DAGLint
Elevating Airflow DAG Quality Through Automated Linting

3.0

Snir Israeli
Data Engineer @Next

Agenda

01 Challenges with maintaining DAGs quality at scale

02 Our solution - DAGLint!

03 Integrations

04 Achievements

05 Q&A

DAGs at Scale Can be Messy

Inconsistency

Teams grow, styles diverge, no standards => **inconsistent DAGs**

Maintenance Hell

Debugging hurts, steep learning curve

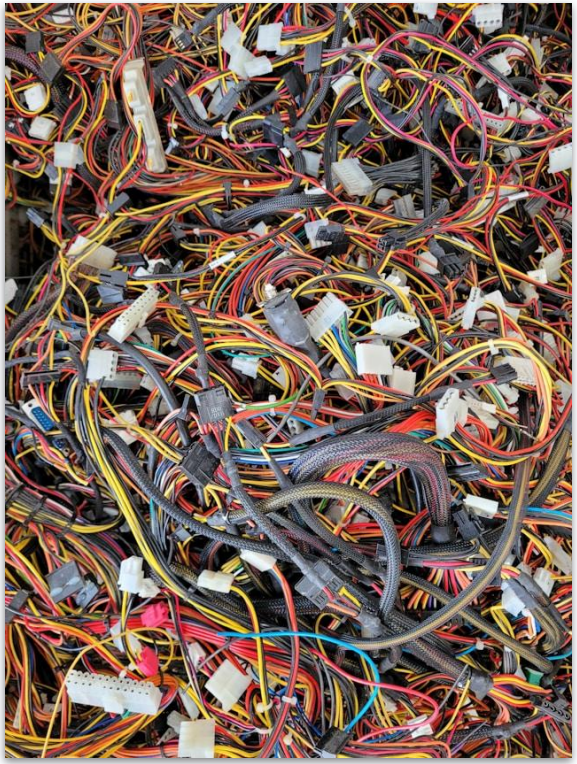


Photo by Nathan Cima on Unsplash

Hidden anti-patterns

Scheduler slowness, avoidable costs

Delayed Processes

Slow Code Reviews, onboarding drags

Inconsistency

Traditional Syntax

```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

def say_hello():
    print("Hello World")

with DAG(
    dag_id="hello_world_traditional",
    start_date=datetime(2025, 1, 1),
    schedule_interval=None,
    catchup=False,
) as dag:
    hello = PythonOperator(
        task_id="hello_task",
        python_callable=say_hello,
    )
```

TaskFlow API

```python
from airflow.decorators import dag, task
from datetime import datetime

@dag(
    dag_id="hello_world_taskflow",
    start_date=datetime(2025, 1, 1),
    schedule_interval=None,
    catchup=False,
)
def hello_dag():
    @task
    def say_hello():
        print("Hello World")

    say_hello()

hello_dag()
```

CodeImage

Traditional Syntax, logic imported

```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime
from hello_functions import say_hello

with DAG(
    dag_id="hello_world_external",
    start_date=datetime(2025, 1, 1),
    schedule_interval=None,
    catchup=False,
) as dag:
    hello = PythonOperator(
        task_id="hello_task",
        python_callable=say_hello,
    )
```

CodeImage

DAG missing Documentation & Tags

```python
# ❌ Missing doc_md - no owner, playbook, purpose
# ❌ Missing tags   - hard to navigate and filter

with DAG(
  dag_id="dag_with_no_documentation_nor_tags",
  start_date=datetime(2025,1,1),
  schedule_interval="@daily",
  catchup=False,
)
```

Top-level “expansive” code

```python
# ANTI-PATTERN: running expensive code at import time
import psycopg2
conn = psycopg2.connect("dbname=prod user=airflow")  # ❌ should be inside a task
cur = conn.cursor()
cur.execute("SELECT 1")  # runs on every scheduler/worker import

from airflow import DAG
from airflow.operators.empty import EmptyOperator
from datetime import datetime

with DAG(
  "bad_top_level_side_effects",
  start_date=datetime(2025,1,1),
  schedule_interval=None,
  catchup=False
) as dag:

  EmptyOperator(task_id="dummy")
```

What we tried before

...and why it didn't work

Training Sessions

The Problem

Knowledge fades over time. People forget best practices weeks after training.

Code Reviews

The Problem

Relies on reviewers remembering to check for anti-patterns. Inconsistent enforcement.

Documentation

The Problem

Becomes stale quickly. Doesn't evolve with changing best practices.

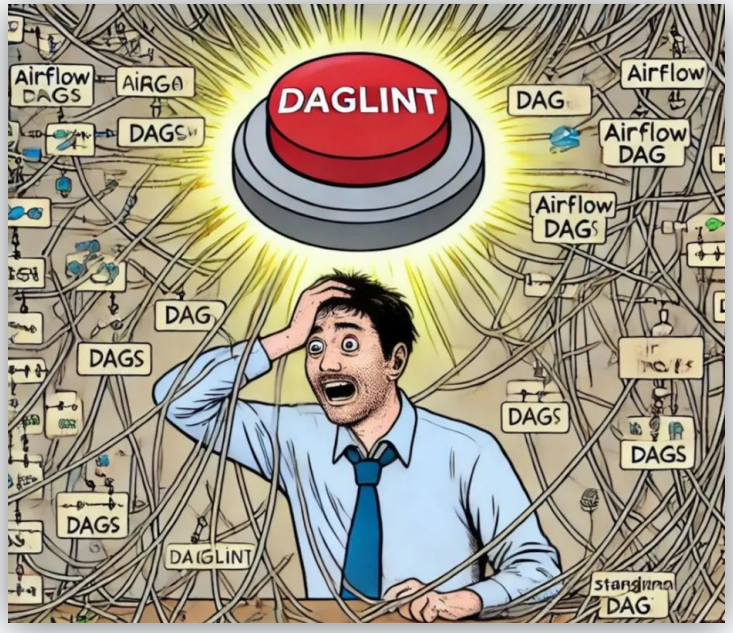
Monitor Performance

The Problem

Reactive approach. Issues are caught only after they've already impacted production.

Enter, DAGLint

- Linter for Airflow DAGs
- Runs locally on terminal
- Fast and deterministic
- Identify and prevent anti-patterns
- Clear & useful output



Architecture (High Level)

CLI & Local Dev Workflow

Lint single DAG

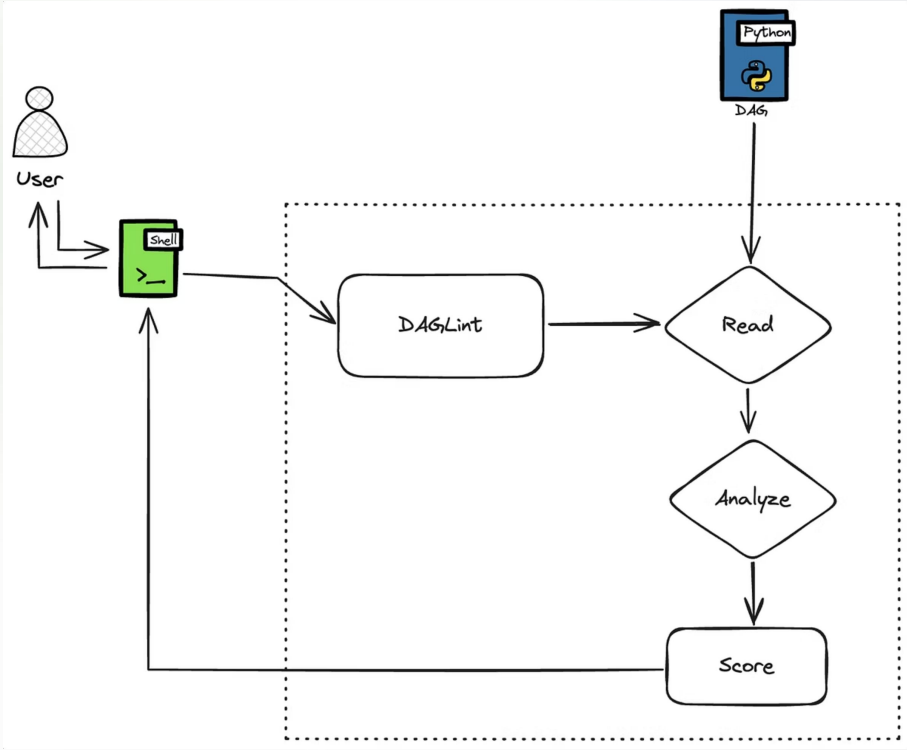
```
daglint /path/to/dags/ my_dag_name
```

Lint all active DAGs

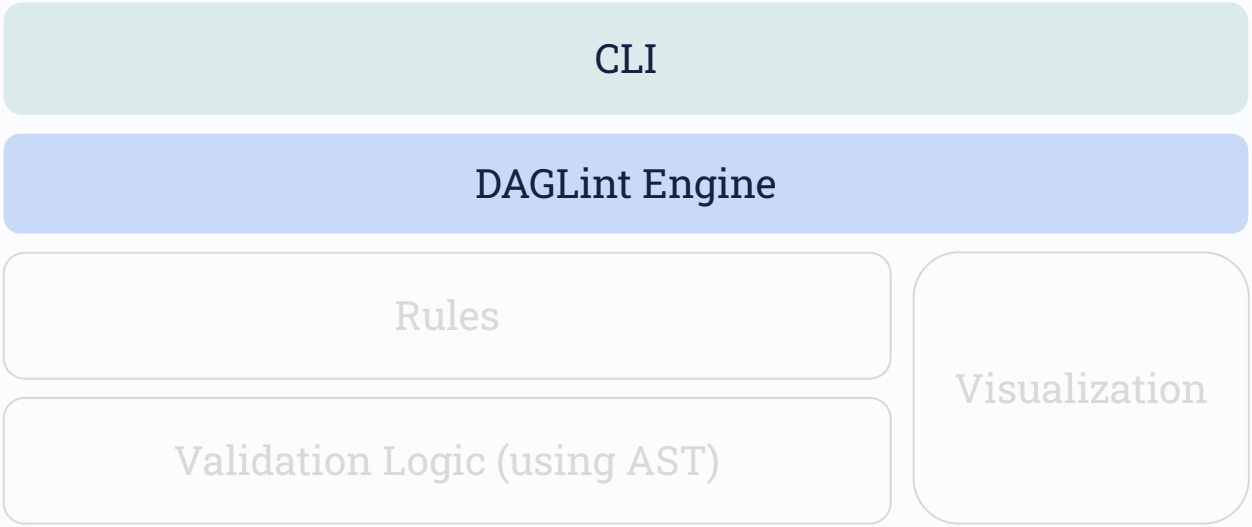
```
daglint /path/to/dags/ --all
```

Run specific rules

```
daglint /path/to/dags/ my_dag --rules_to_run R01,R16
```



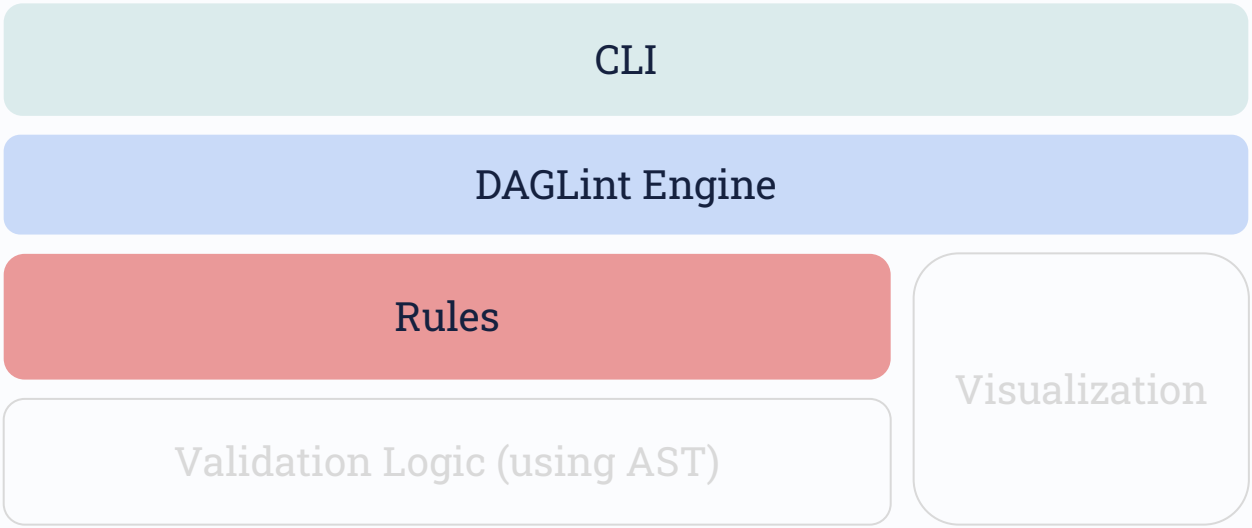
Architecture (High Level)



The Engine

- Walk DAGs directory & identify DAG files
- Lint DAG file/s
- Respect comment-based exclusions
- Scoring Mechanism

Architecture (High Level)



Rules Framework

- File Organization Rules
- DAG Structuring Rules
- Code Quality Rules

Naming Conventions

File Organizations

Valid DAG ID formats

Rules Framework

- File Organization Rules
- DAG Structuring Rules
- Code Quality Rules

Context managers

No function definitions

No business logic

Rules Framework

- File Organization Rules
- DAG Structuring Rules
- Code Quality Rules

No top-level expansive calls

README.md

How’s a rule defined?

```python
class CustomRule(LintRule):

  def __init__(self, **kwargs):
    super().__init__(
      name="custom_rule_name",
      description="Rule Description",
      id="R99",
      **kwargs
    )

  def validate(self):
    # Custom AST analysis logic...
```

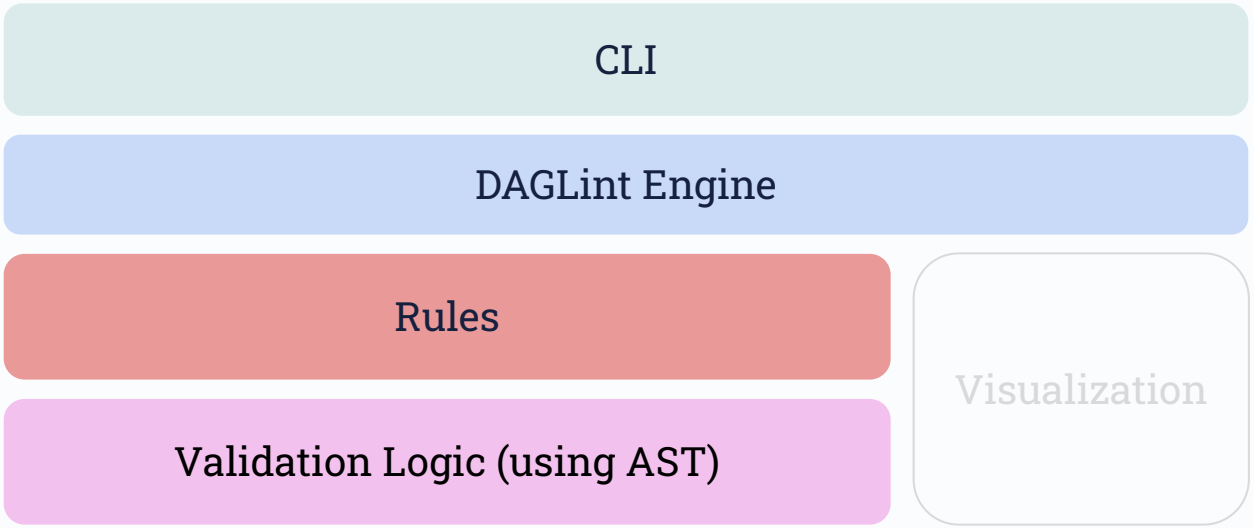

- Creating rules is simple; Just Inherit from *LintRule* and implement the *validate* method
- New rules are automatically discovered via inheritance

Exclusions (Granular & Documented)

- Localized opt-outs
- Force **documentation** of intent

```
# daglint-exclude: <<R04 Reason for exclusion

some_task = PythonOperator(task_id="task_id_that_violates_a_rule", …)

# R04>>
```

Architecture (High Level)



Python AST 🐍

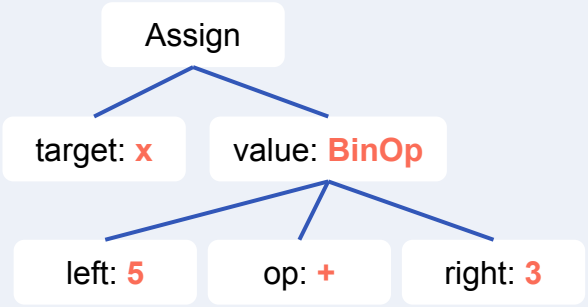
What is an AST?

- **Tree Structure:** Python breaks your code into a tree of nodes, where each node represents a construct in your code (like operations, variables, functions)
- **Abstract:** It ignores unnecessary details like whitespace and focuses on the structure and meaning
- **No execution:** Code is parsed and inspected without being executed, fast and safe.
- **Used For:** Code analysis, linters, formatters, transpilers, and understanding code structure programmatically

Your Python Code

```
x = 5 + 3
```

How Python "Sees" It (AST)



AST Node Visitors

What is a Node Visitor?

A Node Visitor is a pattern that lets you "walk" through every node in the AST tree and perform actions when you encounter specific node types.

How it works?

You create a class that inherits from `ast.NodeVisitor` and define `visit_*` methods for each node type you care about.

Simple Example

```python
import ast

class FunctionCounter(ast.NodeVisitor):
    def __init__(self):
        self.count = 0

    def visit_FunctionDef(self, node):
        self.count += 1

        self.generic_visit(node)


# Count all functions in code
counter = FunctionCounter()
counter.visit(ast.parse(code))
```

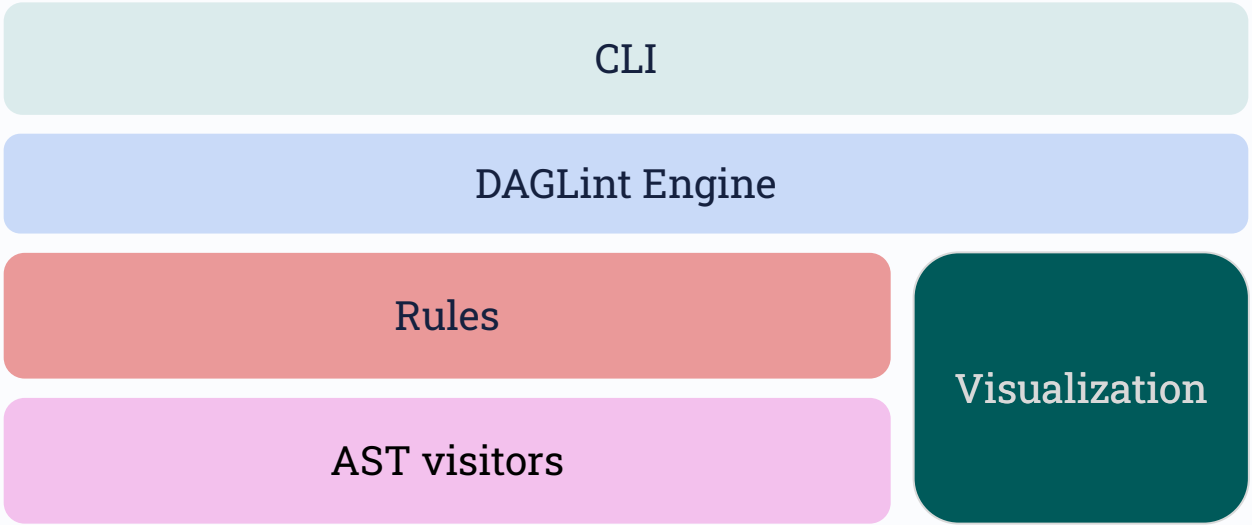
Airflow Example

```python
# dag_example.py
from airflow import DAG
from datetime import datetime

with DAG(
    dag_id="my_dag",
    start_date=datetime(2025, 1, 1),
    schedule_interval="@daily",
) as dag:
```



```python
import ast

class DAGDocValidator(ast.NodeVisitor):
    def visit_Call(self, node):
        # Check if this is a call to DAG(...)
        if isinstance(node.func, ast.Name) and node.func.id == "DAG":
            # Collect keyword argument names
            kwarg_names = {kw.arg for kw in node.keywords if kw.arg is not None}

            if "doc_md" not in kwarg_names:
                print("❌ DAG definition is missing 'doc_md' attribute")
            else:
                print("✅ DAG definition includes 'doc_md'")

        # Keep traversing child nodes
        self.generic_visit(node)


# Load and parse the DAG file
with open("dag_example.py", "r") as f:
    tree = ast.parse(f.read())

# Run validator
validator = DAGDocValidator()
validator.visit(tree)
```

Architecture (High Level)



Linting Results

Clear DAG Score

Linting Results for [redacted] *Score: 73.00%* 🤔

Rule Name	Description	Status	Line Numbers
dag_has_no_top_level_expensive_calls	The DAG must not have top-level calls to expensive classes/services, such as AWS services directly or via bi_toolbox utility classes. i.e. SecretsManager, S3, Athena, Redshift, etc.	Passed	N/A
dag_is_defined_only_as_context_manager	The DAG object should be instantiated using a context manager	Passed	N/A
dag_has_dag_description_configured	DAGs should have a description as a README.md file, located right next to the DAG's file, configured as a doc_md keyword argument.	Failed	N/A
dag_id_does_not_match_dag_file_name	The DAG file name should match its dag_id.	Passed	N/A
dag_filename_must_be_all_lowercase_characters	The DAG file name must be all lowercase characters.	Passed	N/A
dag_has_logger_defined	The DAG should have a logger defined in the file and are configured with all relevant parameters	Passed	N/A
dag_file_enclosed_within_its_own_folder	Every DAG should be placed in its own folder, its file name is part of its folder name and the folder is under the dags folder hierarchy.	Failed	N/A
dag_uses_only_[redacted]	The DAG should always use [redacted] and not directly the DataQualityHandler class	Passed	N/A
dag_has_a_valid_team_tag	The DAG object must have a tag for the team owning the DAG in the form of a team constant. i.e [redacted]	Passed	N/A
dag_qg_config_path_is_valid	The DAG should use [redacted] and have its corresponding config file in the config folder with the suffix '_dq_tests.json'.	Passed	N/A
[redacted]	[redacted]	Failed	26, 40, 54, 104, 131

Easily find violations

CI/CD

01

GitHub Actions

GitHub Actions runs DAGLint on every PR where a DAG file was modified

02

Merge Blocks

Critical rule failures block merges with detailed output

03

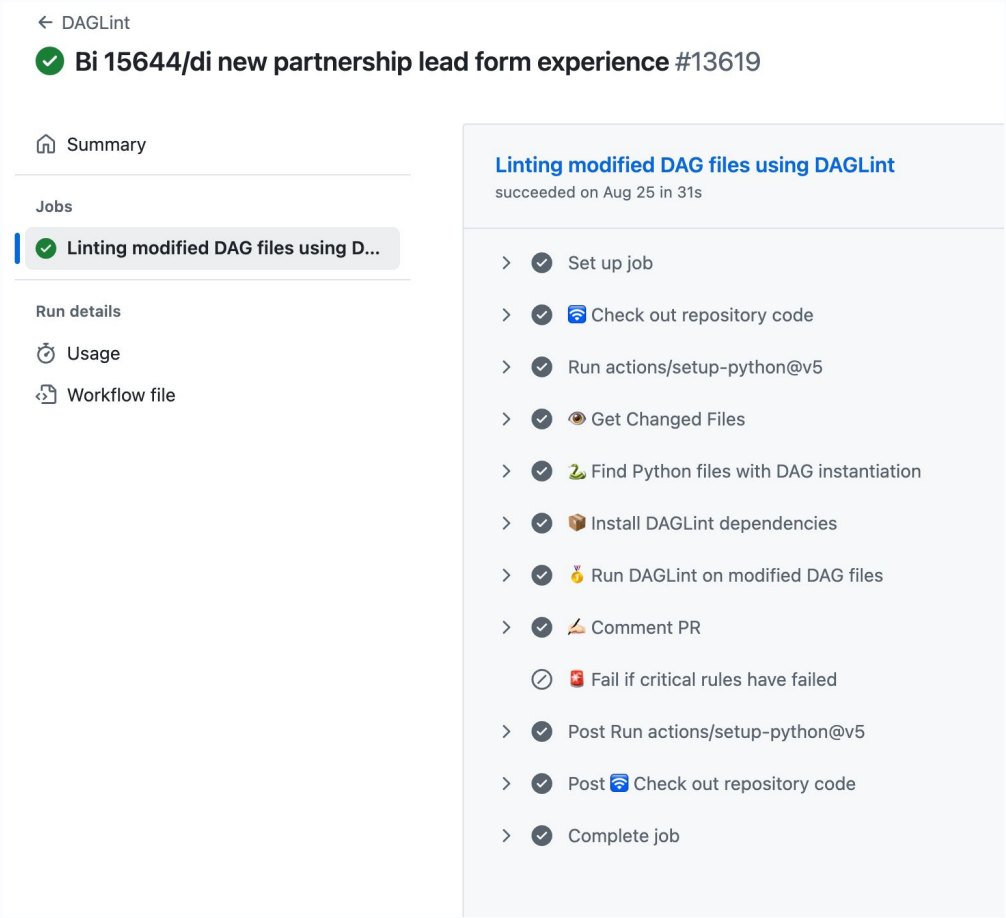
Uninterrupted Code Reviews

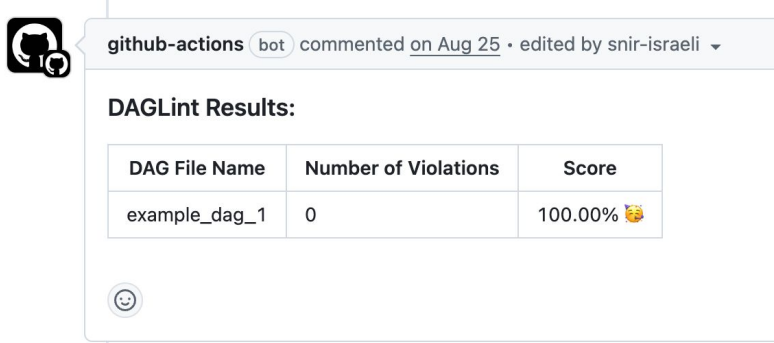
Reviewer can focus on what's important

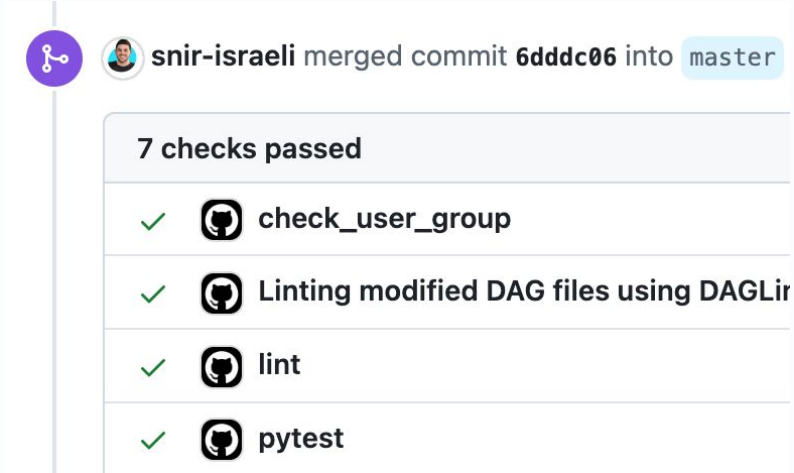
04

Clean Main Branch

Keeps main branch green & consistent

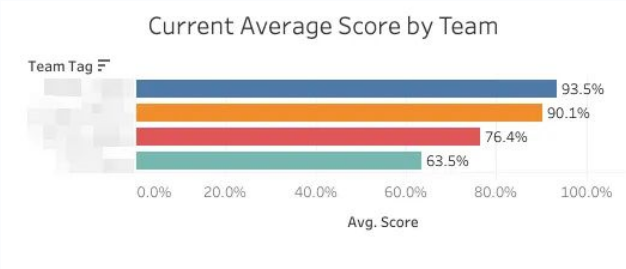

DAGLint
Bi 15644/di new partnership lead form experience #13619
Summary
Jobs
Linting modified DAG files using D...
Run details
Usage
Workflow file
Linting modified DAG files using DAGLint
succeeded on Aug 25 in 31s
Set up job
Check out repository code
Run actions/setup-python@v5
Get Changed Files
Find Python files with DAG instantiation
Install DAGLint dependencies
Run DAGLint on modified DAG files
Comment PR
Fail if critical rules have failed
Post Run actions/setup-python@v5
Post Check out repository code
Complete job

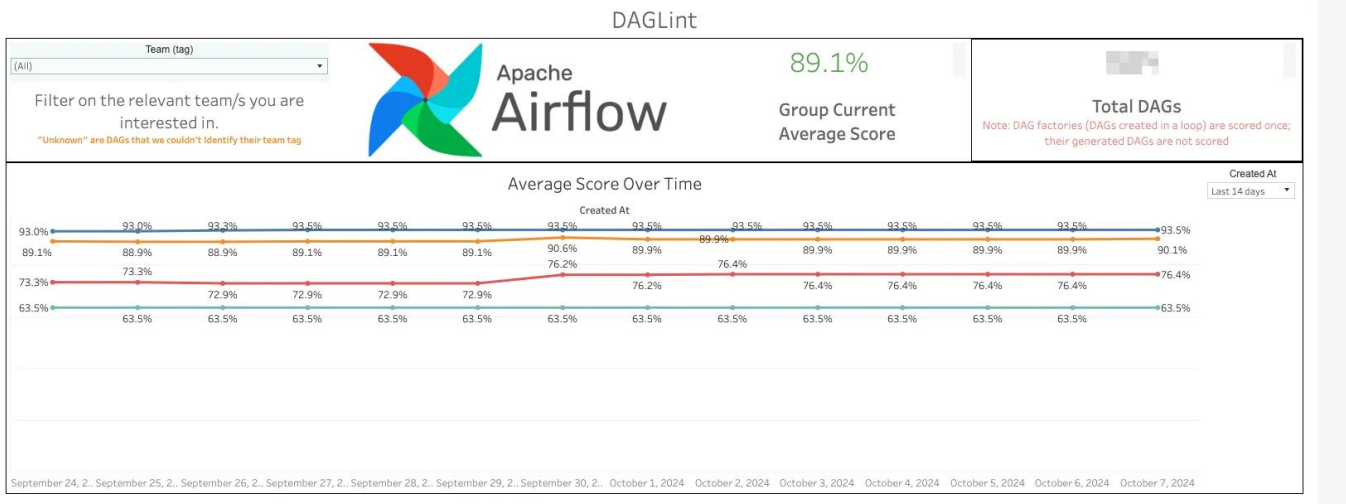

github-actions bot commented on Aug 25 • edited by snir-israeli
DAGLint Results:
DAG File Name
Number of Violations
Score
example_dag_1
0
100.00%


snir-israeli merged commit 6dddc06 into master
7 checks passed
check_user_group
Linting modified DAG files using DAGLir
lint
pytest

Monitoring & Analytics

- Daily **Jenkins** pipeline runs org-wide scoring
- Persist results to PostgreSQL
- Tableau Dashboard





Creative Use Case 💡

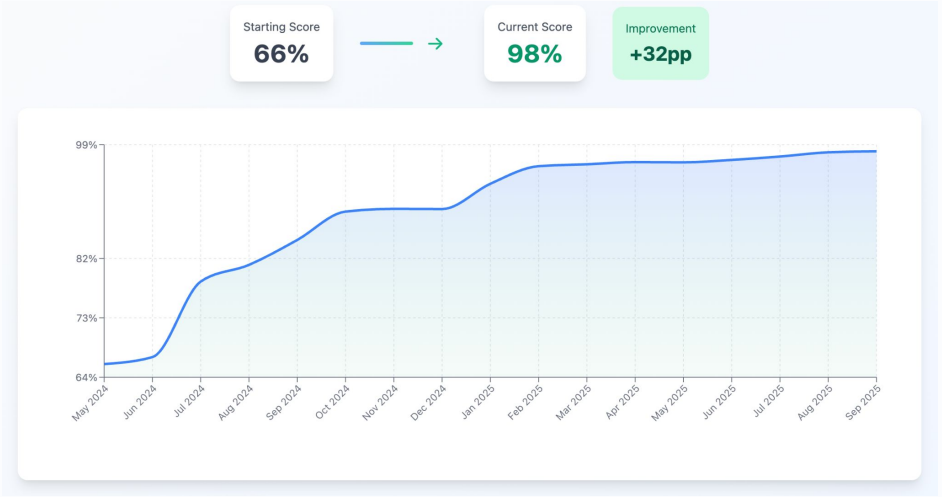
Gradual Migration without Regression

- Replace old custom operator usage with new version, **gradually**
- While transitioning safely, we didn't want new DAGs or updates to existing DAGs cause degradation
- Created a rule that disallow the usage of the old operator
- **Any code change to a DAG using the old operator will fail linting**

What did we achieve?

Achievements

- Developer Productivity
- Improved Code Reviews
- Ops & Compliance
- Improved Engineer Onboardings
- Quality enforced at scale



Takeaways

- Violating best-practices and inconsistencies = Quality Issues
- Airflow needs **domain-aware** linting
- Automatic enforcement + Monitoring = Quality & Compliance at scale!
- If you try it, focus on the developer experience!

Medium Article

Questions?

Thank you.

3.0

➔ LinkedIn: www.linkedin.com/in/snir-israeli