

## Laboratorio Práctico Clase 4: Diagramación y Orquestación de Pipelines

**Objetivo:** Los estudiantes aprenderán a diagramar, orquestar y ejecutar pipelines de datos complejos aplicando prácticas de DataOps.

### Materiales y Herramientas:

- Herramienta de diagramación.
- Git, GitHub

### Entrega Esperada:

- Repositorio GitHub con código completo
- Diagramas del pipeline en /docs/
- Ejecuciones exitosas de CI/CD en GitHub Actions
- Reporte de pruebas de orquestación
- Documentación de estrategias de escalabilidad

### Fase 1: Diagramación del Pipeline

1. Definir los componentes del pipeline para incluir:
  - **Fuentes de datos:** API, archivos CSV, base de datos
  - **Procesos:** Validación, transformación, enriquecimiento
  - **Destinos:** Data Warehouse, archivos procesados, reportes
  - **Controles:** Monitoreo, logging, alertas
2. Crear el diagrama con la herramienta de diagramación de confianza, agregarlo en el repositorio de GitHub con la siguiente ruta *docs/pipeline\_diagram.md*.
3. Justificar diagrama y documentar dependencias, agregarlo en la ruta *docs/dependencies.md*.
  - Ejemplo de dependencias:
    - Dependencias del Pipeline:
      - Validación: Requiere esquema versionado
      - Transformación: Depende de validación exitosa
      - Enrichimiento: Requiere catálogo de productos actualizado
      - Carga: Depende de enriquecimiento exitoso
      - Reportes: Depende de carga exitosa

### Fase 2: Implementación de Orquestación

1. Crear el Orquestador Principal (*src/orchestrator.py*):
  - Guía:

```
import yaml
import logging
from datetime import datetime
```

```

from src.data_validation import DataValidator
from src.data_processing import DataProcessor
from src.data_enrichment import DataEnricher
from src.quality_checks import QualityChecker

class PipelineOrchestrator:
    def __init__(self, config_path='config/pipeline_config.yaml'):
        self.config = self.load_config(config_path)
        self.setup_logging()

    def load_config(self, config_path):
        with open(config_path, 'r') as file:
            return yaml.safe_load(file)

    def setup_logging(self):
        logging.basicConfig(
            level=logging.INFO,
            format='%(asctime)s - %(levelname)s - %(message)s',
            handlers=[
                logging.FileHandler('logs/pipeline_execution.log'),
                logging.StreamHandler()
            ]
        )
        self.logger = logging.getLogger(__name__)

    def execute_pipeline(self):
        """Ejecuta el pipeline completo con manejo de dependencias"""
        execution_id = datetime.now().strftime("%Y%m%d_%H%M%S")
        self.logger.info(f"Iniciando ejecución del pipeline: {execution_id}")

        try:
            # 1. Validación de datos
            self.logger.info("Ejecutando validación de datos...")
            validator = DataValidator(self.config['validation'])
            validation_result = validator.validate()

            if not validation_result['success']:
                raise Exception(f"Validación fallida: {validation_result['errors']}")

            # 2. Procesamiento de datos
            self.logger.info("Ejecutando procesamiento de datos...")
            processor = DataProcessor(self.config['processing'])
            processing_result = processor.process()
        
```

```

# 3. Enriquecimiento de datos
self.logger.info("Ejecutando enriquecimiento de datos...")
enricher = DataEnricher(self.config['enrichment'])
enrichment_result =
enricher.enrich(processing_result['processed_data'])

# 4. Validación de calidad
self.logger.info("Ejecutando validación de calidad...")
quality_checker = QualityChecker(self.config['quality'])
quality_result =
quality_checker.check_quality(enrichment_result['enriched_data'])

if not quality_result['passed']:
    raise Exception(f"Validación de calidad fallida:
{quality_result['issues']}")

# 5. Generación de reportes
self.logger.info("Generando reportes...")
self.generate_reports(enrichment_result['enriched_data'],
execution_id)

self.logger.info(f"Pipeline completado exitosamente:
{execution_id}")
return {
    'success': True,
    'execution_id': execution_id,
    'records_processed': processing_result['record_count']
}

except Exception as e:
    self.logger.error(f"Error en el pipeline: {str(e)}")
    self.send_alert(f"Pipeline falló: {str(e)}")
    return {'success': False, 'error': str(e)}

def generate_reports(self, data, execution_id):
    """Genera reportes de ejecución"""
    report = {
        'execution_id': execution_id,
        'timestamp': datetime.now().isoformat(),
        'records_processed': len(data),
        'pipeline_version': self.config['version']
    }

```

```

# Guardar reporte
with open(f'data/outputs/report_{execution_id}.json', 'w') as f:
    json.dump(report, f, indent=2)

def send_alert(self, message):
    """Envía alertas (simulado para el ejercicio)"""
    self.logger.info(f"ALERTA: {message}")
    # En un escenario real, integrar con Slack, Email, etc.

```

2. Configurar el pipeline, creando *config/pipeline\_config.yaml*.

- Guía:

```

version: "1.0"
pipeline:
    name: "sales-data-pipeline"
    description: "Pipeline de procesamiento de datos de ventas"

validation:
    schema_path: "data/schemas/sales_schema_v1.json"
    required_files:
        - "sales_data.csv"
        - "product_catalog.csv"

processing:
    output_path: "data/processed/"
    steps:
        - "clean_duplicates"
        - "handle_missing_values"
        - "calculate_totals"

enrichment:
    catalog_path: "data/reference/product_catalog.csv"
    lookup_tables:
        - "region_mapping"
        - "product_categories"

quality:
    checks:
        - "completeness_threshold: 0.95"
        - "freshness_max_hours: 24"
        - "row_count_variation: 0.1"

notifications:

```

```
on_success: true
on_failure: true
channels:
  - "log_file"
```

### Fase 3: CI/CD con Orquestación

1. Crear `.github/workflows/ci_orchestration.yml`:

- Guía:

```
name: CI - Pipeline Orchestration Tests
on: [push, pull_request]
```

```
jobs:
  test-orchestration:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.8, 3.9]

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

      - name: Set up Python ${{ matrix.python-version }}
        uses: actions/setup-python@v4
        with:
          python-version: ${{ matrix.python-version }}

      - name: Install dependencies
        run:
          pip install -r requirements.txt
          pip install pytest pytest-mock

      - name: Test orchestration components
        run:
          python -m pytest tests/test_orchestration.py -v

      - name: Validate pipeline configuration
        run:
          python scripts/validate_config.py

      - name: Dry-run pipeline
        run:
```

```

python -c "
from src.orchestrator import PipelineOrchestrator
orchestrator = PipelineOrchestrator('config/pipeline_config.yaml')
print('Orchestrator inicializado correctamente')
"

integration-test:
  runs-on: ubuntu-latest
  needs: test-orchestration
  steps:
    - name: Checkout code
      uses: actions/checkout@v3

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

    - name: Run integration test
      run: |
        python tests/integration_test.py

```

## 2. Crear `.github/workflows/cd_orchestrated_pipeline.yml`.

- Guía:

```

name: CD - Orchestrated Data Pipeline
on:
  workflow_dispatch:
    inputs:
      environment:
        description: 'Environment'
        required: true
        default: 'staging'
        type: choice
        options:
          - staging
          - production
    schedule:
      - cron: '0 6 * * *' # Ejecutar diariamente a las 6 AM

```

```

jobs:
  execute-pipeline:
    runs-on: ubuntu-latest
    environment: ${{ github.event.inputs.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 -r requirements.txt

  - name: Download source data
    run: |
      mkdir -p data/raw
      # Simular descarga de datos
      python scripts/download_sample_data.py

  - name: Execute orchestrated pipeline
    run: |
      mkdir -p logs
      python src/orchestrator.py
      echo "Pipeline execution completed"

  - name: Upload execution artifacts
    uses: actions/upload-artifact@v3
    with:
      name: pipeline-outputs-${{ github.run_id }}
      path: |
        data/outputs/
        logs/
      retention-days: 7

  - name: Generate execution report
    if: always()
    run: |
      python scripts/generate_execution_report.py

```

#### Fase 4: Pruebas de Orquestación

1. Crear Pruebas de Integración, en la ruta *tests/test\_orchestration.py*.
  - Guía:

```
import pytest
from unittest.mock import Mock, patch
from src.orchestrator import PipelineOrchestrator

class TestPipelineOrchestration:

    def test_pipeline_initialization(self):
        """Test que el orquestador se inicializa correctamente"""
        orchestrator = PipelineOrchestrator('config/pipeline_config.yaml')
        assert orchestrator.config is not None
        assert 'version' in orchestrator.config

    def test_execution_flow_success(self):
        """Test flujo de ejecución exitoso"""
        with patch('src.data_validation.DataValidator') as mock_validator, \
            patch('src.data_processing.DataProcessor') as mock_processor:

            # Configurar mocks para flujo exitoso
            mock_validator.return_value.validate.return_value = {'success': True}

            mock_processor.return_value.process.return_value = {
                'processed_data': [],
                'record_count': 100
            }

        orchestrator = PipelineOrchestrator('config/pipeline_config.yaml')
        result = orchestrator.execute_pipeline()

        assert result['success'] == True
```

```

        assert 'execution_id' in result

def test_execution_flow_failure(self):
    """Test flujo de ejecución con fallo en validación"""
    with patch('src.data_validation.DataValidator') as mock_validator:

        # Configurar mock para fallo en validación
        mock_validator.return_value.validate.return_value = {
            'success': False,
            'errors': ['Schema validation failed']
        }

    orchestrator = PipelineOrchestrator('config/pipeline_config.yaml')
    result = orchestrator.execute_pipeline()

    assert result['success'] == False
    assert 'error' in result

```

## Fase 5: Ejercicio de Escalabilidad

1. Crear diagrama de Escalabilidad, en la ruta *docs/scalability\_diagram.md*.
2. Documentar estrategias de escalabilidad, en la ruta *docs/scalability\_strategies.md*.

- Ejemplo:  
# Estrategias de Escalabilidad

```

## Nivel 1: Procesamiento Local
- Volumen: < 1GB
- Herramientas: Pandas, Python puro
- Ejecución: GitHub Actions estándar

```

```

## Nivel 2: Procesamiento en la Nube (Azure)
- Volumen: 1GB - 10GB
- Herramientas: Azure Functions, Azure Batch
- Ventajas: Escalado automático, costo por uso

```

## Nivel 3: Procesamiento Distribuido

- Volumen: > 10GB
- Herramientas: Azure Databricks, Azure Synapse
- Ventajas: Procesamiento paralelo, optimizado para big data

### Reflexión Final (Responder en la ruta *docs/reflection\_questions.md*)

1. **Diseño de Pipelines:** ¿Cómo decidió dividir un pipeline en componentes? ¿Qué criterios uso para definir las dependencias?
2. **Orquestación vs Ejecución:** ¿Cuál es la diferencia entre orquestar un pipeline y ejecutar sus componentes individualmente?
3. **Manejo de Fallos:** ¿Qué estrategias implementaría para:
  - Reintentos automáticos?
  - Continuación desde el punto de fallo?
  - Notificaciones escalonadas?
4. **Monitoreo:** ¿Qué métricas monitorearía para evaluar la salud del pipeline?
5. **Costos:** ¿Cómo optimizaría los costos de ejecución en la nube?