

Migrating Apache Airflow from 2 to 3 for Infrastructure Operations at Scale

Airflow Summit 2026
August 31, 2026

Rumeysa Ozaydin, Software Engineer
Ethan Lin, Site Reliability Engineer

TechAtBloomberg.com

© 2026 Bloomberg Finance L.P. All rights reserved.

Engineering

Bloomberg

Infrastructure Workflows

What Kinds of Workflows Can Airflow Orchestrate?

Launch

Provisioning new infrastructure resources through multi-stage orchestration.

Decommission

Retiring infrastructure and cleaning up resources at the end of lifecycle.

Relaunch

Rebuild and reprovision workflows combining decommission and launch.

Post-OS Configuration

Applying machine configuration after system bring-up:

- Filesystem setup
- Network configuration

Maintenance

Handling fleet-wide maintenance workflows:

- Operating system patching
- Firmware upgrades
- Image-based updates
- Coordinated reboot orchestration

The Original Model: SaltStack

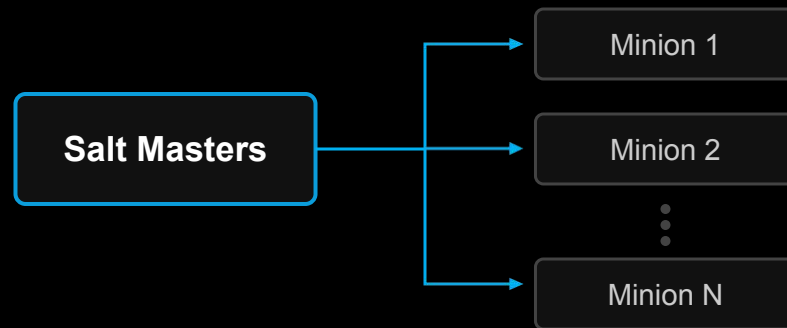
Before Airflow: The Existing Automation Stack

SaltStack Platform

Primary automation platform with essential distributed capabilities:

- Remote command execution
- Orchestration support

Salt Architecture



The "Invisible" Pipeline

Operational challenges with master-minion architecture:

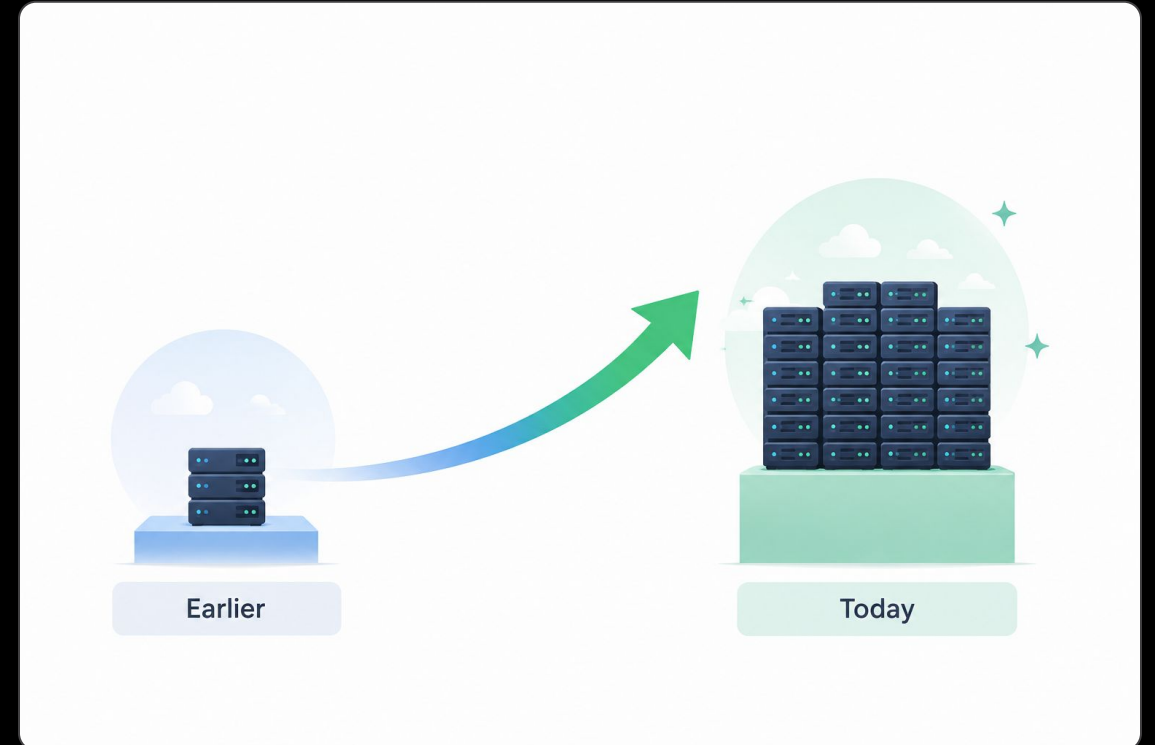
- **No restart** from failure point
- **Lacks documentation** unlike self-documented DAGs
- **No Native UI**
- **Logs spread** across multiple systems
- **Failures buried** in noisy debug logs

Orchestration Scaling Challenges

As orchestration workflows grew larger and more complex, scaling challenges emerged:

- Large orchestration states could fail midway
- Debugging depended heavily on logs, and workflow execution paths were difficult to visualize
- Rerunning a single failed step was difficult, and supporting teams needed deep Salt expertise
- Operational visibility became a bottleneck

At scale, observability and recoverability matter as much as execution.



Airflow Did Not Replace Salt

Layered Architecture: Separation of Concerns

Salt: Execution Engine

Lower-level implementation details:

- **Machine Execution**
Direct interaction with fleet nodes.
- **Orchestration Actions**
Triggering specific remote procedures.

Airflow: Control Plane

Higher-level flow management:

- Workflow orchestration
- Retries and recovery
- Dependency management
- Branching decisions
- Operational visibility
- Execution tracking
- Deferred operations

What Fleet Orchestration Needs

Workflow Characteristics

- **Long-running**
Workflows that span hours, days, or weeks.
- **Asynchronous**
Triggers tasks without waiting for immediate execution.
- **Dependency-heavy**
Complex relationships and precise order of operations.
- **Operationally recoverable**
Resilient to infrastructure failures and retrievable.

Typical Workflow Shape

1. Submit work to downstream
2. Wait for external completion
3. Resume based on status/events
4. Branch on operational outcome

Understanding Dependencies

Infrastructure Workflows Are Dependency Graphs

Infrastructure workflows naturally follow a logical order. Tasks run in sequence, failures stop downstream execution, and progress is observable.

```
# infra_workflow.py
from airflow import DAG
with DAG("infrastructure_workflow" ) as dag:
    validations = EmptyOperator(task_id= "validations")
    provision_vm = EmptyOperator(task_id= "provision_vm")
    configure_network = EmptyOperator(task_id= "config_net")
    filesystems = EmptyOperator(task_id= "filesystems")
    qc = EmptyOperator(task_id= "quality_checks" )
    validations >> provision_vm >> configure_network >> filesystems >> qc
```

Benefit 01

Clear execution order

Benefit 02

Automatic stop on failure

Benefit 03

Centralized operational view

Not Every Workflow Is Linear

Infrastructure orchestration often requires workflows to change direction dynamically.

Airflow BranchOperator

Allows a DAG to choose different execution paths at runtime based on workflow state or external conditions.

Common Example: Success vs. Failure

Why This Matters

- Model operational decision trees
- Robust failure handling & rollbacks
- Environment-specific orchestration paths

Implementation Example

```
@task.branch
def choose_path(validation_passed):
    if validation_passed:
        return "deploy_service"
        return "rollback_host"

# Define pipeline execution sequence
validate_health >> choose_path() >>
[deploy_service, rollback_host]
```

Deferred Execution for Long-Running Workflows

Infrastructure Workflows Spend Most of Their Time Waiting

THE PROBLEM: IDLE WORKERS

Traditional execution keeps worker processes occupied while waiting. At scale, this leads to:

- Wasted worker capacity
- Reduced fleet concurrency
- High infrastructure overhead

THE SOLUTION: DEFERRED TASKS

Airflow allows tasks to submit work and release the worker process, resuming when ready.

- Submit work & release worker
- Asynchronous waiting
- Resume execution later

THEY WAIT FOR

Wait for External APIs

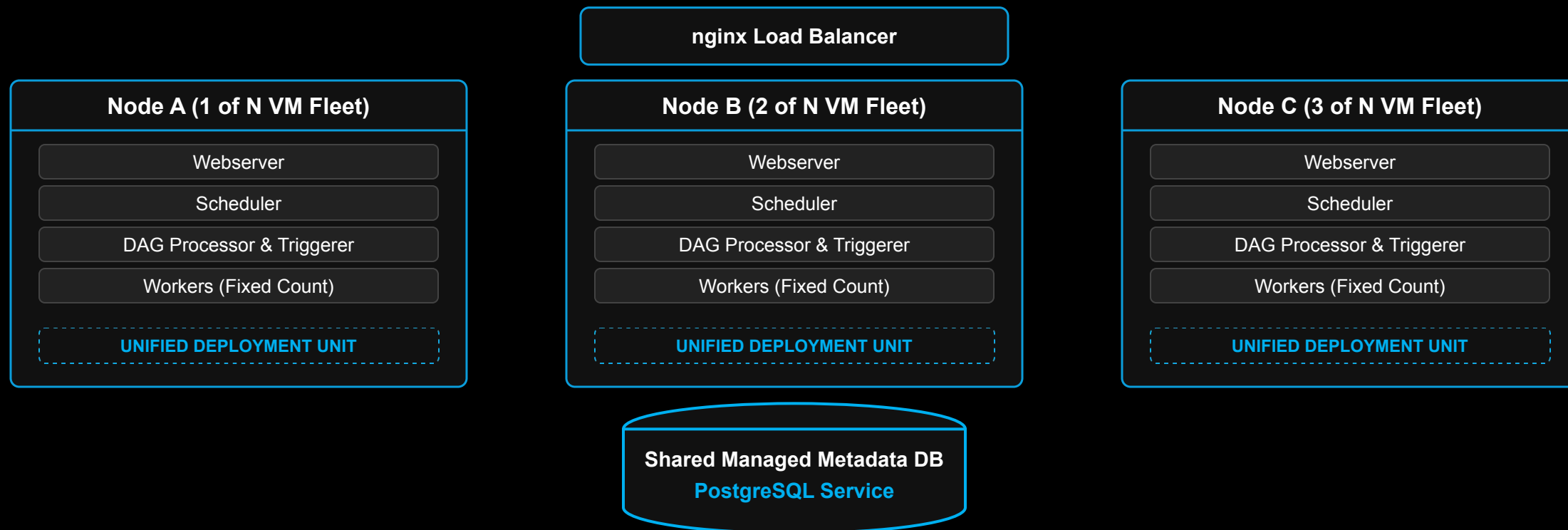
Poll Downstream Systems

Wait for Async Events

Maintenance Windows

Where We Were: The Airflow 2 Monolith

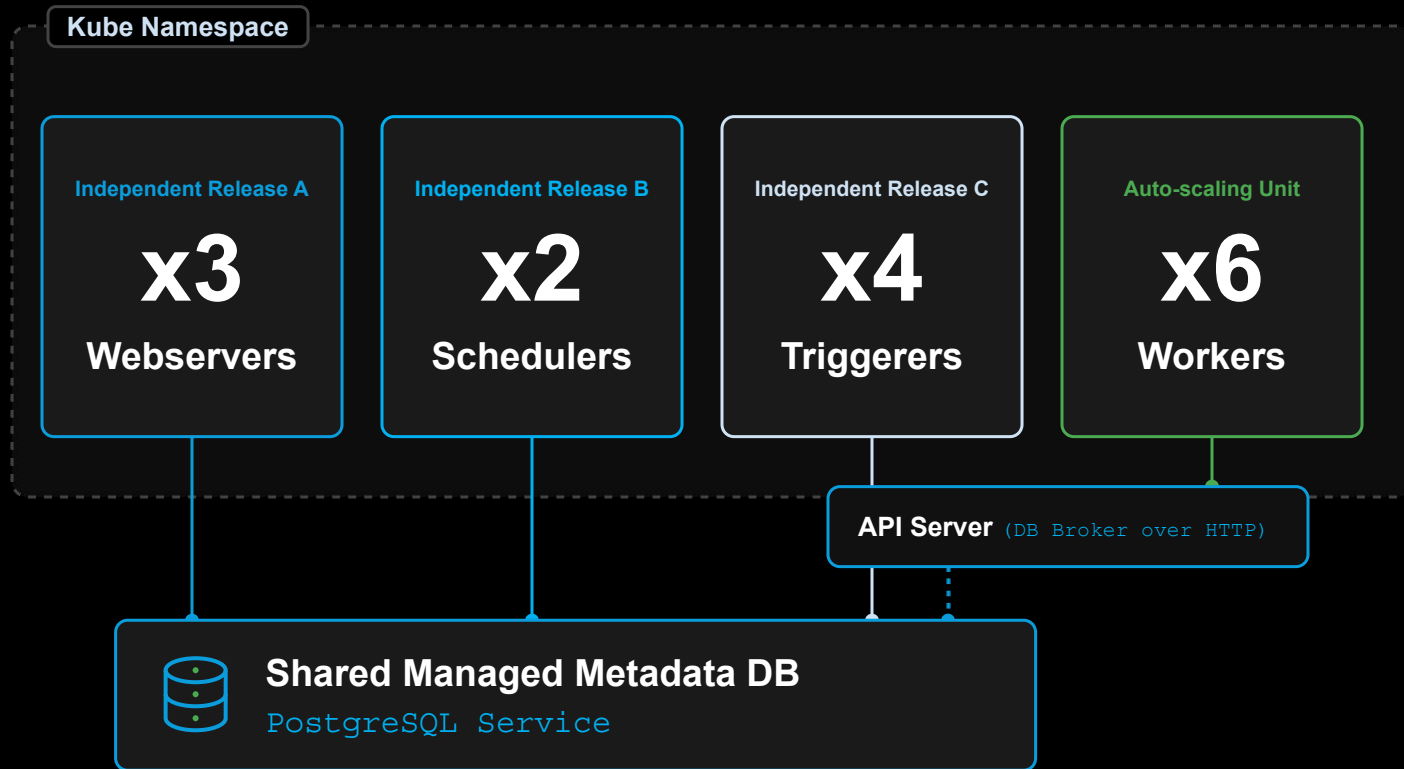
Airflow 2 Architecture: Co-located Components & Shared State



BOTTLENECK: No independent scaling. Components are tightly coupled. Direct DB access required from every node.

Where We're Going: Airflow 3 on Kubernetes

De-coupled Architecture: Independent Scaling & Release Cycles



Expected Outcomes

- ✓ **Elastic scaling** per-component
- ✓ **Zero-downtime** rolling upgrades
- ✓ **Optimized** resource utilization

The Breaking Change

The Change

Airflow 2 Implicitly searches upstream tasks for the key.

Airflow 3 Returns **None** — no error, no warning.

The Problem

```
val1 = ti.xcom_pull(key="my_key") # None
val2 = ti.xcom_pull(task_ids="src", key="k") #
      "Hello"
```

Docs: *"Just add task_ids to every call."*

The Problem at Scale

- ◆ Hundreds of legacy DAGs using implicit resolution.
- ◆ Manual refactoring is error-prone and unsustainable.
- ◆ Required a transparent compatibility layer.



Silent failure leads to unpredictable behavior in production workflows.

Gen 1: Walk the DAG

The Context

The docs say "`just add task_ids`" to every call.

The Reality

- **Scale:** Hundreds of legacy DAGs using implicit resolution.
- **Goal:** Resolve task_id dynamically at runtime.

Traversal Logic (Gen 1)

Sequential lookup approach

- 01 Get current TaskInstance context
- 02 Traverse DAG graph upstream
- 03 Collect all ancestor IDs
- 04 Iterative XCom queries until match

Not Sustainable

O(n) Overhead

Sequential XCom queries per pull execution.

Intrusive Setup

Requires wrapping every single `xcom_pull` call.

Performance Degradation

Execution becomes dramatically slow at production DAG scale.

Gen 2: External Index with Wrapper Functions

Mechanism: Dual-Write & Direct Read

Dual-write on push:

- Updates native XCom (for UI visibility) + `xcom_key_index` table.

Index Table:

- Maps `(dag_id, run_id, key) task_id + value`.

Direct read on pull:

- Single DB query on index; no `task_ids` needed.

⚠ Still Invasive / Requires Import Changes

Remaining Problem: Every legacy DAG must change its imports to use wrapper functions.

- Still requires "touching every DAG" at scale.
- Manual refactoring of hundreds of production files.
- Dependency on custom internal libraries.

Gen 3: CompatXComBackend

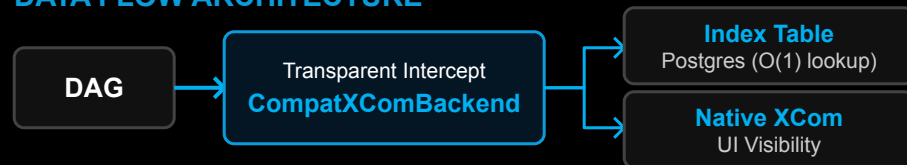
INFRASTRUCTURE CONFIGURATION

- Subclasses **BaseXCom**: Transparently hooks into Airflow.
- `xcom_backend = ...CompatXComBackend`

DAG AUTHOR EXPERIENCE (ZERO CHANGE)

```
ti.xcom_push(key="k", value="v")
val = ti.xcom_pull(key="k") # Just works
```

DATA FLOW ARCHITECTURE



SMART INTERCEPTION LOGIC

ON PUSH: DUAL-WRITE

- **Native XCom**: Stores value for UI visibility.
- **Index Table**: Records mapping in Postgres.
- **Design**: UPSERT semantics & thread-safe.

ON PULL: SMART DETECTION

- **Auto-filled**: Query index first (O(1)), then native.
- **Explicit IDs**: Straight to native XCom.
- **Robust**: Graceful degradation to native on DB error.

KEY WIN: Transparent, **O(1)** lookup, **zero code changes** required, and completely invisible to DAG authors.

Three Generations of Solving XCom Migration

OFFICIAL FIX: Manual `task_ids`

- Explicitly add `task_ids` to every call.

The Burden

Touch every DAG • Massive manual effort • High error risk

GENERATION 1: DAG Traversal

- Automated runtime search for ancestors.

The Burden

O(n) DB overhead • Sequential queries • Code changes needed

GENERATION 2: Wrapper DB Lookup

- External index table with direct mapping.

The Burden

Must change imports • Wrapper functions • Library dependency

GENERATION 3: Custom XCom Backend

- Framework-level override via `CompatXComBackend`.

The Optimal Fix

Truly ZERO code changes • O(1) Lookups • Invisible to authors • Pure config

KEY WIN: Gen 3 achieves transparent compatibility — the fix is completely decoupled from the DAG code.

XCom Performance Across Generations

ENTIRELY EXPERIMENTAL DATA

Test: 20 branches / 200 ops

Metric	Gen 1 (DAG Traversal)	Gen 2 (External Index)	Gen 3 (CompatXComBackend)
vs. Airflow 2	21x slower	on par	1.4x faster
Total Runtime	155s	4.3s	2.9s
Avg. Latency	775ms	21ms	14ms
p50	166ms	17ms	12ms
p90	691ms	40ms	25ms
Throughput	1.3 QPS	47 QPS	68 QPS

Key-only pulls (no task_id) also show no degradation in Gen 3

Thank you!

<https://TechAtBloomberg.com>

<https://www.bloomberg.com/careers>

Engineering

Bloomberg

TechAtBloomberg.com

© 2026 Bloomberg Finance L.P. All rights reserved.