DataOps Lab Guide - Step by Step Implementation

Hands-On Workshop for DBT + Airflow + SQL Server Project

Lab Overview

This comprehensive lab guide will walk you through implementing DataOps principles in your data pipeline project. You'll learn by doing, implementing CI/CD, testing, monitoring, and automation step by step.

Duration: 8-10 hours (can be split across multiple sessions)

Prerequisites:

- Completed basic setup from HANDS_ON_LAB_GUIDE.md
- Docker and Docker Compose installed
- Git installed
- Basic understanding of SQL, Python, and command line

What You'll Build:

- Complete CI/CD pipeline with GitHub Actions
- Automated testing framework
- Monitoring and alerting system
- Multi-environment deployment strategy
- Data quality validation framework

Table of Contents

Module 1: Version Control & Git Setup

↑ Back to Table of Contents

Objective

Set up proper version control with Git, implement branching strategy, and configure pre-commit hooks.

Step 1.1: Initialize Git Repository (5 minutes)

Task: Initialize your project as a Git repository if not already done.

```
# Navigate to your project directory
cd dbt_airflow_project

# Initialize Git (if not already initialized)
git init

# Rename branch
git branch -m master main

# Check status
git status
```

Expected Output:

```
Initialized empty Git repository in /path/to/dbt_airflow_project/.git/
```

Step 1.2: Create .gitignore File (5 minutes)

Task: Create a comprehensive .gitignore file to exclude unnecessary files.

Create file: `.gitignore`

```
# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
env/
venv/
ENV/
*.egg-info/

# DBT
dbt/target/
dbt/dbt_packages/
dbt/logs/
dbt/.user.yml
dbt/profiles.yml

# Airflow
airflow/logs/
airflow/airflow.db
airflow/airflow-webserver.pid
airflow/unittests.cfg
```

```
# Docker
*.log

# Environment variables
.env
.env.local

# IDE
.vscode/
.idea/
*.swp
*.swo

# OS
.DS_Store
Thumbs.db

# Backups
backup/
*.bak
```

Verify:

```
git status
# Should not show target/, logs/, or other ignored files
```

Step 1.3: Create Initial Commit (5 minutes)

Task: Make your first commit with all project files.

```
# Add all files
git add .

# Create initial commit
git commit -m "feat: initial project setup with DBT, Airflow, and SQL Server"

# View commit history
git log --oneline
```

Expected Output:

```
abc1234 feat: initial project setup with DBT, Airflow, and SQL Server
```

Step 1.4: Set Up Branch Strategy (10 minutes)

Task: Create development and feature branches following GitFlow.

```
# Create and switch to develop branch
git checkout -b develop

# Create a feature branch for DataOps implementation
git checkout -b feature/dataops-setup

# View all branches
git branch -a
```

Expected Output:

```
* feature/dataops-setup
  develop
  main
```

Understanding Branch Strategy:

- `main` - Production-ready code
- `develop` - Integration branch for features
- `feature/*` - New features or enhancements
- `bugfix/*` - Bug fixes
- `hotfix/*` - Emergency production fixes

Step 1.5: Install Pre-commit Hooks (20 minutes)

Task: Set up pre-commit hooks to enforce code quality.

Step 1: Install pre-commit

```
pip install pre-commit
```

Step 2: Create pre-commit configuration

Create file: `.pre-commit-config.yaml`

```
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.4.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
      - id: check-added-large-files
        args: ['--maxkb=1000']
      - id: check-merge-conflict
```

```
      - id: check-json
      - id: pretty-format-json
        args: ['--autofix']

  - repo: https://github.com/psf/black
    rev: 23.7.0
    hooks:
      - id: black
        language_version: python3
        files: \.py$
        args: ['--line-length=120']

  - repo: https://github.com/pycqa/flake8
    rev: 6.1.0
    hooks:
      - id: flake8
        args: ['--max-line-length=120', '--ignore=E203,W503']
        files: \.py$
```

Step 3: Install the hooks

```
pre-commit install
```

Step 4: Test the hooks

```
# Run on all files
# pre-commit run --all-files
pre-commit run --files dbt\models\gold\gld_sales_summary.sql
dbt\models\gold\schema.yml
```

Expected Output:

```
Trim Trailing Whitespace.....................................................Passed
Fix End of Files.............................................................Passed
Check Yaml...................................................................Passed
Check for added large files..................................................Passed
Check for merge conflicts....................................................Passed
black........................................................................Passed
flake8.......................................................................Passed
```

Step 5: Commit the pre-commit configuration

```
git add .pre-commit-config.yaml
git commit -m "chore: add pre-commit hooks for code quality"
```

✅ Module 1 Checkpoint

Verify your progress:

- ☐ Git repository initialized
- ☐ .gitignore file created and working
- ☐ Initial commit made
- ☐ Branch strategy implemented
- ☐ Pre-commit hooks installed and tested

Test your understanding:

1. What branch should you create for a new feature?
2. Why do we use .gitignore?
3. What happens when you try to commit code that fails pre-commit checks?

Module 2: CI/CD Pipeline Implementation

↑ Back to Table of Contents

Objective

Create a complete CI/CD pipeline using GitHub Actions to automate testing and deployment.

Step 2.1: Create GitHub Repository (10 minutes)

Task: Push your local repository to GitHub.

Step 1: Create a new repository on GitHub

- Go to https://github.com/new
- Name: `dbt-airflow-dataops`
- Description: "DataOps implementation with DBT, Airflow, and SQL Server"
- Keep it Private (or Public if you prefer)
- Don't initialize with README (we already have one)

Step 2: Connect local repo to GitHub

```
# Add remote origin
git remote add origin https://github.com/YOUR_USERNAME/dbt-airflow-dataops.git

# Push all branches
git push -u origin main
git push -u origin develop
git push -u origin feature/dataops-setup
```

Verify: Visit your GitHub repository and confirm all files are there.

Step 2.2: Create GitHub Actions Workflow Directory (5 minutes)

Task: Set up the directory structure for GitHub Actions.

```
# Create workflows directory
mkdir -p .github/workflows

# Verify structure
ls -la .github/workflows/
```

Step 2.3: Create DBT CI Workflow (30 minutes)

Task: Create a GitHub Actions workflow to test DBT models.

Create file: `.github/workflows/dbt-ci.yml`

```
name: DBT CI Pipeline

on:
  push:
    branches: [main, develop, 'feature/**']
  pull_request:
    branches: [main, develop]

env:
  DBT_PROFILES_DIR: ./dbt

jobs:
  lint-sql:
    name: Lint SQL Files
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install SQLFluff
        run: |
          pip install sqlfluff sqlfluff-templater-dbt

      - name: Lint SQL files
        run: |
          cd dbt
          sqlfluff lint models/ --dialect tsql || true
        continue-on-error: true

  lint-python:
```

```
    name: Lint Python Files
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install flake8
        run: pip install flake8

      - name: Lint Python files
        run: |
          flake8 airflow/dags/ --max-line-length=120 --count --statistics || true
        continue-on-error: true

  dbt-compile:
    name: Compile DBT Models
    runs-on: ubuntu-latest
    needs: [lint-sql, lint-python]

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install DBT
        run: |
          pip install dbt-sqlserver

      - name: Install DBT dependencies
        run: |
          cd dbt
          dbt deps

      - name: Compile DBT models
        run: |
          cd dbt
          dbt compile --profiles-dir . || true
        continue-on-error: true

      - name: Upload compiled artifacts
        uses: actions/upload-artifact@v3
        with:
          name: dbt-compiled
          path: dbt/target/
```

```
          retention-days: 7

  generate-docs:
    name: Generate DBT Documentation
    runs-on: ubuntu-latest
    needs: dbt-compile
    if: github.ref == 'refs/heads/main' || github.ref == 'refs/heads/develop'

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install DBT
        run: pip install dbt-sqlserver

      - name: Install dependencies
        run: |
          cd dbt
          dbt deps

      - name: Generate documentation
        run: |
          cd dbt
          dbt docs generate --profiles-dir . || true
        continue-on-error: true

      - name: Upload documentation
        uses: actions/upload-artifact@v3
        with:
          name: dbt-docs
          path: |
            dbt/target/catalog.json
            dbt/target/manifest.json
            dbt/target/index.html
          retention-days: 30
```

Step 2: Commit and push the workflow

```
git add .github/workflows/dbt-ci.yml
git commit -m "ci: add DBT CI pipeline with linting and compilation"
git push origin feature/dataops-setup
```

Step 3: Verify the workflow runs

- Go to your GitHub repository

- Click on "Actions" tab
- You should see the workflow running
- Wait for it to complete

Expected Result: All jobs should complete (some may show warnings, that's okay for now)

Step 2.4: Create Python Linting Workflow (15 minutes)

Task: Add a separate workflow for Python code quality.

Create file: `.github/workflows/python-quality.yml`

```
name: Python Code Quality

on:
  push:
    branches: [main, develop, 'feature/**']
    paths:
      - '**.py'
      - 'airflow/**'
  pull_request:
    branches: [main, develop]
    paths:
      - '**.py'
      - 'airflow/**'

jobs:
  code-quality:
    name: Check Python Code Quality
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install dependencies
        run: |
          pip install flake8 black pylint

      - name: Run Black formatter check
        run: |
          black --check --line-length=120 airflow/dags/ || true
        continue-on-error: true

      - name: Run Flake8
        run: |
          flake8 airflow/dags/ --max-line-length=120 --statistics || true
```

```
        continue-on-error: true

      - name: Run Pylint
        run: |
          pylint airflow/dags/ --max-line-length=120 || true
        continue-on-error: true
```

Commit and push:

```
git add .github/workflows/python-quality.yml
git commit -m "ci: add Python code quality checks"
git push origin feature/dataops-setup
```

Step 2.5: Add Workflow Status Badges (10 minutes)

Task: Add status badges to your README to show CI/CD status.

Edit: `README.md` (add at the top)

```
# DBT + Airflow + SQL Server DataOps Project

![DBT CI](https://github.com/YOUR_USERNAME/dbt-airflow-
dataops/workflows/DBT%20CI%20Pipeline/badge.svg)
![Python Quality](https://github.com/YOUR_USERNAME/dbt-airflow-
dataops/workflows/Python%20Code%20Quality/badge.svg)

[Rest of your README content...]
```

Commit:

```
git add README.md
git commit -m "docs: add CI/CD status badges to README"
git push origin feature/dataops-setup
```

Step 2.6: Create Pull Request Workflow (20 minutes)

Task: Set up automated checks for pull requests.

Create file: `.github/workflows/pr-checks.yml`

```
name: Pull Request Checks

on:
  pull_request:
    branches: [main, develop]
```

```
jobs:
  pr-validation:
    name: Validate Pull Request
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3
        with:
          fetch-depth: 0

      - name: Check PR title format
        run: |
          PR_TITLE="${{ github.event.pull_request.title }}"
          if [[ ! "$PR_TITLE" =~ ^(feat|fix|docs|style|refactor|test|chore)(\
(.+\))?: ]]; then
            echo "❌ PR title must follow conventional commits format"
            echo "Examples: feat: add new model, fix: correct date calculation"
            exit 1
          fi

      - name: Check for large files
        run: |
          git diff --name-only origin/${{ github.base_ref }}...HEAD | while read
file; do
            if [ -f "$file" ]; then
              size=$(wc -c < "$file")
              if [ $size -gt 1000000 ]; then
                echo "❌ File $file is larger than 1MB"
                exit 1
              fi
            fi
          done

      - name: Check for merge conflicts
        run: |
          if git diff --name-only origin/${{ github.base_ref }}...HEAD | xargs
grep -l "<<<<<<< HEAD" 2>/dev/null; then
            echo "❌ Merge conflicts detected"
            exit 1
          fi

      - name: Validate DBT models changed
        run: |
          echo "📊 DBT models changed in this PR:"
          git diff --name-only origin/${{ github.base_ref }}...HEAD | grep
"dbt/models/" || echo "No DBT models changed"

  size-check:
    name: Check PR Size
    runs-on: ubuntu-latest

    steps:
```

```
      - name: Checkout code
        uses: actions/checkout@v3
        with:
          fetch-depth: 0

      - name: Check PR size
        run: |
          CHANGED_FILES=$(git diff --name-only origin/${{ github.base_ref
}}...HEAD | wc -l)
          CHANGED_LINES=$(git diff origin/${{ github.base_ref }}...HEAD | wc -l)

          echo "📝 Files changed: $CHANGED_FILES"
          echo "📝 Lines changed: $CHANGED_LINES"

          if [ $CHANGED_FILES -gt 50 ]; then
            echo "⚠️  Warning: This PR changes more than 50 files. Consider
splitting it."
          fi

          if [ $CHANGED_LINES -gt 1000 ]; then
            echo "⚠️  Warning: This PR changes more than 1000 lines. Consider
splitting it."
          fi
```

Commit:

```
git add .github/workflows/pr-checks.yml
git commit -m "ci: add pull request validation checks"
git push origin feature/dataops-setup
```

✅ Module 2 Checkpoint

Verify your progress:

- ☐ GitHub repository created and connected
- ☐ DBT CI workflow created and running
- ☐ Python quality workflow created
- ☐ Status badges added to README
- ☐ PR validation workflow created

Test your understanding:

1. What triggers the DBT CI workflow?
2. Why do we use `continue-on-error: true` in some steps?
3. What does the PR validation workflow check for?

Troubleshooting:

- If workflows fail, check the Actions tab for error messages
- Ensure all file paths are correct

- Verify Python and DBT versions match your local setup

Module 3: Automated Testing Framework

↑ Back to Table of Contents

Objective

Implement comprehensive automated testing for your DBT models and data pipelines.

Step 3.1: Add DBT Schema Tests (30 minutes)

Task: Add schema tests to your existing DBT models.

Edit: `dbt/models/bronze/schema.yml`

```
version: 2

models:
  - name: brnz_customers
    description: "Bronze layer customer data from AdventureWorks"
    columns:
      - name: CustomerID
        description: "Primary key for customer"
        tests:
          - unique
          - not_null

      - name: FirstName
        description: "Customer first name"

      - name: LastName
        description: "Customer last name"

      - name: EmailPromotion
        description: "Email promotion preference"

      - name: StoreID
        description: "Store identifier"

      - name: TerritoryID
        description: "Sales territory"

      - name: last_modified_date
        description: "Last modified timestamp"
        tests:
          - not_null

  - name: brnz_sales_orders
    description: "Bronze layer sales order data"
    columns:
```

```
      - name: sales_order_id
        description: "Primary key"
        tests:
          - not_null

      - name: order_date
        description: "Order date"
        tests:
          - not_null

      - name: due_date
        description: "Due date"

      - name: ship_date
        description: "Ship date"

      - name: status
        description: "Order status"

      - name: online_order_flag
        description: "Online order flag"

      - name: sales_order_number
        description: "Sales order number"

      - name: purchase_order_number
        description: "Purchase order number"

      - name: customer_id
        description: "Foreign key to customer"
        tests:
          - not_null
          - relationships:
              to: ref('brnz_customers')
              field: CustomerID

      - name: sales_person_id
        description: "Sales person ID"

      - name: territory_id
        description: "Territory ID"

      - name: order_detail_id
        description: "Order detail ID"

      - name: product_id
        description: "Product ID"

      - name: order_qty
        description: "Order quantity"

      - name: unit_price
        description: "Unit price"
```

```
      - name: unit_price_discount
        description: "Unit price discount"

      - name: line_total
        description: "Line total"
```

Test locally:

```
cd dbt
dbt test --select brnz_customers brnz_sales_orders
```

Expected Output:

```
Completed successfully
Done. PASS=7 WARN=0 ERROR=0 SKIP=0 TOTAL=7
```

Step 3.2: Create Custom Data Quality Tests (30 minutes)

Task: Create custom SQL tests for business logic validation.

Create file: `dbt/tests/generic/test_positive_values.sql`

```
{% test positive_values(model, column_name) %}

SELECT *
FROM {{ model }}
WHERE {{ column_name }} < 0

{% endtest %}
```

Create file: `dbt/tests/generic/test_no_future_dates.sql`

```
{% test no_future_dates(model, column_name) %}

SELECT *
FROM {{ model }}
WHERE {{ column_name }} > GETDATE()

{% endtest %}
```

Create file: `dbt/tests/generic/test_valid_email.sql`

```

```

```
{% test valid_email(model, column_name) %}

SELECT *
FROM {{ model }}
WHERE {{ column_name }} IS NOT NULL
  AND {{ column_name }} NOT LIKE '%_@__%.__%'

{% endtest %}
```

Apply custom tests to models:

Edit: `dbt/models/bronze/schema.yml` (add to existing)

```
  - name: brnz_sales_orders
    columns:
      - name: TotalDue
        tests:
          - positive_values

      - name: OrderDate
        tests:
          - no_future_dates
```

Test:

```
cd dbt
dbt test --select brnz_sales_orders
```

Step 3.3: Add Source Freshness Tests (20 minutes)

Task: Configure source freshness monitoring.

Create file: `dbt/models/sources.yml`

```
version: 2

sources:
  - name: raw
    description: "AdventureWorks 2014 source database"
    database: AdventureWorks2014
    schema: Sales

    # Default freshness for all tables
    freshness:
      warn_after: {count: 24, period: hour}
      error_after: {count: 48, period: hour}
```

```
    loaded_at_field: ModifiedDate

    tables:
      - name: Customer
        description: "Customer master data"
        identifier: Customer
        freshness:
          warn_after: {count: 12, period: hour}
          error_after: {count: 24, period: hour}
        columns:
          - name: CustomerID
            description: "Primary key"
            tests:
              - unique
              - not_null

          - name: ModifiedDate
            description: "Last modification timestamp"

      - name: SalesOrderHeader
        description: "Sales order header data"
        identifier: SalesOrderHeader
        freshness:
          warn_after: {count: 6, period: hour}
          error_after: {count: 12, period: hour}
        columns:
          - name: SalesOrderID
            tests:
              - unique
              - not_null

          - name: CustomerID
            tests:
              - not_null
```

Test source freshness:

```
cd dbt
dbt source freshness
```

Expected Output:

```
02:36:41  1 of 2 START freshness of raw.Customer
..................................... [RUN]
02:36:41  2 of 2 START freshness of raw.SalesOrderHeader
..............................  [RUN]
02:36:41  1 of 2 ERROR STALE freshness of raw.Customer
.............................. [ERROR STALE in 0.08s]
```

```
02:36:41  2 of 2 ERROR STALE freshness of raw.SalesOrderHeader
.......................... [ERROR STALE in 0.10s]
```

Step 3.4: Install and Configure DBT Expectations (30 minutes)

Task: Add the dbt-expectations package for advanced testing.

Edit: `dbt/packages.yml`

```
packages:
  - package: calogica/dbt_expectations
    version: 0.10.1

  - package: dbt-labs/dbt_utils
    version: 1.1.1
```

Install packages:

```
cd dbt
dbt deps
```

Add advanced tests:

Edit: `dbt/models/bronze/schema.yml`

```
version: 2

models:
  - name: brnz_customers
    description: "Bronze layer customer data from AdventureWorks"
    columns:
      - name: CustomerID
        description: "Primary key for customer"
        tests:
          - unique
          - not_null
          - dbt_expectations.expect_column_values_to_be_of_type:
              column_type: int

      - name: FirstName
        description: "Customer first name (NULL for store customers)"

      - name: LastName
        description: "Customer last name (NULL for store customers)"

      - name: EmailPromotion
        description: "Email promotion preference"
```

```
        tests:
          - accepted_values:
              values: [0, 1, 2]

      - name: StoreID
        description: "Store identifier"

      - name: TerritoryID
        description: "Sales territory"

      - name: last_modified_date
        description: "Last modified timestamp"
        tests:
          - not_null

  - name: brnz_sales_orders
    description: "Bronze layer sales order data"
    columns:
      - name: sales_order_id
        description: "Primary key"
        tests:
          - not_null
          - dbt_expectations.expect_column_values_to_be_of_type:
              column_type: int

      - name: order_date
        description: "Order date"
        tests:
          - not_null
          - no_future_dates

      - name: due_date
        description: "Due date"

      - name: ship_date
        description: "Ship date"

      - name: status
        description: "Order status"
        tests:
          - accepted_values:
              values: [1, 2, 3, 4, 5, 6]

      - name: online_order_flag
        description: "Online order flag"
        tests:
          - dbt_expectations.expect_column_values_to_be_in_set:
              value_set: [0, 1]

      - name: sales_order_number
        description: "Sales order number"
        tests:
          - not_null
```

```
      - name: purchase_order_number
        description: "Purchase order number"

      - name: customer_id
        description: "Foreign key to customer"
        tests:
          - not_null
          - relationships:
              to: ref('brnz_customers')
              field: CustomerID

      - name: sales_person_id
        description: "Sales person ID"

      - name: territory_id
        description: "Territory ID"

      - name: order_detail_id
        description: "Order detail ID"

      - name: product_id
        description: "Product ID"
        tests:
          - not_null

      - name: order_qty
        description: "Order quantity"
        tests:
          - positive_values

      - name: unit_price
        description: "Unit price"
        tests:
          - positive_values

      - name: unit_price_discount
        description: "Unit price discount"

      - name: line_total
        description: "Line total"
        tests:
          - positive_values
```

Test:

```
cd dbt
dbt test --select brnz_customers brnz_sales_orders
```

Step 3.5: Create Data Quality Test Suite (30 minutes)

Task: Create comprehensive data quality tests.

Create file: dbt/tests/data_quality/test_no_duplicate_orders.sql

```
-- Test: No duplicate order detail IDs should exist
SELECT
    order_detail_id,
    COUNT(*) as duplicate_count
FROM {{ ref('brnz_sales_orders') }}
GROUP BY order_detail_id
HAVING COUNT(*) > 1
```

Create file: dbt/tests/data_quality/test_order_customer_consistency.sql

```
-- Test: All orders must have valid customers
SELECT
    o.sales_order_id,
    o.customer_id
FROM {{ ref('brnz_sales_orders') }} o
LEFT JOIN {{ ref('brnz_customers') }} c
    ON o.customer_id = c.CustomerID
WHERE c.CustomerID IS NULL
```

Create file: dbt/tests/data_quality/test_positive_revenue.sql

```
-- Test: All revenue values must be positive
SELECT
    sales_order_id,
    line_total
FROM {{ ref('brnz_sales_orders') }}
WHERE line_total < 0
```

Run all data quality tests:

```
cd dbt
dbt test --select path:tests/data_quality
```

✅ Module 3 Checkpoint

Verify your progress:

- ☐ Schema tests added to all models
- ☐ Custom generic tests created
- ☐ Source freshness tests configured

- [ ] dbt-expectations package installed
- [ ] Data quality test suite created

Test your understanding:

1. What's the difference between schema tests and data tests?
2. When should you use source freshness tests?
3. How do generic tests differ from singular tests?

Commit your work:

```
git add dbt/models/ dbt/tests/ dbt/packages.yml
git commit -m "test: add comprehensive testing framework with schema, custom, and
data quality tests"
git push origin feature/dataops-setup
```

Module 4: Monitoring & Observability

Objective

Implement monitoring and observability for your data pipelines.

Step 4.1: Add Logging and Metrics Collection (30 minutes)

Task: Enhance your DAGs with structured logging.

Create file: `airflow/dags/utils/logging_utils.py`

```
import logging
from datetime import datetime


class DataOpsLogger:
    """Logger for DataOps pipeline events and metrics"""

    def __init__(self, pipeline_name, component_name):
        self.pipeline_name = pipeline_name
        self.component_name = component_name
        self.logger = logging.getLogger(f"{pipeline_name}.{component_name}")

    def log_event(self, event_type, message, level="info"):
        """Log a pipeline event"""
        log_message = f"[{self.pipeline_name}] [{self.component_name}]
{event_type}: {message}"

        if level == "info":
            self.logger.info(log_message)
        elif level == "warning":
            self.logger.warning(log_message)
```

```
        elif level == "error":
            self.logger.error(log_message)
        else:
            self.logger.debug(log_message)

        return {
            "timestamp": datetime.now().isoformat(),
            "pipeline": self.pipeline_name,
            "component": self.component_name,
            "event_type": event_type,
            "message": message,
            "level": level,
        }

    def log_metric(self, metric_name, value, unit=None):
        """Log a pipeline metric"""
        unit_str = f" {unit}" if unit else ""
        message = f"Metric: {metric_name} = {value}{unit_str}"
        self.logger.info(f"[{self.pipeline_name}] [{self.component_name}]
{message}")

        return {
            "timestamp": datetime.now().isoformat(),
            "pipeline": self.pipeline_name,
            "component": self.component_name,
            "metric_name": metric_name,
            "value": value,
            "unit": unit,
        }


def setup_logger(name, log_file=None, level=logging.INFO):
    """Setup a logger with console and optional file handler"""
    logger = logging.getLogger(name)
    logger.setLevel(level)

    # Console handler
    console_handler = logging.StreamHandler()
    console_handler.setLevel(level)
    formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %
(message)s")
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)

    # File handler (optional)
    if log_file:
        file_handler = logging.FileHandler(log_file)
        file_handler.setLevel(level)
        file_handler.setFormatter(formatter)
        logger.addHandler(file_handler)

    return logger
```

```
def log_task_execution(task_id, status, duration=None, error=None):
    """Log task execution details"""
    logger = logging.getLogger("airflow.task")
    log_data = {
        "task_id": task_id,
        "status": status,
        "timestamp": datetime.now().isoformat(),
        "duration": duration,
        "error": str(error) if error else None,
    }

    if status == "success":
        logger.info(f"Task completed: {log_data}")
    elif status == "failed":
        logger.error(f"Task failed: {log_data}")
    else:
        logger.warning(f"Task status: {log_data}")

    return log_data
```

Step 4.2: Create Alerting Configuration (30 minutes)

Task: Set up alerting for pipeline failures.

Create file: `airflow/dags/utils/alerting.py`

```
import requests
import json
from datetime import datetime


class AlertManager:
    def __init__(self, webhook_url=None):
        # Get webhook URL from environment variable or parameter
        import os

        self.webhook_url = webhook_url or os.getenv("SLACK_WEBHOOK_URL", "")

    def send_slack_alert(self, title, message, severity="info"):
        color_map = {"info": "#36a64f", "warning": "#ff9900", "error": "#ff0000",
"critical": "#8b0000"}
        payload = {
            "attachments": [
                {
                    "color": color_map.get(severity, "#36a64f"),
                    "title": title,
                    "text": message,
                    "footer": "DataOps Monitoring",
                    "ts": int(datetime.now().timestamp()),
                }
```

```
            ]
        }

        # For testing without Slack webhook
        print("=== ALERT TRIGGERED ===")
        print(f"Severity: {severity}")
        print(f"Title: {title}")
        print(f"Message: {message}")
        print(f"Payload: {json.dumps(payload, indent=2)}")
        print("======================")

        try:
            response = requests.post(
                self.webhook_url, data=json.dumps(payload), headers={"Content-
Type": "application/json"}, timeout=10
            )
            print(f"Slack API Response: Status={response.status_code}, Body=
{response.text}")
            if response.status_code == 200:
                print("Alert successfully sent to Slack!")
                return True
            else:
                print(f"Slack returned error: {response.status_code} -
{response.text}")
                return False
        except Exception as e:
            print(f"Failed to send alert: {e}")
            return False

    def alert_pipeline_failure(self, dag_id, task_id, error_message):
        title = f"Pipeline Failure: {dag_id}"
        message = f"Task {task_id} failed\n\nError: {error_message}"
        return self.send_slack_alert(title, message, "error")

    def alert_test_failure(self, test_name, failure_count):
        title = "Data Quality Alert"
        message = f"Test {test_name} failed\n\nFailures: {failure_count}"
        return self.send_slack_alert(title, message, "warning")

    def alert_slow_pipeline(self, dag_id, execution_time, threshold):
        title = f"Slow Pipeline: {dag_id}"
        message = f"Execution time: {execution_time:.2f}s (threshold:
{threshold}s)"
        return self.send_slack_alert(title, message, "warning")
```

Add logging and alerting to your monitoring DAG:

Create: `airflow/dags/dbt_bronze_layer_test_alert_dag.py` (add)

```
from datetime import datetime, timedelta
from airflow import DAG
```

```
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from utils.logging_utils import DataOpsLogger
from utils.alerting import AlertManager

default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

dag = DAG(
    "dbt_bronze_test_alert_layer",
    default_args=default_args,
    description="Run dbt bronze layer models",
    schedule_interval=timedelta(minutes=30),
    start_date=datetime(2024, 1, 1),
    catchup=False,
    tags=["dbt", "bronze", "sqlserver"],
)


def log_pipeline_start(**context):
    """Log the start of the pipeline"""
    logger = DataOpsLogger("dbt_bronze_pipeline", "pipeline_start")
    logger.log_event("pipeline_start", "Starting DBT bronze layer execution")
    return {"start_time": datetime.now().isoformat()}


def log_pipeline_complete(**context):
    """Log the completion of the pipeline"""
    logger = DataOpsLogger("dbt_bronze_pipeline", "pipeline_complete")
    ti = context["ti"]
    start_data = ti.xcom_pull(task_ids="log_start")

    if start_data:
        start_time = datetime.fromisoformat(start_data["start_time"])
        execution_time = (datetime.now() - start_time).total_seconds()
        logger.log_metric("execution_time_seconds", execution_time, "seconds")
        logger.log_event("pipeline_complete", f"Completed in
{execution_time:.2f}s")
    else:
        logger.log_event("pipeline_complete", "Pipeline completed successfully")


def test_alert_failure(**context):
    """Intentionally fail to test alert system - REMOVE AFTER TESTING"""
    alert_manager = AlertManager()
    alert_manager.alert_pipeline_failure(
        dag_id="dbt_bronze_layer",
        task_id="test_alert_failure",
```

```
            error_message="This is a test failure to verify alert system is working",
        )
        raise Exception("TEST FAILURE: This task intentionally fails to test alerts")


    # Log pipeline start
    log_start = PythonOperator(
        task_id="log_start",
        python_callable=log_pipeline_start,
        dag=dag,
    )

    # Run bronze models only
    dbt_run_bronze = BashOperator(
        task_id="dbt_run_bronze",
        bash_command="docker exec dbt_airflow_project-dbt-1 dbt run --select bronze",
        dag=dag,
    )

    # Test bronze models
    dbt_test_bronze = BashOperator(
        task_id="dbt_test_bronze",
        bash_command="docker exec dbt_airflow_project-dbt-1 dbt test --select bronze",
        dag=dag,
    )

    # Log pipeline completion
    log_complete = PythonOperator(
        task_id="log_complete",
        python_callable=log_pipeline_complete,
        dag=dag,
    )

    test_failure = PythonOperator(
        task_id="test_alert_failure",
        python_callable=test_alert_failure,
        retries=0,  # No retries for test task
        dag=dag,
    )

    # Set task dependencies
    # Test failure runs independently so it doesn't block the pipeline
    log_start >> [test_failure, dbt_run_bronze]
    dbt_run_bronze >> dbt_test_bronze >> log_complete
```

✅ Module 4 Checkpoint

Verify your progress:

- ☐ Structured logging implemented
- ☐ Alerting system configured

- [ ] Metrics collection added

Test your work:

```
# Test the monitoring DAG
docker exec dbt_airflow_project-airflow-scheduler-1 airflow dags test
dbt_bronze_layer_test_alert
```

Commit:

```
git add airflow/dags/
git commit -m "feat: add monitoring, logging, and alerting infrastructure"
git push origin feature/dataops-setup
```

Module 5: Data Quality Framework

Objective

Implement a comprehensive data quality framework using Great Expectations.

Step 5.1: Install Great Expectations (15 minutes)

Task: Set up Great Expectations in your project.

Step 1: Add to requirements

Edit: `dbt/requirements.txt` (add)

```
great-expectations==0.18.8
sqlalchemy==1.4.48
pyodbc==4.0.39
```

Step 2: Install in container

```
docker exec dbt_airflow_project-dbt-1 pip install great-expectations sqlalchemy
pyodbc
```

Step 3: Initialize Great Expectations

```
docker exec -it dbt_airflow_project-dbt-1 bash
cd /usr/app
great_expectations init
```

Follow prompts:

- Would you like to proceed? [Y/n]: Y
- Would you like to configure a Datasource? [Y/n]: n (we'll do this manually)

Step 5.2: Configure Great Expectations (30 minutes)

Task: Configure GE to work with SQL Server.

Create file: `great_expectations/great_expectations.yml`

```
config_version: 3.0
config_variables_file_path: uncommitted/config_variables.yml

datasources:
  adventureworks_datasource:
    class_name: Datasource
    execution_engine:
      class_name: SqlAlchemyExecutionEngine
      connection_string:
mssql+pyodbc://sa:YourStrong@Passw0rd@sqlserver:1433/AdventureWorks2014?
driver=ODBC+Driver+17+for+SQL+Server

    data_connectors:
      default_runtime_data_connector:
        class_name: RuntimeDataConnector
        batch_identifiers:
          - default_identifier_name

      default_inferred_data_connector:
        class_name: InferredAssetSqlDataConnector
        include_schema_name: true

stores:
  expectations_store:
    class_name: ExpectationsStore
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: expectations/

  validations_store:
    class_name: ValidationsStore
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: uncommitted/validations/

  evaluation_parameter_store:
    class_name: EvaluationParameterStore

  checkpoint_store:
    class_name: CheckpointStore
    store_backend:
```

```
      class_name: TupleFilesystemStoreBackend
      base_directory: checkpoints/

expectations_store_name: expectations_store
validations_store_name: validations_store
evaluation_parameter_store_name: evaluation_parameter_store
checkpoint_store_name: checkpoint_store

data_docs_sites:
  local_site:
    class_name: SiteBuilder
    store_backend:
      class_name: TupleFilesystemStoreBackend
      base_directory: uncommitted/data_docs/local_site/
    site_index_builder:
      class_name: DefaultSiteIndexBuilder
```

Step 5.3: Create Expectation Suites (30 minutes)

Task: Create expectation suites for your data.

Create file: `great_expectations/expectations/customer_suite.json`

```
{
  "expectation_suite_name": "customer_quality_suite",
  "ge_cloud_id": null,
  "expectations": [
    {
      "expectation_type": "expect_table_row_count_to_be_between",
      "kwargs": {
        "min_value": 100,
        "max_value": 100000
      },
      "meta": {}
    },
    {
      "expectation_type": "expect_column_to_exist",
      "kwargs": {
        "column": "CustomerID"
      },
      "meta": {}
    },
    {
      "expectation_type": "expect_column_values_to_be_unique",
      "kwargs": {
        "column": "CustomerID"
      },
      "meta": {}
    },
    {
      "expectation_type": "expect_column_values_to_not_be_null",
```

```
      "kwargs": {
        "column": "CustomerID"
      },
      "meta": {}
    },
    {
      "expectation_type": "expect_column_values_to_not_be_null",
      "kwargs": {
        "column": "AccountNumber"
      },
      "meta": {}
    }
  ],
  "meta": {
    "great_expectations_version": "0.18.8"
  }
}
```

Create file: great_expectations/expectations/sales_order_suite.json

```
{
  "expectation_suite_name": "sales_order_quality_suite",
  "ge_cloud_id": null,
  "expectations": [
    {
      "expectation_type": "expect_column_values_to_be_unique",
      "kwargs": {
        "column": "SalesOrderID"
      },
      "meta": {}
    },
    {
      "expectation_type": "expect_column_values_to_not_be_null",
      "kwargs": {
        "column": "SalesOrderID"
      },
      "meta": {}
    },
    {
      "expectation_type": "expect_column_values_to_be_between",
      "kwargs": {
        "column": "TotalDue",
        "min_value": 0,
        "max_value": 1000000
      },
      "meta": {}
    },
    {
      "expectation_type": "expect_column_values_to_not_be_null",
      "kwargs": {
        "column": "OrderDate"
```

```
      },
      "meta": {}
    }
  ],
  "meta": {
    "great_expectations_version": "0.18.8"
  }
}
```

Step 5.4: Create Checkpoints (20 minutes)

Task: Create checkpoints to run validations.

Create file: `great_expectations/checkpoints/customer_checkpoint.yml`

```
name: customer_checkpoint
config_version: 1.0
class_name: SimpleCheckpoint
run_name_template: "%Y%m%d-%H%M%S-customer-validation"

validations:
  - batch_request:
      datasource_name: adventureworks_datasource
      data_connector_name: default_runtime_data_connector
      data_asset_name: Sales.Customer
    expectation_suite_name: customer_quality_suite
```

Create file: `great_expectations/checkpoints/sales_order_checkpoint.yml`

```
name: sales_order_checkpoint
config_version: 1.0
class_name: SimpleCheckpoint
run_name_template: "%Y%m%d-%H%M%S-sales-order-validation"

validations:
  - batch_request:
      datasource_name: adventureworks_datasource
      data_connector_name: default_runtime_data_connector
      data_asset_name: Sales.SalesOrderHeader
    expectation_suite_name: sales_order_quality_suite
```

Step 5.5: Integrate with Airflow (30 minutes)

Task: Create an Airflow DAG to run Great Expectations validations.

Create file: `airflow/dags/data_quality_dag.py`

```
```

```python
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.operators.bash import BashOperator
import great_expectations as ge
from great_expectations.data_context import DataContext

default_args = {
    'owner': 'dataops',
    'depends_on_past': False,
    'email_on_failure': True,
    'email': ['data-team@company.com'],
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(
    'data_quality_validation',
    default_args=default_args,
    description='Run Great Expectations data quality validations',
    schedule_interval='0 */6 * * *',  # Every 6 hours
    start_date=datetime(2024, 1, 1),
    catchup=False,
    tags=['data-quality', 'great-expectations'],
)

def run_ge_checkpoint(checkpoint_name, **context):
    """Run a Great Expectations checkpoint"""

    # Initialize Data Context
    context_root_dir = '/opt/airflow/great_expectations'
    data_context = DataContext(context_root_dir=context_root_dir)

    # Run checkpoint
    print(f"Running checkpoint: {checkpoint_name}")
    result = data_context.run_checkpoint(checkpoint_name=checkpoint_name)

    # Check results
    if not result["success"]:
        failed_validations = []
        for run_result in result.run_results.values():
            if not run_result["success"]:
                failed_validations.append(run_result)

        error_msg = f"Data quality validation failed for {checkpoint_name}"
        print(f"❌ {error_msg}")
        print(f"Failed validations: {len(failed_validations)}")
        raise Exception(error_msg)

    print(f"✅ Checkpoint {checkpoint_name} passed successfully")
    return result

# Tasks
```

```
validate_customers = PythonOperator(
    task_id='validate_customer_data',
    python_callable=run_ge_checkpoint,
    op_kwargs={'checkpoint_name': 'customer_checkpoint'},
    dag=dag,
)

validate_orders = PythonOperator(
    task_id='validate_sales_order_data',
    python_callable=run_ge_checkpoint,
    op_kwargs={'checkpoint_name': 'sales_order_checkpoint'},
    dag=dag,
)

generate_docs = BashOperator(
    task_id='generate_data_docs',
    bash_command='cd /opt/airflow/great_expectations && great_expectations docs
build',
    dag=dag,
)

# Dependencies
[validate_customers, validate_orders] >> generate_docs
```

✅ Module 5 Checkpoint

Verify your progress:

- ☐ Great Expectations installed
- ☐ Datasource configured
- ☐ Expectation suites created
- ☐ Checkpoints configured
- ☐ Airflow integration complete

Test:

```
# Test GE checkpoint
docker exec dbt_airflow_project-dbt-1 great_expectations checkpoint run
customer_checkpoint

# Test Airflow DAG
docker exec dbt_airflow_project-airflow-scheduler-1 airflow dags test
data_quality_validation
```

Commit:

```
git add great_expectations/ airflow/dags/data_quality_dag.py
git commit -m "feat: implement Great Expectations data quality framework"
git push origin feature/dataops-setup
```

Module 6: Deployment Automation

Objective

Automate deployment processes with scripts and multi-environment support.

Step 6.1: Create Deployment Script (30 minutes)

Task: Create a deployment script for different environments.

Create file: `scripts/deploy.sh`

```
#!/bin/bash

set -e

# Colors for output
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
NC='\033[0m' # No Color

ENVIRONMENT=$1
TARGET=$2

if [ -z "$ENVIRONMENT" ] || [ -z "$TARGET" ]; then
    echo -e "${RED}Usage: ./deploy.sh <environment> <target>${NC}"
    echo "Example: ./deploy.sh staging staging"
    exit 1
fi

echo -e "${GREEN}🚀 Starting deployment to $ENVIRONMENT environment...${NC}"

# 1. Pre-deployment checks
echo -e "${YELLOW}📋 Running pre-deployment checks...${NC}"
cd dbt

# Install dependencies
echo "Installing DBT dependencies..."
dbt deps --target $TARGET

# Compile models
echo "Compiling DBT models..."
dbt compile --target $TARGET

if [ $? -ne 0 ]; then
    echo -e "${RED}❌ Compilation failed${NC}"
    exit 1
fi
```

```
# 2. Run tests on current state
echo -e "${YELLOW}🧪 Running tests...${NC}"
dbt test --target $TARGET

if [ $? -ne 0 ]; then
    echo -e "${RED}❌ Tests failed${NC}"
    exit 1
fi

# 3. Backup current state
echo -e "${YELLOW}💾 Creating backup...${NC}"
timestamp=$(date +%Y%m%d_%H%M%S)
backup_dir="../backups/${ENVIRONMENT}_${timestamp}"
mkdir -p $backup_dir
cp -r target/ $backup_dir/

echo "Backup created at: $backup_dir"

# 4. Deploy models
echo -e "${YELLOW}🔨 Deploying models...${NC}"
dbt run --target $TARGET

if [ $? -ne 0 ]; then
    echo -e "${RED}❌ Deployment failed${NC}"
    exit 1
fi

# 5. Run post-deployment tests
echo -e "${YELLOW}✅ Running post-deployment tests...${NC}"
dbt test --target $TARGET

if [ $? -ne 0 ]; then
    echo -e "${RED}❌ Post-deployment tests failed${NC}"
    exit 1
fi

# 6. Generate documentation
echo -e "${YELLOW}📚 Generating documentation...${NC}"
dbt docs generate --target $TARGET

# 7. Success message
echo -e "${GREEN}✨ Deployment to $ENVIRONMENT completed successfully!${NC}"
echo ""
echo "Summary:"
echo "  Environment: $ENVIRONMENT"
echo "  Target: $TARGET"
echo "  Time: $(date)"
echo "  Backup: $backup_dir"
```

Make executable:

```
chmod +x scripts/deploy.sh
```

Step 6.2: Configure Multiple Environments (30 minutes)

Task: Set up configuration for dev, staging, and production environments.

Edit: `dbt/profiles.yml` (update with all environments)

```
adventureworks:
  target: dev
  outputs:
    dev:
      type: sqlserver
      driver: 'ODBC Driver 17 for SQL Server'
      server: localhost
      port: 1433
      database: AdventureWorks2014
      schema: dbo
      user: sa
      password: "YourStrong@Passw0rd"
      threads: 4
      trust_cert: true

    ci:
      type: sqlserver
      driver: 'ODBC Driver 17 for SQL Server'
      server: localhost
      port: 1433
      database: AdventureWorks2014_CI
      schema: dbo
      user: sa
      password: "{{ env_var('DBT_CI_PASSWORD', 'YourStrong@Passw0rd') }}"
      threads: 2
      trust_cert: true

    staging:
      type: sqlserver
      driver: 'ODBC Driver 17 for SQL Server'
      server: "{{ env_var('STAGING_SQL_SERVER', 'localhost') }}"
      port: 1433
      database: AdventureWorks2014_Staging
      schema: dbo
      user: "{{ env_var('STAGING_SQL_USER', 'sa') }}"
      password: "{{ env_var('STAGING_SQL_PASSWORD', 'YourStrong@Passw0rd') }}"
      threads: 8
      trust_cert: true

    prod:
      type: sqlserver
      driver: 'ODBC Driver 17 for SQL Server'
      server: "{{ env_var('PROD_SQL_SERVER', 'localhost') }}"
      port: 1433
```

```
        database: AdventureWorks2014_Prod
        schema: dbo
        user: "{{ env_var('PROD_SQL_USER', 'sa') }}"
        password: "{{ env_var('PROD_SQL_PASSWORD') }}"
        threads: 16
        trust_cert: true
```

Create environment files:

Create file: `.env.dev`

```
# Development Environment
DBT_TARGET=dev
SQL_SERVER=localhost
SQL_USER=sa
SQL_PASSWORD=YourStrong@Passw0rd
AIRFLOW_ENV=development
```

Create file: `.env.staging`

```
# Staging Environment
DBT_TARGET=staging
STAGING_SQL_SERVER=staging-server.company.com
STAGING_SQL_USER=staging_user
STAGING_SQL_PASSWORD=staging_password
AIRFLOW_ENV=staging
```

Create file: `.env.prod`

```
# Production Environment
DBT_TARGET=prod
PROD_SQL_SERVER=prod-server.company.com
PROD_SQL_USER=prod_user
PROD_SQL_PASSWORD=CHANGE_ME
AIRFLOW_ENV=production
```

Update .gitignore:

```
echo ".env.*" >> .gitignore
echo "!.env.example" >> .gitignore
```

Step 6.3: Create Rollback Script (20 minutes)

Task: Create a script to rollback deployments if needed.

Create file: scripts/rollback.sh

```
#!/bin/bash

set -e

RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
NC='\033[0m'

ENVIRONMENT=$1
BACKUP_DIR=$2

if [ -z "$ENVIRONMENT" ] || [ -z "$BACKUP_DIR" ]; then
    echo -e "${RED}Usage: ./rollback.sh <environment> <backup_directory>${NC}"
    echo "Example: ./rollback.sh staging backups/staging_20240101_120000"
    exit 1
fi

if [ ! -d "$BACKUP_DIR" ]; then
    echo -e "${RED}❌ Backup directory not found: $BACKUP_DIR${NC}"
    exit 1
fi

echo -e "${YELLOW}⚠️  WARNING: This will rollback to a previous state${NC}"
echo "Environment: $ENVIRONMENT"
echo "Backup: $BACKUP_DIR"
echo ""
read -p "Are you sure you want to proceed? (yes/no): " confirm

if [ "$confirm" != "yes" ]; then
    echo "Rollback cancelled"
    exit 0
fi

echo -e "${GREEN}🔄 Starting rollback...${NC}"

# Restore backup
echo "Restoring from backup..."
cd dbt
rm -rf target/
cp -r ../$BACKUP_DIR/target/ ./

# Re-run models from backup state
echo "Re-deploying previous state..."
dbt run --target $ENVIRONMENT

if [ $? -ne 0 ]; then
    echo -e "${RED}❌ Rollback failed${NC}"
    exit 1
fi
```

```
# Run tests
echo "Running tests..."
dbt test --target $ENVIRONMENT

echo -e "${GREEN}✅ Rollback completed successfully${NC}"
```

Make executable:

```
chmod +x scripts/rollback.sh
```

Step 6.4: Create Smoke Tests (30 minutes)

Task: Create smoke tests to verify deployments.

Create file: `scripts/smoke_tests.py`

```
#!/usr/bin/env python3
"""
Smoke tests for post-deployment validation
"""

import sys
import pyodbc
import argparse
from datetime import datetime

class SmokeTests:
    def __init__(self, server, database, user, password):
        self.conn_string = (
            f"DRIVER={{ODBC Driver 17 for SQL Server}};"
            f"SERVER={server};"
            f"DATABASE={database};"
            f"UID={user};"
            f"PWD={password};"
            f"TrustServerCertificate=yes;"
        )
        self.connection = None

    def connect(self):
        """Establish database connection"""
        try:
            self.connection = pyodbc.connect(self.conn_string)
            print("✅ Database connection successful")
            return True
        except Exception as e:
            print(f"❌ Database connection failed: {e}")
            return False
```

```
    def test_table_exists(self, schema, table):
        """Test if a table exists"""
        query = f"""
        SELECT COUNT(*) as cnt
        FROM INFORMATION_SCHEMA.TABLES
        WHERE TABLE_SCHEMA = '{schema}'
        AND TABLE_NAME = '{table}'
        """
        cursor = self.connection.cursor()
        cursor.execute(query)
        result = cursor.fetchone()

        if result[0] > 0:
            print(f"✅ Table {schema}.{table} exists")
            return True
        else:
            print(f"❌ Table {schema}.{table} not found")
            return False

    def test_table_has_data(self, schema, table, min_rows=1):
        """Test if a table has minimum number of rows"""
        query = f"SELECT COUNT(*) as cnt FROM {schema}.{table}"
        cursor = self.connection.cursor()
        cursor.execute(query)
        result = cursor.fetchone()
        row_count = result[0]

        if row_count >= min_rows:
            print(f"✅ Table {schema}.{table} has {row_count} rows (min:
{min_rows})")
            return True
        else:
            print(f"❌ Table {schema}.{table} has {row_count} rows (expected min:
{min_rows})")
            return False

    def test_no_nulls_in_column(self, schema, table, column):
        """Test that a column has no null values"""
        query = f"""
        SELECT COUNT(*) as null_count
        FROM {schema}.{table}
        WHERE {column} IS NULL
        """
        cursor = self.connection.cursor()
        cursor.execute(query)
        result = cursor.fetchone()
        null_count = result[0]

        if null_count == 0:
            print(f"✅ Column {schema}.{table}.{column} has no nulls")
            return True
        else:
            print(f"❌ Column {schema}.{table}.{column} has {null_count} null
values")
```

```
            return False

    def test_data_freshness(self, schema, table, date_column, max_age_hours=24):
        """Test that data is fresh"""
        query = f"""
        SELECT MAX({date_column}) as latest_date
        FROM {schema}.{table}
        """
        cursor = self.connection.cursor()
        cursor.execute(query)
        result = cursor.fetchone()
        latest_date = result[0]

        if latest_date:
            age_hours = (datetime.now() - latest_date).total_seconds() / 3600
            if age_hours <= max_age_hours:
                print(f"✅ Data in {schema}.{table} is fresh (age:
{age_hours:.1f}h)")
                return True
            else:
                print(f"❌ Data in {schema}.{table} is stale (age:
{age_hours:.1f}h, max: {max_age_hours}h)")
                return False
        else:
            print(f"❌ No data found in {schema}.{table}")
            return False

    def run_all_tests(self):
        """Run all smoke tests"""
        print("\n🧪 Running smoke tests...\n")

        tests_passed = 0
        tests_failed = 0

        # Test 1: Check bronze tables exist
        if self.test_table_exists('dbo', 'brnz_customers'):
            tests_passed += 1
        else:
            tests_failed += 1

        # Test 2: Check bronze tables have data
        if self.test_table_has_data('dbo', 'brnz_customers', min_rows=100):
            tests_passed += 1
        else:
            tests_failed += 1

        # Test 3: Check for nulls in primary key
        if self.test_no_nulls_in_column('dbo', 'brnz_customers', 'CustomerID'):
            tests_passed += 1
        else:
            tests_failed += 1

        # Test 4: Check data freshness
        if self.test_data_freshness('dbo', 'brnz_customers', 'ModifiedDate',
```

```
max_age_hours=168):
            tests_passed += 1
        else:
            tests_failed += 1

        # Summary
        print(f"\n📊 Test Summary:")
        print(f"   Passed: {tests_passed}")
        print(f"   Failed: {tests_failed}")
        print(f"   Total: {tests_passed + tests_failed}")

        return tests_failed == 0

    def close(self):
        """Close database connection"""
        if self.connection:
            self.connection.close()

def main():
    parser = argparse.ArgumentParser(description='Run smoke tests')
    parser.add_argument('--server', default='localhost', help='SQL Server
hostname')
    parser.add_argument('--database', default='AdventureWorks2014', help='Database
name')
    parser.add_argument('--user', default='sa', help='Username')
    parser.add_argument('--password', default='YourStrong@Passw0rd',
help='Password')

    args = parser.parse_args()

    # Run tests
    smoke_tests = SmokeTests(args.server, args.database, args.user, args.password)

    if not smoke_tests.connect():
        sys.exit(1)

    success = smoke_tests.run_all_tests()
    smoke_tests.close()

    if success:
        print("\n✅ All smoke tests passed!")
        sys.exit(0)
    else:
        print("\n❌ Some smoke tests failed!")
        sys.exit(1)

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

Make executable:

```
chmod +x scripts/smoke_tests.py
```

Test it:

```
python scripts/smoke_tests.py --server localhost --database AdventureWorks2014
```

Step 6.5: Update CI/CD for Deployment (30 minutes)

Task: Add deployment jobs to GitHub Actions.

Create file: `.github/workflows/deploy.yml`

```yaml
name: Deploy to Environments

on:
  push:
    branches:
      - develop
      - main
  workflow_dispatch:
    inputs:
      environment:
        description: 'Environment to deploy to'
        required: true
        type: choice
        options:
          - staging
          - production

jobs:
  deploy-staging:
    name: Deploy to Staging
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/develop' || (github.event_name ==
'workflow_dispatch' && github.event.inputs.environment == 'staging')
    environment: staging

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install dependencies
        run: |
          pip install dbt-sqlserver pyodbc
```

```
      - name: Run deployment
        env:
          STAGING_SQL_SERVER: ${{ secrets.STAGING_SQL_SERVER }}
          STAGING_SQL_USER: ${{ secrets.STAGING_SQL_USER }}
          STAGING_SQL_PASSWORD: ${{ secrets.STAGING_SQL_PASSWORD }}
        run: |
          chmod +x scripts/deploy.sh
          ./scripts/deploy.sh staging staging

      - name: Run smoke tests
        run: |
          pip install pyodbc
          python scripts/smoke_tests.py \
            --server ${{ secrets.STAGING_SQL_SERVER }} \
            --database AdventureWorks2014_Staging \
            --user ${{ secrets.STAGING_SQL_USER }} \
            --password ${{ secrets.STAGING_SQL_PASSWORD }}

      - name: Upload artifacts
        uses: actions/upload-artifact@v3
        with:
          name: staging-deployment
          path: |
            dbt/target/
            backups/

  deploy-production:
    name: Deploy to Production
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main' || (github.event_name ==
'workflow_dispatch' && github.event.inputs.environment == 'production')
    environment: production
    needs: []

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install dependencies
        run: |
          pip install dbt-sqlserver pyodbc

      - name: Run deployment
        env:
          PROD_SQL_SERVER: ${{ secrets.PROD_SQL_SERVER }}
          PROD_SQL_USER: ${{ secrets.PROD_SQL_USER }}
          PROD_SQL_PASSWORD: ${{ secrets.PROD_SQL_PASSWORD }}
        run: |
          chmod +x scripts/deploy.sh
```

```
            ./scripts/deploy.sh production prod

        - name: Run smoke tests
          run: |
            pip install pyodbc
            python scripts/smoke_tests.py \
              --server ${{ secrets.PROD_SQL_SERVER }} \
              --database AdventureWorks2014_Prod \
              --user ${{ secrets.PROD_SQL_USER }} \
              --password ${{ secrets.PROD_SQL_PASSWORD }}

        - name: Notify team
          if: always()
          run: |
            echo "Deployment to production completed"
            # Add Slack notification here
```

✅ Module 6 Checkpoint

Verify your progress:

- ☐ Deployment script created
- ☐ Multiple environments configured
- ☐ Rollback script created
- ☐ Smoke tests implemented
- ☐ CI/CD deployment workflow added

Test:

```
# Test deployment script
./scripts/deploy.sh dev dev

# Test smoke tests
python scripts/smoke_tests.py
```

Commit:

```
git add scripts/ .github/workflows/deploy.yml dbt/profiles.yml
git commit -m "feat: add deployment automation with multi-environment support"
git push origin feature/dataops-setup
```

Module 7: Incident Response

Objective

Create incident response procedures and runbooks.

Step 7.1: Create Incident Response Runbook (20 minutes)

Task: Document incident response procedures.

Create file: `docs/runbooks/incident_response.md`

```
# Incident Response Runbook

## Overview
This runbook provides step-by-step procedures for responding to data pipeline
incidents.

## Severity Levels

### P0 - Critical
- Production data pipeline completely down
- Data corruption affecting business decisions
- Security breach

**Response Time:** Immediate
**Escalation:** Notify management immediately

### P1 - High
- Partial pipeline failure
- Data quality issues affecting reports
- Performance degradation

**Response Time:** Within 1 hour
**Escalation:** Notify team lead

### P2 - Medium
- Non-critical test failures
- Minor data quality issues
- Slow pipeline execution

**Response Time:** Within 4 hours
**Escalation:** Standard team notification

### P3 - Low
- Documentation issues
- Non-blocking warnings
- Optimization opportunities

**Response Time:** Within 24 hours
**Escalation:** None required

## Incident Response Procedures

### 1. Data Pipeline Failure

**Symptoms:**
- Airflow DAG failing
- DBT models not running
```

```
- No data updates

**Diagnosis Steps:**

```bash
# 1. Check Airflow logs
docker logs dbt_airflow_project-airflow-scheduler-1 --tail 100

# 2. Check DBT logs
docker exec dbt_airflow_project-dbt-1 cat /usr/app/dbt/logs/dbt.log

# 3. Check database connectivity
docker exec dbt_airflow_project-dbt-1 dbt debug

# 4. Check recent changes
git log --oneline -10
```

Resolution Steps:

```
# 1. Identify failed task
# Check Airflow UI: http://localhost:8080

# 2. Review error messages
# Look for specific error in logs

# 3. Fix the issue
# Apply fix based on error

# 4. Clear failed task and retry
# In Airflow UI: Clear task and re-run

# 5. Verify fix
docker exec dbt_airflow_project-dbt-1 dbt run --select <model_name>
```

2. Data Quality Issue

Symptoms:

- DBT tests failing
- Unexpected data values
- Missing data

Diagnosis Steps:

```
# 1. Run specific test
cd dbt
dbt test --select <model_name>

# 2. Check test results
```
```

```
cat target/run_results.json | jq '.results[] | select(.status == "fail")'

# 3. Query the data
docker exec -it dbt_airflow_project-sqlserver-1 /opt/mssql-tools/bin/sqlcmd \
  -S localhost -U sa -P "YourStrong@Passw0rd" \
  -Q "SELECT TOP 100 * FROM dbo.brnz_customers ORDER BY ModifiedDate DESC"

# 4. Check data lineage
dbt docs generate
# Open docs and review lineage
```

Resolution Steps:

```
# 1. Identify root cause
# Review source data and transformations

# 2. Fix data or model
# Update SQL or fix source data

# 3. Re-run affected models
dbt run --select <model_name>+

# 4. Verify fix
dbt test --select <model_name>
```

3. Performance Degradation

Symptoms:

- Slow pipeline execution
- Timeouts
- High resource usage

Diagnosis Steps:

```
# 1. Check execution times
cat dbt/target/run_results.json | jq '.results[] | {name: .unique_id, time:
.execution_time}'

# 2. Check resource usage
docker stats

# 3. Check database performance
# Review SQL Server query plans

# 4. Identify slow models
dbt run --select <model_name> --debug
```

Resolution Steps:

```
# 1. Optimize slow models
# Add indexes, optimize SQL

# 2. Implement incremental models
# Convert full-refresh to incremental

# 3. Adjust resources
# Increase threads or memory

# 4. Test improvements
dbt run --select <model_name>
```

Rollback Procedures

Quick Rollback

```
# 1. Identify backup to restore
ls -la backups/

# 2. Run rollback script
./scripts/rollback.sh <environment> <backup_directory>

# 3. Verify rollback
dbt test --target <environment>
```

Git Rollback

```
# 1. Identify commit to revert
git log --oneline -10

# 2. Revert commit
git revert <commit_hash>

# 3. Deploy reverted code
./scripts/deploy.sh <environment> <target>
```

Post-Incident Actions

1. Document the Incident

Create incident report in `incidents/` directory:

```

```

```
cp docs/templates/incident_report_template.md incidents/$(date +%Y-%m-%d)-
incident-name.md
```

2. Update Runbooks

Document any new procedures discovered during incident response.

3. Implement Preventive Measures

- Add tests to catch similar issues
- Improve monitoring
- Update alerts
- Enhance documentation

4. Conduct Post-Mortem

Schedule team meeting to review:

- What happened
- Why it happened
- How we responded
- What we learned
- How to prevent it

Contact Information

On-Call Rotation:

- Week 1: Team Member A
- Week 2: Team Member B
- Week 3: Team Member C

Escalation:

- Team Lead: [email]
- Manager: [email]
- VP Engineering: [email]

External Contacts:

- Database Admin: [email]
- Infrastructure Team: [email]
- Security Team: [email]

```
### Step 7.2: Create Incident Report Template (15 minutes)

**Task:** Create a template for documenting incidents.
```

Create file: `docs/templates/incident_report_template.md`

```markdown
# Incident Report: [Incident Name]

## Incident Summary

**Date:** YYYY-MM-DD
**Time:** HH:MM (Timezone)
**Duration:** X hours Y minutes
**Severity:** P0 / P1 / P2 / P3
**Status:** Resolved / Investigating / Monitoring
**Incident Commander:** [Name]

## Impact

**Systems Affected:**
- [ ] Production data pipeline
- [ ] Staging environment
- [ ] Development environment
- [ ] Data quality
- [ ] Reporting/Analytics

**Business Impact:**
- Users affected: [number/description]
- Data affected: [tables/models]
- Reports impacted: [list]
- Revenue impact: [if applicable]

**Metrics:**
- Downtime: X hours
- Data delay: X hours
- Records affected: X rows

## Timeline

| Time | Event | Action Taken |
|------|-------|--------------|
| HH:MM | Issue detected | Alert triggered |
| HH:MM | Team notified | On-call engineer paged |
| HH:MM | Investigation started | Reviewed logs |
| HH:MM | Root cause identified | Found issue in X |
| HH:MM | Fix applied | Deployed patch |
| HH:MM | Verification | Ran tests |
| HH:MM | Issue resolved | Monitoring |

## Root Cause

### What Happened
[Detailed explanation of what went wrong]

### Why It Happened
[Root cause analysis]
```

```
### Contributing Factors
- Factor 1
- Factor 2
- Factor 3

## Detection

**How was it detected?**
- [ ] Automated alert
- [ ] User report
- [ ] Monitoring dashboard
- [ ] Manual check
- [ ] Other: [specify]

**Alert/Monitoring:**
[Details about alert that fired or should have fired]

## Resolution

### Immediate Fix
[What was done to resolve the incident]

```bash
# Commands used
git revert abc123
./scripts/deploy.sh production prod
```

Verification

[How the fix was verified]

```
# Verification commands
dbt test --select affected_models
python scripts/smoke_tests.py
```

Prevention

Short-term Actions

- [ ] Action 1 - Owner: [Name] - Due: [Date]
- [ ] Action 2 - Owner: [Name] - Due: [Date]
- [ ] Action 3 - Owner: [Name] - Due: [Date]

Long-term Actions

- [ ] Action 1 - Owner: [Name] - Due: [Date]
- [ ] Action 2 - Owner: [Name] - Due: [Date]
- [ ] Action 3 - Owner: [Name] - Due: [Date]
```

Lessons Learned

What Went Well

- Item 1
- Item 2
- Item 3

What Could Be Improved

- Item 1
- Item 2
- Item 3

Action Items

1. **Monitoring:** [Improvements needed]
2. **Testing:** [Additional tests to add]
3. **Documentation:** [Updates needed]
4. **Process:** [Process improvements]
5. **Training:** [Team training needs]

Related Issues

Similar Past Incidents:

- [Link to incident #1]
- [Link to incident #2]

Related Tickets:

- [Link to Jira/GitHub issue]

Attachments

- ☐ Error logs
- ☐ Screenshots
- ☐ Database queries
- ☐ Monitoring graphs
- ☐ Code changes

Sign-off

Reviewed by:

- ☐ Incident Commander
- ☐ Team Lead
- ☐ Engineering Manager

Date Closed: YYYY-MM-DD

````
### Step 7.3: Create Health Check Script (25 minutes)

**Task:** Create a script to check system health.

**Create file:** `scripts/health_check.py`

```python
#!/usr/bin/env python3
"""
System health check script
Checks the health of all components in the data pipeline
"""

import sys
import subprocess
import requests
import pyodbc
from datetime import datetime

class HealthCheck:
    def __init__(self):
        self.checks_passed = 0
        self.checks_failed = 0
        self.warnings = 0

    def print_header(self, text):
        print(f"\n{'='*60}")
        print(f"  {text}")
        print(f"{'='*60}\n")

    def check_docker_containers(self):
        """Check if Docker containers are running"""
        print("🐳 Checking Docker containers...")

        try:
            result = subprocess.run(
                ['docker', 'ps', '--format', '{{.Names}}\t{{.Status}}'],
                capture_output=True,
                text=True,
                check=True
            )

            containers = result.stdout.strip().split('\n')
            required_containers = [
                'airflow-scheduler',
                'airflow-webserver',
                'sqlserver',
                'dbt'
            ]

            running_containers = [c.split('\t')[0] for c in containers]
````
````

```
            for req in required_containers:
                found = any(req in container for container in running_containers)
                if found:
                    print(f"  ✅ {req} is running")
                    self.checks_passed += 1
                else:
                    print(f"  ❌ {req} is not running")
                    self.checks_failed += 1

            return self.checks_failed == 0

        except Exception as e:
            print(f"  ❌ Error checking Docker: {e}")
            self.checks_failed += 1
            return False

    def check_airflow_webserver(self):
        """Check if Airflow webserver is accessible"""
        print("\n🌐 Checking Airflow webserver...")

        try:
            response = requests.get('http://localhost:8080/health', timeout=5)
            if response.status_code == 200:
                print("  ✅ Airflow webserver is healthy")
                self.checks_passed += 1
                return True
            else:
                print(f"  ❌ Airflow webserver returned status
{response.status_code}")
                self.checks_failed += 1
                return False
        except Exception as e:
            print(f"  ❌ Cannot reach Airflow webserver: {e}")
            self.checks_failed += 1
            return False

    def check_database_connection(self):
        """Check SQL Server database connection"""
        print("\n🗄️  Checking database connection...")

        try:
            conn_string = (
                "DRIVER={ODBC Driver 17 for SQL Server};"
                "SERVER=localhost;"
                "DATABASE=AdventureWorks2014;"
                "UID=sa;"
                "PWD=YourStrong@Passw0rd;"
                "TrustServerCertificate=yes;"
            )

            conn = pyodbc.connect(conn_string, timeout=5)
            cursor = conn.cursor()
            cursor.execute("SELECT @@VERSION")
            version = cursor.fetchone()[0]
```

```
            print(f"  ✅ Database connection successful")
            print(f"     Version: {version.split('\\n')[0]}")

            conn.close()
            self.checks_passed += 1
            return True

        except Exception as e:
            print(f"  ❌ Database connection failed: {e}")
            self.checks_failed += 1
            return False

    def check_dbt_models(self):
        """Check if DBT models exist and are compiled"""
        print("\n📊 Checking DBT models...")

        try:
            # Check if models directory exists
            result = subprocess.run(
                ['ls', '-la', 'dbt/models/'],
                capture_output=True,
                text=True
            )

            if result.returncode == 0:
                print("  ✅ DBT models directory exists")
                self.checks_passed += 1
            else:
                print("  ❌ DBT models directory not found")
                self.checks_failed += 1
                return False

            # Check if models can be compiled
            result = subprocess.run(
                ['docker', 'exec', 'dbt_airflow_project-dbt-1', 'dbt', 'compile'],
                capture_output=True,
                text=True,
                cwd='.'
            )

            if 'Completed successfully' in result.stdout:
                print("  ✅ DBT models compile successfully")
                self.checks_passed += 1
                return True
            else:
                print("  ⚠️  DBT compilation has warnings")
                self.warnings += 1
                return True

        except Exception as e:
            print(f"  ❌ Error checking DBT models: {e}")
            self.checks_failed += 1
            return False
```

```
    def check_disk_space(self):
        """Check available disk space"""
        print("\n💾 Checking disk space...")

        try:
            result = subprocess.run(
                ['df', '-h', '.'],
                capture_output=True,
                text=True
            )

            lines = result.stdout.strip().split('\n')
            if len(lines) > 1:
                parts = lines[1].split()
                usage = parts[4].rstrip('%')

                if int(usage) < 80:
                    print(f"  ✅ Disk usage: {usage}% (healthy)")
                    self.checks_passed += 1
                elif int(usage) < 90:
                    print(f"  ⚠️  Disk usage: {usage}% (warning)")
                    self.warnings += 1
                else:
                    print(f"  ❌ Disk usage: {usage}% (critical)")
                    self.checks_failed += 1

                return True

        except Exception as e:
            print(f"  ⚠️  Could not check disk space: {e}")
            self.warnings += 1
            return True

    def check_recent_pipeline_runs(self):
        """Check if pipelines have run recently"""
        print("\n⏰ Checking recent pipeline runs...")

        try:
            # Check DBT run results
            result = subprocess.run(
                ['cat', 'dbt/target/run_results.json'],
                capture_output=True,
                text=True
            )

            if result.returncode == 0:
                print("  ✅ Recent DBT run results found")
                self.checks_passed += 1
                return True
            else:
                print("  ⚠️  No recent DBT run results")
                self.warnings += 1
                return True
```

```
        except Exception as e:
            print(f"  ⚠️  Could not check pipeline runs: {e}")
            self.warnings += 1
            return True

    def run_all_checks(self):
        """Run all health checks"""
        self.print_header("DataOps Health Check")
        print(f"Timestamp: {datetime.now().isoformat()}\n")

        # Run all checks
        self.check_docker_containers()
        self.check_airflow_webserver()
        self.check_database_connection()
        self.check_dbt_models()
        self.check_disk_space()
        self.check_recent_pipeline_runs()

        # Print summary
        self.print_header("Health Check Summary")
        print(f"✅ Checks Passed: {self.checks_passed}")
        print(f"❌ Checks Failed: {self.checks_failed}")
        print(f"⚠️  Warnings: {self.warnings}")
        print(f"\nTotal Checks: {self.checks_passed + self.checks_failed +
self.warnings}")

        # Determine overall health
        if self.checks_failed == 0:
            if self.warnings == 0:
                print("\n🎉 System is HEALTHY")
                return 0
            else:
                print("\n⚠️  System is HEALTHY with warnings")
                return 0
        else:
            print("\n🚨 System has ISSUES that need attention")
            return 1

def main():
    health_check = HealthCheck()
    exit_code = health_check.run_all_checks()
    sys.exit(exit_code)

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

Make executable:

```
chmod +x scripts/health_check.py
```

Test it:

```
python scripts/health_check.py
```

✅ Module 7 Checkpoint

Verify your progress:

- ☐ Incident response runbook created
- ☐ Incident report template created
- ☐ Health check script created and tested

Test:

```
# Run health check
python scripts/health_check.py
```

Commit:

```
git add docs/ scripts/health_check.py
git commit -m "docs: add incident response procedures and health check script"
git push origin feature/dataops-setup
```

Final Integration & Testing

Step 8.1: Create Pull Request (15 minutes)

Task: Create a pull request to merge your DataOps implementation.

```
# Ensure all changes are committed
git status

# Push final changes
git push origin feature/dataops-setup
```

On GitHub:

1. Go to your repository
2. Click "Pull Requests"
3. Click "New Pull Request"
4. Base: `develop`, Compare: `feature/dataops-setup`
5. Title: `feat: implement DataOps framework with CI/CD, testing, and monitoring`
6. Description: Use the PR template and fill in details

7. Create Pull Request

Step 8.2: Review and Merge (10 minutes)

Review checklist:

- ☐ All CI/CD workflows passing
- ☐ Code follows style guidelines
- ☐ Documentation is complete
- ☐ Tests are passing
- ☐ No security issues

Merge the PR:

1. Review all changes
2. Approve the PR
3. Merge to `develop`
4. Delete the feature branch

Step 8.3: Deploy to Staging (20 minutes)

Task: Deploy your DataOps implementation to staging.

```
# Pull latest develop
git checkout develop
git pull origin develop

# Run deployment
./scripts/deploy.sh staging staging

# Run smoke tests
python scripts/smoke_tests.py --server localhost --database AdventureWorks2014

# Run health check
python scripts/health_check.py
```

Step 8.4: Final Verification (15 minutes)

Task: Verify everything is working end-to-end.

Checklist:

- ☐ All Docker containers running
- ☐ Airflow webserver accessible
- ☐ DBT models compile and run
- ☐ All tests passing
- ☐ Monitoring DAG running
- ☐ Data quality checks passing
- ☐ Documentation generated

- ☐ CI/CD pipelines working

Commands:

```
# Check containers
docker ps

# Check Airflow
curl http://localhost:8080/health

# Run DBT
docker exec dbt_airflow_project-dbt-1 dbt run
docker exec dbt_airflow_project-dbt-1 dbt test

# Check monitoring
docker exec dbt_airflow_project-airflow-scheduler-1 airflow dags list

# Health check
python scripts/health_check.py
```

Lab Completion

🎉 Congratulations!

You've successfully implemented a complete DataOps framework for your data pipeline!

What You've Accomplished

Version Control & CI/CD:

- ✅ Git repository with proper branching strategy
- ✅ Pre-commit hooks for code quality
- ✅ GitHub Actions workflows for automated testing
- ✅ Pull request validation

Testing Framework:

- ✅ DBT schema tests
- ✅ Custom data quality tests
- ✅ Source freshness monitoring
- ✅ dbt-expectations integration
- ✅ Comprehensive test suite

Monitoring & Observability:

- ✅ Pipeline monitoring DAG
- ✅ Structured logging
- ✅ Alerting system
- ✅ Metrics collection

Data Quality:

- ✅ Great Expectations framework
- ✅ Expectation suites
- ✅ Automated validations
- ✅ Data quality DAG

Deployment Automation:

- ✅ Multi-environment configuration
- ✅ Deployment scripts
- ✅ Rollback procedures
- ✅ Smoke tests
- ✅ CI/CD deployment workflows

Incident Response:

- ✅ Incident response runbook
- ✅ Incident report template
- ✅ Health check script

Key Metrics

Track these metrics to measure your DataOps success:

Speed:

- Deployment frequency: [Track]
- Lead time for changes: [Track]
- Mean time to recovery (MTTR): [Track]

Quality:

- Test coverage: [Track]
- Data quality score: [Track]
- Incident frequency: [Track]

Efficiency:

- Pipeline execution time: [Track]
- Resource utilization: [Track]
- Cost per pipeline run: [Track]

Next Steps

Immediate (This Week):

1. Monitor the pipelines daily
2. Review alerts and adjust thresholds
3. Document any issues encountered
4. Share knowledge with team

Short-term (This Month):

1. Add more comprehensive tests
2. Optimize slow-running models
3. Enhance monitoring dashboards
4. Conduct team training

Long-term (This Quarter):

1. Implement advanced monitoring
2. Add more data quality checks
3. Optimize costs
4. Establish DataOps culture

Resources

Documentation:

- DBT Documentation
- Airflow Documentation
- Great Expectations
- GitHub Actions

Community:

- DBT Slack Community
- Airflow Slack Community
- DataOps Community
- r/dataengineering

Feedback

Please provide feedback on this lab:

- What worked well?
- What was challenging?
- What could be improved?
- What additional topics would you like to see?

Certificate of Completion

This certifies that you have completed:

DataOps Lab Guide - Step by Step Implementation

Completed by: [Your Name] **Date:** [Date] **Duration:** [Hours]

Skills Acquired:

- CI/CD for data pipelines
- Automated testing strategies
- Monitoring and observability

- Data quality frameworks
- Deployment automation
- Incident response

Appendix

A. Troubleshooting Guide

Issue: Docker containers not starting

```
# Check Docker status
docker ps -a

# Check logs
docker logs <container_name>

# Restart containers
docker-compose restart
```

Issue: DBT connection errors

```
# Test connection
docker exec dbt_airflow_project-dbt-1 dbt debug

# Check profiles
cat dbt/profiles.yml

# Verify SQL Server is running
docker exec -it dbt_airflow_project-sqlserver-1 /opt/mssql-tools/bin/sqlcmd -S
localhost -U sa -P "YourStrong@Passw0rd" -Q "SELECT @@VERSION"
```

Issue: GitHub Actions failing

- Check workflow logs in GitHub Actions tab
- Verify secrets are configured
- Ensure file paths are correct
- Check Python/DBT versions

Issue: Tests failing

```
# Run specific test
dbt test --select <test_name>

# Check test results
cat dbt/target/run_results.json
```

```
# Debug test
dbt test --select <test_name> --debug
```

B. Command Reference

Git Commands:

```
git status                          # Check status
git add .                           # Stage all changes
git commit -m "message"             # Commit changes
git push origin <branch>            # Push to remote
git pull origin <branch>            # Pull from remote
git checkout -b <branch>            # Create new branch
git merge <branch>                  # Merge branch
```

Docker Commands:

```
docker ps                           # List running containers
docker ps -a                        # List all containers
docker logs <container>             # View logs
docker exec -it <container> bash    # Enter container
docker-compose up -d                # Start services
docker-compose down                 # Stop services
docker-compose restart              # Restart services
```

DBT Commands:

```
dbt run                             # Run all models
dbt test                            # Run all tests
dbt compile                         # Compile models
dbt docs generate                   # Generate documentation
dbt source freshness                # Check source freshness
dbt run --select <model>            # Run specific model
dbt test --select <model>           # Test specific model
```

Airflow Commands:

```
airflow dags list                   # List all DAGs
airflow dags test <dag_id>          # Test DAG
airflow tasks test <dag> <task>     # Test task
airflow dags trigger <dag_id>       # Trigger DAG
```

C. Additional Resources

Sample .sqlfluff Configuration:

```
[sqlfluff]
dialect = tsql
templater = dbt
max_line_length = 120

[sqlfluff:rules]
tab_space_size = 4
indent_unit = space

[sqlfluff:rules:L010]
capitalisation_policy = upper

[sqlfluff:rules:L030]
capitalisation_policy = upper
```

Sample Makefile:

```
.PHONY: help setup test deploy clean

help:
    @echo "Available commands:"
    @echo "  make setup    - Set up development environment"
    @echo "  make test     - Run all tests"
    @echo "  make deploy   - Deploy to staging"
    @echo "  make clean    - Clean up temporary files"

setup:
    pip install -r dbt/requirements.txt
    pip install -r airflow/requirements.txt
    pre-commit install

test:
    cd dbt && dbt test
    pytest tests/
    python scripts/smoke_tests.py

deploy:
    ./scripts/deploy.sh staging staging

clean:
    find . -type d -name "__pycache__" -exec rm -rf {} +
    find . -type f -name "*.pyc" -delete
    rm -rf dbt/target/
    rm -rf dbt/dbt_packages/
```

End of Lab Guide

Thank you for completing this comprehensive DataOps lab! 🚀