AIRFLOW SUMMIT | ONLINE RECONNECT

Orchestrating the AI Revolution

Self-Healing Airflow: Pipelines that fix themselves

Shrividya Hegde

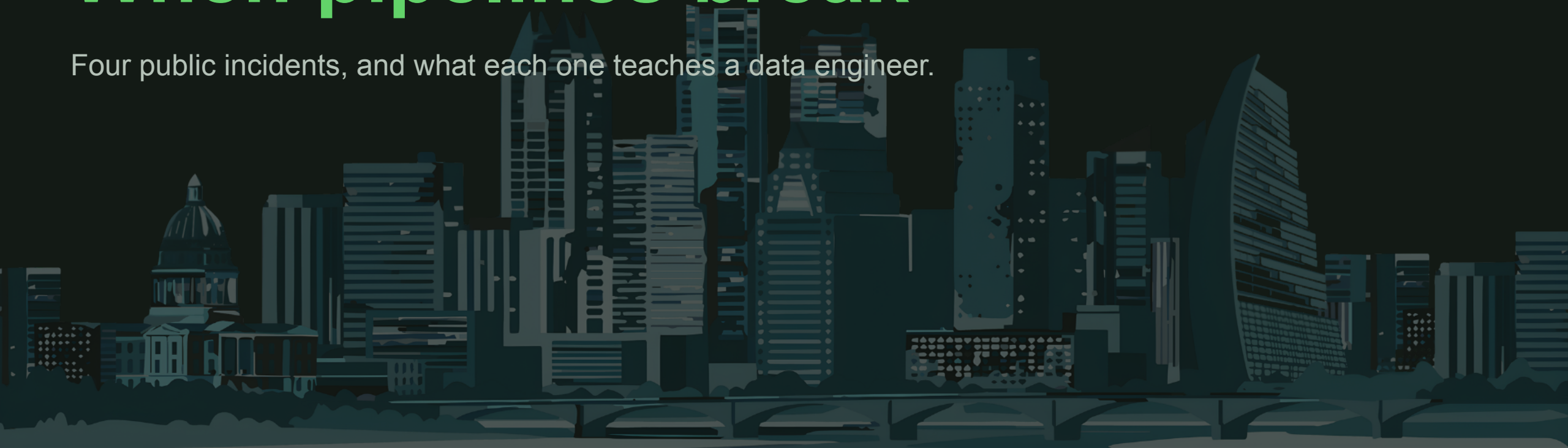
Apache Airflow Champion · Senior AI Data Engineer



AIRFLOW SUMMIT

When pipelines break

Four public incidents, and what each one teaches a data engineer.



CASE 1 · SILENT DATA LOSS

Public Health England, October 2020

15,841

positive Covid-19 cases left out of England's daily figures, 25 Sep – 2 Oct 2020

What happened

The technical issue had caused by the fact that some files containing positive test results exceeded the maximum file size that takes these data files and loads then into central systems. Records past the limit never reached the dashboards or contact tracers, and nothing failed loudly.

LESSON

A task that "succeeds" with missing data is the worst failure. Check row counts in vs. out, and fail the task when they don't match.

Sources: PHE statement and The Register, 5 Oct 2020

CASE 2 · BAD DATA IN AN AI PIPELINE

Unity Software, 2022

~$110M

estimated 2022 revenue impact, as Unity reported on its Q1 2022 earnings call

What happened

Unity's machine-learning ad-targeting tool (Audience Pinpointer) lost accuracy, and Unity said it lost the value of part of its training data, partly from ingesting bad data from a large customer. The first visible symptom was revenue slowing.

LESSON

AI pipelines need data-quality gates before training, and alerts on the data, not just on task status. Unity's own fix: monitoring, alerting and recovery systems.

More examples for bad data quality issues: https://montecarlo.ai/blog-the-cost-of-poor-data-quality

Source: Unity Software Q1 2022 earnings call, 10 May 2022

CASE 3 · A BROKEN CONTRACT SHIPPED EVERYWHERE

CrowdStrike, July 2024

8.5M

Windows devices crashed, by Microsoft's estimate

What happened

A content update (Channel File 291) gave the sensor 20 input fields when it expected 21. The out-of-bounds read resulted in system crash.

LESSON

Validate the contract at the boundary before data ships, and never auto-retry a permanent error. Roll changes out in stages.

Source: CrowdStrike Root Cause Analysis, Channel File 291, 6 Aug 2024

CASE 4 · RECOVERY THAT CAUSED A SECOND OUTAGE

Google Cloud, June 2025

~7.5 h

incident duration, 10:51 to 18:18 PT on 12 June 2025

What happened

A policy record with unintended blank fields reached Service Control, which hit a null pointer and crash-looped worldwide. As it restarted in us-central1, it had no randomized exponential backoff and overloaded Spanner: a herd effect that slowed recovery.

LESSON

Retries without backoff and jitter turn recovery into another outage. Back off, cap the delay, and stop calling what is down.

Source: Google Cloud Service Health incident report, June 2025

THE PATTERN

Four failures, four Airflow defences

Incident	Failure mode	Airflow 3 defence
Public Health England	Silent partial load	Row-count checks that raise AirflowFailException
Unity	Bad input poisons a model	Data-quality gate before training, routed alerts
CrowdStrike	Contract mismatch	Fail fast on permanent errors, validate at the boundary
Google Cloud	Retry storm, herd effect	Exponential backoff, max_retry_delay, a circuit breaker

WITHOUT SELF-HEALING

Data teams firefight by hand

67

data incidents a month, on average, per respondent's organisation

15 h

average time to resolve each one

74%

say business stakeholders spot the issue first, all or most of the time

Gartner puts the average cost of poor data quality at **$12.9M a year** per organisation (2020 survey of data-quality vendors' customers).

Sources: Monte Carlo / Wakefield, State of Data Quality 2023 (200 data professionals); Gartner

WITHOUT SELF-HEALING

The on-call engineer is the retry policy

Alert fatigue

Pages for blips teach people to ignore the page that matters.

Retry storms

Blind, instant retries pile onto a system that is already down.

Silent hangs

A stuck task never raises, so nothing fires at all.

DEFINITION

A self-healing pipeline detects a failure, works out what kind it is, recovers on its own when that is safe, and escalates with context when it isn’t.

1

Detect

Notice the failure, including the silent ones.

2

Diagnose

Transient, permanent, stuck, or late?

3

Recover

Retry, resume, clear or skip, within a cap.

4

Escalate

Page a human, with context, at the right volume.

TWO FAMILIES

Traditional and smart self-healing

Traditional

Rules you write in advance

- Retries with exponential backoff
- Fail-fast and skip exceptions
- Rule-based retry policies (3.3)
- Callbacks and notifiers
- Timeouts and Deadline Alerts
- Watchdogs and circuit breakers

Deterministic, low cost , auditable.

Smart

A model helps decide at failure time

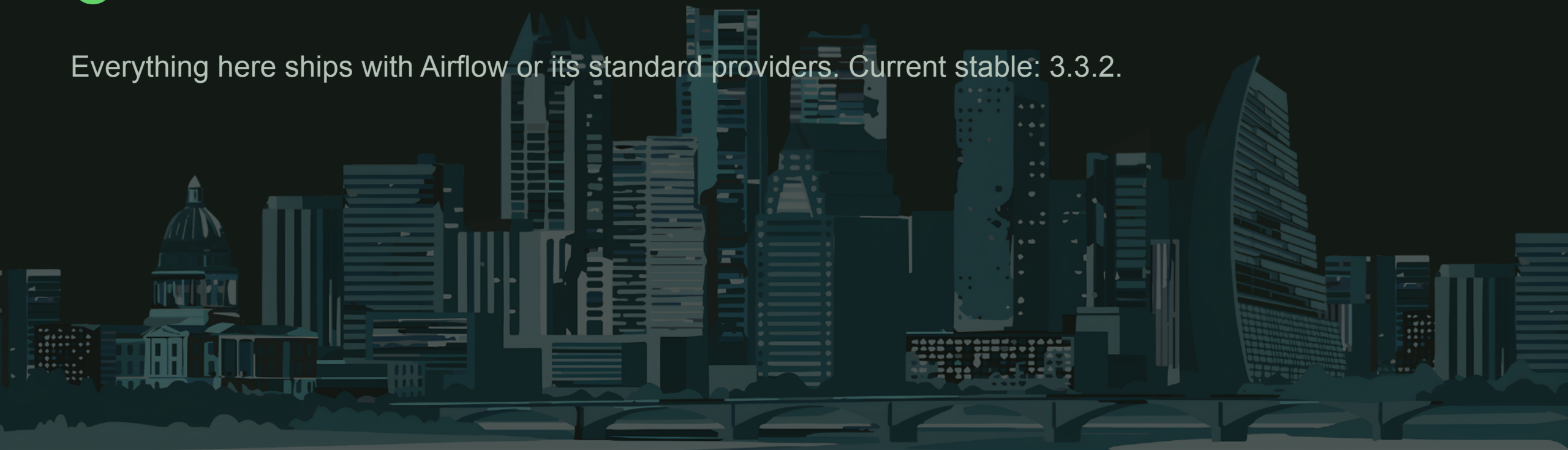
- LLMRetryPolicy
- ClassifierRetryPolicy
- LLM-written incident summaries
- Agents that diagnose, with human approval

Handles errors nobody wrote a rule for.

Smart sits on top of traditional, never instead of it.

Traditional self-healing in Airflow 3

Everything here ships with Airflow or its standard providers. Current stable: 3.3.2.



THE MAP

The self-healing ladder in Airflow 3

Airflow feature	Question it answers
retries + exponential backoff	Can time alone fix it?
AirflowFailException	Should we stop right now?
retry_policy (ExceptionRetryPolicy)	Which errors deserve a retry?
on_*_callback + Notifiers	Who needs to know?
execution_timeout, Deadline Alerts	Is it stuck, or late?
Heartbeat timeout + watchdog DAG	Is it a zombie?
@task.short_circuit breaker	Should we stop calling?
Task State Store	Can a retry resume instead of redo?

Back off, don’t hammer

```
default_args = {
    "retries": 4,
    "retry_delay": timedelta(minutes=1),
    "retry_exponential_backoff": 2.0,  # 3.2+: a multiplier
    "max_retry_delay": timedelta(minutes=15),
}
```



Why back off

Each wait is longer than the last, capped by max_retry_delay.



Global cap

[core] max_task_retry_delay caps every task (default 24 h).

Retry only what can heal

Transient → retry

Timeouts, 429 rate limits, 503s, a restarted node. Time fixes these.

Permanent → fail now

Revoked credentials, schema drift, malformed input. Attempt five fails like attempt one.

```
raise AirflowFailException("401: API key revoked")   # fails now, remaining retries skipped
raise AirflowSkipException("no new files today")    # marks the task skipped, not failed
```

THE PRICE OF RETRIES Every retry reruns the task. Overwrite or upsert on a key, never blindly append.

Decide per error, not per task

```python
from airflow.sdk import ExceptionRetryPolicy, RetryAction, RetryRule

policy = ExceptionRetryPolicy(rules=[
    RetryRule(exception=RateLimited, action=RetryAction.RETRY,
              retry_delay=timedelta(seconds=20)),
    RetryRule(exception=Unauthorized, action=RetryAction.FAIL),
])

@task(retries=4, retry_policy=policy)
def fetch_payments(): ...
```

- Rules run in order; first match wins
- No match: normal retries and backoff
- retries stays the ceiling
- Every decision is written to the task log
- Need more? Subclass RetryPolicy and write evaluate()

NEW IN 3.3 (AIP-105) Works on every operator and decorator. AirflowFailException still wins.

Know what happened, every time

Callback	Fires when	Use it to
on_retry_callback	a retry is queued (policy retries too)	log quietly, count it, never page
on_failure_callback	retries are exhausted	alert, start a deeper heal
on_success_callback	the task succeeds	auto-resolve; flag “passed on try 3”
on_skipped_callback	AirflowSkipException is raised	track work a breaker skipped
on_execute_callback	right before the task runs	stamp start times, mark in-flight

TWO RULES Pass a Notifier (e.g. SlackNotifier) or a list. Keep callbacks tiny: one that raises hides the original failure.

RUNG 4 · TIMEOUTS

A time box at every level

execution_timeout

Per attempt. Kills a hang and raises AirflowTaskTimeout, so silence becomes a normal failure that retries.

dagrun_timeout

Per run. Bounds the whole DAG run, so one stuck branch can’t hold it open forever.

Sensor timeout

With mode="reschedule" the sensor frees its worker slot between checks; timeout ends the wait.

Late is a failure too

```
with DAG(
    dag_id="payments_daily_summary",
    deadline=DeadlineAlert(
        reference=DeadlineReference.DAGRUN_QUEUED_AT,
        interval=timedelta(minutes=15),
        callback=AsyncCallback(
            SlackWebhookNotifier,
            kwargs={"text": "{{ dag_run.dag_id }} is late"},
        ),
    ),
): ...
```

- SLAs were removed in Airflow 3.0
- Deadline Alerts replace them: 3.1, still experimental
- References: queued time, logical date, a fixed time, or average runtime
- SyncCallback added in 3.2; a Deadlines page in 3.3

Source: Airflow docs, Deadline Alerts how-to

Two ways to be stuck

Zombie: the worker is gone

Still marked running after its worker died (OOM-killed, pod evicted). The scheduler fails or retries it after [scheduler] task_instance_heartbeat_timeout, renamed in 3.0 from scheduler_zombie_task_threshold.

Stuck: alive but going nowhere

Queued or running far longer than it should, with a heartbeat. task_queued_timeout covers the queue; nothing in core covers a live hang without execution_timeout.

THE GAP The scheduler catches some of these. A watchdog DAG catches the rest.

Your on-call routine, written as a DAG

Detect
REST API: tasks queued or running too long

Clear
clear them so they run again

Verify
next sweep: did it finish?

Escalate
still stuck after 2 clears: page

Why the REST API

In Airflow 3, task code can no longer use the metadata database directly. The watchdog uses the API with a token from a connection.

Cap the healing

Count auto-clears per task run. Two, then a human. Healing with no limit is a loop that hides a real problem.

Stop calling what is down

250

extra calls into a system that is already down: 50 tasks × 5 retries

Closed
calls flow

→

Open
skip fast, cool down

→

Half-open
one probe

```
@task.short_circuit
def breaker_allows():
    return BREAKER.allow()

breaker_allows() >> load_to_crm()
```

WATCH OUT Skipped is not succeeded: make sure skipped work is picked up once the breaker closes.

Retry without redoing

```python
@task(retries=3)
def run_spark_job(**context):
    tss = context["task_state_store"]
    job_id = tss.get("job_id")
    if job_id is None:
        job_id = spark_client.submit_job(...)
        tss.set("job_id", job_id, retention=NEVER_EXPIRE)
    # a retry reattaches instead of resubmitting
    return spark_client.wait_for_completion(job_id)
```

- Key/value state per task instance
- Unlike XCom, it survives retries
- Resume from a checkpoint: last page, job ID, row count
- SparkSubmitOperator (Spark provider 6.2.0+) already uses it

WHY IT HEALS A retry after a worker crash no longer leaves two copies of a long job running.

Source: Airflow 3.3 docs, Task State Store (AIP-103)

RUNG 8 · ALERT ROUTING

Page for impact, not for noise

Level	Example	Goes to
Info	Retry succeeded	Logs only
Warning	Healed after retries, watchdog cleared it, breaker opened	Slack channel
Critical	Retries exhausted, watchdog gave up	PagerDuty, with a dedup key



In the DAG

Only connection IDs: alert_slack, alert_pagerduty, pydanticai_default.



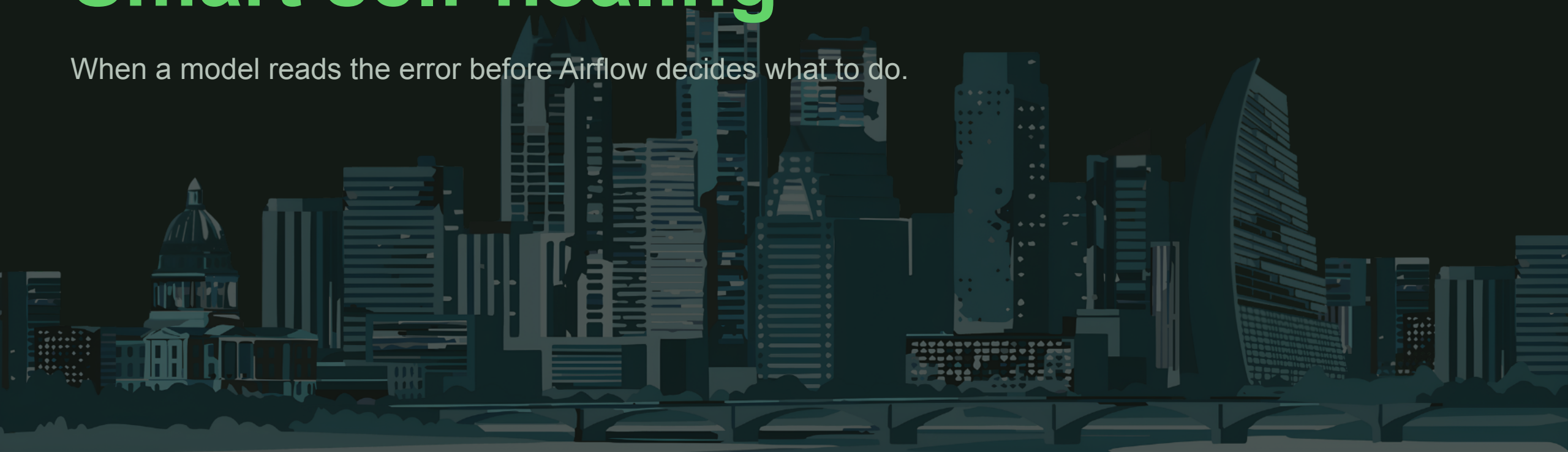
In the secrets backend

Webhook URLs, routing keys, LLM API keys. Rotate without a deploy.

PART 3

Smart self-healing

When a model reads the error before Airflow decides what to do.



THE PROBLEM

Static retries are blind



429 rate limit

Retried too soon. Throttled again.



401 revoked key

Retried pointlessly. Burns every attempt, delays the page.



Schema drift

Nobody wrote a rule for it. Falls through to plain retries.

Same delay. Every error. Every time.

RULES HELP …but only for the errors you predicted. The long tail is where 2 AM pages come from.

THE BUILDING BLOCK

The Common AI provider, 1.0

Built on Pydantic AI

One connection type for OpenAI, Anthropic, Google, Bedrock, Ollama and more.

The connection picks the model

Change vendor or model in the connection, not the DAG.

AI tasks, native

@task.llm, @task.agent, @task.llm_batch, plus LLM and classifier retry policies.

```
pip install 'apache-airflow-providers-common-ai[anthropic]'
```

1.0 released 19 Oct 2026. Installs on Airflow 2.11+; retry policies need Airflow 3.3.

Source: Apache Airflow blog, “Airflow AI Provider 1.0”, Oct 2026

Let a model read the error

```python
from airflow.providers.common.ai.policies.retry import (
    LLMRetryPolicy,
)

llm_policy = LLMRetryPolicy(
    llm_conn_id="pydanticai_default",
    timeout=30.0,          # then fall back
    fallback_rules=RULES,  # the rung-2 rules
)

@task(retries=4, retry_policy=llm_policy)
def fetch_payments(): ...
```

1. **Task fails**
   exception class and message
2. **Redact**
   registered secrets masked first
3. **Model classifies**
   category, should_retry, suggested_delay_seconds, reasoning
4. **Decide**
   retry after N seconds, or fail now
5. **Log**
   the reasoning lands in the task log

Source: apache-airflow-providers-common-ai docs, LLM Retry Policies

A model names it. Your table decides.

Category	Action
rate_limit	retry after 60 s
network	retry after 10 s
transient	retry after 30 s
auth	fail
data	fail
resource	fail
permanent	fail

You own the decision

The model only picks a category. Retry, delay and the confidence bar come from your table.

Your own taxonomy

Add warehouse_suspended (retry in 30 s) or schema_drift (fail). Set min_confidence; below it, a fallback_policy decides.

Defaults: DEFAULT_CATEGORIES, Common AI provider docs

SMART RETRIES

Pick your level of trust

1 · Rules

ExceptionRetryPolicy. Match on exception type. No model. The floor.

2 · Classifier

The model picks a category. Your table decides retry and delay.

3 · LLM policy

The model decides retry and delay, and explains why.

```
ClassifierRetryPolicy(llm_conn_id="pydanticai_default", min_confidence=0.8,
    fallback_policy=LLMRetryPolicy(llm_conn_id="pydanticai_default"),
    fallback_rules=RULES)
```

Start at the floor. Add one layer of trust at a time.

BEYOND RETRIES

Let a model diagnose. Keep a human on the fix.

@task.llm

One prompt in, one answer out. Turn a stack trace into two plain sentences for the Slack alert.

@task.agent + toolsets

Read-only SQL, hooks, MCP servers. Mark a tool approval_required() and the agent pauses for a yes (Airflow 3.3).

HITL operators (3.1+)

ApprovalOperator, HITLBranchOperator: a person picks the recovery path in the Airflow UI.

RULE OF THUMB Automate the diagnosis. Gate anything that writes, posts or deletes behind a human.

Guardrails for a model in the failure path

Safe by design

- No tools, no data access: it only sees the exception
- 30 s timeout, then your fallback rules
- Registered secrets masked before sending
- Can never retry beyond retries

Watch out

- Runs on every failure, outside usage_limits
- Masking is not PII detection: add a redactor
- Sensitive errors? Use a local model (Ollama)
- Confidence ≠ accuracy; delays aren't clamped to max_retry_delay

Source: Common AI provider docs, LLM Retry Policies: guardrails and cost notes

Make the pipeline easy to heal

01 Idempotent first

Retries and clears rerun the task. Upsert or overwrite on a key.

02 Classify, then retry

Transient waits; permanent fails fast. Write the rules down.

03 Validate at the boundary

Row counts and schema contracts that fail loudly.

04 Test the failure path

Break it in staging. If you've never seen it heal, it's a hope.

AI tasks fail differently

01 429s are normal

Rate limits and provider outages are routine. Give them their own retry rule or policy.

02 Don’t pay twice

durable=True on @task.agent replays finished model and tool calls on a retry.

03 Validate the output

Set output_type to a Pydantic model; a malformed answer is retried inside the task.

04 Gate the side effects*

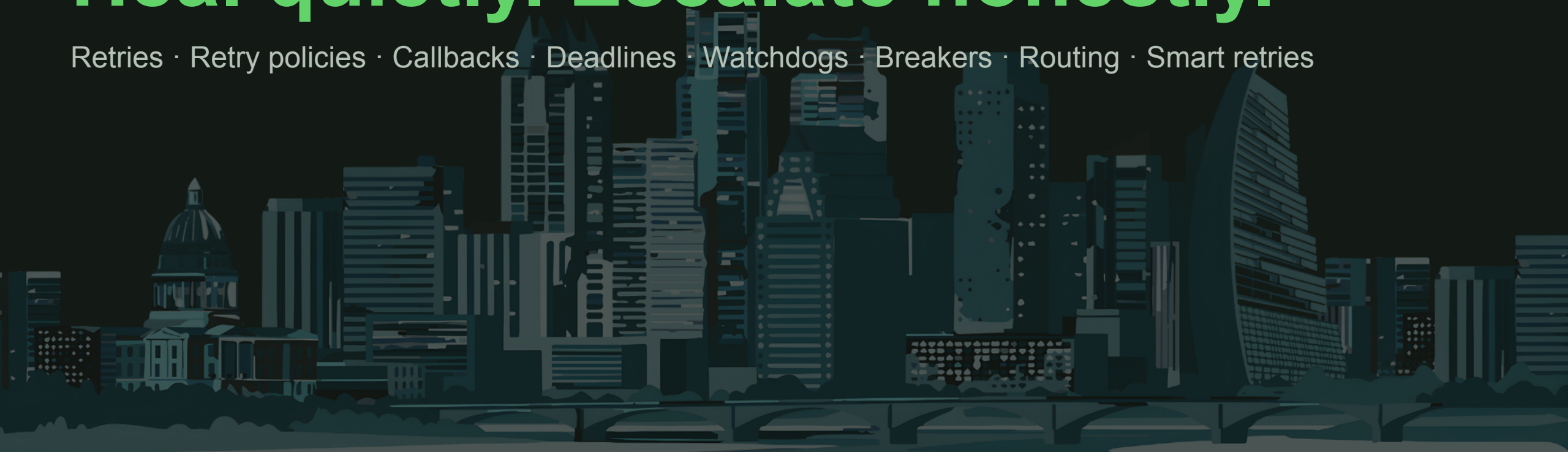
approval_required() tools and HITL review before an agent writes, posts or deletes.

**Note: durable=True and approval_required() cannot be used on the same agent task*

AIRFLOW SUMMIT

Heal quietly. Escalate honestly.

Retries · Retry policies · Callbacks · Deadlines · Watchdogs · Breakers · Routing · Smart retries



REFERENCES

Sources

- PHE: statement of 4 Oct 2020; The Register, “test and trace”, 5 Oct 2020
- Unity Software Q1 2022 earnings call, 10 May 2022
- CrowdStrike, External Technical Root Cause Analysis: Channel File 291, 6 Aug 2024; Microsoft device estimate
- Google Cloud Service Health, Service Control incident report, 12–14 Jun 2025
- Monte Carlo / Wakefield Research, State of Data Quality 2023; Gartner data-quality research (2020)
- Apache Airflow docs 3.3.2: Tasks (retry policies), Callbacks, Deadline Alerts, Task State Store, HITL, release notes
- apache-airflow-providers-common-ai docs: LLM Retry Policies; Airflow blog, “Airflow AI Provider 1.0”, Oct 2026



Thank you.

Shrividya Hegde · Apache Airflow Champion · Senior AI Data Engineer

Demo repo: https://github.com/Shrividya/Airflow-Summit-Reconnect-Demo

Portfolio: https://shrividya.github.io