Airflow Packaged Project Structure

DAGs organized as an installable Python package. Proper versioning, unit testing, and clean separation of concerns.

#airflow #python #data #orchestration #package #ci-cd

PNG PDF Copy Prompt

Project Directory

```
airflow-dags/
  src/ Package source
    my_dags/ Installable pac…
      __init__.py
      dags/
        __init__.py
        etl_daily.py
        ml_pipeline.py
      operators/ Custom operators
        __init__.py
        slack_operator.py
      hooks/
        __init__.py
        api_hook.py
      utils/
        __init__.py
        dates.py
        alerts.py
  include/
    sql/
      queries.sql
  tests/
    __init__.py
    conftest.py Pytest fixtures
    test_dag_integrity.py
    test_etl_daily.py
  pyproject.toml Package config
  docker-compose.yml
  Dockerfile
  .gitignore
  README.md
```

Why This Structure?

DAGs as an installable package (`pip install -e .`) enables proper versioning, unit testing, and IDE support. Custom operators and hooks live alongside DAGs. The package is installed into the Airflow environment, not just copied to a folder.

Key Directories

`src/my_dags/dags/` - DAG files discovered by Airflow

`src/my_dags/operators/` - Custom operators as proper classes

`src/my_dags/hooks/` - Custom hooks for external systems

`tests/` - Unit and DAG integrity tests

Getting Started

1. `pip install -e ".[dev]"` to install package in editable mode
2. Set `dags_folder` in `airflow.cfg` to `src/my_dags/dags`
3. `pytest tests/` to run tests
4. Use Docker Compose for local development

DAG Integrity Test

```python
# tests/test_dag_integrity.py
import pytest
from airflow.models import DagBag

def test_no_import_errors():
    dag_bag = DagBag(include_examples=False)
    assert len(dag_bag.import_errors) == 0

def test_dag_has_tags():
    dag_bag = DagBag(include_examples=False)
    for dag_id, dag in dag_bag.dags.items():
        assert dag.tags, f"{dag_id} has no tags"
```

Best Practices

- Use `DagBag` tests to catch import errors in CI
- Pin Airflow version in `pyproject.toml` constraints
- Use factory functions for repeated DAG patterns
- Keep DAG files thin—business logic in utils/operators
- Tag all DAGs for filtering in the UI

When To Use This

- Teams needing proper CI/CD for DAGs
- Projects with custom operators and hooks
- Airflow deployments via Docker/Kubernetes
- 20-50+ DAGs with shared logic

Trade-offs

`Setup overhead` - More initial configuration than flat structure

`Deployment complexity` - Need to install package, not just sync files

`Learning curve` - Team needs Python packaging knowledge