# Building Enterprise-Ready Data Platforms: An Evolved Guide

## Table of Contents

# 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.

## 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.

## 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.

# 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:**

| Benefit Area | Current State Challenge | Platform Impact | Estimated ROI / Metrics |
|---|---|---|---|
| **Faster Time-to-Value** | Manual setup, slow iteration cycles | **~30%** faster **feature delivery** due to robust local dev & CI/CD. Reduces developer onboarding from weeks to days. | **Reduce feature delivery time by 3-4 weeks/quarter.** Accelerate new data product launches. |
| **Reduced Cloud Spend** | Inefficient resource utilization in dev/test | **~20-40% reduction in dev/test cloud costs** by shifting workloads locally. Optimized cloud resource scaling. | **Anticipated $50K-$150K annual savings** in non-production cloud infrastructure. |
| **Improved Data Quality** | Inconsistent data, manual validation | Automated schema enforcement, contract testing, and data quality checks (e.g., Great Expectations). | **Decrease data-related incidents by 50%**, ensuring reliable insights and reporting. |
| **Enhanced Compliance & Audit** | Scattered data, unclear lineage, manual audits | Centralized metadata catalog (OpenMetadata), | **Streamline audit preparation by 70%**, reducing compliance |

| | | | |
|---|---|---|---|
| | | automated lineage (Spline), and robust access controls. | burden and risk penalties. |
| **Operational Efficiency** | Reactive issue resolution, alert fatigue | Proactive monitoring (Grafana), clear SLIs/SLOs, and structured incident response (DR Playbook). | **Reduce Mean Time To Resolution (MTTR) by 40%** for data-related incidents. Lower operational overhead. |
| **Scalability & Future-Proofing** | Monolithic systems, limited real-time capabilities | Modular, distributed architecture (Kafka, Spark, Delta Lake) built for petabyte scale and real-time processing. | **Support 5x data growth over 3 years** without major re-architecture. Enable new real-time use cases (e.g., fraud detection). |

**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.

**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.

## 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.

**Conceptual docker-compose.yml Snippet (Starter):** (The full docker-compose.yml is detailed in Appendix E, but here's a focused view for this track.)

```yaml
# Simplified docker-compose.yml for Starter Track
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

  fastapi_ingestor:
    build: ./fastapi_app # Path to your FastAPI Dockerfile
    container_name: starter-fastapi-ingestor
    restart: unless-stopped
    ports:
      - "8000:8000" # Expose FastAPI API port
```

```
environment:
  # These variables would direct FastAPI to store data directly into Postgres or MinIO
  DATABASE_TYPE: "postgres" # Or "minio" for direct file writes
  POSTGRES_HOST: postgres
  POSTGRES_PORT: 5432
  MINIO_HOST: minio
  MINIO_PORT: 9000
  MINIO_ACCESS_KEY: minioadmin
  MINIO_SECRET_KEY: minioadmin
volumes:
  # Mount application code for development and hot-reloading
  - ./src/fastapi_app_starter:/app/app # Simplified ingestor for direct DB/MinIO writes
depends_on:
  postgres:
    condition: service_healthy # Ensure Postgres is ready
  minio:
    condition: service_healthy # Ensure MinIO is ready
```

**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.

**Conceptual docker-compose.yml Snippet (Intermediate):** (Extends Starter components, full docker-compose.yml is in Appendix E)

```
# Intermediate Track: Add Kafka & Spark for streaming
```

```
version: '3.8'
services:
  # ... (postgres, minio services - still present for reference/metadata) ...

  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:
```

```yaml
      - ./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
```

**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.

**Conceptual docker-compose.yml Snippet (Advanced):** (Extends Intermediate components, the full docker-compose.yml is in Appendix E)

```
# Advanced Track: Add Airflow, Observability, Lineage, Metadata
version: '3.8'
```

```yaml
services:
  # ... (postgres, mongodb, minio, zookeeper, kafka, fastapi_ingestor,
spark-master/workers/history services) ...

  # 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
```

```yaml
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
```

```yaml
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
```

```yaml
      - "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
```

**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.
- **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.

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.

**Platform Architecture Diagram:**

```
@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 (Appendix E):

- **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.

| **Feature / Criteria** | **ETL (Extract, Transform, Load)** | **ELT (Extract, Load, Transform)** |
|---|---|---|
| **Data Volume** | Better for smaller, more controlled datasets | Ideal for large, unbounded datasets (petabytes to exabytes) |
| **Latency Needs** | Typically batch-oriented, higher latency acceptable | Suited for real-time or near real-time, lower latency required |
| **Transformation Logic** | Performed *before* loading into target; requires dedicated staging area | Performed *after* loading into the data lake; leverages lakehouse compute |
| **Tooling** | Traditional ETL tools (Talend, Informatica), custom scripting | Cloud data warehouses (Snowflake, BigQuery), Spark, Databricks, Glue |
| **Team Skills** | May lean towards SQL/ETL tool expertise | Strong programming (Python/Scala) and distributed systems knowledge |
| **Cost Model** | Fixed infrastructure for ETL tools; less flexible scaling | Scalable compute (Spark, cloud DWs); compute cost scales with usage |
| **Schema Flexibility** | Schema-on-write, stricter schema enforcement | Schema-on-read, more flexible, handles schema evolution better |

**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.

| Feature / Criteria | Apache Kafka (Self-Managed) | AWS Kinesis | Google Cloud Pub/Sub |
|---|---|---|---|
| **Throughput/Scale** | Extremely high; scales horizontally with brokers and partitions | High; scales with shards | Very high; scales automatically |
| **Operational Overhead** | **High** (requires managing Zookeeper, brokers, maintenance) | **Medium** (managed service, but shard management is manual) | **Low** (fully managed, serverless, no infrastructure to manage) |
| **Cloud Lock-in** | **Low** (open-source, portable across clouds/on-prem) | **High** (AWS-specific) | **High** (Google Cloud-specific) |
| **Pricing Model** | Infrastructure costs + operational expertise | Per shard-hour, data transfer, and data ingested/egressed | Per message operation (publish/subscribe), data transfer |
| **Feature Set** | Rich ecosystem (Kafka Connect, Streams API); flexible | Data Firehose (integrations), Data Analytics (real-time SQL) | Global access, automatic scaling, robust IAM integration |
| **Use Case Sweet Spot** | Hybrid/multi-cloud, complex streaming apps, full control needed | AWS-native streaming, tight integration with other AWS services | Google Cloud-native, event-driven architectures, simple messaging |

**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.

### 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.

| Feature / Criteria | Apache Spark (Self-Managed/Open Source) | AWS Glue (Serverless ETL) | Amazon EMR (Managed Clusters) | Apache Flink (Stream Processing) |
|---|---|---|---|---|
| **Workload Focus** | Both Batch & Streaming (Structured Streaming) | Primarily Batch ETL, can do micro-batch streaming | Both Batch & Streaming (various engines) | Primarily Real-time Streaming |
| **Cost Model** | Infrastructure + operational | Serverless, pay-per-use | Instance-based pricing, cluster | Infrastructure + operational |

| | | | | |
|---|---|---|---|---|
| | overhead; flexible | (DPUs/duration) | management costs | overhead; flexible |
| **Resource Isolation** | Manual configuration per job/cluster | Automatic, jobs are isolated | Cluster-level isolation | Fine-grained resource control, low-latency stateful processing |
| **Management Overhead** | High (setup, maintenance, scaling) | Low (fully managed, no servers to provision) | Medium (managed, but cluster configuration and lifecycle remain) | High (setup, maintenance, scaling) |
| **Development Experience** | PySpark/Scala/Java; high control, rich APIs | PySpark/Scala; managed environment; integrates with Data Catalog | PySpark/Scala/Java; integrates with other AWS services | Java/Scala; complex stateful processing APIs, event-time processing |
| **Use Case Sweet Spot** | Diverse workloads, fine-grained control, on-prem/hybrid cloud | Ad-hoc ETL, event-driven jobs, data lake transformations | Big data analytics, ad-hoc queries, transient/long-running clusters | Real-time dashboards, fraud detection, complex event processing |

**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.

# 5. Development Best Practices & Operational Excellence

## 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.

- **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:**

| Solution | Type | Strengths | Weaknesses | Usage Tips |
|---|---|---|---|---|
| **AWS** Secrets **Manager** | Cloud-Native | Fully managed, automated rotation, granular IAM policies, integrates with other AWS services. | AWS lock-in. | Use for most AWS-native applications. Implement rotation policies for databases. |
| **HashiCorp Vault** | Vendor-Neutral | Strong audit logging, dynamic secrets (on-demand credentials), robust access controls, supports multiple backends. | Requires self-management (or Vault Enterprise), steeper learning curve. | Run Vault Agent as a sidecar in Kubernetes/ECS to inject secrets. Implement transit encryption. |
| **Doppler** | SaaS | Centralized secrets management for multiple environments, easy integration with CI/CD. | SaaS dependency, potential for vendor lock-in. | Good for smaller teams or those prioritizing ease of use and cross-environment consistency. |
| **SOPS (Secrets OPerationS)** | Open-Source | Encrypts secrets in Git (YAML, JSON), works well with GitOps, easy CLI. | Less dynamic than Vault, requires key management (KMS, GPG). | Ideal for encrypting static configuration secrets in Git repos (e.g., Kubernetes manifests). |

**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.

```yaml
# .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
```

## 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
```

```yaml
    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.

```python
# 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"
```

```python
    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.rep
lace('/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
```

```python
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

# 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.

## 5.5. Data Contracts & Schema Governance

Formalizing data contracts and governing schema evolution is critical for data quality and interoperability.

- **Formalizing Contracts:**
  - Define formal data schemas using schema-driven serialization formats like **Apache Avro** or **Google Protobuf**.
  - **Pydantic** models in FastAPI automatically generate JSON Schema definitions, which are embedded in the OpenAPI specification (/openapi.json), providing immediate contract documentation and validation.
- **Schema Registry:** Implement a **Schema Registry** (e.g., Confluent Schema Registry, Apicurio) to centralize, manage, and evolve schemas. Producers register their schemas, and consumers can retrieve them to ensure data compatibility.
- **Schema Evolution Governance:** The Schema Registry facilitates controlled schema evolution (e.g., adding nullable fields, reordering fields) while ensuring backward and forward compatibility.

- **Contract Lifecycle in Git & Governance Board Process:**
    1. **Schema Change Proposal:** A data producer (e.g., the FastAPI team) opens a Git Pull Request (PR) to modify a schema definition (e.g., an Avro .avsc file or a Pydantic model in src/models).
    2. **Automated Contract Tests:** The CI pipeline (see 5.3) automatically triggers contract tests (e.g., using Pact or a pytest-based schema validator). These tests verify that existing consumers can still process data conforming to the *new* schema (backward compatibility) or that new producers adhere to the *old* schema (forward compatibility for rolling upgrades).
    3. **Governance Board Review:** For significant changes (e.g., removal of mandatory fields), a "Data Governance Board" (comprising data owners, architects, and key consumers) reviews the PR.
        - **Sign-off Criteria:** Backward compatibility confirmed, potential downstream impact documented, data migration strategy (if any) defined, and impact on historical data assessed.
        - **Documentation:** Updates to the data catalog (OpenMetadata) and any consumer-facing documentation are mandatory.
    4. **Rolling Upgrades:** Deploy new producers and consumers with the updated schema in a rolling fashion to ensure zero downtime. The Schema Registry plays a key role here by allowing consumers to fetch schemas by version.

## 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.

| Layer | Example SLI | Example SLO |
|---|---|---|
| **Data Ingestion** | End-to-end ingest latency (API call to Raw Zone persistence) | <5 seconds for 99% of transactions |
| **Streaming Pipeline** | Kafka Consumer Lag (number of messages behind) | <10,000 messages for 99.9% of time |
| **Batch ETL Jobs** | Job completion rate / Daily job completion time (end-to-end) | 99% of daily jobs complete successfully; <60 minutes for 95% of runs |
| **Data Quality** | % of records failing schema validation / data quality checks | <0.1% of records rejected / flagged for correction |
| **Data Storage** | Delta Lake write amplification (ratio of physical written bytes | <2.0 (maintaining storage efficiency) |

| | | |
|---|---|---|
| | to logical changes) | |
| **API Availability** | Uptime of FastAPI Ingestor | 99.99% uptime |

### 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:**
    1. **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.
    2. **Review Spark Logs:** Examine executor logs and driver logs for specific errors (deserialization, processing exceptions, OutOfMemoryError), continuous restarts, or backpressure warnings.
    3. **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>
       ```
    4. **Verify Kafka Broker Health:** Check kafka and zookeeper container logs for any errors (e.g., disk full, network issues).
    5. **Grafana Consumer Lag Panel:** Monitor a pre-built Grafana dashboard (see "Health-Check Dashboard" in Section 8.1) 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:**
    1. **Identify Schema Change:** Compare incoming DataFrame schema with the existing Delta table schema. The error message usually highlights the problematic column or type.
    2. **Review Error Message:** Understand if a column was added, removed, renamed, or its type changed.
    3. **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:**
        1. **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.
        2. **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
```
        3. **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.
        4. **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.

# 6. Disaster Recovery (DR) Playbook

Disaster Recovery (DR) is critical for ensuring business continuity and minimizing data loss in the event of major failures. A DR playbook outlines the procedures to recover systems and data.

## 6.1. RPO and RTO in Context

- **Recovery Point Objective (RPO):** The maximum acceptable amount of data loss, measured in time. (e.g., an RPO of 1 hour means you can tolerate losing up to 1 hour of data).
- **Recovery Time Objective (RTO):** The maximum acceptable downtime before a system must be restored to operation after a disaster. (e.g., an RTO of 4 hours means the system must be fully operational within 4 hours of an outage).

These objectives are set based on business criticality and define the necessary backup, replication, and restoration strategies.

## 6.2. Backup & Restore Verification

A backup strategy is only effective if its restoration process is regularly tested.

- **Automated Backups:** Implement automated backups for all critical data stores (PostgreSQL, MongoDB, MinIO/S3, Kafka offsets, Airflow metadata).
    - For databases, use point-in-time recovery (PITR) features.
    - For object storage, leverage versioning, cross-region replication, and lifecycle policies.
- **Regular Restore Drills:** Periodically perform full or partial restoration drills into a separate, isolated environment.
    - **Verification:** After restoration, run data integrity checks and smoke tests to confirm data consistency and system functionality.
    - **Documentation:** Document the restoration steps in a dedicated runbook and update it with lessons learned from drills.

## 6.3. Runbook Templates for Critical Systems

A runbooks/ directory in your mono-repo is essential for documenting operational procedures, especially for DR. This library provides clear, step-by-step instructions for diagnosing, mitigating, and recovering from common incidents or system failures.

**Conceptual /runbooks/ Directory Structure:**

```
runbooks/
├── kafka_consumer_lag.md        # From previous section
├── spark_job_hang.md            # From previous section
├── kafka_restore_from_backup.md
```

├── delta_lake_time_travel_recovery.md
├── airflow_metadata_db_recovery.md
└── incident_response_flowchart.png # Or .puml for diagram

**Example Runbook Snippets (full details in Appendix G):**

- **/runbooks/kafka_restore_from_backup.md (Conceptual):**

  # Kafka Topic Restore from Backup

  **Incident:** Data loss or corruption in a Kafka topic (e.g., `raw_financial_insurance_data`).

  **Purpose:** Restore Kafka topic data from a predefined backup (e.g., S3 archive).

  **RPO/RTO:** Varies per topic. (e.g., RPO: 2 hours, RTO: 4 hours)

  **Steps:**

  1. **Stop Consumers:** Ensure all consumers of the affected topic are stopped to prevent further consumption of corrupted data or writing to the restored topic during the process.
     * `kubectl scale deployment/spark-consumer-deployment --replicas=0` (in K8s)
     * Or disable Airflow DAGs: `airflow dags pause <dag_id>`

  2. **Identify Backup:** Locate the last good backup of the Kafka topic data in your backup storage (e.g., S3 bucket `s3://kafka-backups/raw_financial_insurance_data/2023-10-26_14-00-00/`).

  3. **Delete Existing Topic (CAUTION!):** If the topic is corrupted, it might be necessary to delete and recreate it. **Ensure you have a valid backup before this step.**
     * `docker exec kafka kafka-topics.sh --bootstrap-server localhost:9092 --delete --topic raw_financial_insurance_data`

  4. **Recreate Topic:** Recreate the topic with its original configuration (partitions, replication factor).
     * `docker exec kafka kafka-topics.sh --bootstrap-server localhost:9092 --create --topic raw_financial_insurance_data --partitions 3 --replication-factor 1`

  5. **Load Data from Backup:** Use a Kafka producer or a custom script to ingest the backed-up data into the newly recreated topic. This could involve reading from S3 and producing to Kafka.
     * `python scripts/kafka_data_loader.py --topic raw_financial_insurance_data --file s3://kafka-backups/raw_financial_insurance_data/...`

6.  **Verify Data Integrity:** After loading, run data integrity checks or consume a sample of the topic to ensure data is correct.

7.  **Restart Consumers/Producers:** Once verification is complete, restart your consumers and producers. Consumers should start from `earliest` offset to process all restored data.

8.  **Post-Recovery Monitoring:** Closely monitor Kafka consumer lag and overall pipeline health.

**Related Runbooks:**
* `./kafka_consumer_lag.md`
* `./spark_job_hang.md`

- **/runbooks/delta_lake_time_travel_recovery.md (Conceptual):**
# Delta Lake Time Travel Recovery

**Incident:** Accidental data deletion, update, or corruption in a Delta Lake table.

**Purpose:** Recover a Delta Lake table to a previous good state using Time Travel.

**RPO/RTO:** Near-instant recovery for point-in-time and version recovery.

**Steps:**

1.  **Identify Last Good Version/Timestamp:**
    * Use `DESCRIBE HISTORY` on the Delta table to review past operations and identify the version number or timestamp of the last known good state.
        ```sql
        -- In Spark SQL
        DESCRIBE HISTORY delta.`s3a://your-bucket/curated/transactions`
        ```
        Look for `version` and `timestamp`.

2.  **Query Previous Version:** Confirm the data looks correct at that historical version.
    ```python
    # In PySpark
    df_good_version = spark.read.format("delta").option("versionAsOf", <good_version_number>).load("s3a://your-bucket/curated/transactions")
    df_good_version.show()
    ```sql
    -- In Spark SQL
    SELECT * FROM delta.`s3a://your-bucket/curated/transactions` VERSION AS OF

<good_version_number>
    ```

3.  **Restore the Table (Option 1: Overwrite):** If the corruption affects a large portion or the entire table, you can overwrite the current table with the good version.
    * **CAUTION:** This is destructive to any changes *after* the good version.
    ```python
    # In PySpark
    df_good_version.write.format("delta").mode("overwrite").option("overwriteSchema", "true").save("s3a://your-bucket/curated/transactions")
    ```

4.  **Restore the Table (Option 2: Selective Merge/Upsert):** If only a subset of data is affected, read the good version, filter for the affected data, and then merge it back into the current table. This is more surgical.
    * Requires careful `MERGE` logic (similar to SCD Type 2) to insert/update specific rows.

5.  **Verify Restoration:** After restoring, run data quality checks and smoke tests on the table.

6.  **Inform Downstream:** Notify consumers that the table was restored and might have temporary data inconsistencies if not a full restore.

**Related Runbooks:**
* `./delta_lake_schema_drift.md` (if schema changes were involved)

- **/runbooks/airflow_metadata_db_recovery.md (Conceptual):**
# Airflow Metadata Database Recovery

**Incident:** Corruption or loss of the Airflow metadata PostgreSQL database.

**Purpose:** Restore Airflow's operational state by recovering its metadata database.

**RPO/RTO:** High impact, critical for DAG scheduling. (e.g., RPO: 1 hour, RTO: 2 hours)

**Steps:**

1.  **Stop Airflow Services:** Stop `airflow-webserver`, `airflow-scheduler`, `airflow-triggerer`.
    * `docker compose stop airflow-webserver airflow-scheduler airflow-triggerer`

2.  **Stop PostgreSQL:** Stop the Airflow metadata database.
    ```

* `docker compose stop postgres`

3. **Backup Current (Corrupted) DB (Optional but Recommended):** If there's any chance of forensic analysis, backup the corrupted volume.
   * `docker cp postgres:/var/lib/postgresql/data ./data/postgres_corrupted_backup`

4. **Restore PostgreSQL Volume:** Replace the current PostgreSQL data volume with a backup. This might involve:
   * Deleting the existing volume: `docker volume rm data_ingestion_platform_data_postgres` (if using named volumes)
   * Restoring from a snapshot or a file-level backup to `./data/postgres/`
   * Or, if using a fresh container, attach a restored data volume.

5. **Start PostgreSQL:** Start the restored PostgreSQL container.
   * `docker compose start postgres`
   * Verify health: `docker compose logs postgres`

6. **Verify Airflow DB Connection:** Use a `psql` client to confirm Airflow's user can connect and see the `main_db`.

7. **Run Airflow DB Upgrade/Check:** Sometimes, after restoration, Airflow might need to run a database upgrade command if schema mismatches occur.
   * `docker exec airflow-webserver airflow db check`
   * `docker exec airflow-webserver airflow db upgrade` (Use with caution)

8. **Restart Airflow Services:**
   * `docker compose start airflow-webserver airflow-scheduler airflow-triggerer`

9. **Post-Recovery Monitoring:** Check Airflow Web UI (`http://localhost:8081`) for DAG status, task history, and scheduler health. Verify new DAG runs are triggered correctly.

**Related Runbooks:**
* `./database_backup_strategy.md` (general database backup)

# 7. Performance, Scale, & Quantitative Benchmarks

Optimizing performance and scalability is crucial for handling large volumes of financial and insurance data efficiently.

## 7.1. Data Partitioning & File Layout (Delta Lake)

Efficient data organization within Delta Lake directly impacts query performance and storage costs.

- **Partitioning Strategies:**
    - **Date-Based Partitioning:** For time-series data like financial transactions or insurance claims, partitioning by date (e.g., year, month, day) is highly effective. This allows Spark to prune irrelevant data quickly based on query filters (e.g., WHERE transaction_date = '2023-10-26').
        - Example: /data/curated/transactions/year=2023/month=10/day=26/
    - **Hash Bucketing (or other categorical partitioning):** For high-cardinality columns that are frequently filtered or joined upon (e.g., customer_id, policy_number), bucketing can distribute data evenly across a fixed number of directories, improving join and filter performance without creating too many small files. This is often used *in* conjunction with partitioning.
- **Compaction Best Practices:** Delta Lake tables can accumulate many small files from frequent micro-batch writes (e.g., from Spark Structured Streaming). Many small files lead to inefficient reads and increased metadata overhead.
    - **OPTIMIZE Command:** Regularly run the OPTIMIZE command on Delta tables to compact small files into larger, more optimal ones.
```
# PySpark: Compacting a Delta table example
from delta.tables import DeltaTable
# Assume 'spark' is an initialized SparkSession
# Assume 'target_delta_table_path' is defined (e.g., "/data/curated/transactions")
delta_table = DeltaTable.forPath(spark, target_delta_table_path)
print(f"Optimizing Delta table: {target_delta_table_path}")
delta_table.optimize().execute()
print("Optimization completed.")
# You can also optimize specific partitions
# delta_table.optimize().where("year = '2023' AND month = '10'").execute()
```
    - **Z-Ordering:** For tables with many columns that are often used in query predicates, Z-ordering (a multi-dimensional clustering technique) can further improve data skipping. OPTIMIZE ... ZORDER BY (col1, col2).
    - **File Sizing Recommendations:** Aim for file sizes between 128MB and 1GB for optimal read performance in distributed systems like Spark. This balances the overhead of opening too many small files against the cost of reading unnecessarily large blocks of data.

## 7.2. Indexing & Caching for Databases (Postgres/MongoDB)

Proper indexing and leveraging caching mechanisms are vital for accelerating queries on relational and NoSQL databases.

- **PostgreSQL Indexing:**
    - B-tree Indexes: Default and most common, good for equality and range queries.
```
-- Example: Index on transaction_id for faster lookups
CREATE INDEX idx_financial_transactions_transaction_id ON
```

financial_transactions (transaction_id);
-- Example: Composite index for common filters/sorts
CREATE INDEX idx_insurance_claims_customer_date ON insurance_claims (customer_id, incident_date DESC);

  - GIN (Generalized Inverted Index): For columns storing JSONB data or arrays, useful for querying keys or elements within them.
  - BRIN (Block Range Index): For very large tables where data is naturally ordered (e.g., time-series data, often inserted sequentially), BRIN indexes are much smaller and faster than B-trees for range queries.
- **MongoDB Indexing:**
  - Single-Field Indexes: For frequently queried fields.
    // Example: Index on claim_id
    db.insurance_claims.createIndex({ claim_id: 1 })
    // Example: Index on a date field for sorting
    db.financial_transactions.createIndex({ timestamp: -1 })

  - Compound Indexes: For queries that involve multiple fields (e.g., db.collection.find({fieldA: ..., fieldB: ...})).
    // Example: Compound index for queries filtering by account_id and amount
    db.financial_transactions.createIndex({ account_id: 1, amount: -1 })

  - Text Indexes: For full-text search capabilities on string content.
- **In-Memory Caching (Database Specific):**
  - **PostgreSQL (pg_prewarm):** Can be used to explicitly load specified relations into the operating system's file system cache or the PostgreSQL buffer pool. This is useful after a restart or for specific critical tables.
    -- Example: Pre-warm a table into OS file system cache
    SELECT pg_prewarm('public.financial_transactions');

  - **MongoDB:** MongoDB leverages the operating system's file system cache for its working set. Ensuring sufficient RAM on the server or Docker container for MongoDB to keep its frequently accessed data (indexes and data) in memory is paramount. Monitor wiredTiger.cache.trackedDirtyBytes and wiredTiger.cache.pagesReadIntoCache for cache performance.

## 7.3. Throughput Targets & Sizing Guidance

Approximating resource requirements for Spark and Kafka is crucial for meeting performance targets.

- **Kafka Throughput:** Kafka's throughput scales linearly with the number of brokers and partitions.
  - **Partitions:** A good starting point is to have 2-4 partitions per core on each Kafka broker. More partitions enable higher parallelism for consumers.

    - **Replication Factor:** For fault tolerance, a replication factor of 3 is common in production (local dev often uses 1).
- **Spark Cluster Sizing:** Sizing Spark involves balancing executor resources (cores, memory) and the number of executors. The goal is to maximize parallelism and minimize data shuffling.

| Expected TPS (Kafka Ingestion) | Estimated Spark Cluster (Approx. Cores) | Example Configuration (3 Workers) | Key Sizing Factor Notes |
|---|---|---|---|
| **100-500** | 4-8 | 1 Master (4GB RAM, 2 Cores) + 2-3 Workers (8GB RAM, 4 Cores each) | For ingestion, focus on Kafka consumer parallelism. For transformations, consider data volume and complexity. These are starting points, and actual sizing requires profiling against realistic workloads. |
| **500-2000** | 16-32 | 1 Master (8GB RAM, 4 Cores) + 5-8 Workers (16GB RAM, 8 Cores each) | More workers and more cores/memory per worker. Monitor Spark UI for executor utilization and garbage collection. Real-world performance will heavily depend on data skew, transformation complexity, and network I/O. |
| **2000+** | 64+ | 1 Master (16GB RAM, 8 Cores) + 10+ Workers (32GB RAM, 16 Cores each) | Requires careful tuning and potentially specialized hardware/cloud instances. Consider dedicated instances, high-throughput network bandwidth, and optimizing Spark configurations like spark.sql.shuffle.partitions, spark.memory.fraction, |

| | | | |
|---|---|---|---|
| | | | and custom shuffle services. Extensive profiling is mandatory. |

- **Approximating Spark Resources (per Executor):**
  - --executor-cores: Number of CPU cores for each executor. Generally, 2-5 cores per executor is a good range to avoid too many threads causing context switching overhead.
  - --executor-memory: Amount of memory (RAM) allocated to each executor. This needs to be sufficient to hold intermediate data, especially for shuffles and joins. Account for JVM overhead (e.g., subtract 10-20% for overhead).
  - --num-executors: Number of executors. This depends on the total available cores in your cluster and the desired level of parallelism. A common rule of thumb is to have enough executors such that num_executors * executor_cores is slightly less than the total available cores in your worker nodes.
- Example Sizing for a PySpark Job:
  If you have a 3-worker cluster, each with 4 cores and 16GB RAM:
  You might configure 2 executors per worker, each with --executor-cores 2 and --executor-memory 6GB (leaving some RAM for OS/other processes).
  Total executors: 6. Total cores: 12. Total memory: 36GB.

## 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.

- **Benchmarking Harness Components:**
  1. **Load Generator (Locust):** Simulates concurrent users sending financial/insurance data to the FastAPI ingestion API.
  2. **FastAPI Ingestor:** Receives data and publishes it to Kafka.
  3. **Kafka Cluster:** Buffers the incoming data stream.
  4. **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.
  5. **Metrics Collector (Grafana Alloy):** Collects metrics from FastAPI, Kafka, Spark, and cAdvisor.
  6. **Monitoring (Grafana):** Visualizes end-to-end latency, throughput, and resource utilization.
- **Conceptual Benchmarking Steps:**
  1. **Setup Environment:** Bring up the full Advanced Track Docker Compose environment.
  2. **Run Load Generator:** Start Locust to simulate X users sending Y requests per second to FastAPI.
  3. **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).
    4. **Analyze Data:** Record and analyze average/p99 latency, throughput, and resource bottlenecks.
    5. **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 - Full script in Appendix H):**

```
# locust_fastapi_ingestor.py (Conceptual)
from locust import HttpUser, task, between
import json
from datetime import datetime, timedelta
import random

class FinancialDataUser(HttpUser):
  wait_time = between(0.1, 0.5) # Simulate delay between requests
  host = "http://localhost:8000" # Target FastAPI endpoint

  @task(1)
  def ingest_financial_transaction(self):
    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),
      "currency": random.choice(["USD", "EUR", "GBP"]),
      "transaction_type": random.choice(["debit", "credit", "transfer"]),
      "merchant_id": f"MER-{random.randint(100, 999)}" if random.random() > 0.3 else
None,
      "category": random.choice(["groceries", "utilities", "salary"])
    }
    self.client.post("/ingest-financial-transaction/", json=transaction_data,
name="/ingest-financial-transaction")

  @task(1)
  def ingest_insurance_claim(self):
    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),
          "claim_type": random.choice(["auto", "health", "home"]),
          "claim_status": random.choice(["submitted", "under_review", "approved"]),
          "customer_id": f"CUST-{random.randint(10000, 99999)}",
          "incident_date": (datetime.now() - timedelta(days=random.randint(0,
365))).isoformat()
      }
      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.

| **Scale Point (Kafka Partitions/ Spark Cores)** | **Ingestion Throughput (messages /sec)** | **End-to-End Latency (P99, ms)** | **FastAPI RPS (Average)** | **Kafka Lag (Avg Messages)** | **Spark CPU Util (Avg %)** | **Notes** |
|---|---|---|---|---|---|---|
| **Small (1-2 Kafka, 1 Spark Worker)** | 50-200 | 200-500 | 50-200 | < 1000 | 60-80% | CPU-bound, single-threaded bottlenecks possible |
| **Medium (3-5 Kafka, 2-3 Spark Workers)** | 200-800 | 100-300 | 200-800 | < 5000 | 50-70% | Increased parallelism, more stable performance |
| **Large (8-10 Kafka, 4-6 Spark Workers)** | 800-1500+ | 50-150 | 800-1500+ | < 10000 | 40-60% | Network/disk I/O can become bottleneck |

- **Key Takeaways:**
  - **Initial Bottleneck:** Often the FastAPI instance or network I/O if not optimized.
  - **Scaling Kafka:** Adding more partitions (and corresponding consumers) increases parallelism.

- **Scaling Spark:** More executors and cores lead to higher processing throughput, but also increased resource consumption.
- **Disk I/O:** MinIO/Delta Lake performance is heavily influenced by underlying disk speed.

### 7.5. Cost vs. Performance Analysis (Conceptual)

Understanding the trade-offs at different scales, particularly when considering cloud migration.

| Environment | Characteristics | Estimated Monthly Cost (Conceptual) | Performance Level | Cost-Benefit Observation |
|---|---|---|---|---|
| **Local Dev (Docker)** | Free software, personal hardware, no cloud cost. Excellent for rapid iteration. | $0 (excluding electricity/hardware depreciation) | Low-Medium (single machine limits) | Lowest cost, invaluable for early-stage development and learning. Not for production. |
| **Small Cloud Cluster (e.g., AWS MWAA, MSK Serverless, Glue, small EMR)** | Managed services reduce ops burden. Good for initial production, smaller datasets, or PoCs. | $100 - $1,000 | Medium (initial production scale, good for typical business loads) | Balance of cost and reduced operational overhead. Faster time to market than self-hosting. |
| **Large Prod Cluster (e.g., AWS MWAA, large MSK, EMR, dedicated EC2 instances)** | Highly available, multi-AZ, large instances, robust monitoring, dedicated operations. For high-volume, critical data. | $5,000 - $50,000+ | High (enterprise-grade throughput, low latency for critical apps) | Highest performance and reliability. Justified by critical business use cases and significant data volumes. Requires strong FinOps practices. |

# 8. Accelerating Onboarding & Developer Experience

Streamlining the onboarding process for new team members is crucial for productivity.

### 8.1. Quick-Start Checklist & Bootstrap Script Output

A single script to set up the entire local environment from scratch. This script assumes that conceptual code examples (like data_ingestor.py, Dockerfile, requirements.txt, data_transformer_spark.py, alloy-config.river, and data_generator.py) are located in a conceptual_code/ directory relative to where this quick_start.sh script is placed. It also assumes the full docker-compose.yml (provided in Appendix E) is present in the project root.

**Conceptual quick_start.sh:**

```
#!/bin/bash
# quick_start.sh - Bootstrap script for local data platform environment

# --- Prerequisites Check ---
echo "--- Checking Prerequisites ---"
command -v docker >/dev/null 2>&1 || { echo >&2 "Docker is required but not installed.
Aborting."; exit 1; }
command -v docker-compose >/dev/null 2>&1 || { echo >&2 "Docker Compose is required but
not installed. Aborting."; exit 1; }
command -v python3 >/dev/null 2>&1 || { echo >&2 "Python3 is required but not installed.
Aborting."; exit 1; }
command -v uv >/dev/null 2>&1 || { echo >&2 "uv (pip install uv) is recommended but not
installed. Proceeding anyway."; }
command -v sam >/dev/null 2>&1 || { echo >&2 "AWS SAM CLI is required but not installed.
Aborting."; exit 1; }
echo "All prerequisites appear to be met."

# --- Project Setup ---
echo "--- Setting up project directories ---"
mkdir -p
data/{postgres,mongodb,minio,spark-events,grafana,airflow_logs,openmetadata_mysql,open
metadata_elasticsearch}
mkdir -p
src/{common,models,fastapi_app_starter,fastapi_app_intermediate,fastapi_app_advanced}
mkdir -p pyspark_jobs/{tests/unit}
mkdir -p airflow_dags
mkdir -p
terraform_infra/{modules/{s3_data_lake,msk_kafka,rds_postgres},environments/{dev,staging,p
rod}}
mkdir -p
observability/{dashboards,grafana_dashboards_provisioning,grafana_datasources_provisionin
g}
mkdir -p openmetadata_ingestion_scripts
mkdir -p runbooks
echo "Project directories created."

echo "--- Copying conceptual code snippets to working directories ---"
# Copy FastAPI app code for each track
cp conceptual_code/fastapi_app_starter/* ./src/fastapi_app_starter/
cp conceptual_code/fastapi_app_intermediate/* ./src/fastapi_app_intermediate/
```

```
cp conceptual_code/fastapi_app_advanced/* ./src/fastapi_app_advanced/
cp conceptual_code/fastapi_app_dockerfile/Dockerfile ./fastapi_app/Dockerfile
cp conceptual_code/fastapi_app_dockerfile/requirements.txt ./fastapi_app/requirements.txt
# Copy PySpark jobs
cp conceptual_code/pyspark_jobs/* ./pyspark_jobs/
# Copy Airflow DAGs
cp conceptual_code/airflow_dags/* ./airflow_dags/
# Copy Observability config
cp conceptual_code/observability/alloy-config.river ./observability/alloy-config.river
cp conceptual_code/observability/health_dashboard.json
./observability/dashboards/health_dashboard.json
# Copy sample runbooks
cp conceptual_code/runbooks/* ./runbooks/
echo "Conceptual code snippets copied."

# --- Docker Compose Setup ---
echo "--- Bringing up Docker Compose services (Advanced Track) ---"
echo "This may take a few minutes for all services to start and stabilize."
docker compose -f docker-compose.yml up --build -d \
  zookeeper kafka minio minio_client cadvisor \
  fastapi_ingestor \
  postgres mongodb \
  spark-master spark-worker-1 spark-worker-2 spark-worker-3 spark-history-server \
  airflow-webserver airflow-scheduler \
  grafana grafana_alloy \
  spline-rest spline-ui \
  openmetadata-mysql openmetadata-elasticsearch openmetadata-server

# --- Initialize Airflow Database ---
echo "--- Initializing Airflow Database (first time setup) ---"
# Give postgres some time to fully initialize
sleep 20
docker compose up airflow-init
echo "Airflow database initialization started. Check logs if it fails."

echo "--- Waiting for services to become healthy (this may take a few more minutes) ---"
# Simple loop to wait for key services. For production, use better health checks.
services_to_check=("fastapi_ingestor" "kafka" "spark-master" "airflow-webserver" "grafana"
"openmetadata-server")
for service in "${services_to_check[@]}"; do
  echo "Waiting for $service..."
  until docker compose logs $service | grep -q "healthy"; do
    printf "."
```

```
    sleep 5
  done
  echo "$service is healthy."
done

echo "--- Environment Setup Complete! ---"
echo "You can now access the following UIs:"
echo "  MinIO Console: http://localhost:9901 (User: minioadmin, Pass: minioadmin)"
echo "  FastAPI Docs:  http://localhost:8000/docs"
echo "  Spark Master UI: http://localhost:8080"
echo "  Spark History Server UI: http://localhost:18080"
echo "  Airflow Web UI: http://localhost:8081 (User: airflow, Pass: airflow)"
echo "  Grafana Web UI: http://localhost:3000 (User: admin, Pass: admin)"
echo "  Spline UI:     http://localhost:9090"
echo "  OpenMetadata UI: http://localhost:8585"
echo ""
echo "To stop all services: docker compose down"
echo "To run the data generator: python conceptual_code/data_generator.py"
echo "Remember to update the IP address in data_generator.py if running from a different machine."
```

**Example Bootstrap Script Output Preview (docker compose up -d):**

When you run docker compose up -d, you'll see output similar to this, indicating containers being created and started:

```
[+] Running 20/20
 ✔ Container advanced-mongodb       Started
0.0s
 ✔ Container advanced-postgres      Started
0.0s
 ✔ Container advanced-minio         Started
0.0s
 ✔ Container advanced-zookeeper     Started
0.0s
 ✔ Container advanced-kafka         Started
0.0s
 ✔ Container advanced-cadvisor      Started
0.0s
 ✔ Container advanced-spark-master  Started
0.0s
 ✔ Container advanced-grafana_alloy Started
0.0s
 ✔ Container advanced-fastapi-ingestor Started
```

0.0s
✔ Container advanced-spark-worker-1 Started
0.0s
✔ Container advanced-spark-worker-2 Started
0.0s
✔ Container advanced-spark-worker-3 Started
0.0s
✔ Container advanced-spark-history-server Started
0.0s
✔ Container advanced-airflow-webserver Started
0.0s
✔ Container advanced-airflow-scheduler Started
0.0s
✔ Container advanced-grafana      Started
0.0s
✔ Container advanced-spline-rest   Started
0.0s
✔ Container advanced-spline-ui     Started
0.0s
✔ Container advanced-openmetadata-mysql Started
0.0s
✔ Container advanced-openmetadata-elasticsearch Started
0.0s

Subsequent checks by the quick_start.sh script will confirm health of key services.

## 8.2. Default Credentials & Environment Variables

For local development, consistent credentials simplify setup.
(Sensitive credentials should always be externalized in production.)

| Service | Default User | Default Password | Exposed Host Port | Internal Docker Hostname:Port (for containers) |
|---|---|---|---|---|
| PostgreSQL | user | password | 5432 | postgres:5432 |
| MongoDB | rootuser | rootpassword | 27017 | mongodb:27017 |
| MinIO | minioadmin | minioadmin | 9000 (API), 9901 (UI) | minio:9000 |
| Kafka | N/A | N/A | 9092 | kafka:29092 |
| FastAPI | N/A | N/A | 8000 | fastapi_ingestor:8000 |
| Airflow Web UI | airflow | airflow | 8081 | airflow-webserver:8080 |
| Grafana Web UI | admin | admin | 3000 | grafana:3000 |

| | | | | |
|---|---|---|---|---|
| OpenMetadata Web UI | admin (default setup) | admin (default setup) | 8585 | openmetadata-server:8585 |

## 8.3. Troubleshooting Tips

A quick reference for common issues.

- **"Container exited unexpectedly" / Service won't start:**
  - **Check logs:** docker compose logs <service_name>. Look for error messages (e.g., port conflicts, misconfigurations, resource limits).
  - **Resource limits:** Increase Docker Desktop's allocated CPU/Memory.
  - **Port conflicts:** Change exposed host ports in docker-compose.yml if another process is using it.
- **"Connection Refused" / Cannot connect between containers:**
  - **Verify Docker Network:** Ensure containers are in the same Docker network (default if not specified).
  - **Use Service Names:** Containers should refer to each other by their service_name (e.g., kafka:29092).
  - **Check depends_on:** Ensure services are healthy before dependents try to connect (condition: service_healthy).
  - **Firewall:** Temporarily disable local firewall on host, if applicable (Windows specific).
- **Airflow DAGs not appearing or running:**
  - **Scheduler Logs:** Check docker compose logs airflow-scheduler for parsing errors or database issues.
  - **Permissions:** Ensure airflow_dags/ has correct read permissions for the Airflow container (often AIRFLOW_UID helps).
  - **Database:** Verify PostgreSQL (postgres) container is healthy and accessible to Airflow.
- **Spark job hangs or fails:**
  - **Spark UI (http://localhost:8080):** Check "Running Applications" and "Completed Applications" for detailed logs, failed stages, and executor status.
  - **Worker Logs:** docker compose logs spark-worker-1 for specific worker issues.
  - **Resource Exhaustion:** Monitor host CPU/Memory. Increase Spark executor memory/cores in docker-compose.yml.
  - **Data Skew:** Large data imbalances can cause tasks to hang on specific executors.
- **MinIO issues:**
  - **Logs:** docker compose logs minio.
  - **Bucket Creation:** Ensure minio_client service (if used) ran successfully to create buckets. Check http://localhost:9901 for created buckets.

# 9. Cloud Migration Strategy: Focused on AWS

This appendix outlines a potential migration path for future hosting of this data platform on Amazon Web Services (AWS), detailing which AWS services would replace the local components and providing a step-by-step guide on how to get them up and running. This transition from local Docker Compose setup to AWS-managed services offers increased scalability, reliability, security, and reduced operational overhead.

## 9.1. Overview of AWS Service Replacements

| Local Component | AWS Service Replacement | Primary Role in AWS |
|---|---|---|
| Docker/Docker Compose | AWS Elastic Container Service (ECS) or AWS Fargate | Container orchestration and serverless compute for containerized applications. |
| Python | AWS Lambda, AWS Glue (PySpark), Amazon EMR (PySpark), AWS Fargate | Execution environment for Python code in various AWS services. |
| FastAPI | AWS Lambda (with Amazon API Gateway) or AWS Fargate (on ECS) | Scalable, serverless API endpoint for data ingestion. |
| Apache Kafka | Amazon Managed Streaming for Apache Kafka (MSK) | Fully managed, highly available Kafka cluster for streaming data. |
| Apache Spark | Amazon EMR (for managed clusters) or AWS Glue (for serverless Spark ETL) | Distributed processing engine for large-scale data transformations. |
| Delta Lake (files on MinIO) | Amazon S3 (Simple Storage Service) + AWS Glue Data Catalog | Scalable object storage for Delta Lake files; central metadata repository. |
| PostgreSQL | Amazon Relational Database Service (RDS) for PostgreSQL | Managed relational database service. |
| MongoDB | Amazon DocumentDB (with MongoDB compatibility) | Managed NoSQL document database service. |
| MinIO | Amazon S3 (Simple Storage Service) | Highly durable, scalable, and secure object storage. |
| Apache Airflow | Amazon Managed Workflows for Apache Airflow (MWAA) | Fully managed Apache Airflow service for pipeline orchestration. |
| OpenTelemetry | AWS Distro for OpenTelemetry (ADOT) + AWS X-Ray, AWS CloudWatch | Standardized telemetry data collection; distributed tracing and logging. |
| Grafana Alloy | AWS Distro for OpenTelemetry (ADOT) + Amazon Managed Grafana | Centralized telemetry collection; managed visualization dashboards. |
| Spline | Run on Amazon EMR or AWS | Automated data lineage |

| | | |
|---|---|---|
| | Glue (Spark-based) | tracking for Spark jobs within AWS. |
| OpenMetadata | Self-hosted on AWS EC2/ECS (backed by RDS/DocumentDB/OpenSearch Service) | Centralized data catalog and metadata management platform. |

## 9.2. Step-by-Step AWS Migration Guide with IaC Examples

This guide provides high-level steps for setting up the corresponding AWS services using Terraform. Detailed configurations will vary based on specific requirements and data volumes. Full Terraform modules would reside in terraform_infra/modules/.

1. **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.
2. **Amazon S3 (Data Lake Storage - Replaces MinIO):**
   - **Terraform Snippet (terraform_infra/modules/s3_data_lake/main.tf):**

```
# S3 Data Lake Module
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
}
```

3. **Amazon MSK (Managed Apache Kafka - Replaces Apache Kafka):**
   - **Terraform Snippet (terraform_infra/modules/msk_kafka/main.tf):**

```
# MSK Kafka Cluster Module
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
}
```

4. **AWS Lambda + Amazon API Gateway (FastAPI Replacement):**
   - **Terraform Snippet (terraform_infra/modules/lambda_api_ingestor/main.tf):**

```
# Lambda API Ingestor Module
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
}
```

5. **Amazon RDS for PostgreSQL (Relational Database - Replaces local PostgreSQL):**
   - **Terraform Snippet (terraform_infra/modules/rds_postgres/main.tf):**

```
# RDS PostgreSQL Module
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
}
```

6. **Amazon DocumentDB (MongoDB Compatible Database - Replaces local MongoDB):**
   - Creation steps via Console or AWS CLI. Terraform resources aws_docdb_cluster, aws_docdb_cluster_instance.
7. **Amazon EMR or AWS Glue (Spark Replacement):**
   - **Option A: Amazon EMR (Managed Spark Clusters):**
     - **Terraform Snippet (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):**
  - **Terraform Snippet (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.
```

8. **Amazon MWAA (Managed Workflows for Apache Airflow - Replaces Apache Airflow):**
   - Creation via Console or Terraform resources aws_mwaa_environment.
9. **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.
10. **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.

## 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.

- **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:**
        1. Run LocalStack (e.g., via Docker Compose).
        2. Configure your Python boto3 clients to point to LocalStack's endpoint URL (e.g., s3 = boto3.client('s3', endpoint_url='http://localhost:4566')).
        3. 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.

# 10. Innovation & Future Roadmap

This section outlines potential future enhancements and strategic directions for the data platform, focusing on continuous innovation and expanding its capabilities.

## 10.1. Next 6 Months Roadmap

- **Real-time DML CDC via Debezium Integration:**
    - **Goal:** Transition from batch-oriented or timestamp-based CDC to a robust, log-based Change Data Capture mechanism for critical source databases (e.g., PostgreSQL, MySQL).
    - **Technology:** Integrate **Debezium** with Kafka Connect. Debezium, an open-source distributed platform, builds on Apache Kafka and provides a set of Kafka Connect connectors that monitor database transaction logs.
    - **Benefit:** Enables true real-time synchronization of dimensional and fact data from OLTP systems to the Delta Lake. This will reduce data latency, simplify ingestion logic, and support more immediate analytical use cases. Captured change events will feed directly into Spark Structured Streaming jobs for processing and application of SCD logic.
- **Expanded ML Features with Feature Store Integration:**
    - **Goal:** Centralize the creation, management, and serving of machine learning features to promote reusability, consistency, and reduce model training/serving skew.
    - **Technology:** Research and implement an open-source **Feature Store** (e.g., Feast, Hopsworks Feature Store). A Feature Store acts as a centralized repository for curated features, enabling data scientists to discover, share, and reuse features across different ML models and projects.
    - **Benefit:** Accelerates the ML development lifecycle by providing a consistent source of truth for features. It streamlines the transition from offline feature engineering (for model training) to online feature serving (for real-time inference),

ensuring that features used in training are identical to those used in production.

## 10.2. Spotlight on Emerging Trends

- **Kubernetes-based Runtime:** Explore transitioning from Docker Compose to a Kubernetes-based runtime (e.g., EKS, AKS) when scaling requirements exceed Docker Compose's capabilities. This enables more robust orchestration, auto-scaling, self-healing, and sophisticated traffic management. It would involve packaging services as Kubernetes Deployments, StatefulSets, and potentially leveraging service meshes like Istio for advanced network control.
- **Decentralized Lakehouses & Data Formats:** Preview integrations with emerging data lakehouse formats like **Apache Iceberg** and **Apache Hudi**, alongside **Delta Lake**. These offer alternative transactional capabilities on data lakes, promoting interoperability with various query engines and potentially different optimization strategies.

## 10.3. Data Mesh Alignment

The modular design of this platform inherently aligns with Data Mesh principles, promoting decentralized data ownership and data as a product.

- **Data as a Product:** Each core component or pipeline (e.g., Financial Transactions Ingestion, Insurance Claims Processing) can be viewed as a "data product." The project structure (fastapi_app/, pyspark_jobs/) supports this by isolating domain-specific logic.
    - **Case Study Example:** The "Financial Transactions Data Product" team owns the FastAPI ingestor, the Kafka topic, the Spark streaming job, and the resulting Delta Lake tables (both raw and curated). They define the data contracts, manage the CI/CD pipeline for their components, and are responsible for the product's quality and observability. This shifts accountability and fosters domain expertise.
- **Decentralized Ownership:** Different teams can own different data domains or data products, fostering autonomy while adhering to platform-wide governance standards (e.g., common observability tools, shared metadata catalog).
- **Self-Serve Data Infrastructure:** The local environment and IaC examples are foundational steps towards providing self-serve capabilities, allowing data product teams to provision and manage their own infrastructure within defined guardrails.

## 10.4. Delta Sharing Use Case Example

Delta Sharing is an open protocol that enables secure, real-time data sharing between organizations or within an organization, regardless of the computing platform.

- **Use Case:** Securely sharing aggregated financial transaction data with a partner organization for fraud analysis, without replicating data.
- **Configuration (Conceptual steps in local dev):**
    1. **Enable Delta Sharing Server:** In a production setting, you'd deploy a Delta Sharing server (e.g., an OSS reference server or a commercial offering). For local dev, you might simulate this or use a simple HTTP server to serve the shared

manifest.

2. **Create a Share:** Define a "share" that includes the Delta Lake table(s) you want to share (e.g., curated.aggregated_financial_transactions).
3. **Grant Recipient Access:** Create a recipient and provide them with a credential file. This file contains a URL to the sharing server and a bearer token.
4. **Configure Row-Level Filtering (Advanced Governance):** For sensitive data, you can implement row-level filtering based on recipient identity (though this requires a more advanced sharing server configuration). For example, share only transactions pertaining to specific customer segments for a particular partner.
   - **SQL Example for Shared View:**
     CREATE SHARE fraud_analysis_share;
     ALTER SHARE fraud_analysis_share ADD TABLE curated.aggregated_financial_transactions;
     -- Optionally, create a view with row-level filtering for a specific recipient
     CREATE VIEW shared_aggregated_financial_transactions AS
     SELECT * FROM curated.aggregated_financial_transactions
     WHERE recipient_id = current_recipient_id(); -- Hypothetical function for recipient filtering
     ALTER SHARE fraud_analysis_share ADD TABLE shared_aggregated_financial_transactions;

5. **Recipient Access:** The partner uses standard data tools (Spark, Pandas, Power BI, Tableau) with the credential file to access the shared Delta Lake table as if it were a local table. They don't need to be Delta Lake users or have a specific cloud account.

# 11. Glossary

This section defines key concepts and components, serving as a quick reference.

- **ACID Transactions:** Atomicity, Consistency, Isolation, Durability – properties guaranteeing reliable database transactions.
- **Apache Airflow:** Open-source platform for programmatically authoring, scheduling, and monitoring workflows.
- **Apache Avro:** Row-oriented data serialization framework using JSON for schema, compact binary for data.
- **Apache Flink:** Open-source stream processing framework for high-throughput, low-latency applications.
- **Apache Hudi:** Open-source data lake platform that brings transactional capabilities to HDFS and cloud storage.
- **Apache Iceberg:** Open table format for huge analytic datasets, providing ACID transactions and schema evolution.
- **Apache Kafka:** Distributed streaming platform for high-throughput, fault-tolerant real-time data ingestion.

- **Apache Spark:** Unified analytics engine for large-scale batch and streaming data processing.
- **Apache Spark ML:** Scalable machine learning library built on Apache Spark.
- **API & Metadata Contracts:** Formal agreements defining data schemas (e.g., Avro, Protobuf) and ensuring adherence.
- **Apicurio:** Open-source schema registry and API design tool.
- **Attributes:** Descriptive characteristics within a dimension table.
- **AWS Kinesis:** Managed streaming data service in AWS.
- **AWS Lambda:** Serverless, event-driven compute service.
- **AWS SAM CLI:** Command-line tool for local serverless application development.
- **cAdvisor:** (Container Advisor) Daemon that collects and exports container performance metrics.
- **CDC (Change Data Capture):** Tracking data changes over time in a database.
- **CI/CD (Continuous Integration/Continuous Delivery):** Practices for rapid, reliable, and automated software delivery.
- **CloudFormation:** AWS Infrastructure as Code service for provisioning AWS resources.
- **Compaction (Delta Lake):** Combining small Delta Lake files into larger ones for performance.
- **Confluent Schema Registry:** Centralized service for managing and serving schemas for Kafka.
- **Containerization:** Packaging code and dependencies into isolated units (e.g., Docker).
- **Contract Testing:** Verifying adherence to shared data agreements between systems.
- **Data Cataloging:** Collecting, organizing, and managing metadata about data assets.
- **Data Encryption in Transit/At Rest:** Securing data during movement and storage.
- **Data Governance:** Framework for effective and responsible management of data assets.
- **Data Ingestion:** Collecting raw data from sources into storage.
- **Data Integrity:** Accuracy, consistency, and reliability of data.
- **Data Lakehouse Paradigm:** Combines data lake flexibility with data warehouse reliability (e.g., Delta Lake).
- **Data Lineage:** Record of data's lifecycle (origin, transformations, consumption).
- **Data Mesh:** Decentralized socio-technical approach to data management, treating data as a product.
- **Data Partitioning (Delta Lake):** Organizing data into logical segments for performance.
- **Data Quality Management:** Practices for ensuring data accuracy, completeness, etc.
- **Data-Rich Industries:** Sectors (finance, insurance) heavily reliant on vast, complex data.
- **Data Silos:** Isolated data repositories not easily shared.
- **Data Stores:** General term for any data repository.
- **Debezium:** Open-source platform for change data capture.
- **Delta Lake:** Open-source storage layer for data lakes with ACID transactions, schema enforcement, time travel.

- **Delta Sharing:** Open protocol for secure data sharing from Delta Lake.
- **Denormalization:** Intentionally introducing redundancy in database design for query performance.
- **Dimensional Modeling:** Data warehouse design technique using fact and dimension tables.
- **Distributed Processing:** Executing tasks concurrently across multiple computers (nodes).
- **Docker:** Software platform for creating and managing containers.
- **Docker Compose:** Tool for defining and running multi-container Docker applications.
- **Doppler:** SaaS centralized secrets management solution.
- **ECS-Local:** Tool for local testing of ECS task definitions.
- **ELT (Extract, Load, Transform):** Data integration where transformation occurs *after* loading into the target system.
- **ETL (Extract, Transform, Load):** Data integration where transformation occurs *before* loading into the target system.
- **Executor Cores/Memory (Spark):** CPU cores/RAM allocated to each Spark executor.
- **Fact Table:** Central table in dimensional modeling storing quantitative measures.
- **FastAPI:** High-performance Python web framework for building APIs.
- **Feast:** Open-source feature store for machine learning.
- **Grafana Alloy:** OpenTelemetry Collector distribution for telemetry data.
- **Grafana:** Open-source platform for data visualization and monitoring.
- **Great Expectations:** Open-source tool for data quality and data testing.
- **HashiCorp Vault:** Tool for managing secrets and protecting sensitive data.
- **H2O.ai:** Open-source, in-memory, distributed ML platform.
- **Hopsworks Feature Store:** Managed feature store for machine learning.
- **Indexing (Database):** Data structure for quick data location and access.
- **Istio:** Open-source service mesh for managing microservices.
- **JMeter:** Apache tool for load testing and performance measurement.
- **Kafka Producer:** Application component sending messages to Kafka topics.
- **Locust:** Open-source load testing tool for defining user behavior in Python.
- **Local Development Environment:** Self-contained setup mimicking production.
- **LocalStack:** Local cloud service emulator for AWS services.
- **Machine Learning (ML):** Building algorithms that learn from data.
- **MinIO:** S3-compatible object storage server for local data lake simulation.
- **MongoDB:** Open-source NoSQL document database.
- **Number of Executors (Spark):** Total Spark executors in a cluster.
- **Observability:** Inferring internal system state from telemetry data (metrics, logs, traces).
- **OLAP (Online Analytical Processing):** Querying method for multi-dimensional data analysis.
- **OLTP (Online Transactional Processing):** Processing transactions for data entry, updates.
- **OpenAPI (Swagger):** Specification for describing RESTful web services.

- **OpenMetadata:** Open-source metadata management platform and data catalog.
- **OpenTelemetry:** Open-source tools for standardized telemetry data collection.
- **Pact:** Consumer-driven contract testing framework.
- **PostgreSQL:** Powerful, open-source object-relational database.
- **Primary Key:** Field(s) uniquely identifying records in a table.
- **Protobuf (Protocol Buffers):** Language-neutral mechanism for serializing structured data.
- **PySpark:** Python API for Apache Spark.
- **pytest:** Python testing framework.
- **Real-time Analytics:** Analyzing data as it is generated for immediate insights.
- **Recovery Point Objective (RPO):** Max acceptable data loss.
- **Recovery Time Objective (RTO):** Max acceptable downtime.
- **Role-Based Access Control (RBAC):** Regulating access based on user roles.
- **Scikit-learn (sklearn):** Popular Python ML library.
- **SCD Type 2/3 (Slowly Changing Dimension):** Methods for handling changes in dimension attributes by preserving history.
- **Schema Enforcement:** Ensuring data conforms to a predefined schema.
- **Schema Evolution:** Modifying a table's schema without disrupting existing queries.
- **Schema Registry:** Centralized service for managing data schemas.
- **Secure Credential Management:** Securely storing/managing sensitive authentication information.
- **Serverless Emulation:** Running cloud serverless functions locally (e.g., AWS SAM CLI).
- **SLI (Service Level Indicator):** Quantitative measure of service level.
- **SLO (Service Level Objective):** Target value for an SLI.
- **SOPS (Secrets OPerationS):** Open-source tool for encrypting secrets in data files.
- **Spline:** Open-source tool for automated data lineage tracking in Apache Spark.
- **Star Schema:** Dimensional data model with a central fact table and surrounding dimension tables.
- **Streaming Data:** Continuously generated and processed data.
- **Testcontainers:** Library for programmatic setup/teardown of Docker containers for tests.
- **Terraform:** Infrastructure as Code (IaC) tool for defining/provisioning infrastructure.
- **Terragrunt:** Thin wrapper for Terraform, providing extra tools for working with multiple Terraform modules.
- **Time Travel:** Accessing/querying historical versions of a dataset (e.g., Delta Lake).
- **TPS (Transactions Per Second):** Measure of system throughput.
- **uv:** Ultrafast Python package installer.
- **Z-Ordering:** Multi-dimensional clustering technique for data skipping in Delta Lake.

# 12. Technology Index

This index provides quick references to key technologies discussed in this guide. (Page numbers are conceptual for this Markdown format).

# 13. Appendices

## Appendix A: Ingestion Setup Details

This appendix provides conceptual Python code for the FastAPI ingestion application for each of the progressive complexity tracks, demonstrating how the data handling evolves.

### A.1. Starter Track: FastAPI with Direct DB/MinIO Writes

In this track, the FastAPI application either writes directly to PostgreSQL for structured data or saves files directly to MinIO, simulating simple ingestion without a message queue.

- **src/fastapi_app_starter/main.py (Conceptual):**

```python
# src/fastapi_app_starter/main.py
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from datetime import datetime
import os
import json
from sqlalchemy import create_engine, Column, String, Float, DateTime
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from minio import Minio
from minio.error import S3Error

app = FastAPI(title="Starter Data Ingestor")

# --- Database Setup (PostgreSQL) ---
DATABASE_URL = os.getenv("DATABASE_URL",
```

```
"postgresql://user:password@postgres:5432/starter_db")
engine = create_engine(DATABASE_URL)
Base = declarative_base()
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

class FinancialTransactionDB(Base):
    __tablename__ = "financial_transactions"
    transaction_id = Column(String, primary_key=True, index=True)
    timestamp = Column(DateTime, index=True)
    account_id = Column(String)
    amount = Column(Float)
    currency = Column(String)
    transaction_type = Column(String)
    merchant_id = Column(String, nullable=True)
    category = Column(String, nullable=True)

Base.metadata.create_all(bind=engine) # Create tables on startup

# --- MinIO Setup ---
MINIO_HOST = os.getenv("MINIO_HOST", "minio:9000")
MINIO_ACCESS_KEY = os.getenv("MINIO_ACCESS_KEY", "minioadmin")
MINIO_SECRET_KEY = os.getenv("MINIO_SECRET_KEY", "minioadmin")
MINIO_BUCKET = os.getenv("MINIO_BUCKET", "raw-starter-data")

minio_client = Minio(
    MINIO_HOST,
    access_key=MINIO_ACCESS_KEY,
    secret_key=MINIO_SECRET_KEY,
    secure=False # Use True for HTTPS
)

# Ensure MinIO bucket exists
try:
    if not minio_client.bucket_exists(MINIO_BUCKET):
        minio_client.make_bucket(MINIO_BUCKET)
        print(f"Created MinIO bucket: {MINIO_BUCKET}")
except S3Error as e:
    print(f"Error checking/creating MinIO bucket: {e}")
    # Depending on env, you might want to exit or retry

# --- Pydantic Models for Input Validation ---
class FinancialTransaction(BaseModel):
    transaction_id: str
```

```
    timestamp: datetime
    account_id: str
    amount: float
    currency: str
    transaction_type: str
    merchant_id: str | None = None
    category: str | None = None

class InsuranceClaim(BaseModel):
    claim_id: str
    timestamp: datetime
    policy_number: str
    claim_amount: float
    claim_type: str
    claim_status: str
    customer_id: str
    incident_date: datetime

@app.get("/")
async def read_main():
    return {"message": "Welcome to Starter Data Ingestor API!"}

@app.get("/health")
async def health_check():
    # Simple health check, could expand to check DB/MinIO connectivity
    return {"status": "healthy"}

@app.post("/ingest-financial-transaction-db/")
async def ingest_financial_transaction_db(transaction: FinancialTransaction):
    db = SessionLocal()
    try:
        db_transaction = FinancialTransactionDB(**transaction.dict())
        db.add(db_transaction)
        db.commit()
        db.refresh(db_transaction)
        return {"message": "Financial transaction ingested to DB successfully",
"transaction_id": transaction.transaction_id}
    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Database ingestion failed: {e}")
    finally:
        db.close()
```

```
@app.post("/ingest-insurance-claim-file/")
async def ingest_insurance_claim_file(claim: InsuranceClaim):
    try:
        # Save claim data as a JSON file in MinIO
        object_name =
f"insurance_claims/{claim.timestamp.strftime('%Y/%m/%d')}/{claim.claim_id}.json"
        json_data = json.dumps(claim.dict(by_alias=True, exclude_unset=True),
default=str).encode('utf-8')
        minio_client.put_object(
            MINIO_BUCKET,
            object_name,
            data=io.BytesIO(json_data),
            length=len(json_data),
            content_type="application/json"
        )
        return {"message": "Insurance claim ingested to MinIO successfully", "claim_id":
claim.claim_id}
    except S3Error as e:
        raise HTTPException(status_code=500, detail=f"MinIO ingestion failed: {e}")
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"File ingestion failed: {e}")
```

### A.2. Intermediate Track: FastAPI with Kafka Producer

Here, the FastAPI application acts as a Kafka producer, sending ingested data to a Kafka topic. This decouples the ingestion API from storage, introducing asynchronous processing.

- **src/fastapi_app_intermediate/main.py (Conceptual):**

```
# src/fastapi_app_intermediate/main.py
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from kafka import KafkaProducer
import json
import os
from datetime import datetime

app = FastAPI(title="Intermediate Data Ingestor (Kafka Producer)")

# Kafka producer configuration
KAFKA_BROKER = os.getenv("KAFKA_BROKER", "kafka:29092") # Use 'kafka' as
hostname inside Docker network
KAFKA_TOPIC_FINANCIAL = os.getenv("KAFKA_TOPIC_FINANCIAL",
"raw_financial_transactions")
```

```
KAFKA_TOPIC_INSURANCE = os.getenv("KAFKA_TOPIC_INSURANCE",
"raw_insurance_claims")

producer = None

@app.on_event("startup")
async def startup_event():
    global producer
    try:
        producer = KafkaProducer(
            bootstrap_servers=[KAFKA_BROKER],
            value_serializer=lambda v: json.dumps(v, default=str).encode('utf-8'), # Handle
datetime serialization
            acks='all', # Ensure all in-sync replicas have received the message
            retries=3
        )
        print(f"Kafka producer connected to {KAFKA_BROKER}")
    except Exception as e:
        print(f"Failed to connect to Kafka: {e}")
        raise HTTPException(status_code=500, detail=f"Could not connect to Kafka: {e}")

@app.on_event("shutdown")
async def shutdown_event():
    if producer:
        producer.flush() # Ensure all buffered records are sent
        producer.close()
        print("Kafka producer closed.")

# Pydantic Models (same as Starter Track, for consistency)
class FinancialTransaction(BaseModel):
    transaction_id: str
    timestamp: datetime
    account_id: str
    amount: float
    currency: str
    transaction_type: str
    merchant_id: str | None = None
    category: str | None = None

class InsuranceClaim(BaseModel):
    claim_id: str
    timestamp: datetime
    policy_number: str
```

```
    claim_amount: float
    claim_type: str
    claim_status: str
    customer_id: str
    incident_date: datetime

@app.get("/")
async def read_main():
    return {"message": "Welcome to Intermediate Data Ingestor API!"}

@app.get("/health")
async def health_check():
    if producer and producer.bootstrap_connected():
        return {"status": "healthy", "kafka_status": "connected"}
    return {"status": "degraded", "kafka_status": "disconnected"}

@app.post("/ingest-financial-transaction/")
async def ingest_financial_transaction(transaction: FinancialTransaction):
    if not producer:
        raise HTTPException(status_code=503, detail="Kafka producer not initialized.")
    try:
        future = producer.send(KAFKA_TOPIC_FINANCIAL,
value=transaction.dict(by_alias=True, exclude_unset=True))
        record_metadata = await future # Await the delivery report
        print(f"Sent financial transaction to topic {record_metadata.topic} partition
{record_metadata.partition} offset {record_metadata.offset}")
        return {"message": "Financial transaction ingested successfully", "transaction_id":
transaction.transaction_id}
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed to send to Kafka: {e}")

@app.post("/ingest-insurance-claim/")
async def ingest_insurance_claim(claim: InsuranceClaim):
    if not producer:
        raise HTTPException(status_code=503, detail="Kafka producer not initialized.")
    try:
        future = producer.send(KAFKA_TOPIC_INSURANCE, value=claim.dict(by_alias=True,
exclude_unset=True))
        record_metadata = await future # Await the delivery report
        print(f"Sent insurance claim to topic {record_metadata.topic} partition
{record_metadata.partition} offset {record_metadata.offset}")
        return {"message": "Insurance claim ingested successfully", "claim_id":
claim.claim_id}
```

```
    except Exception as e:
      raise HTTPException(status_code=500, detail=f"Failed to send to Kafka: {e}")
```

### A.3. Advanced Track: FastAPI with OpenTelemetry Instrumentation

The Advanced track FastAPI app includes OpenTelemetry instrumentation for distributed tracing and metrics, enabling comprehensive observability. It maintains Kafka production capabilities.

- **src/fastapi_app_advanced/main.py (Conceptual):**

```
# src/fastapi_app_advanced/main.py
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from kafka import KafkaProducer
import json
import os
from datetime import datetime
import io

# OpenTelemetry Instrumentation
from opentelemetry.instrumentation.fastapi import FastAPIInstrumentor
from opentelemetry import trace
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExporter
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
# For metrics (optional, if you want to send custom metrics directly)
# from opentelemetry.metrics import get_meter_provider, set_meter_provider
# from opentelemetry.exporter.otlp.proto.grpc.metric_exporter import
OTLPMetricExporter
# from opentelemetry.sdk.metrics import MeterProvider
# from opentelemetry.sdk.metrics.export import PeriodicExportingMetricReader

# Configure OpenTelemetry
resource = Resource.create({"service.name": "fastapi-ingestor"})
trace.set_tracer_provider(TracerProvider(resource=resource))
tracer = trace.get_tracer(__name__)

# Configure OTLP gRPC exporter for traces (to Grafana Alloy)
otlp_exporter =
OTLPSpanExporter(endpoint=os.getenv("OTEL_EXPORTER_OTLP_ENDPOINT",
"grafana_alloy:4317"), insecure=True)
span_processor = BatchSpanProcessor(otlp_exporter)
trace.get_tracer_provider().add_span_processor(span_processor)
```

```
# For metrics, similar setup (example, not fully detailed here)
# meter_provider =
MeterProvider(metric_readers=[PeriodicExportingMetricReader(OTLPMetricExporter(en
dpoint="grafana_alloy:4317", insecure=True))], resource=resource)
# set_meter_provider(meter_provider)
# meter = get_meter_provider().get_meter(__name__)
# ingest_counter = meter.create_counter("ingested_records_total", description="Total
number of records ingested")


app = FastAPI(title="Advanced Data Ingestor (OpenTelemetry, Kafka Producer)")

# Instrument FastAPI
FastAPIInstrumentor.instrument_app(app)

# Kafka producer configuration (same as Intermediate Track)
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")

producer = None

@app.on_event("startup")
async def startup_event():
    global producer
    try:
        producer = KafkaProducer(
            bootstrap_servers=[KAFKA_BROKER],
            value_serializer=lambda v: json.dumps(v, default=str).encode('utf-8'),
            acks='all',
            retries=3
        )
        print(f"Kafka producer connected to {KAFKA_BROKER}")
    except Exception as e:
        print(f"Failed to connect to Kafka: {e}")
        raise HTTPException(status_code=500, detail=f"Could not connect to Kafka: {e}")

@app.on_event("shutdown")
async def shutdown_event():
    if producer:
```

```python
        producer.flush()
        producer.close()
        print("Kafka producer closed.")
    # Ensure OpenTelemetry exporters are shut down
    if trace.get_tracer_provider():
        trace.get_tracer_provider().shutdown()
    # if get_meter_provider():
    #     get_meter_provider().shutdown()


# Pydantic Models (same as Intermediate Track)
class FinancialTransaction(BaseModel):
    transaction_id: str
    timestamp: datetime
    account_id: str
    amount: float
    currency: str
    transaction_type: str
    merchant_id: str | None = None
    category: str | None = None

class InsuranceClaim(BaseModel):
    claim_id: str
    timestamp: datetime
    policy_number: str
    claim_amount: float
    claim_type: str
    claim_status: str
    customer_id: str
    incident_date: datetime

@app.get("/")
async def read_main():
    return {"message": "Welcome to Advanced Data Ingestor API!"}

@app.get("/health")
async def health_check():
    if producer and producer.bootstrap_connected():
        return {"status": "healthy", "kafka_status": "connected"}
    return {"status": "degraded", "kafka_status": "disconnected"}

@app.post("/ingest-financial-transaction/")
async def ingest_financial_transaction(transaction: FinancialTransaction):
```

```
   with tracer.start_as_current_span("ingest_financial_transaction_api"):
      if not producer:
         raise HTTPException(status_code=503, detail="Kafka producer not initialized.")
      try:
         # ingest_counter.add(1, {"transaction_type": "financial"}) # Example custom
metric
         future = producer.send(KAFKA_TOPIC_FINANCIAL,
value=transaction.dict(by_alias=True, exclude_unset=True))
         record_metadata = await future
         trace.get_current_span().set_attribute("kafka.offset", record_metadata.offset)
         trace.get_current_span().set_attribute("kafka.topic", record_metadata.topic)
         print(f"Sent financial transaction to topic {record_metadata.topic} partition
{record_metadata.partition} offset {record_metadata.offset}")
         return {"message": "Financial transaction ingested successfully",
"transaction_id": transaction.transaction_id}
      except Exception as e:
         trace.get_current_span().record_exception(e)
         raise HTTPException(status_code=500, detail=f"Failed to send to Kafka: {e}")

@app.post("/ingest-insurance-claim/")
async def ingest_insurance_claim(claim: InsuranceClaim):
   with tracer.start_as_current_span("ingest_insurance_claim_api"):
      if not producer:
         raise HTTPException(status_code=503, detail="Kafka producer not initialized.")
      try:
         # ingest_counter.add(1, {"transaction_type": "insurance"}) # Example custom
metric
         future = producer.send(KAFKA_TOPIC_INSURANCE,
value=claim.dict(by_alias=True, exclude_unset=True))
         record_metadata = await future
         trace.get_current_span().set_attribute("kafka.offset", record_metadata.offset)
         trace.get_current_span().set_attribute("kafka.topic", record_metadata.topic)
         print(f"Sent insurance claim to topic {record_metadata.topic} partition
{record_metadata.partition} offset {record_metadata.offset}")
         return {"message": "Insurance claim ingested successfully", "claim_id":
claim.claim_id}
      except Exception as e:
         trace.get_current_span().record_exception(e)
         raise HTTPException(status_code=500, detail=f"Failed to send to Kafka: {e}")
```

### A.4. fastapi_app/Dockerfile and requirements.txt

These files would remain largely consistent across Intermediate and Advanced tracks, with additional dependencies for OpenTelemetry in the Advanced track.

- **fastapi_app/Dockerfile (Conceptual):**
```
# fastapi_app/Dockerfile
FROM python:3.10-slim-buster

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY src/fastapi_app_advanced/main.py /app/app/main.py
# For Starter/Intermediate, adjust the COPY command to point to the correct main.py

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

- **fastapi_app/requirements.txt (Conceptual for Advanced):**
```
fastapi==0.104.1
uvicorn[standard]==0.24.0.post1
kafka-python==2.0.2
pydantic==2.5.2
python-dotenv==1.0.0 # For local .env files
SQLAlchemy==2.0.23 # For Starter Track if using Postgres
psycopg2-binary==2.9.9 # For Starter Track if using Postgres
minio==7.1.17 # For Starter Track if using MinIO

# OpenTelemetry dependencies for Advanced Track
opentelemetry-api==1.21.0
opentelemetry-sdk==1.21.0
opentelemetry-exporter-otlp-proto-grpc==1.21.0
opentelemetry-instrumentation-fastapi==0.43b0 # Or latest compatible
opentelemetry-instrumentation-httpx==0.43b0
opentelemetry-instrumentation-requests==0.43b0
# Add other instrumentation packages as needed
```

## Appendix B: Platform Architecture Diagram (PlantUML)

This appendix provides the PlantUML code for the proposed scalable architecture. You can paste this code into a .puml file and render it using a PlantUML renderer (e.g., PlantUML online server, VS Code extension).

```
@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
```

## Appendix C: API Consumption Flow Diagram (PlantUML)

This section visualizes how external systems and internal applications would interact with the FastAPI ingestion API.

```
@startuml
!theme toy
skinparam sequence {
  ArrowColor #A80036
  ActorBorderColor #A80036
  LifeLineBorderColor #A80036
  LifeLineBackgroundColor #F8F8F8
  ParticipantBorderColor #A80036
  ParticipantBackgroundColor #F8F8F8
```

```
    ParticipantFontName Arial
    ParticipantFontSize 12
    ParticipantFontColor #A80036
    ActorBackgroundColor #F8F8F8
    ActorFontName Arial
    ActorFontSize 12
    ActorFontColor #A80036
}
title API Consumption Flow

actor "Mobile App\n(e.g., Banking App)" as MobileApp
actor "Internal System\n(e.g., Claims Processing)" as InternalSystem
participant "FastAPI Ingestor\n(API Gateway/Load Balancer)" as FastAPI

box "Data Platform"
    participant "Kafka Topic\n(Raw Data)" as KafkaTopic
    participant "Spark Structured Streaming\n(Kafka Consumer)" as SparkConsumer
    database "Delta Lake\n(Raw Zone)" as DeltaLakeRaw
end box

MobileApp -> FastAPI : POST /ingest-financial-transaction (JSON)
activate FastAPI
FastAPI -> KafkaTopic : Produce message (FinancialTransaction)
deactivate FastAPI

InternalSystem -> FastAPI : POST /ingest-insurance-claim (JSON)
activate FastAPI
FastAPI -> KafkaTopic : Produce message (InsuranceClaim)
deactivate FastAPI

KafkaTopic -> SparkConsumer : Stream Data
activate SparkConsumer
SparkConsumer -> DeltaLakeRaw : Write to Delta (Parquet)
deactivate SparkConsumer
@enduml
```

## Appendix D: Data Flow Diagram (PlantUML)

This section provides the PlantUML code for the detailed data flow within the platform.

```
@startuml
!theme toy
skinparam activityBorderThickness 1
skinparam activityBorderColor black
```

```
skinparam activityArrowThickness 2
skinparam activityArrowColor #555
skinparam activityFontSize 14
skinparam activityFontName SansSerif
skinparam activityEndColor #FF6347
skinparam activityStartColor #7FFFD4
title Data Flow Diagram

start

partition "Data Ingestion" {
  :External Data Sources\n(Financial, Insurance);
  -> (HTTP/S POST)
  :FastAPI Ingestor;
  -> (Serialize to JSON, Publish)
  :Kafka Topic\n(raw_financial_insurance_data);
}

partition "Data Processing & Transformation" {
  :Spark Structured Streaming\n(Kafka Consumer);
  -> (Read Raw Data)
  :MinIO (Delta Lake Raw Zone);
  -> (Read Raw Data)
  :PySpark Transformation Job;
  -> (Data Cleansing, Dimensional Modeling,\nSchema Enforcement/Evolution, MERGE)
  :MinIO (Delta Lake Curated Zone);
  -> (Write to Curated Zone);
}

partition "Data Storage" {
  :MinIO (Delta Lake Raw Zone);
  :MinIO (Delta Lake Curated Zone);
  :PostgreSQL (e.g., Reference Data,\nAirflow Metadata);
  :MongoDB (e.g., Flexible Schemas,\nApplication Data);

  ' Internal data flows for storage interaction
  minio_curated <-- spark_transform
  postgres_db <-> spark_transform
  mongodb_db <-> spark_transform
}

partition "Analytics & Consumption" {
  :Spark SQL / MLlib Analytics\n(Anomaly Detection, Reporting);
```

```
  -> (Query Curated Data)
  :Business Users / Downstream Applications\n(e.g., BI Dashboards, Internal Systems);
  spark_analytics --> minio_curated
}

partition "Orchestration, Observability & Governance" {
  :Apache Airflow\n(Workflow Orchestration);
  -> (Schedule & Execute)
  :Spark Jobs;
  :OpenTelemetry\n(Instrumentation);
  -> (Generate Traces, Metrics, Logs)
  :Grafana Alloy\n(Telemetry Collector);
  -> (Forward)
  :Grafana (Monitoring);
  -> (Forward)
  :OpenMetadata (Data Catalog);
  :Spline (Spark Lineage);
  -> (Capture Lineage)
  :OpenMetadata (Data Catalog);
  :cAdvisor (Container Metrics);
  -> (Collect)
  :Grafana Alloy;

  airflow --> spark_transform : "Trigger"
  spark_analytics ..> openmetadata : "Metadata/Profiles"
  grafana_alloy --> openmetadata : "Telemetry Data/Metadata"
  spline --> openmetadata : "Lineage Metadata"
  openmetadata <-> "Analysts/Engineers" : "Data Discovery & Governance"
}

end
@enduml
```

## Appendix E: docker-compose.yml Full Configuration

This appendix provides the complete docker-compose.yml file, consolidating all service definitions for the local development.
The docker-compose.yml conceptual snippets for the Starter, Intermediate, and Advanced tracks are included within the document text.

- **Starter Track Conceptual docker-compose.yml Snippet:**
  ```
  # Simplified docker-compose.yml for Starter Track
  version: '3.8'
  services:
  ```

```yaml
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
fastapi_ingestor:
  build: ./fastapi_app # Path to your FastAPI Dockerfile
  container_name: starter-fastapi-ingestor
  restart: unless-stopped
  ports:
    - "8000:8000" # Expose FastAPI API port
  environment:
    # These variables would direct FastAPI to store data directly into Postgres or MinIO
    DATABASE_TYPE: "postgres" # Or "minio" for direct file writes
    POSTGRES_HOST: postgres
    POSTGRES_PORT: 5432
    MINIO_HOST: minio
    MINIO_PORT: 9000
```

```
    MINIO_ACCESS_KEY: minioadmin
    MINIO_SECRET_KEY: minioadmin
  volumes:
    # Mount application code for development and hot-reloading
    - ./src/fastapi_app_starter:/app/app # Simplified ingestor for direct DB/MinIO writes
  depends_on:
    postgres:
      condition: service_healthy # Ensure Postgres is ready
    minio:
      condition: service_healthy # Ensure MinIO is ready
```

- **Intermediate Track Conceptual docker-compose.yml Snippet:**

```
# Intermediate Track: Add Kafka & Spark for streaming
version: '3.8'
services:
  # ... (postgres, minio services - still present for reference/metadata) ...
  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
```

```yaml
      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
```

- **Advanced Track Conceptual docker-compose.yml Snippet:**

```yaml
# Advanced Track: Add Airflow, Observability, Lineage, Metadata
version: '3.8'
services:
  # ... (postgres, mongodb, minio, zookeeper, kafka, fastapi_ingestor,
spark-master/workers/history services) ...
  # 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
```

```yaml
      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
```

```yaml
    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
```

```yaml
  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:
```

```yaml
      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
```

## Appendix F: Testing Framework Detail Expansion

## Appendix F: Testing Framework Detail Expansion

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):**

```python
# 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.
- **Conceptual Locust Load Test Script (locust_fastapi_ingestor.py - Full script in Appendix H):**

```
# locust_fastapi_ingestor.py (Conceptual)
from locust import HttpUser, task, between
import json
from datetime import datetime, timedelta
import random

class FinancialDataUser(HttpUser):
  wait_time = between(0.1, 0.5) # Simulate delay between requests
  host = "http://localhost:8000" # Target FastAPI endpoint

  @task(1)
  def ingest_financial_transaction(self):
```

```
        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),
            "currency": random.choice(["USD", "EUR", "GBP"]),
            "transaction_type": random.choice(["debit", "credit", "transfer"]),
            "merchant_id": f"MER-{random.randint(100, 999)}" if random.random() > 0.3 else
None,
            "category": random.choice(["groceries", "utilities", "salary"])
        }
        self.client.post("/ingest-financial-transaction/", json=transaction_data,
name="/ingest-financial-transaction")

    @task(1)
    def ingest_insurance_claim(self):
        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),
            "claim_type": random.choice(["auto", "health", "home"]),
            "claim_status": random.choice(["submitted", "under_review", "approved"]),
            "customer_id": f"CUST-{random.randint(10000, 99999)}",
            "incident_date": (datetime.now() - timedelta(days=random.randint(0,
365))).isoformat()
        }
        self.client.post("/ingest-insurance-claim/", json=claim_data,
name="/ingest-insurance-claim")
```

## Appendix G: Disaster Recovery (DR) Runbook Examples

This appendix provides conceptual runbooks for critical systems. An example runbook for Airflow Metadata Database Recovery is provided:

**Airflow Metadata Database Recovery**

- **Incident:** Corruption or loss of the Airflow metadata PostgreSQL database.
- **Purpose:** Restore Airflow's operational state by recovering its metadata database.
- **RPO/RTO:** High impact, critical for DAG scheduling (e.g., RPO: 1 hour, RTO: 2 hours).
- **Steps:**

1. Stop Airflow Services: Stop airflow-webserver, airflow-scheduler, airflow-triggerer.
   docker compose stop airflow-webserver airflow-scheduler airflow-triggerer
2. Stop PostgreSQL: Stop the Airflow metadata database.
   docker compose stop postgres
3. Backup Current (Corrupted) DB (Optional but Recommended): If there's any chance of forensic analysis, backup the corrupted volume.
   docker cp postgres:/var/lib/postgresql/data ./data/postgres_corrupted_backup
4. **Restore PostgreSQL Volume:** Replace the current PostgreSQL data volume with a backup. This might involve:
   - Deleting the existing volume: docker volume rm data_ingestion_platform_data_postgres (if using named volumes).
   - Restoring from a snapshot or a file-level backup to ./data/postgres/.
   - Or, if using a fresh container, attach a restored data volume.
5. Start PostgreSQL: Start the restored PostgreSQL container.
   docker compose start postgres
   Verify health: docker compose logs postgres
6. **Verify Airflow DB Connection:** Use a psql client to confirm Airflow's user can connect and see the main_db.
7. Run Airflow DB Upgrade/Check: Sometimes, after restoration, Airflow might need to run a database upgrade command if schema mismatches occur.
   docker exec airflow-webserver airflow db check
   docker exec airflow-webserver airflow db upgrade (Use with caution)
8. Restart Airflow Services:
   docker compose start airflow-webserver airflow-scheduler airflow-triggerer
9. **Post-Recovery Monitoring:** Check Airflow Web UI (http://localhost:8081) for DAG status, task history, and scheduler health. Verify new DAG runs are triggered correctly.

- **Related Runbooks:** ./database_backup_strategy.md (general database backup)

## Appendix H: Quantitative Benchmarking Harness Details

## 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:

1. **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.
2. **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.
3. **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).
4. **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.
5. **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).
6. **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:

1. **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.
2. **Establish Baseline:** Run the system under a typical, low-load condition. Record baseline performance metrics (latency, throughput, resource usage) to understand normal operating characteristics.
3. **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).
4. **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.
5. **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.
6. **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.

| Scale Point (Kafka Partitions/Spark Cores) | Ingestion Throughput (messages/sec) | End-to-End Latency (P99, ms) | FastAPI RPS (Average) | Kafka Lag (Avg Messages) | Spark CPU Util (Avg %) | Notes |
|---|---|---|---|---|---|---|
| **Small** (1-2 Kafka, 1 Spark Worker) | 50-200 | 200-500 | 50-200 | < 1000 | 60-80% | CPU-bound, single-threaded bottlenecks possible for higher loads. Good for initial functional testing. |
| **Medium** (3-5 Kafka, 2-3 Spark Workers) | 200-800 | 100-300 | 200-800 | < 5000 | 50-70% | Increased parallelism across Kafka and Spark. More stable performance under moderate loads. Balances resource consumption with throughput. |
| **Large** (8-10 Kafka, 4-6 Spark Workers) | 800-1500+ | 50-150 | 800-1500+ | < 10000 | 40-60% | Approaching limits of a single local machine. Network/disk I/O can become the bottleneck. Requires careful tuning of Spark configuratio |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | ns like spark.sql.shuffle.partitions and consideration of memory management. |

**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 I: AWS IaC Snippets

This appendix provides conceptual Terraform Infrastructure as Code (IaC) snippets for deploying various components of the data platform on AWS.

- **Lambda API Ingestor Module (terraform_infra/modules/lambda_api_ingestor/main.tf):**

```
# Lambda API Ingestor Module
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 (terraform_infra/modules/rds_postgres/main.tf):**

```
# RDS PostgreSQL Module
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):** Creation steps via Console or AWS CLI. Terraform resources aws_docdb_cluster, aws_docdb_cluster_instance.
- **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.
- **Hybrid Testing with LocalStack/ECS-Local:**
    - **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')).
    - **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.