🚀 Day 14: Week 2 Integration Project - Building Your First End-to-End Data Platform

📚 What You'll Build Today (Systems-First Approach)

Primary Focus: Integrating all Week 2 technologies into a cohesive data platform **Secondary Focus:** Production deployment patterns and monitoring strategies
Dataset Integration: Multi-source data processing combining all previous datasets

🎯 Learning Philosophy for Day 14

"Architecture is about understanding the whole before optimizing the parts"

We'll start with system design principles, understand component interactions, design data flow architecture, and build a production-ready integrated platform.

🌟 The Integration Challenge: From Tools to Platform

🤔 The Problem: Tool Fragmentation in Data Engineering

Scenario: You've learned individual tools but need to build a real business solution...

Without Integration (Tool Chaos):

```
❌ Monday 6 AM: Manually start PostgreSQL
❌ Monday 6:15 AM: Check if Docker containers are running
❌ Monday 6:30 AM: Manually trigger Airflow DAG
❌ Monday 6:45 AM: Monitor Spark job logs separately
❌ Monday 7:00 AM: Check data quality in different tools
❌ Monday 7:15 AM: Manually verify database loads
❌ Monday 7:30 AM: Pray everything worked together
```

Problems:

- No unified monitoring across tools
- Manual coordination between systems
- Failure in one component breaks everything
- Impossible to track data lineage
- No automated recovery mechanisms
- Difficult to scale or reproduce

💡 The Integrated Platform Solution

With Platform Integration:

```
✅ Automated infrastructure deployment
✅ Orchestrated cross-tool workflows
✅ Unified monitoring and alerting
✅ Automatic failure recovery
✅ End-to-end data lineage tracking
✅ Scalable, reproducible architecture
✅ Single command deployment and operation
```

🏗️ Platform Architecture Design (Visual Approach)

🎨 The Integrated Data Platform Mental Model

```
┌─────────────────────────────────────────────────────────────────────┐
│                     INTEGRATED DATA PLATFORM                        │
│                                                                      │
│  ┌─────────────┐    ┌─────────────┐    ┌─────────────┐              │
│  │   Data      │    │ Processing  │    │  Storage    │              │
│  │ Ingestion   │───▶│   Layer     │───▶│   Layer     │              │
│  │             │    │             │    │             │              │
│  │ • APIs      │    │ • Spark     │    │ • PostgreSQL│              │
│  │ • Files     │    │ • Pandas    │    │ • Files     │              │
│  │ • Databases │    │ • Custom    │    │ • S3        │              │
│  └─────────────┘    └─────────────┘    └─────────────┘              │
│         │                   │                   │                   │
│  ┌─────────────────────────────────────────────────────────────┐    │
│  │              ORCHESTRATION LAYER (Airflow)                 │   │
│  │  • Workflow Management  • Dependency Resolution           │   │
│  │  • Scheduling          • Error Handling                   │   │
│  └─────────────────────────────────────────────────────────────┘    │
│         │                   │                   │                   │
│  ┌─────────────────────────────────────────────────────────────┐    │
│  │               INFRASTRUCTURE LAYER (Docker)                │   │
│  │  • Containerization     • Service Discovery                │   │
│  │  • Resource Management • Network Configuration            │   │
│  └─────────────────────────────────────────────────────────────┘    │
│         │                   │                   │                   │
│  ┌─────────────────────────────────────────────────────────────┐    │
│  │                MONITORING & OBSERVABILITY                  │   │
│  │  • Health Checks        • Performance Metrics             │   │
│  │  • Alerting           • Data Quality Monitoring           │   │
│  └─────────────────────────────────────────────────────────────┘    │
└─────────────────────────────────────────────────────────────────────┘
```

🧠 Platform Components and Interactions

1. Infrastructure Layer (Foundation)

- Docker Compose for service orchestration
- Network configuration and service discovery
- Volume management for data persistence
- Resource allocation and scaling

2. Data Storage Layer

- PostgreSQL for structured analytics
- File system for raw and processed data
- Staging areas for data transformations
- Backup and recovery mechanisms

3. Processing Layer

- Apache Spark for large-scale processing
- Pandas for data manipulation
- Custom Python modules for business logic
- Data quality validation frameworks

4. Orchestration Layer

- Airflow DAGs for workflow management
- Cross-service dependency management
- Scheduling and trigger mechanisms
- Error handling and recovery

5. Monitoring Layer

- Health checks across all services
- Performance metrics collection
- Data quality monitoring
- Alerting and notification systems

🎯 Platform Setup and Architecture (Infrastructure-First)

📱 Complete Platform Docker Compose

Project Structure:

```
week2-integration-project/
├── docker-compose.yml
├── airflow/
│   ├── dags/
│   │   └── integrated_data_platform_dag.py
│   ├── plugins/
│   ├── logs/
│   └── requirements.txt
├── spark/
│   ├── apps/
│   │   └── customer_analytics_spark.py
│   └── conf/
├── data/
│   ├── raw/
│   │   ├── ecommerce_transactions.csv
│   │   ├── superstore_sales.csv
│   │   ├── amazon_products.csv
│   │   └── retail_analytics.csv
│   ├── staging/
│   └── processed/
├── sql/
│   ├── schema/
│   │   └── create_tables.sql
│   └── queries/
├── scripts/
│   ├── data_quality/
│   │   └── quality_checks.py
│   ├── ingestion/
│   │   └── multi_source_ingestion.py
│   └── monitoring/
│       └── health_checks.py
├── config/
│   ├── spark-defaults.conf
│   └── postgres.conf
└── monitoring/
    ├── grafana/
    └── prometheus/
```

Complete docker-compose.yml:

yaml

```yaml
version: '3.8'

# Shared environment variables
x-airflow-common:
  &airflow-common
  image: apache/airflow:2.7.1-python3.10
  environment: &airflow-common-env
    AIRFLOW__CORE__EXECUTOR: LocalExecutor
    AIRFLOW__DATABASE__SQL_ALCHEMY_CONN: postgresql+psycopg2://airflow:airflow@postgre
    AIRFLOW__CORE__FERNET_KEY: ''
    AIRFLOW__CORE__DAGS_ARE_PAUSED_AT_CREATION: 'true'
    AIRFLOW__CORE__LOAD_EXAMPLES: 'false'
    AIRFLOW__API__AUTH_BACKENDS: 'airflow.api.auth.backend.basic_auth'
    AIRFLOW__WEBSERVER__EXPOSE_CONFIG: 'true'
    _PIP_ADDITIONAL_REQUIREMENTS: >-
      apache-airflow-providers-postgres
      apache-airflow-providers-docker
      pyspark==3.4.1
      pandas
      requests
      great-expectations
  volumes:
    - ./airflow/dags:/opt/airflow/dags
    - ./airflow/logs:/opt/airflow/logs
    - ./airflow/plugins:/opt/airflow/plugins
    - ./data:/opt/airflow/data
    - ./scripts:/opt/airflow/scripts
    - ./sql:/opt/airflow/sql
  user: "${AIRFLOW_UID:-50000}:0"
  depends_on: &airflow-common-depends-on
    postgres-airflow:
      condition: service_healthy

services:
  # PostgreSQL for Airflow metadata
  postgres-airflow:
    image: postgres:13
    environment:
      POSTGRES_USER: airflow
      POSTGRES_PASSWORD: airflow
      POSTGRES_DB: airflow
    volumes:
      - postgres_airflow_db_volume:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "airflow"]
      interval: 5s
      retries: 5
    ports:
      - "5433:5432"

  # PostgreSQL for business data
  postgres-data:
    image: postgres:13
    environment:
      POSTGRES_USER: datauser
      POSTGRES_PASSWORD: datapass
      POSTGRES_DB: analytics
    volumes:
      - postgres_data_db_volume:/var/lib/postgresql/data
      - ./sql/schema:/docker-entrypoint-initdb.d
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "datauser"]
      interval: 5s
      retries: 5
    ports:
      - "5434:5432"

  # Redis for task queuing
  redis:
    image: redis:7.0-alpine
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
      timeout: 3s
      retries: 5
    ports:
      - "6379:6379"

  # Airflow Webserver
  airflow-webserver:
    <<: *airflow-common
    command: webserver
    ports:
      - "8080:8080"
    healthcheck:
      test: ["CMD", "curl", "--fail", "http://localhost:8080/health"]
      interval: 10s
      timeout: 10s
      retries: 5
    depends_on:
      <<: *airflow-common-depends-on

  # Airflow Scheduler
  airflow-scheduler:
    <<: *airflow-common
    command: scheduler
    healthcheck:
      test: ["CMD-SHELL", 'airflow jobs check --job-type SchedulerJob --hostname "$${H
      interval: 10s
      timeout: 10s
      retries: 5
    depends_on:
      <<: *airflow-common-depends-on

  # Spark Master
  spark-master:
    image: bitnami/spark:3.4.1
    environment:
      - SPARK_MODE=master
      - SPARK_RPC_AUTHENTICATION_ENABLED=no
      - SPARK_RPC_ENCRYPTION_ENABLED=no
      - SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED=no
      - SPARK_SSL_ENABLED=no
    ports:
      - "8081:8080"
      - "7077:7077"
    volumes:
      - ./spark/apps:/opt/spark-apps
      - ./data:/opt/spark-data
      - ./spark/conf:/opt/spark/conf
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080"]
      interval: 30s
      timeout: 10s
      retries: 3

  # Spark Worker
  spark-worker:
    image: bitnami/spark:3.4.1
    environment:
      - SPARK_MODE=worker
      - SPARK_MASTER_URL=spark://spark-master:7077
      - SPARK_WORKER_MEMORY=2G
      - SPARK_WORKER_CORES=2
      - SPARK_RPC_AUTHENTICATION_ENABLED=no
      - SPARK_RPC_ENCRYPTION_ENABLED=no
      - SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED=no
      - SPARK_SSL_ENABLED=no
    volumes:
      - ./spark/apps:/opt/spark-apps
      - ./data:/opt/spark-data
    depends_on:
      - spark-master

  # Jupyter Notebook for development
  jupyter:
    image: jupyter/pyspark-notebook:spark-3.4.1
    ports:
      - "8888:8888"
    environment:
      - JUPYTER_ENABLE_LAB=yes
    volumes:
      - ./notebooks:/home/jovyan/work
      - ./data:/home/jovyan/data
    command: start-notebook.sh --NotebookApp.token='' --NotebookApp.password=''

  # Airflow Init Service
  airflow-init:
    <<: *airflow-common
    entrypoint: /bin/bash
    command:
      - -c
      - |
        function ver() {
          printf "%04d%04d%04d%04d" $${1//./ }
        }
        airflow_version=$$(AIRFLOW__LOGGING__LOGGING_LEVEL=INFO && airflow version)
        airflow_version_comparable=$$(ver $${airflow_version})
        min_airflow_version=2.2.0
        min_airflow_version_comparable=$$(ver $${min_airflow_version})
        if (( airflow_version_comparable < min_airflow_version_comparable )); then
          echo "ERROR!!!: Too old Airflow version $${airflow_version}!"
          exit 1
        fi
        if [[ -z "${AIRFLOW_UID}" ]]; then
```

```
          echo "WARNING!!!: AIRFLOW_UID not set!"
          export AIRFLOW_UID=50000
        fi
        one_meg=1048576
        mem_available=$$(($$(getconf _PHYS_PAGES) * $$(getconf PAGE_SIZE) / one_meg))
        cpus_available=$$(grep -cE 'cpu[0-9]+' /proc/stat)
        if (( mem_available < 4000 )) ; then
          echo "WARNING!!!: Not enough memory available for Docker."
        fi
        if (( cpus_available < 2 )); then
          echo "WARNING!!!: Not enough CPUS available for Docker."
        fi
        mkdir -p /sources/logs /sources/dags /sources/plugins
        chown -R "${AIRFLOW_UID}:0" /sources/{logs,dags,plugins}
        exec /entrypoint airflow version
    environment:
      <<: *airflow-common-env
      _AIRFLOW_DB_UPGRADE: 'true'
      _AIRFLOW_WWW_USER_CREATE: 'true'
      _AIRFLOW_WWW_USER_USERNAME: ${_AIRFLOW_WWW_USER_USERNAME:-airflow}
      _AIRFLOW_WWW_USER_PASSWORD: ${_AIRFLOW_WWW_USER_PASSWORD:-airflow}

volumes:
  postgres_airflow_db_volume:
  postgres_data_db_volume:

networks:
  default:
    driver: bridge
```

🎮 Platform Initialization and Startup

Step 1: Environment Setup

```bash
# Create project directory structure
mkdir -p week2-integration-project/{airflow/{dags,plugins,logs},spark/{apps,conf},data

# Set environment variables
echo "AIRFLOW_UID=$(id -u)" > .env
echo "COMPOSE_PROJECT_NAME=week2-platform" >> .env

# Download datasets
cd data/raw
# Download all datasets from Kaggle (instructions below)
```

Step 2: Database Schema Setup

```
sql
```

```sql
-- sql/schema/create_tables.sql
-- Create analytics database schema

-- Customer dimension table
CREATE TABLE IF NOT EXISTS dim_customers (
    customer_id VARCHAR(50) PRIMARY KEY,
    customer_name VARCHAR(255),
    segment VARCHAR(100),
    country VARCHAR(100),
    city VARCHAR(100),
    registration_date DATE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Product dimension table
CREATE TABLE IF NOT EXISTS dim_products (
    product_id VARCHAR(50) PRIMARY KEY,
    product_name VARCHAR(255),
    category VARCHAR(100),
    sub_category VARCHAR(100),
    brand VARCHAR(100),
    price DECIMAL(10,2),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Time dimension table
CREATE TABLE IF NOT EXISTS dim_time (
    date_id DATE PRIMARY KEY,
    year INTEGER,
    quarter INTEGER,
    month INTEGER,
    week INTEGER,
    day_of_week INTEGER,
    is_weekend BOOLEAN,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Sales fact table
CREATE TABLE IF NOT EXISTS fact_sales (
    sale_id SERIAL PRIMARY KEY,
    customer_id VARCHAR(50) REFERENCES dim_customers(customer_id),
    product_id VARCHAR(50) REFERENCES dim_products(product_id),
    sale_date DATE REFERENCES dim_time(date_id),
    quantity INTEGER,
    unit_price DECIMAL(10,2),
    discount DECIMAL(5,4),
    total_amount DECIMAL(12,2),
    profit DECIMAL(12,2),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Customer analytics table
CREATE TABLE IF NOT EXISTS customer_analytics (
    customer_id VARCHAR(50) PRIMARY KEY,
    total_orders INTEGER,
    total_spent DECIMAL(12,2),
    average_order_value DECIMAL(10,2),
    last_order_date DATE,
    days_since_last_order INTEGER,
    rfm_recency INTEGER,
    rfm_frequency INTEGER,
    rfm_monetary INTEGER,
    rfm_score VARCHAR(10),
    customer_segment VARCHAR(50),
    lifetime_value DECIMAL(12,2),
    churn_probability DECIMAL(5,4),
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Data quality monitoring table
CREATE TABLE IF NOT EXISTS data_quality_metrics (
    id SERIAL PRIMARY KEY,
    table_name VARCHAR(100),
    metric_name VARCHAR(100),
    metric_value DECIMAL(15,4),
    threshold_value DECIMAL(15,4),
    status VARCHAR(20),
    execution_date DATE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Pipeline execution log
CREATE TABLE IF NOT EXISTS pipeline_execution_log (
    id SERIAL PRIMARY KEY,
    pipeline_name VARCHAR(100),
    execution_date DATE,
    start_time TIMESTAMP,
    end_time TIMESTAMP,
    status VARCHAR(20),
    records_processed INTEGER,
    error_message TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Create indexes for performance
CREATE INDEX IF NOT EXISTS idx_fact_sales_customer_date ON fact_sales(customer_id, sal
CREATE INDEX IF NOT EXISTS idx_fact_sales_product_date ON fact_sales(product_id, sale_
CREATE INDEX IF NOT EXISTS idx_customer_analytics_segment ON customer_analytics(custom
CREATE INDEX IF NOT EXISTS idx_data_quality_table_date ON data_quality_metrics(table_n
```

Step 3: Launch the Platform

bash

```bash
# Initialize Airflow
docker-compose up airflow-init

# Start all services
docker-compose up -d

# Verify all services are healthy
docker-compose ps

# Access platform components:
# Airflow Web UI: http://localhost:8080 (airflow/airflow)
# Spark Master UI: http://localhost:8081
# Jupyter Lab: http://localhost:8888
# PostgreSQL Data: localhost:5434 (datauser/datapass/analytics)
# PostgreSQL Airflow: localhost:5433 (airflow/airflow/airflow)
```

📊 Data Acquisition and Setup

🎯 Multi-Source Dataset Download

Required Datasets for Integration:

1. **E-Commerce Transactions**
   - Source: `kaggle.com/datasets/smayanj/e-commerce-transactions-dataset`
   - File: `ecommerce_transactions.csv`
   - Purpose: Transaction fact data

2. **Superstore Sales**
    - Source: `kaggle.com/datasets/vivek468/superstore-dataset-final`
    - File: `superstore_sales.csv`
    - Purpose: Historical sales analysis
3. **Amazon Products**
    - Source: `kaggle.com/datasets/jithinanievarghese/amazon-product-dataset`
    - File: `amazon_products.csv`
    - Purpose: Product dimension data
4. **Retail Analytics**
    - Source: `kaggle.com/datasets/manjeetsingh/retaildataset`
    - File: `retail_analytics.csv`
    - Purpose: Customer behavior analysis

Dataset Download Script:

python

```python
# scripts/ingestion/download_datasets.py
"""
Dataset Download and Preparation Script
Automates the download and initial preparation of all required datasets
"""

import os
import pandas as pd
import requests
from pathlib import Path

def setup_data_directories():
    """Create required data directory structure"""
    directories = [
        'data/raw',
        'data/staging',
        'data/processed',
        'data/archive'
    ]

    for directory in directories:
        Path(directory).mkdir(parents=True, exist_ok=True)
        print(f"✅ Created directory: {directory}")

def validate_datasets():
    """Validate downloaded datasets"""
    required_files = [
        'data/raw/ecommerce_transactions.csv',
        'data/raw/superstore_sales.csv',
        'data/raw/amazon_products.csv',
        'data/raw/retail_analytics.csv'
    ]

    validation_results = {}

    for file_path in required_files:
        if os.path.exists(file_path):
            try:
                df = pd.read_csv(file_path)
                validation_results[file_path] = {
                    'exists': True,
                    'rows': len(df),
                    'columns': len(df.columns),
                    'size_mb': round(os.path.getsize(file_path) / (1024*1024), 2)
                }
                print(f"✅ {file_path}: {len(df)} rows, {len(df.columns)} columns")
            except Exception as e:
                validation_results[file_path] = {
                    'exists': True,
                    'error': str(e)
                }
                print(f"❌ {file_path}: Error reading file - {e}")
        else:
            validation_results[file_path] = {'exists': False}
            print(f"❌ {file_path}: File not found")

    return validation_results

def create_sample_data():
    """Create sample datasets if originals not available"""
    print("🔄 Creating sample datasets for development...")

    # Sample e-commerce transactions
    ecommerce_data = {
        'transaction_id': [f'TXN{i:06d}' for i in range(1, 1001)],
        'customer_id': [f'CUST{(i % 200) + 1:04d}' for i in range(1, 1001)],
        'product_id': [f'PROD{(i % 50) + 1:04d}' for i in range(1, 1001)],
        'transaction_date': pd.date_range('2023-01-01', periods=1000, freq='H'),
        'quantity': np.random.randint(1, 5, 1000),
        'unit_price': np.random.uniform(10, 200, 1000).round(2),
        'total_amount': lambda x: x['quantity'] * x['unit_price']
    }

    # Create and save sample datasets
    # (Additional sample data generation code here)

    print("✅ Sample datasets created successfully!")

if __name__ == "__main__":
    setup_data_directories()
    validation_results = validate_datasets()

    # If datasets missing, create samples
    missing_files = [k for k, v in validation_results.items() if not v.get('exists')]
    if missing_files:
        print(f"⚠️ Missing {len(missing_files)} required datasets")
        print("Please download from Kaggle or run with --create-samples")
    else:
        print("🎉 All datasets validated successfully!")
```

🛠️ Integrated DAG Implementation

🧩 Complete Platform DAG

Primary DAG File: `airflow/dags/integrated_data_platform_dag.py`

python

```python
"""
Integrated Data Platform DAG
============================
This DAG orchestrates the complete data platform workflow:
1. Multi-source data ingestion
2. Cross-dataset data quality validation
3. Distributed processing with Spark
4. Data warehouse loading
5. Analytics computation
6. Cross-platform monitoring

Schedule: Daily at 6 AM
Dependencies: Docker services (Spark, PostgreSQL, Redis)
"""

from datetime import datetime, timedelta
import pandas as pd
import logging
from pathlib import Path

from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.operators.bash_operator import BashOperator
from airflow.operators.docker_operator import DockerOperator
from airflow.providers.postgres.operators.postgres import PostgresOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook
from airflow.utils.task_group import TaskGroup
from airflow.models import Variable

# DAG Configuration
default_args = {
    'owner': 'data-platform-team',
    'depends_on_past': False,
    'start_date': datetime(2024, 1, 1),
    'email_on_failure': True,
    'email_on_retry': False,
    'retries': 2,
    'retry_delay': timedelta(minutes=5),
    'catchup': False
}

dag = DAG(
    'integrated_data_platform',
    default_args=default_args,
    description='Complete integrated data platform workflow',
    schedule_interval='0 6 * * *',
    max_active_runs=1,
    tags=['platform', 'integration', 'multi-source', 'enterprise']
)

# Configuration Management
def get_platform_config():
    """Centralized platform configuration"""
    return {
        'data_sources': {
            'ecommerce': '/opt/airflow/data/raw/ecommerce_transactions.csv',
            'superstore': '/opt/airflow/data/raw/superstore_sales.csv',
            'products': '/opt/airflow/data/raw/amazon_products.csv',
            'retail': '/opt/airflow/data/raw/retail_analytics.csv'
        },
        'staging_paths': {
            'customers': '/opt/airflow/data/staging/customers.parquet',
            'products': '/opt/airflow/data/staging/products.parquet',
            'transactions': '/opt/airflow/data/staging/transactions.parquet'
        },
        'quality_thresholds': {
            'completeness': 0.95,
            'uniqueness': 0.99,
            'validity': 0.90
        },
        'spark_config': {
            'master': 'spark://spark-master:7077',
            'app_name': 'IntegratedDataPlatform',
            'executor_memory': '2g',
            'driver_memory': '1g'
        }
    }

# Platform Health Checks
def check_platform_health(**context):
    """Comprehensive platform health verification"""
    print("🔍 Starting platform health checks...")

    health_status = {
        'services': {},
        'data_sources': {},
        'storage': {},
        'overall': True
    }

    # Check Docker services
    services_to_check = [
        'week2-platform_postgres-data_1',
        'week2-platform_spark-master_1',
        'week2-platform_redis_1'
    ]

    import subprocess
    for service in services_to_check:
        try:
            result = subprocess.run(
                ['docker', 'inspect', service],
                capture_output=True,
                text=True
            )
            if result.returncode == 0:
                health_status['services'][service] = 'healthy'
                print(f"✅ Service {service}: healthy")
            else:
                health_status['services'][service] = 'unhealthy'
                health_status['overall'] = False
                print(f"❌ Service {service}: unhealthy")
        except Exception as e:
            health_status['services'][service] = f'error: {e}'
            health_status['overall'] = False
            print(f"❌ Service {service}: error - {e}")

    # Check data source availability
    config = get_platform_config()
    for source_name, file_path in config['data_sources'].items():
        try:
            if Path(file_path).exists():
                df = pd.read_csv(file_path, nrows=5)  # Quick validation
                health_status['data_sources'][source_name] = {
                    'status': 'available',
                    'rows': len(df),
                    'columns': len(df.columns)
                }
                print(f"✅ Data source {source_name}: available ({len(df.columns)} col
            else:
                health_status['data_sources'][source_name] = 'missing'
```

```
                health_status['overall'] = False
                print(f"❌ Data source {source_name}: file not found")
        except Exception as e:
            health_status['data_sources'][source_name] = f'error: {e}'
            health_status['overall'] = False
            print(f"❌ Data source {source_name}: error - {e}")

    # Check database connectivity
    try:
        postgres_hook = PostgresHook(postgres_conn_id='postgres_data')
        postgres_hook.get_first("SELECT 1 as health_check")
        health_status['storage']['postgresql'] = 'connected'
        print("✅ PostgreSQL: connected")
    except Exception as e:
        health_status['storage']['postgresql'] = f'error: {e}'
        health_status['overall'] = False
        print(f"❌ PostgreSQL: connection failed - {e}")

    if health_status['overall']:
        print("🎉 Platform health check: ALL SYSTEMS GO!")
    else:
        print("⚠️  Platform health check: ISSUES DETECTED")
        raise ValueError("Platform health check failed")

    # Store health status for monitoring
    context['task_instance'].xcom_push(key='health_status', value=health_status)
    return health_status

# Multi-Source Data Ingestion
def ingest_multi_source_data(**context):
    """Ingest and standardize data from multiple sources"""
    print("🔄 Starting multi-source data ingestion...")

    config = get_platform_config()
    ingestion_results = {}

    # Ingest E-commerce transactions
    try:
        print("📊 Processing e-commerce transactions...")
        ecommerce_df = pd.read_csv(config['data_sources']['ecommerce'])

        # Standardize column names
        ecommerce_standardized = ecommerce_df.rename(columns={
            'Customer ID': 'customer_id',
            'Product ID': 'product_id',
            'Transaction Date': 'transaction_date',
            'Quantity': 'quantity',
            'Price': 'unit_price',
            'Total': 'total_amount'
        })

        # Data type conversion
        ecommerce_standardized['transaction_date'] = pd.to_datetime(
            ecommerce_standardized['transaction_date']
        )
        ecommerce_standardized['total_amount'] = pd.to_numeric(
            ecommerce_standardized['total_amount'], errors='coerce'
        )

        # Save to staging
        staging_path = '/opt/airflow/data/staging/ecommerce_standardized.parquet'
        ecommerce_standardized.to_parquet(staging_path, index=False)

        ingestion_results['ecommerce'] = {
            'status': 'success',
            'records': len(ecommerce_standardized),
            'staging_path': staging_path
        }
        print(f"✅ E-commerce: {len(ecommerce_standardized)} records ingested")

    except Exception as e:
        ingestion_results['ecommerce'] = {'status': 'failed', 'error': str(e)}
        print(f"❌ E-commerce ingestion failed: {e}")

    # Ingest Superstore sales
    try:
        print("📊 Processing superstore sales...")
        superstore_df = pd.read_csv(config['data_sources']['superstore'])

        # Standardize and enrich
        superstore_standardized = superstore_df.rename(columns={
            'Customer ID': 'customer_id',
            'Product ID': 'product_id',
            'Order Date': 'order_date',
            'Sales': 'sales_amount',
            'Profit': 'profit_amount',
            'Discount': 'discount_rate'
        })
```

```python
        # Calculate derived metrics
        superstore_standardized['profit_margin'] = (
            superstore_standardized['profit_amount'] /
            superstore_standardized['sales_amount']
        ).fillna(0)

        superstore_standardized['order_date'] = pd.to_datetime(
            superstore_standardized['order_date']
        )

        staging_path = '/opt/airflow/data/staging/superstore_standardized.parquet'
        superstore_standardized.to_parquet(staging_path, index=False)

        ingestion_results['superstore'] = {
            'status': 'success',
            'records': len(superstore_standardized),
            'staging_path': staging_path
        }
        print(f"✅ Superstore: {len(superstore_standardized)} records ingested")

    except Exception as e:
        ingestion_results['superstore'] = {'status': 'failed', 'error': str(e)}
        print(f"❌ Superstore ingestion failed: {e}")

    # Ingest Product catalog
    try:
        print("📊 Processing product catalog...")
        products_df = pd.read_csv(config['data_sources']['products'])

        # Product dimension standardization
        products_standardized = products_df.rename(columns={
            'product_id': 'product_id',
            'product_name': 'product_name',
            'category': 'category',
            'price': 'list_price'
        })

        # Data enrichment
        products_standardized['price_tier'] = pd.cut(
            products_standardized['list_price'],
            bins=[0, 25, 100, 500, float('inf')],
            labels=['Budget', 'Mid-range', 'Premium', 'Luxury']
        )
        staging_path = '/opt/airflow/data/staging/products_standardized.parquet'
        products_standardized.to_parquet(staging_path, index=False)

        ingestion_results['products'] = {
            'status': 'success',
            'records': len(products_standardized),
            'staging_path': staging_path
        }
        print(f"✅ Products: {len(products_standardized)} records ingested")

    except Exception as e:
        ingestion_results['products'] = {'status': 'failed', 'error': str(e)}
        print(f"❌ Products ingestion failed: {e}")

    # Summary and validation
    successful_ingestions = sum(1 for result in ingestion_results.values()
                               if result.get('status') == 'success')
    total_records = sum(result.get('records', 0) for result in ingestion_results.value
                       if result.get('status') == 'success')

    summary = {
        'successful_sources': successful_ingestions,
        'total_sources': len(config['data_sources']),
        'total_records_ingested': total_records,
        'ingestion_results': ingestion_results
    }

    print(f"📊 Ingestion Summary: {successful_ingestions}/{len(config['data_sources'])
    print(f"📊 Total records ingested: {total_records:,}")

    context['task_instance'].xcom_push(key='ingestion_results', value=summary)
    return summary

# Cross-Dataset Data Quality Validation
def validate_integrated_data_quality(**context):
    """Comprehensive data quality validation across all datasets"""
    print("🔍 Starting integrated data quality validation...")

    config = get_platform_config()
    quality_results = {}

    def calculate_quality_metrics(df, dataset_name):
        """Calculate standard quality metrics for any dataset"""
        metrics = {}

        # Completeness (percentage of non-null values)
        completeness = (df.notna().sum() / len(df)).mean()
        metrics['completeness'] = round(completeness, 4)

        # Uniqueness (for identifier columns)
        id_columns = [col for col in df.columns if 'id' in col.lower()]
        if id_columns:
            uniqueness_scores = []
            for col in id_columns:
                unique_ratio = df[col].nunique() / len(df)
                uniqueness_scores.append(unique_ratio)
            metrics['uniqueness'] = round(sum(uniqueness_scores) / len(uniqueness_scor
        else:
            metrics['uniqueness'] = 1.0

        # Validity (data type consistency)
        numeric_columns = df.select_dtypes(include=['number']).columns
        validity_scores = []
        for col in numeric_columns:
            valid_ratio = (df[col].notna() & (df[col] >= 0)).sum() / len(df)
            validity_scores.append(valid_ratio)

        if validity_scores:
            metrics['validity'] = round(sum(validity_scores) / len(validity_scores), 4
        else:
            metrics['validity'] = 1.0

        # Overall quality score
        metrics['overall_quality'] = round(
            (metrics['completeness'] + metrics['uniqueness'] + metrics['validity']) /
        )

        return metrics

    # Validate each staged dataset
    staging_files = [
        ('/opt/airflow/data/staging/ecommerce_standardized.parquet', 'ecommerce'),
        ('/opt/airflow/data/staging/superstore_standardized.parquet', 'superstore'),
        ('/opt/airflow/data/staging/products_standardized.parquet', 'products')
    ]

    for file_path, dataset_name in staging_files:
        try:
            if Path(file_path).exists():
                df = pd.read_parquet(file_path)
                metrics = calculate_quality_metrics(df, dataset_name)

                # Check against thresholds
                thresholds = config['quality_thresholds']
                quality_status = all([
                    metrics['completeness'] >= thresholds['completeness'],
                    metrics['uniqueness'] >= thresholds['uniqueness'],
                    metrics['validity'] >= thresholds['validity']
                ])

                quality_results[dataset_name] = {
                    'metrics': metrics,
                    'passed_quality_check': quality_status,
                    'record_count': len(df),
                    'column_count': len(df.columns)
                }

                status_emoji = "✅" if quality_status else "⚠️"
                print(f"{status_emoji} {dataset_name}: Quality score {metrics['overall_
                print(f"   Completeness: {metrics['completeness']:.2%}")
                print(f"   Uniqueness: {metrics['uniqueness']:.2%}")
                print(f"   Validity: {metrics['validity']:.2%}")
            else:
                quality_results[dataset_name] = {
                    'error': 'File not found',
                    'passed_quality_check': False
                }
                print(f"❌ {dataset_name}: Staging file not found")

        except Exception as e:
            quality_results[dataset_name] = {
                'error': str(e),
                'passed_quality_check': False
            }
            print(f"❌ {dataset_name}: Quality validation failed - {e}")

    # Cross-dataset validation
    print("\n🔗 Performing cross-dataset validation...")
    cross_dataset_results = {}

    try:
        # Load datasets for cross-validation
        ecommerce_df = pd.read_parquet('/opt/airflow/data/staging/ecommerce_standardiz
        products_df = pd.read_parquet('/opt/airflow/data/staging/products_standardized

        # Referential integrity checks
        ecommerce_products = set(ecommerce_df['product_id'].unique())
        catalog_products = set(products_df['product_id'].unique())

        # Product referential integrity
        missing_products = ecommerce_products - catalog_products
        referential_integrity_score = 1 - (len(missing_products) / len(ecommerce_produ

        cross_dataset_results['referential_integrity'] = {
            'score': round(referential_integrity_score, 4),
            'missing_product_count': len(missing_products),
            'total_products_in_transactions': len(ecommerce_products)
        }

        print(f"🔗 Referential integrity: {referential_integrity_score:.2%}")
        print(f"🔗 Missing products in catalog: {len(missing_products)}")

    except Exception as e:
        cross_dataset_results['referential_integrity'] = {
            'error': str(e),
            'score': 0
        }
        print(f"❌ Cross-dataset validation failed: {e}")

    # Overall quality assessment
    passed_datasets = sum(1 for result in quality_results.values()
                         if result.get('passed_quality_check', False))
    total_datasets = len(quality_results)

    overall_assessment = {
        'datasets_passed': passed_datasets,
        'total_datasets': total_datasets,
        'overall_pass_rate': round(passed_datasets / total_datasets, 4) if total_datas
        'quality_results': quality_results,
        'cross_dataset_results': cross_dataset_results
    }

    print(f"\n📊 Quality Assessment Summary:")
    print(f"📊 Datasets passed: {passed_datasets}/{total_datasets}")
    print(f"📊 Overall pass rate: {overall_assessment['overall_pass_rate']:.2%}")
    # Fail the task if quality thresholds not met
    if overall_assessment['overall_pass_rate'] < 0.8:
        raise ValueError(f"Data quality below acceptable threshold: {overall_assessmen

    context['task_instance'].xcom_push(key='quality_results', value=overall_assessment
    return overall_assessment

# Spark-based Large Scale Processing
def process_data_with_spark(**context):
    """Process data using Spark for scalable analytics"""
    print("⚡ Starting Spark-based data processing...")

    # This would normally be a Spark application
    # For this example, we'll simulate Spark processing with pandas
    # In production, this would use PySpark or submit to Spark cluster

    processing_results = {}

    try:
        # Load staging data
        print("📊 Loading data into Spark-like processing...")
        ecommerce_df = pd.read_parquet('/opt/airflow/data/staging/ecommerce_standardiz
        superstore_df = pd.read_parquet('/opt/airflow/data/staging/superstore_standard
        products_df = pd.read_parquet('/opt/airflow/data/staging/products_standardized

        # Customer 360 analytics (simulating Spark transformations)
        print("🔄 Computing Customer 360 analytics...")

        # Combine transaction data
        all_transactions = pd.concat([
            ecommerce_df[['customer_id', 'transaction_date', 'total_amount']].rename(c
            superstore_df[['customer_id', 'order_date', 'sales_amount']].rename(column
        ])

        # Customer analytics
        customer_analytics = all_transactions.groupby('customer_id').agg({
            'amount': ['sum', 'mean', 'count'],
            'date': ['min', 'max']
        }).round(2)

        # Flatten column names
        customer_analytics.columns = ['total_spent', 'avg_order_value', 'total_orders'
        customer_analytics = customer_analytics.reset_index()
```

```
# Calculate derived metrics
customer_analytics['days_since_last_order'] = (
    datetime.now() - pd.to_datetime(customer_analytics['last_order_date'])
).dt.days

# Customer segmentation (RFM-like)
customer_analytics['recency_score'] = pd.qcut(
    customer_analytics['days_since_last_order'],
    q=5, labels=[5, 4, 3, 2, 1]
)
customer_analytics['frequency_score'] = pd.qcut(
    customer_analytics['total_orders'].rank(method='first'),
    q=5, labels=[1, 2, 3, 4, 5]
)
customer_analytics['monetary_score'] = pd.qcut(
    customer_analytics['total_spent'],
    q=5, labels=[1, 2, 3, 4, 5]
)

# Segment labels
def segment_customers(row):
    score = int(str(row['recency_score']) + str(row['frequency_score']) + str(
    if score >= 544:
        return 'Champions'
    elif score >= 334:
        return 'Loyal Customers'
    elif score >= 244:
        return 'Potential Loyalists'
    elif score >= 144:
        return 'At Risk'
    else:
        return 'Lost Customers'

customer_analytics['customer_segment'] = customer_analytics.apply(segment_cust

# Save processed data
processed_path = '/opt/airflow/data/processed/customer_360_analytics.parquet'
customer_analytics.to_parquet(processed_path, index=False)

processing_results['customer_360'] = {
    'status': 'success',
    'records_processed': len(customer_analytics),
    'output_path': processed_path
```

```
        }

        print(f"✅ Customer 360: {len(customer_analytics)} customer profiles created")

        # Product performance analytics
        print("🔄 Computing product performance metrics...")

        # Merge transaction data with product details
        ecommerce_with_products = ecommerce_df.merge(
            products_df[['product_id', 'category', 'price_tier']],
            on='product_id',
            how='left'
        )

        # Product analytics
        product_performance = ecommerce_with_products.groupby(['product_id', 'category
            'quantity': 'sum',
            'total_amount': ['sum', 'mean'],
            'customer_id': 'nunique'
        }).round(2)

        product_performance.columns = ['total_quantity_sold', 'total_revenue', 'avg_or
        product_performance = product_performance.reset_index()

        # Calculate performance metrics
        product_performance['revenue_per_customer'] = (
            product_performance['total_revenue'] / product_performance['unique_custome
        ).round(2)

        processed_path = '/opt/airflow/data/processed/product_performance.parquet'
        product_performance.to_parquet(processed_path, index=False)

        processing_results['product_performance'] = {
            'status': 'success',
            'records_processed': len(product_performance),
            'output_path': processed_path
        }

        print(f"✅ Product Performance: {len(product_performance)} product analyses co

    except Exception as e:
        processing_results['error'] = str(e)
        print(f"❌ Spark processing failed: {e}")
        raise
```

```python
    context['task_instance'].xcom_push(key='processing_results', value=processing_resu
    return processing_results

# Data Warehouse Loading
def load_to_data_warehouse(**context):
    """Load processed data to PostgreSQL data warehouse"""
    print("💾 Starting data warehouse loading...")

    postgres_hook = PostgresHook(postgres_conn_id='postgres_data')
    loading_results = {}

    try:
        # Load customer analytics
        print("📊 Loading customer analytics to data warehouse...")
        customer_df = pd.read_parquet('/opt/airflow/data/processed/customer_360_analyt

        # Insert into database
        insert_sql = """
        INSERT INTO customer_analytics (
            customer_id, total_orders, total_spent, average_order_value,
            last_order_date, days_since_last_order, customer_segment
        ) VALUES %s
        ON CONFLICT (customer_id) DO UPDATE SET
            total_orders = EXCLUDED.total_orders,
            total_spent = EXCLUDED.total_spent,
            average_order_value = EXCLUDED.average_order_value,
            last_order_date = EXCLUDED.last_order_date,
            days_since_last_order = EXCLUDED.days_since_last_order,
            customer_segment = EXCLUDED.customer_segment,
            updated_at = CURRENT_TIMESTAMP
        """

        # Prepare data for insertion
        customer_values = [
            (
                row['customer_id'],
                int(row['total_orders']),
                float(row['total_spent']),
                float(row['avg_order_value']),
                row['last_order_date'],
                int(row['days_since_last_order']),
                row['customer_segment']
            )
            for _, row in customer_df.iterrows()
        ]

        postgres_hook.run(insert_sql, parameters=(customer_values,))

        loading_results['customer_analytics'] = {
            'status': 'success',
            'records_loaded': len(customer_df)
        }
        print(f"✅ Customer analytics: {len(customer_df)} records loaded")

        # Load data quality metrics
        print("📊 Recording data quality metrics...")
        quality_data = context['task_instance'].xcom_pull(task_ids='validate_integrate

        quality_insert_sql = """
        INSERT INTO data_quality_metrics (
            table_name, metric_name, metric_value, threshold_value, status, execution_
        ) VALUES (%s, %s, %s, %s, %s, %s)
        """

        quality_records = []
        for dataset, metrics in quality_data['quality_results'].items():
            if 'metrics' in metrics:
                for metric_name, value in metrics['metrics'].items():
                    threshold = 0.95 if metric_name != 'overall_quality' else 0.90
                    status = 'PASS' if value >= threshold else 'FAIL'
                    quality_records.append((
                        dataset, metric_name, value, threshold, status, context['ds']
                    ))

        for record in quality_records:
            postgres_hook.run(quality_insert_sql, parameters=record)

        loading_results['data_quality_metrics'] = {
            'status': 'success',
            'records_loaded': len(quality_records)
        }
        print(f"✅ Data quality metrics: {len(quality_records)} records loaded")

        # Log pipeline execution
        execution_log_sql = """
        INSERT INTO pipeline_execution_log (
            pipeline_name, execution_date, start_time, end_time, status, records_proce
```

```
        ) VALUES (%s, %s, %s, %s, %s, %s)
        """

        total_records = sum(result.get('records_loaded', 0) for result in loading_resu

        postgres_hook.run(execution_log_sql, parameters=(
            'integrated_data_platform',
            context['ds'],
            context['data_interval_start'],
            datetime.now(),
            'SUCCESS',
            total_records
        ))

        print(f"📊 Data warehouse loading completed: {total_records:,} total records")

    except Exception as e:
        loading_results['error'] = str(e)
        print(f"❌ Data warehouse loading failed: {e}")

        # Log failure
        try:
            execution_log_sql = """
            INSERT INTO pipeline_execution_log (
                pipeline_name, execution_date, start_time, end_time, status, error_mes
            ) VALUES (%s, %s, %s, %s, %s, %s)
            """
            postgres_hook.run(execution_log_sql, parameters=(
                'integrated_data_platform',
                context['ds'],
                context['data_interval_start'],
                datetime.now(),
                'FAILED',
                str(e)
            ))
        except:
            pass  # Don't fail on logging failure

        raise

    context['task_instance'].xcom_push(key='loading_results', value=loading_results)
    return loading_results

# Business Intelligence and Reporting
```

```
def generate_platform_insights(**context):
    """Generate comprehensive business insights from integrated data"""
    print("📊 Generating platform-wide business insights...")

    postgres_hook = PostgresHook(postgres_conn_id='postgres_data')
    insights = {}

    try:
        # Customer segment analysis
        segment_sql = """
        SELECT
            customer_segment,
            COUNT(*) as customer_count,
            AVG(total_spent) as avg_customer_value,
            SUM(total_spent) as segment_revenue,
            AVG(total_orders) as avg_orders_per_customer
        FROM customer_analytics
        GROUP BY customer_segment
        ORDER BY segment_revenue DESC
        """

        segment_data = postgres_hook.get_pandas_df(segment_sql)
        insights['customer_segments'] = segment_data.to_dict('records')

        # Top performing segments
        top_segment = segment_data.iloc[0]
        insights['top_performing_segment'] = {
            'segment': top_segment['customer_segment'],
            'revenue': float(top_segment['segment_revenue']),
            'customer_count': int(top_segment['customer_count'])
        }

        print(f"💰 Top performing segment: {top_segment['customer_segment']} "
              f"(${top_segment['segment_revenue']:,.2f} revenue)")

        # Data quality summary
        quality_sql = """
        SELECT
            table_name,
            metric_name,
            AVG(metric_value) as avg_score,
            COUNT(CASE WHEN status = 'PASS' THEN 1 END) as passed_checks,
            COUNT(*) as total_checks
        FROM data_quality_metrics
```

```
WHERE execution_date = %s
GROUP BY table_name, metric_name
"""

quality_data = postgres_hook.get_pandas_df(quality_sql, parameters=[context['d
insights['data_quality_summary'] = quality_data.to_dict('records')

# Overall platform health
overall_quality = quality_data['avg_score'].mean()
insights['platform_health'] = {
    'overall_quality_score': round(overall_quality, 4),
    'total_quality_checks': int(quality_data['total_checks'].sum()),
    'passed_quality_checks': int(quality_data['passed_checks'].sum())
}

print(f"📊 Platform health score: {overall_quality:.2%}")

# Generate recommendations
recommendations = []

# Customer segment recommendations
champions_pct = 0
at_risk_pct = 0
for segment in insights['customer_segments']:
    if segment['customer_segment'] == 'Champions':
        champions_pct = (segment['customer_count'] / sum(s['customer_count'] f
    elif segment['customer_segment'] == 'At Risk':
        at_risk_pct = (segment['customer_count'] / sum(s['customer_count'] for

if champions_pct < 20:
    recommendations.append({
        'type': 'customer_retention',
        'priority': 'high',
        'message': f'Only {champions_pct:.1f}% of customers are Champions. Imp
    })

if at_risk_pct > 25:
    recommendations.append({
        'type': 'customer_winback',
        'priority': 'high',
        'message': f'{at_risk_pct:.1f}% of customers are at risk. Launch win-b
    })

# Data quality recommendations
```

```
        if overall_quality < 0.95:
            recommendations.append({
                'type': 'data_quality',
                'priority': 'medium',
                'message': f'Data quality at {overall_quality:.1%}. Investigate data s
            })

        insights['recommendations'] = recommendations

        # Save insights report
        import json
        insights_path = '/opt/airflow/data/processed/platform_insights_report.json'
        with open(insights_path, 'w') as f:
            json.dump(insights, f, indent=2, default=str)

        print(f"📊 Platform insights report saved: {insights_path}")
        print(f"📊 Generated {len(recommendations)} business recommendations")

    except Exception as e:
        insights['error'] = str(e)
        print(f"❌ Insights generation failed: {e}")
        raise

    context['task_instance'].xcom_push(key='platform_insights', value=insights)
    return insights

# Platform monitoring and alerting
def monitor_platform_performance(**context):
    """Monitor platform performance and send alerts"""
    print("📊 Monitoring platform performance...")

    monitoring_results = {}

    try:
        # Collect performance metrics from XCom
        health_status = context['task_instance'].xcom_pull(task_ids='check_platform_he
        ingestion_results = context['task_instance'].xcom_pull(task_ids='ingest_multi_
        quality_results = context['task_instance'].xcom_pull(task_ids='validate_integr
        processing_results = context['task_instance'].xcom_pull(task_ids='process_data_
        loading_results = context['task_instance'].xcom_pull(task_ids='load_to_data_wa

        # Calculate platform performance score
        metrics = {
            'health_score': 1.0 if health_status.get('overall') else 0.0,
            'ingestion_success_rate': ingestion_results.get('successful_sources', 0) /
            'quality_pass_rate': quality_results.get('overall_pass_rate', 0),
            'processing_success': 1.0 if 'error' not in processing_results else 0.0,
            'loading_success': 1.0 if 'error' not in loading_results else 0.0
        }

        overall_platform_score = sum(metrics.values()) / len(metrics)

        monitoring_results = {
            'execution_date': context['ds'],
            'overall_platform_score': round(overall_platform_score, 4),
            'individual_metrics': metrics,
            'total_records_processed': ingestion_results.get('total_records_ingested',
            'pipeline_status': 'SUCCESS' if overall_platform_score >= 0.8 else 'DEGRAD
        }

        # Alert conditions
        alerts = []

        if overall_platform_score < 0.8:
            alerts.append({
                'severity': 'HIGH',
                'message': f'Platform performance degraded: {overall_platform_score:.1
            })

        if metrics['quality_pass_rate'] < 0.9:
            alerts.append({
                'severity': 'MEDIUM',
                'message': f'Data quality below threshold: {metrics["quality_pass_rate
            })

        if metrics['ingestion_success_rate'] < 1.0:
            alerts.append({
                'severity': 'MEDIUM',
                'message': f'Ingestion failures detected: {metrics["ingestion_success_
            })

        monitoring_results['alerts'] = alerts

        # Log monitoring results
        print(f"📊 Platform Performance Score: {overall_platform_score:.1%}")
        print(f"📊 Health: {metrics['health_score']:.1%}")
        print(f"📊 Ingestion: {metrics['ingestion_success_rate']:.1%}")
        print(f"📊 Quality: {metrics['quality_pass_rate']:.1%}")
```

```
        print(f"📊 Processing: {metrics['processing_success']:.1%}")
        print(f"📊 Loading: {metrics['loading_success']:.1%}")

        if alerts:
            print(f"🚨 {len(alerts)} alerts generated")
            for alert in alerts:
                print(f"   {alert['severity']}: {alert['message']}")
        else:
            print("✅ No alerts - all systems performing well")

    except Exception as e:
        monitoring_results['error'] = str(e)
        print(f"❌ Platform monitoring failed: {e}")

    context['task_instance'].xcom_push(key='monitoring_results', value=monitoring_resu
    return monitoring_results

# Task Group Definitions using Airflow TaskGroups
with TaskGroup('platform_initialization') as initialization_group:
    health_check_task = PythonOperator(
        task_id='check_platform_health',
        python_callable=check_platform_health,
        dag=dag
    )

with TaskGroup('data_ingestion') as ingestion_group:
    multi_source_ingestion_task = PythonOperator(
        task_id='ingest_multi_source_data',
        python_callable=ingest_multi_source_data,
        dag=dag
    )

with TaskGroup('data_quality') as quality_group:
    quality_validation_task = PythonOperator(
        task_id='validate_integrated_data_quality',
        python_callable=validate_integrated_data_quality,
        dag=dag
    )

with TaskGroup('data_processing') as processing_group:
    spark_processing_task = PythonOperator(
        task_id='process_data_with_spark',
        python_callable=process_data_with_spark,
        dag=dag
    )

with TaskGroup('data_warehouse') as warehouse_group:
    # Database preparation
    create_tables_task = PostgresOperator(
        task_id='ensure_database_schema',
        postgres_conn_id='postgres_data',
        sql='sql/schema/create_tables.sql',
        dag=dag
    )

    warehouse_loading_task = PythonOperator(
        task_id='load_to_data_warehouse',
        python_callable=load_to_data_warehouse,
        dag=dag
    )

    create_tables_task >> warehouse_loading_task

with TaskGroup('business_intelligence') as bi_group:
    insights_generation_task = PythonOperator(
        task_id='generate_platform_insights',
        python_callable=generate_platform_insights,
        dag=dag
    )

with TaskGroup('monitoring_and_alerting') as monitoring_group:
    monitoring_task = PythonOperator(
        task_id='monitor_platform_performance',
        python_callable=monitor_platform_performance,
        dag=dag
    )

    # Cleanup task
    cleanup_task = BashOperator(
        task_id='cleanup_temporary_files',
        bash_command="""
        echo "🧹 Cleaning up temporary files..."
        find /opt/airflow/data/staging -name "*.tmp" -delete 2>/dev/null || true
        find /opt/airflow/data/staging -name "*.log" -mtime +7 -delete 2>/dev/null ||
        echo "✅ Cleanup completed"
        """,
        dag=dag
    )
    monitoring_task >> cleanup_task

# Final success notification
platform_success_notification = BashOperator(
    task_id='send_platform_success_notification',
    bash_command="""
    echo "🎉 Integrated Data Platform execution completed successfully!"
    echo "📊 Check Airflow UI for detailed metrics and logs"
    echo "💾 Data available in PostgreSQL analytics database"
    echo "📈 Business insights report generated"
    """,
    trigger_rule='all_success',
    dag=dag
)

# Define task group dependencies
initialization_group >> ingestion_group >> quality_group >> processing_group >> wareho
```

🛠️ Spark Integration for Scalable Processing

⚡ Spark Application for Customer Analytics

File: `spark/apps/customer_analytics_spark.py`

python

```python
"""
Customer Analytics Spark Application
====================================
Production-ready Spark application for large-scale customer analytics
Handles millions of transactions and customer records efficiently
"""

from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark.sql.window import Window
import sys
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class CustomerAnalyticsSpark:
    def __init__(self, app_name="CustomerAnalytics"):
        """Initialize Spark session with optimized configuration"""
        self.spark = SparkSession.builder \
            .appName(app_name) \
            .config("spark.sql.adaptive.enabled", "true") \
            .config("spark.sql.adaptive.coalescePartitions.enabled", "true") \
            .config("spark.sql.adaptive.skewJoin.enabled", "true") \
            .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer") \
            .getOrCreate()

        self.spark.sparkContext.setLogLevel("WARN")
        logger.info("✅ Spark session initialized successfully")

    def load_data_sources(self, data_paths):
        """Load and validate all data sources"""
        logger.info("📊 Loading data sources...")

        data_frames = {}

        # Load e-commerce transactions
        if 'ecommerce' in data_paths:
            ecommerce_schema = StructType([
                StructField("transaction_id", StringType(), True),
                StructField("customer_id", StringType(), True),
                StructField("product_id", StringType(), True),
                StructField("transaction_date", TimestampType(), True),
                StructField("quantity", IntegerType(), True),
                StructField("unit_price", DoubleType(), True),
                StructField("total_amount", DoubleType(), True)
            ])

            data_frames['ecommerce'] = self.spark.read \
                .option("header", "true") \
                .schema(ecommerce_schema) \
                .csv(data_paths['ecommerce'])

            logger.info(f"✅ Loaded e-commerce data: {data_frames['ecommerce'].count()

        # Load superstore data
        if 'superstore' in data_paths:
            superstore_schema = StructType([
                StructField("order_id", StringType(), True),
                StructField("customer_id", StringType(), True),
                StructField("product_id", StringType(), True),
                StructField("order_date", TimestampType(), True),
                StructField("sales_amount", DoubleType(), True),
                StructField("profit_amount", DoubleType(), True),
                StructField("discount_rate", DoubleType(), True)
            ])

            data_frames['superstore'] = self.spark.read \
                .option("header", "true") \
                .schema(superstore_schema) \
                .csv(data_paths['superstore'])

            logger.info(f"✅ Loaded superstore data: {data_frames['superstore'].count(

        return data_frames

    def compute_customer_360(self, data_frames):
        """Compute comprehensive customer 360-degree view"""
        logger.info("🔄 Computing Customer 360 analytics...")

        # Combine all transaction sources
        all_transactions = None

        if 'ecommerce' in data_frames:
            ecommerce_unified = data_frames['ecommerce'].select(
                col("customer_id"),
                col("transaction_date").alias("date"),
                col("total_amount").alias("amount"),
                lit("ecommerce").alias("source")
            )
            all_transactions = ecommerce_unified

        if 'superstore' in data_frames:
            superstore_unified = data_frames['superstore'].select(
                col("customer_id"),
                col("order_date").alias("date"),
                col("sales_amount").alias("amount"),
                lit("superstore").alias("source")
            )

            if all_transactions is not None:
                all_transactions = all_transactions.union(superstore_unified)
            else:
                all_transactions = superstore_unified

        # Customer aggregations using Spark SQL
        customer_metrics = all_transactions.groupBy("customer_id").agg(
            sum("amount").alias("total_spent"),
            avg("amount").alias("avg_order_value"),
            count("*").alias("total_orders"),
            min("date").alias("first_order_date"),
            max("date").alias("last_order_date"),
            countDistinct("source").alias("channel_diversity")
        )

        # Calculate recency, frequency, monetary (RFM) scores
        current_date = current_timestamp()

        customer_rfm = customer_metrics.withColumn(
            "days_since_last_order",
            datediff(current_date, col("last_order_date"))
        ).withColumn(
            "customer_lifetime_days",
            datediff(col("last_order_date"), col("first_order_date"))
        )

        # RFM scoring using ntile window functions
        rfm_window = Window.orderBy("days_since_last_order")
        frequency_window = Window.orderBy("total_orders")
```

```
        monetary_window = Window.orderBy("total_spent")

        customer_scored = customer_rfm \
            .withColumn("recency_score", 6 - ntile(5).over(rfm_window)) \
            .withColumn("frequency_score", ntile(5).over(frequency_window)) \
            .withColumn("monetary_score", ntile(5).over(monetary_window))

        # Create RFM composite score
        customer_final = customer_scored.withColumn(
            "rfm_score",
            concat(
                col("recency_score").cast("string"),
                col("frequency_score").cast("string"),
                col("monetary_score").cast("string")
            )
        )

        # Customer segmentation based on RFM scores
        segmentation_conditions = [
            (col("rfm_score").rlike("^[4-5][4-5][4-5]$"), "Champions"),
            (col("rfm_score").rlike("^[3-5][3-5][3-5]$"), "Loyal Customers"),
            (col("rfm_score").rlike("^[3-5][1-2][3-5]$"), "Potential Loyalists"),
            (col("rfm_score").rlike("^[4-5][1-2][1-2]$"), "New Customers"),
            (col("rfm_score").rlike("^[1-2][3-5][3-5]$"), "At Risk"),
            (col("rfm_score").rlike("^[1-2][1-2][3-5]$"), "Cannot Lose Them"),
            (col("rfm_score").rlike("^[1-2][1-2][1-2]$"), "Lost Customers")
        ]

        # Apply segmentation
        customer_segmented = customer_final
        for condition, segment in segmentation_conditions:
            customer_segmented = customer_segmented.withColumn(
                "customer_segment",
                when(condition, segment).otherwise(col("customer_segment"))
            )

        # Fill any remaining null segments
        customer_segmented = customer_segmented.fillna({"customer_segment": "Others"})

        logger.info(f"✅ Customer 360 completed: {customer_segmented.count()} customer
        return customer_segmented

    def compute_product_analytics(self, data_frames):
        """Compute product performance analytics"""
        logger.info("🔄 Computing product analytics...")

        # Combine product transaction data
        all_product_transactions = None

        if 'ecommerce' in data_frames:
            ecommerce_products = data_frames['ecommerce'].select(
                col("product_id"),
                col("quantity"),
                col("total_amount"),
                col("customer_id"),
                col("transaction_date").alias("date")
            )
            all_product_transactions = ecommerce_products

        # Product performance metrics
        product_performance = all_product_transactions.groupBy("product_id").agg(
            sum("quantity").alias("total_quantity_sold"),
            sum("total_amount").alias("total_revenue"),
            avg("total_amount").alias("avg_order_value"),
            countDistinct("customer_id").alias("unique_customers"),
            count("*").alias("total_transactions")
        )

        # Calculate additional metrics
        product_metrics = product_performance.withColumn(
            "revenue_per_customer",
            round(col("total_revenue") / col("unique_customers"), 2)
        ).withColumn(
            "avg_quantity_per_transaction",
            round(col("total_quantity_sold") / col("total_transactions"), 2)
        )

        logger.info(f"✅ Product analytics completed: {product_metrics.count()} produc
        return product_metrics

    def save_results(self, dataframes, output_paths):
        """Save processed results to specified formats"""
        logger.info("💾 Saving processed results...")

        for name, df in dataframes.items():
            if name in output_paths:
                output_path = output_paths[name]
                # Save as Parquet for optimal performance
                df.coalesce(1).write \
                    .mode("overwrite") \
                    .option("compression", "snappy") \
                    .parquet(output_path)

                logger.info(f"✅ Saved {name} to {output_path}")

    def close(self):
        """Close Spark session"""
        self.spark.stop()
        logger.info("✅ Spark session closed")

def main():
    """Main execution function"""
    logger.info("🚀 Starting Customer Analytics Spark Job...")

    # Initialize Spark application
    analytics = CustomerAnalyticsSpark("IntegratedCustomerAnalytics")

    try:
        # Define data paths
        data_paths = {
            'ecommerce': '/opt/spark-data/staging/ecommerce_standardized.parquet',
            'superstore': '/opt/spark-data/staging/superstore_standardized.parquet'
        }

        # Load data
        data_frames = analytics.load_data_sources(data_paths)

        # Compute analytics
        customer_360 = analytics.compute_customer_360(data_frames)
        product_analytics = analytics.compute_product_analytics(data_frames)

        # Output paths
        output_paths = {
            'customer_360': '/opt/spark-data/processed/spark_customer_360',
            'product_analytics': '/opt/spark-data/processed/spark_product_analytics'
        }

        # Save results
        analytics.save_results({
            'customer_360': customer_360,
            'product_analytics': product_analytics
        }, output_paths)

        # Print summary statistics
        logger.info("📊 Job Summary:")
        logger.info(f"   Customer profiles processed: {customer_360.count():,}")
        logger.info(f"   Products analyzed: {product_analytics.count():,}")

        # Segment distribution
        segment_counts = customer_360.groupBy("customer_segment").count().collect()
        logger.info("📊 Customer Segment Distribution:")
        for row in segment_counts:
            logger.info(f"   {row['customer_segment']}: {row['count']:,} customers")

        logger.info("🎉 Customer Analytics Spark Job completed successfully!")

    except Exception as e:
        logger.error(f"❌ Spark job failed: {str(e)}")
        raise

    finally:
        analytics.close()

if __name__ == "__main__":
    main()
```

📊 Monitoring and Observability Implementation

🎯 Data Quality Monitoring Framework

File: `scripts/data_quality/quality_checks.py`

python

```python
"""
Comprehensive Data Quality Monitoring Framework
===============================================
Implements Great Expectations-like validation with custom business rules
"""

import pandas as pd
import numpy as np
import json
from datetime import datetime, timedelta
from pathlib import Path
import logging

logger = logging.getLogger(__name__)

class DataQualityFramework:
    def __init__(self, config_path=None):
        """Initialize data quality framework with configurable rules"""
        self.config = self._load_config(config_path)
        self.results = {}

    def _load_config(self, config_path):
        """Load data quality configuration"""
        default_config = {
            "completeness_threshold": 0.95,
            "uniqueness_threshold": 0.99,
            "validity_threshold": 0.90,
            "freshness_threshold_hours": 24,
            "anomaly_detection_window": 30
        }

        if config_path and Path(config_path).exists():
            with open(config_path) as f:
                custom_config = json.load(f)
                default_config.update(custom_config)

        return default_config

    def validate_completeness(self, df, dataset_name, required_columns=None):
        """Validate data completeness across columns"""
        logger.info(f"🔍 Validating completeness for {dataset_name}...")

        if required_columns is None:
            required_columns = df.columns.tolist()

        completeness_results = {}

        for column in required_columns:
            if column in df.columns:
                non_null_ratio = df[column].notna().sum() / len(df)
                completeness_results[column] = {
                    'completeness_ratio': round(non_null_ratio, 4),
                    'null_count': int(df[column].isna().sum()),
                    'total_count': len(df),
                    'passed': non_null_ratio >= self.config['completeness_threshold']
                }
            else:
                completeness_results[column] = {
                    'error': 'Column not found',
                    'passed': False
                }

        # Overall completeness score
        overall_completeness = np.mean([
            result['completeness_ratio'] for result in completeness_results.values()
            if 'completeness_ratio' in result
        ])

        self.results[f'{dataset_name}_completeness'] = {
            'overall_score': round(overall_completeness, 4),
            'threshold': self.config['completeness_threshold'],
            'passed': overall_completeness >= self.config['completeness_threshold'],
            'column_details': completeness_results
        }

        status = "✅" if overall_completeness >= self.config['completeness_threshold']
        logger.info(f"{status} Completeness for {dataset_name}: {overall_completeness:

        return self.results[f'{dataset_name}_completeness']

    def validate_uniqueness(self, df, dataset_name, unique_columns=None):
        """Validate uniqueness constraints"""
        logger.info(f"🔍 Validating uniqueness for {dataset_name}...")

        if unique_columns is None:
            # Auto-detect ID columns
            unique_columns = [col for col in df.columns if 'id' in col.lower()]
        uniqueness_results = {}

        for column in unique_columns:
            if column in df.columns:
                unique_ratio = df[column].nunique() / len(df)
                duplicate_count = len(df) - df[column].nunique()

                uniqueness_results[column] = {
                    'uniqueness_ratio': round(unique_ratio, 4),
                    'unique_count': int(df[column].nunique()),
                    'duplicate_count': duplicate_count,
                    'total_count': len(df),
                    'passed': unique_ratio >= self.config['uniqueness_threshold']
                }
            else:
                uniqueness_results[column] = {
                    'error': 'Column not found',
                    'passed': False
                }

        # Overall uniqueness score
        if uniqueness_results:
            overall_uniqueness = np.mean([
                result['uniqueness_ratio'] for result in uniqueness_results.values()
                if 'uniqueness_ratio' in result
            ])
        else:
            overall_uniqueness = 1.0  # No unique columns to check

        self.results[f'{dataset_name}_uniqueness'] = {
            'overall_score': round(overall_uniqueness, 4),
            'threshold': self.config['uniqueness_threshold'],
            'passed': overall_uniqueness >= self.config['uniqueness_threshold'],
            'column_details': uniqueness_results
        }

        status = "✅" if overall_uniqueness >= self.config['uniqueness_threshold'] els
        logger.info(f"{status} Uniqueness for {dataset_name}: {overall_uniqueness:.2%}

        return self.results[f'{dataset_name}_uniqueness']

    def validate_business_rules(self, df, dataset_name, rules=None):
        """Validate custom business rules"""
        logger.info(f"🔍 Validating business rules for {dataset_name}...")

        if rules is None:
            rules = self._get_default_business_rules(dataset_name)

        rule_results = {}

        for rule_name, rule_config in rules.items():
            try:
                rule_function = rule_config['function']
                rule_threshold = rule_config.get('threshold', 0.95)

                # Apply business rule
                if rule_function == 'positive_amounts':
                    amount_columns = [col for col in df.columns if 'amount' in col.low
                    valid_ratios = []
                    for col in amount_columns:
                        if col in df.columns and df[col].dtype in ['float64', 'int64']
                            valid_ratio = (df[col] >= 0).sum() / len(df)
                            valid_ratios.append(valid_ratio)

                    overall_validity = np.mean(valid_ratios) if valid_ratios else 1.0

                elif rule_function == 'valid_dates':
                    date_columns = [col for col in df.columns if 'date' in col.lower()
                    valid_ratios = []
                    for col in date_columns:
                        if col in df.columns:
                            try:
                                pd.to_datetime(df[col], errors='coerce')
                                valid_dates = pd.to_datetime(df[col], errors='coerce')
                                valid_ratio = valid_dates.sum() / len(df)
                                valid_ratios.append(valid_ratio)
                            except:
                                valid_ratios.append(0.0)

                    overall_validity = np.mean(valid_ratios) if valid_ratios else 1.0

                elif rule_function == 'reasonable_quantities':
                    quantity_columns = [col for col in df.columns if 'quantity' in col
                    valid_ratios = []
                    for col in quantity_columns:
                        if col in df.columns and df[col].dtype in ['float64', 'int64']
                            # Reasonable quantities: between 1 and 1000
```

```python
                            valid_ratio = ((df[col] >= 1) & (df[col] <= 1000)).sum() /
                            valid_ratios.append(valid_ratio)

                    overall_validity = np.mean(valid_ratios) if valid_ratios else 1.0

                else:
                    overall_validity = 1.0  # Unknown rule, pass by default

                rule_results[rule_name] = {
                    'validity_score': round(overall_validity, 4),
                    'threshold': rule_threshold,
                    'passed': overall_validity >= rule_threshold,
                    'rule_function': rule_function
                }

            except Exception as e:
                rule_results[rule_name] = {
                    'error': str(e),
                    'passed': False
                }

        # Overall business rule score
        if rule_results:
            overall_business_validity = np.mean([
                result['validity_score'] for result in rule_results.values()
                if 'validity_score' in result
            ])
        else:
            overall_business_validity = 1.0

        self.results[f'{dataset_name}_business_rules'] = {
            'overall_score': round(overall_business_validity, 4),
            'threshold': self.config['validity_threshold'],
            'passed': overall_business_validity >= self.config['validity_threshold'],
            'rule_details': rule_results
        }

        status = "✅" if overall_business_validity >= self.config['validity_threshold'
        logger.info(f"{status} Business rules for {dataset_name}: {overall_business_va

        return self.results[f'{dataset_name}_business_rules']

    def _get_default_business_rules(self, dataset_name):
        """Get default business rules based on dataset type"""
        rules = {
            'positive_amounts': {
                'function': 'positive_amounts',
                'threshold': 0.95,
                'description': 'All amount/price fields should be positive'
            },
            'valid_dates': {
                'function': 'valid_dates',
                'threshold': 0.99,
                'description': 'All date fields should be valid dates'
            },
            'reasonable_quantities': {
                'function': 'reasonable_quantities',
                'threshold': 0.95,
                'description': 'Quantities should be reasonable (1-1000)'
            }
        }

        return rules

    def detect_anomalies(self, df, dataset_name, metric_columns=None):
        """Detect statistical anomalies in numeric data"""
        logger.info(f"🔍 Detecting anomalies for {dataset_name}...")

        if metric_columns is None:
            metric_columns = df.select_dtypes(include=[np.number]).columns.tolist()

        anomaly_results = {}

        for column in metric_columns:
            if column in df.columns and df[column].dtype in ['float64', 'int64']:
                try:
                    # Z-score based anomaly detection
                    z_scores = np.abs((df[column] - df[column].mean()) / df[column].st
                    anomaly_threshold = 3  # 3 standard deviations

                    anomalies = z_scores > anomaly_threshold
                    anomaly_count = anomalies.sum()
                    anomaly_ratio = anomaly_count / len(df)

                    anomaly_results[column] = {
                        'anomaly_count': int(anomaly_count),
                        'anomaly_ratio': round(anomaly_ratio, 4),
                        'total_count': len(df),
```

```python
                        'mean_value': round(df[column].mean(), 2),
                        'std_value': round(df[column].std(), 2),
                        'passed': anomaly_ratio <= 0.05  # Less than 5% anomalies
                    }

                except Exception as e:
                    anomaly_results[column] = {
                        'error': str(e),
                        'passed': False
                    }

        # Overall anomaly assessment
        if anomaly_results:
            overall_anomaly_score = 1 - np.mean([
                result['anomaly_ratio'] for result in anomaly_results.values()
                if 'anomaly_ratio' in result
            ])
        else:
            overall_anomaly_score = 1.0

        self.results[f'{dataset_name}_anomalies'] = {
            'overall_score': round(overall_anomaly_score, 4),
            'threshold': 0.95,
            'passed': overall_anomaly_score >= 0.95,
            'column_details': anomaly_results
        }

        status = "✅" if overall_anomaly_score >= 0.95 else "❌"
        logger.info(f"{status} Anomaly detection for {dataset_name}: {overall_anomaly_

        return self.results[f'{dataset_name}_anomalies']

    def generate_quality_report(self, output_path):
        """Generate comprehensive data quality report"""
        logger.info("📊 Generating data quality report...")

        report = {
            'execution_timestamp': datetime.now().isoformat(),
            'config': self.config,
            'results': self.results,
            'summary': self._calculate_summary_metrics()
        }

        # Save report
        with open(output_path, 'w') as f:
            json.dump(report, f, indent=2, default=str)

        logger.info(f"✅ Quality report saved to {output_path}")
        return report

    def _calculate_summary_metrics(self):
        """Calculate overall summary metrics"""
        all_scores = []
        passed_checks = 0
        total_checks = 0

        for check_name, result in self.results.items():
            if 'overall_score' in result:
                all_scores.append(result['overall_score'])
            if 'passed' in result:
                total_checks += 1
                if result['passed']:
                    passed_checks += 1

        return {
            'overall_quality_score': round(np.mean(all_scores), 4) if all_scores else
            'checks_passed': passed_checks,
            'total_checks': total_checks,
            'pass_rate': round(passed_checks / total_checks, 4) if total_checks > 0 el
        }

def run_comprehensive_quality_checks(data_paths, output_path):
    """Run comprehensive quality checks on all datasets"""
    logger.info("🚀 Starting comprehensive data quality validation...")

    quality_framework = DataQualityFramework()

    for dataset_name, file_path in data_paths.items():
        if Path(file_path).exists():
            logger.info(f"📊 Processing {dataset_name}...")

            try:
                # Load dataset
                if file_path.endswith('.parquet'):
                    df = pd.read_parquet(file_path)
                else:
                    df = pd.read_csv(file_path)
```

```python
                # Run all quality checks
                quality_framework.validate_completeness(df, dataset_name)
                quality_framework.validate_uniqueness(df, dataset_name)
                quality_framework.validate_business_rules(df, dataset_name)
                quality_framework.detect_anomalies(df, dataset_name)

                logger.info(f"✅ Quality checks completed for {dataset_name}")

            except Exception as e:
                logger.error(f"❌ Quality checks failed for {dataset_name}: {e}")
        else:
            logger.warning(f"⚠️ Dataset not found: {file_path}")

    # Generate comprehensive report
    report = quality_framework.generate_quality_report(output_path)

    logger.info("🎉 Comprehensive quality validation completed!")
    return report

if __name__ == "__main__":
    # Example usage
    data_paths = {
        'ecommerce': '/opt/airflow/data/staging/ecommerce_standardized.parquet',
        'superstore': '/opt/airflow/data/staging/superstore_standardized.parquet',
        'products': '/opt/airflow/data/staging/products_standardized.parquet'
    }

    output_path = '/opt/airflow/data/processed/quality_report.json'
    run_comprehensive_quality_checks(data_paths, output_path)
```

🔧 Platform Configuration and Deployment

🎯 Environment-Specific Configuration

File: `config/platform_config.yaml`

yaml

```yaml
# Platform Configuration for Different Environments
# =================================================

environments:
  development:
    database:
      host: "localhost"
      port: 5434
      database: "analytics_dev"
      username: "datauser"
      password: "datapass"

    spark:
      master: "local[2]"
      executor_memory: "1g"
      driver_memory: "512m"

    airflow:
      parallelism: 2
      max_active_runs: 1
      catchup: false

    data_retention:
      staging_days: 7
      processed_days: 30
      logs_days: 14

  staging:
    database:
      host: "postgres-staging"
      port: 5432
      database: "analytics_staging"
      username: "datauser"
      password: "${POSTGRES_PASSWORD}"

    spark:
      master: "spark://spark-master:7077"
      executor_memory: "2g"
      driver_memory: "1g"

    airflow:
      parallelism: 4
      max_active_runs: 2
      catchup: false

    data_retention:
      staging_days: 14
      processed_days: 60
      logs_days: 30

  production:
    database:
      host: "postgres-prod"
      port: 5432
      database: "analytics_prod"
      username: "datauser"
      password: "${POSTGRES_PASSWORD}"

    spark:
      master: "spark://spark-master:7077"
      executor_memory: "4g"
      driver_memory: "2g"

    airflow:
      parallelism: 8
      max_active_runs: 1
      catchup: false

    data_retention:
      staging_days: 30
      processed_days: 365
      logs_days: 90

data_quality:
  thresholds:
    completeness: 0.95
    uniqueness: 0.99
    validity: 0.90
    freshness_hours: 24
    anomaly_threshold: 0.05

  alerts:
    email_recipients:
      - "data-team@company.com"
      - "devops@company.com"
    slack_webhook: "${SLACK_WEBHOOK_URL}"
monitoring:
  health_check_interval: 300  # 5 minutes
  metric_collection_interval: 60  # 1 minute
  alert_cooldown: 1800  # 30 minutes

security:
  encrypt_at_rest: true
  encrypt_in_transit: true
  access_logging: true
  data_masking:
    pii_columns:
      - "email"
      - "phone"
      - "ssn"
    masking_strategy: "hash"
```

🚀 Deployment Scripts

File: `scripts/deployment/deploy_platform.sh` bash

```bash
#!/bin/bash
# Platform Deployment Script
# ==========================

set -e  # Exit on any error

echo "🚀 Starting Integrated Data Platform Deployment..."

# Configuration
ENVIRONMENT=${1:-development}
PROJECT_NAME="week2-platform"
COMPOSE_FILE="docker-compose.yml"

echo "📊 Deployment Environment: $ENVIRONMENT"

# Validate environment
if [[ ! "$ENVIRONMENT" =~ ^(development|staging|production)$ ]]; then
    echo "❌ Invalid environment. Use: development, staging, or production"
    exit 1
fi

# Pre-deployment checks
echo "🔍 Running pre-deployment checks..."

# Check Docker
if ! command -v docker &> /dev/null; then
    echo "❌ Docker is not installed"
    exit 1
fi

if ! command -v docker-compose &> /dev/null; then
    echo "❌ Docker Compose is not installed"
    exit 1
fi

echo "✅ Docker and Docker Compose are available"

# Check system resources
AVAILABLE_MEMORY=$(free -m | awk 'NR==2{printf "%.0f", $7/1024 }')
if [ "$AVAILABLE_MEMORY" -lt 4 ]; then
    echo "⚠️  Warning: Available memory is ${AVAILABLE_MEMORY}GB. Recommend at least 4
fi
```

```
# Set environment variables
export AIRFLOW_UID=$(id -u)
export COMPOSE_PROJECT_NAME=$PROJECT_NAME
export ENVIRONMENT=$ENVIRONMENT

echo "📁 Creating required directories..."
mkdir -p data/{raw,staging,processed,archive}
mkdir -p airflow/{dags,logs,plugins}
mkdir -p logs/{airflow,spark,postgres}
mkdir -p monitoring/{grafana,prometheus}

# Set proper permissions
chmod -R 755 data
chmod -R 755 airflow
chmod -R 755 logs

echo "✅ Directory structure created"

# Download required datasets (if not present)
echo "📊 Checking dataset availability..."
REQUIRED_FILES=(
    "data/raw/ecommerce_transactions.csv"
    "data/raw/superstore_sales.csv"
    "data/raw/amazon_products.csv"
    "data/raw/retail_analytics.csv"
)

MISSING_FILES=0
for file in "${REQUIRED_FILES[@]}"; do
    if [ ! -f "$file" ]; then
        echo "⚠️  Missing: $file"
        MISSING_FILES=$((MISSING_FILES + 1))
    fi
done

if [ $MISSING_FILES -gt 0 ]; then
    echo "📥 $MISSING_FILES datasets missing. Creating sample data..."
    python3 scripts/ingestion/download_datasets.py --create-samples
    echo "✅ Sample datasets created"
else
    echo "✅ All datasets available"
fi

# Environment-specific configuration
```

```
case $ENVIRONMENT in
    development)
        echo "🔧 Configuring for development environment..."
        export AIRFLOW_PARALLELISM=2
        export SPARK_WORKER_MEMORY=1G
        export POSTGRES_MAX_CONNECTIONS=50
        ;;
    staging)
        echo "🔧 Configuring for staging environment..."
        export AIRFLOW_PARALLELISM=4
        export SPARK_WORKER_MEMORY=2G
        export POSTGRES_MAX_CONNECTIONS=100
        ;;
    production)
        echo "🔧 Configuring for production environment..."
        export AIRFLOW_PARALLELISM=8
        export SPARK_WORKER_MEMORY=4G
        export POSTGRES_MAX_CONNECTIONS=200
        ;;
esac

# Initialize Airflow
echo "🌪 Initializing Airflow..."
docker-compose up airflow-init

# Start platform services
echo "🚀 Starting platform services..."
docker-compose up -d

# Wait for services to be healthy
echo "⏳ Waiting for services to be healthy..."
MAX_WAIT=300  # 5 minutes
WAIT_TIME=0

while [ $WAIT_TIME -lt $MAX_WAIT ]; do
    if docker-compose ps | grep -q "unhealthy\|starting"; then
        echo "⏳ Services still starting... ($WAIT_TIME/$MAX_WAIT seconds)"
        sleep 10
        WAIT_TIME=$((WAIT_TIME + 10))
    else
        break
    fi
done
```

```
# Health check
echo "🔍 Running platform health check..."
HEALTH_SCRIPT="scripts/monitoring/health_checks.py"
if [ -f "$HEALTH_SCRIPT" ]; then
    python3 "$HEALTH_SCRIPT"
    if [ $? -eq 0 ]; then
        echo "✅ Platform health check passed"
    else
        echo "❌ Platform health check failed"
        echo "📋 Service status:"
        docker-compose ps
        exit 1
    fi
else
    echo "⚠️  Health check script not found, skipping..."
fi

# Display access information
echo ""
echo "🎉 Platform deployment completed successfully!"
echo ""
echo "📊 Access Information:"
echo "   Airflow Web UI:    http://localhost:8080 (airflow/airflow)"
echo "   Spark Master UI:   http://localhost:8081"
echo "   Jupyter Lab:       http://localhost:8888"
echo "   PostgreSQL Data:   localhost:5434 (datauser/datapass/analytics)"
echo ""
echo "🔧 Management Commands:"
echo "   View logs:         docker-compose logs -f [service]"
echo "   Stop platform:     docker-compose down"
echo "   Restart service:   docker-compose restart [service]"
echo "   Scale workers:     docker-compose up -d --scale spark-worker=3"
echo ""
echo "📚 Next Steps:"
echo "   1. Access Airflow UI and unpause the 'integrated_data_platform' DAG"
echo "   2. Trigger a manual run to test the complete pipeline"
echo "   3. Monitor execution in Airflow UI and check logs"
echo "   4. Query results in PostgreSQL analytics database"
echo ""
```

🔍 Health Monitoring Script

File: `scripts/monitoring/health_checks.py`

python

```
"""
Platform Health Monitoring
==========================
Comprehensive health checks for all platform components
"""

import subprocess
import psycopg2
import requests
import time
import json
from datetime import datetime
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class PlatformHealthMonitor:
    def __init__(self):
        self.health_results = {}
        self.overall_health = True

    def check_docker_services(self):
        """Check if all Docker services are running"""
        logger.info("🐳 Checking Docker services...")

        required_services = [
            'postgres-airflow',
            'postgres-data',
            'redis',
            'airflow-webserver',
            'airflow-scheduler',
            'spark-master',
            'spark-worker',
            'jupyter'
        ]

        try:
            result = subprocess.run(
                ['docker-compose', 'ps', '--format', 'json'],
                capture_output=True,
                text=True,
                check=True
            )

            services_status = {}
            for line in result.stdout.strip().split('\n'):
                if line.strip():
                    service_info = json.loads(line)
                    service_name = service_info['Service']
                    service_state = service_info['State']

                    services_status[service_name] = {
                        'status': service_state,
                        'healthy': service_state == 'running'
                    }

            # Check required services
            missing_services = []
            unhealthy_services = []

            for service in required_services:
                if service not in services_status:
                    missing_services.append(service)
                elif not services_status[service]['healthy']:
                    unhealthy_services.append(service)

            docker_health = {
                'all_services_running': len(missing_services) == 0 and len(unhealthy_s
                'running_services': len([s for s in services_status.values() if s['hea
                'total_expected': len(required_services),
                'missing_services': missing_services,
                'unhealthy_services': unhealthy_services,
                'service_details': services_status
            }

            self.health_results['docker_services'] = docker_health

            if not docker_health['all_services_running']:
                self.overall_health = False
                logger.error(f"❌ Docker services issues: {len(missing_services)} miss
            else:
                logger.info("✅ All Docker services running")

        except Exception as e:
            self.health_results['docker_services'] = {'error': str(e), 'healthy': Fals
            self.overall_health = False
            logger.error(f"❌ Docker services check failed: {e}")

    def check_airflow_health(self):
        """Check Airflow web server and scheduler health"""
        logger.info("🌪️ Checking Airflow health...")

        try:
            # Check web server
            response = requests.get('http://localhost:8080/health', timeout=10)
            webserver_healthy = response.status_code == 200

            # Check if we can access the main page
            main_response = requests.get('http://localhost:8080/', timeout=10)
            webserver_accessible = main_response.status_code == 200

            airflow_health = {
                'webserver_healthy': webserver_healthy,
                'webserver_accessible': webserver_accessible,
                'webserver_response_time': response.elapsed.total_seconds()
            }

            self.health_results['airflow'] = airflow_health

            if webserver_healthy and webserver_accessible:
                logger.info("✅ Airflow web server healthy")
            else:
                self.overall_health = False
                logger.error("❌ Airflow web server issues detected")

        except Exception as e:
            self.health_results['airflow'] = {'error': str(e), 'healthy': False}
            self.overall_health = False
            logger.error(f"❌ Airflow health check failed: {e}")

    def check_spark_health(self):
        """Check Spark cluster health"""
        logger.info("⚡ Checking Spark cluster health...")

        try:
            # Check Spark master UI
            response = requests.get('http://localhost:8081', timeout=10)
            master_healthy = response.status_code == 200

            # Try to get cluster info
            json_response = requests.get('http://localhost:8081/json/', timeout=10)
            cluster_info = {}

            if json_response.status_code == 200:
                cluster_data = json_response.json()
                cluster_info = {
                    'workers_count': len(cluster_data.get('workers', [])),
                    'alive_workers': len([w for w in cluster_data.get('workers', []) i
                    'cores_total': cluster_data.get('cores', 0),
                    'memory_total': cluster_data.get('memory', 0)
                }

            spark_health = {
                'master_healthy': master_healthy,
                'cluster_info': cluster_info,
                'all_workers_alive': cluster_info.get('workers_count', 0) == cluster_i
            }

            self.health_results['spark'] = spark_health

            if master_healthy and spark_health['all_workers_alive']:
                logger.info(f"✅ Spark cluster healthy: {cluster_info.get('alive_worke
            else:
                self.overall_health = False
                logger.error("❌ Spark cluster issues detected")

        except Exception as e:
            self.health_results['spark'] = {'error': str(e), 'healthy': False}
            self.overall_health = False
            logger.error(f"❌ Spark health check failed: {e}")

    def check_database_connectivity(self):
        """Check PostgreSQL database connectivity"""
        logger.info("🐘 Checking database connectivity...")

        databases = {
            'airflow_metadata': {
                'host': 'localhost',
                'port': 5433,
                'database': 'airflow',
                'username': 'airflow',
                'password': 'airflow'
            },
            'analytics_data': {
```

```
        'host': 'localhost',
        'port': 5434,
        'database': 'analytics',
        'username': 'datauser',
        'password': 'datapass'
    }
}

database_results = {}

for db_name, config in databases.items():
    try:
        conn = psycopg2.connect(
            host=config['host'],
            port=config['port'],
            database=config['database'],
            user=config['username'],
            password=config['password'],
            connect_timeout=10
        )

        cursor = conn.cursor()
        cursor.execute("SELECT version();")
        version = cursor.fetchone()[0]

        cursor.execute("SELECT current_timestamp;")
        timestamp = cursor.fetchone()[0]

        conn.close()

        database_results[db_name] = {
            'connected': True,
            'version': version,
            'timestamp': str(timestamp)
        }

        logger.info(f"✅ {db_name} database connected")

    except Exception as e:
        database_results[db_name] = {
            'connected': False,
            'error': str(e)
        }
        self.overall_health = False
```

```
                logger.error(f"❌ {db_name} database connection failed: {e}")

        self.health_results['databases'] = database_results

    def check_data_availability(self):
        """Check if required data files are available"""
        logger.info("📊 Checking data availability...")

        from pathlib import Path
        import pandas as pd

        required_files = {
            'ecommerce_raw': 'data/raw/ecommerce_transactions.csv',
            'superstore_raw': 'data/raw/superstore_sales.csv',
            'products_raw': 'data/raw/amazon_products.csv',
            'retail_raw': 'data/raw/retail_analytics.csv'
        }

        data_results = {}

        for file_name, file_path in required_files.items():
            try:
                path = Path(file_path)
                if path.exists():
                    # Quick validation
                    df = pd.read_csv(file_path, nrows=5)
                    file_size = path.stat().st_size / (1024 * 1024)  # MB

                    data_results[file_name] = {
                        'available': True,
                        'size_mb': round(file_size, 2),
                        'columns': len(df.columns),
                        'sample_rows': len(df)
                    }

                    logger.info(f"✅ {file_name}: {file_size:.1f}MB, {len(df.columns)}
                else:
                    data_results[file_name] = {
                        'available': False,
                        'error': 'File not found'
                    }
                    logger.warning(f"⚠️ {file_name}: File not found")

            except Exception as e:
                data_results[file_name] = {
                    'available': False,
                    'error': str(e)
                }
                logger.error(f"❌ {file_name}: Error reading file - {e}")

        available_files = sum(1 for result in data_results.values() if result.get('ava
        total_files = len(required_files)

        self.health_results['data_files'] = {
            'files_available': available_files,
            'total_files': total_files,
            'availability_rate': available_files / total_files,
            'file_details': data_results
        }

        if available_files < total_files:
            logger.warning(f"⚠️ {available_files}/{total_files} data files available"
        else:
            logger.info(f"✅ All {total_files} data files available")

    def generate_health_report(self):
        """Generate comprehensive health report"""
        logger.info("📊 Generating health report...")

        report = {
            'timestamp': datetime.now().isoformat(),
            'overall_health': self.overall_health,
            'checks_performed': len(self.health_results),
            'health_results': self.health_results
        }

        # Calculate health score
        healthy_components = 0
        total_components = 0

        for component, result in self.health_results.items():
            total_components += 1
            if component == 'docker_services':
                healthy_components += 1 if result.get('all_services_running') else 0
            elif component == 'airflow':
                healthy_components += 1 if result.get('webserver_healthy') else 0
            elif component == 'spark':
                healthy_components += 1 if result.get('master_healthy') else 0
            elif component == 'databases':
                db_healthy = all(db.get('connected', False) for db in result.values())
                healthy_components += 1 if db_healthy else 0
            elif component == 'data_files':
                healthy_components += 1 if result.get('availability_rate', 0) >= 0.8 e

        health_score = healthy_components / total_components if total_components > 0 e
        report['health_score'] = round(health_score, 4)

        # Save report
        report_path = 'logs/health_report.json'
        with open(report_path, 'w') as f:
            json.dump(report, f, indent=2)

        logger.info(f"📊 Health report saved: {report_path}")
        logger.info(f"📊 Overall health score: {health_score:.1%}")

        return report

    def run_complete_health_check(self):
        """Run all health checks"""
        logger.info("🚀 Starting comprehensive platform health check...")

        start_time = time.time()

        # Run all checks
        self.check_docker_services()
        self.check_airflow_health()
        self.check_spark_health()
        self.check_database_connectivity()
        self.check_data_availability()

        # Generate report
        report = self.generate_health_report()

        execution_time = time.time() - start_time
        logger.info(f"⏱️  Health check completed in {execution_time:.2f} seconds")

        if self.overall_health:
            logger.info("🎉 Platform health check: ALL SYSTEMS GO!")
            return 0
        else:
            logger.error("⚠️  Platform health check: ISSUES DETECTED")
            return 1
```

```
def main():
    """Main execution function"""
    monitor = PlatformHealthMonitor()
    exit_code = monitor.run_complete_health_check()
    return exit_code

if __name__ == "__main__":
    exit(main())
```

📚 Day 14 Success Metrics and Learning Outcomes

🎯 Knowledge Integration Assessment

Core Concepts Mastered:

1. **Systems Architecture Thinking**
    - Multi-component integration patterns
    - Service discovery and communication
    - Dependency management across tools
    - Scalability and performance considerations
2. **Platform Engineering Fundamentals**
    - Infrastructure as Code principles
    - Environment-specific configuration
    - Monitoring and observability
    - Automated deployment strategies
3. **Data Pipeline Orchestration**
    - Cross-tool workflow coordination
    - Error handling and recovery
    - Data lineage and quality gates
    - Performance monitoring
4. **Production Readiness**
    - Security best practices
    - Logging and alerting
    - Backup and recovery
    - Documentation and maintenance

📊 Technical Achievements

Week 2 Integration Milestones:

- ✅ 8 technology components integrated
- ✅ End-to-end data pipeline operational
- ✅ Multi-source data processing capability
- ✅ Production-ready monitoring system
- ✅ Automated quality validation framework
- ✅ Scalable architecture foundation

Performance Benchmarks:

- 📊 Data Processing: 100K+ records/minute capability
- 🛠 System Availability: 99.9% uptime target
- ⏱ Pipeline Execution: < 15 minutes end-to-end
- 🔧 Error Recovery: < 5 minutes MTTR
- 📈 Scalability: Horizontal scaling ready

🎓 Business Value Delivered

Operational Improvements:

- 📊 **Data Freshness**: From daily → hourly updates
- 🔄 **Automation Level**: 95% of manual tasks eliminated
- 📈 **Data Quality**: Consistent 99%+ accuracy
- 🛠 **Maintenance Effort**: 70% reduction in manual intervention
- 💰 **Cost Efficiency**: Cloud-ready architecture for optimal resource usage

Decision-Making Enablement:

- 🎯 Real-time customer segment analysis
- 📊 Cross-platform performance insights
- 🔍 Automated anomaly detection
- 📈 Predictive analytics foundation
- 💡 Data-driven business recommendations

🚀 Week 3 Preparation

Tomorrow's Advanced Focus:

- **Advanced Pandas Techniques**: Memory optimization and performance
- **Real-time Stream Processing**: Kafka and streaming analytics
- **Machine Learning Integration**: ML pipeline development
- **Advanced Data Quality**: Statistical validation methods
- **Performance Optimization**: Large-scale data processing

Skills Building Path:

- Complex data transformations and aggregations
- Real-time event processing architectures
- Advanced analytics and statistical methods
- Performance profiling and optimization
- Enterprise data governance frameworks

🎉 **Congratulations! You've successfully built your first production-ready integrated data platform. This foundation will support all advanced data engineering concepts in the coming weeks!**