

Spark streaming batch processing time and records processed.

CPU, memory, network utilization for all Docker containers (via cAdvisor).

Analyze Data: Record and analyze average/p99 latency, throughput, and resource bottlenecks.

Scale Up/Down: Repeat tests by varying Kafka partitions, Spark executor counts, cores, and memory to identify optimal configurations for different load levels.

Conceptual Locust Load Test Script (locust\_fastapi\_ingestor.py):

# locust\_fastapi\_ingestor.py  
"""  
Locust load test script for the FastAPI Data Ingestor.  
This script defines two tasks to simulate traffic:  
1. ingest\_financial\_transaction: Sends mock financial transaction data.  
2. ingest\_insurance\_claim: Sends mock insurance claim data.  
  
The user can configure the host, number of users, and spawn rate via the Locust UI  
(usually http://localhost:8089 after running `locust -f locust\_fastapi\_ingestor.py`).  
"""  
from locust import HttpUser, task, between  
import json  
from datetime import datetime, timedelta  
import random  
  
class FinancialDataUser(HttpUser):  
 """  
 User class that simulates sending financial and insurance data to the FastAPI ingestor.  
 """  
 # Wait time between requests for each simulated user.  
 # This helps simulate more realistic user behavior rather than hammering the API constantly.  
 wait\_time = between(0.1, 0.5) # Simulate delay between requests (0.1 to 0.5 seconds)  
  
 # The host URL for the FastAPI application. This should match the exposed port in docker-compose.  
 # In a local Docker Compose setup, FastAPI is often exposed on localhost:8000.  
 host = "http://localhost:8000" # Target FastAPI endpoint  
  
 @task(1) # This task has a weight of 1, meaning it will be executed proportionally to other tasks.  
 def ingest\_financial\_transaction(self):  
 """  
 Simulates sending a financial transaction POST request to the FastAPI ingestor.  
 Generates realistic-looking mock data for a financial transaction.  
 """  
 transaction\_data = {  
 "transaction\_id": f"FT-{datetime.now().strftime('%Y%m%d%H%M%S%f')}-{random.randint(1000, 9999)}",  
 "timestamp": datetime.now().isoformat(),  
 "account\_id": f"ACC-{random.randint(100000, 999999)}",  
 "amount": round(random.uniform(1.0, 10000.0), 2), # Random amount between 1.00 and 10000.00  
 "currency": random.choice(["USD", "EUR", "GBP", "JPY"]), # Random currency  
 "transaction\_type": random.choice(["debit", "credit", "transfer", "payment"]), # Random type  
 "merchant\_id": f"MER-{random.randint(100, 999)}" if random.random() > 0.3 else None, # Optional merchant ID  
 "category": random.choice(["groceries", "utilities", "salary", "entertainment", "transport", "housing", "healthcare", "education"])  
 }  
 # Send the POST request. The 'name' argument groups requests in Locust's statistics.  
 self.client.post("/ingest-financial-transaction/", json=transaction\_data, name="/ingest-financial-transaction")  
  
 @task(1) # This task also has a weight of 1.  
 def ingest\_insurance\_claim(self):  
 """  
 Simulates sending an insurance claim POST request to the FastAPI ingestor.  
 Generates realistic-looking mock data for an insurance claim.  
 """  
 claim\_data = {  
 "claim\_id": f"IC-{datetime.now().strftime('%Y%m%d%H%M%S%f')}-{random.randint(1000, 9999)}",  
 "timestamp": datetime.now().isoformat(),  
 "policy\_number": f"POL-{random.randint(1000000, 9999999)}",  
 "claim\_amount": round(random.uniform(500.0, 50000.0), 2), # Random amount  
 "claim\_type": random.choice(["auto", "health", "home", "life", "property"]), # Random claim type  
 "claim\_status": random.choice(["submitted", "under\_review", "approved", "rejected", "paid"]), # Random status  
 "customer\_id": f"CUST-{random.randint(10000, 99999)}",  
 "incident\_date": (datetime.now() - timedelta(days=random.randint(0, 365))).isoformat() # Incident date within last year  
 }  
 # Send the POST request.  
 self.client.post("/ingest-insurance-claim/", json=claim\_data, name="/ingest-insurance-claim")

Observed Throughput and Latency (Illustrative for Local Dev Environment):

These figures are conceptual and will vary significantly based on your machine's hardware, other running processes, and exact configuration. They serve as a guide for what to measure and expect. Real-world results will necessitate profiling against your specific hardware and workloads.

Key Takeaways from Benchmarking:

Initial Bottleneck Identification: Often, the FastAPI instance itself or the underlying network I/O on the host machine can become the initial bottleneck if not optimized or scaled adequately.

Scaling Kafka: Increasing the number of Kafka partitions (and ensuring a corresponding increase in Kafka consumer parallelism) is a primary way to scale Kafka's throughput.

Scaling Spark: Adding more Spark executors and allocating more cores and memory per executor directly leads to higher data processing throughput. However, this also increases resource consumption and can quickly saturate a local development machine.

Disk I/O Impact: The performance of MinIO (simulating S3) and the Delta Lake operations are heavily influenced by the underlying disk speed and I/O capabilities of the host machine. SSDs are highly recommended for local testing.

Iterative Tuning: Benchmarking is an iterative process. Observe, identify bottlenecks, tune relevant parameters (e.g., Kafka partitions, Spark resources, network settings), and re-test.

Cloud Implications: Benchmarking on a local environment provides valuable insights into architectural bottlenecks and scaling patterns, which are transferable to cloud environments. However, cloud environments (AWS MSK, EMR, Glue) offer significantly more scalable and elastic resources, requiring a separate, dedicated benchmarking phase once migrated.

Appendix F: Testing Framework Detail Expansion

This appendix provides a detailed elaboration on the sample testing approaches, complementing the general overview in Section 5.4.

Unit Tests:

Purpose: Verify the correctness of individual, isolated components or functions.

Application: FastAPI endpoint logic, PySpark transformation functions (e.g., specific UDFs, data cleansing functions), and any custom Python utilities.

Tools: pytest for Python code.

Sample Snippet (fastapi\_app/tests/unit/test\_api.py):

# fastapi\_app/tests/unit/test\_api.py  
import pytest  
from fastapi.testclient import TestClient  
# Assuming your FastAPI app is structured like app.main.app  
from fastapi\_app.app.main import app  
from datetime import datetime  
  
client = TestClient(app)  
  
def test\_read\_main():  
 response = client.get("/")  
 assert response.status\_code == 200  
 assert response.json() == {"message": "Welcome to Financial/Insurance Data Ingestor API!"}  
  
def test\_ingest\_financial\_transaction\_invalid\_data():  
 response = client.post("/ingest-financial-transaction/", json={  
 "transaction\_id": "FT-001",  
 "timestamp": "invalid-date", # Invalid timestamp  
 "account\_id": "ACC-XYZ",  
 "amount": "not-a-number", # Invalid amount  
 "currency": "USD",  
 "transaction\_type": "debit"  
 })  
 assert response.status\_code == 422 # Unprocessable Entity due to validation error  
 assert "validation error" in response.text

Integration Tests:

Purpose: Verify that different components of the pipeline work together as expected.

Application: FastAPI to Kafka, Kafka to Spark (Streaming), Spark transformations.

Tools: docker-compose.test.yml, pytest, Testcontainers (for robust service orchestration in tests), Kafka client libraries, MinIO SDK.

Conceptual docker-compose.test.yml for Integration Tests:

This file defines a stripped-down set of services specifically for integration testing, focusing on inter-service communication.

# docker-compose.test.yml (for integration testing)  
version: '3.8'  
services:  
 zookeeper:  
 image: confluentinc/cp-zookeeper:7.4.0  
 environment:  
 ZOOKEEPER\_CLIENT\_PORT: 2181  
 healthcheck:  
 test: ["CMD", "sh", "-c", "nc -z localhost 2181"]  
 interval: 10s  
 timeout: 5s  
 retries: 5  
  
 kafka:  
 image: confluentinc/cp-kafka:7.4.0  
 depends\_on:  
 zookeeper:  
 condition: service\_healthy  
 ports:  
 - "9092:9092"  
 environment:  
 KAFKA\_BROKER\_ID: 1  
 KAFKA\_ZOOKEEPER\_CONNECT: 'zookeeper:2181'  
 KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT\_HOST://localhost:9092  
 KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT\_HOST:PLAINTEXT  
 KAFKA\_INTER\_BROKER\_LISTENER\_NAME: PLAINTEXT  
 KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1  
 healthcheck:  
 test: ["CMD", "sh", "-c", "kafka-topics --bootstrap-server localhost:9092 --list"]  
 interval: 10s  
 timeout: 5s  
 retries: 5  
  
 minio:  
 image: minio/minio:latest  
 ports:  
 - "9000:9000"  
 environment:  
 MINIO\_ROOT\_USER: test\_user  
 MINIO\_ROOT\_PASSWORD: test\_password  
 command: server /data --console-address ":9000"  
 healthcheck:  
 test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]  
 interval: 30s  
 timeout: 20s  
 retries: 3  
  
 fastapi\_ingestor:  
 build: ./fastapi\_app  
 environment:  
 KAFKA\_BROKER: kafka:29092  
 KAFKA\_TOPIC: raw\_data\_test  
 depends\_on:  
 kafka:  
 condition: service\_healthy  
 healthcheck:  
 test: ["CMD", "curl", "-f", "http://localhost:8000/health || exit 1"]  
 interval: 5s  
 timeout: 3s  
 retries: 5  
  
 # Spark service for integration testing (can be a standalone driver in test, or a small cluster)  
 spark-test-runner:  
 image: bitnami/spark:3.5.0  
 depends\_on:  
 kafka:  
 condition: service\_healthy  
 minio:  
 condition: service\_healthy  
 environment:  
 SPARK\_MASTER\_URL: "local[\*]" # Run Spark in local mode for test  
 KAFKA\_BROKER: kafka:29092  
 MINIO\_HOST: minio  
 MINIO\_ACCESS\_KEY: test\_user  
 MINIO\_SECRET\_KEY: test\_password  
 volumes:  
 - ./pyspark\_jobs:/opt/bitnami/spark/data/pyspark\_jobs # Mount jobs  
 - ./data/test\_spark\_output:/tmp/spark\_output # Output dir for tests  
 # No exposed ports unless needed for Spark UI inspection during debug  
 command: ["tail", "-f", "/dev/null"] # Keep container running

Conceptual Integration Test (fastapi\_app/tests/integration/test\_data\_flow.py):

# fastapi\_app/tests/integration/test\_data\_flow.py  
import pytest  
import requests  
import subprocess  
import time  
from kafka import KafkaConsumer  
import json  
import os  
from datetime import datetime  
from minio import Minio # Assuming minio client library is installed  
  
# Define the path to your test compose file  
COMPOSE\_FILE = os.path.join(os.path.dirname(\_\_file\_\_), '../../docker-compose.test.yml')  
  
@pytest.fixture(scope="module")  
def docker\_services(request):  
 """Starts and stops docker-compose services for integration tests."""  
 print(f"\nStarting Docker services from: {COMPOSE\_FILE}")  
 # Ensure services are down first  
 subprocess.run(["docker", "compose", "-f", COMPOSE\_FILE, "down", "-v"], check=True)  
 subprocess.run(["docker", "compose", "-f", COMPOSE\_FILE, "up", "--build", "-d"], check=True)  
  
 # Wait for FastAPI to be healthy  
 api\_url = "http://localhost:8000"  
 for \_ in range(30): # Wait up to 30 seconds  
 try:  
 response = requests.get(f"{api\_url}/health")  
 if response.status\_code == 200:  
 print("FastAPI is healthy.")  
 break  
 except requests.exceptions.ConnectionError:  
 pass  
 time.sleep(1)  
 else:  
 pytest.fail("FastAPI did not become healthy in time.")  
  
 # Wait for Kafka to be healthy  
 kafka\_broker = "localhost:9092"  
 print(f"Waiting for Kafka at {kafka\_broker}...")  
 # More robust check could involve kafka-topics --list or similar  
 time.sleep(10) # Give Kafka some time to initialize  
  
 # Wait for MinIO to be healthy and create test bucket  
 minio\_client = Minio("localhost:9000", access\_key="test\_user", secret\_key="test\_password", secure=False)  
 bucket\_name = "raw-data-bucket-test"  
 if not minio\_client.bucket\_exists(bucket\_name):  
 minio\_client.make\_bucket(bucket\_name)  
 print(f"MinIO healthy and bucket '{bucket\_name}' ready.")  
  
 yield # Tests run here  
  
 print("Stopping Docker services.")  
 subprocess.run(["docker", "compose", "-f", COMPOSE\_FILE, "down", "-v"], check=True)  
  
def test\_end\_to\_end\_financial\_transaction\_flow(docker\_services):  
 """Tests ingestion via FastAPI, consumption via Kafka, and processing to Delta Lake."""  
 api\_url = "http://localhost:8000"  
 kafka\_broker = "localhost:9092"  
 kafka\_topic = "raw\_data\_test" # As defined in docker-compose.test.yml  
 minio\_host = "localhost:9000"  
 minio\_access\_key = "test\_user"  
 minio\_secret\_key = "test\_password"  
 minio\_bucket = "raw-data-bucket-test"  
 spark\_output\_dir = "/tmp/spark\_output/financial\_data\_delta" # Matches volume in spark-test-runner  
  
 # 1. Send data via FastAPI  
 transaction\_data = {  
 "transaction\_id": "INT-001",  
 "timestamp": datetime.now().isoformat(),  
 "account\_id": "ACC-INT-001",  
 "amount": 123.45,  
 "currency": "USD",  
 "transaction\_type": "deposit"  
 }  
 response = requests.post(f"{api\_url}/ingest-financial-transaction/", json=transaction\_data)  
 assert response.status\_code == 200  
 assert response.json()["message"] == "Financial transaction ingested successfully"  
  
 # 2. Consume data from Kafka and verify (optional, for explicit check)  
 consumer = KafkaConsumer(  
 kafka\_topic,  
 bootstrap\_servers=[kafka\_broker],  
 auto\_offset\_reset='earliest',  
 enable\_auto\_commit=False,  
 group\_id='test-consumer-group',  
 value\_deserializer=lambda x: json.loads(x.decode('utf-8'))  
 )  
 consumed\_message = None  
 start\_time = time.time()  
 for msg in consumer:  
 consumed\_message = msg.value  
 print(f"Consumed: {consumed\_message}")  
 if consumed\_message.get("transaction\_id") == transaction\_data["transaction\_id"]:  
 break  
 if time.time() - start\_time > 10: # Timeout after 10 seconds  
 break  
 consumer.close()  
 assert consumed\_message is not None, "Did not consume message from Kafka"  
 assert consumed\_message["transaction\_id"] == transaction\_data["transaction\_id"]  
  
 # 3. Trigger Spark job to process from Kafka to Delta Lake  
 # Create a simplified Spark job script for testing that reads from Kafka  
 # and writes to Delta Lake in MinIO.  
 # Example: pyspark\_jobs/streaming\_consumer\_test.py  
 # This script needs to be mounted into spark-test-runner  
 # For this test, we'll assume a simple job that writes raw Kafka messages to Delta Lake.  
 spark\_submit\_command = [  
 "docker", "exec", "spark-test-runner", "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=test\_user",  
 "--conf", "spark.hadoop.fs.s3a.secret.key=test\_password",  
 "--conf", "spark.hadoop.fs.s3a.path.style.access=true",  
 "pyspark\_jobs/streaming\_consumer\_test.py", # This script will read from Kafka and write to MinIO  
 kafka\_topic,  
 "kafka:29092", # Kafka broker for Spark  
 f"s3a://{minio\_bucket}/{spark\_output\_dir.replace('/tmp/spark\_output/', '')}" # S3a path  
 ]  
 print(f"Running Spark job: {' '.join(spark\_submit\_command)}")  
 spark\_process = subprocess.run(spark\_submit\_command, capture\_output=True, text=True, check=True)  
 print(spark\_process.stdout)  
 print(spark\_process.stderr)  
 time.sleep(15) # Give Spark time to consume and write  
  
 # 4. Verify data in Delta Lake (MinIO)  
 minio\_client = Minio(minio\_host, access\_key=minio\_access\_key, secret\_key=minio\_secret\_key, secure=False)  
 # List objects in the Delta Lake path to confirm data written  
 found\_delta\_files = False  
 for obj in minio\_client.list\_objects(minio\_bucket, prefix=f"{spark\_output\_dir.replace('/tmp/spark\_output/', '')}/", recursive=True):  
 if "\_delta\_log" in obj.object\_name or ".parquet" in obj.object\_name:  
 found\_delta\_files = True  
 break  
 assert found\_delta\_files, "No Delta Lake files found in MinIO after Spark job execution."  
 # Optional: Read data back from Delta Lake using a local SparkSession (if `pyspark` is installed locally)  
 # from pyspark.sql import SparkSession  
 # spark\_read = (SparkSession.builder.appName("DeltaReadTest")  
 # .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 # .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 # .config("spark.hadoop.fs.s3a.endpoint", f"http://{minio\_host}")  
 # .config("spark.hadoop.fs.s3a.access.key", minio\_access\_key)  
 # .config("spark.hadoop.fs.s3a.secret.key", minio\_secret\_key)  
 # .config("spark.hadoop.fs.s3a.path.style.access", "true")  
 # .getOrCreate())  
 #  
 # delta\_df = spark\_read.read.format("delta").load(f"s3a://{minio\_bucket}/{spark\_output\_dir.replace('/tmp/spark\_output/', '')}")  
 # delta\_df.show()  
 # assert delta\_df.count() >= 1 # At least one row should be there  
 # assert delta\_df.filter(delta\_df.value.contains(transaction\_data["transaction\_id"])).count() == 1  
 # spark\_read.stop()

Note for streaming\_consumer\_test.py:

You'd need a simple PySpark script like this in pyspark\_jobs/:

# pyspark\_jobs/streaming\_consumer\_test.py  
import sys  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import col, from\_json  
from pyspark.sql.types import StructType, StringType, FloatType, TimestampType, MapType  
  
def create\_spark\_session(app\_name):  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10\_2.12:3.5.0,io.delta:delta-core\_2.12:2.4.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if len(sys.argv) != 4:  
 print("Usage: streaming\_consumer\_test.py <kafka\_topic> <kafka\_broker> <delta\_output\_path>")  
 sys.exit(-1)  
  
 kafka\_topic = sys.argv[1]  
 kafka\_broker = sys.argv[2]  
 delta\_output\_path = sys.argv[3]  
  
 spark = create\_spark\_session("KafkaToDeltaTest")  
  
 # Define schema for the incoming Kafka message value (adjust as per your FastAPI data)  
 schema = StructType() \  
 .add("transaction\_id", StringType()) \  
 .add("timestamp", StringType()) \  
 .add("account\_id", StringType()) \  
 .add("amount", FloatType()) \  
 .add("currency", StringType()) \  
 .add("transaction\_type", StringType()) \  
 .add("merchant\_id", StringType(), True) \  
 .add("category", StringType(), True)  
  
 # Read from Kafka  
 kafka\_df = (spark.readStream  
 .format("kafka")  
 .option("kafka.bootstrap.servers", kafka\_broker)  
 .option("subscribe", kafka\_topic)  
 .option("startingOffsets", "earliest")  
 .load())  
  
 # Parse the value column from Kafka  
 parsed\_df = kafka\_df.selectExpr("CAST(value AS STRING) as json\_value") \  
 .select(from\_json(col("json\_value"), schema).alias("data")) \  
 .select("data.\*")  
  
 # Write to Delta Lake  
 query = (parsed\_df.writeStream  
 .format("delta")  
 .outputMode("append")  
 .option("checkpointLocation", f"{delta\_output\_path}/\_checkpoints")  
 .start(delta\_output\_path))  
  
 query.awaitTermination(30) # Run for 30 seconds to capture test data  
 query.stop()  
 spark.stop()

Data Quality Tests:

Purpose: Ensure accuracy, completeness, consistency, validity, and timeliness of data.

Application: Integrate data quality checks within Spark jobs or as separate validation steps.

Tools: Great Expectations, Pydantic (for schema validation), custom validation logic.

Conceptual Pact Contract Testing Snippet:

Pact is a "consumer-driven contract" testing tool. This would typically be a separate test suite (pyspark\_jobs/tests/contract/financial\_transaction\_consumer\_pact.py).

# pyspark\_jobs/tests/contract/financial\_transaction\_consumer\_pact.py  
import pytest  
from pact import Consumer, Provider  
from pyspark.sql import SparkSession  
from pyspark.sql.types import StructType, StringType, FloatType, TimestampType  
import json  
from datetime import datetime  
from pyspark.sql.functions import current\_timestamp  
  
# Define Pact mock server details  
PACT\_MOCK\_HOST = 'localhost'  
PACT\_MOCK\_PORT = 1234  
PACT\_DIR = './pacts' # Directory where pact files will be written  
  
# Define the consumer and provider for this contract  
consumer = Consumer('FinancialTransactionSparkConsumer')  
provider = Provider('FastAPIIngestor')  
  
@pytest.fixture(scope='module')  
def pact\_spark\_session():  
 """Fixture for a local SparkSession to be used in contract tests."""  
 spark = (SparkSession.builder  
 .appName("PactSparkConsumer")  
 .master("local[\*]")  
 .getOrCreate())  
 yield spark  
 spark.stop()  
  
@pytest.fixture(scope='module')  
def pact():  
 """Starts and stops the Pact mock service."""  
 pact\_instance = consumer.has\_pact\_with(  
 provider,  
 host\_name=PACT\_MOCK\_HOST,  
 port=PACT\_MOCK\_PORT,  
 pact\_dir=PACT\_DIR  
 )  
 print(f"\nStarting Pact mock service on {PACT\_MOCK\_HOST}:{PACT\_MOCK\_PORT}")  
 pact\_instance.start\_service()  
 yield pact\_instance  
 print("Stopping Pact mock service")  
 pact\_instance.stop\_service()  
  
def test\_spark\_can\_process\_financial\_transaction\_from\_kafka(pact, pact\_spark\_session):  
 """  
 Verifies that the Spark consumer can correctly process a financial transaction  
 message from Kafka, based on the contract with the FastAPI Ingestor.  
 """  
 # Define the expected message structure from the producer (FastAPI)  
 expected\_message\_body = {  
 "transaction\_id": "TRANS-12345",  
 "timestamp": "2023-10-26T14:30:00.000Z",  
 "account\_id": "ACC-FIN-001",  
 "amount": 500.75,  
 "currency": "USD",  
 "transaction\_type": "credit",  
 "merchant\_id": "MER-ABC",  
 "category": "utilities"  
 }  
  
 # Define the interaction for the Kafka message  
 (pact  
 .given('a financial transaction is published to Kafka')  
 .upon\_receiving('a Kafka message with financial transaction data')  
 .with\_message(  
 'application/json', # Mime type of the message  
 json.dumps(expected\_message\_body) # The expected message content  
 ))  
  
 with pact:  
 # Simulate receiving the message as if from Kafka  
 # In a real Spark job, this would be the actual Kafka consumer logic  
 # For a contract test, we feed the expected message directly to the Spark logic  
 # Convert the expected message body to a Spark DataFrame  
 schema = StructType() \  
 .add("transaction\_id", StringType()) \  
 .add("timestamp", StringType()) \  
 .add("account\_id", StringType()) \  
 .add("amount", FloatType()) \  
 .add("currency", StringType()) \  
 .add("transaction\_type", StringType()) \  
 .add("merchant\_id", StringType(), True) \  
 .add("category", StringType(), True)  
  
 # Create a DataFrame from the single expected message  
 df\_from\_kafka = pact\_spark\_session.createDataFrame([expected\_message\_body], schema=schema)  
  
 # Apply a dummy transformation that resembles your actual Spark job logic  
 # This ensures your Spark code can parse and work with the contract-defined schema  
 processed\_df = df\_from\_kafka.withColumn("processed\_at", current\_timestamp())  
  
 # Collect and assert the processed data  
 collected\_data = processed\_df.collect()  
 assert len(collected\_data) == 1  
 assert collected\_data[0]['transaction\_id'] == expected\_message\_body['transaction\_id']  
 assert collected\_data[0]['amount'] == expected\_message\_body['amount']  
 assert 'processed\_at' in collected\_data[0]

Performance and Load Testing:

Purpose: Assess the system's performance under expected and peak load conditions, identify bottlenecks, and ensure it meets non-functional requirements (e.g., latency, throughput).

Application: Use tools to simulate high volumes of data being sent to the FastAPI endpoint and monitor Kafka, Spark, and database performance using Grafana dashboards.

Tools: Locust (for API load testing), JMeter, Spark UI, Grafana.

Appendix H: Quantitative Benchmarking Harness Details

This appendix provides a detailed elaboration on the sample benchmarking harness and observed data mentioned in Section 7.4 of the main document. It outlines how performance benchmarks are conducted and analyzed to ensure the data platform meets its non-functional requirements for throughput and latency.

To truly understand performance, theoretical sizing must be combined with empirical measurements. This section provides a conceptual benchmarking harness and illustrative observed data, emphasizing the components and steps involved in comprehensive load testing.

Benchmarking Harness Components:

The benchmarking harness is designed to simulate realistic workloads and collect comprehensive metrics across the entire data pipeline. It comprises the following key components:

Load Generator (Locust):

Role: Simulates concurrent users sending a high volume of financial and insurance data to the FastAPI ingestion API. This is crucial for mimicking real-world data producers and generating peak load conditions.

Configuration: Configured to vary the number of concurrent users and requests per second (RPS) to test different load levels.

FastAPI Ingestor:

Role: The entry point for all incoming data. It receives data from the load generator, performs initial validation (via Pydantic models), and publishes the messages to the designated Kafka topics.

Monitoring Focus: Key metrics include request per second (RPS), end-to-end API latency (average and P99), and error rates.

Kafka Cluster:

Role: Acts as a distributed, fault-tolerant message buffer. It receives and stores the high-volume data streams published by the FastAPI ingestor.

Monitoring Focus: Key metrics include producer throughput (messages/sec, MB/sec), consumer throughput (messages/sec), and critically, Kafka consumer lag (number of messages remaining in the backlog for the Spark consumer).

Spark Structured Streaming Job:

Role: Consumes data from the raw Kafka topics, performs essential transformations (e.g., parsing, schema enforcement, data cleansing, and basic aggregations), and writes the processed data to the Raw Delta Lake zone in MinIO.

Monitoring Focus: Metrics include batch processing time, records processed per batch, micro-batch latency, and resource utilization (CPU, memory) of Spark executors.

Metrics Collector (Grafana Alloy):

Role: Collects telemetry data (metrics, logs, traces) from all instrumented components within the Docker Compose environment. It acts as a central collection agent for observability data.

Integration: Configured to receive OpenTelemetry Protocol (OTLP) data from FastAPI and other services, and to scrape Prometheus-compatible metrics (e.g., from cAdvisor, Kafka JMX exporters).

Monitoring (Grafana):

Role: Provides interactive data visualization and monitoring dashboards. It connects to Grafana Alloy (or directly to Prometheus/Loki configured by Alloy) to visualize real-time and historical performance metrics.

Dashboards: Pre-built dashboards show end-to-end latency, throughput for each pipeline stage, resource utilization (CPU, memory, network I/O) for all Docker containers (via cAdvisor), and Kafka consumer lag trends.

Conceptual Benchmarking Steps:

A systematic approach to benchmarking ensures reliable and reproducible results:

Setup Environment: Bring up the full Advanced Track Docker Compose environment (docker compose -f docker-compose.yml up --build -d). Ensure all services are healthy and stable before starting tests.

Establish Baseline: Run the system under a typical, low-load condition. Record baseline performance metrics (latency, throughput, resource usage) to understand normal operating characteristics.

Run Load Generator: Start the Locust load generator, configuring it to simulate a specific number of concurrent users and a target request rate to the FastAPI endpoint.

Example command: locust -f locust\_fastapi\_ingestor.py --host http://localhost:8000 (then access Locust UI at http://localhost:8089).

Monitor Metrics in Real-time: Continuously observe the Grafana dashboards during the load test. Pay close attention to:

FastAPI: Request rate (RPS), average and P99 latency for API calls, and any error spikes.

Kafka: Producer throughput (ensuring data is flowing into Kafka as expected), consumer throughput (ensuring Spark is keeping up), and especially Kafka consumer lag (any increasing lag indicates a bottleneck downstream).

Spark: Batch processing times (for streaming jobs), number of records processed per second, CPU and memory utilization of Spark master and worker nodes (available via Spark UI or Grafana).

Overall System: Container resource utilization (CPU, memory, network I/O) across all services using cAdvisor metrics in Grafana.

Analyze Data: After the load test, analyze the recorded metrics.

Identify the bottleneck: Is it the API, Kafka, Spark, or the underlying storage (MinIO)?

Evaluate latency and throughput against defined SLOs.

Look for correlation between increased load, resource saturation, and performance degradation.

Scale Up/Down and Tune: Repeat tests by systematically varying parameters:

Kafka: Increase/decrease the number of partitions for topics.

Spark: Adjust Spark executor counts, cores per executor, and memory allocated per executor in docker-compose.yml. Experiment with Spark configurations like spark.sql.shuffle.partitions.

FastAPI: If FastAPI becomes a bottleneck, consider increasing the number of FastAPI replicas or optimizing its code.

Databases (PostgreSQL/MongoDB): For intensive workloads, monitor database specific metrics (e.g., connection pool size, query latency, disk I/O) and consider tuning database configurations or scaling resources.

This iterative process of testing, monitoring, analyzing, and tuning is essential to identify the optimal configuration for different load levels and to ensure the platform scales effectively.

Conceptual Locust Load Test Script (locust\_fastapi\_ingestor.py):

This script simulates two types of data ingestion: financial transactions and insurance claims.

# locust\_fastapi\_ingestor.py  
"""  
Locust load test script for the FastAPI Data Ingestor.  
This script defines two tasks to simulate traffic:  
1. ingest\_financial\_transaction: Sends mock financial transaction data.  
2. ingest\_insurance\_claim: Sends mock insurance claim data.  
  
The user can configure the host, number of users, and spawn rate via the Locust UI  
(usually http://localhost:8089 after running `locust -f locust\_fastapi\_ingestor.py`).  
"""  
from locust import HttpUser, task, between  
import json  
from datetime import datetime, timedelta  
import random  
  
class FinancialDataUser(HttpUser):  
 """  
 User class that simulates sending financial and insurance data to the FastAPI ingestor.  
 """  
 # Wait time between requests for each simulated user.  
 # This helps simulate more realistic user behavior rather than hammering the API constantly.  
 wait\_time = between(0.1, 0.5) # Simulate delay between requests (0.1 to 0.5 seconds)  
  
 # The host URL for the FastAPI application. This should match the exposed port in docker-compose.  
 # In a local Docker Compose setup, FastAPI is often exposed on localhost:8000.  
 host = "http://localhost:8000" # Target FastAPI endpoint  
  
 @task(1) # This task has a weight of 1, meaning it will be executed proportionally to other tasks.  
 def ingest\_financial\_transaction(self):  
 """  
 Simulates sending a financial transaction POST request to the FastAPI ingestor.  
 Generates realistic-looking mock data for a financial transaction.  
 """  
 transaction\_data = {  
 "transaction\_id": f"FT-{datetime.now().strftime('%Y%m%d%H%M%S%f')}-{random.randint(1000, 9999)}",  
 "timestamp": datetime.now().isoformat(),  
 "account\_id": f"ACC-{random.randint(100000, 999999)}",  
 "amount": round(random.uniform(1.0, 10000.0), 2), # Random amount between 1.00 and 10000.00  
 "currency": random.choice(["USD", "EUR", "GBP", "JPY"]), # Random currency  
 "transaction\_type": random.choice(["debit", "credit", "transfer", "payment"]), # Random type  
 "merchant\_id": f"MER-{random.randint(100, 999)}" if random.random() > 0.3 else None, # Optional merchant ID  
 "category": random.choice(["groceries", "utilities", "salary", "entertainment", "transport", "housing", "healthcare", "education"])  
 }  
 # Send the POST request. The 'name' argument groups requests in Locust's statistics.  
 self.client.post("/ingest-financial-transaction/", json=transaction\_data, name="/ingest-financial-transaction")  
  
 @task(1) # This task also has a weight of 1.  
 def ingest\_insurance\_claim(self):  
 """  
 Simulates sending an insurance claim POST request to the FastAPI ingestor.  
 Generates realistic-looking mock data for an insurance claim.  
 """  
 claim\_data = {  
 "claim\_id": f"IC-{datetime.now().strftime('%Y%m%d%H%M%S%f')}-{random.randint(1000, 9999)}",  
 "timestamp": datetime.now().isoformat(),  
 "policy\_number": f"POL-{random.randint(1000000, 9999999)}",  
 "claim\_amount": round(random.uniform(500.0, 50000.0), 2), # Random amount  
 "claim\_type": random.choice(["auto", "health", "home", "life", "property"]), # Random claim type  
 "claim\_status": random.choice(["submitted", "under\_review", "approved", "rejected", "paid"]), # Random status  
 "customer\_id": f"CUST-{random.randint(10000, 99999)}",  
 "incident\_date": (datetime.now() - timedelta(days=random.randint(0, 365))).isoformat() # Incident date within last year  
 }  
 # Send the POST request.  
 self.client.post("/ingest-insurance-claim/", json=claim\_data, name="/ingest-insurance-claim")

Observed Throughput and Latency (Illustrative for Local Dev Environment):

These figures are conceptual and will vary significantly based on your machine's hardware, other running processes, and exact configuration. They serve as a guide for what to measure and expect. Real-world results will necessitate profiling against your specific hardware and workloads.

Key Takeaways from Benchmarking:

Initial Bottleneck Identification: Often, the FastAPI instance itself or the underlying network I/O on the host machine can become the initial bottleneck if not optimized or scaled adequately.

Scaling Kafka: Increasing the number of Kafka partitions (and ensuring a corresponding increase in Kafka consumer parallelism) is a primary way to scale Kafka's throughput.

Scaling Spark: Adding more Spark executors and allocating more cores and memory per executor directly leads to higher data processing throughput. However, this also increases resource consumption and can quickly saturate a local development machine.

Disk I/O Impact: The performance of MinIO (simulating S3) and the Delta Lake operations are heavily influenced by the underlying disk speed and I/O capabilities of the host machine. SSDs are highly recommended for local testing.

Iterative Tuning: Benchmarking is an iterative process. Observe, identify bottlenecks, tune relevant parameters (e.g., Kafka partitions, Spark resources, network settings), and re-test.

Cloud Implications: Benchmarking on a local environment provides valuable insights into architectural bottlenecks and scaling patterns, which are transferable to cloud environments. However, cloud environments (AWS MSK, EMR, Glue) offer significantly more scalable and elastic resources, requiring a separate, dedicated benchmarking phase once migrated.

Highlighting Delta Lake: The Data Lakehouse Storage Layer

Delta Lake is an open-source storage layer that brings ACID (Atomicity, Consistency, Isolation, Durability) transactions, schema enforcement, and time travel capabilities to data lakes. It unifies batch and streaming data processing, transforming your MinIO (S3-compatible) storage into a reliable "data lakehouse."

This guide will demonstrate basic and advanced use cases of Delta Lake, leveraging your Advanced Track local environment setup and focusing on its integration with Apache Spark.

Reference: This guide builds upon the concepts and setup described in Section 4.2. Core Technology Deep Dive of the Core Handbook, the Progressive Path Setup Guide Deep-Dive Addendum, and the Spark examples in the "Highlighting Apache Spark" document.

Basic Use Case: Reliable Data Appends to the Data Lakehouse

Objective: To demonstrate how Spark Structured Streaming reliably appends data to a Delta Lake table in MinIO, ensuring transactionality and consistency even in streaming scenarios.

Role in Platform: Provide a solid, transactional foundation for raw and curated data storage, enabling consistent reads for downstream consumers.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root.

Verify Spark and MinIO are accessible: Check their respective container logs and UIs (http://localhost:18080 for Spark History, http://localhost:9001 for MinIO).

Ensure simulate\_data.py is running: Data should be continuously sent to FastAPI and then published to Kafka topics (raw\_financial\_transactions, raw\_insurance\_claims).

Spark Streaming Jobs are running: The pyspark\_jobs/streaming\_consumer.py jobs (from the "Highlighting Apache Spark" document's Basic Use Case) should be actively consuming from Kafka and writing to s3a://raw-data-bucket/financial\_data\_delta and s3a://raw-data-bucket/insurance\_data\_delta.

Steps to Exercise:

Observe Writes: Let the streaming\_consumer.py Spark jobs run for a few minutes.

Inspect MinIO: Access the MinIO Console (http://localhost:9001).

Navigate to raw-data-bucket.

Enter financial\_data\_delta/ (or insurance\_data\_delta/).

You will see a \_delta\_log directory and numerous .parquet files. The presence of \_delta\_log confirms it's a Delta Lake table. The .parquet files are the actual data appended in micro-batches by Spark.

Query Data (via Spark-SQL): You can query the Delta table using Spark's SQL capabilities from within the spark container to verify data content.  
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 COUNT(\*) FROM delta.\`s3a://raw-data-bucket/financial\_data\_delta\`;"  
  
This query will show the growing number of records in the Delta table, demonstrating continuous, transactional appends.

Verification:

MinIO Contents: The \_delta\_log directory and increasing number of .parquet files in financial\_data\_delta and insurance\_data\_delta paths.

Spark-SQL Query: The COUNT(\*) query reflects a continuously growing number, indicating that new data is being reliably appended.

Spark History Server: The streaming jobs remain active and show consistent write operations to Delta Lake.

Advanced Use Case 1: Schema Enforcement and Evolution

Objective: To demonstrate how Delta Lake enforces schema by default, preventing bad data from corrupting tables, and how it allows controlled schema evolution to adapt to changing data structures.

Role in Platform: Maintain data quality and flexibility in data lake schemas, preventing silent data corruption.

Setup/Configuration:

Ensure Basic Use Case is running: financial\_data\_delta is populated.

pyspark\_jobs/streaming\_consumer.py: Ensure this script is using the data\_schema as defined, which acts as the target schema for the Delta table.

Steps to Exercise:

Simulate Schema Violation (Expected to Fail without mergeSchema):

Temporarily modify simulate\_data.py to send a single batch of financial transactions with a data type mismatch for an existing column (e.g., send amount as a string instead of a float).

Original (in simulate\_data.py): "amount": round(random.uniform(1.0, 10000.0), 2),

Temporary change (for a few seconds): "amount": "FIVE HUNDRED",

Let simulate\_data.py run for 5-10 seconds with this change, then immediately revert simulate\_data.py back to its original (correct float) format.

Observe the logs of your financial\_transactions Spark streaming job.

Verify Schema Enforcement:

Expected Behavior (without mergeSchema / overwriteSchema): The Spark streaming job will likely encounter an error (e.g., AnalysisException: Cannot write unknown type string into float type column ... or similar type mismatch error). The job might stop or continuously restart, demonstrating Delta Lake's strict schema enforcement.

Action: If the job failed, restart it with the correct data (simulate\_data.py reverted).

Demonstrate Schema Evolution (.option("mergeSchema", "true")):

Modify simulate\_data.py to add a new optional column to the financial transaction data (e.g., is\_flagged: bool = False).

Add to transaction\_data in simulate\_data.py: "is\_flagged": random.choice([True, False]),

Update pyspark\_jobs/streaming\_consumer.py: Add this new column to the data\_schema definition, and ensure the .writeStream call includes .option("mergeSchema", "true").  
# In data\_schema definition:  
.add("is\_flagged", BooleanType(), True) # Add this line  
  
# In writeStream options:  
.option("mergeSchema", "true") # Ensure this option is present

Restart both simulate\_data.py and the financial\_transactions Spark streaming job.

Let them run for a few minutes.

Verify Schema Evolution:

Query 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 "DESCRIBE HISTORY delta.\`s3a://raw-data-bucket/financial\_data\_delta\`;"  
  
Look for a new version (version) that includes the schema change.  
Then, query the data:  
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 transaction\_id, amount, is\_flagged FROM delta.\`s3a://raw-data-bucket/financial\_data\_delta\` LIMIT 20;"

Verification:

Schema Enforcement (Failure): When sending invalid data without mergeSchema, the Spark job should log errors and potentially fail, demonstrating that Delta Lake prevents malformed data from being written.

Schema Evolution (Success): After adding the new column and using mergeSchema=true, the DESCRIBE HISTORY command will show a new table version with the updated schema. Queries will show the is\_flagged column, with null values for older records and True/False for newly ingested ones. This highlights controlled schema evolution.

Advanced Use Case 2: Time Travel (Data Versioning)

Objective: To demonstrate Delta Lake's "time travel" capability, allowing you to query historical versions of your data, crucial for auditing, reproducing past reports, and recovering from accidental data modifications.

Role in Platform: Provide data versioning for compliance, reproducibility, and disaster recovery.

Setup/Configuration:

Ensure Basic Use Case is running: financial\_data\_delta is being continuously written to.

Note a timestamp/version: Let data stream for some time. Note the timestamp or version from DESCRIBE HISTORY.

Steps to Exercise:

Get Current Table State (Baseline):  
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 COUNT(\*) FROM delta.\`s3a://raw-data-bucket/financial\_data\_delta\`;"  
  
Note the count.

Simulate an "Accidental" Overwrite/Deletion:

Stop the financial\_transactions Spark streaming job.

Run a temporary Spark batch job to OVERWRITE the entire financial\_data\_delta table with a very small subset of data, or even empty data, simulating data loss.  
Example pyspark\_jobs/temp\_overwrite.py (create this file):  
from pyspark.sql import SparkSession  
from delta.tables import DeltaTable  
import sys  
  
def create\_spark\_session(app\_name):  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "io.delta:delta-core\_2.12:2.4.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if len(sys.argv) != 2:  
 print("Usage: temp\_overwrite.py <delta\_table\_path>")  
 sys.exit(-1)  
  
 delta\_table\_path = sys.argv[1]  
 spark = create\_spark\_session("TempOverwrite")  
  
 # Create a very small DataFrame to overwrite the existing table  
 small\_data = [("OVERWRITE-001", "2024-01-01T10:00:00.000Z", "ACC-OVERWRITE", 1.0, "USD", "payment", "MER-OVW", "misc")]  
 columns = ["transaction\_id", "timestamp", "account\_id", "amount", "currency", "transaction\_type", "merchant\_id", "category"]  
 small\_df = spark.createDataFrame(small\_data, schema=columns) # Use the full schema for compatibility  
  
 print(f"Overwriting {delta\_table\_path} with small data...")  
 small\_df.write.format("delta").mode("overwrite").option("overwriteSchema", "true").save(delta\_table\_path)  
 print("Overwrite complete.")  
 spark.stop()

Run this job:  
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/temp\_overwrite.py \  
 s3a://raw-data-bucket/financial\_data\_delta

Now, query the table normally and note the COUNT(\*) again. It should be drastically reduced (e.g., 1 record).

Query History:  
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 "DESCRIBE HISTORY delta.\`s3a://raw-data-bucket/financial\_data\_delta\`;"  
  
This will show a list of all operations, including your initial appends and the recent OVERWRITE. Note the version (e.g., 0, 1, 2, etc.) and timestamp of a version before your overwrite.

Time Travel Query (by Version):  
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 COUNT(\*) FROM delta.\`s3a://raw-data-bucket/financial\_data\_delta\` VERSION AS OF <PREVIOUS\_VERSION\_NUMBER>;"  
  
Replace <PREVIOUS\_VERSION\_NUMBER> with the version before your overwrite. The count should revert to the original large number.

Time Travel Query (by Timestamp):  
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 COUNT(\*) FROM delta.\`s3a://raw-data-bucket/financial\_data\_delta\` TIMESTAMP AS OF 'YYYY-MM-DD HH:MM:SS';"  
  
Replace 'YYYY-MM-DD HH:MM:SS' with a timestamp before your overwrite.

Rollback (Conceptual): While not easily demonstrated via simple spark-sql in this context, in a real scenario, you could use RESTORE TABLE command or read a historical version and write it back to the current version.

Verification:

The current table (SELECT COUNT(\*) FROM delta.\...`) shows the limited data after the overwrite.

Time travel queries (VERSION AS OF or TIMESTAMP AS OF) successfully retrieve the larger dataset from before the overwrite, demonstrating that the historical data is still available.

DESCRIBE HISTORY clearly shows the sequence of operations and versions.

Advanced Use Case 3: Upserts (MERGE INTO) and Change Data Capture (CDC)

Objective: To demonstrate how Delta Lake supports efficient upsert operations (inserting new records and updating existing ones) and how its transaction log can be used for Change Data Capture (CDC). This is crucial for building slowly changing dimensions and replicating data efficiently.

Role in Platform: Support data warehousing patterns, facilitate efficient data synchronization, and enable real-time data replication.

Setup/Configuration:

Populate a "Target" Delta Table: Create a new Delta table that will serve as our target for upserts (e.g., s3a://curated-data-bucket/financial\_transactions\_dim). For simplicity, start it with some initial data.

Source for Updates: Use a simple Spark DataFrame or simulate a small Kafka topic with update/insert records.

Spark Merge Script: Create pyspark\_jobs/delta\_merge\_cdc.py.  
Example pyspark\_jobs/delta\_merge\_cdc.py (conceptual):  
# pyspark\_jobs/delta\_merge\_cdc.py  
import sys  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import col, lit, current\_timestamp  
from delta.tables import DeltaTable # Import DeltaTable class  
  
def create\_spark\_session(app\_name):  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "io.delta:delta-core\_2.12:2.4.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if len(sys.argv) != 2:  
 print("Usage: delta\_merge\_cdc.py <target\_delta\_table\_path>")  
 sys.exit(-1)  
  
 target\_path = sys.argv[1]  
 spark = create\_spark\_session("DeltaMergeAndCDC")  
 spark.sparkContext.setLogLevel("WARN")  
  
 # --- Part 1: Initial Population (if table doesn't exist) ---  
 initial\_data = [  
 ("T001", "2024-05-01", "A001", 100.0, "USD", "credit"),  
 ("T002", "2024-05-01", "A002", 200.0, "USD", "debit"),  
 ("T003", "2024-05-02", "A001", 50.0, "EUR", "payment")  
 ]  
 initial\_schema = ["transaction\_id", "date", "account\_id", "amount", "currency", "type"]  
 initial\_df = spark.createDataFrame(initial\_data, initial\_schema)  
  
 if not DeltaTable.isDeltaTable(spark, target\_path):  
 print(f"Creating initial Delta table at {target\_path}")  
 initial\_df.write.format("delta").mode("overwrite").save(target\_path)  
 else:  
 print(f"Delta table already exists at {target\_path}. Skipping initial population.")  
  
 # Load the target Delta table  
 deltaTable = DeltaTable.forPath(spark, target\_path)  
 print("Current data in target table:")  
 deltaTable.toDF().show()  
  
 # --- Part 2: Simulate incoming changes (new data + updates) ---  
 # T002: updated amount, currency  
 # T004: new transaction  
 # T005: new transaction  
 updates\_data = [  
 ("T002", "2024-05-01", "A002", 250.0, "GBP", "debit"), # Update existing T002  
 ("T004", "2024-05-03", "A003", 150.0, "USD", "transfer"), # New T004  
 ("T005", "2024-05-03", "A001", 75.0, "EUR", "credit") # New T005  
 ]  
 updates\_df = spark.createDataFrame(updates\_data, initial\_schema)  
 print("Incoming updates/inserts:")  
 updates\_df.show()  
  
 # --- Part 3: Perform MERGE INTO operation (Upsert) ---  
 print("Performing MERGE INTO operation...")  
 deltaTable.alias("target") \  
 .merge(  
 updates\_df.alias("source"),  
 "target.transaction\_id = source.transaction\_id" # Match condition  
 ) \  
 .whenMatchedUpdate(set = { # If matched, update columns  
 "date": "source.date",  
 "account\_id": "source.account\_id",  
 "amount": "source.amount",  
 "currency": "source.currency",  
 "type": "source.type",  
 "last\_updated": current\_timestamp() # Add an audit column  
 }) \  
 .whenNotMatchedInsert(values = { # If not matched, insert all columns from source  
 "transaction\_id": "source.transaction\_id",  
 "date": "source.date",  
 "account\_id": "source.account\_id",  
 "amount": "source.amount",  
 "currency": "source.currency",  
 "type": "source.type",  
 "last\_updated": current\_timestamp()  
 }) \  
 .execute()  
  
 print("Data after MERGE INTO:")  
 deltaTable.toDF().show()  
  
 # --- Part 4: Demonstrate Change Data Feed (CDC) ---  
 # Querying the change data feed requires it to be enabled on the table  
 # If not enabled during table creation, you can alter it:  
 # spark.sql(f"ALTER TABLE delta.`{target\_path}` SET TBLPROPERTIES (delta.enableChangeDataFeed = true)")  
 print("\nDemonstrating Change Data Feed (CDC) - Requires 'delta.enableChangeDataFeed = true' on table.")  
 print("Querying changes since last operation:")  
 try:  
 # Get changes from the last version (current version - 1)  
 # You would typically get the version number from `DESCRIBE HISTORY` or track it  
 current\_version = deltaTable.history().select("version").orderBy(col("version").desc()).first()["version"]  
 # To get changes from the last MERGE, we need to know its version  
 # For simplicity here, we'll query changes from version 0  
  
 # This is a bit tricky locally without persistent enablement  
 # Best way to ensure CDC is enabled:  
 # 1. Create a fresh table with cdc enabled:  
 # spark.sql(f"CREATE TABLE IF NOT EXISTS delta.`{target\_path}\_cdc\_enabled` USING DELTA LOCATION '{target\_path}\_cdc\_enabled' TBLPROPERTIES (delta.enableChangeDataFeed = true)")  
 # 2. Then merge into that table  
 # 3. Then query readChangeFeed  
  
 # For conceptual demo, assume CDC is enabled.  
 # You can set this explicitly if you create the table via DDL or initial write  
 # with .option("delta.enableChangeDataFeed", "true")  
 # Or run ALTER TABLE manually in spark-sql before running this script  
 # spark.sql(f"ALTER TABLE delta.`{target\_path}` SET TBLPROPERTIES (delta.enableChangeDataFeed = true)")  
  
 # Querying the change data feed from the version before the merge  
 # This will show 'insert' for new records and 'update\_preimage'/'update\_postimage' for updated ones  
 changes\_df = spark.read.format("delta") \  
 .option("readChangeFeed", "true") \  
 .option("startingVersion", str(current\_version)) \  
 .load(target\_path)  
  
 print(f"Changes from version {current\_version}:")  
 changes\_df.show(truncate=False)  
  
 except Exception as e:  
 print(f"Could not demonstrate CDC (Change Data Feed): {e}")  
 print("Ensure 'delta.enableChangeDataFeed = true' is set on the Delta table properties.")  
 print("You might need to run `ALTER TABLE delta.`<path>` SET TBLPROPERTIES (delta.enableChangeDataFeed = true)` manually in spark-sql, then run this script again after restarting Spark session.")  
  
 spark.stop()

Steps to Exercise:

Stop streaming jobs: Ensure no other jobs are writing to the target path.

Submit Merge/CDC Job: In a new terminal, submit the delta\_merge\_cdc.py job.  
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/delta\_merge\_cdc.py \  
 s3a://curated-data-bucket/financial\_transactions\_dim

Monitor: Observe the console output.

It will show the initial data, then the incoming updates, then the result after the MERGE INTO.

It will also attempt to show the CDC.

Verification:

Upsert Verification: After the job runs, query the financial\_transactions\_dim table (e.g., via spark-sql). You should see:

T002 updated with the new amount and currency (250.0, GBP).

New records T004 and T005 inserted.

CDC Verification (if successful): The console output for CDC will show a DataFrame containing records with \_change\_type columns (insert, update\_preimage, update\_postimage), reflecting exactly what changed in the table during the merge operation. This demonstrates the ability to capture changes for downstream systems.

Highlighting Grafana: Interactive Data Visualization and Monitoring

Grafana is an open-source platform for interactive data visualization and monitoring. In your data platform, it is the primary interface for gaining real-time operational insights, visualizing key performance indicators (KPIs), and setting up alerts based on the telemetry data collected by Grafana Alloy from all services. It transforms raw metrics into actionable dashboards, crucial for maintaining system health and reliability.

This guide will demonstrate basic and advanced use cases of Grafana, leveraging your Advanced Track local environment setup and its integration with Grafana Alloy and other components.

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, specifically emphasizing Grafana's role in the Observability section and the "Highlighting Grafana Alloy" document.

Basic Use Case: Real-time Operational Dashboards

Objective: To demonstrate how Grafana visualizes real-time operational metrics collected by Grafana Alloy from various services (FastAPI, Kafka, Spark, cAdvisor) in a unified dashboard.

Role in Platform: Provide a single pane of glass for monitoring the health, performance, and resource utilization of all data platform components.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root. This includes fastapi\_ingestor (with Prometheus instrumentation), kafka, spark, cAdvisor, grafana-alloy, and grafana.

Verify Grafana is accessible: Navigate to http://localhost:3000 in your web browser. (Initial login: admin/admin if not configured otherwise).

Ensure Grafana Alloy is collecting metrics: Confirm grafana-alloy is running and its logs show successful scraping from fastapi\_ingestor:8000/metrics and cadvisor:8080/metrics, and forwarding to Grafana (as covered in "Highlighting Grafana Alloy: Basic Use Case").

Generate activity: Run python3 simulate\_data.py to continuously send data, ensuring all services are active and generating metrics.

Steps to Exercise:

Access Grafana UI: Open your web browser and go to http://localhost:3000.

Navigate to a Dashboard:

From the left-hand navigation, click on "Dashboards" and select a pre-provisioned dashboard (e.g., "Health Dashboard", "Docker Container Overview", or "Kafka Overview"). Your setup should ideally include a "Health Dashboard" that pulls metrics from various sources.

If no dashboards are pre-provisioned, you can import one from Grafana Labs (e.g., "Node Exporter Full" ID 1860 for host metrics, or "cAdvisor / Prometheus Host and Container" ID 14210, which relies on cAdvisor metrics).

Observe Real-time Metrics:

FastAPI Metrics: Look for panels showing "API Request Rate (RPS)" and "API Latency (P99)". These reflect the performance of your ingestion layer.

Kafka Metrics: Find panels for "Kafka Consumer Lag" (for raw\_financial\_transactions and raw\_insurance\_claims topics) and "Kafka Broker Throughput." These indicate the health of your streaming buffer and whether consumers are keeping up.

Spark Metrics: Observe panels showing "Spark CPU Utilization" or "Spark Memory Usage," reflecting the resources consumed by your data processing jobs.

Container Metrics (via cAdvisor): Look at overall "Container CPU Usage" and "Container Memory Usage" to see resource consumption across individual Docker services.

Verification:

Grafana Dashboards: All panels in the selected dashboards are actively populating with data, and the graphs reflect the ongoing activity of your data platform components (e.g., fluctuating RPS for FastAPI, low and stable consumer lag for Kafka, varying CPU/memory for Spark). This confirms the end-to-end flow of telemetry data and Grafana's ability to visualize it in real-time.

Advanced Use Case 1: Alerting on Service Level Objective (SLO) Violations

Objective: To demonstrate how to configure Grafana alerts to notify operational teams when key Service Level Objectives (SLOs) for the data platform are violated.

Role in Platform: Enable proactive monitoring and rapid response to system health deviations, reducing Mean Time To Detection (MTTD) and preventing minor issues from escalating into major incidents.

Setup/Configuration:

Ensure Basic Use Case is running: Data is flowing and metrics are being collected in Grafana.

Understand SLOs: Refer to testing-observability-addendum (Section 5.6.1) for examples of SLIs and SLOs (e.g., "Kafka Consumer Lag < 10,000 messages for 99.9% of time").

Identify a Metric for Alerting: We'll use Kafka consumer lag for raw\_financial\_transactions.

(Optional) Configure a Notification Channel: For real alerts, you'd configure an email, Slack, PagerDuty, etc., channel in Grafana ("Alerting" -> "Contact points"). For this local demo, you'll primarily observe the alert status in the Grafana UI.

Steps to Exercise:

Create a Grafana Alert Rule:

In Grafana UI, navigate to "Alerting" -> "Alert Rules".

Click "New alert rule".

Choose Data Source: Select your Prometheus data source (which Grafana Alloy sends to).

Define Query (PromQL): Enter a query for Kafka consumer lag.  
kafka\_consumergroup\_group\_lag{consumergroup="spark-financial-consumer-group", topic="raw\_financial\_transactions"}  
# Replace 'spark-financial-consumer-group' with your actual Spark consumer group name

Define Condition: Set a threshold that will be easily violated for demonstration.

Condition: WHEN last() OF A IS ABOVE 5000 (meaning if lag is over 5000 messages).

For: 2m (meaning the condition must be true for 2 consecutive minutes before the alert fires).

Configure Notifications (optional but recommended for real use): Add a "Contact point" to send notifications.

Add Annotations: This is crucial for alert fatigue mitigation. Provide summary, description, remediation steps, dashboard\_link, and runbook\_link (conceptual links to your internal runbook documentation, as detailed in testing-observability-addendum, Section 5.6.2).

summary: "High Kafka Consumer Lag for Financial Transactions"

description: "The Spark job consuming 'raw\_financial\_transactions' is falling behind. Data freshness for financial data is impacted."

remediation: "1. Check Spark job logs for errors (e.g., OOM). 2. Verify Spark cluster resources (CPU/memory) in Grafana. 3. Consider scaling Spark executors or optimizing job logic. 4. Refer to runbook for detailed steps."

dashboard\_link: http://localhost:3000/d/<your-kafka-dashboard-uid>

runbook\_link: /runbooks/kafka\_consumer\_lag.md

Save Alert Rule.

Simulate an SLO Violation:

Ensure simulate\_data.py is running and actively sending financial data to Kafka.

Pause the Spark container: docker compose pause spark. This will cause the Kafka consumer lag for raw\_financial\_transactions to rapidly increase.

Wait for the "For" duration you set (e.g., 2 minutes).

Observe Alert Firing:

In Grafana, go back to "Alerting" -> "Alert Rules".

Your newly created alert rule should transition from "OK" to "Pending" and then to "Firing" (red status).

Click on the alert rule to see its current state, active alerts, and the annotation details.

Verification:

Grafana Alerting: The alert rule correctly changes status to "Firing" when the defined condition (high Kafka consumer lag) is met for the specified duration.

Dashboards: The corresponding consumer lag panel in your Kafka dashboard will clearly show the spike in lag, providing visual context for the alert.

Alert Annotations: The alert's details display the rich context (summary, description, remediation, links), demonstrating how Grafana helps in managing alert fatigue and guiding incident response.

Advanced Use Case 2: Templating Dashboards for Dynamic Views

Objective: To demonstrate how Grafana's templating feature allows you to create dynamic dashboards where users can filter or select data based on variables (e.g., Kafka topic, container name, service name).

Role in Platform: Provide flexible, self-service monitoring interfaces, allowing different teams or users to customize their view of the data without creating countless static dashboards.

Setup/Configuration:

Ensure Basic Use Case is running: Metrics from multiple sources (FastAPI, Kafka, cAdvisor) are flowing into Grafana with various labels (job, topic, instance, container).

Steps to Exercise:

Create a New Dashboard:

In Grafana, click the "+" icon on the left-hand navigation and select "New Dashboard".

Click "Add a new panel" to start.

Add a Template Variable:

Go to "Dashboard settings" (the gear icon on the top right).

Select "Variables" from the left menu.

Click "Add variable".

Name: topic

Type: Query

Data source: Your Prometheus data source.

Query: label\_values(kafka\_consumergroup\_group\_lag, topic) (This fetches all unique Kafka topic names that report consumer lag metrics).

Selection Options: Enable "Multi-value" and "Include All option".

Click "Add". You'll now see a dropdown menu at the top of your dashboard.

Create Panels Using the Variable:

Go back to your dashboard (click "Dashboard" in the top left).

Add a new panel.

Query: In the PromQL query editor, use the variable:  
kafka\_consumergroup\_group\_lag{topic=~"$topic"}  
  
The ~ regex operator allows selecting multiple values from the dropdown.

Observe: The panel will initially show data for all topics (if "All" is selected).

Change Variable: Select a specific topic from the "topic" dropdown (e.g., raw\_financial\_transactions). The panel will instantly update to show data only for that topic.

Repeat for other metrics (Optional):

Add another variable for job (using label\_values(\_\_name\_\_, job) to get all job names).

Create a panel for FastAPI RPS: rate(http\_requests\_total{job=~"$job"}[1m]).

Change the job variable to switch between viewing FastAPI, cAdvisor, etc.

Verification:

Dynamic Dashboards: The dashboard demonstrates dynamic filtering and visualization based on the selected template variable, proving that Grafana can provide highly customizable views of your telemetry data. This empowers users to quickly focus on specific pipelines or services.

Advanced Use Case 3: Integrating with OpenMetadata for Data Context (Conceptual)

Objective: To conceptually demonstrate how Grafana dashboards can be enriched with context from OpenMetadata, bridging the gap between operational monitoring and data governance information.

Role in Platform: Provide a holistic view for data practitioners, allowing them to understand not just if a pipeline is failing, but what specific data assets are affected, their owners, and their business descriptions.

Setup/Configuration:

Ensure OpenMetadata is running and populated: Your Kafka topics and Delta Lake tables should be cataloged in OpenMetadata (http://localhost:8585).

Identify Relevant Assets: Know the URL structure for OpenMetadata assets (e.g., http://localhost:8585/metadata/explore/topics/<topic-fqn> or http://localhost:8585/metadata/explore/tables/<table-fqn>).

Steps to Exercise (Conceptual):

Add Custom Link to a Grafana Panel:

Go to an existing panel in a Grafana dashboard (e.g., your Kafka consumer lag panel for raw\_financial\_transactions).

Click on the panel title, then "Edit Panel".

Go to the "Links" tab in the panel options.

Click "Add link".

Title: "View in OpenMetadata"

Type: Absolute URL

URL:  
http://localhost:8585/metadata/explore/topics/{{\_\_series.labels.topic}}  
  
Explanation: {{\_\_series.labels.topic}} is a Grafana variable that will dynamically pull the topic label from the metric currently displayed in the panel.

Open in new tab: Yes.

Save the link and the panel.

Interact with the Link:

Go back to the dashboard view.

Click on the panel title (e.g., the Kafka consumer lag panel). A small dropdown will appear with "View in OpenMetadata".

Click the link. It should open a new tab directly to the corresponding Kafka topic's page in OpenMetadata.

Discuss Enhancements:

Annotations for Lineage: While not directly pulling full lineage into Grafana, you could set up Airflow DAGs to create Grafana annotations that point to specific Spline/OpenMetadata lineage URLs whenever a critical batch job completes, showing "New data loaded for X, see lineage at Y".

Embedded iFrames (Less Recommended): For internal tools, one could theoretically embed OpenMetadata views within Grafana using iFrame panels, though this can have security and layout complexities.

Enrichment in Grafana Alloy: Conceptually, Grafana Alloy could be extended (though more advanced) to pull descriptive metadata from OpenMetadata and attach it as labels to metrics, enriching them with business context (e.g., data\_owner, data\_sensitivity).

Verification (Conceptual):

Contextual Links: The ability to navigate directly from an operational metric in Grafana to its corresponding asset in OpenMetadata demonstrates a practical integration of observability and governance. This provides a richer understanding of data assets and their operational status for data consumers and operations teams. This bridges the gap between "what's broken" and "whose data is affected."

This concludes the guide for Grafana.

Highlighting Spline: Automated Data Lineage for Spark

Spline is an open-source tool designed specifically for automated data lineage tracking within Apache Spark jobs. It captures metadata about Spark transformations and provides a user interface for visualizing data flow. In your data platform, Spline provides critical visibility into how data is transformed from its raw sources through various Spark processing stages to its curated destinations, ensuring transparency and enabling easier debugging and governance.

This guide will demonstrate basic and advanced use cases of Spline, leveraging your Advanced Track local environment setup and its integration with Spark and OpenMetadata.

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, specifically Spline's role in the Orchestration & Governance Layer.

Basic Use Case: Capturing and Visualizing Spark Job Lineage

Objective: To demonstrate how Spline automatically intercepts and records the execution plan of Spark jobs, making the lineage visible in its UI.

Role in Platform: Provide immediate, granular insights into Spark data transformations, helping developers understand data flow and verify changes.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root. This includes spark and spline.

Verify Spline is accessible: Navigate to http://localhost:8081 in your web browser.

Ensure Spark is configured with Spline Agent: Your docker-compose.yml for the spark service should include the Spline agent configuration.  
Example docker-compose.yml snippet for Spark + Spline:  
# ...  
spark:  
 image: bitnami/spark:3.5.0  
 # ... other config  
 environment:  
 # ... other Spark env vars  
 # SPLINE CONFIGURATION  
 SPARK\_SUBMIT\_ARGS: >  
 --jars /opt/bitnami/spark/jars/spline-spark-agent-bundle\_2.12-0.7.1.jar  
 --driver-java-options "-javaagent:/opt/bitnami/spark/jars/spline-agent-bundle-0.7.1.jar"  
 --conf spark.spline.producer.url=http://spline:8081/producer  
 --conf spark.spline.persistence.factory=za.co.absa.spline.harvester.json.JsonLineagePersistenceFactory  
 --conf spark.spline.mode=ENABLED  
 --conf spark.spline.log.level=WARN # To reduce verbose logging  
 volumes:  
 - ./pyspark\_jobs:/opt/bitnami/spark/jobs  
 - spline\_jars:/opt/bitnami/spark/jars # Mount spline jars if needed, or pre-built into image  
 depends\_on:  
 spline:  
 condition: service\_healthy  
spline:  
 image: absaspline/spline:0.7.1 # Use the correct version  
 ports:  
 - "8081:8081" # Spline UI  
 environment:  
 SPLINE\_DATABASE\_CONNECTION\_URL: jdbc:h2:mem:spline;DB\_CLOSE\_DELAY=-1 # Or use a persistent DB  
 SPLINE\_DATABASE\_DRIVER: org.h2.Driver  
 SPLINE\_DATABASE\_USER: sa  
 SPLINE\_DATABASE\_PASSWORD: ""  
 healthcheck:  
 test: ["CMD", "curl", "-f", "http://localhost:8081/status || exit 1"]  
 interval: 10s  
 timeout: 5s  
 retries: 5  
# ...  
volumes:  
 spline\_jars: # Define volume if you're externalizing jars  
  
Note: The Spline agent JARs (spline-spark-agent-bundle\_2.12-0.7.1.jar and spline-agent-bundle-0.7.1.jar) need to be present in the Spark container's classpath. The spline\_jars volume is a conceptual way to manage this, or you might build a custom Spark image that includes them.

Run a Spark Job:

Submit one of your Spark jobs that writes to Delta Lake (e.g., pyspark\_jobs/streaming\_consumer.py or pyspark\_jobs/batch\_transformations.py).

For streaming\_consumer.py:  
docker exec -it 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 \  
 raw\_financial\_transactions kafka:29092 s3a://raw-data-bucket/financial\_data\_delta

Let it run for a few seconds/minutes to generate lineage.

Steps to Exercise:

Access Spline UI: Open your web browser and go to http://localhost:8081.

View Executions:

On the main Spline UI page, you should see a list of recent Spark job "Executions."

Find the entry corresponding to the Spark job you just submitted (e.g., KafkaToDeltaStream\_raw\_financial\_transactions or BatchETLTransformation).

Click on the execution.

Explore Lineage Graph:

Spline will display a visual graph representing the data lineage for that Spark job.

Observe: The graph typically shows:

Sources: Input data assets (e.g., Kafka topic raw\_financial\_transactions).

Transformation: A node representing the Spark job itself (the process).

Destinations: Output data assets (e.g., Delta Lake table raw-data-bucket/financial\_data\_delta).

You can click on nodes and edges to see more details about the data, schema, and transformation operations.

Verification:

Spline UI: A clear and accurate visual lineage graph is displayed for the executed Spark job, showing the source(s), the Spark transformation process, and the destination(s). This confirms Spline is correctly capturing and visualizing Spark lineage.

Advanced Use Case 1: Detailed Schema and Operation Tracking

Objective: To demonstrate how Spline captures not just the overall data flow, but also detailed information about schema evolution and the specific Spark operations performed within a job.

Role in Platform: Provide forensic detail for debugging schema drift issues, understanding the exact transformations applied to data, and validating data contract adherence.

Setup/Configuration:

Ensure Basic Use Case setup is complete.

Run a Spark Job with Transformations: Submit a job that involves multiple steps or schema changes, such as pyspark\_jobs/batch\_transformations.py, which reads raw Delta, performs joins/transformations, and writes to curated Delta.  
docker exec -it spark spark-submit \  
 --packages io.delta:delta-core\_2.12:2.4.0,org.postgresql:postgresql:42.6.0 \  
 --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\_batch  
  
(Ensure raw-data-bucket/financial\_data\_delta has some data).

Steps to Exercise:

Access Spline UI: http://localhost:8081.

Select the Batch Transformation Execution: Find the execution for BatchETLTransformation.

Explore "Schema" and "Operations" Details:

Schema Tab: Click on the input (e.g., s3a://raw-data-bucket/financial\_data\_delta) and output (e.g., s3a://curated-data-bucket/financial\_data\_curated\_batch) nodes in the lineage graph. You should see a detailed view of the schema (column names, types) for each dataset. Compare how the schema changes from raw to curated.

Operations Tab: Click on the central Spark job (process) node. This tab provides a breakdown of the logical plan of the Spark job. You'll see individual operations like LogicalRelation, Project, Filter, Join, Aggregate, SaveIntoDataSourceCommand.

Drill Down: Click on individual operations to see their specific parameters and the attributes (columns) they affect. For example, a Join operation will show the join condition, and a Project operation will show the selected and transformed columns.

Verification:

Spline UI: The "Schema" tab accurately displays the schema for input and output datasets, reflecting transformations. The "Operations" tab details the logical plan of the Spark job, including specific transformations like joins and projections, demonstrating Spline's deep insight into Spark's execution.

Advanced Use Case 2: Integrating Lineage with OpenMetadata

Objective: To demonstrate how Spline's collected lineage data is pushed to OpenMetadata, providing a unified view of data assets and their end-to-end data flow within the central data catalog.

Role in Platform: Centralize lineage information from various processing engines (starting with Spark), enabling a holistic understanding of data dependencies for governance, impact analysis, and compliance.

Setup/Configuration:

Ensure Basic Use Case and Advanced Use Case 1 (Spline data collection) are working.

Ensure OpenMetadata is running: http://localhost:8585.

Ensure OpenMetadata Spline Connector is configured and running: Your Airflow DAGs (e.g., openmetadata\_ingestion\_dag) should include a task to run the OpenMetadata Spline connector, which pulls lineage from Spline and ingests it into OpenMetadata.

This typically involves running a script in openmetadata\_ingestion\_scripts/ which uses the OpenMetadata Python client.

You might need to manually trigger this Airflow DAG if it's not on a schedule.

Steps to Exercise:

Run a Spark Job: Submit any Spark job that writes to Delta Lake (e.g., streaming\_consumer.py) to ensure fresh lineage is generated by Spline.

Trigger OpenMetadata Lineage Ingestion: Manually trigger the relevant Airflow DAG in the Airflow UI (http://localhost:8080) that runs the OpenMetadata Spline connector.

Access OpenMetadata UI: Go to http://localhost:8585.

Search for an Output Table: Search for the Delta Lake table that was the destination of your Spark job (e.g., raw-data-bucket.financial\_data\_delta).

Navigate to the "Lineage" Tab:

On the table's detail page, click the "Lineage" tab.

Observe: You should see a graphical representation of the lineage, showing the Kafka topic as a source, the Spark job as a process, and the Delta table as the destination. This lineage is pulled from Spline via the OpenMetadata connector.

Explore: Hover over nodes to see details, and if configured, you might see column-level lineage within OpenMetadata, showing how source columns map to target columns.

Verification:

OpenMetadata UI: The "Lineage" tab for the Spark-generated Delta Lake table correctly displays the lineage graph, confirming that OpenMetadata successfully ingested the lineage metadata from Spline. This is a crucial integration point for comprehensive data governance.

Advanced Use Case 3: Customizing Lineage and Event-Driven Lineage Capture (Conceptual)

Objective: To conceptually discuss how Spline can be customized (e.g., by adding custom attributes to lineage events) and how lineage capture can be made more event-driven, rather than relying solely on post-execution polling.

Role in Platform: Extend lineage capabilities to include custom business metadata, and enable more real-time updates to the data catalog's lineage graph.

Setup/Configuration (Conceptual Discussion):

Custom Attributes:

Spline allows adding custom "extra" metadata to lineage events. This can be done by configuring the Spline agent to read specific Spark properties or by modifying the Spark job itself to set these properties.

Example: Add a spark.spline.extra.tags=batch\_id:123,data\_owner:data\_team

Event-Driven Lineage (beyond current stable Spline):

While the current Spline version typically relies on the Spark agent sending lineage after a job completes, future developments or custom integrations could involve more real-time eventing.

Conceptually, a lightweight service could listen for Spark "job completion" events (e.g., from an Airflow XCom or Spark listener API) and immediately trigger a push of that job's lineage from Spline to OpenMetadata, reducing latency in metadata updates.

Steps to Exercise (Conceptual/Discussion):

Discuss Customizing Lineage:

Explain how you might configure Spark jobs to emit custom attributes (e.g., project\_name, pipeline\_id, business\_domain) as part of the Spline lineage.

Show how these custom attributes would then appear in the Spline UI when you inspect an execution's details, providing richer context for data governance.

Discuss how this allows data teams to tailor lineage to their specific organizational needs and reporting requirements.

Discuss Event-Driven Lineage Capture:

Explain the benefits of real-time lineage updates (e.g., faster visibility in OpenMetadata, more responsive impact analysis).

Describe a conceptual architecture where:

A Spark custom listener (or a mechanism in Airflow) detects Spark job completion.

An event (e.g., a message to Kafka/SQS) is sent with the Spline execution ID.

A lightweight Lambda or a small Airflow task consumes this event and immediately triggers the OpenMetadata Spline connector for that specific execution ID, rather than waiting for a scheduled poll.

Verification (Conceptual):

Enhanced Context: The ability to add custom attributes to lineage demonstrates how the data platform can capture business-specific context, making lineage more meaningful for various stakeholders.

Real-time Potential: Understanding event-driven lineage patterns highlights the path toward more immediate and dynamic data governance capabilities, aligning with real-time operational needs.

This concludes the guide for Spline.

Highlighting MinIO: Local S3-Compatible Object Storage

MinIO is an open-source object storage server that is compatible with Amazon S3 APIs. In your local data platform environment, MinIO serves as the crucial simulated S3 data lake, providing a cost-effective and convenient way to develop and test data pipelines that interact with object storage, including those leveraging Delta Lake. It offers the flexibility and scalability of object storage right on your development machine.

This guide will demonstrate basic and advanced use cases of MinIO, leveraging your Advanced Track local environment setup and its integration with Spark and other components.

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, particularly MinIO's role as an S3 replacement.

Basic Use Case: Storing Raw Data from Spark Structured Streaming

Objective: To demonstrate how MinIO acts as the landing zone for raw data, receiving continuous appends from Spark Structured Streaming jobs, simulating data flowing into an S3 raw zone.

Role in Platform: Provide scalable, durable, and cost-effective storage for large volumes of raw, semi-structured, and structured data, accessible via S3 API.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root. This includes minio and spark.

Verify MinIO is accessible: Navigate to http://localhost:9001 in your web browser. (Login with minioadmin/minioadmin).

Ensure Spark Streaming jobs are running: Your pyspark\_jobs/streaming\_consumer.py jobs (from the "Highlighting Apache Spark" document's Basic Use Case) should be actively consuming from Kafka and writing to s3a://raw-data-bucket/financial\_data\_delta and s3a://raw-data-bucket/insurance\_data\_delta.

Generate activity: Run python3 simulate\_data.py to continuously send data to FastAPI, ensuring Kafka topics are populated for Spark to consume.

Steps to Exercise:

Observe MinIO Console:

Go to http://localhost:9001 and log in.

Click on the raw-data-bucket.

You will see the financial\_data\_delta/ and insurance\_data\_delta/ directories.

Periodically refresh or navigate into these directories. You will observe new .parquet files appearing, along with the \_delta\_log directory. Each .parquet file represents a micro-batch of data written by Spark.

Query Data (via Spark-SQL from within Spark container):  
You can directly query the data stored in MinIO using Spark's SQL interface.  
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 COUNT(\*) FROM delta.\`s3a://raw-data-bucket/financial\_data\_delta\`;"  
  
Run this query multiple times over a few minutes.

Verification:

MinIO Console: The presence of \_delta\_log and continuously appearing .parquet files within the financial\_data\_delta and insurance\_data\_delta paths confirms that Spark is successfully writing data to MinIO.

Spark-SQL Count: The COUNT(\*) query should show an increasing number of records, demonstrating continuous data ingestion and storage in MinIO.

Advanced Use Case 1: Hosting Delta Lake for ACID Transactions and Time Travel

Objective: To demonstrate how MinIO, coupled with Delta Lake, provides ACID transaction capabilities and time travel, making your S3-compatible data lake reliable and auditable.

Role in Platform: Elevate raw object storage into a transactional data lakehouse, enabling reliable data upserts, deletes, and historical querying.

Setup/Configuration:

Ensure Basic Use Case is running: financial\_data\_delta is populated with streaming data.

pyspark\_jobs/delta\_merge\_cdc.py: This script (from "Highlighting Delta Lake" Advanced Use Case 3) demonstrates MERGE INTO operations and time travel, which rely on MinIO hosting the Delta Lake files.

Steps to Exercise:

Stop financial\_transactions streaming job: If it's still running from previous examples, stop it (Ctrl+C in its terminal or docker compose stop spark).

Run a Batch Update/Merge Job:

Submit the pyspark\_jobs/delta\_merge\_cdc.py job which performs an upsert (merge) operation on a Delta table hosted in MinIO.

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/delta\_merge\_cdc.py \  
 s3a://curated-data-bucket/financial\_transactions\_dim

Inspect MinIO after Merge:

Navigate to http://localhost:9001, then curated-data-bucket/financial\_transactions\_dim/.

You'll see new .parquet files and an updated \_delta\_log reflecting the merge operation. The number of .parquet files might not directly correspond to the number of records due to Delta Lake's file compaction and versioning.

Perform Time Travel Queries:

Get the history of the table to find previous versions:  
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 "DESCRIBE HISTORY delta.\`s3a://curated-data-bucket/financial\_transactions\_dim\`;"

Note a VERSION number from before your last MERGE INTO operation.

Query the table "as of" that historical version:  
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 \* FROM delta.\`s3a://curated-data-bucket/financial\_transactions\_dim\` VERSION AS OF <PREVIOUS\_VERSION\_NUMBER>;"  
  
Replace <PREVIOUS\_VERSION\_NUMBER> with the actual version number.

Verification:

MinIO Console: The \_delta\_log directory for financial\_transactions\_dim contains transaction files (e.g., 0000000000000000000X.json), demonstrating the transaction log.

Time Travel Query Results: The query VERSION AS OF successfully retrieves the state of the data from a past version, confirming MinIO's capability to host Delta Lake tables with time travel enabled.

Advanced Use Case 2: Serving Curated Data for Analytical Consumption

Objective: To demonstrate how MinIO hosts highly optimized, curated Delta Lake tables, making them readily available for analytical tools like Spark SQL or other query engines (conceptually).

Role in Platform: Provide a performant and structured layer for data consumers (BI tools, data scientists, machine learning models) to access high-quality, transformed data.

Setup/Configuration:

Ensure financial\_data\_curated\_batch is populated: Your batch transformation job (from "Highlighting Apache Spark" Advanced Use Case 1, pyspark\_jobs/batch\_transformations.py) should have written data to s3a://curated-data-bucket/financial\_data\_curated\_batch.

Spark ml\_model\_inference.py: This script (from "Highlighting Apache Spark" Advanced Use Case 2) reads from the curated path, demonstrating a consumer.

Steps to Exercise:

Inspect Curated Data in MinIO:

Navigate to http://localhost:9001, then curated-data-bucket/financial\_data\_curated\_batch/.

You will see the .parquet files (likely larger and fewer than in the raw zone due to compaction) and a \_delta\_log.

Query Curated Data with Spark-SQL:

Use Spark-SQL to directly query the curated 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 transaction\_id, amount, enriched\_category, is\_amount\_valid, processing\_timestamp FROM delta.\`s3a://curated-data-bucket/financial\_data\_curated\_batch\` LIMIT 10;"  
Observe: The data should be transformed, include enriched fields (like enriched\_category), and is\_amount\_valid flags, reflecting the quality and structure expected for analytical consumption.

Run a Conceptual ML Inference Job (reads from curated):

Submit the pyspark\_jobs/ml\_model\_inference.py job which reads from the curated path.

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/ml\_model\_inference.py \  
 s3a://curated-data-bucket/financial\_data\_curated\_batch  
Observe: The job logs will confirm it's reading from the specified MinIO path and performing feature engineering.

Verification:

MinIO Console: The curated-data-bucket contains the expected transformed data.

Spark-SQL Query: The query results confirm the data is cleaned, enriched, and validated, ready for direct use by downstream analytical applications.

ML Inference Job: Successful execution of the ML script demonstrates MinIO's role in providing structured, high-quality features for machine learning.

Advanced Use Case 3: Simulating S3 Event Notifications for Downstream Processes (Conceptual)

Objective: To conceptually demonstrate how MinIO can trigger event notifications (similar to AWS S3 Event Notifications) when new objects are created, which can then be consumed by other services to initiate downstream workflows.

Role in Platform: Enable event-driven micro-ETL or trigger specific processes (e.g., metadata ingestion, lightweight data validation Lambdas) as soon as new data lands in the data lake.

Setup/Configuration:

Enable MinIO Event Notifications: MinIO supports webhook notifications. You'd typically configure this in its startup command or configuration.

Create a Dummy Webhook Listener: For this demo, you'd need a simple HTTP server (e.g., a small Python Flask/FastAPI app) running in another container that acts as a webhook receiver.  
Example docker-compose.yml snippet (conceptual additions for webhook listener):  
# ...  
webhook\_listener:  
 build: ./webhook\_listener\_app # A simple Flask/FastAPI app  
 ports:  
 - "8081:8081" # Expose to host  
 # No healthcheck for simplicity  
# ...  
# MinIO service definition (add 'command' for events)  
minio:  
 image: minio/minio:latest  
 # ... existing config  
 command: server /data --console-address ":9001"  
 # Add configuration for notifications (conceptual, often via mc client or startup script)  
 # This part is typically done \*after\* MinIO starts, via `mc event add` command.  
 # For a Docker Compose setup, you might have an entrypoint script that runs this.  
 # E.g., `command: sh -c "minio server /data --console-address :9001 & mc alias set local http://localhost:9000 minioadmin minioadmin && mc mb local/my-bucket-events --ignore-existing && mc event add local/my-bucket-events arn:minio:sqs::1:webhook --suffix .parquet --event put,delete"`  
 # The `arn:minio:sqs::1:webhook` part is the important webhook target.  
 # You'd need to configure the webhook target: mc admin config set notify\_webhook:webhook\_target endpoint='http://webhook\_listener:8081/minio-event' queue\_limit=100  
  
Example webhook\_listener\_app/app.py (simple Flask app to receive webhooks):  
# webhook\_listener\_app/app.py  
from flask import Flask, request, jsonify  
  
app = Flask(\_\_name\_\_)  
  
@app.route('/minio-event', methods=['POST'])  
def minio\_event():  
 event = request.json  
 print("Received MinIO Event:")  
 print(json.dumps(event, indent=2))  
 # Process the event, e.g., trigger a Lambda, update a database  
 return jsonify({"status": "success", "message": "Event received"}), 200  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(host='0.0.0.0', port=8081, debug=True)

Steps to Exercise (Conceptual):

Start Services: Bring up minio, webhook\_listener, and other necessary services.

Configure MinIO Event Notification:

This is the trickiest part to automate directly in docker-compose.yml. You'd typically use the mc (MinIO Client) after MinIO starts.

From your host machine, after minio container is up, execute:  
docker exec -it minio bash  
# Inside minio container:  
mc alias set local http://localhost:9000 minioadmin minioadmin  
mc mb local/my-event-bucket --ignore-existing # Create a bucket for events  
mc event add local/my-event-bucket arn:minio:sqs::1:webhook --suffix .parquet --event put # Add put object event  
mc admin config set notify\_webhook:webhook\_target endpoint='http://webhook\_listener:8081/minio-event' queue\_limit=100 --console-address ":9001"  
# Exit the minio container.

Restart MinIO service in docker-compose if changes to eventing require it. docker compose restart minio

Upload a File to the Event Bucket:

Go to http://localhost:9001, navigate to my-event-bucket, and manually upload a .parquet file (or any file with the configured suffix).

Or, use mc cp <local\_file.parquet> local/my-event-bucket/.

Alternatively, configure a Spark job to write to s3a://my-event-bucket/.

Observe Webhook Listener Logs: Watch the logs of your webhook\_listener container.

Verification (Conceptual):

Webhook Listener Logs: The webhook\_listener container's logs will show the JSON payload of the MinIO event, containing details about the object put operation (bucket name, object key, timestamp, etc.). This demonstrates MinIO's ability to emit events for new objects, enabling event-driven architectures.

This feature is crucial for implementing patterns like serverless ETL where a Lambda function triggers upon new file arrival in S3.

This concludes the guide for MinIO.

Highlighting Docker/Docker Compose: Local Environment Orchestration

Docker and Docker Compose are the cornerstones of your local enterprise data platform, providing the essential capabilities for containerization and multi-container application orchestration. They ensure that your entire data stack – from FastAPI and Kafka to Spark and Grafana – runs in isolated, reproducible, and portable environments, closely mirroring a production cloud setup without the complexity of managing virtual machines directly.

This guide will demonstrate basic and advanced use cases of Docker and Docker Compose, 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, which extensively uses Docker Compose for environment provisioning.

Basic Use Case: Starting, Stopping, and Inspecting Services

Objective: To demonstrate the fundamental commands for managing your multi-service data platform using Docker Compose, including bringing services up, bringing them down, and inspecting their status and logs.

Role in Platform: Simplify the setup and teardown of the entire complex data stack, making it easy for developers to start and stop their local environment.

Setup/Configuration (Local Environment - Advanced Track):

Ensure Docker Desktop is running: Docker Desktop (or Docker Engine on Linux) must be active for Docker Compose to function.

Navigate to your project root: All Docker Compose commands are executed from the directory containing your docker-compose.yml file.  
cd /path/to/your/data-ingestion-platform

Steps to Exercise:

Bring up all services in detached mode:  
This command reads your docker-compose.yml and starts all defined services in the background. The --build flag ensures Docker images are rebuilt if there are changes to your Dockerfiles, and -d runs them in detached mode.  
docker compose up --build -d  
  
Observe: You will see messages indicating each service being created/started.

Check the status of running services:  
This command lists all services defined in your docker-compose.yml and shows their current state (running, exited, unhealthy) and exposed ports.  
docker compose ps  
  
Observe: All services should show running or healthy (if healthchecks are configured and passed).

View logs for all services:  
This command streams the logs from all running containers, aggregated in a single output. It's invaluable for initial troubleshooting.  
docker compose logs -f  
  
Observe: You'll see startup logs, application output, and any errors from all containers. Use Ctrl+C to exit. To see logs for a specific service: docker compose logs -f <service\_name> (e.g., docker compose logs -f spark).

Stop all services:  
This command gracefully stops all running containers defined in your docker-compose.yml.  
docker compose stop  
  
Observe: Services will transition from running to exited. docker compose ps will confirm they are stopped.

Stop and remove all services and their networks/volumes:  
This command stops all services, removes their containers, and by adding the -v flag, also removes any anonymous volumes created by Docker Compose. This is useful for a clean slate, but be cautious as it will remove persistent data if volumes are not explicitly named or host-mounted.  
docker compose down -v  
  
Observe: Confirmation that containers, networks, and volumes are removed.

Verification:

docker compose ps shows all services in the desired state (running/healthy or exited).

docker compose logs -f displays expected startup messages and no critical errors.

The environment can be reliably started and stopped, demonstrating fundamental control over the platform's lifecycle.

Advanced Use Case 1: Managing Service Dependencies and Health Checks

Objective: To demonstrate how depends\_on and healthcheck configurations in docker-compose.yml ensure that services start in the correct order and are truly ready before dependent services try to connect, enhancing local environment stability.

Role in Platform: Prevent common startup failures (e.g., Spark trying to connect to Kafka before Kafka is ready, Airflow trying to connect to PostgreSQL before it's initialized), leading to a more robust and predictable development environment.

Setup/Configuration:

Review docker-compose.yml dependencies: Look at services like kafka and spark, airflow-webserver and postgres.

kafka typically depends\_on: zookeeper.

spark often depends\_on: kafka and minio.

airflow-webserver depends\_on: postgres and airflow-scheduler.

Crucially, these depends\_on clauses should use condition: service\_healthy.

Example docker-compose.yml snippet illustrating healthchecks and dependencies:version: '3.8'  
services:  
 zookeeper:  
 image: confluentinc/cp-zookeeper:7.4.0  
 environment:  
 ZOOKEEPER\_CLIENT\_PORT: 2181  
 healthcheck: # Healthcheck for Zookeeper  
 test: ["CMD", "sh", "-c", "nc -z localhost 2181 || exit 1"]  
 interval: 5s  
 timeout: 3s  
 retries: 5  
 start\_period: 10s # Give it time to start before checking  
  
 kafka:  
 image: confluentinc/cp-kafka:7.4.0  
 depends\_on:  
 zookeeper:  
 condition: service\_healthy # Kafka waits for Zookeeper to be healthy  
 environment:  
 KAFKA\_BROKER\_ID: 1  
 KAFKA\_ZOOKEEPER\_CONNECT: 'zookeeper:2181'  
 KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT\_HOST://localhost:9092  
 KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT\_HOST:PLAINTEXT  
 KAFKA\_INTER\_BROKER\_LISTENER\_NAME: PLAINTEXT  
 KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1  
 healthcheck: # Healthcheck for Kafka  
 test: ["CMD", "sh", "-c", "kafka-topics --bootstrap-server localhost:9092 --list || exit 1"]  
 interval: 10s  
 timeout: 5s  
 retries: 5  
 start\_period: 20s  
  
 spark:  
 image: bitnami/spark:3.5.0  
 depends\_on:  
 kafka:  
 condition: service\_healthy # Spark waits for Kafka to be healthy  
 minio:  
 condition: service\_healthy # Spark waits for MinIO to be healthy  
 # ... other Spark configurations

Steps to Exercise:

Bring down services (if running): docker compose down -v to ensure a fresh start.

Bring up services again: docker compose up --build -d

Observe startup order and health:

Use docker compose ps repeatedly. You'll notice services like zookeeper and minio becoming (healthy) first.

Then, kafka will become (healthy) after zookeeper is healthy.

Finally, spark will start and become (healthy) after both kafka and minio are healthy.

Observe docker compose logs -f for specific messages indicating healthcheck probes passing or services waiting for dependencies.

Simulate a dependency failure (optional, for advanced testing):

While all services are running, manually stop zookeeper: docker compose stop zookeeper.

Observe Kafka's state and logs. It will likely become unhealthy or exited because its dependency is gone.

Restart zookeeper: docker compose start zookeeper.

Observe kafka recovering its healthy state.

Verification:

docker compose ps consistently reports (healthy) for all services once dependencies are met.

Logs clearly show services waiting for service\_healthy conditions before starting their main processes, demonstrating proper orchestration of startup order.

Advanced Use Case 2: Volume Management & Data Persistence

Objective: To demonstrate how Docker volumes are used to persist data generated by stateful services (databases, Kafka logs, Delta Lake files) across container restarts and even docker compose down operations.

Role in Platform: Ensure that your valuable data (e.g., PostgreSQL data, Kafka messages, Delta Lake snapshots) is not lost when containers are stopped, removed, or updated, providing a production-like persistence layer for local development.

Setup/Configuration:

Review docker-compose.yml for volumes: Identify named volumes and host-mounted bind mounts.

Named volumes: (e.g., postgres\_data, minio\_data) are managed by Docker and typically live in /var/lib/docker/volumes/ on your host. They are automatically created by Docker Compose if they don't exist and persist across docker compose down unless -v is explicitly used.

Bind mounts: (e.g., ./data/postgres:/var/lib/postgresql/data) map a host directory directly into the container. Data persists in the host directory.

Example docker-compose.yml snippet illustrating volume types:# ...  
services:  
 postgres:  
 image: postgres:15  
 environment:  
 POSTGRES\_DB: main\_db  
 POSTGRES\_USER: user  
 POSTGRES\_PASSWORD: password  
 volumes:  
 - postgres\_data:/var/lib/postgresql/data # Named volume for PostgreSQL data  
 # ...  
  
 minio:  
 image: minio/minio:latest  
 environment:  
 MINIO\_ROOT\_USER: minioadmin  
 MINIO\_ROOT\_PASSWORD: minioadmin  
 volumes:  
 - ./data/minio:/data # Bind mount for MinIO data (S3 bucket content)  
 # ...  
  
 kafka:  
 image: confluentinc/cp-kafka:7.4.0  
 environment:  
 KAFKA\_LOG\_DIRS: /kafka/kafka-logs # Internal path for Kafka logs  
 volumes:  
 - kafka\_data:/kafka/kafka-logs # Named volume for Kafka message logs  
 # ...  
  
 spark:  
 # ...  
 volumes:  
 - ./pyspark\_jobs:/opt/bitnami/spark/jobs # Bind mount for Spark job scripts  
 - ./data/spark-events:/opt/bitnami/spark/events # Bind mount for Spark History Server logs  
 # ...  
# Define named volumes at the bottom of the file  
volumes:  
 postgres\_data:  
 kafka\_data:

Ensure data/ subdirectories exist on your host for bind mounts (./data/minio, ./data/spark-events).

Steps to Exercise:

Start Services with Volumes: docker compose up --build -d

Generate Data:

Run python3 simulate\_data.py for a few minutes to ensure data is ingested into FastAPI, then Kafka, then processed by Spark to Delta Lake in MinIO, and finally persisted in PostgreSQL.

Verify data exists in PostgreSQL (e.g., docker exec -it postgres psql -U user -d main\_db -c "SELECT COUNT(\*) FROM financial\_transactions;").

Verify Delta Lake files exist in MinIO (check http://localhost:9001).

Stop and Remove Containers (keeping volumes):  
docker compose stop # Stops containers  
docker compose rm -s -v # Removes stopped containers and their anonymous volumes, but NOT named volumes or bind mounts  
  
Note: To prove named volumes persist, you must NOT use docker compose down -v. Just docker compose down will preserve named volumes. For bind mounts, the data is directly on your host, so it always persists unless you manually delete the host directory.

Verify Volume Persistence (for named volumes and bind mounts):

PostgreSQL:

Run docker compose up -d postgres to bring just the PostgreSQL container back up.

Connect to PostgreSQL and query: docker exec -it postgres psql -U user -d main\_db -c "SELECT COUNT(\*) FROM financial\_transactions;".

Expected: The count should be the same as before, demonstrating data persistence.

MinIO:

Access http://localhost:9001. The previously ingested Delta Lake files should still be visible in raw-data-bucket.

Kafka:

Run docker compose up -d kafka zookeeper.

Connect a Kafka consumer: docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw\_financial\_transactions --from-beginning.

Expected: You should see old messages from before the stop, demonstrating Kafka log persistence.

Clean up (optional): To remove named volumes and start completely fresh, use: docker volume rm <volume\_name> (e.g., docker volume rm data-ingestion-platform\_postgres\_data) or docker compose down -v if you're sure you want to delete all persistent data.

Verification:

Data stored in PostgreSQL, Kafka, and MinIO remains intact and accessible after stopping and restarting their respective containers, proving the effectiveness of volume management for data persistence.

Advanced Use Case 3: Network Isolation & Inter-Container Communication

Objective: To demonstrate how Docker Compose creates a private, isolated network for all your services, enabling seamless and secure communication between them using service names as hostnames, while also showing how to expose services to your host machine.

Role in Platform: Mimic a cloud-native private network, ensuring services can discover and communicate with each other securely without exposing all ports directly to the host's public network.

Setup/Configuration:

Review docker-compose.yml networking:

By default, Docker Compose creates a single bridge network for all services.

Services can communicate with each other using their service names (e.g., fastapi\_ingestor can connect to kafka:29092).

ports mapping ("HOST\_PORT:CONTAINER\_PORT") makes a container's port accessible from the host.

Example docker-compose.yml snippet illustrating networking:# ...  
services:  
 fastapi\_ingestor:  
 # ...  
 environment:  
 KAFKA\_BROKER: kafka:29092 # Refers to 'kafka' service name within the Docker network  
 ports:  
 - "8000:8000" # Exposes FastAPI to localhost:8000 on the host  
  
 kafka:  
 # ...  
 environment:  
 KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT\_HOST://localhost:9092  
 # PLAINTEXT://kafka:29092 is for inter-container communication  
 # PLAINTEXT\_HOST://localhost:9092 is for host-to-container communication (e.g., console consumers from host)  
 ports:  
 - "9092:9092" # Exposes Kafka to localhost:9092 on the host (for external clients)  
  
 spark:  
 # ...  
 environment:  
 KAFKA\_BROKER: kafka:29092 # Spark connects to Kafka using its service name  
 MINIO\_HOST: minio:9000 # Spark connects to MinIO using its service name and port  
 # No ports exposed by Spark itself for general use in this setup, only for Spark UI

Steps to Exercise:

Start all services: docker compose up --build -d

Verify Inter-Container Communication (FastAPI to Kafka):

Run python3 simulate\_data.py.

Open docker compose logs fastapi\_ingestor. You should see logs confirming successful message publishing to kafka:29092, demonstrating internal communication.

Verify Host-to-Container Communication (Accessing FastAPI/Kafka from Host):

Access FastAPI health check from your host browser/curl: http://localhost:8000/health.

Run a Kafka console consumer directly from your host machine (if Kafka CLI is installed, otherwise use docker exec -it kafka ... which implicitly uses the host network to connect to Kafka's exposed port):  
# If Kafka CLI is installed on host  
kafka-console-consumer --bootstrap-server localhost:9092 --topic raw\_financial\_transactions --from-beginning

These actions confirm that services with exposed ports are accessible from your host machine.

Verify Network Isolation (Conceptual):

Try to directly access an internal-only port of a container from your host that is not mapped in ports (e.g., PostgreSQL's internal port 5432 if not mapped). This attempt should fail with a connection refused error, demonstrating isolation.

Inside any container (e.g., docker exec -it fastapi\_ingestor bash), try ping kafka or curl http://minio:9000. These commands should succeed, confirming that service names resolve within the Docker network.

Verification:

FastAPI successfully publishes to Kafka using the Kafka service name within the Docker network.

You can access FastAPI and Kafka (via exposed ports) from your host machine's browser/terminal.

Attempting to access non-exposed internal ports from the host fails, while internal container-to-container communication by service name succeeds, demonstrating the effective network isolation and routing provided by Docker Compose.

This concludes the guide for Docker and Docker Compose.

Deep Dive: Applying Platform Concepts to Snowflake

This document explores how the architectural principles and component functionalities of your locally deployed enterprise data platform translate to a cloud data warehouse environment, specifically Snowflake. While your local setup uses open-source components like Spark, Kafka, and Delta Lake, Snowflake offers managed, highly scalable, and performant services that fulfill similar roles, often with a serverless or "data warehouse as a service" paradigm.

Understanding these equivalences is crucial for migrating your platform to a cloud-native solution and leveraging the specific strengths of Snowflake for modern data analytics.

1. Core Platform Components: Snowflake Equivalents

Let's map the key components of your local data platform to their corresponding functionalities and services within the Snowflake ecosystem:

2. Interactive How-Tos: Applying Concepts to Snowflake

Let's walk through key data platform scenarios and how they are implemented using Snowflake.

Basic Use Case: Ingesting Semi-Structured Data and Querying in Snowflake

Objective: To demonstrate how Snowflake efficiently ingests semi-structured data (like your Kafka messages) into a table using its VARIANT data type and COPY INTO command, and then queries it using SQL.

Role in Platform: Act as the central repository for raw and structured data, leveraging Snowflake's native capabilities for semi-structured data.

Setup/Configuration (Conceptual Snowflake Environment):

Snowflake Account: Access to a Snowflake account and a database/schema.

Snowflake Stage: An internal or external stage (e.g., pointing to an S3 bucket or MinIO if accessible from Snowflake) where your raw JSON data would land. For this example, we'll assume files are landed on an internal stage.

Steps to Exercise (Conceptual Snowflake Operations):

Prepare Sample Data (JSON Lines):  
Imagine this JSON is in files in a Snowflake Internal Stage (e.g., @my\_internal\_stage/financial\_transactions/).  
# financial\_transaction\_1.json  
{"transaction\_id": "FT-001", "timestamp": "2024-01-01T10:00:00Z", "account\_id": "ACC-001", "amount": 100.50, "currency": "USD"}  
{"transaction\_id": "FT-002", "timestamp": "2024-01-01T10:05:00Z", "account\_id": "ACC-002", "amount": 200.75, "currency": "EUR", "merchant": "ShopCo"}  
  
Note the merchant field in FT-002 is optional, demonstrating semi-structured nature.

Create a Target Table with VARIANT Column:  
-- Connect to your Snowflake worksheet  
USE DATABASE YOUR\_DATABASE;  
USE SCHEMA YOUR\_SCHEMA;  
  
CREATE TABLE IF NOT EXISTS RAW\_FINANCIAL\_TRANSACTIONS (  
 RAW\_DATA VARIANT,  
 LOAD\_TIMESTAMP TIMESTAMP\_NTZ DEFAULT CURRENT\_TIMESTAMP()  
);

Load Data using COPY INTO:  
In a real scenario, you would have files in an external stage (e.g., S3) or use Snowpipe. For this basic example, we simulate copying from an internal stage where files were uploaded.  
-- Assume files are already in @my\_internal\_stage/financial\_transactions/  
-- You can upload files to an internal stage using Snowflake UI or SnowSQL PUT command.  
-- Example PUT command (from your local machine assuming file exists):  
-- PUT file://<local\_path>/financial\_transaction\_1.json @my\_internal\_stage/financial\_transactions/ AUTO\_COMPRESS=TRUE;  
  
COPY INTO RAW\_FINANCIAL\_TRANSACTIONS (RAW\_DATA)  
FROM @my\_internal\_stage/financial\_transactions/  
FILE\_FORMAT = (TYPE = JSON);  
  
-- Or, if loading from external S3 bucket directly:  
-- COPY INTO RAW\_FINANCIAL\_TRANSACTIONS (RAW\_DATA)  
-- FROM 's3://your-s3-bucket/path/to/json/'  
-- CREDENTIALS = (AWS\_KEY\_ID = 'your\_key\_id' AWS\_SECRET\_KEY = 'your\_secret\_key')  
-- FILE\_FORMAT = (TYPE = JSON);

Query the Semi-Structured Data:  
SELECT  
 RAW\_DATA:transaction\_id::VARCHAR AS transaction\_id,  
 RAW\_DATA:timestamp::TIMESTAMP\_NTZ AS transaction\_timestamp,  
 RAW\_DATA:account\_id::VARCHAR AS account\_id,  
 RAW\_DATA:amount::FLOAT AS amount,  
 RAW\_DATA:currency::VARCHAR AS currency,  
 RAW\_DATA:merchant::VARCHAR AS merchant\_name, -- Accessing an optional field  
 LOAD\_TIMESTAMP  
FROM  
 RAW\_FINANCIAL\_TRANSACTIONS  
LIMIT 10;  
  
Observe: The query extracts specific fields from the VARIANT column using dot notation and type casting. The merchant\_name will be NULL for FT-001 and populated for FT-002.

Verification:

Snowflake Worksheet Output: The COPY INTO command reports successful rows loaded. The SELECT query correctly parses and extracts fields from the VARIANT JSON, demonstrating Snowflake's ability to handle schema flexibility.

Advanced Use Case 1: Streamlining Data Ingestion with Snowpipe

Objective: To demonstrate how Snowpipe (or Snowpipe Streaming for even lower latency) can automatically ingest new data files arriving in an external stage (like S3) into a Snowflake table, providing a continuous ingestion pipeline similar to Kafka + Spark streaming.

Role in Platform: Automate continuous, low-latency data ingestion from cloud storage into Snowflake, replacing the need for a manually managed streaming consumer.

Setup/Configuration (Conceptual Snowflake Environment):

Snowflake Account & External Stage: An external stage configured to an S3 bucket.

AWS S3 Bucket & Event Notifications: An S3 bucket where new files will land, with S3 Event Notifications configured to publish messages to an SQS queue.

Snowflake Integration: A Snowflake STORAGE INTEGRATION and NOTIFICATION INTEGRATION to securely connect to S3 and SQS.

Steps to Exercise (Conceptual Snowflake & AWS Operations):

Create File Format (if not exists):  
CREATE FILE FORMAT IF NOT EXISTS JSON\_FORMAT  
TYPE = JSON  
STRIP\_OUTER\_ARRAY = FALSE; -- Important if your JSON is an array of objects per file

Create Stage (External):  
CREATE OR REPLACE STAGE RAW\_FINANCIAL\_TRANSASACTIONS\_STAGE  
 URL = 's3://your-s3-raw-bucket/financial\_transactions/'  
 STORAGE\_INTEGRATION = s3\_storage\_integration\_for\_data\_platform; -- Your pre-configured storage integration

Create Target Table:  
CREATE TABLE IF NOT EXISTS RAW\_FINANCIAL\_TRANSACTIONS\_SNOWPIPE (  
 RAW\_DATA VARIANT,  
 FILE\_NAME VARCHAR,  
 LOAD\_TIMESTAMP TIMESTAMP\_NTZ DEFAULT CURRENT\_TIMESTAMP()  
);

Create Snowpipe:  
CREATE OR REPLACE PIPE FINANCIAL\_TRANSACTIONS\_PIPE  
 AUTO\_INGEST = TRUE  
 AS  
 COPY INTO RAW\_FINANCIAL\_TRANSACTIONS\_SNOWPIPE (RAW\_DATA, FILE\_NAME)  
 FROM (SELECT $1, METADATA$FILENAME FROM @RAW\_FINANCIAL\_TRANSACTIONS\_STAGE)  
 FILE\_FORMAT = (TYPE = JSON);  
  
This creates a pipe that listens to SQS notifications from S3 and automatically loads new files.

Get SQS Queue ARN from Snowpipe:  
SHOW PIPES LIKE 'FINANCIAL\_TRANSACTIONS\_PIPE';  
-- Look for 'notification\_channel' column in the output, this is the SQS Queue ARN.  
  
You would take this SQS ARN and configure your S3 bucket's event notification to publish s3:ObjectCreated:\* events to this SQS queue.

Simulate New File Arrival (Manual Upload or Programmatic from FastAPI/Lambda):  
In your local environment, you would run a script that uploads a new JSON file to your S3 bucket (which the external stage points to).  
From your local FastAPI (now configured to upload to S3 instead of MinIO or Kafka in a cloud-migrated scenario), trigger new financial transactions.

Monitor Snowpipe Progress:  
SELECT \*  
FROM TABLE(INFORMATION\_SCHEMA.PIPE\_USAGE\_HISTORY(  
 DATE\_RANGE\_START=>DATEADD('hour', -1, CURRENT\_TIMESTAMP()),  
 PIPE\_NAME=>'FINANCIAL\_TRANSACTIONS\_PIPE'));  
  
This query shows if files were processed and any errors.

Query Data in Snowflake:  
SELECT COUNT(\*) FROM RAW\_FINANCIAL\_TRANSACTIONS\_SNOWPIPE;  
  
Wait a few moments after uploading a file, then run this query. The count should increase.

Verification:

Snowflake Pipe History: The PIPE\_USAGE\_HISTORY shows new files being detected and loaded.

Snowflake Table Count: The RAW\_FINANCIAL\_TRANSACTIONS\_SNOWPIPE table accumulates new records automatically after files are dropped into S3, demonstrating Snowpipe's automated, continuous ingestion.

Advanced Use Case 2: Feature Engineering and Transformations with Snowpark

Objective: To demonstrate how Snowpark (Snowflake's developer experience for Python/Java/Scala) can be used to perform complex data transformations and feature engineering directly within Snowflake's compute engine, similar to how PySpark jobs run on Spark.

Role in Platform: Perform scalable data transformations, aggregations, and feature engineering on data stored in Snowflake, leveraging its elastic compute without moving data out of the warehouse.

Setup/Configuration (Conceptual Snowflake Environment + Python):

Snowflake Account & Warehouse: Access to a Snowflake account and an active warehouse.

Snowflake Table with Raw Data: Ensure RAW\_FINANCIAL\_TRANSACTIONS\_SNOWPIPE (from previous use case) has data.

Snowflake Client (Python): Install snowflake-snowpark-python on your local machine.

Steps to Exercise (Conceptual Python Script using Snowpark):

Create a Python Script (snowpark\_feature\_engineering.py):  
# snowpark\_feature\_engineering.py  
from snowflake.snowpark import Session  
from snowflake.snowpark.functions import col, count, sum, avg, to\_date, lit, current\_timestamp  
import json  
import os  
  
# --- Snowflake Connection Configuration ---  
# Replace with your actual Snowflake connection details  
connection\_parameters = {  
 "account": os.getenv("SNOWFLAKE\_ACCOUNT", "your\_account\_id"),  
 "user": os.getenv("SNOWFLAKE\_USER", "your\_user"),  
 "password": os.getenv("SNOWFLAKE\_PASSWORD", "your\_password"),  
 "role": os.getenv("SNOWFLAKE\_ROLE", "SYSADMIN"),  
 "warehouse": os.getenv("SNOWFLAKE\_WAREHOUSE", "COMPUTE\_WH"),  
 "database": os.getenv("SNOWFLAKE\_DATABASE", "YOUR\_DATABASE"),  
 "schema": os.getenv("SNOWFLAKE\_SCHEMA", "YOUR\_SCHEMA")  
}  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 session = None  
 try:  
 print("Creating Snowpark session...")  
 session = Session.builder.configs(connection\_parameters).create()  
 print(f"Snowpark session created successfully. Current database: {session.get\_current\_database()}, schema: {session.get\_current\_schema()}")  
  
 input\_table\_name = "RAW\_FINANCIAL\_TRANSACTIONS\_SNOWPIPE"  
 output\_table\_name = "CURATED\_FINANCIAL\_FEATURES\_DAILY"  
  
 print(f"Reading raw data from: {input\_table\_name}")  
 # Read from the raw table, parsing the VARIANT column  
 df\_raw = session.table(input\_table\_name).select(  
 col("RAW\_DATA"):("transaction\_id").as\_("transaction\_id"),  
 col("RAW\_DATA"):("timestamp").as\_("timestamp"),  
 col("RAW\_DATA"):("account\_id").as\_("account\_id"),  
 col("RAW\_DATA"):("amount").as\_("amount").cast("float"),  
 col("RAW\_DATA"):("currency").as\_("currency")  
 # Add other fields as needed  
 )  
 df\_raw.show(5)  
  
 print("Performing feature engineering: daily aggregates per account...")  
 # Convert timestamp to date, then group and aggregate  
 df\_features = df\_raw.withColumn("transaction\_date", to\_date(col("timestamp"))) \  
 .groupBy("account\_id", "transaction\_date") \  
 .agg(  
 count(col("transaction\_id")).alias("daily\_transaction\_count"),  
 sum(col("amount")).alias("daily\_total\_amount"),  
 avg(col("amount")).alias("daily\_average\_amount")  
 ) \  
 .withColumn("feature\_created\_at", current\_timestamp())  
  
 print("Schema of engineered features:")  
 df\_features.show(5)  
  
 # Write the engineered features to a new curated table in Snowflake  
 print(f"Writing engineered features to: {output\_table\_name}")  
 df\_features.write.mode("overwrite").save\_as\_table(output\_table\_name)  
 print(f"Feature engineering job completed. Data written to {output\_table\_name}.")  
  
 except Exception as e:  
 print(f"An error occurred: {e}")  
 import traceback  
 traceback.print\_exc()  
 finally:  
 if session:  
 session.close()  
 print("Snowpark session closed.")

Run the Python Script:  
python3 snowpark\_feature\_engineering.py  
  
Ensure SNOWFLAKE\_ACCOUNT, SNOWFLAKE\_USER, SNOWFLAKE\_PASSWORD (or other auth methods) are set as environment variables or directly in the script.

Verify Data in Snowflake:

In your Snowflake worksheet, query the new table:

SELECT \* FROM CURATED\_FINANCIAL\_FEATURES\_DAILY LIMIT 10;

Verification:

Script Output: The Python script prints messages indicating successful session creation, data reading, transformation, and writing.

Snowflake Worksheet: The CURATED\_FINANCIAL\_FEATURES\_DAILY table is created and populated with the aggregated features, demonstrating Snowpark's capability to perform complex ETL/feature engineering directly in Snowflake.

Advanced Use Case 3: Data Lineage and Governance with Snowflake

Objective: To conceptually explain how Snowflake's native features (ACCESS\_HISTORY, QUERY\_HISTORY) provide rich data lineage, and how this integrates with external data catalog tools (like OpenMetadata).

Role in Platform: Provide robust, automated data lineage within the warehouse, enabling comprehensive data governance, impact analysis, and compliance.

Setup/Configuration (Conceptual Snowflake & OpenMetadata Integration):

Snowflake Account: Ensure query history and access history are enabled (they are by default for most accounts).

OpenMetadata with Snowflake Connector: An OpenMetadata instance configured with a Snowflake connector.

Steps to Exercise (Conceptual Discussion):

Snowflake Native Lineage (ACCESS\_HISTORY & QUERY\_HISTORY):

ACCOUNT\_USAGE.ACCESS\_HISTORY View: This view captures comprehensive lineage information, including which objects (tables, views) were read and written by which queries.  
-- Example query to see access history for a table  
SELECT  
 QUERY\_ID,  
 QUERY\_TEXT,  
 BASE\_OBJECTS\_ACCESSED, -- Objects read  
 DIRECT\_OBJECTS\_MODIFIED -- Objects written  
FROM  
 SNOWFLAKE.ACCOUNT\_USAGE.ACCESS\_HISTORY  
WHERE  
 QUERY\_TYPE = 'INSERT' OR QUERY\_TYPE = 'CREATE\_TABLE\_AS\_SELECT'  
 AND QUERY\_START\_TIME >= DATEADD('day', -7, CURRENT\_TIMESTAMP())  
ORDER BY  
 QUERY\_START\_TIME DESC  
LIMIT 10;

ACCOUNT\_USAGE.QUERY\_HISTORY View: Provides details about every query executed, including user, warehouse, duration, and associated tags. This can be joined with ACCESS\_HISTORY for a full picture.  
-- Example: Find queries that read from RAW\_FINANCIAL\_TRANSACTIONS\_SNOWPIPE  
SELECT  
 qh.QUERY\_ID,  
 qh.QUERY\_TEXT,  
 qh.USER\_NAME,  
 qh.WAREHOUSE\_NAME,  
 ah.BASE\_OBJECTS\_ACCESSED,  
 ah.DIRECT\_OBJECTS\_MODIFIED  
FROM  
 SNOWFLAKE.ACCOUNT\_USAGE.QUERY\_HISTORY qh  
JOIN  
 SNOWFLAKE.ACCOUNT\_USAGE.ACCESS\_HISTORY ah ON qh.QUERY\_ID = ah.QUERY\_ID  
WHERE  
 ARRAY\_CONTAINS(  
 'YOUR\_DATABASE.YOUR\_SCHEMA.RAW\_FINANCIAL\_TRANSACTIONS\_SNOWPIPE'::VARIANT, -- Replace with full object path  
 ah.BASE\_OBJECTS\_ACCESSED  
 )  
ORDER BY qh.START\_TIME DESC  
LIMIT 10;

OpenMetadata with Snowflake Connector:

Configure Connector: In OpenMetadata UI, add a new "Service" of type "Database" and choose "Snowflake." Provide connection details (account, role, warehouse, database, schema).

Ingestion Workflows: Configure and run ingestion workflows in OpenMetadata to:

Metadata Ingestion: Pull table/view schemas, descriptions, and column details.

Profiler Ingestion: Run data profiling to get statistics (row counts, min/max, nulls) for tables.

Usage Ingestion: This is where the magic happens for lineage. The OpenMetadata Snowflake connector can parse QUERY\_HISTORY to infer lineage relationships (e.g., if SELECT \* FROM A JOIN B creates C, it infers A and B feed C).

View Lineage in OpenMetadata: After successful ingestion, navigate to a Snowflake table in OpenMetadata (e.g., CURATED\_FINANCIAL\_FEATURES\_DAILY). Go to its "Lineage" tab.

Expected: You should see a graphical representation showing the RAW\_FINANCIAL\_TRANSACTIONS\_SNOWPIPE table (source), a conceptual "transformation" node (representing the Snowpark job or SQL transformation), and the CURATED\_FINANCIAL\_FEATURES\_DAILY table (destination).

Verification (Conceptual):

Snowflake Query History: Successfully query ACCESS\_HISTORY and QUERY\_HISTORY to manually trace data flow.

OpenMetadata UI: The data catalog accurately reflects Snowflake schemas, and the "Lineage" tab displays end-to-end data flow for tables processed within Snowflake, demonstrating effective data governance and lineage tracking. This highlights how Snowflake's native capabilities, combined with tools like OpenMetadata, provide a robust solution for data transparency.

This concludes the deep dive into applying your platform concepts to Snowflake.

Highlighting PostgreSQL: Robust Relational Database

PostgreSQL is a powerful, open-source object-relational database system known for its strong adherence to SQL standards, reliability, feature robustness, and performance. In your data platform, PostgreSQL serves multiple critical roles: as a reliable datastore for structured data, a repository for application-specific metadata (e.g., configurations), and most notably, as the metastore for Apache Airflow, storing DAG definitions, task states, and historical runs.

This guide will demonstrate basic and advanced use cases of PostgreSQL, leveraging your Advanced Track local environment setup and its integration with FastAPI, Airflow, and Spark.

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: Application Metadata and Health Checks

Objective: To demonstrate PostgreSQL's role as a reliable storage for application-specific metadata and how basic connectivity can be verified.

Role in Platform: Provide a transactional and consistent store for structured data, configuration, and state management for various services.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root. This includes the postgres service.

Verify PostgreSQL is accessible: Check Docker logs for the postgres container (docker compose logs postgres). It should show messages indicating readiness.

Install psql (PostgreSQL interactive terminal): If not already installed on your host, you can use docker exec to access it inside the container.

Install psql locally: PostgreSQL Downloads

Or execute inside container: docker exec -it postgres psql -U user -d main\_db

Steps to Exercise:

Connect to PostgreSQL:  
psql -h localhost -p 5432 -U user -d main\_db  
  
(Enter password when prompted).  
If connecting from inside another container via docker exec, use psql -h postgres -p 5432 -U user -d main\_db as postgres is the service name in the Docker network.

Verify Connection and Database Existence:

Once connected, you should see the main\_db=# prompt.

Run a simple query: SELECT current\_database(); (Expected: main\_db).

Create a Sample Table for Application Metadata:  
CREATE TABLE IF NOT EXISTS app\_configs (  
 config\_key VARCHAR(255) PRIMARY KEY,  
 config\_value TEXT,  
 last\_updated TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);

Insert and Query Sample Metadata:  
INSERT INTO app\_configs (config\_key, config\_value) VALUES ('data\_ingestion\_enabled', 'true');  
INSERT INTO app\_configs (config\_key, config\_value) VALUES ('max\_batch\_size', '1000');  
SELECT \* FROM app\_configs;

Verification:

psql Output: You successfully connect, create a table, insert data, and retrieve it.

Docker Logs: The postgres container logs show the executed queries and successful operations. This confirms basic connectivity and data manipulation capabilities.

Advanced Use Case 1: Apache Airflow Metastore Management

Objective: To demonstrate PostgreSQL's crucial role as the backend metastore for Apache Airflow, storing all DAG definitions, task instances, historical runs, and connections.

Role in Platform: Provide a robust, transactional, and scalable persistence layer for Airflow's operational data, enabling Airflow to function reliably and recover from failures.

Setup/Configuration:

Ensure all Advanced Track services are running: This includes postgres, airflow-init, airflow-webserver, airflow-scheduler, airflow-worker.

Verify Airflow Initialization: The airflow-init service should have completed successfully on startup (check its logs). This service typically runs airflow db upgrade and creates the admin user.

Access Airflow UI: http://localhost:8080 (login admin/admin).

Steps to Exercise:

Access Airflow UI: Navigate to http://localhost:8080.

Observe DAGs and Runs:

In the Airflow UI, go to "DAGs". You should see your defined DAGs (e.g., financial\_ingestion\_dag.py, insurance\_transformation\_dag.py).

Trigger a DAG (e.g., financial\_ingestion\_dag).

Observe the DAG's status and task instances in the UI.

Inspect PostgreSQL for Airflow Metastore Data:

Connect to the main\_db in PostgreSQL: psql -h localhost -p 5432 -U user -d main\_db.

List Airflow tables: \dt airflow.\*; (Expected: A long list of tables prefixed with airflow\_ or ab\_, such as airflow\_dag, airflow\_taskinstance, airflow\_log, etc.).

Query DAG information:  
SELECT dag\_id, is\_active FROM airflow\_dag ORDER BY dag\_id;

Query Task Instance information:  
SELECT dag\_id, task\_id, state, execution\_date FROM airflow\_taskinstance ORDER BY execution\_date DESC LIMIT 10;

Query Connections: (Airflow stores connection details in the metastore).  
SELECT conn\_id, conn\_type, host, port FROM connection;  
  
Note: Sensitive details like passwords are encrypted in the metastore.

Simulate Airflow Component Restart and Verify State Persistence:

Stop airflow-webserver and airflow-scheduler: docker compose stop airflow-webserver airflow-scheduler.

Start them again: docker compose start airflow-webserver airflow-scheduler.

Access Airflow UI. The state of your DAGs (e.g., if a DAG was running, its state should be restored, or if a new run was scheduled, it should still be there) should be preserved.

Verification:

Airflow UI: DAGs, task states, and connections are correctly displayed and managed, even after restarting Airflow components.

PostgreSQL Queries: Direct queries to the main\_db reveal the underlying Airflow metadata, confirming that PostgreSQL is robustly storing all Airflow operational data, which enables state persistence and recovery.

Advanced Use Case 2: Reference Data Management for Spark Jobs

Objective: To demonstrate how PostgreSQL can serve as a central repository for "reference data" (e.g., lookup tables, master data) that Spark batch or streaming jobs can join with to enrich raw incoming data.

Role in Platform: Provide a consistent and accessible source of static or slowly changing dimension data for enrichment processes within the data pipeline.

Setup/Configuration:

Ensure PostgreSQL and Spark are running.

Populate Reference Data: Connect to PostgreSQL and create a merchant\_lookup table with sample data.  
Example SQL:  
CREATE TABLE IF NOT EXISTS merchant\_lookup (  
 merchant\_id VARCHAR(50) PRIMARY KEY,  
 merchant\_name VARCHAR(255) NOT NULL,  
 category VARCHAR(100)  
);  
  
INSERT INTO merchant\_lookup (merchant\_id, merchant\_name, category) VALUES  
('MER-ABC', 'Alpha Mart', 'Groceries'),  
('MER-XYZ', 'Tech Gadgets Inc.', 'Electronics'),  
('MER-123', 'HealthPlus Pharmacy', 'Healthcare');

Spark Job: Your pyspark\_jobs/batch\_transformations.py script (from "Highlighting Apache Spark" Advanced Use Case 1) already includes logic to connect to PostgreSQL and join with merchant\_lookup.

Steps to Exercise:

Ensure raw-data-bucket/financial\_data\_delta is populated with some data that includes merchant\_id.

Submit the Spark batch transformation job:  
docker exec -it spark spark-submit \  
 --packages io.delta:delta-core\_2.12:2.4.0,org.postgresql:postgresql:42.6.0 \  
 --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\_enriched  
  
(Note: Using a new output path to clearly show the enriched data).

Monitor Spark Job: Observe the console output for Spark logs indicating a successful read from PostgreSQL and the transformation process.

Query Enriched Data: After the Spark job completes, query the output Delta Lake table in MinIO using Spark SQL to confirm the enriched\_category field is populated.  
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 transaction\_id, merchant\_id, enriched\_category FROM delta.\`s3a://curated-data-bucket/financial\_data\_curated\_enriched\` WHERE merchant\_id IS NOT NULL LIMIT 10;"

Verification:

Spark Job Logs: Confirm that Spark successfully connected to PostgreSQL and performed the join operation.

Delta Lake Data: The enriched\_category column in the financial\_data\_curated\_enriched table contains values from the merchant\_lookup table, demonstrating successful data enrichment from PostgreSQL.

Advanced Use Case 3: Database Monitoring and Backup/Restore (Conceptual)

Objective: To conceptually demonstrate how PostgreSQL's operational health can be monitored and how backup and restore procedures are critical for disaster recovery, especially for the Airflow metastore.

Role in Platform: Ensure the availability and recoverability of critical structured data and Airflow's operational state, a key aspect of defining RPO/RTO.

Setup/Configuration (Conceptual Discussion):

Monitoring: Your grafana-alloy and grafana setup would typically include a Prometheus postgres\_exporter (often run as a sidecar container to PostgreSQL) to collect PostgreSQL-specific metrics (e.g., active connections, query duration, replication lag, disk I/O).  
Example docker-compose.yml (conceptual for postgres\_exporter):  
# services:  
# postgres:  
# # ... existing config  
# postgres\_exporter:  
# image: prometheuscommunity/postgres-exporter:latest  
# environment:  
# DATA\_SOURCE\_NAME: "postgresql://user:password@postgres:5432/main\_db?sslmode=disable"  
# ports:  
# - "9187:9187" # Exporter's metrics port  
# depends\_on:  
# postgres:  
# condition: service\_healthy  
  
And in observability/alloy-config.river:  
# prometheus.scrape "postgres" {  
# targets = [{"\_\_address\_\_" = "postgres\_exporter:9187"}]  
# forward\_to = [prometheus.remote\_write.default.receiver]  
# job = "postgres\_db"  
# }

Backup/Restore: PostgreSQL offers various backup strategies (e.g., pg\_dump for logical backups, filesystem-level backups, streaming replication for continuous archiving and point-in-time recovery).

Steps to Exercise (Conceptual/Discussion):

Discuss Database Monitoring in Grafana:

Explain how metrics from postgres\_exporter would be visualized in Grafana dashboards:

Connection Usage: See active connections, identify connection leaks.

Query Performance: Monitor long-running queries, identify slow queries.

Replication Status: (If replica set is configured) Monitor replication lag to ensure high availability.

Disk I/O: Track read/write operations to assess storage performance.

Relate these metrics to potential bottlenecks (e.g., high active connections could mean your FastAPI/Spark connection pool is too large or too small, leading to contention).

Discuss Backup and Restore Procedure for Airflow Metastore:

Importance: Emphasize that the Airflow metastore is critical. Its loss means losing all DAG run history, task states, and potentially DAG definitions.

Backup Strategy:

Logical Backup (pg\_dump): Explain pg\_dump -U user -d main\_db > backup.sql to create a SQL dump.

Physical Backup (Filesystem): Discuss copying the entire data directory (/var/lib/postgresql/data) after stopping the DB or using pg\_basebackup.

Continuous Archiving: For production, explain WAL (Write-Ahead Log) archiving for Point-In-Time Recovery (PITR).

Restore Scenario (DR & Runbooks Deep-Dive Addendum, Section 6.3.1):

Walk through the steps in the DR Runbook Example: Critical Database Restoration for PostgreSQL.

Highlight the importance of stopping applications (Airflow components), restoring the database, verifying integrity, and then restarting dependent applications in the correct order.

Emphasize the RPO (Recovery Point Objective) and RTO (Recovery Time Objective) implications for the Airflow metastore.

Verification (Conceptual):

Monitoring Insights: Ability to articulate how PostgreSQL metrics would provide insights into database health and performance.

DR Understanding: Clear explanation of PostgreSQL backup/restore mechanisms and their importance for the data platform's disaster recovery strategy, especially for the Airflow metastore, directly aligning with the DR & Runbooks addendum.

This concludes the guide for PostgreSQL.

Highlighting cAdvisor: Container Performance Monitoring

cAdvisor (Container Advisor) is a running daemon that collects, aggregates, processes, and exports information about running containers. It provides an essential layer of insight into the resource utilization and performance of your Dockerized services. When integrated with Grafana Alloy and Grafana, cAdvisor empowers you with real-time operational visibility into your local data platform.

This guide will demonstrate basic and advanced use cases of cAdvisor, 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, specifically emphasizing cAdvisor's role in the Observability section.

Basic Use Case: Monitoring Container CPU and Memory Usage

Objective: To demonstrate how cAdvisor collects fundamental resource metrics (CPU, memory) for all running containers and how these metrics are visualized in Grafana.

Role in Platform: Provide foundational visibility into individual container health and resource consumption, allowing for basic performance monitoring and identification of resource-hungry services.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root. This includes cAdvisor, grafana-alloy, and grafana.

Verify cAdvisor is running: Check Docker logs for the cAdvisor container to ensure it's healthy (docker compose logs cadvisor). It should be collecting metrics.

Verify Grafana is accessible: Go to http://localhost:3000 (initially anonymous or configure admin user).

Confirm Grafana Alloy is configured to scrape cAdvisor: Review your observability/alloy-config.river to ensure it includes a Prometheus scrape\_config for cAdvisor (usually on port 8080 or as configured in docker-compose.yml).  
Example docker-compose.yml snippet for cAdvisor and Grafana Alloy integration:  
# ... other services  
grafana-alloy:  
 # ...  
 depends\_on:  
 # ...  
 cAdvisor:  
 condition: service\_healthy  
cAdvisor:  
 image: gcr.io/cadvisor/cadvisor:v0.47.0 # Or latest stable  
 volumes:  
 - /:/rootfs:ro  
 - /var/run:/var/run:rw  
 - /sys:/sys:ro  
 - /var/lib/docker/:/var/lib/docker:ro  
 - /dev/disk/:/dev/disk:ro  
 privileged: true # Required for cAdvisor to access host information  
 ports:  
 - "8080:8080" # cAdvisor UI and metrics endpoint  
  
Example observability/alloy-config.river snippet (conceptual):  
# ... other components  
prometheus.scrape "cadvisor" {  
 targets = [{"\_\_address\_\_" = "cadvisor:8080"}] # 'cadvisor' is the service name in docker-compose  
 forward\_to = [prometheus.remote\_write.default.receiver]  
}  
# ...

Steps to Exercise:

Generate some activity:

Start python3 simulate\_data.py to generate traffic to FastAPI, which will then generate activity for Kafka and Spark. This will ensure your containers are actively working and consuming resources.

Access Grafana Dashboards:

Go to http://localhost:3000.

Navigate to a pre-provisioned dashboard designed for container monitoring (e.g., "Docker Container Overview" or "Host & Container Metrics"). If you don't have a specific one, you can import a community dashboard (e.g., "cAdvisor / Prometheus Host and Container" dashboard ID 14210) or create a new panel.

Look for panels showing "CPU Usage by Container" and "Memory Usage by Container."

Observe Metrics:

You should see graphs displaying the CPU and memory consumption for individual Docker containers (e.g., fastapi\_ingestor, kafka, spark, airflow-webserver, etc.).

The graphs will fluctuate based on the workload generated by simulate\_data.py. For example, the fastapi\_ingestor will show CPU usage when receiving requests, and spark will show higher CPU/memory during data processing.

Verification:

Grafana: Clear and continuously updating graphs for CPU and memory usage per container are visible, reflecting the activity of your data platform services.

cAdvisor UI (Optional): You can also directly access the cAdvisor UI at http://localhost:8080 (though grafana-alloy primarily scrapes its /metrics endpoint). Here, you can see detailed resource usage per container.

Advanced Use Case 1: Identifying Resource Bottlenecks and "Noisy Neighbors"

Objective: To demonstrate how cAdvisor metrics in Grafana can help identify which containers are consuming the most resources, potentially bottlenecking the host machine or impacting other services ("noisy neighbors").

Role in Platform: Facilitate resource optimization, capacity planning, and troubleshooting performance degradation due to resource contention.

Setup/Configuration:

Ensure Basic Use Case setup is complete.

Increase Workload: Temporarily increase the workload significantly to push resource limits.

Modify simulate\_data.py to have a very low DELAY\_SECONDS (e.g., 0.001 or remove it) to generate maximum traffic.

Run multiple instances of simulate\_data.py concurrently.

Alternatively, use Locust (locust -f locust\_fastapi\_ingestor.py) with a high number of users (e.g., 200) and a high spawn rate.

Steps to Exercise:

Apply Heavy Load: Start simulate\_data.py with very low delay, or use Locust as described.

Monitor Grafana: Go to your container monitoring dashboard in Grafana.

Focus on CPU and Memory utilization graphs that show metrics broken down by container name.

Observe the host machine's overall CPU/Memory usage (often available on the same dashboard from node\_exporter, which grafana-alloy might also collect).

Identify Bottlenecks:

Look for containers whose CPU usage consistently hits high percentages (e.g., 80-100%) or whose memory usage steadily climbs towards its allocated limit.

If the overall host CPU/memory is saturated, identify which individual containers are contributing most to that saturation. For example, spark containers typically consume high CPU/memory during processing, but if fastapi\_ingestor is unexpectedly high under normal load, it might indicate an issue.

Simulate Resource Constraint:

Edit your docker-compose.yml to intentionally limit resources for a container (e.g., fastapi\_ingestor).  
# In docker-compose.yml  
fastapi\_ingestor:  
 # ... existing config  
 deploy: # Use deploy for resource limits in Compose V3  
 resources:  
 limits:  
 cpus: '0.5' # Limit to 0.5 CPU core  
 memory: 128M # Limit memory to 128MB

Run docker compose up -d --build to apply changes.

Resume simulate\_data.py (or Locust) and observe the FastAPI container's performance and cAdvisor metrics in Grafana.

Verification:

Grafana: Under heavy load, you will clearly see which containers are consuming the most CPU and memory. When you apply resource limits, you'll observe the container's CPU usage capping at the defined limit, and if the workload exceeds that, its latency will increase, and errors might occur, demonstrating how limits are enforced and detectable via cAdvisor.

Performance Impact: The service with limited resources (e.g., FastAPI with cpus: '0.5') will show degraded performance (higher latency, lower RPS) even if the host has available capacity, highlighting the impact of container resource configuration.

Advanced Use Case 2: Custom Metric Collection (Conceptual via Sidecar)

Objective: While cAdvisor primarily collects system-level container metrics, it's often used in conjunction with Prometheus/Grafana Alloy for a holistic view. This use case demonstrates how you would conceptually expose application-specific metrics from your FastAPI application so they can be scraped by Grafana Alloy, complementing cAdvisor's data.

Role in Platform: Extend observability beyond infrastructure to application-level insights (e.g., API request counts, processing times, business metrics), critical for defining SLIs/SLOs.

Setup/Configuration:

Modify FastAPI to expose Prometheus metrics:

In your fastapi\_app/app/main.py, install prometheus\_fastapi\_instrumentator.

Instrument your FastAPI app to expose metrics on a dedicated endpoint (e.g., /metrics).

Example fastapi\_app/app/main.py snippet:# fastapi\_app/app/main.py (conceptual additions)  
from fastapi import FastAPI  
from prometheus\_fastapi\_instrumentator import Instrumentator  
# ... other imports and FastAPI app initialization ...  
  
app = FastAPI(title="Financial/Insurance Data Ingestor API")  
  
# Instrument FastAPI for Prometheus metrics  
Instrumentator().instrument(app).expose(app) # Exposes metrics on /metrics endpoint  
  
@app.get("/health", tags=["Monitoring"])  
async def health\_check():  
 return {"status": "healthy", "message": "Welcome to Financial/Insurance Data Ingestor API!"}  
  
# ... your existing FastAPI ingestion endpoints ...

Update docker-compose.yml: Ensure the FastAPI service's Prometheus endpoint is accessible to Grafana Alloy.  
Example docker-compose.yml snippet:  
# ...  
fastapi\_ingestor:  
 build: ./fastapi\_app  
 # ... other config  
 ports:  
 - "8000:8000" # Your main API port  
 # No need to expose metrics port if Grafana Alloy is in the same docker network

Update observability/alloy-config.river: Add a scrape config for FastAPI's metrics endpoint.  
Example observability/alloy-config.river snippet (conceptual):  
# ...  
prometheus.scrape "fastapi\_ingestor" {  
 targets = [{"\_\_address\_\_" = "fastapi\_ingestor:8000"}] # 'fastapi\_ingestor' is the service name  
 metrics\_path = "/metrics" # The path where FastAPI exposes its metrics  
 forward\_to = [prometheus.remote\_write.default.receiver]  
}  
# ...

Steps to Exercise:

Rebuild and restart affected services: docker compose up --build -d fastapi\_ingestor grafana-alloy (or all services).

Generate traffic: Run python3 simulate\_data.py to create API activity.

Access Grafana: Go to http://localhost:3000.

Create Custom Dashboard Panel:

Create a new dashboard or add a panel to an existing one.

Configure the panel to use your Prometheus data source (which Grafana Alloy sends metrics to).

Write a PromQL query for FastAPI metrics (e.g., http\_requests\_total{job="fastapi\_ingestor"}, http\_request\_duration\_seconds\_bucket{job="fastapi\_ingestor"}).

Observe Custom Metrics:

You should see graphs showing API request counts, latency histograms, etc., derived directly from your FastAPI application.

Verification:

Grafana: Your custom panels correctly display metrics from the FastAPI application, demonstrating that Grafana Alloy can scrape application-specific Prometheus endpoints, thus enriching your observability data beyond what cAdvisor provides.

API /metrics endpoint (Optional): You can directly access http://localhost:8000/metrics to see the raw Prometheus metrics exposed by FastAPI.

Advanced Use Case 3: Monitoring Cluster-wide Resource Utilization & Alerting

Objective: To use cAdvisor and Grafana to get an aggregated view of resource utilization across the entire Docker Compose environment (simulating a cluster) and set up basic alerts for high resource consumption.

Role in Platform: Provide a holistic view of the local environment's health, enable proactive alerting on potential resource exhaustion before it impacts critical data pipelines.

Setup/Configuration:

Ensure all Advanced Track services are running and generating data.

Confirm grafana-alloy is collecting from all cAdvisor targets (as in Basic Use Case).

Configure Grafana Alerting:

In Grafana, go to "Alerting" -> "Alert Rules".

Create a new alert rule.

Example Alert Rule (conceptual):# Alert for high overall CPU usage on the Docker host (simulated cluster)  
Rule type: Grafana managed alert  
Data source: Prometheus (Grafana Alloy target)  
Query (PromQL): sum(rate(container\_cpu\_usage\_seconds\_total{container!=""}[1m])) by (instance) / sum(machine\_cpu\_cores) \* 100  
 # This query calculates the total CPU usage across all containers on a given instance (host)  
 # and divides by total CPU cores to get percentage.  
 # Adjust `container!=""` to filter containers if needed.  
Condition: WHEN last() OF A IS ABOVE 80  
For: 5m # Wait for 5 minutes of sustained high usage  
Labels:  
 severity: warning  
 component: infrastructure  
 service: docker\_host  
Annotations:  
 summary: "High CPU usage on Docker host"  
 description: "The Docker host (simulating your data platform cluster) CPU utilization has been above 80% for 5 minutes. This may impact overall performance and lead to degraded service."  
 remediation: "1. Identify top CPU-consuming containers via Grafana 'Docker Container Overview' dashboard. 2. Consider optimizing Spark jobs, reducing ingestion rate, or allocating more CPU to your Docker Desktop/VM."  
 dashboard\_link: "http://localhost:3000/d/<your\_container\_dashboard\_uid>"

Steps to Exercise:

Generate Sustained High Load:

Use Locust with a very high number of users (e.g., 500) and a high spawn rate, running for a sustained period (e.g., 10-15 minutes). This will push your local machine's resources to their limits.

Alternatively, run multiple simulate\_data.py instances with DELAY\_SECONDS=0.

Monitor Grafana:

Observe the "Host & Container Metrics" dashboard, looking at overall CPU and memory usage of the docker\_host (or whichever instance name cAdvisor reports).

Trigger Alert: Allow the load to run long enough for the CPU (or memory) to consistently stay above the 80% threshold for 5 minutes.

Observe Alert:

In Grafana, navigate to "Alerting" -> "Alert Rules".

The alert rule you created should transition to "Firing" status.

You might see a notification within Grafana or via configured notification channels (if set up, e.g., email, Slack, which is typically outside Docker Compose scope for local setup).

Verification:

Grafana: The alert rule changes status to "Firing", and its history shows the alert state changes. The dashboard metrics clearly show the sustained high resource utilization that triggered the alert. This demonstrates how cAdvisor metrics, combined with Grafana's alerting capabilities, enable proactive monitoring and notification for potential system health issues at a cluster-wide level.

This concludes the guide for cAdvisor.

Deep Dive: ML Tooling in the Platform

This document delves into how your enterprise-ready data platform provides a robust foundation for Machine Learning (ML) workloads. A successful ML initiative relies heavily on high-quality, accessible data and efficient processes for feature engineering, model training, and inference. Your platform, with its integrated components, is designed to support the entire ML lifecycle, from raw data ingestion to delivering predictions.

1. Core Concepts: Data Readiness for ML

At its heart, any ML process requires structured, clean, and consistent data. Your data platform facilitates this through:

Curated Data Lakehouse (Delta Lake on MinIO): This is the primary storage layer for ML. Data is transformed, cleaned, and enriched here, providing a reliable and versioned source of truth for features and labels. The ACID properties of Delta Lake ensure data quality and reproducibility for ML experiments.

Feature Engineering with Apache Spark: Spark's distributed processing capabilities are ideal for transforming raw data into features at scale, handling large volumes and complex logic.

Operational Databases (PostgreSQL, MongoDB): These can serve specific roles, such as storing lookup tables for real-time feature enrichment, or even acting as a simple "feature store" for frequently accessed, low-latency features.

Data Lineage (Spline, OpenMetadata): Understanding where your ML features come from and how they were transformed is critical for debugging, reproducibility, and compliance. OpenMetadata, enriched by Spline, provides this transparency.

Observability (OpenTelemetry, Grafana Alloy, Grafana): Monitoring the health and performance of your ML pipelines, including data quality, model inference latency, and drift, is crucial for MLOps.

2. ML Lifecycle Stages in the Platform

Let's map the key stages of an ML lifecycle to how they are supported by your platform components:

Data Ingestion (FastAPI, Kafka): Raw data (e.g., transaction events) enters the platform via FastAPI, is buffered by Kafka, and lands in the raw data lake (MinIO/Delta Lake). This forms the basis for feature extraction.

Feature Engineering (Apache Spark, Delta Lake): Raw data is transformed and aggregated into features. Spark processes these large datasets, and the results are stored in curated Delta Lake tables.

Model Training (Apache Spark MLlib / External ML Frameworks): While a full ML training cluster setup is beyond the scope of this local environment, Spark MLlib can perform distributed training. For more complex models (e.g., deep learning), data can be exported from Delta Lake to specialized training environments.

Model Inference (Apache Spark Structured Streaming / FastAPI + PostgreSQL/MongoDB):

Batch/Streaming Inference: Spark Structured Streaming can load trained models and apply them to new incoming data streams in near real-time.

Real-time Inference (APIs): A separate FastAPI service (or an extension of the existing one) can serve real-time predictions, often backed by pre-computed features in a database or a dedicated feature store.

Model Monitoring (OpenTelemetry, Grafana, OpenMetadata):

Operational Monitoring: OpenTelemetry and Grafana track the performance and resource utilization of inference services and feature engineering pipelines.

Data/Model Drift: Data quality checks on features and output predictions (tracked via OpenMetadata profiling) can help detect data drift or concept drift.

3. Interactive How-Tos: Leveraging ML Tooling

Let's explore some practical examples of how ML workflows can be built using your platform.

Basic Use Case: Feature Engineering with Spark

Objective: To demonstrate how Apache Spark can be used to perform basic feature engineering, transforming raw financial transactions into aggregated features suitable for an ML model (e.g., daily transaction counts per account, average amount).

Role in Platform: Create derived features from raw or intermediate data, making it ready for model training and inference.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d. This includes spark and minio.

Ensure raw-data-bucket/financial\_data\_delta is populated: Your streaming\_consumer.py job should have written some data to this path.

Create a PySpark script for feature engineering: In pyspark\_jobs/, create feature\_engineering\_job.py.  
# pyspark\_jobs/feature\_engineering\_job.py  
import sys  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import col, count, sum, avg, to\_date, lit, current\_timestamp  
from pyspark.sql.types import StructType, StringType, FloatType, TimestampType  
from delta.tables import DeltaTable # Ensure DeltaTable import if writing to Delta  
  
def create\_spark\_session(app\_name):  
 """Helper function to create a SparkSession with Delta Lake packages."""  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "io.delta:delta-core\_2.12:2.4.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if len(sys.argv) != 3:  
 print("Usage: feature\_engineering\_job.py <input\_delta\_path> <output\_delta\_path>")  
 sys.exit(-1)  
  
 input\_delta\_path = sys.argv[1]  
 output\_delta\_path = sys.argv[2]  
  
 spark = create\_spark\_session("FinancialFeatureEngineering")  
 spark.sparkContext.setLogLevel("WARN")  
  
 print(f"Reading raw financial data from: {input\_delta\_path}")  
 try:  
 df\_raw = spark.read.format("delta").load(input\_delta\_path)  
 df\_raw.printSchema()  
 df\_raw.show(5, truncate=False)  
 except Exception as e:  
 print(f"Error reading input Delta Lake: {e}. Ensure data exists at {input\_delta\_path}")  
 spark.stop()  
 sys.exit(-1)  
  
 print("Performing feature engineering: daily aggregates per account...")  
 # Convert timestamp string to actual timestamp, then to date  
 df\_features = df\_raw.withColumn("transaction\_date", to\_date(col("timestamp"), "yyyy-MM-dd'T'HH:mm:ss'Z'")) \  
 .groupBy("account\_id", "transaction\_date") \  
 .agg(  
 count(col("transaction\_id")).alias("daily\_transaction\_count"),  
 sum(col("amount")).alias("daily\_total\_amount"),  
 avg(col("amount")).alias("daily\_average\_amount")  
 ) \  
 .withColumn("feature\_created\_at", current\_timestamp())  
  
 print("Schema of engineered features:")  
 df\_features.printSchema()  
 df\_features.show(5, truncate=False)  
  
 # Write the engineered features to a new curated Delta Lake table  
 print(f"Writing engineered features to: {output\_delta\_path}")  
 df\_features.write.format("delta") \  
 .mode("overwrite") \  
 .option("overwriteSchema", "true") \  
 .save(output\_delta\_path)  
 print("Feature engineering job completed.")  
  
 spark.stop()

Steps to Exercise:

Ensure raw-data-bucket/financial\_data\_delta has data. If not, run python3 simulate\_data.py and then trigger streaming\_consumer.py from your financial\_data\_lake\_pipeline Airflow DAG.

Submit the Spark Feature Engineering Job:  
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/feature\_engineering\_job.py \  
 s3a://raw-data-bucket/financial\_data\_delta \  
 s3a://curated-data-bucket/financial\_features\_daily\_account

Monitor Spark Job: Observe the console output for Spark logs, confirming the reading, aggregation, and writing process.

Verify Features in MinIO: Access the MinIO Console (http://localhost:9001). Navigate to curated-data-bucket/financial\_features\_daily\_account/. You should see new Delta Lake files.

Query Engineered Features via Spark SQL:  
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 \* FROM delta.\`s3a://curated-data-bucket/financial\_features\_daily\_account\` LIMIT 10;"

Verification:

Spark Job Completion: The Spark job executes successfully, indicated by the console output.

MinIO Contents: The financial\_features\_daily\_account Delta Lake table is created with new .parquet files and \_delta\_log.

Spark SQL Query: The query results show aggregated data, confirming that the raw transaction data has been transformed into daily account-level features.

Advanced Use Case 1: Model Training with PySpark MLlib (Conceptual)

Objective: To conceptually demonstrate how a Spark job could train a simple ML model (e.g., Logistic Regression for fraud detection) using PySpark MLlib on the curated features.

Role in Platform: Perform large-scale, distributed model training directly on your data lakehouse data, leveraging Spark's optimized algorithms.

Setup/Configuration:

Ensure curated features are available: The financial\_features\_daily\_account Delta Lake table (from the previous Basic Use Case) should be populated.

Create a PySpark script for model training: In pyspark\_jobs/, create model\_training\_job.py.  
# pyspark\_jobs/model\_training\_job.py  
import sys  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import col, lit, rand, monotonically\_increasing\_id  
from pyspark.ml.feature import VectorAssembler, StandardScaler # For feature preprocessing  
from pyspark.ml.classification import LogisticRegression # Example ML algorithm  
from pyspark.ml import Pipeline  
from delta.tables import DeltaTable # Ensure DeltaTable import  
  
def create\_spark\_session(app\_name):  
 """Helper function to create a SparkSession with Delta Lake packages."""  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "io.delta:delta-core\_2.12:2.4.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if len(sys.argv) != 3:  
 print("Usage: model\_training\_job.py <input\_features\_path> <model\_output\_path>")  
 sys.exit(-1)  
  
 input\_features\_path = sys.argv[1]  
 model\_output\_path = sys.argv[2]  
  
 spark = create\_spark\_session("MLModelTraining")  
 spark.sparkContext.setLogLevel("WARN")  
  
 print(f"Reading engineered features from: {input\_features\_path}")  
 try:  
 # For demonstration, let's assume a 'label' column exists for training.  
 # In a real scenario, this 'label' would come from historical labeled data.  
 df\_features = spark.read.format("delta").load(input\_features\_path) \  
 .withColumn("id", monotonically\_increasing\_id()) # Add unique ID for partitioning  
  
 # Simulate a 'label' column (e.g., 'is\_fraud' or 'high\_risk') for demo purposes  
 # In a real scenario, this label would come from your actual labeled dataset.  
 df\_labeled\_features = df\_features.withColumn("label", when(col("daily\_total\_amount") > 5000 and rand() < 0.2, 1.0).otherwise(0.0))  
  
 df\_labeled\_features.printSchema()  
 df\_labeled\_features.show(5, truncate=False)  
  
 except Exception as e:  
 print(f"Error reading input features Delta Lake: {e}. Ensure data exists at {input\_features\_path}")  
 spark.stop()  
 sys.exit(-1)  
  
 # Prepare features for MLlib  
 # Assume numerical features are 'daily\_transaction\_count', 'daily\_total\_amount', 'daily\_average\_amount'  
 feature\_columns = ["daily\_transaction\_count", "daily\_total\_amount", "daily\_average\_amount"]  
  
 # Assemble features into a vector  
 assembler = VectorAssembler(inputCols=feature\_columns, outputCol="raw\_features")  
  
 # Scale features (optional, but good practice for many ML algorithms)  
 scaler = StandardScaler(inputCol="raw\_features", outputCol="features", withStd=True, withMean=False)  
  
 # Train a Logistic Regression model (example classifier)  
 lr = LogisticRegression(labelCol="label", featuresCol="features", maxIter=10)  
  
 # Create a Pipeline to chain preprocessing and model training  
 pipeline = Pipeline(stages=[assembler, scaler, lr])  
  
 print("Training ML model...")  
 # Split data into training and test sets (conceptual)  
 (trainingData, testData) = df\_labeled\_features.randomSplit([0.8, 0.2], seed=42)  
  
 # Fit the model  
 model = pipeline.fit(trainingData)  
 print("Model training completed.")  
  
 # Evaluate the model (conceptual)  
 # predictions = model.transform(testData)  
 # evaluator = BinaryClassificationEvaluator(labelCol="label")  
 # auc = evaluator.evaluate(predictions)  
 # print(f"Area under ROC: {auc}")  
  
 # Save the trained model  
 print(f"Saving trained model to: {model\_output\_path}")  
 model.save(model\_output\_path)  
 print("Model saved.")  
  
 spark.stop()

Steps to Exercise (Conceptual):

Ensure financial\_features\_daily\_account is populated.

Submit the Spark Model Training Job:  
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/model\_training\_job.py \  
 s3a://curated-data-bucket/financial\_features\_daily\_account \  
 s3a://models-bucket/financial\_fraud\_model

Monitor Spark Job: Observe the console output for messages indicating model training and saving.

Verify Model in MinIO: Check http://localhost:9001 in the models-bucket/financial\_fraud\_model/ path. You should see Spark MLlib's serialized model files.

Verification:

Spark Job Completion: The job runs and indicates "Model saved."

MinIO Contents: The specified MinIO path contains the serialized Spark MLlib model, demonstrating the platform's capability to store trained models.

Advanced Use Case 2: Model Inference with Spark Structured Streaming

Objective: To demonstrate how a trained ML model (from MinIO) can be loaded by a Spark Structured Streaming job and applied to new incoming data (e.g., from Kafka or a raw Delta table) for real-time or near-real-time inference.

Role in Platform: Deploy ML models to production for continuous scoring and anomaly detection on live data streams.

Setup/Configuration:

Ensure trained model is available: s3a://models-bucket/financial\_fraud\_model should be populated (from previous use case).

Ensure raw data source is active: raw-data-bucket/financial\_data\_delta should be continuously receiving data from Kafka via streaming\_consumer.py or you can adjust to read directly from Kafka if preferred.

Create a PySpark script for streaming inference: In pyspark\_jobs/, create streaming\_inference\_job.py.  
# pyspark\_jobs/streaming\_inference\_job.py  
import sys  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import col, from\_json, current\_timestamp  
from pyspark.sql.types import StructType, StringType, FloatType, TimestampType, BooleanType  
from pyspark.ml import PipelineModel # To load the saved pipeline model  
from pyspark.ml.feature import VectorAssembler, StandardScaler # Necessary for schema of loaded model  
from pyspark.ml.classification import LogisticRegression # Necessary for schema of loaded model  
  
def create\_spark\_session(app\_name):  
 """Helper function to create a SparkSession with Delta Lake and Kafka packages."""  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "io.delta:delta-core\_2.12:2.4.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if len(sys.argv) != 4:  
 print("Usage: streaming\_inference\_job.py <input\_delta\_path> <model\_path> <output\_delta\_path>")  
 sys.exit(-1)  
  
 input\_delta\_path = sys.argv[1]  
 model\_path = sys.argv[2]  
 output\_delta\_path = sys.argv[3]  
  
 spark = create\_spark\_session("StreamingModelInference")  
 spark.sparkContext.setLogLevel("WARN")  
  
 print(f"Loading trained model from: {model\_path}")  
 try:  
 loaded\_model = PipelineModel.load(model\_path)  
 print("Model loaded successfully.")  
 except Exception as e:  
 print(f"Error loading model from {model\_path}: {e}. Ensure model exists and is valid.")  
 spark.stop()  
 sys.exit(-1)  
  
 # Define schema for the incoming streaming data (should match source raw data)  
 data\_schema = StructType() \  
 .add("transaction\_id", StringType(), True) \  
 .add("timestamp", StringType(), True) \  
 .add("account\_id", StringType(), True) \  
 .add("amount", FloatType(), True) \  
 .add("currency", StringType(), True) \  
 .add("transaction\_type", StringType(), True) \  
 .add("merchant\_id", StringType(), True) \  
 .add("category", StringType(), True) \  
 .add("is\_flagged", BooleanType(), True)  
  
  
 print(f"Reading streaming data from: {input\_delta\_path}")  
 # Read from Delta Lake as a streaming DataFrame  
 # For a truly 'real-time' demo, this could read directly from Kafka  
 streaming\_df = (spark.readStream  
 .format("delta")  
 .option("startingOffsets", "latest") # Start consuming new data  
 .load(input\_delta\_path))  
  
 # Basic feature preparation (must match preprocessing in training pipeline)  
 # The model expects a 'features' column as input, created by VectorAssembler/StandardScaler  
 # In a real scenario, you would extract/compute these features from the incoming data  
 # based on the model's requirements. For this simple demo, we'll recreate the feature vector  
 # structure expected by the loaded model.  
  
 feature\_columns = ["amount"] # Simplified feature for quick demo.  
 # In real life, this would be daily\_transaction\_count etc.  
 # and would require complex stateful processing or lookup for live data.  
  
 # To make this example runnable with simple raw data, we will just pass 'amount' as the feature.  
 # In a true scenario, you'd apply the same feature engineering logic as the training job.  
 # This requires more complex stateful streaming or a feature store.  
 # For now, let's just make sure the 'features' vector is created from available data.  
  
 # If your model was trained on 'daily\_transaction\_count', 'daily\_total\_amount', 'daily\_average\_amount',  
 # you'd need to compute these in a streaming fashion or join with a feature store.  
 # For simplicity of this demo, we use a single numerical feature present in raw data.  
  
 # IMPORTANT: The schema of the DataFrame passed to `model.transform` must exactly  
 # match the input schema expected by the `PipelineModel`.  
 # This means the 'raw\_features' and 'features' columns created by the assembler and scaler  
 # in the training pipeline must be recreated here.  
 # For a demo, let's simplify to directly use 'amount' as the input feature for now,  
 # assuming the model can handle it, or adjust the model training to use simple features.  
  
 # Let's assume the model was trained on a single feature 'amount' for simplicity of streaming demo  
 # (This is a simplification for a real-world complex model)  
 assembler = VectorAssembler(inputCols=["amount"], outputCol="features") # Adjust to actual features used in training  
  
 df\_with\_features = assembler.transform(streaming\_df)  
  
 print("Applying model inference to streaming data...")  
 # Apply the loaded model to the streaming data  
 predictions\_df = loaded\_model.transform(df\_with\_features) \  
 .withColumn("inference\_timestamp", current\_timestamp()) \  
 .withColumn("prediction\_score", col("probability")[1]) # Get score for positive class  
  
  
 # Select relevant columns for the output  
 output\_df = predictions\_df.select(  
 col("transaction\_id"),  
 col("timestamp"),  
 col("account\_id"),  
 col("amount"),  
 col("currency"),  
 col("transaction\_type"),  
 col("prediction").alias("is\_fraud\_prediction"), # Binary prediction (0 or 1)  
 col("prediction\_score"), # Probability score for fraud  
 col("inference\_timestamp")  
 )  
  
 # Define checkpoint location for fault tolerance  
 checkpoint\_location = f"{output\_delta\_path}/\_checkpoints"  
  
 # Write predictions to a new Delta Lake table  
 query = (output\_df.writeStream  
 .format("delta")  
 .outputMode("append") # Append new predictions  
 .option("checkpointLocation", checkpoint\_location)  
 .option("mergeSchema", "true") # Enable schema evolution  
 .start(output\_delta\_path))  
  
 print(f"Spark Structured Streaming inference job started, writing predictions to: {output\_delta\_path}")  
 print(f"Checkpoint location: {checkpoint\_location}")  
  
 query.awaitTermination()  
 spark.stop()

Steps to Exercise:

Ensure financial\_fraud\_model is in MinIO.

Ensure raw-data-bucket/financial\_data\_delta is receiving data. (Run simulate\_data.py and the streaming\_consumer.py or the financial\_data\_lake\_pipeline DAG).

Submit the Spark Streaming Inference Job:  
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/streaming\_inference\_job.py \  
 s3a://raw-data-bucket/financial\_data\_delta \  
 s3a://models-bucket/financial\_fraud\_model \  
 s3a://predictions-bucket/financial\_fraud\_predictions\_stream

Monitor Spark Job: Observe the console output for messages confirming model loading and stream processing.

Verify Predictions in MinIO: Access http://localhost:9001. Navigate to predictions-bucket/financial\_fraud\_predictions\_stream/. You should see new Delta Lake files.

Query Predictions via Spark SQL:  
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 transaction\_id, amount, is\_fraud\_prediction, prediction\_score, inference\_timestamp FROM delta.\`s3a://predictions-bucket/financial\_fraud\_predictions\_stream\` LIMIT 10;"

Verification:

Spark Streaming Job: The job runs continuously, processing new data and applying the model.

MinIO Contents: The financial\_fraud\_predictions\_stream Delta Lake table is continuously updated with predictions.

Spark SQL Query: The query results show the original transaction data augmented with is\_fraud\_prediction and prediction\_score, demonstrating successful real-time model inference.

Advanced Use Case 3: Simple Feature Store Integration (Conceptual with PostgreSQL)

Objective: To conceptually demonstrate how PostgreSQL can act as a lightweight "feature store" for pre-computed, low-latency features that can be quickly retrieved by real-time inference services (e.g., another FastAPI application).

Role in Platform: Serve pre-computed, frequently accessed features for low-latency online inference, decoupling feature computation from real-time serving.

Setup/Configuration (Conceptual Discussion):

Feature Computation (Spark Job): A batch Spark job (similar to feature\_engineering\_job.py) would compute features and then write them to PostgreSQL instead of (or in addition to) Delta Lake.

Example: Daily aggregated features per account\_id could be stored in a PostgreSQL table.

PostgreSQL Table for Features:  
CREATE TABLE IF NOT EXISTS daily\_account\_features (  
 account\_id VARCHAR(255) PRIMARY KEY,  
 feature\_date DATE,  
 daily\_transaction\_count INTEGER,  
 daily\_total\_amount NUMERIC,  
 daily\_average\_amount NUMERIC,  
 last\_updated TIMESTAMP DEFAULT CURRENT\_TIMESTAMP  
);  
  
Spark would MERGE INTO this table for upserts.

Real-time Inference Service (FastAPI): A new or existing FastAPI endpoint would receive an account\_id and query PostgreSQL for its latest features.  
Example FastAPI endpoint (conceptual):  
# fastapi\_app/app/main.py (conceptual addition for real-time inference)  
# ... other imports ...  
import psycopg2 # For PostgreSQL connection  
from pydantic import BaseModel  
from typing import Dict, Any  
  
# Assuming pg\_conn is initialized on startup as in Starter Track example  
# ...  
  
class AccountFeaturesRequest(BaseModel):  
 account\_id: str  
  
class AccountFeaturesResponse(BaseModel):  
 account\_id: str  
 features: Dict[str, Any]  
 status: str  
  
@app.post("/get-account-features/", response\_model=AccountFeaturesResponse, tags=["ML Features"])  
async def get\_account\_features(request: AccountFeaturesRequest):  
 if not pg\_conn:  
 raise HTTPException(status\_code=503, detail="Database connection not available.")  
  
 try:  
 with pg\_conn.cursor() as cur:  
 cur.execute(  
 """  
 SELECT daily\_transaction\_count, daily\_total\_amount, daily\_average\_amount, feature\_date  
 FROM daily\_account\_features  
 WHERE account\_id = %s  
 ORDER BY feature\_date DESC LIMIT 1;  
 """,  
 (request.account\_id,)  
 )  
 result = cur.fetchone()  
  
 if result:  
 features = {  
 "daily\_transaction\_count": result[0],  
 "daily\_total\_amount": float(result[1]), # Convert Decimal to float  
 "daily\_average\_amount": float(result[2]),  
 "feature\_date": result[3].isoformat()  
 }  
 return AccountFeaturesResponse(account\_id=request.account\_id, features=features, status="found")  
 else:  
 return AccountFeaturesResponse(account\_id=request.account\_id, features={}, status="not\_found")  
 except Exception as e:  
 print(f"Error fetching features from PostgreSQL: {e}")  
 raise HTTPException(status\_code=500, detail=f"Failed to fetch features: {e}")

Steps to Exercise (Conceptual/Discussion):

Discuss Feature Generation and Load to PG:

Explain how the feature\_engineering\_job.py could be modified to write its output (df\_features) to the daily\_account\_features PostgreSQL table using Spark's JDBC connector (similar to how batch\_transformations.py reads from PG).

Discuss Real-time Feature Lookup:

Explain how an API call to /get-account-features/ with an account\_id would trigger a quick lookup in the PostgreSQL daily\_account\_features table.

Emphasize the low latency for retrieving pre-computed features, ideal for augmenting real-time inference requests.

Explain the "Online" vs. "Offline" Feature Store Concept:

Online Feature Store (PostgreSQL in this case): Optimized for low-latency reads during online inference.

Offline Feature Store (Delta Lake in MinIO): Optimized for large-scale batch reads during model training and batch inference.

The platform supports both paradigms.

Verification (Conceptual):

Architectural Understanding: Demonstrate how the platform can support a simple feature store pattern using existing components, providing a clear path for more sophisticated MLOps patterns. This highlights the flexibility of the platform to integrate different data access patterns for ML.

4. Integrations and Future Enhancements for MLOps

While your current platform provides core ML capabilities, a full MLOps (Machine Learning Operations) setup would involve further integrations:

MLflow: For tracking ML experiments (parameters, metrics, models), managing model versions, and deploying models.

Kubeflow / Airflow for Orchestration: Complex ML pipelines (data prep, training, evaluation, deployment) can be orchestrated by Airflow. For Kubernetes-native environments, Kubeflow Pipelines would be an option.

Model Registry: A centralized repository for managing the lifecycle of ML models, from development to production deployment.

Monitoring Model Drift: Extending Grafana dashboards with custom metrics and alerts to detect concept drift (model performance degrades due to changes in data distribution) or data drift (changes in input feature distributions). OpenMetadata's profiling capabilities are a good starting point for data quality monitoring that feeds into drift detection.

Automated Retraining: Setting up Airflow DAGs to automatically trigger model retraining when performance degrades or data drift is detected.

CI/CD for ML Models: Applying the existing CI/CD principles (from IaC & CI/CD Recipes addendum) to ML models, ensuring reproducible builds and automated deployment of new model versions.

Your current platform provides the essential data infrastructure (ingestion, storage, processing, basic observability) that is prerequisite for implementing these advanced MLOps practices.

This concludes the deep dive into ML Tooling in your platform.

Highlighting Grafana Alloy: Unified Telemetry Collection

Grafana Alloy is an OpenTelemetry Collector distribution that acts as a powerful and highly configurable agent for collecting, processing, and exporting telemetry data – metrics, logs, and traces – from your data platform services. It is the central nervous system for your observability stack, gathering data from various sources (like cAdvisor, FastAPI, Spark) and forwarding it to monitoring tools like Grafana.

This guide will demonstrate basic and advanced use cases of Grafana Alloy, leveraging your Advanced Track local environment setup and its integration with other observability components.

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, specifically emphasizing Grafana Alloy's role in the Observability section.

Basic Use Case: Collecting Metrics from Prometheus Endpoints

Objective: To demonstrate how Grafana Alloy scrapes Prometheus-compatible metrics endpoints (like those exposed by cAdvisor and FastAPI) and forwards them to Grafana for visualization.

Role in Platform: Act as the primary metrics collector, standardizing the ingestion of operational data from diverse services into your monitoring system.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root. This includes cAdvisor, fastapi\_ingestor (with Prometheus instrumentation), grafana-alloy, and grafana.

Verify Grafana Alloy configuration: Review your observability/alloy-config.river file. It should contain prometheus.scrape blocks for various services.  
Example observability/alloy-config.river snippets (conceptual):  
# observability/alloy-config.river  
# ... other components ...  
  
prometheus.remote\_write "default" {  
 # This receiver sends all scraped metrics to Grafana (acting as a Prometheus storage)  
 url = "http://grafana:9090/api/prom/push" # Grafana's Prometheus-compatible remote write endpoint  
}  
  
# Scrape metrics from cAdvisor (container metrics)  
prometheus.scrape "cadvisor" {  
 targets = [{"\_\_address\_\_" = "cadvisor:8080"}] # 'cadvisor' is the service name in docker-compose  
 forward\_to = [prometheus.remote\_write.default.receiver]  
 job = "cadvisor" # Label for the metrics  
}  
  
# Scrape metrics from FastAPI ingestor (application metrics)  
prometheus.scrape "fastapi\_ingestor" {  
 targets = [{"\_\_address\_\_" = "fastapi\_ingestor:8000"}] # 'fastapi\_ingestor' is the service name  
 metrics\_path = "/metrics" # The path where FastAPI exposes its metrics  
 forward\_to = [prometheus.remote\_write.default.receiver]  
 job = "fastapi\_ingestor" # Label for the metrics  
}  
  
# ... and potentially scrape Spark JMX exporter, Kafka JMX exporter, etc.

Ensure Grafana is accessible: http://localhost:3000.

Generate activity: Run python3 simulate\_data.py to create traffic to FastAPI.

Steps to Exercise:

Observe Grafana Alloy Logs:

Open a terminal and watch the logs of the grafana-alloy container:  
docker compose logs -f grafana-alloy

You should see messages indicating that Grafana Alloy is actively "scraping" or "collecting" metrics from cadvisor:8080/metrics and fastapi\_ingestor:8000/metrics.

Access Grafana Dashboards:

Go to http://localhost:3000.

Navigate to a dashboard that displays container metrics (e.g., "Docker Container Overview") and another that shows FastAPI application metrics (e.g., "Health Dashboard" or a custom one).

Look for panels showing CPU, memory usage for containers (from cAdvisor via Alloy) and API request rates, latency (from FastAPI via Alloy).

Verification:

Grafana Alloy Logs: Confirm that Alloy logs show successful scraping attempts and no connection errors to the target endpoints.

Grafana Dashboards: Metrics from both cAdvisor (container resources) and FastAPI (API performance) are populating correctly in Grafana, demonstrating that Grafana Alloy is successfully collecting and forwarding this data.

Advanced Use Case 1: Processing and Relabeling Metrics

Objective: To demonstrate how Grafana Alloy can process and transform metrics before forwarding them, including relabeling, filtering, and adding new attributes. This is crucial for standardizing metric names, adding useful metadata, and reducing noise.

Role in Platform: Cleanse, enrich, and standardize telemetry data, ensuring consistency and usability for monitoring and alerting.

Setup/Configuration:

Ensure Basic Use Case is running.

Modify observability/alloy-config.river: Add a relabel block to an existing prometheus.scrape component. Let's relabel the instance label for FastAPI metrics to be more descriptive, and add a static environment label.  
Example observability/alloy-config.river snippet (modification to fastapi\_ingestor scrape):  
# observability/alloy-config.river  
# ...  
prometheus.scrape "fastapi\_ingestor" {  
 targets = [{"\_\_address\_\_" = "fastapi\_ingestor:8000"}]  
 metrics\_path = "/metrics"  
 forward\_to = [prometheus.remote\_write.default.receiver]  
 job = "fastapi\_ingestor"  
  
 # Relabeling example:  
 # Change 'instance' label (which might be 'fastapi\_ingestor:8000')  
 # to just the service name and add an environment label.  
 relabel\_configs {  
 source\_labels = ["\_\_address\_\_"]  
 regex = "(.\*):.\*"  
 target\_label = "instance"  
 replacement = "$1" # Keep only the service name part (e.g., "fastapi\_ingestor")  
 }  
 relabel\_configs {  
 source\_labels = [] # Empty means apply to all metrics  
 target\_label = "environment"  
 replacement = "local\_dev" # Add a static 'environment' label  
 }  
 # Example: drop metrics starting with 'go\_'  
 # relabel\_configs {  
 # source\_labels = ["\_\_name\_\_"]  
 # regex = "go\_.\*"  
 # action = "drop"  
 # }  
}  
# ...

Steps to Exercise:

Restart Grafana Alloy: docker compose restart grafana-alloy to load the new configuration.

Generate traffic: Run python3 simulate\_data.py.

Inspect Metrics in Grafana:

Go to http://localhost:3000.

Open the "Explore" view in Grafana and select your Prometheus data source.

Enter a PromQL query for a FastAPI metric, for example: http\_requests\_total.

Observe: Instead of instance="fastapi\_ingestor:8000", you should now see instance="fastapi\_ingestor" and a new label environment="local\_dev". This confirms the relabeling.

If you applied the drop rule, verify that metrics like go\_goroutines are no longer present.

Verification:

Grafana Metrics Explorer: Queries show the modified labels (e.g., instance="fastapi\_ingestor", environment="local\_dev"), confirming that Grafana Alloy successfully applied the relabeling rules.

Logs: Grafana Alloy logs might show "processing" or "relabeling" messages if debug logging is enabled.

Advanced Use Case 2: Aggregating Metrics for Service-Level Objectives (SLOs)

Objective: To demonstrate how Grafana Alloy can preprocess and aggregate metrics from multiple instances of a service (or different services) before sending them to the monitoring backend. This is vital for calculating service-level metrics needed for SLOs (e.g., total requests across all replicas, aggregated latency).

Role in Platform: Enable the calculation of high-level service health indicators and facilitate alerting on SLO violations, crucial for data platform reliability.

Setup/Configuration:

Simulate multiple FastAPI instances (conceptual in docker-compose.yml): While we only have one fastapi\_ingestor in the standard setup, you can conceptualize scaling it or having another similar service. For this demo, we'll demonstrate aggregation across multiple jobs or instances that might appear from different sources.

Modify observability/alloy-config.river to aggregate: Use prometheus.rule component within Alloy to define recording rules.  
Example observability/alloy-config.river snippet (conceptual, assumes fastapi\_ingestor and another ingestor\_replica job):  
# observability/alloy-config.river  
# ...  
prometheus.scrape "fastapi\_ingestor" {  
 targets = [{"\_\_address\_\_" = "fastapi\_ingestor:8000"}]  
 metrics\_path = "/metrics"  
 forward\_to = [prometheus.remote\_write.default.receiver]  
 job = "fastapi\_ingestor" # Job label for this instance  
}  
  
# CONCEPTUAL: If you had another FastAPI replica or a similar service  
# prometheus.scrape "ingestor\_replica" {  
# targets = [{"\_\_address\_\_" = "another\_ingestor:8000"}]  
# metrics\_path = "/metrics"  
# forward\_to = [prometheus.remote\_write.default.receiver]  
# job = "ingestor\_replica" # Different job label  
# }  
  
# Define a recording rule to aggregate total requests across all ingestor instances/jobs  
prometheus.rule "ingestor\_total\_requests\_sum" {  
 label\_match {  
 name = "job"  
 value = "(fastapi\_ingestor|ingestor\_replica)" # Matches both original and conceptual replica  
 }  
 rules = [  
 {  
 record = "ingestor\_api\_total\_requests\_sum" # New aggregated metric name  
 expr = "sum by (le, path) (rate(http\_requests\_total[1m]))" # Sums rate over all matched jobs/instances  
 }  
 ]  
 forward\_to = [prometheus.remote\_write.default.receiver]  
}  
# ...

Steps to Exercise:

Restart Grafana Alloy: docker compose restart grafana-alloy.

Generate traffic: Run python3 simulate\_data.py.

Query Aggregated Metrics in Grafana:

Go to http://localhost:3000.

Open the "Explore" view.

Query the new aggregated metric: ingestor\_api\_total\_requests\_sum.

Observe: This metric should now represent the combined request rate from all configured FastAPI instances (in this conceptual example, it will just reflect fastapi\_ingestor if ingestor\_replica is not active, but the rule is set up for aggregation).

Verification:

Grafana Explore: The aggregated metric ingestor\_api\_total\_requests\_sum can be queried, demonstrating that Grafana Alloy is applying recording rules to create new, higher-level metrics from your raw scraped data. This is foundational for calculating SLOs like "total ingestion RPS."

Advanced Use Case 3: Fan-out Data to Multiple Monitoring Backends (Conceptual)

Objective: To demonstrate Grafana Alloy's flexibility in sending telemetry data to multiple destinations simultaneously. For example, metrics to Grafana (as Prometheus storage) and traces to a separate Jaeger/Tempo backend.

Role in Platform: Enable a multi-faceted observability strategy, allowing different types of telemetry data to be routed to specialized analysis tools without requiring multiple agents on each service.

Setup/Configuration:

Ensure Basic Use Case is running.

Add another telemetry backend (conceptual): For this local setup, let's simulate sending traces to a dummy endpoint (representing Jaeger/Tempo) while metrics still go to Grafana.

Modify observability/alloy-config.river:

Add an otelcol.exporter.otlp block for traces.

Modify otelcol.receiver.otlp to forward traces to this new exporter.

Ensure your fastapi\_app is instrumented for traces (e.g., opentelemetry-instrument fastapi\_app.app.main).

Example observability/alloy-config.river snippet (additions for traces):# observability/alloy-config.river  
# ... existing prometheus.scrape and remote\_write "default" for metrics ...  
  
# 1. Define an OTLP receiver for traces (FastAPI/Spark would send traces here)  
otelcol.receiver.otlp "default" {  
 http { } # Listen for OTLP HTTP  
 grpc { } # Listen for OTLP gRPC  
 output {  
 # Forward traces to a specific exporter  
 traces = [otelcol.exporter.otlp.jaeger\_mock.input]  
 metrics = [prometheus.remote\_write.default.receiver] # Metrics still go to Grafana  
 logs = [] # No specific log forwarding for this example  
 }  
}  
  
# 2. Define an OTLP exporter for traces (e.g., to a mock Jaeger/Tempo endpoint)  
otelcol.exporter.otlp "jaeger\_mock" {  
 client {  
 endpoint = "http://jaeger-all-in-one:4318" # Conceptual Jaeger/Tempo endpoint in Docker Compose  
 # In a real setup, this would be your Jaeger/Tempo URL  
 }  
}  
  
# Update docker-compose.yml for conceptual Jaeger/Tempo service  
# services:  
# jaeger-all-in-one:  
# image: jaegertracing/all-in-one:latest  
# ports:  
# - "16686:16686" # Jaeger UI  
# - "4318:4318" # OTLP HTTP receiver  
# - "4317:4317" # OTLP gRPC receiver  
# healthcheck:  
# test: ["CMD-SHELL", "wget -q -O - http://localhost:16686/actuator/health | grep -q 'UP'"]  
# interval: 5s  
# timeout: 3s  
# retries: 5

Steps to Exercise:

Update docker-compose.yml (if adding Jaeger/Tempo mock): Add the conceptual jaeger-all-in-one service.

Rebuild and Restart services: docker compose up --build -d.

Ensure FastAPI is instrumented for OpenTelemetry traces:

You would typically run your FastAPI app with opentelemetry-instrument uvicorn fastapi\_app.app.main:app --host 0.0.0.0 --port 8000. This is usually done via entrypoint in Dockerfile or command in docker-compose.yml.

Ensure the environment variable OTEL\_EXPORTER\_OTLP\_ENDPOINT=http://grafana-alloy:4318 (or 4317 for gRPC) is set for FastAPI (and Spark if instrumented). This tells your services to send traces to Alloy.

Generate traffic: Run python3 simulate\_data.py.

Observe Grafana Alloy Logs: You should see logs indicating Alloy receiving and forwarding both metrics (to Grafana) and traces (to jaeger-mock endpoint).

Verify Traces (Conceptual): If you had a real Jaeger UI running, you would access it (http://localhost:16686) and search for traces from your fastapi\_ingestor service.

Verification:

Grafana Alloy Logs: Logs confirm dual forwarding of telemetry data types to different conceptual backends.

Backend Status: While a full trace visualization isn't possible without a fully deployed Jaeger, the logs confirm Alloy's capability to fan out data. This demonstrates Alloy's ability to act as a universal collector and router for various telemetry signals, enabling a comprehensive and flexible observability strategy.

Deep-Dive Addendum: Progressive Path Setup Guide

This addendum provides a practical, step-by-step guide for setting up each track of the "Progressive Complexity Path" on your local development machine using Docker Compose. It outlines the core steps and configuration required to bring up the various components of your enterprise data platform locally.

Prerequisites

Before starting, ensure you have the following installed:

Docker Desktop: (or Docker Engine on Linux) for running containers.

Git: For cloning the project repository.

Python 3.x: With pip for installing dependencies.

docker-compose: (usually included with Docker Desktop, or installed separately).

Project Structure Reminder:

This guide assumes you have cloned the data-ingestion-platform mono-repo, which contains all the necessary Dockerfiles, application code, and docker-compose.yml configurations.

data-ingestion-platform/  
├── data/ # Persistent Docker volumes for all services  
├── src/ # Core Python application logic  
├── fastapi\_app/ # FastAPI ingestion service  
├── pyspark\_jobs/ # Apache Spark transformation jobs (PySpark)  
├── airflow\_dags/ # Apache Airflow DAG definitions  
├── observability/ # Grafana dashboards, Grafana Alloy configurations  
├── openmetadata\_ingestion\_scripts/ # Python scripts for OpenMetadata connectors  
├── docker-compose.yml # Central Docker Compose file for local environment  
└── README.md

Onboarding Script and External Data Generation

To streamline the onboarding process and facilitate testing with realistic data, a conceptual onboarding script and an external data generator are utilized.

Onboarding Script

Role Needed: Data Engineer, Developer

This script automates the initial setup of the local environment, ensuring consistency across development machines. It performs tasks such as checking prerequisites, initializing Docker Compose, creating necessary Kafka topics and S3 buckets (MinIO), and setting up initial database schemas.

Conceptual onboard.sh script:

#!/bin/bash  
# onboard.sh - Onboarding script for local data platform environment  
  
echo "Starting local data platform environment onboarding..."  
  
# --- 1. Check Prerequisites ---  
echo "Checking prerequisites (Docker, Git, Python, docker-compose)..."  
command -v docker >/dev/null 2>&1 || { echo >&2 "Docker is not installed. Please install Docker Desktop or Docker Engine."; exit 1; }  
command -v docker-compose >/dev/null 2>&1 || { echo >&2 "Docker Compose is not installed. Please install it."; exit 1; }  
command -v python3 >/dev/null 2>&1 || { echo >&2 "Python 3 is not installed. Please install it."; exit 1; }  
echo "Prerequisites met."  
  
# --- 2. Build and Start Core Services (using the main docker-compose.yml) ---  
echo "Building and starting core services via docker compose..."  
docker compose up --build -d --remove-orphans  
  
# Give services some time to start up and become healthy  
echo "Waiting for services to become healthy (this may take a few minutes)..."  
# You might add more specific health checks here, e.g., waiting for FastAPI /health endpoint  
sleep 60 # Arbitrary wait time, adjust as needed  
  
# --- 3. Initialize Kafka Topics (if Kafka is part of the track) ---  
# This part assumes Kafka is running and accessible within the Docker network  
if docker ps --format "{{.Names}}" | grep -q "kafka"; then  
 echo "Initializing Kafka topics..."  
 # Create raw financial transactions topic  
 docker exec -it kafka kafka-topics --create --topic raw\_financial\_transactions --bootstrap-server kafka:29092 --partitions 3 --replication-factor 1 --if-not-exists  
 # Create raw insurance claims topic  
 docker exec -it kafka kafka-topics --create --topic raw\_insurance\_claims --bootstrap-server kafka:29092 --partitions 3 --replication-factor 1 --if-not-exists  
 echo "Kafka topics created."  
else  
 echo "Kafka service not detected, skipping topic creation."  
fi  
  
# --- 4. Initialize MinIO Buckets (if MinIO is part of the track) ---  
# This part assumes MinIO is running and accessible  
if docker ps --format "{{.Names}}" | grep -q "minio"; then  
 echo "Initializing MinIO buckets..."  
 # Ensure mc client is available in a separate service or installed on host  
 # For simplicity, we assume mc is available within a utility container or you create buckets manually via API/UI  
 # Example if using 'mc' client directly (might need a dedicated 'minio-client' service in docker-compose)  
 # docker exec -it minio-client mc alias set local http://minio:9000 minioadmin minioadmin  
 # docker exec -it minio-client mc mb local/raw-data-bucket --ignore-existing  
 # docker exec -it minio-client mc mb local/curated-data-bucket --ignore-existing  
  
 # Alternative: Use Python MinIO client to create buckets from FastAPI or another init script  
 echo "Please ensure raw-data-bucket and curated-data-bucket are created in MinIO manually or via an automated script."  
 echo "You can access MinIO console at http://localhost:9001 and create them manually."  
else  
 echo "MinIO service not detected, skipping bucket creation."  
fi  
  
# --- 5. Database Schema Initialization (if applicable) ---  
# For PostgreSQL, you might have a script to apply migrations  
if docker ps --format "{{.Names}}" | grep -q "postgres"; then  
 echo "Applying PostgreSQL database migrations..."  
 # Example: Run a Flyway/Alembic migration script from a dedicated container or a FastAPI init script  
 echo "Ensure your FastAPI service or a dedicated migration container handles database schema initialization."  
fi  
  
echo "Onboarding complete. Your local data platform should now be running."  
echo "Access FastAPI at http://localhost:8000"  
echo "Access MinIO Console at http://localhost:9001"  
echo "Access Airflow UI at http://localhost:8080 (if Advanced Track is enabled)"

External Data Generator

Role Needed: Data Analyst, QA Engineer, Developer

The external data generator is a standalone Python script (e.g., using Locust, as detailed in the Testing & Observability Patterns Deep-Dive Addendum) that simulates incoming data from various sources. This is critical for testing the ingestion layer and populating the data lake with mock, yet realistic, data volumes to simulate production load.

Purpose: To continuously send mock financial transactions and insurance claims to the FastAPI Ingestor, allowing for load testing, functional validation, and populating the data pipeline for downstream processing and analysis.

Usage: Run this script from your local machine after the FastAPI Ingestor service is up and running. It interacts with the exposed API endpoint.

Conceptual Python script (simulate\_data.py - simplified version, not a full Locust script):

# simulate\_data.py  
import requests  
import json  
import time  
from datetime import datetime, timedelta  
import random  
  
FASTAPI\_URL = "http://localhost:8000" # Ensure this matches your FastAPI exposure  
DELAY\_SECONDS = 0.1 # Time between sending each record  
  
def generate\_financial\_transaction():  
 """Generates a mock financial transaction."""  
 return {  
 "transaction\_id": f"FT-{datetime.now().strftime('%Y%m%d%H%M%S%f')}-{random.randint(1000, 9999)}",  
 "timestamp": datetime.now().isoformat(),  
 "account\_id": f"ACC-{random.randint(100000, 999999)}",  
 "amount": round(random.uniform(1.0, 10000.0), 2),  
 "currency": random.choice(["USD", "EUR", "GBP", "JPY"]),  
 "transaction\_type": random.choice(["debit", "credit", "transfer", "payment"]),  
 "merchant\_id": f"MER-{random.randint(100, 999)}" if random.random() > 0.3 else None,  
 "category": random.choice(["groceries", "utilities", "salary", "entertainment", "transport", "housing", "healthcare", "education"])  
 }  
  
def generate\_insurance\_claim():  
 """Generates a mock insurance claim."""  
 return {  
 "claim\_id": f"IC-{datetime.now().strftime('%Y%m%d%H%M%S%f')}-{random.randint(1000, 9999)}",  
 "timestamp": datetime.now().isoformat(),  
 "policy\_number": f"POL-{random.randint(1000000, 9999999)}",  
 "claim\_amount": round(random.uniform(500.0, 50000.0), 2),  
 "claim\_type": random.choice(["auto", "health", "home", "life", "property"]),  
 "claim\_status": random.choice(["submitted", "under\_review", "approved", "rejected", "paid"]),  
 "customer\_id": f"CUST-{random.randint(10000, 99999)}",  
 "incident\_date": (datetime.now() - timedelta(days=random.randint(0, 365))).isoformat()  
 }  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 print(f"Starting data generation. Sending to {FASTAPI\_URL}")  
 while True:  
 try:  
 # Send financial transaction  
 financial\_data = generate\_financial\_transaction()  
 response\_ft = requests.post(f"{FASTAPI\_URL}/ingest-financial-transaction/", json=financial\_data)  
 if response\_ft.status\_code == 200:  
 print(f"Sent financial transaction {financial\_data['transaction\_id']} (Status: {response\_ft.status\_code})")  
 else:  
 print(f"Error sending financial transaction: {response\_ft.status\_code} - {response\_ft.text}")  
  
 time.sleep(DELAY\_SECONDS)  
  
 # Send insurance claim  
 insurance\_data = generate\_insurance\_claim()  
 response\_ic = requests.post(f"{FASTAPI\_URL}/ingest-insurance-claim/", json=insurance\_data)  
 if response\_ic.status\_code == 200:  
 print(f"Sent insurance claim {insurance\_data['claim\_id']} (Status: {response\_ic.status\_code})")  
 else:  
 print(f"Error sending insurance claim: {response\_ic.status\_code} - {response\_ic.text}")  
  
 time.sleep(DELAY\_SECONDS)  
  
 except requests.exceptions.ConnectionError as e:  
 print(f"Connection error: {e}. Is FastAPI running at {FASTAPI\_URL}? Retrying in 5 seconds...")  
 time.sleep(5)  
 except Exception as e:  
 print(f"An unexpected error occurred: {e}")  
 time.sleep(5)

Setup Guide by Track

The docker-compose.yml provided with the project is designed to be flexible. You will often comment out or enable services based on the track you are working on.

3.1. Starter Track Setup: Minimal Single-Machine Setup

This track focuses on FastAPI, PostgreSQL, and MinIO.

Navigate to Project Root:  
cd /path/to/your/data-ingestion-platform

Prepare docker-compose.yml:

Open docker-compose.yml.

Uncomment the services for fastapi\_ingestor, postgres, minio.

Comment out all other services (zookeeper, kafka, spark, airflow, openmetadata, grafana, etc.) to keep the setup minimal.

Ensure the data/postgres, data/minio directories exist or are created by Docker Compose for persistent volumes.

Conceptual Snippet of docker-compose.yml for Starter Track:# docker-compose.yml (Starter Track focus)  
version: '3.8'  
services:  
 postgres:  
 image: postgres:15-alpine  
 environment:  
 POSTGRES\_DB: main\_db  
 POSTGRES\_USER: user  
 POSTGRES\_PASSWORD: password  
 ports:  
 - "5432:5432"  
 volumes:  
 - ./data/postgres:/var/lib/postgresql/data  
 healthcheck:  
 test: ["CMD-SHELL", "pg\_isready -U user -d main\_db"]  
 interval: 5s  
 timeout: 5s  
 retries: 5  
  
 minio:  
 image: minio/minio:latest  
 environment:  
 MINIO\_ROOT\_USER: minioadmin  
 MINIO\_ROOT\_PASSWORD: minioadmin  
 ports:  
 - "9000:9000"  
 - "9001:9001" # Console port  
 volumes:  
 - ./data/minio:/data  
 command: server /data --console-address ":9001"  
 healthcheck:  
 test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]  
 interval: 30s  
 timeout: 20s  
 retries: 3  
  
 fastapi\_ingestor:  
 build: ./fastapi\_app  
 environment:  
 POSTGRES\_HOST: postgres  
 POSTGRES\_DB: main\_db  
 POSTGRES\_USER: user  
 POSTGRES\_PASSWORD: password  
 MINIO\_HOST: minio:9000  
 MINIO\_ACCESS\_KEY: minioadmin  
 MINIO\_SECRET\_KEY: minioadmin  
 MINIO\_BUCKET: raw-data-bucket  
 ports:  
 - "8000:8000"  
 depends\_on:  
 postgres:  
 condition: service\_healthy  
 minio:  
 condition: service\_healthy  
 healthcheck:  
 test: ["CMD", "curl", "-f", "http://localhost:8000/health || exit 1"]  
 interval: 5s  
 timeout: 3s  
 retries: 5  
# ... other services commented out

Bring Up Services:  
docker compose up --build -d  
  
This command builds the Docker images (if necessary) and starts the selected services in detached mode.

Verify Setup:

Access FastAPI health check: http://localhost:8000/health

Access MinIO Console: http://localhost:9001 (login with minioadmin/minioadmin)

Use a PostgreSQL client to connect to localhost:5432 with user user, password password, database main\_db.

Check Docker logs: docker compose logs -f

3.2. Intermediate Track Setup: Adding Streaming Capabilities

This track adds Apache Kafka and Apache Spark to the Starter Track components.

Navigate to Project Root:  
cd /path/to/your/data-ingestion-platform

Prepare docker-compose.yml:

Open docker-compose.yml.

Uncomment (or keep uncommented from Starter Track) fastapi\_ingestor, postgres, minio.

Uncomment the services for zookeeper, kafka, and spark.

Comment out other Advanced Track services (airflow, openmetadata, grafana, etc.).

Update fastapi\_ingestor to publish to Kafka (remove direct MinIO/PostgreSQL writes, or add Kafka producer logic). Ensure KAFKA\_BROKER environment variable points to kafka:29092.

Ensure spark service configuration is set to connect to kafka and minio. Set checkpoint locations for Spark Structured Streaming.

Ensure data/spark-events directory exists for Spark history server.

Conceptual Snippet of docker-compose.yml for Intermediate Track (partial, focusing on additions):# docker-compose.yml (Intermediate Track focus)  
version: '3.8'  
services:  
 # ... postgres, minio (from Starter Track)  
  
 zookeeper:  
 image: confluentinc/cp-zookeeper:7.4.0  
 environment:  
 ZOOKEEPER\_CLIENT\_PORT: 2181  
 healthcheck:  
 test: ["CMD-SHELL", "echo stat | nc localhost 2181"] # Basic Zookeeper health  
 interval: 5s  
 timeout: 5s  
 retries: 10  
  
 kafka:  
 image: confluentinc/cp-kafka:7.4.0  
 depends\_on:  
 zookeeper:  
 condition: service\_healthy  
 ports:  
 - "9092:9092" # External for host tools  
 environment:  
 KAFKA\_BROKER\_ID: 1  
 KAFKA\_ZOOKEEPER\_CONNECT: 'zookeeper:2181'  
 KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT\_HOST://localhost:9092  
 KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT\_HOST:PLAINTEXT  
 KAFKA\_INTER\_BROKER\_LISTENER\_NAME: PLAINTEXT  
 KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1  
 KAFKA\_DELETE\_TOPIC\_ENABLE: "true" # For development  
 healthcheck:  
 test: ["CMD-SHELL", "kafka-topics --bootstrap-server localhost:9092 --list"]  
 interval: 10s  
 timeout: 5s  
 retries: 10  
  
 fastapi\_ingestor:  
 build: ./fastapi\_app  
 environment:  
 # ... (Postgres, Minio from Starter Track if still needed for metadata)  
 KAFKA\_BROKER: kafka:29092 # New: FastAPI publishes to Kafka  
 KAFKA\_TOPIC\_FINANCIAL: raw\_financial\_transactions  
 KAFKA\_TOPIC\_INSURANCE: raw\_insurance\_claims  
 depends\_on:  
 # ... postgres, minio  
 kafka: # New dependency  
 condition: service\_healthy  
 # ... healthcheck  
  
 spark:  
 image: bitnami/spark:3.5.0  
 command: ["tail", "-f", "/dev/null"] # Keep container alive for spark-submit  
 environment:  
 SPARK\_MASTER\_URL: "spark://spark-master:7077" # Assuming spark-master service  
 SPARK\_LOCAL\_IP: spark # For internal networking  
 SPARK\_DAEMON\_JAVA\_OPTS: "-Dspark.history.fs.logDirectory=file:///opt/bitnami/spark/spark-events"  
 # Kafka connectivity for Spark jobs  
 KAFKA\_BROKER\_ADDRESS: kafka:29092  
 MINIO\_HOST: minio  
 MINIO\_ACCESS\_KEY: minioadmin  
 MINIO\_SECRET\_KEY: minioadmin  
 MINIO\_BUCKET\_RAW: raw-data-bucket  
 MINIO\_BUCKET\_CURATED: curated-data-bucket  
 volumes:  
 - ./pyspark\_jobs:/opt/bitnami/spark/jobs # Mount your PySpark jobs  
 - ./data/spark-events:/opt/bitnami/spark/spark-events # For Spark History Server  
 depends\_on:  
 kafka:  
 condition: service\_healthy  
 minio:  
 condition: service\_healthy  
 # No exposed ports if only used internally by Airflow or manual spark-submit  
 # If you need Spark UI, expose 8080:8080 (master) and 8081:8081 (worker) for manual debugging  
  
 # Optional: Spark History Server  
 spark-history-server:  
 image: bitnami/spark:3.5.0  
 command: ["/opt/bitnami/spark/bin/spark-history-server.sh"]  
 environment:  
 SPARK\_HISTORY\_FS\_LOGDIRECTORY: "/opt/bitnami/spark/spark-events"  
 ports:  
 - "18080:18080" # Spark History UI  
 volumes:  
 - ./data/spark-events:/opt/bitnami/spark/spark-events  
 depends\_on:  
 spark:  
 condition: service\_started  
# ... other services commented out

Bring Up Services:  
docker compose up --build -d

Verify Setup:

Verify Starter Track components are running.

Check Kafka topic creation: docker exec -it kafka kafka-topics --bootstrap-server localhost:9092 --list

Run a sample Spark streaming job (e.g., via docker exec -it spark spark-submit /opt/bitnami/spark/jobs/streaming\_consumer.py <args>).

Check Spark History Server: http://localhost:18080 (if enabled).

3.3. Advanced Track Setup: The Full Production-Ready Stack

This track integrates orchestration, observability, lineage, and metadata management.

Navigate to Project Root:  
cd /path/to/your/data-ingestion-platform

Prepare docker-compose.yml:

Uncomment all services including airflow-init, airflow-webserver, airflow-scheduler, airflow-worker, mongodb, openmetadata, grafana, grafana-alloy, cAdvisor, spline.

Ensure all necessary environment variables for inter-service communication are correctly set (e.g., Airflow connecting to Spark, OpenMetadata connecting to PostgreSQL/MongoDB, Grafana Alloy pointing to Prometheus/OpenTelemetry targets).

Mount airflow\_dags and observability directories as volumes for Airflow DAGs and Grafana configurations.

Ensure all data/ subdirectories for persistent volumes exist.

Conceptual Snippet of docker-compose.yml for Advanced Track (partial, focusing on additions):# docker-compose.yml (Advanced Track focus)  
version: '3.8'  
services:  
 # ... postgres, minio, zookeeper, kafka, spark, spark-history-server  
  
 mongodb:  
 image: mongo:6.0  
 ports:  
 - "27017:27017"  
 volumes:  
 - ./data/mongodb:/data/db  
 healthcheck:  
 test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]  
 interval: 5s  
 timeout: 5s  
 retries: 5  
  
 airflow-init:  
 image: apache/airflow:2.8.0  
 entrypoint: ["/bin/bash", "-c"]  
 command:  
 - "airflow db migrate && airflow users create --username admin --password admin --firstname Admin --lastname User --role Admin --email admin@example.com"  
 environment:  
 \_PIP\_ADDITIONAL\_REQUIREMENTS: "apache-airflow-providers-cncf-kubernetes apache-airflow-providers-apache-kafka apache-airflow-providers-cncf-kubernetes apache-airflow-providers-postgres delta-spark"  
 AIRFLOW\_HOME: /opt/airflow  
 AIRFLOW\_\_CORE\_\_LOAD\_EXAMPLES: "false"  
 AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_CONN: "postgresql+psycopg2://user:password@postgres/main\_db"  
 AIRFLOW\_\_WEBSERVER\_\_RBAC: "True"  
 AIRFLOW\_\_CORE\_\_DAGS\_FOLDER: /opt/airflow/dags  
 volumes:  
 - ./airflow\_dags:/opt/airflow/dags  
 - ./src:/opt/airflow/src # Mount source code for Airflow to access  
 depends\_on:  
 postgres:  
 condition: service\_healthy  
  
 airflow-webserver:  
 image: apache/airflow:2.8.0  
 command: webserver  
 ports:  
 - "8080:8080" # Airflow UI  
 environment:  
 AIRFLOW\_HOME: /opt/airflow  
 AIRFLOW\_\_CORE\_\_LOAD\_EXAMPLES: "false"  
 AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_CONN: "postgresql+psycopg2://user:password@postgres/main\_db"  
 AIRFLOW\_\_WEBSERVER\_\_RBAC: "True"  
 AIRFLOW\_\_CORE\_\_DAGS\_FOLDER: /opt/airflow/dags  
 volumes:  
 - ./airflow\_dags:/opt/airflow/dags  
 - ./src:/opt/airflow/src  
 depends\_on:  
 airflow-init:  
 condition: service\_completed\_successfully  
  
 airflow-scheduler:  
 image: apache/airflow:2.8.0  
 command: scheduler  
 environment:  
 AIRFLOW\_HOME: /opt/airflow  
 AIRFLOW\_\_CORE\_\_LOAD\_EXAMPLES: "false"  
 AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_CONN: "postgresql+psycopg2://user:password@postgres/main\_db"  
 AIRFLOW\_\_WEBSERVER\_\_RBAC: "True"  
 AIRFLOW\_\_CORE\_\_DAGS\_FOLDER: /opt/airflow/dags  
 AIRFLOW\_\_CORE\_\_EXECUTOR: LocalExecutor # For local testing; CeleryExecutor for distributed  
 volumes:  
 - ./airflow\_dags:/opt/airflow/dags  
 - ./src:/opt/airflow/src  
 depends\_on:  
 airflow-webserver:  
 condition: service\_healthy  
  
 # Airflow Worker (optional for local, if using CeleryExecutor)  
 # airflow-worker:  
 # image: apache/airflow:2.8.0  
 # command: worker  
 # environment:  
 # # ... same as scheduler, plus Celery config  
 # depends\_on:  
 # airflow-scheduler:  
 # condition: service\_healthy  
  
 openmetadata:  
 image: openmetadata/openmetadata:1.2.3-release  
 ports:  
 - "8585:8585" # OpenMetadata UI  
 environment:  
 # ... OpenMetadata configuration for MySQL, Elasticsearch (using data/ volumes)  
 # Make sure it points to your local Postgres for ingestion metadata  
 MYSQL\_HOST: openmetadata\_mysql # Or your specific MySQL service  
 ELASTICSEARCH\_HOST: openmetadata\_elasticsearch # Or your specific ES service  
 depends\_on:  
 # ... openmetadata\_mysql, openmetadata\_elasticsearch (or equivalent)  
  
 spline:  
 image: absaoss/spline-rest-server:0.7.1 # Or latest  
 ports:  
 - "8081:8081" # Spline UI  
 environment:  
 SPLINE\_DATABASE\_URL: "jdbc:postgresql://postgres:5432/main\_db"  
 SPLINE\_DATABASE\_USERNAME: user  
 SPLINE\_DATABASE\_PASSWORD: password  
 depends\_on:  
 postgres:  
 condition: service\_healthy  
 spark:  
 condition: service\_started  
  
 grafana:  
 image: grafana/grafana-oss:10.2.0 # Or latest  
 ports:  
 - "3000:3000"  
 volumes:  
 - ./data/grafana:/var/lib/grafana  
 - ./observability/grafana\_datasources\_provisioning:/etc/grafana/provisioning/datasources  
 - ./observability/grafana\_dashboards\_provisioning:/etc/grafana/provisioning/dashboards  
 environment:  
 GF\_AUTH\_ANONYMOUS\_ENABLED: "true" # For quick local access  
 GF\_AUTH\_ANONYMOUS\_ORG\_ROLE: Admin  
 GF\_SERVER\_ROOT\_URL: "http://localhost:3000"  
 GF\_INSTALL\_PLUGINS: "grafana-piechart-panel" # Example plugin  
 depends\_on:  
 grafana-alloy:  
 condition: service\_healthy  
  
 grafana-alloy:  
 build:  
 context: ./observability  
 dockerfile: Dockerfile.alloy # Custom Dockerfile for Grafana Alloy config  
 volumes:  
 - ./observability/alloy-config.river:/etc/alloy/config.river  
 command: ["--config.file=/etc/alloy/config.river"]  
 ports:  
 - "12345:12345" # OpenTelemetry receiver  
 - "9090:9090" # Prometheus metrics endpoint for self-monitoring  
 depends\_on:  
 # Link to services it collects from  
 fastapi\_ingestor:  
 condition: service\_healthy  
 kafka:  
 condition: service\_healthy  
 spark:  
 condition: service\_healthy  
 cAdvisor:  
 condition: service\_healthy  
  
 cAdvisor:  
 image: gcr.io/cadvisor/cadvisor:v0.47.0 # Or latest stable  
 volumes:  
 - /:/rootfs:ro  
 - /var/run:/var/run:rw  
 - /sys:/sys:ro  
 - /var/lib/docker/:/var/lib/docker:ro  
 - /dev/disk/:/dev/disk:ro  
 privileged: true  
 ports:  
 - "8080:8080" # cAdvisor UI (can clash with Airflow, adjust if needed)  
 # No healthcheck for simplicity, but a proper one would check metrics endpoint

Bring Up Services:  
docker compose up --build -d

Verify Setup:

Access Airflow UI: http://localhost:8080 (login admin/admin)

Access Grafana UI: http://localhost:3000 (initially anonymous or configure admin user)

Access OpenMetadata UI: http://localhost:8585

Verify Spline UI: http://localhost:8081

Check for container metrics in Grafana dashboards.

Ensure Airflow DAGs appear and run as expected.

Run data ingestion, processing, and observe metrics, lineage, and metadata.

This guide provides the core steps. Remember that exact configurations may vary based on your specific implementation of each component. Always consult the detailed documentation for each individual technology if you encounter issues.

Highlighting OpenMetadata: Unified Data Catalog & Governance

OpenMetadata is an open-source metadata management platform that provides a unified data catalog, data lineage, and data quality capabilities. It acts as the central hub for data discovery, understanding, and governance within your data platform. By consolidating metadata from various sources (databases, streaming platforms, data lakes, APIs), OpenMetadata empowers users to find, trust, and collaborate on data.

This guide will demonstrate basic and advanced use cases of OpenMetadata, leveraging your Advanced Track local environment setup and its integration with Spark, Spline, and other components.

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, specifically OpenMetadata's role in the Orchestration & Governance Layer.

Basic Use Case: Data Discovery and Browsing

Objective: To demonstrate how users can easily discover and browse various data assets (Kafka topics, Delta tables, PostgreSQL tables, FastAPI endpoints) through the OpenMetadata UI.

Role in Platform: Facilitate self-service data discovery for data analysts, scientists, and engineers, enabling them to quickly find relevant datasets without needing to understand the underlying infrastructure.

Setup/Configuration (Local Environment - Advanced Track):

Ensure all Advanced Track services are running: docker compose up --build -d from your project root. This includes openmetadata, postgres, kafka, minio, fastapi\_ingestor, and spark.

Verify OpenMetadata is accessible: Navigate to http://localhost:8585 in your web browser. (Initial login: admin/admin if not configured otherwise).

Ensure initial metadata ingestion has occurred: Your Airflow DAGs (e.g., those in airflow\_dags/ designed for OpenMetadata ingestion) should have run at least once to populate the catalog. These DAGs typically call scripts in openmetadata\_ingestion\_scripts/ that use OpenMetadata's Python client to extract metadata from different sources.

You might need to manually trigger the Airflow DAGs responsible for metadata ingestion (e.g., openmetadata\_ingestion\_dag) from the Airflow UI (http://localhost:8080).

Steps to Exercise:

Access OpenMetadata UI: Open your web browser and go to http://localhost:8585.

Browse Data Assets:

From the left-hand navigation, click "Explore".

You can browse by "Services" (e.g., kafka\_broker, minio\_s3, my\_postgres\_db, my\_fastapi\_service) or by "Entities" (e.g., Tables, Topics).

Click on a Service: For example, click on your minio\_s3 service. You should see listed tables like raw\_data\_bucket.financial\_data\_delta and raw\_data\_bucket.insurance\_data\_delta (assuming your Spark jobs have created these Delta tables).

Click on an Entity: Navigate to a specific table or topic (e.g., raw\_data\_bucket.financial\_data\_delta).

Explore Details: Observe the "Schema" tab to see column names and data types, and the "Sample Data" tab (if profiling is enabled and run).

Verification:

OpenMetadata UI: The UI successfully loads and displays a list of services (Kafka, S3/MinIO, PostgreSQL, FastAPI), and under each service, you can find and browse the associated data assets (topics, tables). The schema and basic details for these assets are visible, demonstrating basic data discovery.

Advanced Use Case 1: Automated Data Lineage Visualization

Objective: To demonstrate how OpenMetadata automatically captures and visualizes end-to-end data lineage, showing how data flows through your Spark transformation jobs from source Kafka topics to target Delta Lake tables.

Role in Platform: Provide transparency into data origins, transformations, and dependencies, crucial for impact analysis (e.g., "what breaks if I change this column?"), debugging data quality issues, and fulfilling compliance requirements.

Setup/Configuration:

Ensure Basic Use Case setup is complete.

Ensure Spark jobs are running/have run: Your Spark streaming jobs (Kafka to raw Delta) and batch transformation jobs (raw to curated Delta) must have executed at least once (e.g., from Airflow DAGs).

Ensure Spline is running: Spline (http://localhost:8081) must be collecting lineage from your Spark jobs.

Ensure OpenMetadata Lineage Ingestion is active: The Airflow DAG responsible for ingesting lineage from Spline into OpenMetadata should have run successfully. This DAG typically triggers the Spline connector in OpenMetadata.

Steps to Exercise:

Access OpenMetadata UI: Go to http://localhost:8585.

Search for a Curated Data Asset: Use the search bar (e.g., search for financial\_data\_curated or insurance\_data\_curated) and click on one of your curated Delta Lake tables.

Navigate to the "Lineage" Tab:

Once on the asset's detail page, click the "Lineage" tab.

Observe: You should see a visual graph depicting the data flow. This graph will typically show:

An upstream Kafka topic (e.g., raw\_financial\_transactions).

A Spark transformation job (represented as a process node).

The intermediate raw Delta Lake table (e.g., raw-data-bucket.financial\_data\_delta).

Another Spark transformation job.

The final curated Delta Lake table (e.g., curated-data-bucket.financial\_data\_curated).

Explore Column-Level Lineage: Click on individual columns within the graph. OpenMetadata (if configured and data exists from Spline) can show which source columns contribute to which target columns.

Verification:

OpenMetadata UI: The "Lineage" tab for your curated Delta tables accurately displays the upstream Kafka topics, intermediate raw Delta tables, and the Spark jobs as process nodes, confirming end-to-end data lineage visualization. Column-level lineage provides fine-grained dependency tracking.

Advanced Use Case 2: Active Data Governance (Tags, Ownership, Descriptions)

Objective: To demonstrate how data stewards and owners can enrich metadata within OpenMetadata by adding business-friendly descriptions, assigning ownership, and applying classification tags (e.g., PII, sensitive data).

Role in Platform: Enable active data governance, improve data understanding across the organization, facilitate compliance efforts (e.g., identifying PII), and foster data ownership.

Setup/Configuration:

Ensure OpenMetadata UI is accessible and data assets are imported.

Steps to Exercise:

Assign an Owner to a Table:

In OpenMetadata UI, navigate to raw\_data\_bucket.financial\_data\_delta.

Click the "Manage" tab or look for an "Add Owner" button/section.

Assign yourself or a dummy "Data Team" as the owner.

Verify: The owner is now displayed on the table's overview page.

Add a Business Description to a Table:

On the raw\_data\_bucket.financial\_data\_delta overview page, find the "Description" section.

Click the "Edit" icon and add a meaningful business description (e.g., "This table contains raw, unvalidated financial transaction data directly ingested from source systems. Used for initial historical record keeping.").

Verify: The description is saved and visible.

Add a PII Tag to a Column:

On the raw\_data\_bucket.financial\_data\_delta table, go to the "Schema" tab.

Locate a column that might contain sensitive information (e.g., account\_id or user\_id if present in raw).

Click the "Tags" icon next to the column name.

Search for and select a PII.Sensitive or Personal.Sensitive tag (OpenMetadata has a default set of classification tags).

Verify: The tag is now associated with the column and is visible next to its name.

Search by Tag: Go back to the "Explore" page and try searching for the tag (e.g., PII.Sensitive). Your table should appear in the search results, demonstrating discoverability by governance classifications.

Verification:

OpenMetadata UI: The changes (owner, description, tags) are immediately reflected on the asset's detail page. Searching by tags successfully filters relevant assets, demonstrating effective data governance capabilities.

Advanced Use Case 3: Data Quality Profiling & Monitoring Integration (Conceptual)

Objective: To demonstrate (conceptually) how OpenMetadata integrates with data quality tools and displays data quality metrics and profiles directly within the data catalog.

Role in Platform: Surface data quality issues and insights directly alongside data discovery and lineage, empowering data consumers to trust the data and data owners to quickly address issues.

Setup/Configuration (Conceptual Discussion):

Ensure OpenMetadata is running.

Conceptual Integration with Data Quality Tools (e.g., Great Expectations):

In a production setup, your Spark ETL jobs would typically integrate with a data quality framework like Great Expectations. After a Spark job runs, Great Expectations would generate data quality "expectations" (tests) and "validation results" (pass/fail for those tests).

An OpenMetadata ingestion pipeline would then be configured to collect these Great Expectations results and link them to the relevant tables in the data catalog.

OpenMetadata can also run its own "Profiler" workflows, which automatically calculate column-level statistics (e.g., min, max, average, null count, distinct count) and data quality metrics (e.g., completeness, uniqueness) based on data stored in MinIO/S3 (Delta Lake).

Steps to Exercise (Conceptual/Discussion):

View "Profiler" Tab:

In OpenMetadata UI, navigate to one of your Delta Lake tables (e.g., curated-data-bucket.financial\_data\_curated).

Click the "Profiler" tab (if enabled and run).

Observe: You should see various statistical profiles for each column (e.g., column\_name, data\_type, null\_count, distinct\_count, min\_value, max\_value, average\_value).

Discuss Data Quality Metrics: Explain how these profiles can be used to assess data quality (e.g., a high null\_count for a mandatory column indicates incompleteness).

Discuss "Data Quality" Tab (Conceptual with Great Expectations):

If Great Expectations integration were fully set up and results ingested, you would see a "Data Quality" tab.

Discuss: This tab would show a summary of failed/passed data quality tests, potentially with links to detailed Great Expectations validation reports.

Verification (Conceptual):

OpenMetadata UI: The "Profiler" tab (even with basic auto-profiling) demonstrates OpenMetadata's capability to provide statistical insights into data quality. A successful integration with a tool like Great Expectations would show explicit pass/fail results for defined data quality rules, highlighting OpenMetadata's role as a central hub for data quality monitoring and trust. This allows data consumers to quickly assess the reliability of a dataset before using it.

This concludes the guide for OpenMetadata.

Data Platform Usage Guide: Ingestion, Processing & Observation

This guide provides a practical walkthrough on how to use your local enterprise-ready data platform. It covers the core steps for data ingestion, processing, and how to observe the data flow and system health, specifically demonstrating handling of disparate financial and insurance data. This version introduces additional complexity to demonstrate the system's flexibility in handling different data types and its scalability, as well as integrating with Machine Learning, showcasing serverless ETL with Lambdas, and explicitly demonstrating data lineage.

Before you begin, ensure your local environment is fully set up following the "Progressive Path Setup Guide Deep-Dive Addendum", especially the Advanced Track which includes all components.

1. Starting the Platform (Advanced Track)

To ensure all components are running, use the docker compose command from your project root.

Navigate to your project root:  
cd /path/to/your/data-ingestion-platform

Bring up all services (Advanced Track):  
This command will start FastAPI, PostgreSQL, MinIO, Zookeeper, Kafka, Spark, Airflow, MongoDB, OpenMetadata, Spline, Grafana, Grafana Alloy, and cAdvisor.  
docker compose up --build -d  
  
Wait a few minutes for all services to become healthy. You can check their status with docker compose ps and docker compose logs -f.

Perform initial Kafka topic setup (if not done by onboard.sh):  
Ensure both financial and insurance topics exist.  
docker exec -it kafka kafka-topics --create --topic raw\_financial\_transactions --bootstrap-server kafka:29092 --partitions 3 --replication-factor 1 --if-not-exists  
docker exec -it kafka kafka-topics --create --topic raw\_insurance\_claims --bootstrap-server kafka:29092 --partitions 3 --replication-factor 1 --if-not-exists

2. Ingesting Disparate Data (FastAPI & Kafka)

You will use the simulate\_data.py script to generate mock financial and insurance data and send it to your FastAPI ingestor. FastAPI will then publish this data to their respective Kafka topics.

Run the data generator:  
Open a new terminal session in your project root and execute the simulate\_data.py script. This script will continuously send data to your FastAPI endpoints.  
python3 simulate\_data.py  
  
You should see console output indicating data being sent (e.g., "Sent financial transaction...", "Sent insurance claim..."). Let this script run in the background.

Verify data in Kafka:  
You can manually inspect Kafka topics to ensure data is arriving. Open two new terminal sessions.

Financial Data Consumer:  
docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw\_financial\_transactions --from-beginning

Insurance Data Consumer:  
docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw\_insurance\_claims --from-beginning

You should see a continuous stream of JSON messages in both consumer terminals.

3. Processing Data (Spark Structured Streaming)

Now, you will start the Spark Structured Streaming jobs to consume data from Kafka and write it as Delta Lake files into MinIO (your simulated S3 data lake). Since you have two types of data, you will likely have two separate streaming jobs or a single job designed to handle both.

Submit Spark Streaming Job for Financial Data:  
Open another new terminal session and run the following command to submit your financial streaming consumer job. This command assumes your streaming\_consumer.py is designed to take topic and output path as arguments.  
docker exec -it 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 \  
 raw\_financial\_transactions kafka:29092 s3a://raw-data-bucket/financial\_data\_delta

Submit Spark Streaming Job for Insurance Data:  
Open yet another new terminal session and submit the insurance streaming consumer job.  
docker exec -it 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 \  
 raw\_insurance\_claims kafka:29092 s3a://raw-data-bucket/insurance\_data\_delta  
  
Let both Spark jobs run. They will continuously consume from their respective topics.

Verify data in MinIO (Raw Zone):

Access the MinIO Console: http://localhost:9001 (login minioadmin/minioadmin).

Navigate to the raw-data-bucket.

You should see two new directories: financial\_data\_delta and insurance\_data\_delta, each containing .parquet files and a \_delta\_log directory. This confirms your Spark jobs are writing data.

4. Alternative Processing: Lightweight ETL with AWS Lambda (Conceptual)

While not running locally via Docker Compose, it's crucial to understand how AWS Lambda functions would fit into this architecture for specific ETL tasks, typically for lightweight, event-driven transformations.

Concept: Imagine a Lambda function that triggers whenever a new file lands in your raw-data-bucket (MinIO/S3), or a message arrives on a specific Kafka topic. This Lambda then performs a simple transformation (e.g., anonymization, minor field mapping, data enrichment from a small lookup table) and writes the result to a different S3 location or a database.

Role in Platform:

Event-Driven: Ideal for immediate processing of individual records or small files.

Cost-Effective: Pay-per-execution, suitable for intermittent or bursty workloads.

Scalable: Automatically scales to meet demand without managing servers.

Conceptual Workflow:

Event Source: New JSON file in raw-data-bucket/financial\_data\_delta (S3 trigger), or a Kafka message on a specific topic.

Lambda Function:

Reads the event/file content.

Performs a simple, in-memory transformation (e.g., adds a processing\_timestamp, masks sensitive fields).

Writes the transformed data to curated-data-bucket/lambda\_processed\_data (S3) or inserts into a PostgreSQL table.

Observability: Lambda's execution metrics (invocations, errors, duration) would automatically feed into CloudWatch (equivalent to Grafana metrics for local Lambda simulation).

Local Simulation:

While a live Lambda can't run in Docker Compose, you can locally test Lambda functions using AWS SAM CLI.

Step 1: Define a SAM template: In your fastapi\_app or a new lambda\_etl\_app directory, you'd define template.yaml for your Lambda function.

Step 2: Implement Lambda logic: Write your Python code for the transformation.

Step 3: Test locally: Use sam local invoke or sam local start-api to test your Lambda locally, simulating S3 events or Kafka messages.

This demonstrates how agile, serverless components handle specific ETL needs, complementing the batch and streaming capabilities of Spark.

5. Orchestration with Airflow

Your platform includes Apache Airflow for orchestrating complex workflows. In the Advanced Track, you should have DAGs defined in airflow\_dags/ that trigger your Spark jobs, or even metadata ingestion tasks.

Access Airflow UI: http://localhost:8080 (login admin/admin).

Unpause and Trigger DAGs:

Locate DAGs like financial\_data\_processing\_dag and insurance\_data\_processing\_dag (or whatever names you've given them).

Toggle their status to "On" (unpause).

Manually trigger a run for each DAG (click the "Play" icon).

Monitor DAG Runs: Observe the Graph View and Task Logs for each DAG run.

Expected: The DAGs should execute tasks, including Spark job submissions, successfully. Look for green boxes and "success" messages in logs. This demonstrates Airflow's ability to manage complex, multi-component pipelines.

6. Observing the System & Data Governance (Grafana, OpenMetadata, Spline)

The Advanced Track heavily emphasizes observability and governance. These tools provide insights into system health, data flow, and metadata, giving you full control and understanding of your data landscape.

6.1. Grafana for Real-time Monitoring & Performance Metrics

Grafana dashboards visualize metrics from various components (FastAPI, Kafka, Spark, Docker containers via cAdvisor), providing real-time insights into system performance.

Access Grafana UI: http://localhost:3000 (initially anonymous or configure admin user).

Explore Dashboards:

Navigate to pre-provisioned dashboards (e.g., "Health Dashboard", "Kafka Overview", "Spark Jobs").

Observe:

FastAPI: Request rates (RPS), latency for both financial and insurance ingestion endpoints.

Kafka: Producer and consumer throughput for raw\_financial\_transactions and raw\_insurance\_claims. Crucially, monitor Kafka consumer lag for your Spark jobs. It should remain low and relatively stable, indicating Spark is keeping up.

Spark: Resource utilization (CPU, memory) of the Spark container, job completion times, and processed record counts.

Docker Container Metrics (via cAdvisor): Overall CPU/memory usage for all running Docker services.

Demonstration: Introduce a heavy load (e.g., increase DELAY\_SECONDS in simulate\_data.py to 0.01 or less, or run multiple instances of it) and observe how metrics like latency or Kafka lag respond. This shows the system's capacity and helps identify bottlenecks.

6.2. Spline for Automated Data Lineage

Spline automatically captures and visualizes data lineage for your Apache Spark jobs. This is critical for understanding data origins, transformations, and impacts of changes.

Access Spline UI: http://localhost:8081.

View Lineage:

You should see entries for your financial and insurance Spark jobs (from Section 3.2, or triggered by Airflow in Section 5).

Click on an entry (e.g., "Kafka to Delta Lake - Financial Data") to see its detailed lineage graph.

Utilize: Examine the graph to trace the path of financial data from the raw\_financial\_transactions Kafka topic, through the Spark processing job, to the financial\_data\_delta table in MinIO. This visual confirmation is invaluable for auditing, debugging, and understanding data transformations.

Explore Details: Click on specific nodes (e.g., a dataset or a transformation) within the graph to view metadata such as schema, columns, and applied operations.

6.3. OpenMetadata for Centralized Data Catalog & Active Governance

OpenMetadata acts as a centralized data catalog, consolidating metadata from all your data assets, making them discoverable and governable.

Access OpenMetadata UI: http://localhost:8585.

Explore Data Assets:

Once your Airflow DAGs for OpenMetadata ingestion have run (these should be part of your airflow\_dags and periodically run), you should see your Kafka topics (raw\_financial\_transactions, raw\_insurance\_claims), Delta Lake tables (e.g., financial\_data\_delta, insurance\_data\_delta), PostgreSQL tables, and potentially FastAPI endpoints listed.

Search for Assets: Use the search bar (e.g., search for "financial claim" or "insurance policy").

Inspect Asset Details: Click on a Kafka topic or Delta Lake table.

You should see its schema, sample data, and crucially, a "Lineage" tab that provides detailed lineage information (often integrated from Spline). This shows where the data came from and where it flows.

Explore the "Profiler" tab (if configured) for data quality metrics and column-level statistics.

Utilize for Governance:

Add Descriptions: Click the "Edit" icon next to an asset's description or column description and add a meaningful business description. This directly impacts data discoverability and understanding for other users.

Add Tags: Apply relevant tags (e.g., PII, Financial, Sensitive) to assets or individual columns. This demonstrates how data can be classified for governance and security purposes.

Assign Owners: Assign team members or groups as owners to data assets, clarifying responsibility.

7. Exercising the System with Robust Disparate Data

Beyond just verifying data flow, these steps demonstrate the platform's robustness and flexibility, including preparing data for machine learning.

Introduce Schema Evolution:

Modify src/models/financial\_transaction.py (or the generate\_financial\_transaction function in simulate\_data.py) to add a new, optional field (e.g., is\_flagged: bool = False).

Restart simulate\_data.py.

Ensure your financial Spark streaming job is configured with mergeSchema option for Delta Lake writes.

Verify: Observe in MinIO that the Delta Lake schema for financial\_data\_delta has evolved without breaking. Query the Delta table with Spark to see the new column. Old records will have null for this column, new ones will have data. This shows the system's ability to handle evolving data schemas gracefully.

Simulate Backpressure and Recovery:

While simulate\_data.py is running (generating high volume), temporarily pause the Spark container: docker compose pause spark.

Observe: Watch the Kafka consumer lag in Grafana for both financial and insurance topics. It should rapidly increase as Spark stops consuming.

After a minute or two, resume the Spark container: docker compose unpause spark.

Observe: The Kafka consumer lag should start to decrease, demonstrating Spark's ability to catch up on accumulated backlog and the system's resilience.

Stress Testing (using Locust):

Stop your simulate\_data.py script.

Start Locust: locust -f locust\_fastapi\_ingestor.py --host http://localhost:8000

Access Locust UI at http://localhost:8089.

Start a new test with high user concurrency (e.g., 50-100 users, 10 spawn rate) and a short hatch rate.

Observe in Grafana: Watch FastAPI RPS and latency, and Kafka consumer lag for both topics. This demonstrates how the system performs under sustained heavy load. You can identify potential bottlenecks (e.g., FastAPI's ability to handle concurrent requests, Kafka's ability to buffer, Spark's processing speed).

Integrating with Machine Learning (Conceptual):  
The curated Delta Lake serves as a high-quality data source for machine learning tasks. While a full ML model training/deployment isn't run locally, you can conceptually demonstrate the integration.  
Concept: A Spark job (or a separate Python script with pyspark installed locally) can read from your curated Delta Lake table, prepare features, and then apply a pre-trained ML model or perform a simple training step.  
Conceptual Steps:

Prepare Curated Data: Ensure your batch\_transformations.py (or similar) runs to create curated-data-bucket/financial\_data\_curated. This data would typically be cleaned, aggregated, and modeled for ML.

Run a conceptual Spark ML script: Imagine a pyspark\_jobs/ml\_model\_inference.py script.  
# Conceptual pyspark\_jobs/ml\_model\_inference.py  
from pyspark.sql import SparkSession  
from pyspark.ml.feature import VectorAssembler  
from pyspark.ml.classification import LogisticRegression # Or any other ML model  
  
def create\_spark\_session(app\_name):  
 return (SparkSession.builder.appName(app\_name)  
 .config("spark.jars.packages", "io.delta:delta-core\_2.12:2.4.0")  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
 .getOrCreate())  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 spark = create\_spark\_session("ML\_Inference\_Example")  
 curated\_path = "s3a://curated-data-bucket/financial\_data\_curated" # Point to your curated data  
  
 # Simulate reading curated data  
 # In a real scenario, this would be a well-defined feature set  
 try:  
 df = spark.read.format("delta").load(curated\_path)  
 df.show(5, truncate=False)  
  
 # Conceptual ML preparation  
 # Assuming 'amount' is a feature for a simple model  
 assembler = VectorAssembler(inputCols=["amount"], outputCol="features")  
 feature\_df = assembler.transform(df)  
  
 # Simulate a pre-trained model (or a simple training)  
 # For demonstration, we'll just show structure  
 # model = LogisticRegressionModel.load("path/to/pretrained\_model")  
 # predictions = model.transform(feature\_df)  
 # predictions.show()  
  
 print(f"Successfully read data from {curated\_path} and conceptually prepared for ML.")  
 print("In a real scenario, an ML model would now be applied or trained.")  
  
 except Exception as e:  
 print(f"Error reading curated data for ML: {e}")  
 print("Ensure batch\_transformations.py has run and populated the curated-data-bucket/financial\_data\_curated path.")  
  
 spark.stop()

Run the script: docker exec -it spark spark-submit pyspark\_jobs/ml\_model\_inference.py

Verify: The script successfully reads from the curated Delta Lake, demonstrating that the data platform is providing high-quality, consumable data for advanced analytics and ML applications.

By performing these steps, you will gain a profound understanding of how your enterprise-ready data platform operates, its capabilities in handling diverse data, its resilience, and its comprehensive observability features.

Deep-Dive Addendum: IaC & CI/CD Recipes

This addendum provides detailed insights and practical recipes for implementing Infrastructure as Code (IaC) and Continuous Integration/Continuous Delivery (CI/CD) within your enterprise data platform. These practices are crucial for ensuring maintainability, collaboration, consistency, and automated delivery of your data solutions.

5.1. Project Structure & Infrastructure as Code (IaC)

A well-organized project structure and the adoption of Infrastructure as Code (IaC) are crucial for maintainability, collaboration, and consistent deployments.

Mono-repo Skeleton: A mono-repo approach centralizes all project components, enhancing discoverability and simplifying dependency management.

data-ingestion-platform/  
├── .github/ # GitHub Actions CI/CD workflows  
│ └── workflows/  
│ ├── ci.yml # Continuous Integration pipeline  
│ └── release.yml # Release/Deployment pipeline  
├── data/ # Persistent Docker volumes for all services  
│ ├── postgres/  
│ ├── mongodb/  
│ ├── minio/  
│ ├── spark-events/  
│ ├── grafana/  
│ ├── openmetadata\_mysql/  
│ └── openmetadata\_elasticsearch/  
├── src/ # Core Python application logic (e.g., FastAPI, common utils)  
│ └── common/  
│ └── utils.py  
│ └── models/ # Pydantic/Avro schemas for data contracts  
│ └── financial\_transaction.py  
│ └── insurance\_claim.py  
├── fastapi\_app/ # FastAPI ingestion service  
│ ├── Dockerfile  
│ ├── requirements.txt  
│ └── app/  
│ └── main.py # Entry point for FastAPI app  
│ └── tests/  
│ ├── unit/  
│ │ └── test\_api.py  
│ └── integration/  
│ └── test\_data\_flow.py # Integration tests  
├── pyspark\_jobs/ # Apache Spark transformation jobs (PySpark)  
│ ├── \_\_init\_\_.py  
│ ├── batch\_transformations.py  
│ └── streaming\_consumer.py  
│ └── tests/  
│ └── unit/  
│ └── test\_spark\_logic.py  
├── airflow\_dags/ # Apache Airflow DAG definitions  
│ ├── data\_ingestion\_dag.py  
│ └── data\_transformation\_dag.py  
├── terraform\_infra/ # Infrastructure as Code for cloud deployments  
│ ├── modules/ # Reusable Terraform modules  
│ │ ├── s3\_data\_lake/  
│ │ ├── msk\_kafka/  
│ │ └── rds\_postgres/  
│ ├── environments/ # Environment-specific Terraform configurations  
│ │ ├── dev/  
│ │ │ └── main.tf  
│ │ │ └── variables.tf  
│ │ ├── staging/  
│ │ │ └── main.tf  
│ │ │ └── variables.tf  
│ │ └── prod/  
│ │ └── main.tf  
│ │ └── variables.tf  
├── observability/ # Grafana dashboards, Grafana Alloy configurations, Prometheus rules  
│ ├── alloy-config.river  
│ ├── dashboards/  
│ │ └── health\_dashboard.json  
│ └── grafana\_dashboards\_provisioning/  
│ └── grafana\_datasources\_provisioning/  
├── openmetadata\_ingestion\_scripts/ # Python scripts for OpenMetadata connectors  
├── runbooks/ # Operational Runbooks library  
│ ├── kafka\_consumer\_lag.md  
│ └── spark\_job\_hang.md  
├── conceptual\_code/ # Contains conceptual snippets from document for quick reference  
├── docker-compose.yml # Central Docker Compose file for local environment  
├── docker-compose.test.yml # Docker Compose file for integration testing  
└── README.md

5.2. Security Best Practices & Secrets Management

Security is paramount, especially when handling sensitive financial and insurance data. This section, while broader than just IaC, is included here due to its strong ties to how infrastructure is provisioned and applications are deployed securely via CI/CD.

Data Encryption:

In Transit: All data moving between services within the platform, and especially data ingested via the FastAPI API, should be encrypted using HTTPS/TLS. For Kafka, configure SSL/TLS (e.g., KAFKA\_PROTOCOL: SSL in production).

At Rest: Data stored in all persistence layers (PostgreSQL, MongoDB, MinIO/S3) must be encrypted. Locally, this relies on the host's disk encryption. In cloud environments, managed services (e.g., RDS, S3, DocumentDB) offer encryption at rest.

Secure Credential Management: Hardcoding sensitive information (passwords, API keys, tokens) is a critical vulnerability.

Local Development: Use .env files (added to .gitignore) for environment variables or Docker secrets. Docker secrets are safer as they are mounted as files and not directly exposed as environment variables.  
Example .env (.gitignore it!):  
KAFKA\_BROKER="localhost:9092"  
POSTGRES\_USER="user"  
POSTGRES\_PASSWORD="password"

Production (Cloud): Employ dedicated, enterprise-grade secrets management solutions.

Cloud Secrets Management Comparison:

Real-World Usage Tips:

Vault Agent Sidecar: In containerized environments (Kubernetes, ECS), run a Vault Agent as a sidecar container. It can pull secrets from Vault and render them to a shared volume, making them available to the main application container as files (more secure than env vars).

Rotation Policies: Implement automated secret rotation for database credentials, API keys, etc., to minimize the window of compromise.

Least Privilege: Ensure IAM roles/policies for services accessing secrets managers adhere strictly to the principle of least privilege.

5.3. CI/CD: Automating Quality and Delivery

A robust CI/CD pipeline is essential for automating the software development lifecycle, ensuring code quality, consistency, and rapid, reliable deployments.

Version Control: All code (FastAPI, PySpark, Airflow DAGs, Dockerfiles, IaC) resides in a Git repository.

Automated Build & Test (Continuous Integration - CI):

Trigger: On every code commit/pull request.

Steps: Linting (Black, Flake8), static analysis (SonarQube), unit tests (pytest), Docker image builds.

Automated Deployment (Continuous Delivery/Deployment - CD):

Development/Staging Environments: Automatically deploy validated artifacts for further testing.

Production Deployment: Controlled process with manual approval gates, canary deployments, or blue/green strategies.

Infrastructure as Code (IaC): Manage infrastructure (e.g., cloud resources via Terraform) as code within the Git repository and deploy via CI/CD.

Conceptual GitHub Actions Release Workflow (.github/workflows/release.yml):

This workflow demonstrates building, publishing, testing on staging, and conditionally promoting to production.

# .github/workflows/release.yml  
name: Release Pipeline  
  
on:  
 push:  
 branches:  
 - release # Trigger on pushes to a 'release' branch, or tag pushes  
 workflow\_dispatch: # Allows manual trigger from GitHub UI  
 inputs:  
 version:  
 description: 'Release Version (e.g., v1.0.0)'  
 required: true  
  
jobs:  
 build-and-publish-images:  
 runs-on: ubuntu-latest  
 outputs:  
 fastapi\_image: ${{ steps.build\_fastapi.outputs.image\_name }}  
 pyspark\_image: ${{ steps.build\_pyspark.outputs.image\_name }}  
 steps:  
 - name: Checkout code  
 uses: actions/checkout@v3  
 - name: Set up Docker BuildX  
 uses: docker/setup-buildx-action@v2  
 - name: Log in to Docker Hub (or ECR)  
 uses: docker/login-action@v2  
 with:  
 username: ${{ secrets.DOCKER\_USERNAME }}  
 password: ${{ secrets.DOCKER\_TOKEN }}  
 # For ECR: registry: ${{ secrets.AWS\_ACCOUNT\_ID }}.dkr.ecr.${{ secrets.AWS\_REGION }}.amazonaws.com  
 - name: Build and push FastAPI Ingestor image  
 id: build\_fastapi  
 uses: docker/build-push-action@v4  
 with:  
 context: ./fastapi\_app  
 push: true  
 tags: yourusername/fastapi-ingestor:${{ github.sha }} # Use Git SHA for unique tag  
 # For ECR: tags: ${{ secrets.AWS\_ACCOUNT\_ID }}.dkr.ecr.${{ secrets.AWS\_REGION }}.amazonaws.com/fastapi-ingestor:${{ github.sha }}  
 outputs: type=string,name=image\_name  
 - name: Build and push PySpark Job image (base for running jobs)  
 id: build\_pyspark  
 uses: docker/build-push-action@v4  
 with:  
 context: ./pyspark\_jobs # Assuming a Dockerfile here for PySpark environment  
 push: true  
 tags: yourusername/pyspark-job-runner:${{ github.sha }}  
 outputs: type=string,name=image\_name  
  
 deploy-to-staging:  
 needs: build-and-publish-images  
 runs-on: ubuntu-latest  
 environment: staging # Links to GitHub Environments  
 steps:  
 - name: Checkout code  
 uses: actions/checkout@v3  
 - name: Configure AWS Credentials (for IaC deployment)  
 uses: aws-actions/configure-aws-credentials@v3  
 with:  
 aws-access-key-id: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}  
 aws-secret-access-key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}  
 aws-region: us-east-1  
 - name: Set up Terraform  
 uses: hashicorp/setup-terraform@v2  
 with:  
 terraform\_version: 1.5.0 # Or desired version  
 - name: Terraform Init (Staging)  
 run: terraform -chdir=./terraform\_infra/environments/staging init  
 - name: Terraform Apply (Staging)  
 run: terraform -chdir=./terraform\_infra/environments/staging apply -auto-approve \  
 -var="fastapi\_image\_tag=${{ needs.build-and-publish-images.outputs.fastapi\_image }}" \  
 -var="pyspark\_image\_tag=${{ needs.build-and-publish-images.outputs.pyspark\_image }}"  
 env:  
 TF\_VAR\_environment: staging # Pass environment variable to Terraform  
 - name: Run End-to-End Smoke Tests on Staging  
 # This would involve:  
 # 1. Waiting for staging deployment to complete  
 # 2. Triggering data generation against staging API Gateway  
 # 3. Verifying data in S3/Delta Lake or triggering a Spark job  
 # 4. Checking Grafana/CloudWatch for basic health metrics  
 run: |  
 echo "Running smoke tests on staging environment using deployed API and data lake."  
 # Example: python scripts/run\_smoke\_tests.py --env staging --api-url ${{ secrets.STAGING\_API\_URL }}  
 sleep 60 # Simulate test execution  
 echo "Staging smoke tests passed."  
  
 promote-to-production:  
 needs: deploy-to-staging  
 runs-on: ubuntu-latest  
 environment: production # Links to GitHub Environments, requires manual approval  
 if: success() && github.ref == 'refs/heads/release' # Only promote if staging passed and on release branch  
 steps:  
 - name: Checkout code  
 uses: actions/checkout@v3  
 - name: Configure AWS Credentials (for IaC deployment)  
 uses: aws-actions/configure-aws-credentials@v3  
 with:  
 aws-access-key-id: ${{ secrets.AWS\_PROD\_ACCESS\_KEY\_ID }} # Use production specific credentials  
 aws-secret-access-key: ${{ secrets.AWS\_PROD\_SECRET\_ACCESS\_KEY }}  
 aws-region: us-east-1  
 - name: Set up Terraform  
 uses: hashicorp/setup-terraform@v2  
 with:  
 terraform\_version: 1.5.0  
 - name: Terraform Init (Production)  
 run: terraform -chdir=./terraform\_infra/environments/prod init  
 - name: Terraform Apply (Production)  
 run: terraform -chdir=./terraform\_infra/environments/prod apply -auto-approve \  
 -var="fastapi\_image\_tag=${{ needs.build-and-publish-images.outputs.fastapi\_image }}" \  
 -var="pyspark\_image\_tag=${{ needs.build-and-publish-images.outputs.pyspark\_image }}"  
 env:  
 TF\_VAR\_environment: prod

9.3. Hybrid Testing with LocalStack/ECS-Local

For "hybrid" testing, LocalStack or ECS-Local allows you to interact with local AWS-compatible APIs before full cloud cutover. This is a critical part of a robust CI/CD pipeline, enabling faster feedback loops and reduced cloud spend during development and testing phases.

LocalStack: A cloud service emulator that runs in your local environment.

Benefit: Test cloud service integrations (S3, Lambda, SQS, SNS) without deploying to actual AWS, saving costs and speeding up feedback.

Usage:

Run LocalStack (e.g., via Docker Compose).

Configure your Python boto3 clients to point to LocalStack's endpoint URL (e.g., s3 = boto3.client('s3', endpoint\_url='http://localhost:4566')).

Test your application logic that interacts with these AWS services locally.

ECS-Local: A tool that allows you to test ECS task definitions locally without deploying to AWS.

Benefit: Validate your ECS task definitions, Docker images, and container configurations in a local environment before pushing to Amazon ECS.

Usage: Define your ECS task definitions as you would for AWS. Use the ecs-local CLI to run these tasks locally as Docker containers.

Appendix I: AWS IaC Snippets

This appendix provides conceptual Terraform Infrastructure as Code (IaC) snippets for deploying various components of the data platform on AWS. These snippets demonstrate how the local Docker Compose setup can be translated into production-grade cloud infrastructure, forming a core part of your automated deployment pipeline.

AWS Account and Core Networking Setup:

Prerequisites: Active AWS account, AWS CLI configured, basic familiarity with AWS Console.

IAM (Identity and Access Management): Create necessary IAM roles and policies with least privilege for all services and components (e.g., Lambda execution role, EMR instance profile, MWAA execution role).

VPC (Virtual Private Cloud): Design and create a VPC with public and private subnets. Deploy a NAT Gateway in the public subnet for private subnet resources to access the internet. Configure appropriate Route Tables and Network ACLs.

Security Groups: Create security groups for each service to control inbound and outbound traffic.

Amazon S3 (Data Lake Storage - Replaces MinIO):

# S3 Data Lake Module (terraform\_infra/modules/s3\_data\_lake/main.tf)  
resource "aws\_s3\_bucket" "raw\_data\_bucket" {  
 bucket = "${var.project\_name}-raw-${var.environment}-${var.aws\_region}"  
 tags = {  
 Environment = var.environment  
 Project = var.project\_name  
 ManagedBy = "Terraform"  
 }  
}  
  
resource "aws\_s3\_bucket\_server\_side\_encryption\_configuration" "raw\_data\_bucket\_encryption" {  
 bucket = aws\_s3\_bucket.raw\_data\_bucket.id  
 rule {  
 apply\_server\_side\_encryption\_by\_default {  
 sse\_algorithm = "AES256"  
 }  
 }  
}  
  
resource "aws\_s3\_bucket" "curated\_data\_bucket" {  
 bucket = "${var.project\_name}-curated-${var.environment}-${var.aws\_region}"  
 tags = {  
 Environment = var.environment  
 Project = var.project\_name  
 ManagedBy = "Terraform"  
 }  
}  
  
resource "aws\_s3\_bucket\_server\_side\_encryption\_configuration" "curated\_data\_bucket\_encryption" {  
 bucket = aws\_s3\_bucket.curated\_data\_bucket.id  
 rule {  
 apply\_server\_side\_encryption\_by\_default {  
 sse\_algorithm = "AES256"  
 }  
 }  
}  
  
# Output bucket ARNs  
output "raw\_bucket\_arn" {  
 value = aws\_s3\_bucket.raw\_data\_bucket.arn  
}  
  
output "curated\_bucket\_arn" {  
 value = aws\_s3\_bucket.curated\_data\_bucket.arn  
}

Amazon MSK (Managed Apache Kafka - Replaces Apache Kafka):

# MSK Kafka Cluster Module (terraform\_infra/modules/msk\_kafka/main.tf)  
resource "aws\_msk\_cluster" "main" {  
 cluster\_name = "${var.project\_name}-kafka-${var.environment}"  
 kafka\_version = "2.8.1" # Or latest stable  
 number\_of\_broker\_nodes = var.number\_of\_broker\_nodes  
  
 broker\_node\_group\_info {  
 instance\_type = var.broker\_instance\_type  
 ebs\_volume\_info {  
 provisioned\_throughput = 0 # For smaller clusters, adjust for higher IOPS  
 volume\_size = var.broker\_ebs\_volume\_size # GB  
 }  
 client\_subnets = var.subnet\_ids  
 security\_groups = [var.security\_group\_id]  
 }  
  
 encryption\_info {  
 encryption\_in\_transit {  
 client\_broker = "TLS"  
 in\_cluster = true  
 }  
 # key\_arn = aws\_kms\_key.kafka\_kms.arn # Optional: for KMS encryption at rest  
 }  
  
 open\_monitoring {  
 prometheus {  
 jmx\_exporter {  
 enabled\_in\_broker = true  
 }  
 node\_exporter {  
 enabled\_in\_broker = true  
 }  
 }  
 }  
  
 tags = {  
 Environment = var.environment  
 Project = var.project\_name  
 }  
}  
  
# Output MSK broker endpoints  
output "bootstrap\_brokers\_tls" {  
 value = aws\_msk\_cluster.main.bootstrap\_brokers\_tls  
}

AWS Lambda + Amazon API Gateway (FastAPI Replacement):

# Lambda API Ingestor Module (terraform\_infra/modules/lambda\_api\_ingestor/main.tf)  
resource "aws\_ecr\_repository" "fastapi\_repo" {  
 name = "${var.project\_name}/fastapi-ingestor"  
}  
  
# IAM Role for Lambda function  
resource "aws\_iam\_role" "lambda\_exec\_role" {  
 name = "${var.project\_name}-lambda-fastapi-exec-role-${var.environment}"  
 assume\_role\_policy = jsonencode({  
 Version = "2012-10-17"  
 Statement = [{  
 Action = "sts:AssumeRole"  
 Effect = "Allow"  
 Principal = {  
 Service = "lambda.amazonaws.com"  
 }  
 }]  
 })  
}  
  
resource "aws\_iam\_role\_policy\_attachment" "lambda\_basic\_exec" {  
 role = aws\_iam\_role.lambda\_exec\_role.name  
 policy\_arn = "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"  
}  
  
resource "aws\_iam\_role\_policy\_attachment" "lambda\_vpc\_access" {  
 role = aws\_iam\_role.lambda\_exec\_role.name  
 policy\_arn = "arn:aws:iam::aws:policy/service-role/AWSLambdaVPCAccessExecutionRole"  
}  
  
# Policy to allow Lambda to publish to MSK (example)  
resource "aws\_iam\_policy" "lambda\_msk\_publish" {  
 name = "${var.project\_name}-lambda-msk-publish-policy-${var.environment}"  
 policy = jsonencode({  
 Version = "2012-10-17"  
 Statement = [{  
 Action = [  
 "kafka-action:DescribeCluster",  
 "kafka-action:GetBootstrapBrokers",  
 "kafka-action:GetTopicPartitions",  
 "kafka-action:ListTopics",  
 "kafka-action:Produce"  
 ]  
 Effect = "Allow"  
 Resource = var.msk\_cluster\_arn  
 }]  
 })  
}  
  
resource "aws\_iam\_role\_policy\_attachment" "lambda\_msk\_publish\_attach" {  
 role = aws\_iam\_role.lambda\_exec\_role.name  
 policy\_arn = aws\_iam\_policy.lambda\_msk\_publish.arn  
}  
  
resource "aws\_lambda\_function" "fastapi\_ingestor\_lambda" {  
 function\_name = "${var.project\_name}-fastapi-ingestor-${var.environment}"  
 package\_type = "Image"  
 image\_uri = "${aws\_ecr\_repository.fastapi\_repo.repository\_url}:${var.fastapi\_image\_tag}"  
 role = aws\_iam\_role.lambda\_exec\_role.arn  
 timeout = 30 # seconds  
 memory\_size = 512 # MB  
 vpc\_config {  
 subnet\_ids = var.subnet\_ids  
 security\_group\_ids = [var.security\_group\_id]  
 }  
 environment {  
 variables = {  
 KAFKA\_BROKER\_ADDRESSES = var.msk\_bootstrap\_brokers\_tls # From MSK output  
 KAFKA\_TOPIC = var.kafka\_topic\_name  
 # ... other FastAPI env vars  
 }  
 }  
 tags = {  
 Environment = var.environment  
 Project = var.project\_name  
 }  
}  
  
resource "aws\_apigatewayv2\_api" "http\_api" {  
 name = "${var.project\_name}-fastapi-http-api-${var.environment}"  
 protocol\_type = "HTTP"  
}  
  
resource "aws\_apigatewayv2\_integration" "lambda\_integration" {  
 api\_id = aws\_apigatewayv2\_api.http\_api.id  
 integration\_type = "AWS\_PROXY"  
 integration\_method = "POST"  
 integration\_uri = aws\_lambda\_function.fastapi\_ingestor\_lambda.invoke\_arn  
}  
  
resource "aws\_apigatewayv2\_route" "ingest\_financial" {  
 api\_id = aws\_apigatewayv2\_api.http\_api.id  
 route\_key = "POST /ingest-financial-transaction"  
 target = "integrations/${aws\_apigatewayv2\_integration.lambda\_integration.id}"  
}  
  
resource "aws\_apigatewayv2\_route" "ingest\_insurance" {  
 api\_id = aws\_apigatewayv2\_api.http\_api.id  
 route\_key = "POST /ingest-insurance-claim"  
 target = "integrations/${aws\_apigatewayv2\_integration.lambda\_integration.id}"  
}  
  
resource "aws\_apigatewayv2\_stage" "default" {  
 api\_id = aws\_apigatewayv2\_api.http\_api.id  
 name = "$default"  
 auto\_deploy = true  
}  
  
resource "aws\_lambda\_permission" "apigateway\_lambda\_permission" {  
 statement\_id = "AllowAPIGatewayInvoke"  
 action = "lambda:InvokeFunction"  
 function\_name = aws\_lambda\_function.fastapi\_ingestor\_lambda.function\_name  
 principal = "apigateway.amazonaws.com"  
 # The /\*/\* part is to allow all API Gateway methods  
 # to invoke the Lambda  
 source\_arn = "${aws\_apigatewayv2\_api.http\_api.execution\_arn}/\*/\*"  
}  
  
output "api\_gateway\_url" {  
 value = aws\_apigatewayv2\_api.http\_api.api\_endpoint  
}

Amazon RDS for PostgreSQL (Relational Database - Replaces local PostgreSQL):

# RDS PostgreSQL Module (terraform\_infra/modules/rds\_postgres/main.tf)  
resource "aws\_db\_instance" "main" {  
 identifier = "${var.project\_name}-postgres-${var.environment}"  
 engine = "postgres"  
 engine\_version = "15.3"  
 instance\_class = var.instance\_class  
 allocated\_storage = var.allocated\_storage\_gb  
 storage\_type = "gp2" # Or gp3 for higher performance  
 db\_name = var.db\_name  
 username = var.db\_username  
 password = var.db\_password # Use AWS Secrets Manager in production!  
 port = 5432  
 vpc\_security\_group\_ids = [var.security\_group\_id]  
 db\_subnet\_group\_name = var.db\_subnet\_group\_name # Must be created separately  
 skip\_final\_snapshot = var.skip\_final\_snapshot  
 multi\_az = var.multi\_az\_enabled # True for production  
 publicly\_accessible = false  
  
 tags = {  
 Environment = var.environment  
 Project = var.project\_name  
 }  
}  
  
output "rds\_endpoint" {  
 value = aws\_db\_instance.main.address  
}

Amazon DocumentDB (MongoDB Compatible Database - Replaces local MongoDB):

Creation steps via Console or AWS CLI. Terraform resources aws\_docdb\_cluster, aws\_docdb\_cluster\_instance would be used.

Amazon EMR or AWS Glue (Spark Replacement):

Option A: Amazon EMR (Managed Spark Clusters) - Conceptual EMR Cluster Definition:

# EMR Cluster Module  
resource "aws\_emr\_cluster" "spark\_cluster" {  
 name = "${var.project\_name}-spark-cluster-${var.environment}"  
 release\_label = "emr-6.9.0" # Or latest stable  
 applications = ["Spark"]  
  
 ec2\_attributes {  
 subnet\_id = var.subnet\_id  
 instance\_profile = aws\_iam\_instance\_profile.emr\_profile.arn  
 emr\_managed\_master\_security\_group = var.master\_sg\_id  
 emr\_managed\_slave\_security\_group = var.slave\_sg\_id  
 }  
  
 master\_instance\_group {  
 instance\_type = var.master\_instance\_type  
 instance\_count = 1  
 }  
  
 core\_instance\_group {  
 instance\_type = var.core\_instance\_type  
 instance\_count = var.core\_instance\_count  
 }  
  
 configurations\_json = jsonencode([  
 {  
 Classification = "spark-defaults",  
 Properties = {  
 "spark.jars.packages" = "io.delta:delta-core\_2.12:2.4.0,org.apache.spark:spark-sql-kafka-0-10\_2.12:3.5.0",  
 "spark.sql.extensions" = "io.delta.sql.DeltaSparkSessionExtension",  
 "spark.sql.catalog.spark\_catalog" = "org.apache.spark.sql.delta.catalog.DeltaCatalog",  
 "spark.hadoop.fs.s3a.endpoint" = "s3.${var.aws\_region}.amazonaws.com" # Ensure S3 is used  
 }  
 },  
 # ... other configurations for Kafka connectivity etc.  
 ])  
  
 step\_concurrency\_level = 1 # For sequential steps  
  
 tags = {  
 Environment = var.environment  
 Project = var.project\_name  
 }  
}  
  
# Add steps (e.g., PySpark job execution) via aws\_emr\_cluster\_step resource

Option B: AWS Glue (Serverless Spark ETL) - Conceptual Glue ETL Job Definition:

# Glue ETL Job Module  
resource "aws\_glue\_job" "spark\_transform\_job" {  
 name = "${var.project\_name}-spark-transform-${var.environment}"  
 role\_arn = var.glue\_execution\_role\_arn  
 command {  
 name = "glueetl"  
 script\_location = "s3://${var.glue\_scripts\_bucket}/pyspark\_jobs/data\_transformer\_spark.py"  
 python\_version = "3"  
 }  
 default\_arguments = {  
 "--extra-jars" = "s3://delta-lake/delta-core\_2.12-2.4.0.jar" # Or from a public Maven repo  
 "--additional-python-modules" = "delta-spark==2.4.0"  
 "--job-bookmark-option" = "job-bookmark-enable" # To track processed data  
 "--TempDir" = "s3://${var.glue\_temp\_bucket}/temp/"  
 "--source\_kafka\_topic" = var.kafka\_topic\_name  
 "--kafka\_broker\_address" = var.msk\_bootstrap\_brokers\_tls  
 "--raw\_delta\_path" = "s3a://${var.raw\_bucket\_name}/"  
 "--curated\_delta\_path" = "s3a://${var.curated\_bucket\_name}/"  
 }  
 glue\_version = "4.0" # Or desired version (Spark 3.3)  
 number\_of\_workers = var.number\_of\_glue\_workers # DPUs \* 2 for worker type Standard  
 worker\_type = "G.1X" # Or G.2X, Standard  
 timeout = 60 # minutes  
 tags = {  
 Environment = var.environment  
 Project = var.project\_name  
 }  
}  
  
# You would then create aws\_glue\_trigger resources to schedule or event-drive this job.

Amazon MWAA (Managed Workflows for Apache Airflow):

Creation via Console or Terraform resources aws\_mwaa\_environment.

AWS Observability (ADOT, X-Ray, CloudWatch):

Managed services automatically integrate or can be configured via Lambda layers and ECS task definitions.

Amazon Managed Grafana:

Workspace creation and data source linking.

Data Lineage & Cataloging (Spline, OpenMetadata):

Deployment on EC2/ECS with RDS/OpenSearch for backends. OpenMetadata ingestion workflows configured to pull metadata from Glue Data Catalog, MSK, Spline, and CloudWatch.

Deep-Dive Addendum: Cloud Migration + Terraform Snippets

This addendum guides you through the process of migrating your locally developed enterprise data platform to a cloud environment, specifically AWS. It emphasizes a structured approach, service-by-service replacements, and provides conceptual Terraform Infrastructure as Code (IaC) snippets to automate the cloud deployment.

9.1. Overview of AWS Service Replacements

The local development environment, powered by Docker Compose, provides a powerful simulation of a production-grade data platform. When migrating to AWS, each open-source component has a corresponding managed AWS service that offers scalability, reliability, and reduced operational overhead.

9.2. Step-by-Step AWS Migration Guide with IaC Examples

This guide outlines a phased approach to migrating your data platform to AWS, emphasizing the use of Terraform for automated infrastructure provisioning.

Phase 1: Core Networking & IAM Setup

Before deploying any services, establish the foundational network and security components. These are typically set up once per AWS account/region or per environment.

Create a VPC: A Virtual Private Cloud (VPC) provides an isolated network environment.

Define public subnets (for NAT Gateway, Bastion hosts if needed) and private subnets (for most data services).

Configure Route Tables, Internet Gateway (for public subnets), and NAT Gateway (for private subnets to access the internet).

Establish IAM Roles and Policies: Create specific IAM roles with the least privilege necessary for each AWS service (e.g., a role for Lambda to access MSK, a role for Glue to access S3).

Conceptual Terraform for VPC and Subnets:

# terraform\_infra/modules/vpc/main.tf  
resource "aws\_vpc" "main" {  
 cidr\_block = var.vpc\_cidr\_block  
 enable\_dns\_hostnames = true  
 enable\_dns\_support = true  
 tags = {  
 Name = "${var.project\_name}-${var.environment}-vpc"  
 Environment = var.environment  
 }  
}  
  
resource "aws\_internet\_gateway" "main" {  
 vpc\_id = aws\_vpc.main.id  
 tags = {  
 Name = "${var.project\_name}-${var.environment}-igw"  
 }  
}  
  
resource "aws\_subnet" "public" {  
 count = length(var.public\_subnet\_cidrs)  
 vpc\_id = aws\_vpc.main.id  
 cidr\_block = var.public\_subnet\_cidrs[count.index]  
 availability\_zone = data.aws\_availability\_zones.available.names[count.index]  
 map\_public\_ip\_on\_launch = true  
 tags = {  
 Name = "${var.project\_name}-${var.environment}-public-subnet-${count.index}"  
 Environment = var.environment  
 }  
}  
  
resource "aws\_subnet" "private" {  
 count = length(var.private\_subnet\_cidrs)  
 vpc\_id = aws\_vpc.main.id  
 cidr\_block = var.private\_subnet\_cidrs[count.index]  
 availability\_zone = data.aws\_availability\_zones.available.names[count.index]  
 tags = {  
 Name = "${var.project\_name}-${var.environment}-private-subnet-${count.index}"  
 Environment = var.environment  
 }  
}  
  
# ... (NAT Gateway, Route Tables, Security Groups would follow)

Phase 2: Data Lake (S3) and Core Databases (RDS, DocumentDB)

Start with the data persistence layers, as they are fundamental for storing data.

Amazon S3: Replaces MinIO. Create S3 buckets for raw, curated, and any other data zones. Enable versioning, encryption, and cross-region replication for DR.

See Terraform Snippet in IaC & CI/CD Addendum, Appendix I.

Amazon RDS for PostgreSQL: Replaces local PostgreSQL. Deploy a managed PostgreSQL instance for structured data, application metadata, and Airflow metastore. Configure multi-AZ for high availability.

See Terraform Snippet in IaC & CI/CD Addendum, Appendix I.

Amazon DocumentDB: Replaces local MongoDB. Provision a DocumentDB cluster for MongoDB-compatible workloads.

Phase 3: Streaming & Ingestion Layer (MSK, Lambda + API Gateway)

Migrate the real-time ingestion and streaming components.

Amazon MSK: Replaces Apache Kafka. Provision a managed Kafka cluster. Configure Kafka topics, retention policies, and security settings (TLS, IAM authentication).

See Terraform Snippet in IaC & CI/CD Addendum, Appendix I.

AWS Lambda + Amazon API Gateway: Replaces FastAPI. Containerize your FastAPI application into a Docker image, push to ECR, and deploy as a Lambda function triggered by API Gateway. This provides a scalable, serverless ingestion endpoint.

See Terraform Snippet in IaC & CI/CD Addendum, Appendix I.

Phase 4: Data Processing (Glue or EMR)

Transition your Spark batch and streaming jobs to a managed AWS service.

AWS Glue: A serverless ETL service that supports Spark. Ideal for transient jobs and where you prefer to avoid managing clusters.

See Terraform Snippet in IaC & CI/CD Addendum, Appendix I.

Amazon EMR: Managed clusters for Hadoop, Spark, Hive, etc. Offers more control over the cluster environment.

See Terraform Snippet in IaC & CI/CD Addendum, Appendix I.

Phase 5: Orchestration, Governance & Observability (MWAA, Managed Grafana, OpenMetadata)

Deploy the control plane and monitoring solutions.

Amazon MWAA: Replaces Apache Airflow. Deploy a managed Airflow environment. Migrate your DAGs to an S3 bucket connected to MWAA.

See Terraform Snippet in IaC & CI/CD Addendum, Appendix I.

Amazon Managed Grafana: Replaces local Grafana. Create a Grafana workspace and connect it to CloudWatch, Prometheus (via ADOT), and other data sources.

See Terraform Snippet in IaC & CI/CD Addendum, Appendix I.

Data Lineage & Cataloging (OpenMetadata): Deploy OpenMetadata on EC2/ECS with managed databases (RDS/OpenSearch). Configure connectors to pull metadata from Glue Data Catalog, MSK, and other sources.

Phase 6: CI/CD Pipeline Update

Modify your CI/CD pipelines (e.g., GitHub Actions) to deploy to AWS using Terraform.

Update Terraform Configuration: Ensure your Terraform code points to the correct AWS regions, account IDs, and uses appropriate variables for different environments (dev, staging, prod).

Integrate AWS CLI/Terraform in CI/CD: Configure GitHub Actions (or your chosen CI/CD tool) with AWS credentials to run terraform apply for deploying infrastructure and docker push to ECR.

See Conceptual GitHub Actions Workflow in IaC & CI/CD Addendum, Section 5.3.

Cloud Migration Considerations:

Cost Management: AWS services are pay-as-you-go. Monitor costs closely using AWS Cost Explorer and implement budget alerts. Optimize resource sizing.

Security: Leverage AWS IAM for granular access control, VPC security groups, network ACLs, and AWS Key Management Service (KMS) for encryption. Integrate AWS Secrets Manager.

Monitoring & Logging: Utilize CloudWatch Logs, Metrics, and Alarms. Implement AWS Distro for OpenTelemetry (ADOT) for comprehensive tracing and metric collection.

Data Migration: For existing data, plan a data migration strategy (e.g., AWS DataSync, S3 Transfer Acceleration, AWS Database Migration Service (DMS)).

Testing in Cloud: Thoroughly test each migrated component and the end-to-end data pipelines in the AWS environment before going live.

Progressive Rollout: Consider a phased rollout (e.g., migrating one critical pipeline first) rather than a "big bang" approach.

DR Planning in Cloud: AWS provides services (Multi-AZ, Cross-Region Replication, Snapshots) that simplify DR. Integrate these into your DR strategy, defining cloud-native RPOs and RTOs.

Building Enterprise-Ready Data Platforms: Core Handbook

This core handbook provides a high-level overview of building enterprise-ready data platforms, focusing on the fundamental principles, architectural choices, and the progressive path to a robust data ecosystem. It's designed for busy engineers and executives who need a concise understanding of the platform's purpose, structure, and key decision points without wading through exhaustive technical details.

1. Purpose and Introduction

This guide is meticulously crafted for experienced Data Engineers and Senior Software Engineers tasked with modernizing enterprise data ingestion stages. It provides a practical, hands-on approach to building a robust local development environment that mirrors a scalable, production-grade data platform. The focus is on developing Python-based ETL pipelines for disparate data sources, including simple financial and insurance data, emphasizing modern architectural patterns and best practices.

1.1. Why a Robust Local Environment?

A robust local development environment is paramount for building enterprise-ready data platforms. It enables rapid iteration, extensive testing, and critical skill development without incurring cloud costs or dependencies. This approach significantly de-risks subsequent cloud deployments, accelerates development cycles, and allows engineers to experiment with complex distributed systems in a controlled, isolated setting.

From a business perspective, this translates directly into:

Faster Time-to-Value: Rapid prototyping and local testing accelerate the delivery of new data products and features.

Reduced Cloud Spend: Significant cost savings during the development and testing phases by minimizing reliance on expensive cloud resources.

Enhanced Audit Trails & Compliance Readiness: A controlled environment facilitates the implementation and testing of governance features from day one, bolstering compliance efforts.

For more in-depth coverage of testing strategies, refer to the Testing & Observability Patterns Deep-Dive Addendum. For details on automating deployments, see the IaC & CI/CD Recipes Deep-Dive Addendum.

1.2. The Progressive Complexity Path

To avoid the "all-or-nothing" overwhelm often associated with complex data platforms, this guide introduces a "Progressive Complexity" path. Engineers can ramp up feature-by-feature, mastering core components before integrating more advanced elements. This structured approach:

Reduces Cognitive Load: By introducing components incrementally, engineers can focus on understanding one set of interactions at a time.

Accelerates Learning: Hands-on experience with foundational elements provides a solid base for more complex systems.

Facilitates Skill Development: Engineers can gradually build expertise across the entire data platform stack.

Enables Flexible Development: Teams can choose the appropriate track based on their current project needs and scale requirements.

The tracks are designed as follows:

Starter Track: Focuses on a minimal, single-machine setup for foundational data ingestion and storage. Perfect for initial prototyping and simple use cases.

Intermediate Track: Introduces real-time streaming capabilities with Apache Kafka and distributed processing with Apache Spark. Ideal for addressing real-time data needs and scaling transformations.

Advanced Track: Integrates the full suite of tools for comprehensive orchestration, lineage, observability, and metadata management, culminating in a production-grade local environment. This is for building highly robust and governable data platforms.

For a detailed setup guide for each of these tracks, refer to the Progressive Path Setup Guide Deep-Dive Addendum.

1.3. Embracing the Modern Data Engineer Role

Data ingestion represents the critical first step in any data-driven organization, transforming raw, disparate data into actionable insights. This document explores both ETL (Extract, Transform, Load) and ELT (Extract, Load, Transform) methodologies, highlighting their application in different scenarios. By replicating a production-like setting locally—simulating AWS Lambdas with SAM, alongside distributed components like Apache Spark and Apache Kafka—this guide empowers practitioners to ensure smooth transitions and reduced friction when deploying solutions to the cloud.

Furthermore, the comprehensive scope of this guide, encompassing Python, Docker, distributed systems, observability, data lineage, and machine learning (ML) elements, reflects a significant evolution in the role of a data engineer. This expanded scope moves beyond traditional data movement tasks to encompass broader responsibilities in system design, operational excellence, and leveraging advanced analytics. The task of rewriting a data ingestion stage for a large company, involving several disparate data sources, inherently implies a need for handling complexity and scale. This guide aims to empower the practitioner to demonstrate this expanded skill set, beneficial not just for platform builders but also for data scientists and analysts who will consume data from it. For a deeper dive into these areas, please refer to the respective deep-dive addendums: IaC & CI/CD Recipes, Testing & Observability Patterns, DR & Runbooks, and Cloud Migration + Terraform Snippets.

2. Executive Summary: Platform Pitch & ROI

This section provides a high-level overview of the proposed data platform, designed for internal pitching to stakeholders, highlighting key benefits and estimated returns on investment.

Building an Enterprise-Ready Data Platform: Strategic Imperative

Problem: Our current data infrastructure struggles with diverse, high-volume data ingestion, real-time analytics, and comprehensive data governance, leading to slow insights, high operational costs, and compliance risks.

Solution: Implement a modern, scalable, and observable data platform leveraging open-source technologies for a robust local development environment, seamlessly transitioning to a cost-efficient cloud-native architecture.

Key Business Benefits & ROI:

For more details on CI/CD benefits, refer to the IaC & CI/CD Recipes Deep-Dive Addendum. For insights into data quality, operational efficiency, and observability, see the Testing & Observability Patterns Deep-Dive Addendum. For information on disaster recovery and MTTR, consult the DR & Runbooks Deep-Dive Addendum.

Project Milestones (Conceptual):

Q3 202X: Establish core local dev environment (Starter & Intermediate Tracks).

Q4 202X: Implement full Advanced Track locally, complete initial CI/CD pipelines.

Q1 202Y: Pilot AWS migration for a critical data ingestion pipeline.

Q2 202Y: Full production rollout on AWS.

For detailed guidance on cloud migration, see the Cloud Migration + Terraform Snippets Deep-Dive Addendum.

Ask: Secure resources for dedicated engineering focus to implement this strategic platform modernization, unlocking significant business value and long-term capabilities.

3. The Progressive Path to an Enterprise Data Platform

This section details the step-by-step approach to building the local data platform, starting simple and progressively adding complexity. Each track builds upon the previous one. For a detailed setup guide for each of these tracks, including docker-compose.yml configurations, refer to the Progressive Path Setup Guide Deep-Dive Addendum.

3.1. Starter Track: Minimal Single-Machine Setup

The starter track provides the bare essentials for data ingestion and structured storage, ideal for rapid prototyping and understanding fundamental data flow. This minimal setup requires low computational resources and serves as an excellent entry point for engineers new to the platform.

Components:

FastAPI: A lightweight, high-performance web framework for building data ingestion APIs.

PostgreSQL: A robust relational database for structured data and API-specific metadata.

MinIO (as File-based Delta Lake): An S3-compatible object storage server, simulating a data lake where immutable Delta Lake files reside.

Key Learnings:

API Design: How to create secure and well-documented endpoints for data reception using FastAPI.

Database Interaction: Storing and retrieving structured data efficiently with PostgreSQL.

Object Storage Basics: Understanding the S3-compatible interface for local data lake operations with MinIO.

Containerization Fundamentals: Running individual services in isolated Docker containers.

Direct Storage Patterns: Simple ETL/ELT patterns where data is written directly to a database or object store.

3.2. Intermediate Track: Adding Streaming Capabilities

This track expands the platform to handle real-time data streams and distributed transformations. It introduces two powerful, industry-standard components that form the backbone of many modern data architectures.

Components (in addition to Starter):

Apache Kafka: A distributed streaming platform for high-throughput, fault-tolerant real-time data ingestion and event streaming. It decouples producers from consumers.

Apache Spark: A powerful, distributed processing engine for large-scale data transformations, supporting both batch and streaming workloads. It will consume data from Kafka and write to Delta Lake.

Key Learnings:

Asynchronous Ingestion: Decoupling producers and consumers using a message broker like Kafka for resilience and scalability.

Distributed Stream Processing: Consuming from Kafka and writing to Delta Lake with Spark Structured Streaming, enabling near real-time data pipelines.

Data Lakehouse Concepts: Implementing ACID transactions, schema enforcement, and time travel capabilities with Delta Lake on object storage.

Scaling Data Pipelines: Understanding the basics of distributed systems and how Spark partitions and processes data across workers.

3.3. Advanced Track: The Full Production-Ready Stack

The advanced track integrates robust solutions for orchestration, observability, lineage, and metadata management, simulating a comprehensive enterprise-grade platform. This track represents the complete vision for the local development environment, providing all the tools necessary for building, monitoring, and governing complex data ecosystems.

Components (in addition to Intermediate):

Apache Airflow: Workflow orchestrator for scheduling and managing complex data pipelines and their dependencies.

OpenTelemetry & Grafana Alloy: Standardized telemetry collection and forwarding, enabling comprehensive monitoring.

Grafana: Interactive data visualization and monitoring dashboards for operational insights.

Spline: Automated data lineage tracking specifically for Spark jobs, providing visibility into data transformations.

OpenMetadata: Comprehensive metadata management and data cataloging, consolidating information from various sources.

MongoDB: A flexible NoSQL document database, suitable for semi-structured data or specific application use cases requiring schema flexibility.

cAdvisor: Container resource usage and performance analysis agent, providing metrics for Grafana.

Key Learnings:

Orchestration Mastery: Managing complex workflows and dependencies with Airflow, including scheduling Spark jobs and metadata ingestion tasks.

End-to-End Observability: Gaining deep insights into system health, performance, and bottlenecks using OpenTelemetry, Grafana Alloy, Grafana, and cAdvisor. For detailed patterns, refer to the Testing & Observability Patterns Deep-Dive Addendum.

Data Lineage & Governance: Tracking data transformations with Spline and providing a unified data catalog for discovery, understanding, and compliance with OpenMetadata.

Comprehensive Data Management: Integrating diverse data stores (relational, NoSQL, object storage) and tools for a holistic, enterprise-ready data platform.

4. Foundational Architecture & Core Technologies

This section provides a concise, high-level overview of the platform's architecture and the core technologies integrated into the local data platform, outlining each component's primary function and contribution to the overall scalable system. Docker Compose is the pivotal tool for managing the interdependencies and orchestration of this complex local data stack, simplifying the simulation of distributed systems. For the full docker-compose.yml and project structure, refer to the IaC & CI/CD Recipes Deep-Dive Addendum.

The proposed architecture transforms data pipelines into a scalable, distributed system, adopting the "data lakehouse" paradigm. By leveraging Delta Lake as the primary storage layer, the architecture creates a unified solution for both raw and curated data, simplifying the system, reducing redundancy, minimizing data movement, and ensuring data consistency. The introduction of Apache Kafka and Spark Structured Streaming addresses the need for real-time analytics, critical for immediate analysis in security, financial, or insurance scenarios. The decoupling of ingestion from storage via Kafka significantly improves the resilience and availability of the ingestion layer by buffering events and preventing backpressure.

4.1. Architectural Overview

The platform is logically divided into several layers:

Ingestion Layer: The entry point for all raw data, handling external data sources and publishing to a streaming buffer.

Processing Layer: Where data is transformed, cleansed, validated, and modeled using distributed computing.

Storage Layer (Data Lakehouse): The unified, reliable repository for all data states (raw, curated), providing ACID properties and flexible schema management.

Analytical Layer: Facilitates querying, reporting, and advanced analytics, including machine learning model training and inference.

Orchestration & Governance Layer: Manages workflow scheduling, ensures data quality, provides end-to-end observability, and offers a centralized data catalog with lineage capabilities.

For a detailed mapping of local components to AWS cloud services, refer to the Cloud Migration + Terraform Snippets Deep-Dive Addendum.

Here is a PlantUML diagram illustrating the architectural overview:

@startuml  
!theme toy  
skinparam componentStyle uml2  
  
' Define Actors/External Systems  
actor "Disparate Data Sources\n(e.g., Financial, Insurance Systems)" as data\_sources  
  
' Define Layers/Zones  
rectangle "Ingestion Layer" {  
 component "FastAPI Ingestor" as fastapi\_ingestor  
 queue "Apache Kafka\n(Raw Data Topic)" as kafka\_topic  
}  
  
rectangle "Processing Layer" {  
 component "Apache Spark Cluster" as spark\_cluster  
 rectangle "Spark Structured Streaming\n(Raw Data Consumer)" as spark\_raw\_consumer  
 rectangle "PySpark Transformation Job\n(ELT/Batch)" as spark\_transform  
 spark\_cluster -- spark\_raw\_consumer  
 spark\_cluster -- spark\_transform  
}  
  
rectangle "Storage Layer (Data Lakehouse)" {  
 database "MinIO (S3 Compatible)\n(Delta Lake Raw Zone)" as minio\_raw  
 database "MinIO (S3 Compatible)\n(Delta Lake Curated Zone)" as minio\_curated  
 database "PostgreSQL\n(Structured Data/Metadata)" as postgres\_db  
 database "MongoDB\n(Semi-Structured Data)" as mongodb\_db  
 minio\_raw <--> minio\_curated : "Delta Lake"  
}  
  
rectangle "Orchestration & Governance Layer" {  
 cloud "Apache Airflow" as airflow  
 component "OpenTelemetry" as opentelemetry  
 component "Grafana Alloy\n(OTLP Collector)" as grafana\_alloy  
 database "OpenMetadata\n(Data Catalog)" as openmetadata  
 component "Spline\n(Spark Lineage)" as spline  
 component "Grafana\n(Monitoring & Visualization)" as grafana  
 component "cAdvisor\n(Container Metrics)" as cadvisor  
}  
  
rectangle "Analytical Layer" {  
 component "Spark SQL / MLlib Analytics" as spark\_analytics  
}  
  
' Data Flow  
data\_sources --> fastapi\_ingestor : "Send Data (HTTP/S)"  
fastapi\_ingestor --> kafka\_topic : "Publish Data (JSON/Protobuf)"  
kafka\_topic --> spark\_raw\_consumer : "Consume Stream"  
spark\_raw\_consumer --> minio\_raw : "Write to Raw Zone"  
minio\_raw --> spark\_transform : "Read Raw Data"  
spark\_transform --> minio\_curated : "Write Curated Data (MERGE)"  
minio\_curated --> spark\_analytics : "Query for Analytics"  
postgres\_db <--> spark\_transform : "Dim Data / Metadata"  
mongodb\_db <--> spark\_transform : "Semi-Structured Data"  
spark\_analytics --> data\_sources : "Insights/Reports"  
  
' Observability Flow  
opentelemetry --> grafana\_alloy : "Telemetry Data (Traces, Metrics, Logs)"  
fastapi\_ingestor .. opentelemetry : "Instrumented"  
spark\_cluster .. opentelemetry : "Instrumented"  
airflow .. opentelemetry : "Instrumented"  
cadvisor --> grafana\_alloy : "Container Metrics"  
grafana\_alloy --> grafana : "Forward to Grafana"  
grafana\_alloy --> openmetadata : "Forward Metadata/Telemetry"  
spark\_cluster --> spline : "Capture Lineage"  
spline --> openmetadata : "Send Lineage Metadata"  
airflow --> spark\_cluster : "Orchestrate Jobs"  
airflow --> openmetadata : "Orchestrate Metadata Ingestion"  
openmetadata <--> grafana : "Share Metadata/Context"  
@enduml

4.2. Core Technology Deep Dive

The following summarizes the key technologies and their roles within this platform, directly correlating to the services defined in the docker-compose.yml. For the complete docker-compose.yml and detailed setup instructions, refer to the IaC & CI/CD Recipes Deep-Dive Addendum. For cloud-native replacements and their Terraform snippets, see the Cloud Migration + Terraform Snippets Deep-Dive Addendum.

Apache Airflow: Workflow orchestrator for scheduling, monitoring, and managing complex data pipelines and dependencies, including Spark jobs. Provides a robust framework for defining complex data pipelines as Directed Acyclic Graphs (DAGs).

Apache Kafka: A distributed streaming platform designed for building real-time data pipelines and streaming applications. It serves as a durable buffer for raw event streams, decoupling ingestion from downstream processing.

Apache Spark: A powerful, distributed processing engine for large-scale data transformations (ELT), supporting both batch and streaming workloads with PySpark. It reads from Kafka and Delta Lake.

AWS SAM CLI: (Serverless Application Model Command Line Interface) Enables local development and testing of AWS Lambda functions, simulating the serverless environment.

cAdvisor: (Container Advisor) A running daemon that collects, aggregates, processes, and exports information about running containers, providing performance metrics to Grafana.

Delta Lake: An open-source storage layer that brings ACID transactions, schema enforcement, and time travel capabilities to data lakes, unifying batch and streaming data processing within Spark.

Docker/Docker Compose: Essential for containerization and orchestration of all services in this local development environment, ensuring isolated, reproducible, and portable environments.

FastAPI: A modern, high-performance web framework for building data ingestion APIs with Python 3.7+, offering automatic interactive documentation (Swagger UI). It acts as a Kafka producer.

Grafana Alloy: An OpenTelemetry Collector distribution that is highly configurable and optimized for collecting, processing, and exporting telemetry data (metrics, logs, traces). It acts as a central hub for observability data.

Grafana: An open-source platform for interactive data visualization and monitoring. It is used to create dashboards and visualize metrics and traces.

MinIO: An open-source object storage server that is compatible with Amazon S3 APIs. It simulates an S3-compatible data lake locally.

MongoDB: A popular open-source NoSQL document database. It provides flexible storage for semi-structured data.

OpenMetadata: An open-source metadata management platform that provides a unified data catalog, data lineage, and data quality capabilities, enabling data discovery and governance.

OpenTelemetry: A set of open-source tools, APIs, and SDKs that standardize the collection and export of telemetry data (metrics, logs, and traces) from software applications.

PostgreSQL: A powerful, open-source object-relational database system, serving as a robust SQL datastore for structured data, reference data, and the metadata database for Apache Airflow.

Python: The primary programming language for all ETL pipelines, APIs, scripting, and machine learning components.

Spline: An open-source tool specifically designed for automated data lineage tracking within Apache Spark jobs. It captures metadata about Spark transformations and provides a UI for visualizing data flow.

4.3. Decision Frameworks for Technology Choices

Choosing the right tool for the job is critical. Here, we present frameworks to guide your architectural decisions.

4.3.1. ETL vs. ELT

The choice between ETL (Extract, Transform, Load) and ELT (Extract, Load, Transform) depends on your specific needs regarding data volume, latency, and team capabilities.

Recommendation: For modern enterprise data platforms dealing with diverse and high-volume data, ELT with a data lakehouse (like Delta Lake + Spark) is generally preferred due to its scalability, flexibility, and ability to handle both batch and streaming workloads efficiently. ETL still holds value for highly structured, pre-defined integrations into traditional data warehouses.

4.3.2. Messaging Queues: Kafka vs. Kinesis vs. Pub/Sub

Choosing a streaming platform depends on your operational overhead tolerance, specific features, and cloud strategy.

Recommendation: For a local development environment, Apache Kafka is chosen due to its open-source nature, comprehensive feature set, and high relevance in the industry, which prepares engineers for diverse production environments. For cloud deployments, the choice shifts based on your primary cloud provider and operational preferences. For detailed cloud migration strategies for messaging queues, refer to the Cloud Migration + Terraform Snippets Deep-Dive Addendum.

4.3.3. Distributed Processing: Spark vs. Glue/EMR vs. Flink

Choosing a distributed processing engine depends on your workload (batch vs. streaming), cost model, and management preference.

Recommendation: For the local environment, Apache Spark is chosen because it offers flexibility for both batch and streaming, a rich PySpark API, and a broad industry presence. This provides a strong foundation for understanding distributed processing patterns before transitioning to managed cloud services. In the cloud, the choice shifts based on whether serverless ETL (Glue) or more controlled cluster management (EMR) is preferred for Spark workloads, or if pure low-latency stream processing (Flink) is the priority. For conceptual Terraform snippets for Glue and EMR, refer to the Cloud Migration + Terraform Snippets Deep-Dive Addendum.

Data Platform Local Environment Walkthrough & Proficiency Checklist

This document provides a step-by-step learning resource and practical checklist to set up a robust local data platform environment. By following these steps, you will gain hands-on experience with key technologies and be able to demonstrate proficiency for two critical job roles: Lead Data Engineer and AWS Engineer.

This walkthrough leverages the architectural principles, setup guides, and best practices detailed in the "Building Enterprise-Ready Data Platforms: Core Handbook" and its associated Deep-Dive Addendums.

1. General Setup: Laying the Foundation

These steps are foundational and apply to both job roles.

1.1. Prerequisites Installation

Action: Ensure your local machine has the necessary software installed.

Install Docker Desktop (or Docker Engine if on Linux).

Install Git.

Install Python 3.x with pip.

Verify docker-compose is installed (usually included with Docker Desktop, or install separately if not).

1.2. Project Repository Setup

Action: Clone the project mono-repo, which contains all necessary code and configuration files.

Navigate to your desired development directory in your terminal.

Execute: git clone <your-repo-url>/data-ingestion-platform

Change into the cloned directory: cd data-ingestion-platform

2. Local Environment Setup: The Progressive Path

This section guides you through building the local data platform incrementally, mirroring the "Progressive Complexity Path" outlined in the Core Handbook. Each track builds upon the previous one.

Reference: For detailed docker-compose.yml configurations and specific instructions, refer to the Progressive Path Setup Guide Deep-Dive Addendum.

2.1. Starter Track: Minimal Single-Machine Setup

Purpose: Understand foundational data ingestion and structured storage.

Components: FastAPI (Ingestor), PostgreSQL, MinIO (S3 compatible data lake).

Steps:

Configure docker-compose.yml:

Open the docker-compose.yml file in the project root.

Uncomment the services for fastapi\_ingestor, postgres, and minio.

Comment out all other services (Kafka, Spark, Airflow, etc.) to keep the setup minimal.

Ensure the data/postgres and data/minio directories exist in your project root for persistent volumes (Docker will create them if they don't).

Bring Up Services:

Execute the onboard.sh script (from Progressive Path Setup Guide Deep-Dive Addendum) or manually run: docker compose up --build -d

Verify Setup:

Access FastAPI health check: http://localhost:8000/health

Access MinIO Console: http://localhost:9001 (login with minioadmin/minioadmin)

Connect to PostgreSQL: Use a client (e.g., psql) to connect to localhost:5432 with user user, password password, database main\_db.

Check Docker logs for all services: docker compose logs -f

2.2. Intermediate Track: Adding Streaming Capabilities

Purpose: Introduce real-time data streams and distributed transformations.

Components (in addition to Starter): Apache Kafka, Apache Spark.

Steps:

Configure docker-compose.yml:

Open docker-compose.yml.

Uncomment (or keep uncommented) fastapi\_ingestor, postgres, minio.

Uncomment the services for zookeeper, kafka, and spark (and optionally spark-history-server).

Comment out other Advanced Track services.

Review fastapi\_ingestor's environment variables to ensure it publishes to Kafka (KAFKA\_BROKER: kafka:29092).

Verify spark service is configured to connect to Kafka and MinIO.

Ensure data/spark-events exists for Spark history server logs.

Bring Up Services:

Run the onboard.sh script again or docker compose up --build -d. The onboard.sh script will initialize Kafka topics.

Generate External Data:

Run the simulate\_data.py script (from Progressive Path Setup Guide Deep-Dive Addendum). This will continuously send mock financial and insurance data to your FastAPI endpoint.

python3 simulate\_data.py

Trigger Spark Job:

Manually submit a Spark streaming job from the spark container to consume from Kafka and write to Delta Lake in MinIO.

Example: docker exec -it 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 pyspark\_jobs/streaming\_consumer.py raw\_financial\_transactions kafka:29092 s3a://raw-data-bucket/financial\_data\_delta (adjust script path and arguments).

Verify Setup:

Check Kafka topic creation: docker exec -it kafka kafka-topics --bootstrap-server localhost:9092 --list

Observe data flowing into MinIO (Delta Lake raw zone) using the MinIO console.

Check Spark History Server: http://localhost:18080 (if enabled) to see your Spark job status.

2.3. Advanced Track: The Full Production-Ready Stack

Purpose: Integrate orchestration, observability, lineage, and metadata management for a comprehensive environment.

Components (in addition to Intermediate): Apache Airflow, OpenTelemetry & Grafana Alloy, Grafana, Spline, OpenMetadata, MongoDB, cAdvisor.

Steps:

Configure docker-compose.yml:

Open docker-compose.yml.

Uncomment ALL services including airflow-init, airflow-webserver, airflow-scheduler, airflow-worker, mongodb, openmetadata, grafana, grafana-alloy, cAdvisor, spline.

Ensure all environment variables for inter-service communication are correctly set.

Ensure necessary data/ subdirectories for persistent volumes exist.

Mount airflow\_dags and observability directories as volumes for Airflow DAGs and Grafana configurations.

Bring Up Services:

Run the onboard.sh script or docker compose up --build -d. The airflow-init service will set up Airflow's database.

Generate External Data (if not already running):

python3 simulate\_data.py

Verify Setup:

Access Airflow UI: http://localhost:8080 (login admin/admin)

Access Grafana UI: http://localhost:3000 (initially anonymous or configure admin user)

Access OpenMetadata UI: http://localhost:8585

Verify Spline UI: http://localhost:8081

Check for container metrics in Grafana dashboards (e.g., CPU, memory of individual services).

Ensure Airflow DAGs (e.g., data\_ingestion\_dag.py, data\_transformation\_dag.py) appear and run as expected in the Airflow UI, triggering Spark jobs.

Observe data flowing through the entire pipeline, including lineage in Spline and metadata in OpenMetadata.

3. Job Role 1: Lead Data Engineer - Proficiency Checklist

This section outlines key areas and activities to demonstrate expertise relevant to a Lead Data Engineer, leveraging the local environment.

4. Job Role 2: AWS Engineer - Proficiency Checklist

This section outlines key areas and activities to demonstrate expertise relevant to an AWS Engineer, with a focus on data pipelines and cloud infrastructure.

5. Conclusion

By actively engaging with the setup, operation, and troubleshooting of this local data platform environment, and by systematically addressing the points in the proficiency checklists, you will develop a strong, demonstrable understanding of enterprise-ready data platforms from both a Lead Data Engineer and an AWS Engineer perspective. This hands-on experience, coupled with a solid theoretical foundation from the provided documentation, will be invaluable for your career growth.

Full docker-compose.yml for AI review

version: '3.8'

services:

postgres:

image: postgres:15

container\_name: starter-postgres

restart: unless-stopped

environment:

POSTGRES\_USER: user

POSTGRES\_PASSWORD: password

POSTGRES\_DB: starter\_db

volumes:

- ./data/starter-postgres:/var/lib/postgresql/data

ports:

- "5432:5432" # Exposed for direct access and FastAPI connectivity

minio:

image: minio/minio:latest

container\_name: starter-minio

restart: unless-stopped

ports:

- "9000:9000" # MinIO API port

- "9901:9001" # MinIO Console UI port

environment:

MINIO\_ROOT\_USER: minioadmin

MINIO\_ROOT\_PASSWORD: minioadmin

volumes:

- ./data/starter-minio:/data # Persistent volume for MinIO data

command: server /data --console-address ":9001"

healthcheck:

test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]

interval: 30s

timeout: 20s

retries: 3

zookeeper:

image: confluentinc/cp-zookeeper:7.4.0

container\_name: intermediate-zookeeper

restart: unless-stopped

ports:

- "2181:2181"

environment:

ZOOKEEPER\_CLIENT\_PORT: 2181

ZOOKEEPER\_TICK\_TIME: 2000

kafka:

image: confluentinc/cp-kafka:7.4.0

container\_name: intermediate-kafka

restart: unless-stopped

depends\_on:

- zookeeper

ports:

- "9092:9092" # Expose Kafka broker port for external access

environment:

KAFKA\_BROKER\_ID: 1

KAFKA\_ZOOKEEPER\_CONNECT: 'zookeeper:2181'

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT\_HOST://localhost:9092

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT\_HOST:PLAINTEXT

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: PLAINTEXT

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

fastapi\_ingestor:

build: ./fastapi\_app

container\_name: intermediate-fastapi-ingestor

restart: unless-stopped

ports:

- "8000:8000"

environment:

KAFKA\_BROKER: kafka:29092 # Important: use Kafka service name for internal Docker communication

KAFKA\_TOPIC: raw\_financial\_insurance\_data

volumes:

- ./src/fastapi\_app\_intermediate:/app/app # Updated ingestor to publish to Kafka

depends\_on:

kafka:

condition: service\_healthy # Ensure Kafka is healthy before FastAPI tries to connect

spark-master:

image: bitnami/spark:3.5.0

container\_name: intermediate-spark-master

restart: unless-stopped

command: /opt/bitnami/spark/bin/spark-shell # Or spark-class org.apache.spark.deploy.master.Master

environment:

SPARK\_MODE: master

SPARK\_RPC\_AUTHENTICATION\_ENABLED: "no"

SPARK\_EVENT\_LOG\_ENABLED: "true"

SPARK\_EVENT\_LOG\_DIR: "/opt/bitnami/spark/events"

SPARK\_SUBMIT\_ARGS: --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"

ports:

- "8080:8080" # Spark Master UI

- "7077:7077" # Spark Master internal communication

volumes:

- ./data/spark-events:/opt/bitnami/spark/events # For Spark History Server

- ./pyspark\_jobs:/opt/bitnami/spark/data/pyspark\_jobs # Mount PySpark jobs

spark-worker-1:

image: bitnami/spark:3.5.0

container\_name: intermediate-spark-worker-1

restart: unless-stopped

environment:

SPARK\_MODE: worker

SPARK\_MASTER\_URL: spark://spark-master:7077

SPARK\_WORKER\_CORES: 1

SPARK\_WORKER\_MEMORY: 1G

SPARK\_EVENT\_LOG\_ENABLED: "true"

SPARK\_EVENT\_LOG\_DIR: "/opt/bitnami/spark/events"

SPARK\_SUBMIT\_ARGS: --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"

volumes:

- ./data/spark-events:/opt/bitnami/spark/events

depends\_on:

spark-master:

condition: service\_healthy

kafka:

condition: service\_healthy # Dependency on Kafka

minio:

condition: service\_healthy # Dependency on MinIO

# Airflow Services

airflow-scheduler:

image: apache/airflow:2.8.1

container\_name: advanced-airflow-scheduler

restart: always

depends\_on:

airflow-webserver:

condition: service\_healthy

postgres: # Airflow metadata database

condition: service\_healthy

kafka: # For DAGs that interact with Kafka (e.g., Spark jobs)

condition: service\_healthy

environment:

AIRFLOW\_HOME: /opt/airflow

AIRFLOW\_\_CORE\_\_DAGS\_FOLDER: /opt/airflow/dags

AIRFLOW\_\_CORE\_\_EXECUTOR: LocalExecutor # For local dev;

CeleryExecutor for production

AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_CONN: postgresql+psycopg2://user:password@postgres/main\_db

AIRFLOW\_\_WEBSERVER\_\_WEB\_SERVER\_PORT: 8080

AIRFLOW\_\_CORE\_\_LOAD\_EXAMPLES: "false"

volumes:

- ./airflow\_dags:/opt/airflow/dags

- ./data/airflow\_logs:/opt/airflow/logs

- ./orchestrator/plugins:/opt/airflow/plugins # If you have custom plugins

command: scheduler

healthcheck:

test: ["CMD-SHELL", "airflow jobs check --job-type SchedulerJob --hostname $$HOSTNAME"]

interval: 10s

timeout: 10s

retries: 5

airflow-webserver:

image: apache/airflow:2.8.1

container\_name: advanced-airflow-webserver

restart: always

depends\_on:

postgres:

condition: service\_healthy

ports:

- "8081:8080" # Mapped to 8081 to avoid conflict with Spark Master UI

environment:

AIRFLOW\_HOME: /opt/airflow

AIRFLOW\_\_CORE\_\_DAGS\_FOLDER: /opt/airflow/dags

AIRFLOW\_\_CORE\_\_EXECUTOR: LocalExecutor

AIRFLOW\_\_DATABASE\_\_SQL\_ALCHEMY\_CONN: postgresql+psycopg2://user:password@postgres/main\_db

AIRFLOW\_\_WEBSERVER\_\_WEB\_SERVER\_PORT: 8080

AIRFLOW\_\_CORE\_\_LOAD\_EXAMPLES: "false"

volumes:

- ./airflow\_dags:/opt/airflow/dags

- ./data/airflow\_logs:/opt/airflow/logs

- ./orchestrator/plugins:/opt/airflow/plugins

command: webserver

healthcheck:

test: ["CMD-SHELL", "curl --silent --fail http://localhost:8080/health"]

interval: 10s

timeout: 10s

retries: 5

# Observability Components

grafana:

image: grafana/grafana:latest

container\_name: advanced-grafana

restart: unless-stopped

ports:

- "3000:3000" # Grafana Web UI

volumes:

- ./data/grafana:/var/lib/grafana # Persistent storage for Grafana data

- ./observability/grafana\_dashboards:/etc/grafana/provisioning/dashboards # Mount dashboards

- ./observability/grafana\_datasources:/etc/grafana/provisioning/datasources # Mount datasources

environment:

GF\_SECURITY\_ADMIN\_USER: admin

GF\_SECURITY\_ADMIN\_PASSWORD: admin

depends\_on:

grafana\_alloy:

condition: service\_started

cadvisor:

condition: service\_started

grafana\_alloy:

image: grafana/alloy:latest

container\_name: advanced-grafana\_alloy

restart: unless-stopped

ports:

- "4317:4317" # OTLP gRPC endpoint for receiving telemetry

- "4318:4318" # OTLP HTTP endpoint for receiving telemetry

- "12345:12345" # Example Prometheus scrape port for Grafana to pull metrics from Alloy

volumes:

- ./observability/alloy-config.river:/etc/alloy/config.river # Mount your Alloy configuration

command: -config.file=/etc/alloy/config.river

cadvisor:

image: gcr.io/cadvisor/cadvisor:v0.47.0 # Stable version for container metrics

container\_name: advanced-cadvisor

restart: unless-stopped

ports:

- "8082:8080" # Default cAdvisor UI/metrics port (mapped to 8082 to avoid conflicts)

volumes:

- /:/rootfs:ro

- /var/run:/var/run:rw

- /sys:/sys:ro

- /var/lib/docker/:/var/lib/docker:ro

- /dev/disk/:/dev/disk:ro

command: --listen\_ip=0.0.0.0 --port=8080 # Expose on all interfaces on port 8080

healthcheck:

test: ["CMD-SHELL", "wget -q --spider http://localhost:8080/metrics || exit 1"]

interval: 30s

timeout: 10s

retries: 3

start\_period: 10s

# Data Lineage (Spline) Components

spline-rest:

image: aballon/spline-rest-server:latest # Use a specific version, e.g., 0.7.1

container\_name: advanced-spline-rest

restart: unless-stopped

ports:

- "8083:8080" # Spline REST API server (mapped to 8083 to avoid conflicts)

depends\_on:

postgres: # Spline can use a persistent DB for metadata

condition: service\_healthy

spline-ui:

image: aballon/spline-web-ui:latest # Use a specific version, e.g., 0.7.1

container\_name: advanced-spline-ui

restart: unless-stopped

ports:

- "9090:80" # Spline Web UI

environment:

SPLINE\_API\_URL: http://spline-rest:8080 # Connects to the spline-rest service

depends\_on:

- spline-rest

# Metadata Management (OpenMetadata) Components

openmetadata-mysql:

image: mysql:8.0

container\_name: advanced-openmetadata-mysql

restart: unless-stopped

environment:

MYSQL\_ROOT\_PASSWORD: openmetadata\_user

MYSQL\_USER: openmetadata\_user

MYSQL\_PASSWORD: openmetadata\_password

MYSQL\_DATABASE: openmetadata\_db

volumes:

- ./data/openmetadata\_mysql:/var/lib/mysql

ports:

- "3306:3306"

command: --default-authentication-plugin=mysql\_native\_password

healthcheck:

test: ["CMD", "mysqladmin", "ping", "-h", "localhost", "-u$$MYSQL\_USER", "-p$$MYSQL\_PASSWORD"]

interval: 10s

timeout: 5s

retries: 5

openmetadata-elasticsearch:

image: opensearchproject/opensearch:2.11.0 # Or elasticsearch:7.17.10

container\_name: advanced-openmetadata-elasticsearch

restart: unless-stopped

environment:

discovery.type: single-node

OPENSEARCH\_JAVA\_OPTS: "-Xms512m -Xmx512m"

ports:

- "9200:9200" # HTTP API

- "9600:9600" # Transport port

volumes:

- ./data/openmetadata\_elasticsearch:/usr/share/opensearch/data

healthcheck:

test: ["CMD-SHELL", "curl -f http://localhost:9200/\_cat/health?h=st | grep -q green"]

interval: 10s

timeout: 10s

retries: 5

openmetadata-server:

image: openmetadata/openmetadata:1.3.1

container\_name: advanced-openmetadata-server

restart: unless-stopped

depends\_on:

openmetadata-mysql:

condition: service\_healthy

openmetadata-elasticsearch:

condition: service\_healthy

ports:

- "8585:8585" # OpenMetadata Web UI

environment:

MYSQL\_HOST: openmetadata-mysql

MYSQL\_PORT: 3306

MYSQL\_DATABASE: openmetadata\_db

MYSQL\_USER: openmetadata\_user

MYSQL\_PASSWORD: openmetadata\_password

ELASTICSEARCH\_HOST: openmetadata-elasticsearch

ELASTICSEARCH\_PORT: 9200

APP\_ENV: local

command: ["./docker/run\_server.sh"]

healthcheck:

test: ["CMD-SHELL", "curl -f http://localhost:8585/api/v1/health | grep -q OK"]

interval: 30s

timeout: 20s

retries: 5

openmetadata-ingestion:

image: openmetadata/ingestion-base:1.3.1

container\_name: advanced-openmetadata-ingestion

restart: on-failure

depends\_on:

openmetadata-server:

condition: service\_healthy

environment:

OPENMETADATA\_SERVER\_URL: http://openmetadata-server:8585

volumes:

- ./openmetadata\_ingestion\_scripts:/opt/openmetadata/examples/workflows