Deep-Dive Addendum: Testing & Observability Patterns

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

5.4. Comprehensive Testing Approaches

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

Unit Tests:

- **Purpose:** Verify the correctness of individual, isolated components or functions.
- **Application:** FastAPI endpoint logic, PySpark transformation functions (e.g., specific UDFs, data cleansing functions), and any custom Python utilities.
- **Tools:** pytest for Python code.

Sample Snippet (fastapi_app/tests/unit/test_api.py):

```
# fastapi_app/tests/unit/test_api.py
import pytest
from fastapi.testclient import TestClient
# Assuming your FastAPI app is structured like app.main.app
from fastapi_app.app.main import app
from datetime import datetime

client = TestClient(app)

def test_read_main():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Welcome to Financial/Insurance Data Ingestor API!"}

def test_ingest_financial_transaction_invalid_data():
    response = client.post("/ingest-financial-transaction/", json={
        "transaction_id": "FT-001",
        "timestamp": "invalid-date", # Invalid timestamp
        "account_id": "ACC-XYZ",
        "amount": "not-a-number", # Invalid amount
        "currency": "USD",
        "transaction_type": "debit"
    })
```

```
  assert response.status_code == 422 # Unprocessable Entity due to validation error
  assert "validation error" in response.text
```

Integration Tests:

- **Purpose:** Verify that different components of the pipeline work together as expected.
- **Application:** FastAPI to Kafka, Kafka to Spark (Streaming), Spark transformations.
- **Tools:** docker-compose.test.yml, pytest, Testcontainers (for robust service orchestration in tests), Kafka client libraries, MinIO SDK.

Conceptual docker-compose.test.yml for Integration Tests:
This file defines a stripped-down set of services specifically for integration testing, focusing on inter-service communication.

```
# docker-compose.test.yml (for integration testing)
version: '3.8'
services:
 zookeeper:
  image: confluentinc/cp-zookeeper:7.4.0
  environment:
   ZOOKEEPER_CLIENT_PORT: 2181
  healthcheck:
   test: ["CMD", "sh", "-c", "nc -z localhost 2181"]
   interval: 10s
   timeout: 5s
   retries: 5

 kafka:
  image: confluentinc/cp-kafka:7.4.0
  depends_on:
   zookeeper:
    condition: service_healthy
  ports:
   - "9092:9092"
  environment:
   KAFKA_BROKER_ID: 1
   KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
   KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
   KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
   KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
   KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
  healthcheck:
   test: ["CMD", "sh", "-c", "kafka-topics --bootstrap-server localhost:9092 --list"]
```

```yaml
      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
```

```yaml
      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"
    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/s
park_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
```

```python
from pyspark.sql.functions import current_timestamp

# Define Pact mock server details
PACT_MOCK_HOST = 'localhost'
PACT_MOCK_PORT = 1234
PACT_DIR = './pacts' # Directory where pact files will be written

# Define the consumer and provider for this contract
consumer = Consumer('FinancialTransactionSparkConsumer')
provider = Provider('FastAPIIngestor')

@pytest.fixture(scope='module')
def pact_spark_session():
    """Fixture for a local SparkSession to be used in contract tests."""
    spark = (SparkSession.builder
             .appName("PactSparkConsumer")
             .master("local[*]")
             .getOrCreate())
    yield spark
    spark.stop()

@pytest.fixture(scope='module')
def pact():
    """Starts and stops the Pact mock service."""
    pact_instance = consumer.has_pact_with(
        provider,
        host_name=PACT_MOCK_HOST,
        port=PACT_MOCK_PORT,
        pact_dir=PACT_DIR
    )
    print(f"\nStarting Pact mock service on {PACT_MOCK_HOST}:{PACT_MOCK_PORT}")
    pact_instance.start_service()
    yield pact_instance
    print("Stopping Pact mock service")
    pact_instance.stop_service()

def test_spark_can_process_financial_transaction_from_kafka(pact, pact_spark_session):
    """
    Verifies that the Spark consumer can correctly process a financial transaction
    message from Kafka, based on the contract with the FastAPI Ingestor.
    """
    # Define the expected message structure from the producer (FastAPI)
    expected_message_body = {
```

```
    "transaction_id": "TRANS-12345",
    "timestamp": "2023-10-26T14:30:00.000Z",
    "account_id": "ACC-FIN-001",
    "amount": 500.75,
    "currency": "USD",
    "transaction_type": "credit",
    "merchant_id": "MER-ABC",
    "category": "utilities"
  }

  # Define the interaction for the Kafka message
  (pact
   .given('a financial transaction is published to Kafka')
   .upon_receiving('a Kafka message with financial transaction data')
   .with_message(
     'application/json', # Mime type of the message
     json.dumps(expected_message_body) # The expected message content
   ))

  with pact:
    # Simulate receiving the message as if from Kafka
    # In a real Spark job, this would be the actual Kafka consumer logic
    # For a contract test, we feed the expected message directly to the Spark logic
    # Convert the expected message body to a Spark DataFrame
    schema = StructType() \
      .add("transaction_id", StringType()) \
      .add("timestamp", StringType()) \
      .add("account_id", StringType()) \
      .add("amount", FloatType()) \
      .add("currency", StringType()) \
      .add("transaction_type", StringType()) \
      .add("merchant_id", StringType(), True) \
      .add("category", StringType(), True)

    # Create a DataFrame from the single expected message
    df_from_kafka = pact_spark_session.createDataFrame([expected_message_body],
schema=schema)

    # Apply a dummy transformation that resembles your actual Spark job logic
    # This ensures your Spark code can parse and work with the contract-defined schema
    processed_df = df_from_kafka.withColumn("processed_at", current_timestamp())

    # Collect and assert the processed data
```

```
collected_data = processed_df.collect()
assert len(collected_data) == 1
assert collected_data[0]['transaction_id'] == expected_message_body['transaction_id']
assert collected_data[0]['amount'] == expected_message_body['amount']
assert 'processed_at' in collected_data[0]
```

Performance and Load Testing:

- **Purpose:** Assess the system's performance under expected and peak load conditions, identify bottlenecks, and ensure it meets non-functional requirements (e.g., latency, throughput).
- **Application:** Use tools to simulate high volumes of data being sent to the FastAPI endpoint and monitor Kafka, Spark, and database performance using Grafana dashboards.
- **Tools:** Locust (for API load testing), JMeter, Spark UI, Grafana.

5.6. Observability: From Configuration to Practice

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

5.6.1. Defining SLIs and SLOs

- **SLI (Service Level Indicator):** A quantitative measure of some aspect of the level of service that is provided.
- **SLO (Service Level Objective):** A target value or range for an SLI that defines the desired level of service.

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 to logical changes)	<2.0 (maintaining storage efficiency)
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:**
  - **Check Spark Job Status:** Is the Spark Structured Streaming job consuming from Kafka actually running? (http://localhost:8080 for Spark UI). Look at "Running

Applications" and "Completed Applications." Is your job listed? Check its current status, stages, and tasks.
    - **Review Spark Logs:** Examine executor logs and driver logs for specific errors (deserialization, processing exceptions, OutOfMemoryError), continuous restarts, or backpressure warnings.
    - **Inspect Kafka Offsets:** Use kafka-consumer-groups.sh to get current offsets and confirm lag directly.
      ```
      # Conceptual command to inspect Kafka consumer group offsets
      docker exec -it kafka kafka-consumer-groups.sh --bootstrap-server kafka:29092
      --describe --group <your_consumer_group_name>
      ```
    - **Verify Kafka Broker Health:** Check Kafka and Zookeeper container logs for any errors (e.g., disk full, network issues).
    - **Grafana Consumer Lag Panel:** Monitor a pre-built Grafana dashboard (see "Health-Check Dashboard" in Section 8.1, not in this addendum but detailed in the full guide) showing consumer lag metrics, often providing historical context.
- **Action:** If Spark job is failing, debug code logic. If Spark is too slow, scale up Spark executors/cores or optimize transformations. If Kafka is unhealthy, investigate broker issues. Refer to runbooks/kafka_consumer_lag.md.

Delta Lake Writes Failing under Schema Drift:

- **Symptoms:** Spark writes to Delta Lake fail with schema mismatch errors, AnalysisException: Cannot resolve '...' given input columns, Schema is not compatible.
- **Triage Flow:**
    - **Identify Schema Change:** Compare incoming DataFrame schema with the existing Delta table schema. The error message usually highlights the problematic column or type.
    - **Review Error Message:** Understand if a column was added, removed, renamed, or its type changed.
    - **Decide on Schema Evolution Strategy:**
        - mergeSchema (Recommended for evolution): Allows adding new columns or reordering existing ones without breaking the write.
          ```
          # PySpark: Enable schema merging for writes
          df.write.format("delta") \
            .mode("append") \
            .option("mergeSchema", "true") \
            .save("/path/to/delta_table")
          ```
        - overwriteSchema (Use with EXTREME CAUTION): Overwrites the entire table schema. This is destructive and can lead to data loss or make historical data unreadable if not managed carefully.
          ```
          # PySpark: Overwrite schema (use with EXTREME CAUTION)
          df.write.format("delta") \
          ```

```
.mode("overwrite") \
.option("overwriteSchema", "true") \
.save("/path/to/delta_table")
```

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

Docker Networking Pitfalls on M1/Mac vs. Windows:

- **Symptoms:** Containers cannot communicate with each other or with services on the host machine (e.g., fastapi_ingestor cannot reach kafka), Connection Refused, Name or service not known.
- **Triage Flow:**
  - **Check docker-compose.yml:**
    - **Service Names:** Ensure containers reference each other by their service name within the Docker network (e.g., kafka:29092, not localhost:9092).
    - **Port Mappings:** Verify correct ports mappings (e.g., 9092:9092) for external host access. Remember that internal and external ports can differ.
    - depends_on: Use condition: service_healthy to ensure dependencies are fully ready before a dependent service tries to connect.
  - host.docker.internal (Mac/Windows Specific): If a container needs to connect to a service running **directly on the host machine** (e.g., a locally run Python script acting as a mock API), use host.docker.internal as the hostname.
    ```
    # Example: A custom script inside container needs to connect to host-bound service
    my_container:
      environment:
        HOST_API_URL: http://host.docker.internal:8080
    ```
  - **Firewall Rules:** On Windows, explicitly check and configure your firewall rules to allow inbound connections to the exposed Docker ports. Docker Desktop generally manages this for macOS, but custom firewall settings can interfere.
  - **Network Inspection:** Use docker inspect <container_id> or docker network inspect <network_name> to view container IP addresses and network configurations, which can help diagnose routing issues.
- **Action:** Correct hostnames/IPs in environment variables, verify port mappings, adjust host firewall rules.

7.4. Sample Benchmarking Harness & Observed Data

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

Benchmarking Harness Components:

- **Load Generator (Locust):** Simulates concurrent users sending financial/insurance data to the FastAPI ingestion API.
- **FastAPI Ingestor:** Receives data and publishes it to Kafka.
- **Kafka Cluster:** Buffers the incoming data stream.
- **Spark Structured Streaming Job:** Consumes from Kafka, performs basic transformations (e.g., parsing, schema enforcement), and writes to the Raw Delta Lake zone in MinIO.
- **Metrics Collector (Grafana Alloy):** Collects metrics from FastAPI, Kafka, Spark, and cAdvisor.
- **Monitoring (Grafana):** Visualizes end-to-end latency, throughput, and resource utilization.

Conceptual Benchmarking Steps:

- **Setup Environment:** Bring up the full Advanced Track Docker Compose environment.
- **Run Load Generator:** Start Locust to simulate X users sending Y requests per second to FastAPI.
- **Monitor Metrics:** Observe Grafana dashboards for key metrics:
  - FastAPI request rate (RPS) and latency.
  - Kafka producer throughput (messages/sec, MB/sec).
  - Kafka consumer throughput and lag (messages/sec, messages in backlog).
  - Spark streaming batch processing time and records processed.
  - CPU, memory, network utilization for all Docker containers (via cAdvisor).
- **Analyze Data:** Record and analyze average/p99 latency, throughput, and resource bottlenecks.
- **Scale Up/Down:** Repeat tests by varying Kafka partitions, Spark executor counts, cores, and memory to identify optimal configurations for different load levels.

Conceptual Locust Load Test Script (locust_fastapi_ingestor.py):

```
# locust_fastapi_ingestor.py
"""
Locust load test script for the FastAPI Data Ingestor.
This script defines two tasks to simulate traffic:
1.  ingest_financial_transaction: Sends mock financial transaction data.
2.  ingest_insurance_claim: Sends mock insurance claim data.

The user can configure the host, number of users, and spawn rate via the Locust UI
(usually http://localhost:8089 after running `locust -f locust_fastapi_ingestor.py`).
"""
from locust import HttpUser, task, between
import json
from datetime import datetime, timedelta
import random
```

```
class FinancialDataUser(HttpUser):
    """
    User class that simulates sending financial and insurance data to the FastAPI ingestor.
    """
    # Wait time between requests for each simulated user.
    # This helps simulate more realistic user behavior rather than hammering the API
constantly.
    wait_time = between(0.1, 0.5) # Simulate delay between requests (0.1 to 0.5 seconds)

    # The host URL for the FastAPI application. This should match the exposed port in
docker-compose.
    # In a local Docker Compose setup, FastAPI is often exposed on localhost:8000.
    host = "http://localhost:8000" # Target FastAPI endpoint

    @task(1) # This task has a weight of 1, meaning it will be executed proportionally to other
tasks.
    def ingest_financial_transaction(self):
        """
        Simulates sending a financial transaction POST request to the FastAPI ingestor.
        Generates realistic-looking mock data for a financial transaction.
        """
        transaction_data = {
            "transaction_id":
f"FT-{datetime.now().strftime('%Y%m%d%H%M%S%f')}-{random.randint(1000, 9999)}",
            "timestamp": datetime.now().isoformat(),
            "account_id": f"ACC-{random.randint(100000, 999999)}",
            "amount": round(random.uniform(1.0, 10000.0), 2), # Random amount between 1.00
and 10000.00
            "currency": random.choice(["USD", "EUR", "GBP", "JPY"]), # Random currency
            "transaction_type": random.choice(["debit", "credit", "transfer", "payment"]), # Random
type
            "merchant_id": f"MER-{random.randint(100, 999)}" if random.random() > 0.3 else
None, # Optional merchant ID
            "category": random.choice(["groceries", "utilities", "salary", "entertainment",
"transport", "housing", "healthcare", "education"])
        }
        # Send the POST request. The 'name' argument groups requests in Locust's statistics.
        self.client.post("/ingest-financial-transaction/", json=transaction_data,
name="/ingest-financial-transaction")

    @task(1) # This task also has a weight of 1.
    def ingest_insurance_claim(self):
```

```
        """
        Simulates sending an insurance claim POST request to the FastAPI ingestor.
        Generates realistic-looking mock data for an insurance claim.
        """
        claim_data = {
            "claim_id":
f"IC-{datetime.now().strftime('%Y%m%d%H%M%S%f')}-{random.randint(1000, 9999)}",
            "timestamp": datetime.now().isoformat(),
            "policy_number": f"POL-{random.randint(1000000, 9999999)}",
            "claim_amount": round(random.uniform(500.0, 50000.0), 2), # Random amount
            "claim_type": random.choice(["auto", "health", "home", "life", "property"]), # Random
claim type
            "claim_status": random.choice(["submitted", "under_review", "approved", "rejected",
"paid"]), # Random status
            "customer_id": f"CUST-{random.randint(10000, 99999)}",
            "incident_date": (datetime.now() - timedelta(days=random.randint(0, 365))).isoformat()
# Incident date within last year
        }
        # Send the POST request.
        self.client.post("/ingest-insurance-claim/", json=claim_data,
name="/ingest-insurance-claim")
```

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

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

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 configurations 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 F: Testing Framework Detail Expansion

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

Unit Tests:

- **Purpose:** Verify the correctness of individual, isolated components or functions.
- **Application:** FastAPI endpoint logic, PySpark transformation functions (e.g., specific UDFs, data cleansing functions), and any custom Python utilities.
- **Tools:** pytest for Python code.

Sample Snippet (fastapi_app/tests/unit/test_api.py):

```
# fastapi_app/tests/unit/test_api.py
import pytest
from fastapi.testclient import TestClient
# Assuming your FastAPI app is structured like app.main.app
from fastapi_app.app.main import app
from datetime import datetime

client = TestClient(app)

def test_read_main():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Welcome to Financial/Insurance Data Ingestor API!"}

def test_ingest_financial_transaction_invalid_data():
```

```
    response = client.post("/ingest-financial-transaction/", json={
        "transaction_id": "FT-001",
        "timestamp": "invalid-date", # Invalid timestamp
        "account_id": "ACC-XYZ",
        "amount": "not-a-number", # Invalid amount
        "currency": "USD",
        "transaction_type": "debit"
    })
    assert response.status_code == 422 # Unprocessable Entity due to validation error
    assert "validation error" in response.text
```

Integration Tests:

- **Purpose:** Verify that different components of the pipeline work together as expected.
- **Application:** FastAPI to Kafka, Kafka to Spark (Streaming), Spark transformations.
- **Tools:** docker-compose.test.yml, pytest, Testcontainers (for robust service orchestration in tests), Kafka client libraries, MinIO SDK.

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

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

```
# docker-compose.test.yml (for integration testing)
version: '3.8'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.4.0
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    healthcheck:
      test: ["CMD", "sh", "-c", "nc -z localhost 2181"]
      interval: 10s
      timeout: 5s
      retries: 5

  kafka:
    image: confluentinc/cp-kafka:7.4.0
    depends_on:
      zookeeper:
        condition: service_healthy
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
```

```
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    healthcheck:
      test: ["CMD", "sh", "-c", "kafka-topics --bootstrap-server localhost:9092 --list"]
      interval: 10s
      timeout: 5s
      retries: 5

  minio:
    image: minio/minio:latest
    ports:
      - "9000:9000"
    environment:
      MINIO_ROOT_USER: test_user
      MINIO_ROOT_PASSWORD: test_password
    command: server /data --console-address ":9000"
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]
      interval: 30s
      timeout: 20s
      retries: 3

  fastapi_ingestor:
    build: ./fastapi_app
    environment:
      KAFKA_BROKER: kafka:29092
      KAFKA_TOPIC: raw_data_test
    depends_on:
      kafka:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health || exit 1"]
      interval: 5s
      timeout: 3s
      retries: 5

  # Spark service for integration testing (can be a standalone driver in test, or a small cluster)
  spark-test-runner:
    image: bitnami/spark:3.5.0
```

```yaml
    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):

```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"
    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/s
park_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
```

```python
import pytest
from pact import Consumer, Provider
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StringType, FloatType, TimestampType
import json
from datetime import datetime
from pyspark.sql.functions import current_timestamp

# Define Pact mock server details
PACT_MOCK_HOST = 'localhost'
PACT_MOCK_PORT = 1234
PACT_DIR = './pacts' # Directory where pact files will be written

# Define the consumer and provider for this contract
consumer = Consumer('FinancialTransactionSparkConsumer')
provider = Provider('FastAPIIngestor')

@pytest.fixture(scope='module')
def pact_spark_session():
    """Fixture for a local SparkSession to be used in contract tests."""
    spark = (SparkSession.builder
             .appName("PactSparkConsumer")
             .master("local[*]")
             .getOrCreate())
    yield spark
    spark.stop()

@pytest.fixture(scope='module')
def pact():
    """Starts and stops the Pact mock service."""
    pact_instance = consumer.has_pact_with(
        provider,
        host_name=PACT_MOCK_HOST,
        port=PACT_MOCK_PORT,
        pact_dir=PACT_DIR
    )
    print(f"\nStarting Pact mock service on {PACT_MOCK_HOST}:{PACT_MOCK_PORT}")
    pact_instance.start_service()
    yield pact_instance
    print("Stopping Pact mock service")
    pact_instance.stop_service()

def test_spark_can_process_financial_transaction_from_kafka(pact, pact_spark_session):
```

```
    """
    Verifies that the Spark consumer can correctly process a financial transaction
    message from Kafka, based on the contract with the FastAPI Ingestor.
    """
    # Define the expected message structure from the producer (FastAPI)
    expected_message_body = {
        "transaction_id": "TRANS-12345",
        "timestamp": "2023-10-26T14:30:00.000Z",
        "account_id": "ACC-FIN-001",
        "amount": 500.75,
        "currency": "USD",
        "transaction_type": "credit",
        "merchant_id": "MER-ABC",
        "category": "utilities"
    }

    # Define the interaction for the Kafka message
    (pact
     .given('a financial transaction is published to Kafka')
     .upon_receiving('a Kafka message with financial transaction data')
     .with_message(
         'application/json', # Mime type of the message
         json.dumps(expected_message_body) # The expected message content
     ))

    with pact:
        # Simulate receiving the message as if from Kafka
        # In a real Spark job, this would be the actual Kafka consumer logic
        # For a contract test, we feed the expected message directly to the Spark logic
        # Convert the expected message body to a Spark DataFrame
        schema = StructType() \
            .add("transaction_id", StringType()) \
            .add("timestamp", StringType()) \
            .add("account_id", StringType()) \
            .add("amount", FloatType()) \
            .add("currency", StringType()) \
            .add("transaction_type", StringType()) \
            .add("merchant_id", StringType(), True) \
            .add("category", StringType(), True)

        # Create a DataFrame from the single expected message
        df_from_kafka = pact_spark_session.createDataFrame([expected_message_body],
schema=schema)
```

```
# Apply a dummy transformation that resembles your actual Spark job logic
# This ensures your Spark code can parse and work with the contract-defined schema
processed_df = df_from_kafka.withColumn("processed_at", current_timestamp())

# Collect and assert the processed data
collected_data = processed_df.collect()
assert len(collected_data) == 1
assert collected_data[0]['transaction_id'] == expected_message_body['transaction_id']
assert collected_data[0]['amount'] == expected_message_body['amount']
assert 'processed_at' in collected_data[0]
```

Performance and Load Testing:

- **Purpose:** Assess the system's performance under expected and peak load conditions, identify bottlenecks, and ensure it meets non-functional requirements (e.g., latency, throughput).
- **Application:** Use tools to simulate high volumes of data being sent to the FastAPI endpoint and monitor Kafka, Spark, and database performance using Grafana dashboards.
- **Tools:** Locust (for API load testing), JMeter, Spark UI, Grafana.

Appendix H: Quantitative Benchmarking Harness Details

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

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

Benchmarking Harness Components:

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

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

    - **Configuration:** Configured to vary the number of concurrent users and requests per second (RPS) to test different load levels.
- **FastAPI Ingestor:**
    - **Role:** The entry point for all incoming data. It receives data from the load generator, performs initial validation (via Pydantic models), and publishes the messages to the designated Kafka topics.
    - **Monitoring Focus:** Key metrics include request per second (RPS), end-to-end API latency (average and P99), and error rates.
- **Kafka Cluster:**
    - **Role:** Acts as a distributed, fault-tolerant message buffer. It receives and stores the high-volume data streams published by the FastAPI ingestor.
    - **Monitoring Focus:** Key metrics include producer throughput (messages/sec, MB/sec), consumer throughput (messages/sec), and critically, Kafka consumer lag (number of messages remaining in the backlog for the Spark consumer).
- **Spark Structured Streaming Job:**
    - **Role:** Consumes data from the raw Kafka topics, performs essential transformations (e.g., parsing, schema enforcement, data cleansing, and basic aggregations), and writes the processed data to the Raw Delta Lake zone in MinIO.
    - **Monitoring Focus:** Metrics include batch processing time, records processed per batch, micro-batch latency, and resource utilization (CPU, memory) of Spark executors.
- **Metrics Collector (Grafana Alloy):**
    - **Role:** Collects telemetry data (metrics, logs, traces) from all instrumented components within the Docker Compose environment. It acts as a central collection agent for observability data.
    - **Integration:** Configured to receive OpenTelemetry Protocol (OTLP) data from FastAPI and other services, and to scrape Prometheus-compatible metrics (e.g., from cAdvisor, Kafka JMX exporters).
- **Monitoring (Grafana):**
    - **Role:** Provides interactive data visualization and monitoring dashboards. It connects to Grafana Alloy (or directly to Prometheus/Loki configured by Alloy) to visualize real-time and historical performance metrics.
    - **Dashboards:** Pre-built dashboards show end-to-end latency, throughput for each pipeline stage, resource utilization (CPU, memory, network I/O) for all Docker containers (via cAdvisor), and Kafka consumer lag trends.

Conceptual Benchmarking Steps:

A systematic approach to benchmarking ensures reliable and reproducible results:

- **Setup Environment:** Bring up the full Advanced Track Docker Compose environment (docker compose -f docker-compose.yml up --build -d). Ensure all services are healthy and stable before starting tests.
- **Establish Baseline:** Run the system under a typical, low-load condition. Record

baseline performance metrics (latency, throughput, resource usage) to understand normal operating characteristics.

- **Run Load Generator:** Start the Locust load generator, configuring it to simulate a specific number of concurrent users and a target request rate to the FastAPI endpoint.
  - Example command: locust -f locust_fastapi_ingestor.py --host http://localhost:8000 (then access Locust UI at http://localhost:8089).
- **Monitor Metrics in Real-time:** Continuously observe the Grafana dashboards during the load test. Pay close attention to:
  - **FastAPI:** Request rate (RPS), average and P99 latency for API calls, and any error spikes.
  - **Kafka:** Producer throughput (ensuring data is flowing into Kafka as expected), consumer throughput (ensuring Spark is keeping up), and especially Kafka consumer lag (any increasing lag indicates a bottleneck downstream).
  - **Spark:** Batch processing times (for streaming jobs), number of records processed per second, CPU and memory utilization of Spark master and worker nodes (available via Spark UI or Grafana).
  - **Overall System:** Container resource utilization (CPU, memory, network I/O) across all services using cAdvisor metrics in Grafana.
- **Analyze Data:** After the load test, analyze the recorded metrics.
  - Identify the bottleneck: Is it the API, Kafka, Spark, or the underlying storage (MinIO)?
  - Evaluate latency and throughput against defined SLOs.
  - Look for correlation between increased load, resource saturation, and performance degradation.
- **Scale Up/Down and Tune:** Repeat tests by systematically varying parameters:
  - **Kafka:** Increase/decrease the number of partitions for topics.
  - **Spark:** Adjust Spark executor counts, cores per executor, and memory allocated per executor in docker-compose.yml. Experiment with Spark configurations like spark.sql.shuffle.partitions.
  - **FastAPI:** If FastAPI becomes a bottleneck, consider increasing the number of FastAPI replicas or optimizing its code.
  - **Databases (PostgreSQL/MongoDB):** For intensive workloads, monitor database specific metrics (e.g., connection pool size, query latency, disk I/O) and consider tuning database configurations or scaling resources.

This iterative process of testing, monitoring, analyzing, and tuning is essential to identify the optimal configuration for different load levels and to ensure the platform scales effectively.
Conceptual Locust Load Test Script (locust_fastapi_ingestor.py):
This script simulates two types of data ingestion: financial transactions and insurance claims.
locust_fastapi_ingestor.py
"""

Locust load test script for the FastAPI Data Ingestor.
This script defines two tasks to simulate traffic:
1. ingest_financial_transaction: Sends mock financial transaction data.

```
2.  ingest_insurance_claim: Sends mock insurance claim data.

The user can configure the host, number of users, and spawn rate via the Locust UI
(usually http://localhost:8089 after running `locust -f locust_fastapi_ingestor.py`).
"""
from locust import HttpUser, task, between
import json
from datetime import datetime, timedelta
import random

class FinancialDataUser(HttpUser):
    """
    User class that simulates sending financial and insurance data to the FastAPI ingestor.
    """
    # Wait time between requests for each simulated user.
    # This helps simulate more realistic user behavior rather than hammering the API
constantly.
    wait_time = between(0.1, 0.5) # Simulate delay between requests (0.1 to 0.5 seconds)

    # The host URL for the FastAPI application. This should match the exposed port in
docker-compose.
    # In a local Docker Compose setup, FastAPI is often exposed on localhost:8000.
    host = "http://localhost:8000" # Target FastAPI endpoint

    @task(1) # This task has a weight of 1, meaning it will be executed proportionally to other
tasks.
    def ingest_financial_transaction(self):
        """
        Simulates sending a financial transaction POST request to the FastAPI ingestor.
        Generates realistic-looking mock data for a financial transaction.
        """
        transaction_data = {
            "transaction_id":
f"FT-{datetime.now().strftime('%Y%m%d%H%M%S%f')}-{random.randint(1000, 9999)}",
            "timestamp": datetime.now().isoformat(),
            "account_id": f"ACC-{random.randint(100000, 999999)}",
            "amount": round(random.uniform(1.0, 10000.0), 2), # Random amount between 1.00
and 10000.00
            "currency": random.choice(["USD", "EUR", "GBP", "JPY"]), # Random currency
            "transaction_type": random.choice(["debit", "credit", "transfer", "payment"]), # Random
type
            "merchant_id": f"MER-{random.randint(100, 999)}" if random.random() > 0.3 else
None, # Optional merchant ID
```

```
            "category": random.choice(["groceries", "utilities", "salary", "entertainment",
"transport", "housing", "healthcare", "education"])
        }
        # Send the POST request. The 'name' argument groups requests in Locust's statistics.
        self.client.post("/ingest-financial-transaction/", json=transaction_data,
name="/ingest-financial-transaction")

    @task(1) # This task also has a weight of 1.
    def ingest_insurance_claim(self):
        """
        Simulates sending an insurance claim POST request to the FastAPI ingestor.
        Generates realistic-looking mock data for an insurance claim.
        """
        claim_data = {
            "claim_id":
f"IC-{datetime.now().strftime('%Y%m%d%H%M%S%f')}-{random.randint(1000, 9999)}",
            "timestamp": datetime.now().isoformat(),
            "policy_number": f"POL-{random.randint(1000000, 9999999)}",
            "claim_amount": round(random.uniform(500.0, 50000.0), 2), # Random amount
            "claim_type": random.choice(["auto", "health", "home", "life", "property"]), # Random
claim type
            "claim_status": random.choice(["submitted", "under_review", "approved", "rejected",
"paid"]), # Random status
            "customer_id": f"CUST-{random.randint(10000, 99999)}",
            "incident_date": (datetime.now() - timedelta(days=random.randint(0, 365))).isoformat()
# Incident date within last year
        }
        # Send the POST request.
        self.client.post("/ingest-insurance-claim/", json=claim_data,
name="/ingest-insurance-claim")
```

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

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

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 configurations 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.