🌪️ Day 10: Apache Airflow - Mastering Workflow Orchestration for Data Engineers

📚 What You'll Learn Today (Concept-First Approach)

Primary Focus: Understanding workflow orchestration and why it's the heart of data engineering **Secondary Focus**: Hands-on implementation through Airflow Web UI and visual tools **Dataset for Context**: Customer Analytics Dataset from Kaggle for automated processing

🎯 Learning Philosophy for Day 10

"Understand the conductor before learning the orchestra"

We'll start with workflow concepts, explore Airflow's visual interface, understand DAG design patterns, and build production-ready automated pipelines.

🌟 The Orchestration Revolution: Why Airflow Matters

🤔 The Problem: Chaos in Data Pipeline Management

Scenario: You're managing a customer analytics pipeline with multiple dependent tasks...

```
Without Orchestration (Manual Chaos):
📅 Monday 6 AM: Run data extraction script
📅 Monday 6:15 AM: Check if extraction completed
📅 Monday 6:30 AM: Run data cleaning (if extraction worked)
📅 Monday 6:45 AM: Check cleaning status
📅 Monday 7:00 AM: Run analytics (if cleaning worked)
📅 Monday 7:15 AM: Generate reports (if analytics worked)
📅 Monday 7:30 AM: Send email alerts

❌ Problems:
- Manual intervention required at each step
- No automatic retry on failures
- Hard to track what succeeded/failed
- Impossible to scale across teams
- No historical execution tracking
```

With Airflow Orchestration:

✅ Define workflow once
✅ Automatic execution based on schedule
✅ Smart dependency management
✅ Automatic retries on failures
✅ Rich monitoring and alerting
✅ Historical tracking and debugging
✅ Scalable across entire organization

💡 The Airflow Solution: Workflows as Code

Think of Airflow like this:

- **Traditional Way**: Manually conducting an orchestra, cuing each musician
- **Airflow Way**: Written musical score that orchestras can follow automatically

🏗️ Understanding Airflow Architecture (Visual Approach)

🎨 The Airflow Mental Model

```
┌─────────────────────────────────────────────────────────┐
│                     APACHE AIRFLOW                      │
│                                                         │
│  ┌─────────────┐   ┌─────────────┐   ┌─────────────┐   │
│  │   Web UI    │   │ Scheduler   │   │  Executor   │   │
│  │ (Monitor &  │◄─►│ (Brain of   │◄─►│ (Task       │   │
│  │  Control)   │   │  Airflow)   │   │  Runner)    │   │
│  └─────────────┘   └─────────────┘   └─────────────┘   │
│         ▲                 ▲                 ▲           │
│         │                 │                 │           │
│  ┌─────────────┐   ┌─────────────┐   ┌─────────────┐   │
│  │  Metadata   │   │    DAGs     │   │   Workers   │   │
│  │  Database   │   │ (Workflow   │   │ (Execute    │   │
│  │ (State &    │   │ Definitions)│   │  Tasks)     │   │
│  │  History)   │   │             │   │             │   │
│  └─────────────┘   └─────────────┘   └─────────────┘   │
└─────────────────────────────────────────────────────────┘
```

🧠 Key Airflow Components

1. DAGs (Directed Acyclic Graphs)

- **Directed**: Tasks flow in specific directions

- **Acyclic**: No circular dependencies (no infinite loops)
- **Graph**: Visual representation of workflow

2. Tasks

- Individual units of work (extract data, run analysis, send email)
- Can be Python functions, bash commands, SQL queries, etc.

3. Operators

- Templates for creating tasks (PythonOperator, BashOperator, SqlOperator)

4. Scheduler

- Monitors DAGs and triggers tasks when dependencies are met

5. Executor

- Runs tasks on workers (local machine, cluster, cloud)

6. Web UI

- Visual interface for monitoring, debugging, and controlling workflows

🎯 Airflow Installation and Setup (UI-First Approach)

📱 Quick Start with Docker (Visual Learning)

Step 1: Download Airflow with Docker

Create project structure:

```
airflow-customer-analytics/
├── docker-compose.yml
├── dags/
│   └── customer_analytics_dag.py
├── data/
│   └── customer_data.csv
├── logs/
├── plugins/
└── scripts/
```

Step 2: Docker Compose for Airflow

Create `docker-compose.yml`:

yaml

```yaml
version: '3.8'

x-airflow-common:
  &airflow-common
  image: apache/airflow:2.7.1
  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'
  volumes:
    - ./dags:/opt/airflow/dags
    - ./logs:/opt/airflow/logs
    - ./plugins:/opt/airflow/plugins
    - ./data:/opt/airflow/data
  user: "${AIRFLOW_UID:-50000}:0"
  depends_on: &airflow-common-depends-on
    postgres:
      condition: service_healthy

services:
  postgres:
    image: postgres:13
    environment:
      POSTGRES_USER: airflow
      POSTGRES_PASSWORD: airflow
      POSTGRES_DB: airflow
    volumes:
      - postgres_db_volume:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "airflow"]
      interval: 5s
      retries: 5

  airflow-webserver:
    <<: *airflow-common
    command: webserver
    ports:
      - 8080:8080
    healthcheck:
```

```yaml
      test: ["CMD", "curl", "--fail", "http://localhost:8080/health"]
      interval: 10s
      timeout: 10s
      retries: 5

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

  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 -e "\033[1;31mERROR!!!: Too old Airflow version $${airflow_version}!\e[
          exit 1
        fi
        if [[ -z "${AIRFLOW_UID}" ]]; then
          echo -e "\033[1;33mWARNING!!!: AIRFLOW_UID not set!\e[0m"
          echo "Setting AIRFLOW_UID to 50000"
          export AIRFLOW_UID=50000
        fi
        one_meg=1048576
        mem_available=$$(($$(getconf _PHYS_PAGES) * $$(getconf PAGE_SIZE) / one_meg))
        cpus_available=$$(grep -cE 'cpu[0-9]+' /proc/stat)
        disk_available=$$(df / | tail -1 | awk '{print $$4}')
        warning_resources="false"
        if (( mem_available < 4000 )) ; then
          echo -e "\033[1;33mWARNING!!!: Not enough memory available for Docker.\e[0m"
          warning_resources="true"
```

```
        fi
        if (( cpus_available < 2 )); then
          echo -e "\033[1;33mWARNING!!!: Not enough CPUS available for Docker.\e[0m"
          warning_resources="true"
        fi
        if (( disk_available < one_meg )); then
          echo -e "\033[1;33mWARNING!!!: Not enough Disk space available for Docker.\e
          warning_resources="true"
        fi
        if [[ $${warning_resources} == "true" ]]; then
          echo -e "\033[1;33mWARNING!!!: You have not enough resources to run Airflow
          echo "Please follow the instructions to increase amount of resources availab
          echo "   https://airflow.apache.org/docs/apache-airflow/stable/howto/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_db_volume:
```

Step 3: Launch Airflow

bash

```bash
# Set Airflow user ID
echo -e "AIRFLOW_UID=$(id -u)" > .env

# Initialize Airflow
docker-compose up airflow-init

# Start Airflow services
docker-compose up -d

# Access Airflow Web UI
# http://localhost:8080
# Username: airflow
# Password: airflow
```

🎮 First Look at Airflow Web UI

Main Interface Sections:

1. **DAGs View**: List of all workflows
   - Status indicators (running, success, failed)
   - Last run information
   - DAG schedule and next run time
2. **Graph View**: Visual representation of workflow
   - Task dependencies as connected nodes
   - Task status color coding
   - Interactive task details
3. **Tree View**: Historical runs over time
   - Timeline of DAG executions
   - Task success/failure patterns
   - Easy identification of recurring issues
4. **Gantt Chart**: Task duration analysis
   - Performance bottlenecks identification
   - Resource utilization patterns
   - Optimization opportunities

🎯 Understanding DAGs: The Workflow Blueprint

🧩 DAG Fundamentals (Visual Learning)

What is a DAG?

```
Simple DAG Example (Customer Analytics Pipeline):

┌─────────────┐    ┌─────────────┐    ┌─────────────┐
│   Extract   │───▶│  Transform  │───▶│    Load     │
│    Data     │    │    Data     │    │  Database   │
└─────────────┘    └─────────────┘    └─────────────┘

       │                                      │
       │                                    │
       └──────────▶┌─────────────┐◀─────────┘
                   │  Validate   │
                   │    Data     │
                   └─────────────┘

                          │

                   ┌─────────────┐
                   │  Generate   │
                   │   Report    │
                   └─────────────┘
```

Key DAG Properties:

- **Directed**: Arrows show task execution order
- **Acyclic**: No circular dependencies
- **Connected**: Related tasks form a coherent workflow

📝 Creating Your First DAG (Hands-On)

Step 1: Download Sample Data

1. Visit: kaggle.com/datasets/imakash3011/customer-personality-analysis
2. Download `marketing_campaign.csv`
3. Place in `data/customer_data.csv`

Step 2: Basic DAG Structure

Create `dags/customer_analytics_dag.py`:

python

```python
"""
Customer Analytics DAG
======================

This DAG demonstrates a complete customer analytics workflow:
1. Extract customer data from CSV
2. Validate data quality
3. Transform data (RFM analysis)
4. Load to database
5. Generate insights report
6. Send email notification

Schedule: Daily at 6 AM
Owner: Data Engineering Team
"""

from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.operators.bash_operator import BashOperator
from airflow.operators.email_operator import EmailOperator
from airflow.operators.dummy_operator import DummyOperator

# Default arguments for all tasks
default_args = {
    'owner': 'data-engineering-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  # Don't run historical DAGs
}

# Define the DAG
dag = DAG(
    'customer_analytics_pipeline',
    default_args=default_args,
    description='Daily customer analytics processing',
    schedule_interval='0 6 * * *',  # Daily at 6 AM
    max_active_runs=1,  # Only one DAG run at a time
    tags=['customer', 'analytics', 'daily']
```

```python
)

# Task functions (we'll implement these)
def extract_customer_data(**context):
    """Extract customer data from CSV file"""
    print("🔄 Starting data extraction...")
    # Implementation here
    print("✅ Data extraction completed!")

def validate_data_quality(**context):
    """Validate data quality and completeness"""
    print("🔍 Starting data validation...")
    # Implementation here
    print("✅ Data validation completed!")

def transform_customer_data(**context):
    """Perform RFM analysis and customer segmentation"""
    print("⚙️ Starting data transformation...")
    # Implementation here
    print("✅ Data transformation completed!")

def load_to_database(**context):
    """Load processed data to PostgreSQL"""
    print("💾 Starting data loading...")
    # Implementation here
    print("✅ Data loading completed!")

def generate_insights_report(**context):
    """Generate customer insights report"""
    print("📊 Generating insights report...")
    # Implementation here
    print("✅ Report generation completed!")

# Define tasks
start_task = DummyOperator(
    task_id='start_pipeline',
    dag=dag
)

extract_task = PythonOperator(
    task_id='extract_customer_data',
    python_callable=extract_customer_data,
    dag=dag
)
```

```
validate_task = PythonOperator(
    task_id='validate_data_quality',
    python_callable=validate_data_quality,
    dag=dag
)

transform_task = PythonOperator(
    task_id='transform_customer_data',
    python_callable=transform_customer_data,
    dag=dag
)

load_task = PythonOperator(
    task_id='load_to_database',
    python_callable=load_to_database,
    dag=dag
)

report_task = PythonOperator(
    task_id='generate_insights_report',
    python_callable=generate_insights_report,
    dag=dag
)

# Health check task
health_check = BashOperator(
    task_id='health_check',
    bash_command='echo "Pipeline health check passed ✅"',
    dag=dag
)

# Email notification on success
success_email = EmailOperator(
    task_id='send_success_email',
    to=['data-team@company.com'],
    subject='Customer Analytics Pipeline - Success',
    html_content="""
    <h3>Customer Analytics Pipeline Completed Successfully</h3>
    <p>The daily customer analytics pipeline has completed successfully.</p>
    <p><strong>Execution Date:</strong> {{ ds }}</p>
    <p><strong>Total Runtime:</strong> {{ macros.timedelta(dag_run.end_date - dag_run.
    """,
    dag=dag
```

```
)

end_task = DummyOperator(
    task_id='end_pipeline',
    dag=dag
)

# Define task dependencies
start_task >> extract_task >> validate_task >> transform_task >> load_task >> report_t
```

🎯 Exploring DAG in Airflow UI

Step 1: Refresh DAGs

1. Open Airflow UI (http://localhost:8080)
2. Look for "customer_analytics_pipeline" in DAGs list
3. Toggle the DAG "ON" (unpause it)

Step 2: Explore Graph View

1. Click on the DAG name
2. Go to "Graph View" tab
3. Observe the visual workflow representation
4. Click on individual tasks to see details

Step 3: Trigger Manual Run

1. Click "Trigger DAG" button (play icon)
2. Watch tasks execute in real-time
3. Observe color changes:
   - **White**: Not started
   - **Yellow**: Running
   - **Green**: Success
   - **Red**: Failed

🛠️ Airflow Operators: Building Blocks of Workflows

🧩 Understanding Operators (Concept First)

Operators are task templates that define what to do:

```
┌─────────────────────────────────────────────────────────┐
│                    AIRFLOW OPERATORS                    │
│                                                         │
│  ┌─────────────┐  ┌─────────────┐  ┌─────────────┐  │
│  │   Python    │  │    Bash     │  │     SQL     │  │
│  │  Operator   │  │  Operator   │  │  Operator   │  │
│  │             │  │             │  │             │  │
│  │Execute Python│  │Run shell    │  │Execute SQL  │  │
│  │functions    │  │commands     │  │queries      │  │
│  └─────────────┘  └─────────────┘  └─────────────┘  │
│                                                         │
│  ┌─────────────┐  ┌─────────────┐  ┌─────────────┐  │
│  │   Email     │  │    File     │  │   Docker    │  │
│  │  Operator   │  │  Operator   │  │  Operator   │  │
│  │             │  │             │  │             │  │
│  │Send emails  │  │File transfers│  │Run containers│  │
│  │and alerts   │  │and checks   │  │and images   │  │
│  └─────────────┘  └─────────────┘  └─────────────┘  │
└─────────────────────────────────────────────────────────┘
```

🎯 Common Operators for Data Engineering

1. PythonOperator - Most versatile for data processing

python

```python
def process_customer_data(**context):
    import pandas as pd
    # Your data processing logic
    df = pd.read_csv('/opt/airflow/data/customer_data.csv')
    # Process the data
    return df.shape[0]  # Return number of records processed

python_task = PythonOperator(
    task_id='process_data',
    python_callable=process_customer_data,
    dag=dag
)
```

2. BashOperator - For shell commands and scripts

```python
bash_task = BashOperator(
    task_id='run_data_quality_check',
    bash_command='python /opt/airflow/scripts/data_quality.py',
    dag=dag
)
```

3. SqlOperator - For database operations

```python
from airflow.providers.postgres.operators.postgres import PostgresOperator

sql_task = PostgresOperator(
    task_id='create_customer_table',
    postgres_conn_id='postgres_default',
    sql="""
    CREATE TABLE IF NOT EXISTS customer_segments (
        customer_id VARCHAR(50),
        segment VARCHAR(50),
        rfm_score VARCHAR(10),
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );
    """,
    dag=dag
)
```

4. EmailOperator - For notifications

```python
email_task = EmailOperator(
    task_id='send_report',
    to=['team@company.com'],
    subject='Daily Customer Report',
    html_content='<h1>Report attached</h1>',
    dag=dag
)
```

🎮 Hands-On Operator Implementation

Let's implement the customer analytics DAG with real functionality:

python

```python
"""
Complete Customer Analytics DAG Implementation
"""

import pandas as pd
import numpy as np
from datetime import datetime, timedelta
from sqlalchemy import create_engine
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.operators.bash_operator import BashOperator
from airflow.providers.postgres.operators.postgres import PostgresOperator

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

dag = DAG(
    'customer_analytics_complete',
    default_args=default_args,
    description='Complete customer analytics pipeline',
    schedule_interval='0 6 * * *',
    catchup=False,
    tags=['customer', 'rfm', 'analytics']
)

def extract_and_validate_data(**context):
    """Extract customer data and perform basic validation"""
    print("🔄 Starting data extraction and validation...")

    # Load data
    df = pd.read_csv('/opt/airflow/data/customer_data.csv')

    # Basic validation
    validation_results = {
        'total_records': len(df),
        'missing_values': df.isnull().sum().sum(),
```

```
        'duplicate_records': df.duplicated().sum(),
        'date_range': f"{df['Dt_Customer'].min()} to {df['Dt_Customer'].max()}"
    }

    print(f"📊 Validation Results: {validation_results}")

    # Store validation results for downstream tasks
    context['task_instance'].xcom_push(key='validation_results', value=validation_resu

    # Save cleaned data
    df_clean = df.dropna().drop_duplicates()
    df_clean.to_csv('/opt/airflow/data/customer_data_clean.csv', index=False)

    print("✅ Data extraction and validation completed!")
    return validation_results

def perform_rfm_analysis(**context):
    """Perform RFM (Recency, Frequency, Monetary) analysis"""
    print("⚙️ Starting RFM analysis...")

    # Load cleaned data
    df = pd.read_csv('/opt/airflow/data/customer_data_clean.csv')

    # Convert date column
    df['Dt_Customer'] = pd.to_datetime(df['Dt_Customer'])

    # Calculate current date for recency
    current_date = df['Dt_Customer'].max() + pd.Timedelta(days=1)

    # Calculate RFM metrics
    rfm = df.groupby('ID').agg({
        'Dt_Customer': lambda x: (current_date - x.max()).days,  # Recency
        'NumTotalPurchases': 'sum',  # Frequency
        'MntTotal': 'sum'  # Monetary
    }).reset_index()

    rfm.columns = ['CustomerID', 'Recency', 'Frequency', 'Monetary']

    # Create RFM scores (quintiles)
    rfm['R_Score'] = pd.qcut(rfm['Recency'], 5, labels=[5,4,3,2,1])
    rfm['F_Score'] = pd.qcut(rfm['Frequency'].rank(method='first'), 5, labels=[1,2,3,4
    rfm['M_Score'] = pd.qcut(rfm['Monetary'], 5, labels=[1,2,3,4,5])

    # Combine scores
```

```python
    rfm['RFM_Score'] = rfm['R_Score'].astype(str) + rfm['F_Score'].astype(str) + rfm['

    # Customer segmentation
    def segment_customers(score):
        if score in ['555', '554', '544', '545', '454', '455', '445']:
            return 'Champions'
        elif score in ['543', '444', '435', '355', '354', '345', '344', '335']:
            return 'Loyal Customers'
        elif score in ['553', '551', '552', '541', '542', '533', '532', '531']:
            return 'Potential Loyalists'
        elif score in ['512', '511', '422', '421', '412', '411', '311']:
            return 'New Customers'
        elif score in ['155', '154', '144', '214', '215', '115', '114']:
            return 'At Risk'
        else:
            return 'Others'

    rfm['Segment'] = rfm['RFM_Score'].apply(segment_customers)

    # Save results
    rfm.to_csv('/opt/airflow/data/customer_rfm_analysis.csv', index=False)

    # Generate summary statistics
    segment_summary = rfm.groupby('Segment').agg({
        'CustomerID': 'count',
        'Recency': 'mean',
        'Frequency': 'mean',
        'Monetary': ['mean', 'sum']
    }).round(2)

    segment_summary.to_csv('/opt/airflow/data/segment_summary.csv')

    print(f"📊 RFM Analysis completed. Segments identified: {rfm['Segment'].value_coun

    # Store summary for reporting
    context['task_instance'].xcom_push(key='segment_summary', value=segment_summary.to

    return rfm.shape[0]

def generate_business_insights(**context):
    """Generate actionable business insights from RFM analysis"""
    print("📊 Generating business insights...")

    # Load RFM results
```

```python
    rfm = pd.read_csv('/opt/airflow/data/customer_rfm_analysis.csv')

    # Calculate key business metrics
    insights = {
        'total_customers': len(rfm),
        'total_revenue': rfm['Monetary'].sum(),
        'avg_customer_value': rfm['Monetary'].mean(),
        'champions_count': len(rfm[rfm['Segment'] == 'Champions']),
        'at_risk_count': len(rfm[rfm['Segment'] == 'At Risk']),
        'champions_revenue_share': rfm[rfm['Segment'] == 'Champions']['Monetary'].sum(
    }

    # Generate recommendations
    recommendations = []

    if insights['champions_revenue_share'] > 30:
        recommendations.append("High champion revenue share - focus on retention progr

    if insights['at_risk_count'] > insights['total_customers'] * 0.2:
        recommendations.append("High at-risk customers - implement win-back campaigns"

    # Create insights report
    report = {
        'execution_date': context['ds'],
        'insights': insights,
        'recommendations': recommendations
    }

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

    print("✅ Business insights generated successfully!")

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

    return insights

# Create PostgreSQL table
create_table_task = PostgresOperator(
    task_id='create_customer_segments_table',
    postgres_conn_id='postgres_default',
    sql="""
```

```
    CREATE TABLE IF NOT EXISTS customer_segments (
        customer_id VARCHAR(50) PRIMARY KEY,
        recency INTEGER,
        frequency INTEGER,
        monetary DECIMAL(10,2),
        rfm_score VARCHAR(10),
        segment VARCHAR(50),
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );

    CREATE TABLE IF NOT EXISTS segment_summary (
        segment VARCHAR(50) PRIMARY KEY,
        customer_count INTEGER,
        avg_recency DECIMAL(10,2),
        avg_frequency DECIMAL(10,2),
        avg_monetary DECIMAL(10,2),
        total_revenue DECIMAL(15,2),
        updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );
    """,
    dag=dag
)

def load_data_to_postgres(**context):
    """Load RFM analysis results to PostgreSQL"""
    print("💾 Loading data to PostgreSQL...")

    # Database connection
    engine = create_engine('postgresql://airflow:airflow@postgres:5432/airflow')

    # Load RFM data
    rfm = pd.read_csv('/opt/airflow/data/customer_rfm_analysis.csv')

    # Load to database
    rfm.to_sql('customer_segments', engine, if_exists='replace', index=False, method='

    # Load segment summary
    segment_summary = pd.read_csv('/opt/airflow/data/segment_summary.csv')
    segment_summary.to_sql('segment_summary', engine, if_exists='replace', index=False

    print(f"✅ Loaded {len(rfm)} customer records to PostgreSQL!")

    return len(rfm)
```

```
# Define all tasks
extract_validate_task = PythonOperator(
    task_id='extract_and_validate_data',
    python_callable=extract_and_validate_data,
    dag=dag
)

rfm_analysis_task = PythonOperator(
    task_id='perform_rfm_analysis',
    python_callable=perform_rfm_analysis,
    dag=dag
)

insights_task = PythonOperator(
    task_id='generate_business_insights',
    python_callable=generate_business_insights,
    dag=dag
)

load_postgres_task = PythonOperator(
    task_id='load_data_to_postgres',
    python_callable=load_data_to_postgres,
    dag=dag
)

# Data quality check
quality_check_task = BashOperator(
    task_id='data_quality_check',
    bash_command="""
    echo "Running data quality checks..."
    if [ -f /opt/airflow/data/customer_rfm_analysis.csv ]; then
        echo "✅ RFM analysis file exists"
        record_count=$(wc -l < /opt/airflow/data/customer_rfm_analysis.csv)
        echo "📊 Record count: $record_count"
        if [ $record_count -gt 1 ]; then
            echo "✅ Data quality check passed"
        else
            echo "❌ Data quality check failed - insufficient records"
            exit 1
        fi
    else
        echo "❌ Data quality check failed - RFM file not found"
        exit 1
    fi
```

```
    """,
    dag=dag
)

# Define task dependencies
extract_validate_task >> rfm_analysis_task >> insights_task >> create_table_task >> lo
```

🎛️ Task Dependencies and Scheduling Concepts

🕸️ Understanding Dependencies (Visual Approach)

Dependency Types:

```
1. Linear Dependencies (Sequential):
A ──► B ──► C ──► D

2. Parallel Dependencies (Fan-out):
     ┌─► B ──┐
A ──► │       ├─► D
     └─► C ──┘

3. Diamond Dependencies (Fan-out + Fan-in):
     ┌─► B ──┐
A ──► │       ├─► D ──► E
     └─► C ──┘

4. Complex Dependencies:
     ┌─► B ──┐
A ──► │       ├─► E ──► F
     └─► C ──► D ──┘
```

🎯 Implementing Dependencies in Code

Multiple Dependency Syntax Options:

```python
# Method 1: Using >> and << operators
task_a >> task_b >> task_c  # Linear

# Method 2: Using set_downstream/set_upstream
task_a.set_downstream([task_b, task_c])  # Fan-out
[task_b, task_c].set_downstream(task_d)  # Fan-in

# Method 3: Using lists
task_a >> [task_b, task_c] >> task_d  # Diamond pattern

# Method 4: Complex dependencies
task_a >> task_b >> task_d
task_a >> task_c >> task_d
task_d >> task_e
```

📅 Scheduling Concepts (Cron and Beyond)

Understanding Schedule Intervals:

```python
# Common scheduling patterns
dag = DAG(
    'my_dag',
    schedule_interval='@daily',     # Every day at midnight
    # schedule_interval='@hourly',    # Every hour
    # schedule_interval='0 6 * * *', # Every day at 6 AM
    # schedule_interval='0 6 * * 1', # Every Monday at 6 AM
    # schedule_interval=timedelta(hours=2), # Every 2 hours
    # schedule_interval=None,        # Manual trigger only
)
```

Cron Expression Breakdown:

```
0 6 * * *
│ │ │ │ │
│ │ │ │ └─── Day of week (0-7, both 0 and 7 are Sunday)
│ │ │ └───── Month (1-12)
│ │ └─────── Day of month (1-31)
│ └───────── Hour (0-23)
└─────────── Minute (0-59)
```

Visual Cron Examples:

```
'0 6 * * *'      = Daily at 6:00 AM
'0 */2 * * *'    = Every 2 hours
'0 6 * * 1'      = Every Monday at 6:00 AM
'0 6 1 * *'      = First day of every month at 6:00 AM
'0 6 1 1 *'      = January 1st at 6:00 AM (yearly)
```

🎨 Airflow Web UI Mastery

📊 UI Navigation and Monitoring

DAGs View Overview:

```
┌──────────────────────────────────────────────────────────┐
│ DAG Name            │ Status │ Last Run │ Next Run │ Actions│
├──────────────────────────────────────────────────────────┤
│ customer_analytics│   ✅    │ 06:00:00 │ Tomorrow │ 🔄 ⏸  │
│ data_validation    │   ❌    │ 05:30:00 │ 05:30:00 │ 🔄 ⏸  │
│ reporting_pipeline│   🟡    │ Running  │ Next Hour│ 🔄 ⏸  │
└──────────────────────────────────────────────────────────┘
```

Status Indicators:

- **Green (✅)**: All tasks successful
- **Red (❌)**: At least one task failed
- **Yellow (🟡)**: Currently running
- **Purple (🟣)**: Upstream failed
- **Gray**: Not yet started

🎯 Graph View Deep Dive

Interactive Graph Features:

1. **Task Status Colors**:
    - **White**: Not started
    - **Light Green**: Queued
    - **Yellow**: Running
    - **Dark Green**: Success
    - **Red**: Failed

- **Orange**: Retry
- **Purple**: Skipped

2. **Task Details Panel**:
    - Click any task to see:
        - Task Instance Details
        - Logs
        - XCom (cross-communication) data
        - Task Duration
        - Retry History
3. **Zoom and Pan**:
    - Mouse wheel to zoom
    - Click and drag to pan
    - Fit to screen button

🔍 Tree View for Historical Analysis

Tree View Benefits:

```
Task Timeline View:
                Today    Yesterday  2 Days Ago
extract_data     ✅         ✅          ❌
validate_data    ✅         ✅          ⏭️
transform_data   ✅         ✅          ⏭️
load_data        ✅         ✅          ⏭️
generate_report  🟡         ✅          ⏭️
```

Pattern Recognition:

- **Consistent failures**: Infrastructure issues
- **Intermittent failures**: Data quality issues
- **Cascading failures**: Dependency problems
- **Performance degradation**: Resource constraints

📈 Gantt Chart for Performance Analysis

Performance Insights:

```
Task Performance Analysis:
extract_data      |████|                   (2 min)
validate_data       |██|                   (1 min)
transform_data        |████████|           (4 min)
load_data               |████|             (2 min)
generate_report           |██████|         (3 min)

Total Pipeline Duration: 12 minutes
Critical Path: extract → validate → transform → load → report
Bottleneck: transform_data (4 minutes)
```

🔄 XCom: Task Communication

💬 Understanding XCom (Cross-Communication)

Concept: Tasks sharing data with each other

```python
def upstream_task(**context):
    """Task that produces data"""
    result = {"processed_records": 1000, "quality_score": 0.95}

    # Push data to XCom
    context['task_instance'].xcom_push(key='processing_results', value=result)
    return result

def downstream_task(**context):
    """Task that consumes data"""
    # Pull data from XCom
    results = context['task_instance'].xcom_pull(
        task_ids='upstream_task',
        key='processing_results'
    )

    print(f"Processing {results['processed_records']} records")
    print(f"Quality score: {results['quality_score']}")
```

🎯 XCom Best Practices

1. Keep Data Small:

```python
# ✅ Good: Small metadata
xcom_push(key='record_count', value=1000)

# ❌ Bad: Large datasets
# xcom_push(key='full_dataset', value=huge_dataframe)
```

2. Use for Coordination, Not Data Transfer:

```python
# ✅ Good: Status and metrics
return {
    'status': 'success',
    'records_processed': len(df),
    'file_path': '/tmp/processed_data.csv'
}

# ❌ Bad: Actual data transfer
# return df  # Don't pass DataFrames via XCom
```

3. Error Handling:

```python
def consume_xcom_data(**context):
    try:
        data = context['task_instance'].xcom_pull(task_ids='producer_task')
        if data is None:
            raise ValueError("No data received from upstream task")
        # Process data
    except Exception as e:
        print(f"Error processing XCom data: {e}")
        raise
```

🛡️ Error Handling and Retries

🔄 Retry Strategies

Configuring Retries:

```python
default_args = {
    'retries': 3,                             # Number of retries
    'retry_delay': timedelta(minutes=5),      # Wait between retries
    'retry_exponential_backoff': True,        # Exponential backoff
    'max_retry_delay': timedelta(hours=1),    # Maximum retry delay
}

# Task-specific retry configuration
risky_task = PythonOperator(
    task_id='risky_operation',
    python_callable=some_function,
    retries=5,  # Override default
    retry_delay=timedelta(minutes=10),
    dag=dag
)
```

🚨 Email Alerts and Notifications

Email Configuration:

```python
default_args = {
    'email': ['data-team@company.com', 'devops@company.com'],
    'email_on_failure': True,
    'email_on_retry': False,
    'email_on_success': False,  # Usually too noisy
}

# Custom email content
def task_fail_slack_alert(context):
    """Send custom Slack alert on task failure"""
    slack_msg = f"""
    :red_circle: Task Failed
    *Task*: {context.get('task_instance').task_id}
    *DAG*: {context.get('task_instance').dag_id}
    *Execution Time*: {context.get('ds')}
    *Log*: {context.get('task_instance').log_url}
    """
    # Send to Slack (implementation depends on your setup)

task_with_alerts = PythonOperator(
    task_id='important_task',
    python_callable=important_function,
    on_failure_callback=task_fail_slack_alert,
    dag=dag
)
```

🎯 Failure Handling Strategies

1. Task-Level Handling:

```python
def robust_data_processing(**context):
    try:
        # Main processing logic
        df = pd.read_csv('/path/to/data.csv')
        processed_df = transform_data(df)
        processed_df.to_csv('/path/to/output.csv')

    except FileNotFoundError:
        # Handle missing input gracefully
        print("Input file not found, skipping processing")
        return "skipped"

    except Exception as e:
        # Log error and re-raise for Airflow to handle
        print(f"Unexpected error in data processing: {e}")
        raise
```

2. Skip Tasks on Upstream Failure:

```python
from airflow.operators.dummy_operator import DummyOperator
from airflow.utils.trigger_rule import TriggerRule

cleanup_task = PythonOperator(
    task_id='cleanup',
    python_callable=cleanup_function,
    trigger_rule=TriggerRule.ALL_DONE,  # Run regardless of upstream success/failure
    dag=dag
)
```

3. Branching for Error Recovery:

python

```python
from airflow.operators.python_operator import BranchPythonOperator

def check_data_quality(**context):
    """Branch based on data quality results"""
    quality_score = check_quality()

    if quality_score > 0.9:
        return 'high_quality_processing'
    else:
        return 'data_cleaning_task'

branching_task = BranchPythonOperator(
    task_id='quality_check_branch',
    python_callable=check_data_quality,
    dag=dag
)

high_quality_task = PythonOperator(
    task_id='high_quality_processing',
    python_callable=standard_processing,
    dag=dag
)

cleaning_task = PythonOperator(
    task_id='data_cleaning_task',
    python_callable=intensive_cleaning,
    dag=dag
)

branching_task >> [high_quality_task, cleaning_task]
```

🌐 Airflow Connections and Variables

🔌 Managing Connections (UI Approach)

Setting Up Database Connection:

1. **Access Admin Menu**:
    - Go to Admin → Connections in Airflow UI
2. **Add New Connection**:
    - **Conn Id**: `postgres_customer_db`

- **Conn Type**: `Postgres`
- **Host**: `postgres`
- **Schema**: `customer_analytics`
- **Login**: `airflow`
- **Password**: `airflow`
- **Port**: `5432`

3. **Test Connection**:
   - Click "Test" button to verify connectivity

Using Connections in DAGs:

```python
from airflow.providers.postgres.hooks.postgres import PostgresHook

def load_data_with_connection(**context):
    """Load data using Airflow connection"""
    # Get connection
    postgres_hook = PostgresHook(postgres_conn_id='postgres_customer_db')

    # Execute SQL
    postgres_hook.run("""
        INSERT INTO customer_segments (customer_id, segment, rfm_score)
        VALUES (%s, %s, %s)
    """, parameters=('CUST001', 'Champion', '555'))

    print("✅ Data loaded using Airflow connection!")
```

🔧 Managing Variables (Configuration)

Setting Variables via UI:

1. **Go to Admin → Variables**
2. **Add Key-Value Pairs**:
   - `data_source_path`: `/opt/airflow/data/customer_data.csv`
   - `min_quality_threshold`: `0.95`
   - `notification_email`: `data-team@company.com`

Using Variables in DAGs:

```python
from airflow.models import Variable

def configurable_task(**context):
    """Task using Airflow Variables for configuration"""
    # Get variables
    data_path = Variable.get('data_source_path')
    quality_threshold = float(Variable.get('min_quality_threshold'))
    notification_email = Variable.get('notification_email')

    print(f"Processing data from: {data_path}")
    print(f"Quality threshold: {quality_threshold}")

    # Use in processing logic
    df = pd.read_csv(data_path)
    quality_score = calculate_quality(df)

    if quality_score < quality_threshold:
        send_alert(notification_email, f"Quality below threshold: {quality_score}")
```

🎯 Advanced DAG Patterns

🌟 Dynamic DAG Generation

Problem: Creating similar DAGs for different datasets or environments

Solution: Generate DAGs programmatically

python

```python
"""
Dynamic DAG Generation Example
Generate customer analytics DAGs for different regions
"""

from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta

# Configuration for different regions
REGIONS = ['north_america', 'europe', 'asia_pacific']

def create_region_dag(region_name):
    """Factory function to create DAG for specific region"""

    default_args = {
        'owner': 'data-team',
        'depends_on_past': False,
        'start_date': datetime(2024, 1, 1),
        'retries': 2,
        'retry_delay': timedelta(minutes=5)
    }

    dag = DAG(
        f'customer_analytics_{region_name}',
        default_args=default_args,
        description=f'Customer analytics for {region_name}',
        schedule_interval='0 6 * * *',
        catchup=False,
        tags=['customer', 'analytics', region_name]
    )

    def process_region_data(**context):
        print(f"Processing customer data for {region_name}")
        # Region-specific processing logic
        data_path = f'/opt/airflow/data/{region_name}_customers.csv'
        # Process data...

    process_task = PythonOperator(
        task_id=f'process_{region_name}_data',
        python_callable=process_region_data,
        dag=dag
    )
```

```python
    return dag

# Generate DAGs for each region
for region in REGIONS:
    globals()[f'customer_analytics_{region}'] = create_region_dag(region)
```

🔄 SubDAGs for Reusable Workflows

Concept: Encapsulate common workflow patterns

python

```
from airflow.operators.subdag_operator import SubDagOperator
from airflow.models import DAG

def create_data_quality_subdag(parent_dag_id, child_dag_id, schedule_interval, default_
    """Reusable data quality checking workflow"""

    subdag = DAG(
        f'{parent_dag_id}.{child_dag_id}',
        default_args=default_args,
        schedule_interval=schedule_interval,
    )

    # Data quality tasks
    completeness_check = PythonOperator(
        task_id='check_completeness',
        python_callable=check_data_completeness,
        dag=subdag
    )

    accuracy_check = PythonOperator(
        task_id='check_accuracy',
        python_callable=check_data_accuracy,
        dag=subdag
    )

    consistency_check = PythonOperator(
        task_id='check_consistency',
        python_callable=check_data_consistency,
        dag=subdag
    )

    # Set dependencies
    [completeness_check, accuracy_check, consistency_check]

    return subdag

# Use SubDAG in main DAG
main_dag = DAG(...)

quality_check_subdag = SubDagOperator(
    task_id='data_quality_checks',
    subdag=create_data_quality_subdag(
        parent_dag_id='customer_analytics_main',
```

```
            child_dag_id='data_quality_checks',
            schedule_interval=main_dag.schedule_interval,
            default_args=main_dag.default_args
        ),
        dag=main_dag
    )
```

⚡ TaskGroups for Better Organization

Modern Alternative to SubDAGs:

python

```python
from airflow.utils.task_group import TaskGroup

with DAG('customer_analytics_with_groups', ...) as dag:

    # Data ingestion group
    with TaskGroup('data_ingestion') as ingestion_group:
        extract_customers = PythonOperator(
            task_id='extract_customers',
            python_callable=extract_customer_data
        )

        extract_transactions = PythonOperator(
            task_id='extract_transactions',
            python_callable=extract_transaction_data
        )

        validate_data = PythonOperator(
            task_id='validate_data',
            python_callable=validate_extracted_data
        )

        [extract_customers, extract_transactions] >> validate_data

    # Processing group
    with TaskGroup('data_processing') as processing_group:
        clean_data = PythonOperator(
            task_id='clean_data',
            python_callable=clean_customer_data
        )

        rfm_analysis = PythonOperator(
            task_id='rfm_analysis',
            python_callable=perform_rfm_analysis
        )

        segment_customers = PythonOperator(
            task_id='segment_customers',
            python_callable=segment_customers_func
        )

        clean_data >> rfm_analysis >> segment_customers

    # Reporting group
```

```python
    with TaskGroup('reporting') as reporting_group:
        generate_dashboard = PythonOperator(
            task_id='generate_dashboard',
            python_callable=create_dashboard
        )

        send_email_report = EmailOperator(
            task_id='send_email_report',
            to=['stakeholders@company.com'],
            subject='Daily Customer Analytics Report',
            html_content='Dashboard updated successfully!'
        )

        generate_dashboard >> send_email_report

    # Define group dependencies
    ingestion_group >> processing_group >> reporting_group
```

📊 Monitoring and Logging

📈 Built-in Monitoring Features

DAG Performance Metrics:

- **Duration Trends**: Track pipeline execution time
- **Success Rate**: Percentage of successful runs
- **Task Distribution**: Which tasks take longest
- **Resource Usage**: CPU and memory consumption

Accessing Metrics in UI:

1. **Browse → DAG Runs**: Historical execution data
2. **Browse → Task Instances**: Individual task performance
3. **Browse → Jobs**: Scheduler and executor status

📝 Structured Logging

Best Practices for Logging:

python

```python
import logging
from airflow.utils.log.logging_mixin import LoggingMixin

class CustomerAnalyticsProcessor(LoggingMixin):
    """Class with proper logging for Airflow"""

    def process_customers(self, **context):
        """Process customer data with structured logging"""

        # Use Airflow's logging
        self.log.info("Starting customer data processing")

        try:
            # Log processing steps
            self.log.info("Loading customer data from CSV")
            df = pd.read_csv('/opt/airflow/data/customer_data.csv')

            self.log.info(f"Loaded {len(df)} customer records")

            # Process data
            processed_df = self.transform_data(df)

            self.log.info(f"Processed {len(processed_df)} customer records")
            self.log.info("Customer data processing completed successfully")

            return len(processed_df)

        except Exception as e:
            self.log.error(f"Error processing customer data: {str(e)}")
            self.log.error(f"Error type: {type(e).__name__}")
            raise

def airflow_task_with_logging(**context):
    """Airflow task function with proper logging"""
    processor = CustomerAnalyticsProcessor()
    return processor.process_customers(**context)
```

🚨 Custom Alerting

Slack Integration Example:

python

```python
from airflow.providers.slack.operators.slack_webhook import SlackWebhookOperator

def send_success_alert(**context):
    """Send custom success notification"""

    # Get task instance details
    ti = context['task_instance']
    dag_run = context['dag_run']

    # Prepare message
    message = f"""
    ✅ *Pipeline Success*

    *DAG*: {ti.dag_id}
    *Execution Date*: {context['ds']}
    *Duration*: {dag_run.end_date - dag_run.start_date}
    *Tasks Completed*: All tasks successful

    Dashboard: http://localhost:8080/dags/{ti.dag_id}/grid
    """

    slack_alert = SlackWebhookOperator(
        task_id='slack_success_alert',
        http_conn_id='slack_webhook',
        message=message,
        dag=dag
    )

    return slack_alert.execute(context=context)

# Add to DAG
success_alert_task = PythonOperator(
    task_id='send_success_alert',
    python_callable=send_success_alert,
    trigger_rule='all_success',
    dag=dag
)
```

🚀 Production Deployment Considerations

🏗️ Airflow Architecture for Production

Components in Production:

```
┌─────────────────────────────────────────────────────────┐
│                  PRODUCTION AIRFLOW                     │
│                                                         │
│  ┌─────────────┐    ┌─────────────┐    ┌─────────────┐  │
│  │ Load        │    │   Airflow   │    │  Worker     │  │
│  │ Balancer    │◄──►│ Webserver   │    │  Nodes      │  │
│  │ (nginx)     │    │ (Multiple)  │    │ (Celery)    │  │
│  └─────────────┘    └─────────────┘    └─────────────┘  │
│                            │                  ▲         │
│                            ▼                  │         │
│  ┌─────────────┐    ┌─────────────┐    ┌─────────────┐  │
│  │ Metadata    │    │   Airflow   │    │   Message   │  │
│  │ Database    │◄──►│ Scheduler   │◄──►│   Broker    │  │
│  │(PostgreSQL) │    │             │    │  (Redis)    │  │
│  └─────────────┘    └─────────────┘    └─────────────┘  │
└─────────────────────────────────────────────────────────┘
```

🔧 Configuration Management

Environment-Specific Settings:

python

```python
import os
from airflow import DAG

# Environment-aware configuration
ENVIRONMENT = os.getenv('AIRFLOW_ENV', 'development')

if ENVIRONMENT == 'production':
    DEFAULT_ARGS = {
        'retries': 3,
        'retry_delay': timedelta(minutes=10),
        'email_on_failure': True,
        'email': ['data-team@company.com', 'ops@company.com']
    }
    SCHEDULE_INTERVAL = '0 6 * * *'  # Daily at 6 AM

elif ENVIRONMENT == 'staging':
    DEFAULT_ARGS = {
        'retries': 2,
        'retry_delay': timedelta(minutes=5),
        'email_on_failure': True,
        'email': ['data-team@company.com']
    }
    SCHEDULE_INTERVAL = '0 8 * * *'  # Daily at 8 AM

else:  # development
    DEFAULT_ARGS = {
        'retries': 1,
        'retry_delay': timedelta(minutes=1),
        'email_on_failure': False
    }
    SCHEDULE_INTERVAL = None  # Manual trigger only

dag = DAG(
    'customer_analytics_env_aware',
    default_args=DEFAULT_ARGS,
    schedule_interval=SCHEDULE_INTERVAL,
    tags=[ENVIRONMENT, 'customer', 'analytics']
)
```

🛡️ Security Best Practices

1. **Connection Security**:

- Store sensitive credentials in Airflow Connections
- Use environment variables for secrets
- Enable encryption for metadata database

2. DAG Security:

```python
# Secure file access
def secure_file_processing(**context):
    """Process files with security checks"""

    file_path = context['params'].get('file_path')

    # Validate file path
    if not file_path.startswith('/opt/airflow/data/'):
        raise ValueError("Unauthorized file path access")

    # Check file permissions
    if not os.access(file_path, os.R_OK):
        raise PermissionError("Insufficient file permissions")

    # Process file securely
    process_file(file_path)
```

3. Resource Limits:

```python
from airflow.operators.python_operator import PythonOperator
from airflow.configuration import conf

# Set resource limits
task_with_limits = PythonOperator(
    task_id='resource_limited_task',
    python_callable=memory_intensive_function,
    pool='memory_intensive_pool',  # Use resource pool
    dag=dag
)
```

📚 Essential Resources for Day 10

📖 Official Documentation

- **Apache Airflow Documentation**: https://airflow.apache.org/docs/
- **Airflow Best Practices**: https://airflow.apache.org/docs/apache-airflow/stable/best-practices.html
- **Airflow Operators**: https://airflow.apache.org/docs/apache-airflow/stable/operators-and-hooks