Data Platform Local Environment Walkthrough & QA Test Suite

This document provides a step-by-step learning resource and a practical QA test suite to set up and validate a robust local data platform environment. By following these steps and executing the test cases, you will gain hands-on experience with key technologies and develop a deeper understanding of the platform's functionality and operational aspects. This version introduces additional complexity to demonstrate the system's flexibility in handling different data types and its scalability.
This walkthrough leverages the architectural principles, setup guides, and best practices detailed in the "Building Enterprise-Ready Data Platforms: Core Handbook" and its associated Deep-Dive Addendums.

1. General Setup: Laying the Foundation

These steps are foundational and apply to all testing activities.

1.1. Prerequisites Installation

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

- Install Docker Desktop (or Docker Engine if on Linux).
- Install Git.
- Install Python 3.x with pip.
- Verify docker-compose is installed (usually included with Docker Desktop, or install separately if not).

1.2. Project Repository Setup

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

- Navigate to your desired development directory in your terminal.
- Execute: git clone <your-repo-url>/data-ingestion-platform
- Change into the cloned directory: cd data-ingestion-platform

2. Local Environment Setup: The Progressive Path

This section guides you through building the local data platform incrementally, mirroring the "Progressive Complexity Path" outlined in the Core Handbook. Each track builds upon the previous one.
Reference: For detailed docker-compose.yml configurations and specific instructions, refer to the **Progressive Path Setup Guide Deep-Dive Addendum**.

2.1. Starter Track Setup: Minimal Single-Machine Setup

Purpose: Understand foundational data ingestion and structured storage.
Components: FastAPI (Ingestor), PostgreSQL, MinIO (S3 compatible data lake).
Setup Steps:

1. **Configure docker-compose.yml**:
   - Open the docker-compose.yml file in the project root.
   - **Uncomment** the services for fastapi_ingestor, postgres, and minio.
   - **Comment out** all other services (Kafka, Spark, Airflow, etc.) to keep the setup minimal.
   - Ensure the data/postgres and data/minio directories exist in your project root for persistent volumes (Docker will create them if they don't).
2. **Bring Up Services**:
   - Execute the onboard.sh script (from **Progressive Path Setup Guide Deep-Dive Addendum**) or manually run: docker compose up --build -d
3. **Initial Verification**:
   - Access FastAPI health check: http://localhost:8000/health (Expected: HTTP 200 OK with a success message).
   - Access MinIO Console: http://localhost:9001 (Login with minioadmin/minioadmin). Expected: Console loads successfully.
   - Connect to PostgreSQL: Use a client (e.g., psql) to connect to localhost:5432 with user user, password password, database main_db. Expected: Connection successful, basic tables are present (if migrations run automatically).
   - Check Docker logs for all services: docker compose logs -f (Expected: No critical errors, services show healthy startup messages).

2.2. Intermediate Track Setup: Adding Streaming Capabilities

Purpose: Introduce real-time data streams and distributed transformations.
Components (in addition to Starter): Apache Kafka, Apache Spark.
Setup Steps:

1. **Configure docker-compose.yml**:
   - Open docker-compose.yml.
   - **Uncomment** (or keep uncommented) fastapi_ingestor, postgres, minio.
   - **Uncomment** the services for zookeeper, kafka, and spark (and optionally spark-history-server).
   - **Comment out** other Advanced Track services.
   - Review fastapi_ingestor's environment variables to ensure it publishes to **multiple Kafka topics** (e.g., KAFKA_BROKER: kafka:29092, KAFKA_TOPIC_FINANCIAL, KAFKA_TOPIC_INSURANCE).
   - Verify spark service is configured to connect to Kafka and MinIO.
   - Ensure data/spark-events exists for Spark history server logs.
   - **Note on Spark Scalability (Local Simulation):** In a local Docker Compose environment, "more Spark nodes" is simulated by allocating more resources (cores, memory) to the single spark service or by running multiple spark-worker services if configured. The conceptual docker-compose.yml and Spark job

submissions will imply this parallelism.

2. **Bring Up Services**:
   - Run the onboard.sh script again or docker compose up --build -d. The onboard.sh script should now be updated to initialize **both raw_financial_transactions and raw_insurance_claims Kafka topics.**
3. **Initial Verification**:
   - Verify Starter Track components are running and healthy.
   - Check Kafka topic creation: docker exec -it kafka kafka-topics --bootstrap-server localhost:9092 --list (Expected: **Both raw_financial_transactions and raw_insurance_claims topics are listed**).
   - Check Spark History Server: http://localhost:18080 (if enabled). Expected: Spark History Server UI is accessible.
4. **Generate External Data**:
   - Run the simulate_data.py script (from **Progressive Path Setup Guide Deep-Dive Addendum**). This script is designed to continuously send mock financial transactions and insurance claims to the FastAPI Ingestor, which in turn publishes them to their respective Kafka topics.
   - python3 simulate_data.py
5. **Trigger Spark Jobs**:
   - Manually submit two separate Spark streaming jobs from the spark container: one to consume from the financial Kafka topic and write to a financial Delta Lake path, and another for insurance data.
   - **For Financial Data:** docker exec -it spark spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0,io.delta:delta-core_2.12:2.4.0 --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension --conf spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 --conf spark.hadoop.fs.s3a.access.key=minioadmin --conf spark.hadoop.fs.s3a.secret.key=minioadmin --conf spark.hadoop.fs.s3a.path.style.access=true pyspark_jobs/streaming_consumer.py raw_financial_transactions kafka:29092 s3a://raw-data-bucket/financial_data_delta
   - **For Insurance Data:** docker exec -it spark spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0,io.delta:delta-core_2.12:2.4.0 --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension --conf spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 --conf spark.hadoop.fs.s3a.access.key=minioadmin --conf spark.hadoop.fs.s3a.secret.key=minioadmin --conf spark.hadoop.fs.s3a.path.style.access=true pyspark_jobs/streaming_consumer.py raw_insurance_claims kafka:29092 s3a://raw-data-bucket/insurance_data_delta

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

Purpose: Integrate orchestration, observability, lineage, and metadata management for a comprehensive environment.
Components (in addition to Intermediate): Apache Airflow, OpenTelemetry & Grafana Alloy, Grafana, Spline, OpenMetadata, MongoDB, cAdvisor.
Setup Steps:

1. **Configure docker-compose.yml**:
   - Open docker-compose.yml.
   - **Uncomment ALL** services including airflow-init, airflow-webserver, airflow-scheduler, airflow-worker, mongodb, openmetadata, grafana, grafana-alloy, cAdvisor, spline.
   - Ensure all environment variables for inter-service communication are correctly set.
   - Ensure necessary data/ subdirectories for persistent volumes exist.
   - Mount airflow_dags and observability directories as volumes for Airflow DAGs and Grafana configurations. These airflow_dags should now include separate DAGs for financial and insurance data pipelines to demonstrate complexity.
2. **Bring Up Services**:
   - Run the onboard.sh script or docker compose up --build -d. The airflow-init service will set up Airflow's database and load DAGs.
3. **Initial Verification**:
   - Verify Intermediate Track components are running and healthy.
   - Access Airflow UI: http://localhost:8080 (login admin/admin). Expected: Airflow UI accessible, **multiple DAGs (e.g., for financial and insurance data)** are listed (though not necessarily running).
   - Access Grafana UI: http://localhost:3000 (initially anonymous or configure admin user). Expected: Grafana UI accessible.
   - Access OpenMetadata UI: http://localhost:8585. Expected: OpenMetadata UI accessible.
   - Verify Spline UI: http://localhost:8081. Expected: Spline UI accessible.
   - Check Docker logs for all new services (e.g., grafana-alloy, cAdvisor). Expected: Services start without critical errors.

3. QA Test Suite for Data Platform Proficiency

This section provides a detailed, comprehensive test suite designed for a QA tester to validate the functionality and operational aspects of the data platform, demonstrating proficiency relevant to both Lead Data Engineer and AWS Engineer roles.
Reference: This test suite draws heavily from the **Testing & Observability Patterns Deep-Dive Addendum**, **DR & Runbooks Deep-Dive Addendum**, and **Cloud Migration + Terraform Snippets Deep-Dive Addendum**.

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

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

Test ID	Objective	Preconditions	Steps	Expected Results
ST-API-001	Verify successful ingestion of valid financial transaction via FastAPI.	All Starter Track services are up and healthy.	1. Send a valid financial transaction POST request to http://localhost:8000/ingest-financial-transaction/ using simulate_data.py or curl. python3 simulate_data.py (let it run for ~10 seconds, then stop).	HTTP 200 OK response from FastAPI. Console output shows "Financial transaction ingested successfully".
ST-API-002	Verify FastAPI rejects invalid financial transaction data.	All Starter Track services are up and healthy.	1. Send an invalid financial transaction POST request (e.g., timestamp: "invalid-date", amount: "not-a-number") to http://localhost:8000/ingest-financial-transaction/.	HTTP 422 Unprocessable Entity response. Response body contains "validation error" details.
ST-DB-001	Verify financial transaction data is persisted in PostgreSQL.	ST-API-001 passed.	1. Connect to PostgreSQL database (main_db) using psql or a GUI client. 2. Query the relevant table (e.g., SELECT * FROM financial_transactions; or SELECT COUNT(*) FROM financial_transacti	The ingested financial transaction record(s) are visible in the database. Data types match expectations.

			ons;).	
ST-S3-001	Verify raw data is stored in MinIO (simulated S3).	ST-API-001 passed.	1. Access the MinIO console at http://localhost:9001. 2. Navigate to the raw-data-bucket. 3. Look for new files corresponding to the ingested data (e.g., JSON or Parquet files).	Files are present in raw-data-bucket. Content of a sample file matches the ingested data.
ST-CONTAINER-001	Verify Docker containers' basic health and logs.	All Starter Track services are up.	1. Run docker ps. 2. Run docker compose logs -f.	All expected containers (fastapi_ingestor, postgres, minio) are running and healthy. Logs show normal operation without critical errors or frequent restarts.

3.2. Intermediate Track Test Cases (Kafka, Spark)

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

Test ID	Objective	Preconditions	Steps	Expected Results
IT-KAFKA-001	Verify FastAPI successfully publishes messages to **both financial and insurance Kafka topics**.	All Intermediate Track services are up and healthy. simulate_data.py is running.	1. In one terminal, run: docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw_financial_transactions --from-beginning. 2. In another terminal, run: docker exec -it	Messages (JSON payloads) from simulate_data.py appear in the Kafka consumer's output for *both* topics.

			kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw_insurance_claims --from-beginning.	
IT-SPARK-001	Verify Spark Structured Streaming consumes from **specific Kafka topics** and writes to respective raw Delta Lake paths.	All Intermediate Track services are up and healthy. IT-KAFKA-001 passed (data is being produced to both topics). Spark jobs for both data types are submitted (as in Section 2.2, Step 5).	1. Monitor Spark UI at http://localhost:18080 for both running Spark jobs. 2. Check MinIO console (raw-data-bucket) for both financial_data_delta and insurance_data_delta paths.	Both Spark jobs start, process input batches, and new Delta Lake files (parquet, _delta_log) appear in their respective MinIO paths. Spark UI shows progress for both jobs and no errors.
IT-SPARK-002	Verify Spark batch jobs transform **different raw Delta Lake data types** to curated Delta Lake.	All Intermediate Track services are up and healthy. raw-data-bucket contains data from IT-SPARK-001 for both data types.	1. Manually submit two conceptual batch_transformations.py Spark jobs (or a single job designed to handle both data types) from the spark container, configured to read from raw-data-bucket/financial_data_delta and raw-data-bucket/insurance_data_delta and write to curated-data-bucket/financial_data_curated and curated-data-buc	Both Spark batch jobs complete successfully. Curated financial and insurance data are written to their respective curated-data-bucket paths as Delta Lake files.

			ket/insurance_data_curated respectively. 2. Monitor Spark UI at http://localhost:18080. 3. Check MinIO console (curated-data-bucket).	
IT-DELTA-001	Verify Delta Lake ACID properties (conceptual: schema evolution) for a specific data type.	IT-SPARK-002 passed.	1. Modify simulate_data.py to add a new optional field to the financial transaction data (e.g., notes: "some text"). 2. Restart simulate_data.py. 3. Re-run the financial data streaming_consumer.py Spark job configured with mergeSchema option to write to raw-data-bucket/financial_data_delta. 4. Query the financial Delta Lake table with Spark.	The new field is added to the financial Delta Lake table schema without error, and existing data remains readable. Querying shows the new column as null for old records and populated for new ones.

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

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

Test ID	Objective	Preconditions	Steps	Expected Results
AT-AIRFLOW-001	Verify Airflow can successfully orchestrate	All Advanced Track services are up and healthy.	1. Access Airflow UI at http://localhost:80	Both DAGs run successfully to completion. Task

	separate data ingestion DAGs for financial and insurance data.		80. 2. Unpause and trigger financial_ingestion_dag.py and insurance_ingestion_dag.py (or similar DAGs that call FastAPI for respective data types). 3. Monitor the DAG run statuses in Airflow UI.	logs show successful FastAPI calls for their specific data types.
AT-AIRFLOW-002	Verify Airflow can successfully orchestrate **separate Spark transformation DAGs for financial and insurance data**.	All Advanced Track services are up and healthy. Data exists in respective raw Delta Lake zones from AT-AIRFLOW-001.	1. Access Airflow UI. 2. Unpause and trigger financial_transformation_dag.py and insurance_transformation_dag.py (or similar DAGs that submit Spark jobs for respective data types). 3. Monitor the DAG run statuses in Airflow UI and Spark UI (http://localhost:18080). 4. Check MinIO for new data in respective curated zones.	Both DAGs run successfully, their corresponding Spark jobs complete, and curated financial and insurance data appear in MinIO.
AT-OBS-001	Verify Grafana dashboards display real-time metrics for **multiple data streams**.	All Advanced Track services are up. Data is being generated by simulate_data.py.	1. Access Grafana UI at http://localhost:3000. 2. Navigate to pre-provisioned dashboards (e.g., "Health Dashboard"). 3. Observe graphs	Dashboards populate with real-time data for both financial and insurance data streams. Metrics accurately reflect system activity and resource

			for FastAPI RPS, latency, Kafka consumer lag for *both financial and insurance topics*, Spark CPU utilization, etc. 4. (Optional but recommended) Inspect Grafana Alloy's configuration (observability/alloy-config.river) to understand how it collects telemetry from different services.	usage across all components.
AT-OBS-002	Verify alerts are triggered for defined SLO violations (conceptual) for a **specific data stream**.	All Advanced Track services are up.	1. (Simulate) Artificially create a high Kafka consumer lag on *one* of the topics (e.g., raw_financial_transactions) by pausing the Spark consumer associated with it. 2. Monitor Grafana alert panel or configured alert notification channel.	An alert for "High Kafka Consumer Lag" specific to the affected topic (raw_financial_transactions) is triggered, containing summary, description, remediation, dashboard_link, and runbook_link as per configuration.
AT-LINEAGE-001	Verify Spline captures data lineage for **multiple distinct Spark jobs**.	AT-AIRFLOW-002 passed.	1. Access Spline UI at http://localhost:8081. 2. Search for both the financial and insurance Spark jobs executed by Airflow. 3. Drill	The Spline UI displays visual representations of the data flow for *both* financial and insurance pipelines, showing transformations from raw to

			down into the execution plan for each.	curated Delta Lake for each.
AT-METADATA-001	Verify OpenMetadata ingests and catalogs metadata for **multiple data types and components**.	All Advanced Track services are up. Airflow DAGs for OpenMetadata ingestion are configured/run for all relevant sources.	1. Access OpenMetadata UI at http://localhost:8585. 2. Browse or search for ingested assets (e.g., Kafka topics, Delta Lake tables for both financial and insurance data, PostgreSQL tables, FastAPI endpoints). 3. Check for associated schema, data quality, and lineage information for these distinct assets.	OpenMetadata UI displays comprehensive metadata for all relevant platform components and data types. Assets are discoverable and contain expected details and their respective lineage.
AT-DEBUG-001	Troubleshoot a simulated "Kafka Stuck Consumer" scenario for a **specific topic**.	Intermediate Track services are running and ingesting data.	1. Pause the Spark container handling *only* financial data (docker compose pause spark if using a single Spark service, or ideally a specific worker if multiple are configured, or disable the financial DAG in Airflow). This will cause Kafka consumer lag to build up on the raw_financial_transactions topic. 2.	Symptoms (lag, stalled processing) are confirmed for the financial data stream. Troubleshooting steps successfully pinpoint the cause. Resolving the issue (e.g., unpausing Spark) allows lag to reduce.

			Observe Grafana "Kafka Consumer Lag" dashboard, specifically for the financial topic. 3. Follow debug steps from "Common Gotchas & Debug Playbooks" (Section 5.7 in Testing & Observability Addendum): check Spark job status (Spark UI for the financial job), review logs, inspect Kafka offsets for raw_financial_transactions.	
AT-PERF-001	Run a basic load test and observe performance metrics for **multiple data types**.	All Advanced Track services are up and healthy.	1. Start the Locust load generator: locust -f locust_fastapi_ingestor.py --host http://localhost:8000. 2. Access Locust UI at http://localhost:8089 and start a test with moderate users/spawn rate. Locust will send *both* financial and insurance data. 3. Observe Grafana dashboards for FastAPI RPS, latency, and Kafka consumer lag for *both* financial and	Locust shows simulated traffic for both data types. Grafana reflects increased RPS, stable (or slightly increasing) latency, and manageable Kafka lag for *both* data streams, indicating the system can handle basic heterogeneous load.

			insurance topics during the test.	

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

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

Test ID	Objective	Preconditions	Steps (Conceptual/Discussion)	Expected Results (Conceptual/Discussion)
AE-CLOUD-001	Demonstrate understanding of local to AWS service mapping for **expanded components**.	N/A	1. For each local component (FastAPI, Kafka, Spark, MinIO, Airflow, Grafana, MongoDB), identify its primary AWS replacement as per the "Cloud Migration + Terraform Snippets Deep-Dive Addendum". 2. Specifically discuss how multiple Kafka topics map to MSK, how multiple Spark jobs map to Glue jobs or EMR steps, and how distinct Airflow DAGs map to MWAA DAGs. 3. Discuss the benefits (scalability, managed service, operational overhead	Correct mapping of local services to AWS managed services is articulated, emphasizing how increased complexity (more topics, jobs) translates to specific AWS service configurations. Benefits of managed services are clearly explained.

			reduction) of the AWS replacements.	
AE-IAC-001	Demonstrate understanding of Infrastructure as Code (IaC) for AWS, handling **multiple data flows**.	Access to the terraform_infra/ directory.	1. Review the conceptual Terraform modules for VPC, S3, MSK, Lambda/API Gateway, RDS, EMR/Glue in terraform_infra/modules/. 2. Explain how main.tf and variables.tf are used for environment-specific deployments, focusing on how Terraform would manage resources for both financial and insurance data pipelines (e.g., distinct S3 paths, Kafka topics, Glue job definitions). 3. Discuss the terraform init, plan, and apply workflow.	Able to describe the purpose of each Terraform module and how IaC enables automated, consistent cloud infrastructure provisioning, specifically for multiple distinct data pipelines.
AE-CICD-001	Demonstrate understanding of the CI/CD pipeline for AWS deployment, including **multiple application components**.	Access to the .github/workflows/release.yml file.	1. Walk through the Conceptual GitHub Actions Release Workflow. 2. Explain the build, publish (Docker images to ECR for FastAPI and Spark runner), deploy to staging, and promote to	Able to explain the flow of the CI/CD pipeline, how it automates deployments to AWS environments, and the purpose of different GitHub Actions steps, explicitly

			production steps. 3. Discuss how this pipeline would handle updates to *both* FastAPI and PySpark jobs, ensuring atomicity and consistency across the different data pipelines.	referencing how it manages updates for financial and insurance data pipeline components.
AE-SEC-001	Discuss data security best practices in an AWS context for **complex data streams**.	N/A	1. Explain Data Encryption (in transit with TLS on MSK, at rest with S3 SSE/KMS, RDS encryption). 2. Discuss Secure Credential Management using AWS Secrets Manager for database credentials and API keys, and granular IAM roles/policies for Lambda accessing MSK/S3 for different topics.	Able to articulate strategies for data encryption and secure credential management specific to AWS services, considering the separation of concerns for different data types/topics.
AE-MONITOR-001	Discuss monitoring and logging in an AWS context for **heterogeneous data pipelines**.	AT-OBS-001 passed (local observability).	1. Explain how local Grafana + OpenTelemetry concepts translate to AWS CloudWatch Logs, Metrics, Alarms, and AWS Distro for OpenTelemetry (ADOT). 2. Discuss how CloudWatch	Able to describe AWS-native monitoring solutions and how they provide fine-grained observability for cloud-based data systems, distinguishing between financial and insurance

			Container Insights replaces cAdvisor for container metrics, and how to create dashboards that show metrics for individual Kafka topics and Spark jobs (e.g., consumer lag per topic, job duration per data type).	data streams.
AE-DR-001	Demonstrate understanding of Disaster Recovery (DR) planning in AWS for **multiple critical data pipelines**.	N/A	1. Define RPO and RTO in the context of both financial and insurance data pipeline components. 2. Discuss the DR Runbook Examples (Kafka failover, Database restoration) and how AWS services like Multi-AZ, Cross-Region Replication, and Snapshots simplify DR, ensuring recovery strategies are in place for all critical data assets.	Able to explain RPO/RTO goals and outline AWS-specific DR strategies for critical data platform components, considering the separate recovery needs for financial vs. insurance data.
AE-OPTIMIZE-001	Discuss how to optimize data retrieval and API consumption in AWS for **diverse data needs**.	ST-API-001 passed (local API).	1. Discuss how AWS Lambda + API Gateway provide scalable API endpoints for different ingestion paths (e.g., /ingest-financial-t	Able to explain cloud-native approaches to optimizing data retrieval and providing scalable APIs for downstream

			ransaction, /ingest-insurance-claim). 2. Explain how data in S3 (Delta Lake) can be queried efficiently using services like AWS Athena or Redshift Spectrum, and how different data models (e.g., star schemas for reporting, denormalized for analytics) can be optimized for specific consumption patterns.	consumers, considering the varied requirements of financial and insurance data.

4. Conclusion

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