AIRFLOW SUMMIT

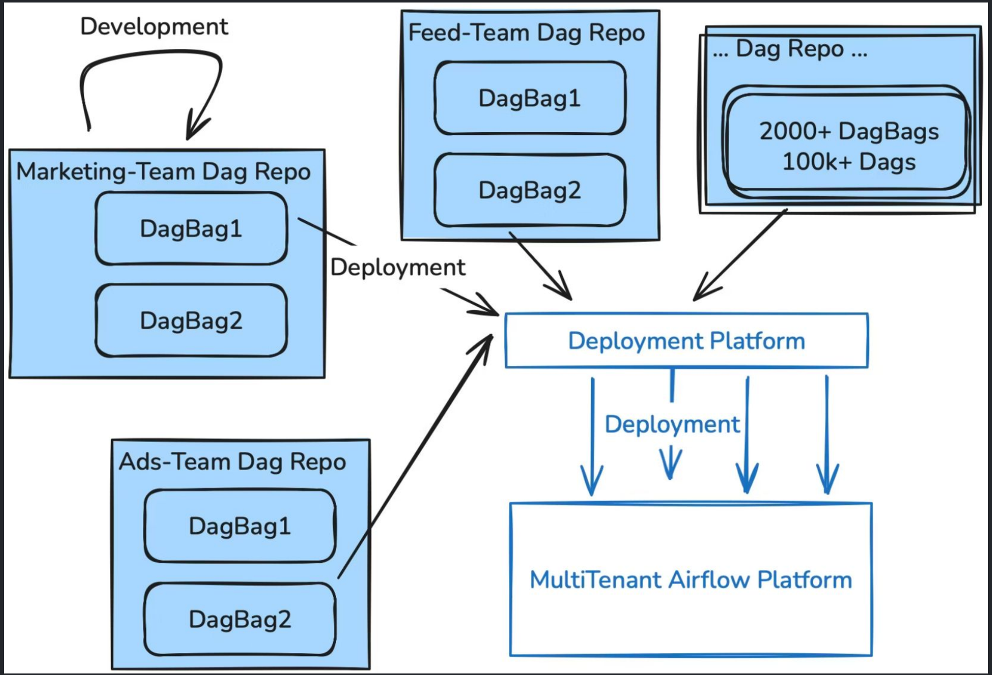
DAGnostics:Shift-Left Airflow Governance With Policy Enforcement Framework

Managing 100k+ DAGs Without Breaking Developer Velocity

Stefan Wang
Senior Software Engineer
Data Infrastructure @ LinkedIn
Airflow Summit 2025

3.0

The LinkedIn Scale Reality



Multi-Tenant Airflow Ecosystem

100K+

Active DAGs

Platform-wide execution

2,000+

DAG Repositories

Team-owned deployments

3,000+

DagBags

Isolated execution contexts

300K+

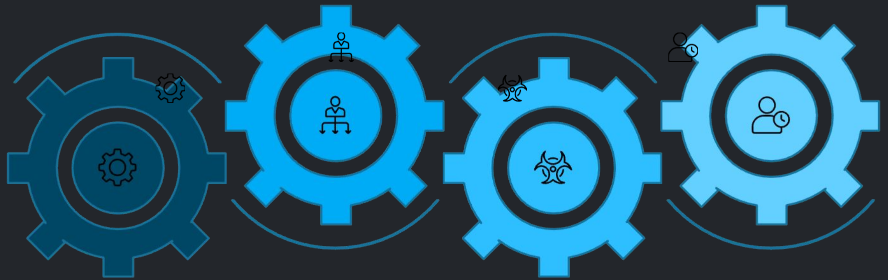
Daily Task Executions

Critical daily volume

These complex workflows handle functional data pipelines, critical revenue generation, and compliance mandates.

Past - The Monolithic Airflow DAG Repository

```
airflow/
├── dags/ # 2000+ teams
│   ├── marketing/
│   │   ├── weekly_email_campaign.py
│   ├── sales/
│   │   ├── monthly_sales_analysis.py
│   └── ...
├── plugins/
│   ├── shared_operators/
│   └── ...
├── requirements.txt # shared dependency across
thousands of teams
└── utils/ # shared utils across thousands of teams
    └── ...
```


Team Inputs
Many small arrows feed into the hub
Operational Impact
Slow, fragile deployments and conflicts
Central Gears
Jammed core causing cascade failures
Blockages
Broken teeth and stalled motion shown

Asks - Autonomous DAG Repository Model

Complete Lifecycle Ownership

Each of our 2,000+ teams controls their own DAG repository lifecycle: development, testing, and production.

Standardized Structure

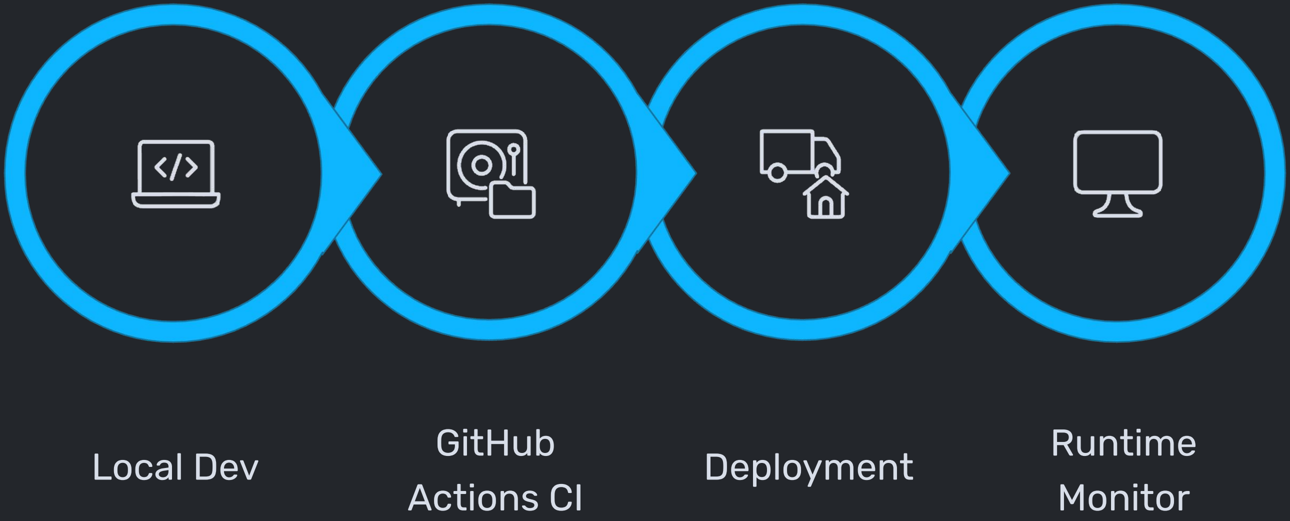
Consistent structure mandates DAG definitions, business logic, comprehensive tests, and isolated dependencies.

Autonomous Deployment

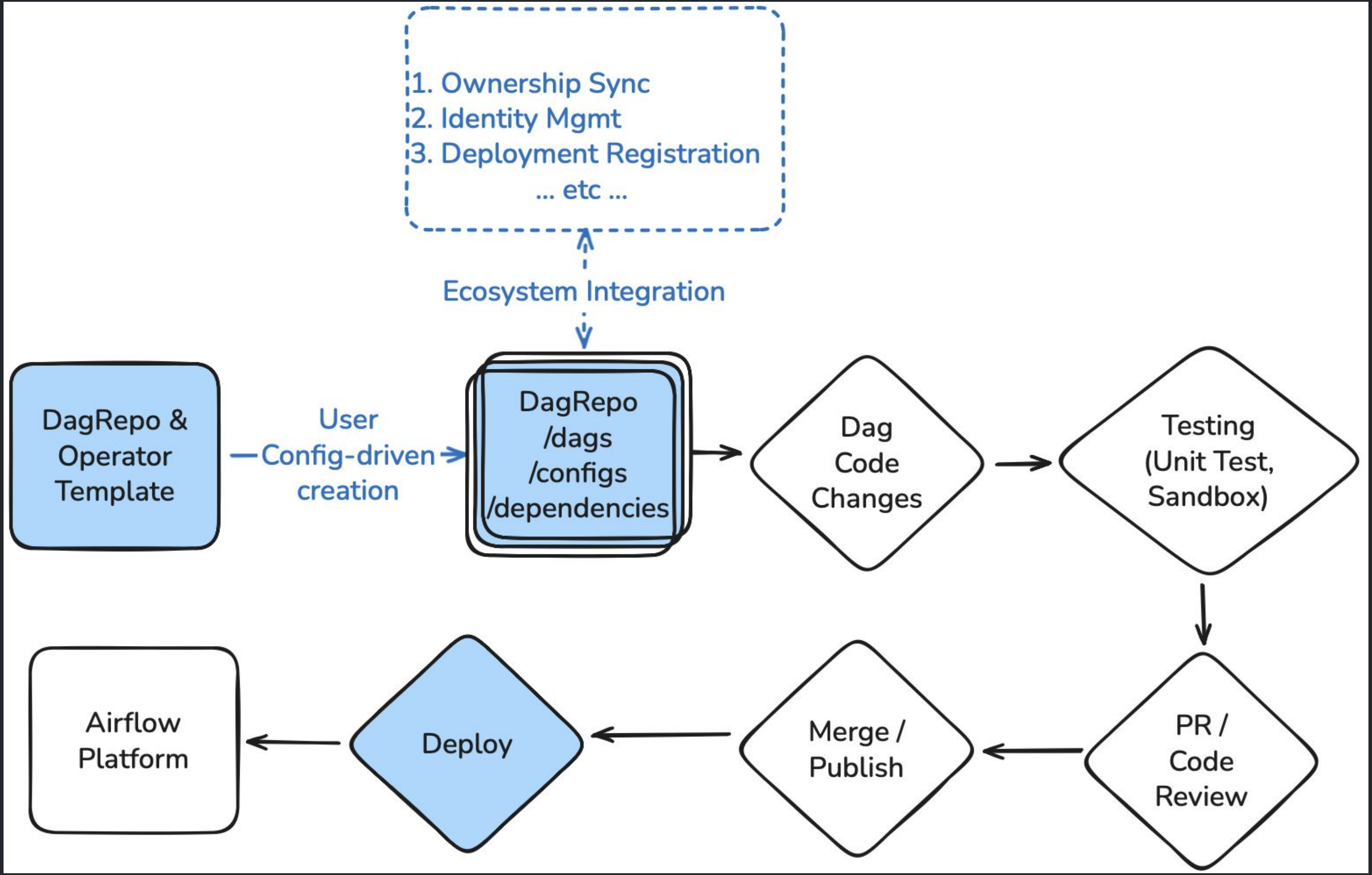
Teams deploy independently for velocity. Sophisticated governance ensures stability.

Vision - Enterprise DAG CI/CD Pipeline

- **Automated validation ensures rigorous policy compliance.**
- **Confidence built through staged verification.**
- **Maintains high deployment velocity.**



Future - Decentralized Dag Management


1. Ownership Sync
2. Identity Mgmt
3. Deployment Registration
... etc ...
Ecosystem Integration
DagRepo & Operator Template
User Config-driven creation
DagRepo /dags /configs /dependencies
Dag Code Changes
Testing (Unit Test, Sandbox)
PR / Code Review
Merge / Publish
Deploy
Airflow Platform

Why Governance Matters in Multi-Tenant Airflow

Chaos (No Governance)

- Failures only discovered in production
- Unclear ownership during critical incidents
- Frequent resource conflicts destabilize infrastructure
- Inconsistent practices across teams
- Compliance risks go undetected

Stability (With Governance)

- Errors caught pre-deployment in CI/CD
- Clear ownership tracking and accountability
- Coordinated resource usage prevents conflicts
- Standardized monitoring and alerting
- Proactive compliance enforcement

When 2,000+ independent teams deploy on shared Airflow infrastructure, governance transforms chaos into stability. Manual review cannot scale to hundreds of daily deployments—automated enforcement is essential.

[Current Runtime-only] Airflow Cluster Policy

Airflow's Native Policy System

Apache Airflow provides a cluster policy system to enforce custom rules. These policies are defined in `airflow_local_settings` and execute during DAG loading. Policies can validate or mutate DAGs, tasks, task instances, and Kubernetes pods. They can reject deployments by raising exceptions.

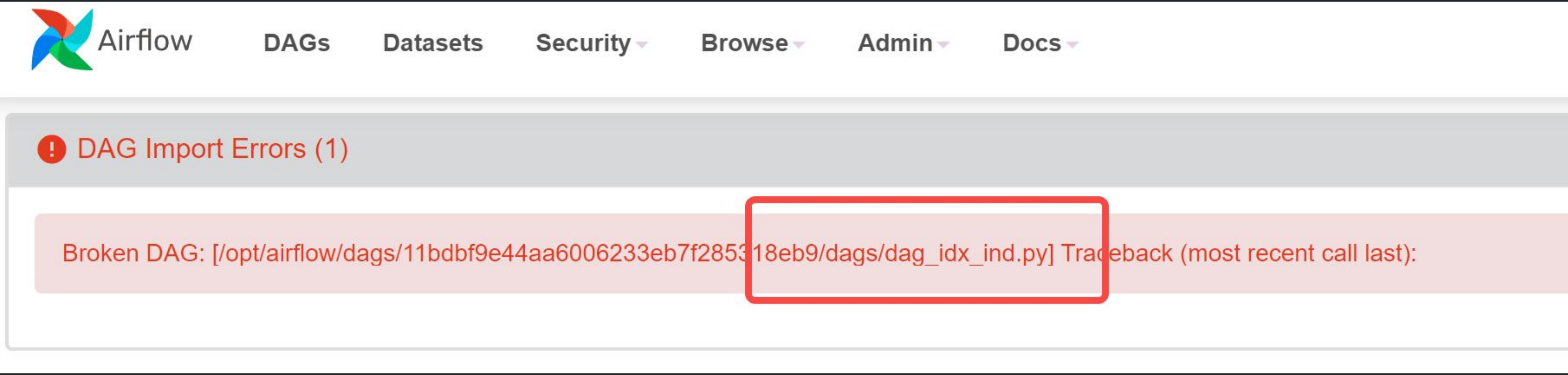
Two Critical Challenges

Late Discovery: Native policies execute within Airflow Platform Runtime. Violations are discovered after deployment in production.

No Preflight Validation: Developers cannot validate policies before committing code, leading to failed deployments and rollbacks.

So here's what we built...

Dag Import Errors - Looks Familiar?

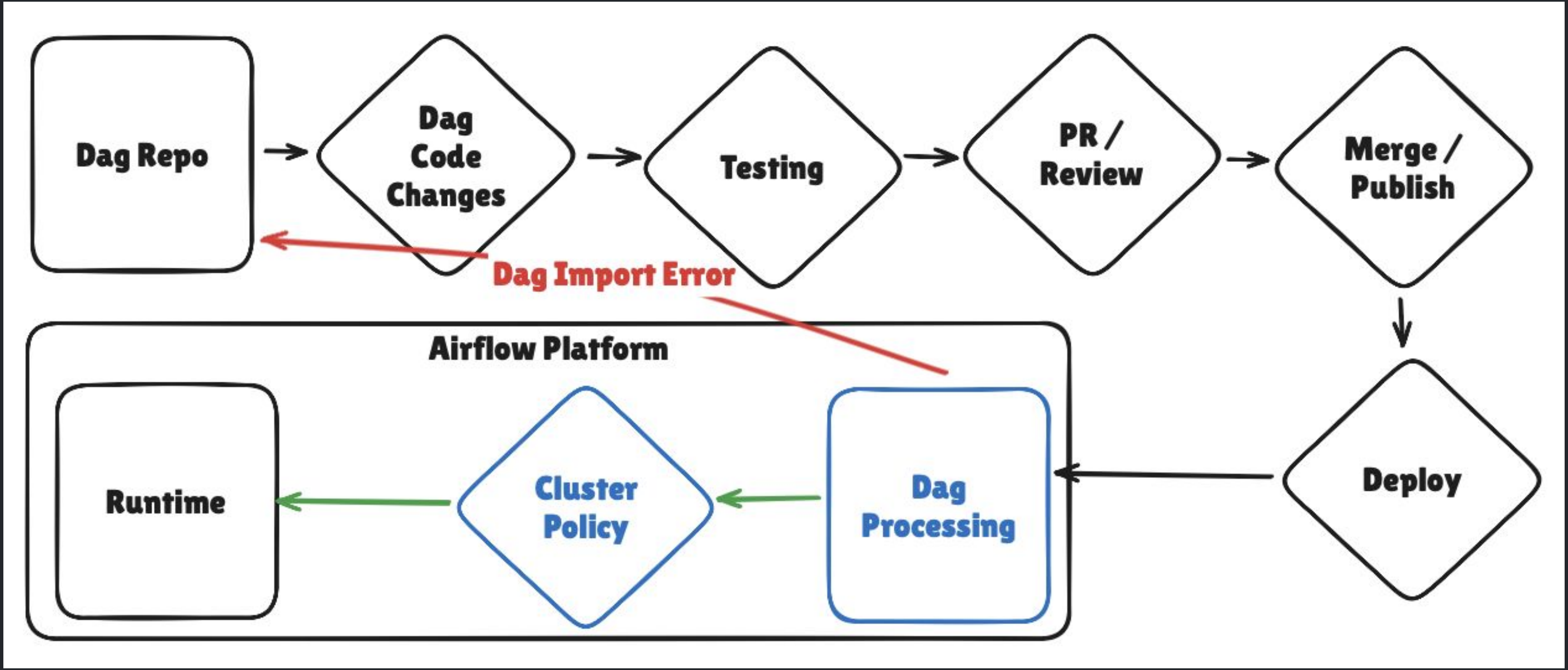


DAG Import Errors (1)

Broken DAG: [/opt/airflow/dags/11bdbf9e44aa6006233eb7f285318eb9/dags/dag_idx_ind.py] Traceback (most recent call last):

Screenshot taken from https://github.com/apache/airflow/issues/29897

Past


Dag Repo
Dag Code Changes
Testing
PR / Review
Merge / Publish
Deploy
Dag Import Error
Airflow Platform
Runtime
Cluster Policy
Dag Processing

Current State Challenges in Native Airflow Policies

Native Airflow policies pose significant challenges, primarily due to late-stage enforcement and rigid technical requirements.

Late Validation & Feedback

Policies run only in production, causing feedback delays (hours/days) and increased incident risk.

Complex Environment Coupling

Requires a full Airflow infrastructure for validation, making pre-deployment checks complex and resource-intensive.

Developer Experience Gaps

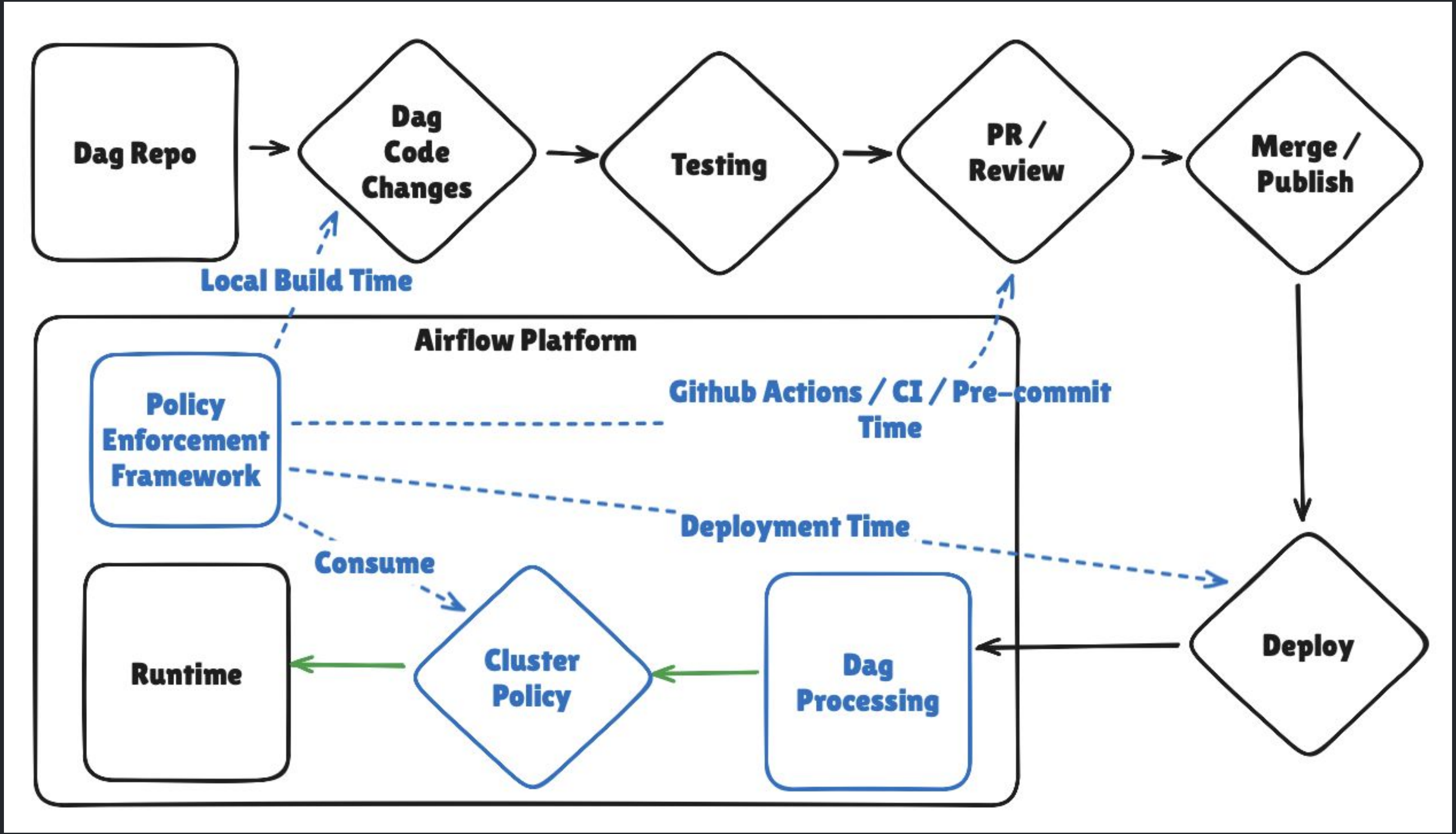
No easy local or CI/CD validation without mirroring a full Airflow environment, hindering early error detection.

Limited Extensibility & Observability

Lacks built-in exemption systems, emergency overrides, and composable validation; failures are hard to diagnose and audit.

A full production Airflow deployment is needed for DAG validation, preventing "shift-left" policy enforcement into development and CI/CD.

Future - No more Dag Import Errors in Production



Technical Solution: Portable DAG Validation Framework

This lightweight framework shifts Airflow DAG policy enforcement left, enabling earlier, more flexible, and developer-friendly validation.

Decoupled Environment Interface

Validate DAGs anywhere via module injection, eliminating full Airflow environment requirements.

Dynamic Dependency Resolution

Validate custom modules and proprietary libraries via runtime `sys.path` injection, no installation needed.

Leveraged Native Error Detection

Utilize `DagBag`'s `import_errors` for comprehensive, production-grade detection of all DAG errors.

Resulting in fast, independent, and comprehensive validation with immediate developer feedback.

The Policy Enforcement Engine

Enforcement Workflow

```
class PolicyEnforcer:

         def enforce_policies(dag_repo_path, environment):

                  # 1. Setup Environment
                  self.setup_environment(...)

                  # 2. Load DAGs (Airflow native)
                  dagbag = DagBag(dag_repo_path)

                  # 3. Apply Policies
                  for dag in dagbag.dags.values()
                           self.apply_dag_policies(dag)
```

This core loop ensures every DAG is tested against required operational standards before deployment.

Policy Definition: Declarative Rules

```
@hookimpldef
dag_policy(dag):

         # Check ID format
         validate_dag_id_format(dag)

         # Enforce alerting
         ensure_alerting_configured(dag)

         # Verify ownership
         validate_owner_metadata(dag)

         # Enforce compliance
         enforce_compliance_rules(dag)
```

Policies are standard Python functions that access the full DAG object. They can validate, mutate, or reject deployment based on any configuration criteria.

Composable Design: New rules can be added without modifying the core enforcement engine.

Error Type	Catch as Dag Import Error	Code Location
Missing dependency	✅ Yes	_load_modules_from_file()
Syntax error	✅ Yes	_load_modules_from_file()
Top-level exceptions	✅ Yes	_load_modules_from_file()
Cycle detection	✅ Yes	_process_modules() catch
DAG validation errors	✅ Yes	_process_modules() catch
AirflowClusterPolicyViolation	✅ Yes	_process_modules() catch
DAG ID collision	✅ Yes	_process_modules() catch
Unknown executor	✅ Yes	_process_modules() catch

The Shift-Left Insight

Runtime Discovery is Costly

Traditional governance enforces policies only in production. This is the worst time to find errors:

- **Immediate failure and customer impact.**
- **Rollback is risky.**
- **Debugging is slow and complex.**

The Shift-Left Advantage

Shift-Left Governance moves validation earlier in the lifecycle (Local & CI/CD). Catch issues when fixes are easiest and cheapest.

100%

Runtime Cost

Maximum impact and risk

10%

CI/CD Cost

Caught before deployment

1%

Local Development Cost

Identified during coding

Use Case #1: Missing Dag Parsing Dependency Errors

The Problem: Environment Drift

DAGs that work locally often fail upon deployment due to critical differences in the production environment:

- **Missing team-specific shared libraries**
- **Python version mismatches**
- **Conflicting package versions**
- **Missing system dependencies**

Import failures cause DAGs to disappear from the Airflow UI, leading to broken workflows and late-stage incidents.

The DAGnostics Solution

CI Environment Simulation

Replicates production Python and system dependencies within the CI pipeline.

Auto-Load Dependencies

Loads team-specific shared libraries using repository metadata.

Validate Imports

Executes actual DAG imports using Airflow's native DagBag loader to guarantee success.

Actionable Reporting

Surfaces import errors and full stack traces directly in the pull request.

0

Production Import Failures

Since implementing CI environment simulation

100%

Pre-Deployment Detection

All import issues caught during pull request validation

Use Case #2: Ensuring DAG ID Uniqueness Identity & Ownership

Challenges with 100,000+ DAGs

At massive scale (2,000+ repositories), maintaining unique identity and tracking ownership is critical. Manual coordination fails:

- **ID collisions cause deployment failures.**
- **Incident response lacks immediate owner identification.**
- **Access control systems need verifiable ownership data.**
- **Compliance audits require clear accountability trails.**

Our Automated Enforcement

1 Standardized DAG ID Format

Enforced format: `{dag_name}-{repo_name}` (using globally unique repo name).

2 Automatic Metadata Sync

Ownership synced instantly from the central repository system.

3 Access Control Integration

Ownership feeds directly into permission systems for automated authorization.

4 Clear Audit Trail

Lineage tracked from DAG to repository to owning team for compliance.

100%

DAG Ownership Visibility

Every single DAG traceable to owning team

0

ID Collisions

Automatic format enforcement prevents conflicts

Use Case #3: Alerting Policy Enforcement

The Risk: Silent Failures

Critical data pipelines that fail without alerting represent one of the highest-impact operational risks. When revenue-generating workflows break silently:

- **Business metrics drift without warning**
- **Compliance deadlines are missed**
- **Customer-facing features degrade**
- **Problems compound before detection**

Manual alerting configuration is prone to human error. Teams forget to add alerts, misconfigure integrations, or use inconsistent escalation paths.

Our Policy Enforcement

DAGnostics enables making alerting a structural requirement, not an optional best practice:

- **All production DAGs must define failure alerting**
- **Standardized timeout policies prevent infinite hangs**
- **Integration with centralized monitoring platforms**
- **Automatic escalation paths based on DAG criticality**
- **SLA monitoring for time-sensitive workflows**

1

Define Requirements

Policy specifies which DAGs require alerting based on tags or metadata

2

Validate Configuration

CI checks verify callback functions or notification integrations exist

3

Block Deployment

Pull requests can't merge without proper alerting setup

4

Runtime Verification

Production policies confirm alerts are still configured and functional

> **"Since enforcing alerting policies, we've eliminated an entire class of incidents where critical workflows failed unnoticed for hours or days."**

Questions?

Ask us About Building Airflow at Scale

Our journey scaled Airflow to 100,000+ DAGs. Shift-left governance enabled high developer velocity at enterprise scale.

LinkedIn

linkedin.com/in/stefanwang

Let's Connect: