

Cross-Deployment Monitoring and Dag Dependencies with the Kafka Event Producer Plugin

Christos Bisias, G-Research



Christos Bisias

RESEARCH OPEN SOURCE SOFTWARE

Introduction

- Open Source Software Engineer
- G-Research Open Source team
- Contributing to Apache Airflow for more than 2 years

Multiple Airflow Deployments

- ➔ Different Teams
- ➔ Different Hardware
- ➔ Different DBs
- ➔ Different Locations

Can they be coordinated?

Can we monitor them?

Can we get their state without accessing their DBs?

Kafka Event Producer Plugin

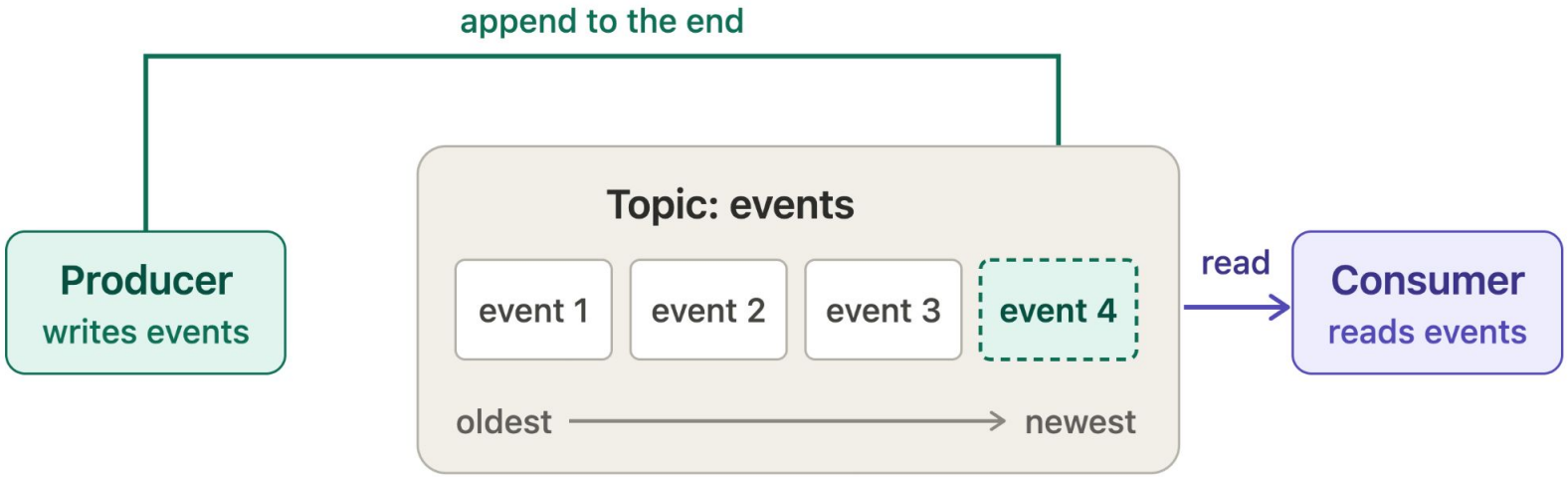
Agenda

- Apache Kafka
- Airflow Apache Kafka provider
- Airflow Listener API
- Kafka Event Producer Plugin
  - What it is?
  - How it can be configured?
  - How it can be used?
- Demos

Apache Kafka Intro

- Kafka is an event streaming platform
- An event is something that has happened in the world or in your business
- Events are organized and stored in topics
- Think of a topic as a folder and the events as files in the folder
- Kafka APIs
  - Producer API
    - Publish/write events to a topic
  - Consumer API
    - Read events from a topic

Kafka Example



Append only: events are never changed or deleted when read.

Airflow Apache Kafka Provider

- Allows Airflow to react to Kafka in different ways
  - Hooks
  - Operators
  - Sensors and Triggers
  - Plugins
    - `EventProducer` plugin
    - version **>= 1.15.0**

Kafka Provider: Hooks

- Producer
  - Creates a Kafka Producer to write to a topic
- Consumer
  - Creates a Kafka Consumer to read from a topic

Kafka Provider: Operators

- `ConsumeFromTopicOperator`
  - Reads messages from a topic in a task and applies a function
- `ProduceToTopicOperator`
  - Writes messages from a task

Kafka Provider: Sensors and Triggers

- `AwaitMessageSensor`
  - The Dag waits for a matching message
- `AwaitMessageTriggerFunctionSensor`
  - The Dag keeps listening and acts on every match
- The sensors use the `AwaitMessageTrigger`
  - Polls the topic inside the triggerer in a loop, so that no workers are blocked and waiting

Airflow Listener API

- Registers hooks that apply a function for every state change event
- Each function contains custom code logic

State Change Events

- Dag Run
  - Running
  - Success
  - Failed
- Task Instance
  - Running
  - Success
  - Failed
  - Skipped

Plugin Overview

- The plugin
  - builds on top of the Listener API and reuses the Kafka Provider hooks
  - Registers a listener per event type
  - Applies a function on every state change event
  - Each function
    - Uses the Producer hook
    - Publishes one message per event to a topic

Message format

RESEARCH | OPEN SOURCE SOFTWARE


```
{
    "schema_version": 1,
    "source": "nyc",
    "event": "dag_run.success",
    "timestamp": "2026-10-30T14:52:53.863374+00:00",
    "dag_id": "demo_dag",
    "run_id": "manual__2026-10-30T14:52:49.322225+00:00",
    "run_type": "manual",
    "logical_date": "2026-10-30T14:52:49+00:00",
    "start_date": "2026-10-30T14:52:52.960270+00:00",
    "end_date": "2026-10-30T14:52:53.863261+00:00",
    "msg": "success"
}
```

Required Configuration

```
[kafka_event_producer]
dag_run_events_enabled = True
task_instance_events_enabled = True
kafka_config_id = kafka_events
topic = airflow.events
```

Event filters

- The amount of events can easily become overwhelming
- Filters can be applied
- All values are comma-separated glob patterns
  - `dag1_*, test_*, …`
- Default to empty, allow all

Dag Run event filters

- Configs
  - `dag_run_dag_id_allowlist`
  - `dag_run_dag_id_denylist`
- Deny takes precedence over allow

Task Instance event filters

- Configs
  - `task_instance_dag_id_allowlist`
  - `task_instance_dag_id_denylist`
  - `task_instance_task_id_allowlist`
  - `task_instance_task_id_denylist`
- Deny takes precedence over allow
- Both checks must pass

TI event filters examples

- Only what's specified is allowed
  - `task_instance_dag_id_allowlist = billing_*`
  - `task_instance_task_id_allowlist = publish`
- Everything allowed except
  - `task_instance_dag_id_denylist = sandbox_*`
  - `task_instance_task_id_denylist = cleanup, *_debug`
- All dags allowed, only task ids are filtered
  - `task_instance_dag_id_allowlist =`
  - `task_instance_dag_id_denylist =`
  - `task_instance_task_id_denylist = cleanup, *_debug`

Demo - Cross deployment dependencies

Overview

- 2 teams
  - New York and London
- Coordination over a shared Kafka topic
- Execution flow
  - Dag1 (NY) -> Dag2 (London) -> Dag3 (NY)

Dag1 (NY)

```
with DAG(
    dag_id="nyc_demo_dag",
    schedule=None,
    start_date=datetime.datetime(2025, 1, 1),
    catchup=False,
) as dag:

    @task
    def task1():
        print("Running task1")

    @task
    def task2():
        print("Running task2")

    @task
    def task3():
        print("Running task3")

    task1() >> task2() >> task3()
```

```python
with DAG(
    dag_id="london_kafka_message_reaction_1",
    schedule=None,
    start_date=datetime.datetime(2025, 1, 1),
    catchup=False,
) as dag:
    wait_for_nyc_demo_dag = AwaitMessageSensor(
        task_id="wait_for_nyc_demo_dag",
        topics=["airflow.events"],
        kafka_config_id="kafka_default",
        apply_function="listener_event_filter.matches_task_event",
        apply_function_kwargs={
            "dag_id": "nyc_demo_dag",
            "task_id": "task1",
        },
        poll_timeout=2.5,
        poll_interval=2,
    )

    @task
    def processing1():
        ...

    wait_for_nyc_demo_dag >> processing1()
```

Dag 2 (London)

```python
with DAG(
    dag_id="nyc_kafka_kafka_message_reaction_2",
    schedule=None,
    start_date=datetime.datetime(2025, 1, 1),
    catchup=False,
) as dag:
    wait_for_london_reaction = AwaitMessageSensor(
        task_id="wait_for_london_reaction",
        topics=["airflow.events"],
        kafka_config_id="kafka_default",
        apply_function="listener_event_filter.matches_task_event",
        apply_function_kwargs={
            "dag_id": "london_kafka_message_reaction_1",
            "task_id": "processing1",
        },
        poll_timeout=2.5,
        poll_interval=2,
    )

    @task
    def processing2():
        ...

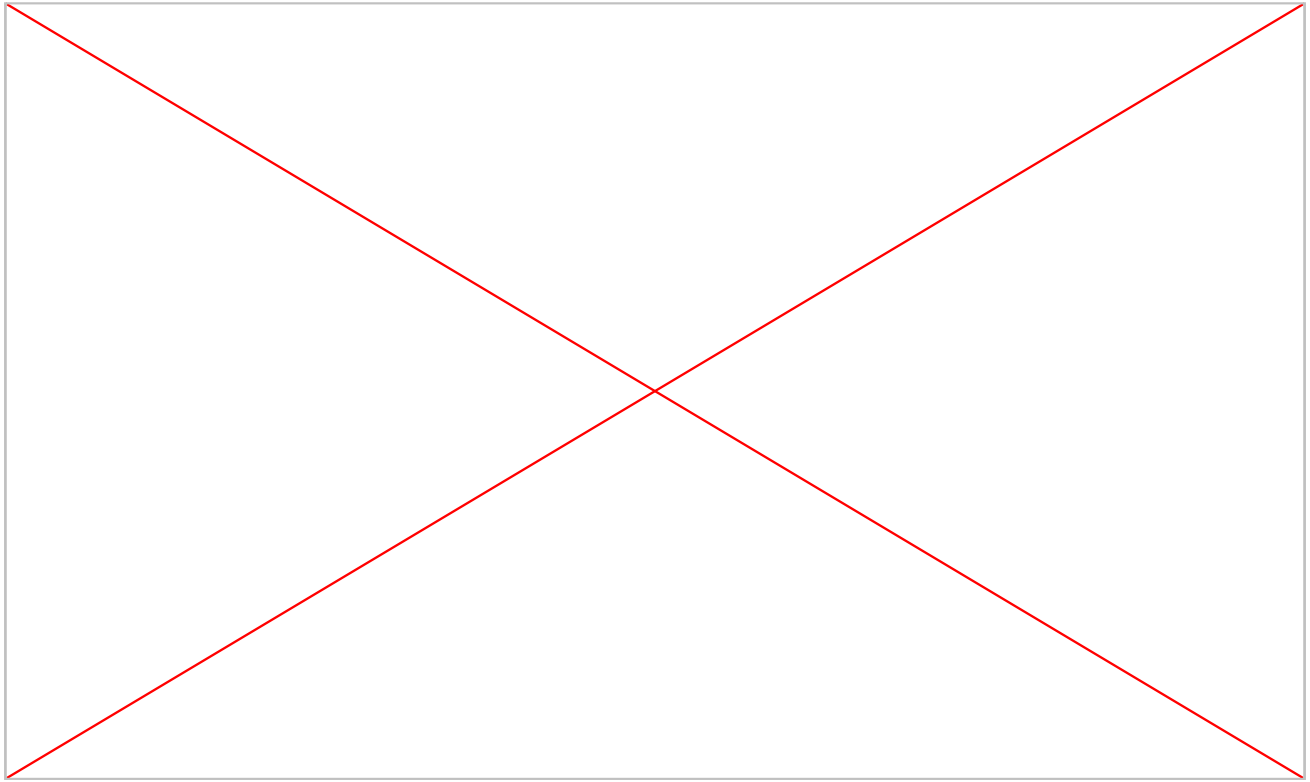
    wait_for_london_reaction >> processing2()
```

Dag 3 (NY)

```python
def matches_task_event(message, dag_id, task_id, event):
    """Match Kafka listener events for a specific dag_id/task_id/event."""
    raw = message.value().decode("utf-8") if message.value() else ""
    try:
        data = json.loads(raw)
    except (TypeError, json.JSONDecodeError):
        return None
    if not isinstance(data, dict):
        return None
    if (
            data.get("event") == event and
            data.get("dag_id") == dag_id and
            data.get("task_id") == task_id
    ):
        return {
            "matched_event": event,
            "matched_dag_id": dag_id,
            "matched_task_id": task_id,
            "run_id": data.get("run_id"),
            "source": data.get("source"),
            "timestamp": data.get("timestamp"),
        }
    return None
```

Match function

Recording



Demo - Multi deployment monitoring

Overview

- Use the messages from the topic for monitoring
- Multiple options
  - Alert on specific events
  - Export metrics from the messages, e.g. counters and histograms
  - Consume the messages from a Dag and alert from Airflow
  - Use an observability store with a kafka integration
  - Store in a table and create dashboards from the queries

Grafana dashboard

- Subscribe a consumer to the topic
- Each message is a JSON that can be parsed
- Create a database table
- Process the message JSON and store it in the table
  - Message -> table row
  - Field -> table column (event, dag_id, task_id, etc.)
- Use the database as a datasource for Grafana
- One panel per DB query

Alerts

- Airflow failure event — Normal — 1 instance — View alert rule
- Failure rate above 5% (15 min) — Normal — 2 instances — View alert rule
- No events from a site for 10 minutes — Normal — 2 instances — View alert rule

New York

Completed (1h)	Failure rate (1h)	In flight	Silent for
27	3.70%	16	0.0 min

Runs per hour



Failure rate per hour



Top failing DAGs

dag_id	failed	runs	fail %	last error
ops_scoring_013	12	669	1.80	botocore.exceptions.ClientError: An error occurred (SlowDown) w
reporting_pricing_031	8	672	1.20	airflow.exceptions.AirflowException: Bash command failed. The cc
finance_enrichment_038	5	28	17.9	MemoryError: worker killed (OOMKilled) while processing partitior
ingest_rollup_047	5	666	0.800	MemoryError: worker killed (OOMKilled) while processing partitior

London

Completed (1h)	Failure rate (1h)	In flight	Silent for
25	0%	5	4.7 min

Runs per hour



Failure rate per hour



Top failing DAGs

dag_id	failed	runs	fail %	last error
reporting_orders_034	4	28	14.3	airflow.exceptions.AirflowException: Bash command failed. The cc
ops_enrichment_009	4	166	2.40	airflow.exceptions.AirflowTaskTimeout: Timeout, PID: 1761
ops_sessions_014	2	166	1.20	MemoryError: worker killed (OOMKilled) while processing partitior
ads_exports_065	2	166	1.20	psycopg2.OperationalError: could not connect to server: Connecti

Thank you! Questions?

RESEARCH | OPEN SOURCE SOFTWARE



