Deep-Dive Addendum: Progressive Path Setup Guide

This addendum provides a practical, step-by-step guide for setting up each track of the "Progressive Complexity Path" on your local development machine using Docker Compose. It outlines the core steps and configuration required to bring up the various components of your enterprise data platform locally.

Prerequisites

Before starting, ensure you have the following installed:

- **Docker Desktop:** (or Docker Engine on Linux) for running containers.
- **Git:** For cloning the project repository.
- **Python 3.x:** With pip for installing dependencies.
- **docker-compose:** (usually included with Docker Desktop, or installed separately).

Project Structure Reminder:

This guide assumes you have cloned the data-ingestion-platform mono-repo, which contains all the necessary Dockerfiles, application code, and docker-compose.yml configurations.

```
data-ingestion-platform/
├── data/                 # Persistent Docker volumes for all services
├── src/                  # Core Python application logic
├── fastapi_app/          # FastAPI ingestion service
├── pyspark_jobs/         # Apache Spark transformation jobs (PySpark)
├── airflow_dags/         # Apache Airflow DAG definitions
├── observability/        # Grafana dashboards, Grafana Alloy configurations
├── openmetadata_ingestion_scripts/ # Python scripts for OpenMetadata connectors
├── docker-compose.yml    # Central Docker Compose file for local environment
└── README.md
```

Onboarding Script and External Data Generation

To streamline the onboarding process and facilitate testing with realistic data, a conceptual onboarding script and an external data generator are utilized.

Onboarding Script

Role Needed: Data Engineer, Developer

This script automates the initial setup of the local environment, ensuring consistency across development machines. It performs tasks such as checking prerequisites, initializing Docker Compose, creating necessary Kafka topics and S3 buckets (MinIO), and setting up initial database schemas.

Conceptual onboard.sh script:

```
#!/bin/bash
# onboard.sh - Onboarding script for local data platform environment

echo "Starting local data platform environment onboarding..."

# --- 1. Check Prerequisites ---
echo "Checking prerequisites (Docker, Git, Python, docker-compose)..."
command -v docker >/dev/null 2>&1 || { echo >&2 "Docker is not installed. Please install Docker
Desktop or Docker Engine."; exit 1; }
command -v docker-compose >/dev/null 2>&1 || { echo >&2 "Docker Compose is not installed.
Please install it."; exit 1; }
command -v python3 >/dev/null 2>&1 || { echo >&2 "Python 3 is not installed. Please install it.";
exit 1; }
echo "Prerequisites met."

# --- 2. Build and Start Core Services (using the main docker-compose.yml) ---
echo "Building and starting core services via docker compose..."
docker compose up --build -d --remove-orphans

# Give services some time to start up and become healthy
echo "Waiting for services to become healthy (this may take a few minutes)..."
# You might add more specific health checks here, e.g., waiting for FastAPI /health endpoint
sleep 60 # Arbitrary wait time, adjust as needed

# --- 3. Initialize Kafka Topics (if Kafka is part of the track) ---
# This part assumes Kafka is running and accessible within the Docker network
if docker ps --format "{{.Names}}" | grep -q "kafka"; then
  echo "Initializing Kafka topics..."
  # Create raw financial transactions topic
  docker exec -it kafka kafka-topics --create --topic raw_financial_transactions
--bootstrap-server kafka:29092 --partitions 3 --replication-factor 1 --if-not-exists
  # Create raw insurance claims topic
  docker exec -it kafka kafka-topics --create --topic raw_insurance_claims --bootstrap-server
kafka:29092 --partitions 3 --replication-factor 1 --if-not-exists
  echo "Kafka topics created."
else
  echo "Kafka service not detected, skipping topic creation."
fi

# --- 4. Initialize MinIO Buckets (if MinIO is part of the track) ---
# This part assumes MinIO is running and accessible
if docker ps --format "{{.Names}}" | grep -q "minio"; then
  echo "Initializing MinIO buckets..."
```

```
  # Ensure mc client is available in a separate service or installed on host
  # For simplicity, we assume mc is available within a utility container or you create buckets manually via API/UI
  # Example if using 'mc' client directly (might need a dedicated 'minio-client' service in docker-compose)
  # docker exec -it minio-client mc alias set local http://minio:9000 minioadmin minioadmin
  # docker exec -it minio-client mc mb local/raw-data-bucket --ignore-existing
  # docker exec -it minio-client mc mb local/curated-data-bucket --ignore-existing

  # Alternative: Use Python MinIO client to create buckets from FastAPI or another init script
  echo "Please ensure raw-data-bucket and curated-data-bucket are created in MinIO manually or via an automated script."
  echo "You can access MinIO console at http://localhost:9001 and create them manually."
else
  echo "MinIO service not detected, skipping bucket creation."
fi

# --- 5. Database Schema Initialization (if applicable) ---
# For PostgreSQL, you might have a script to apply migrations
if docker ps --format "{{.Names}}" | grep -q "postgres"; then
  echo "Applying PostgreSQL database migrations..."
  # Example: Run a Flyway/Alembic migration script from a dedicated container or a FastAPI init script
  echo "Ensure your FastAPI service or a dedicated migration container handles database schema initialization."
fi

echo "Onboarding complete. Your local data platform should now be running."
echo "Access FastAPI at http://localhost:8000"
echo "Access MinIO Console at http://localhost:9001"
echo "Access Airflow UI at http://localhost:8080 (if Advanced Track is enabled)"
```

External Data Generator

Role Needed: Data Analyst, QA Engineer, Developer
The external data generator is a standalone Python script (e.g., using Locust, as detailed in the Testing & Observability Patterns Deep-Dive Addendum) that simulates incoming data from various sources. This is critical for testing the ingestion layer and populating the data lake with mock, yet realistic, data volumes to simulate production load.

- **Purpose:** To continuously send mock financial transactions and insurance claims to the FastAPI Ingestor, allowing for load testing, functional validation, and populating the data pipeline for downstream processing and analysis.
- **Usage:** Run this script from your local machine *after* the FastAPI Ingestor service is up

and running. It interacts with the exposed API endpoint.

Conceptual Python script (simulate_data.py - simplified version, not a full Locust script):

```
# simulate_data.py
import requests
import json
import time
from datetime import datetime, timedelta
import random

FASTAPI_URL = "http://localhost:8000" # Ensure this matches your FastAPI exposure
DELAY_SECONDS = 0.1 # Time between sending each record

def generate_financial_transaction():
    """Generates a mock financial transaction."""
    return {
        "transaction_id":
f"FT-{datetime.now().strftime('%Y%m%d%H%M%S%f')}-{random.randint(1000, 9999)}",
        "timestamp": datetime.now().isoformat(),
        "account_id": f"ACC-{random.randint(100000, 999999)}",
        "amount": round(random.uniform(1.0, 10000.0), 2),
        "currency": random.choice(["USD", "EUR", "GBP", "JPY"]),
        "transaction_type": random.choice(["debit", "credit", "transfer", "payment"]),
        "merchant_id": f"MER-{random.randint(100, 999)}" if random.random() > 0.3 else None,
        "category": random.choice(["groceries", "utilities", "salary", "entertainment", "transport",
"housing", "healthcare", "education"])
    }

def generate_insurance_claim():
    """Generates a mock insurance claim."""
    return {
        "claim_id":
f"IC-{datetime.now().strftime('%Y%m%d%H%M%S%f')}-{random.randint(1000, 9999)}",
        "timestamp": datetime.now().isoformat(),
        "policy_number": f"POL-{random.randint(1000000, 9999999)}",
        "claim_amount": round(random.uniform(500.0, 50000.0), 2),
        "claim_type": random.choice(["auto", "health", "home", "life", "property"]),
        "claim_status": random.choice(["submitted", "under_review", "approved", "rejected",
"paid"]),
        "customer_id": f"CUST-{random.randint(10000, 99999)}",
        "incident_date": (datetime.now() - timedelta(days=random.randint(0, 365))).isoformat()
    }

if __name__ == "__main__":
```

```
    print(f"Starting data generation. Sending to {FASTAPI_URL}")
    while True:
        try:
            # Send financial transaction
            financial_data = generate_financial_transaction()
            response_ft = requests.post(f"{FASTAPI_URL}/ingest-financial-transaction/",
json=financial_data)
            if response_ft.status_code == 200:
                print(f"Sent financial transaction {financial_data['transaction_id']} (Status:
{response_ft.status_code})")
            else:
                print(f"Error sending financial transaction: {response_ft.status_code} -
{response_ft.text}")

            time.sleep(DELAY_SECONDS)

            # Send insurance claim
            insurance_data = generate_insurance_claim()
            response_ic = requests.post(f"{FASTAPI_URL}/ingest-insurance-claim/",
json=insurance_data)
            if response_ic.status_code == 200:
                print(f"Sent insurance claim {insurance_data['claim_id']} (Status:
{response_ic.status_code})")
            else:
                print(f"Error sending insurance claim: {response_ic.status_code} -
{response_ic.text}")

            time.sleep(DELAY_SECONDS)

        except requests.exceptions.ConnectionError as e:
            print(f"Connection error: {e}. Is FastAPI running at {FASTAPI_URL}? Retrying in 5
seconds...")
            time.sleep(5)
        except Exception as e:
            print(f"An unexpected error occurred: {e}")
            time.sleep(5)
```

Setup Guide by Track

The docker-compose.yml provided with the project is designed to be flexible. You will often comment out or enable services based on the track you are working on.

3.1. Starter Track Setup: Minimal Single-Machine Setup

This track focuses on FastAPI, PostgreSQL, and MinIO.

1. **Navigate to Project Root:**
   cd /path/to/your/data-ingestion-platform

2. **Prepare docker-compose.yml:**
   - Open docker-compose.yml.
   - **Uncomment** the services for fastapi_ingestor, postgres, minio.
   - **Comment out** all other services (zookeeper, kafka, spark, airflow, openmetadata, grafana, etc.) to keep the setup minimal.
   - Ensure the data/postgres, data/minio directories exist or are created by Docker Compose for persistent volumes.

   Conceptual Snippet of docker-compose.yml for Starter Track:# docker-compose.yml (Starter Track focus)

```
version: '3.8'
services:
  postgres:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: main_db
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    ports:
      - "5432:5432"
    volumes:
      - ./data/postgres:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user -d main_db"]
      interval: 5s
      timeout: 5s
      retries: 5

  minio:
    image: minio/minio:latest
    environment:
      MINIO_ROOT_USER: minioadmin
      MINIO_ROOT_PASSWORD: minioadmin
    ports:
      - "9000:9000"
      - "9001:9001" # Console port
    volumes:
      - ./data/minio:/data
    command: server /data --console-address ":9001"
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]
```

```
    interval: 30s
    timeout: 20s
    retries: 3

  fastapi_ingestor:
    build: ./fastapi_app
    environment:
      POSTGRES_HOST: postgres
      POSTGRES_DB: main_db
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      MINIO_HOST: minio:9000
      MINIO_ACCESS_KEY: minioadmin
      MINIO_SECRET_KEY: minioadmin
      MINIO_BUCKET: raw-data-bucket
    ports:
      - "8000:8000"
    depends_on:
      postgres:
        condition: service_healthy
      minio:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health || exit 1"]
      interval: 5s
      timeout: 3s
      retries: 5
  # ... other services commented out
```

3. **Bring Up Services:**
   docker compose up --build -d

   This command builds the Docker images (if necessary) and starts the selected services in detached mode.
4. **Verify Setup:**
   - Access FastAPI health check: http://localhost:8000/health
   - Access MinIO Console: http://localhost:9001 (login with minioadmin/minioadmin)
   - Use a PostgreSQL client to connect to localhost:5432 with user user, password password, database main_db.
   - Check Docker logs: docker compose logs -f

3.2. Intermediate Track Setup: Adding Streaming Capabilities

This track adds Apache Kafka and Apache Spark to the Starter Track components.

1. **Navigate to Project Root:**
   cd /path/to/your/data-ingestion-platform

2. **Prepare docker-compose.yml:**
    - Open docker-compose.yml.
    - **Uncomment** (or keep uncommented from Starter Track) fastapi_ingestor, postgres, minio.
    - **Uncomment** the services for zookeeper, kafka, and spark.
    - **Comment out** other Advanced Track services (airflow, openmetadata, grafana, etc.).
    - Update fastapi_ingestor to publish to Kafka (remove direct MinIO/PostgreSQL writes, or add Kafka producer logic). Ensure KAFKA_BROKER environment variable points to kafka:29092.
    - Ensure spark service configuration is set to connect to kafka and minio. Set checkpoint locations for Spark Structured Streaming.
    - Ensure data/spark-events directory exists for Spark history server.

Conceptual Snippet of docker-compose.yml for Intermediate Track (partial, focusing on additions):

```
# docker-compose.yml (Intermediate Track focus)
version: '3.8'
services:
  # ... postgres, minio (from Starter Track)

  zookeeper:
    image: confluentinc/cp-zookeeper:7.4.0
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    healthcheck:
      test: ["CMD-SHELL", "echo stat | nc localhost 2181"] # Basic Zookeeper health
      interval: 5s
      timeout: 5s
      retries: 10

  kafka:
    image: confluentinc/cp-kafka:7.4.0
    depends_on:
      zookeeper:
        condition: service_healthy
    ports:
      - "9092:9092" # External for host tools
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_DELETE_TOPIC_ENABLE: "true" # For development
```

```
    healthcheck:
      test: ["CMD-SHELL", "kafka-topics --bootstrap-server localhost:9092 --list"]
      interval: 10s
      timeout: 5s
      retries: 10

  fastapi_ingestor:
    build: ./fastapi_app
    environment:
      # ... (Postgres, Minio from Starter Track if still needed for metadata)
      KAFKA_BROKER: kafka:29092 # New: FastAPI publishes to Kafka
      KAFKA_TOPIC_FINANCIAL: raw_financial_transactions
      KAFKA_TOPIC_INSURANCE: raw_insurance_claims
    depends_on:
      # ... postgres, minio
      kafka: # New dependency
        condition: service_healthy
    # ... healthcheck

  spark:
    image: bitnami/spark:3.5.0
    command: ["tail", "-f", "/dev/null"] # Keep container alive for spark-submit
    environment:
      SPARK_MASTER_URL: "spark://spark-master:7077" # Assuming spark-master service
      SPARK_LOCAL_IP: spark # For internal networking
      SPARK_DAEMON_JAVA_OPTS:
"-Dspark.history.fs.logDirectory=file:///opt/bitnami/spark/spark-events"
      # Kafka connectivity for Spark jobs
      KAFKA_BROKER_ADDRESS: kafka:29092
      MINIO_HOST: minio
      MINIO_ACCESS_KEY: minioadmin
      MINIO_SECRET_KEY: minioadmin
      MINIO_BUCKET_RAW: raw-data-bucket
      MINIO_BUCKET_CURATED: curated-data-bucket
    volumes:
      - ./pyspark_jobs:/opt/bitnami/spark/jobs # Mount your PySpark jobs
      - ./data/spark-events:/opt/bitnami/spark/spark-events # For Spark History Server
    depends_on:
      kafka:
        condition: service_healthy
      minio:
        condition: service_healthy
    # No exposed ports if only used internally by Airflow or manual spark-submit
    # If you need Spark UI, expose 8080:8080 (master) and 8081:8081 (worker) for
manual debugging

  # Optional: Spark History Server
  spark-history-server:
```

```
  image: bitnami/spark:3.5.0
  command: ["/opt/bitnami/spark/bin/spark-history-server.sh"]
  environment:
   SPARK_HISTORY_FS_LOGDIRECTORY: "/opt/bitnami/spark/spark-events"
  ports:
   - "18080:18080" # Spark History UI
  volumes:
   - ./data/spark-events:/opt/bitnami/spark/spark-events
  depends_on:
   spark:
    condition: service_started
# ... other services commented out
```

3. **Bring Up Services:**
   docker compose up --build -d

4. **Verify Setup:**
   - Verify Starter Track components are running.
   - Check Kafka topic creation: docker exec -it kafka kafka-topics --bootstrap-server localhost:9092 --list
   - Run a sample Spark streaming job (e.g., via docker exec -it spark spark-submit /opt/bitnami/spark/jobs/streaming_consumer.py <args>).
   - Check Spark History Server: http://localhost:18080 (if enabled).

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

This track integrates orchestration, observability, lineage, and metadata management.

1. **Navigate to Project Root:**
   cd /path/to/your/data-ingestion-platform

2. **Prepare docker-compose.yml:**
   - **Uncomment** all services including airflow-init, airflow-webserver, airflow-scheduler, airflow-worker, mongodb, openmetadata, grafana, grafana-alloy, cAdvisor, spline.
   - Ensure all necessary environment variables for inter-service communication are correctly set (e.g., Airflow connecting to Spark, OpenMetadata connecting to PostgreSQL/MongoDB, Grafana Alloy pointing to Prometheus/OpenTelemetry targets).
   - Mount airflow_dags and observability directories as volumes for Airflow DAGs and Grafana configurations.
   - Ensure all data/ subdirectories for persistent volumes exist.

   Conceptual Snippet of docker-compose.yml for Advanced Track (partial, focusing on additions):
```
# docker-compose.yml (Advanced Track focus)
version: '3.8'
services:
```

```
  # ... postgres, minio, zookeeper, kafka, spark, spark-history-server

  mongodb:
    image: mongo:6.0
    ports:
      - "27017:27017"
    volumes:
      - ./data/mongodb:/data/db
    healthcheck:
      test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]
      interval: 5s
      timeout: 5s
      retries: 5

  airflow-init:
    image: apache/airflow:2.8.0
    entrypoint: ["/bin/bash", "-c"]
    command:
      - "airflow db migrate && airflow users create --username admin --password admin
--firstname Admin --lastname User --role Admin --email admin@example.com"
    environment:
      _PIP_ADDITIONAL_REQUIREMENTS: "apache-airflow-providers-cncf-kubernetes
apache-airflow-providers-apache-kafka apache-airflow-providers-cncf-kubernetes
apache-airflow-providers-postgres delta-spark"
      AIRFLOW_HOME: /opt/airflow
      AIRFLOW__CORE__LOAD_EXAMPLES: "false"
      AIRFLOW__DATABASE__SQL_ALCHEMY_CONN:
"postgresql+psycopg2://user:password@postgres/main_db"
      AIRFLOW__WEBSERVER__RBAC: "True"
      AIRFLOW__CORE__DAGS_FOLDER: /opt/airflow/dags
    volumes:
      - ./airflow_dags:/opt/airflow/dags
      - ./src:/opt/airflow/src # Mount source code for Airflow to access
    depends_on:
      postgres:
        condition: service_healthy

  airflow-webserver:
    image: apache/airflow:2.8.0
    command: webserver
    ports:
      - "8080:8080" # Airflow UI
    environment:
      AIRFLOW_HOME: /opt/airflow
      AIRFLOW__CORE__LOAD_EXAMPLES: "false"
      AIRFLOW__DATABASE__SQL_ALCHEMY_CONN:
"postgresql+psycopg2://user:password@postgres/main_db"
      AIRFLOW__WEBSERVER__RBAC: "True"
```

```
      AIRFLOW__CORE__DAGS_FOLDER: /opt/airflow/dags
    volumes:
      - ./airflow_dags:/opt/airflow/dags
      - ./src:/opt/airflow/src
    depends_on:
      airflow-init:
        condition: service_completed_successfully

  airflow-scheduler:
    image: apache/airflow:2.8.0
    command: scheduler
    environment:
      AIRFLOW_HOME: /opt/airflow
      AIRFLOW__CORE__LOAD_EXAMPLES: "false"
      AIRFLOW__DATABASE__SQL_ALCHEMY_CONN:
"postgresql+psycopg2://user:password@postgres/main_db"
      AIRFLOW__WEBSERVER__RBAC: "True"
      AIRFLOW__CORE__DAGS_FOLDER: /opt/airflow/dags
      AIRFLOW__CORE__EXECUTOR: LocalExecutor # For local testing; CeleryExecutor for
distributed
    volumes:
      - ./airflow_dags:/opt/airflow/dags
      - ./src:/opt/airflow/src
    depends_on:
      airflow-webserver:
        condition: service_healthy

  # Airflow Worker (optional for local, if using CeleryExecutor)
  # airflow-worker:
  #   image: apache/airflow:2.8.0
  #   command: worker
  #   environment:
  #     # ... same as scheduler, plus Celery config
  #   depends_on:
  #     airflow-scheduler:
  #       condition: service_healthy

  openmetadata:
    image: openmetadata/openmetadata:1.2.3-release
    ports:
      - "8585:8585" # OpenMetadata UI
    environment:
      # ... OpenMetadata configuration for MySQL, Elasticsearch (using data/ volumes)
      # Make sure it points to your local Postgres for ingestion metadata
      MYSQL_HOST: openmetadata_mysql # Or your specific MySQL service
      ELASTICSEARCH_HOST: openmetadata_elasticsearch # Or your specific ES service
    depends_on:
      # ... openmetadata_mysql, openmetadata_elasticsearch (or equivalent)
```

```
  spline:
    image: absaoss/spline-rest-server:0.7.1 # Or latest
    ports:
      - "8081:8081" # Spline UI
    environment:
      SPLINE_DATABASE_URL: "jdbc:postgresql://postgres:5432/main_db"
      SPLINE_DATABASE_USERNAME: user
      SPLINE_DATABASE_PASSWORD: password
    depends_on:
      postgres:
        condition: service_healthy
      spark:
        condition: service_started

  grafana:
    image: grafana/grafana-oss:10.2.0 # Or latest
    ports:
      - "3000:3000"
    volumes:
      - ./data/grafana:/var/lib/grafana
      -
./observability/grafana_datasources_provisioning:/etc/grafana/provisioning/datasources
      -
./observability/grafana_dashboards_provisioning:/etc/grafana/provisioning/dashboards
    environment:
      GF_AUTH_ANONYMOUS_ENABLED: "true" # For quick local access
      GF_AUTH_ANONYMOUS_ORG_ROLE: Admin
      GF_SERVER_ROOT_URL: "http://localhost:3000"
      GF_INSTALL_PLUGINS: "grafana-piechart-panel" # Example plugin
    depends_on:
      grafana-alloy:
        condition: service_healthy

  grafana-alloy:
    build:
      context: ./observability
      dockerfile: Dockerfile.alloy # Custom Dockerfile for Grafana Alloy config
    volumes:
      - ./observability/alloy-config.river:/etc/alloy/config.river
    command: ["--config.file=/etc/alloy/config.river"]
    ports:
      - "12345:12345" # OpenTelemetry receiver
      - "9090:9090" # Prometheus metrics endpoint for self-monitoring
    depends_on:
      # Link to services it collects from
      fastapi_ingestor:
        condition: service_healthy
```

```
      kafka:
        condition: service_healthy
      spark:
        condition: service_healthy
      cAdvisor:
        condition: service_healthy

  cAdvisor:
    image: gcr.io/cadvisor/cadvisor:v0.47.0 # Or latest stable
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
      - /dev/disk/:/dev/disk:ro
    privileged: true
    ports:
      - "8080:8080" # cAdvisor UI (can clash with Airflow, adjust if needed)
    # No healthcheck for simplicity, but a proper one would check metrics endpoint
```

3. **Bring Up Services:**
   docker compose up --build -d

4. **Verify Setup:**
   - Access Airflow UI: http://localhost:8080 (login admin/admin)
   - Access Grafana UI: http://localhost:3000 (initially anonymous or configure admin user)
   - Access OpenMetadata UI: http://localhost:8585
   - Verify Spline UI: http://localhost:8081
   - Check for container metrics in Grafana dashboards.
   - Ensure Airflow DAGs appear and run as expected.
   - Run data ingestion, processing, and observe metrics, lineage, and metadata.

This guide provides the core steps. Remember that exact configurations may vary based on your specific implementation of each component. Always consult the detailed documentation for each individual technology if you encounter issues.