AIRFLOW. SUMMIT

Taming AI workloads in Apache Airflow

Dag patterns to avoid infrastructure instability

Jason(Zhe-You) Liu

Astronomer • github.com/jason810496 • linkedin.com/in/zhe-you-liu

Any workload that outlives its Airflow task.

Data processing
Spark, Flink, dbt Cloud

AI / ML
fine-tuning, batch inference, training

Infrastructure
cluster provisioning, EMR, Dataproc

The pattern is generic. AI workloads just make it expensive to get wrong.

01 / 06

The problem

How a task actually runs, and why SIGTERM lies.

A STORY YOU HAVE PROBABLY LIVED

Your GPU job didn't fail. Airflow killed it.

THE JOB

8× H100 fine-tune
14 hours planned · ~$30/hr

HOUR 11

Worker pod evicted.
Not the training job — the *watcher*.

THE RESULT

operator.on_kill fired. Job cancelled.
~$330 burned. Next run starts at zero.

Representative scenario, illustrative numbers — not a specific customer.

A task instance pushes its own liveness.

Push-based. Airflow never asks the task how it is doing. **Silence is interpreted as death.**

What operator.on_kill usually contains.

finetune_operator.py — the shape everyone ships

```python
class SubmitJobOperator(BaseOperator):

    def execute(self, context):
        self.job_id = client.submit(self.spec)
        client.wait_for_completion(self.job_id)

    def on_kill(self):
        client.cancel(self.job_id)      # ← this line
```

One line. Nobody would review it and object. The defect is in **what calls it**.

FOUR CAUSES. ONE SIGNAL.

ALL FOUR PATHS VERIFIED IN SUPERVISOR.PY

Mark Failed or clear
404 / 409 / 410

API server unreachable
3 failed heartbeats

k8s drain or evict
SIGTERM from outside

Preemption by the engine
job moves under you

↓ ↓ ↓ ↓

`operator.on_kill()`
no reason · no flag · no ask

Only Mark Failed / clear **wants the job dead.**

Default: **15 seconds** of API-server silence is enough.

AIRFLOW CANNOT TELL THE DIFFERENCE.

If you cancel: false kills

A drain, or a 15-second network blip, murders a healthy 11-hour job.

If you don't: leaked jobs

"Mark Failed" leaves $30/hr running until somebody notices.

Neither is a bug. The information needed to choose correctly never reaches the operator.

Two questions

Can the watcher dodge operator.on_kill's blind spot — and still stop the job on purpose?

QUESTION ONE

Can the watcher dodge operator.on_kill?

What has to change is **what's listening for SIGTERM in the first place.**

QUESTION TWO

Then how do we explicitly stop the job?

Something else has to carry the **"the user wants this dead"** signal.

03 / 06

Defer

Where the watch belongs, and why.

Operator vs trigger: two different runtimes.

Operator runs in a subprocess on a Worker

execute()	operator.on_kill()
blocks for hours	fires on any SIGTERM
supervised, heartbeated	no reason attached

self.defer()

Trigger runs as an async coroutine on a Triggerer

rebuilt from (classpath, kwargs)	no subprocess operator.on_kill() is unreachable

Inside the trigger.

finetune_trigger.py — the constructor

```python
class WatchJob(BaseTrigger):

    def __init__(self, job_id: str):
        self.job_id = job_id        # the whole state there is
```

No execute(), no operator, no task context — only **constructor arguments**.

The serialized tuple is the recovery snapshot.

finetune_trigger.py — what crosses the process boundary

```python
class WatchJob(BaseTrigger):

    def serialize(self) -> tuple:
        return "finetune_trigger.WatchJob", {"job_id": self.job_id}
               # classpath                  kwargs
```

The *only* thing that survives a rebuild. Anything else is gone after a restart.

Inside the trigger.

finetune_trigger.py — the watch loop

```python
class WatchJob(BaseTrigger):

    async def run(self):
        while True:
            status = await check(self.job_id)
            if status.is_terminal():
                yield TriggerEvent({"status": status})
                return
            await asyncio.sleep(30)
```

An **async generator**. This poll costs a coroutine, not a worker slot.

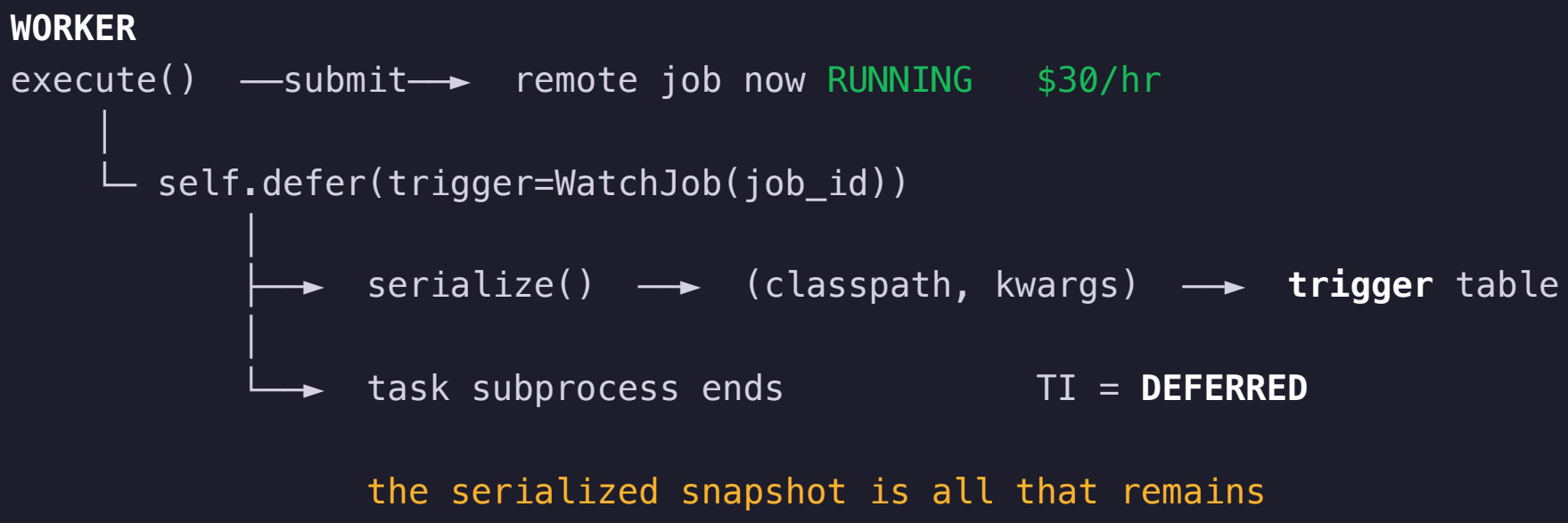
The operator hands off to it.

finetune_operator.py — calling the trigger

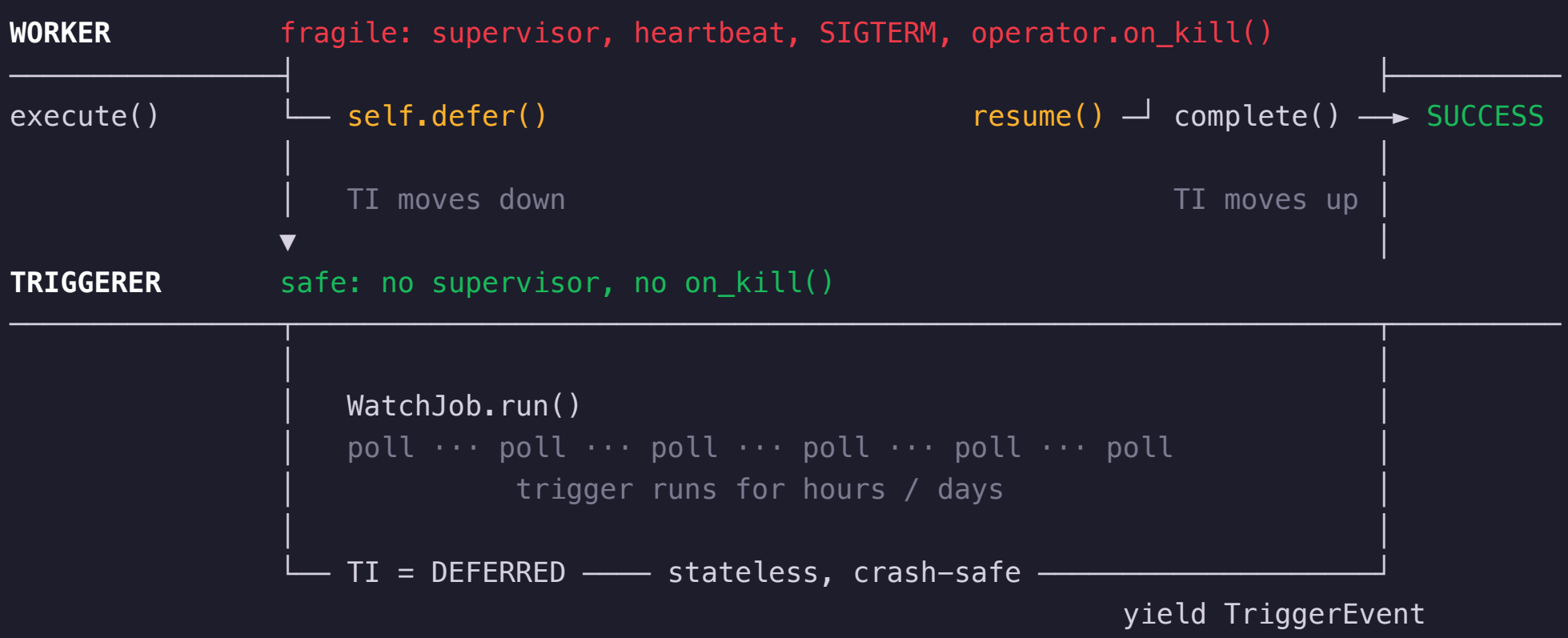
```python
class SubmitJobOperator(BaseOperator):

    def execute(self, context):
        job_id = client.submit(self.spec)
        self.defer(trigger=WatchJob(job_id=job_id), method_name="complete")

    def complete(self, context, event):
        return event["status"]
```

One call, right after submit. Everything from here happens **without this operator's process**.

What defer actually does.

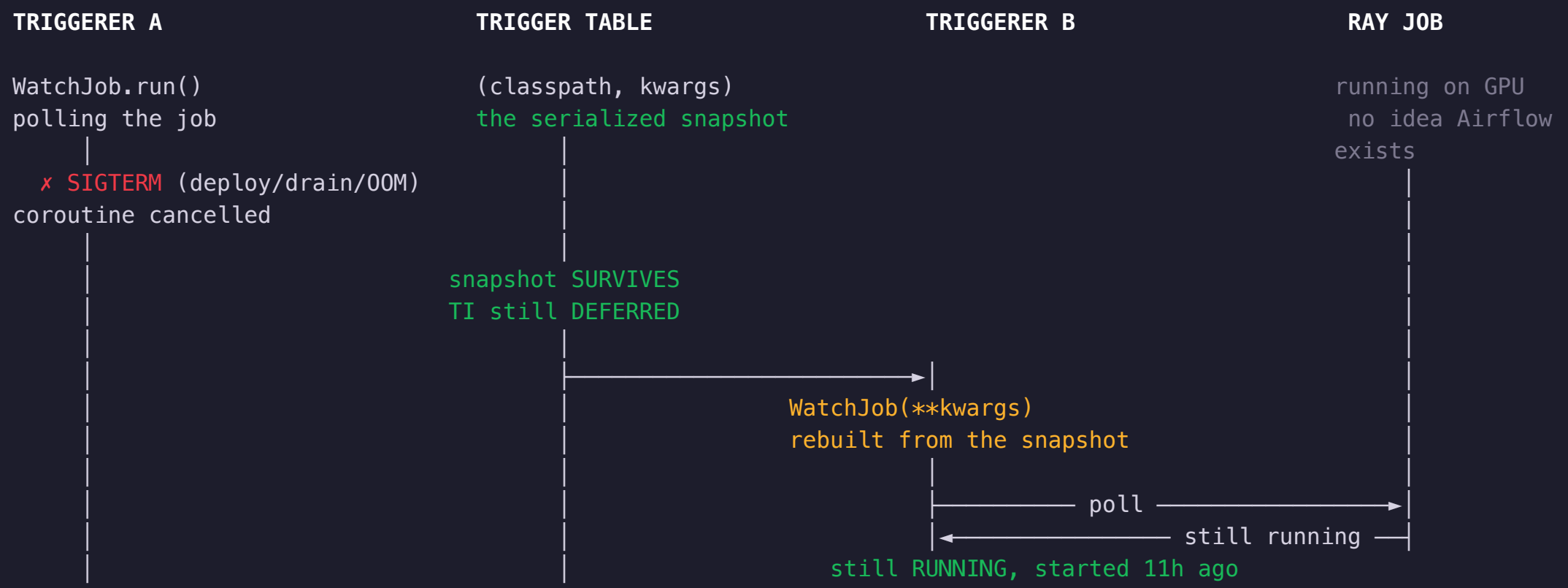


Where the task instance lives at each phase.



The triggerer is stateless and crash-safe.

THE COROUTINE IS STATELESS. RECOVERING COSTS NOTHING TO LOSE.



THE CENTRE OF THE TALK

The task instance never left DEFERRED.

It did not fail. It did not retry. operator.on_kill never fired.
An entire Airflow component died and the Dag did not notice.

A trigger is already a watcher. The statelessness this pattern needs is not a discipline you impose — it is a *precondition the runtime enforces*.

QUESTION ONE — ANSWERED

Can the watcher dodge operator.on_kill?

Yes. Defer moves the TI to the triggerer. No subprocess, no SIGTERM, no on_kill.

QUESTION TWO — STILL OPEN

Then how do we explicitly stop the job?

The watcher is safe, but we lost the cancel path entirely. Something else has to carry the **"the user wants this dead"** signal.

The cancel path

Submit, watch, cancel — and a @teardown that carries intent.

SUBMIT. WATCH. CANCEL.

Three tasks because they have three different **failure semantics**. Retrying watch must be free, retrying submit must be a no-op, cancel must fire on intent and nothing else.

Use @teardown as the cancel.

Airflow docs — setup & teardown howto

```
create_cluster >> run_query >> delete_cluster.as_teardown(setups=create_cluster)
```

dags/finetune_dag.py — our nouns

```
submit >> watch >> cancel.as_teardown(setups=submit)
```

Your remote AI job is the cluster. Three operators, one @teardown, since 2.7.

Why @teardown discriminates correctly.

EVENT	WATCH REACHES	TEARDOWN FIRES?	REMOTE JOB
triggerer redeployed	DEFERRED	no	LIVES ✓
API server blip → SIGTERM	DEFERRED	no	LIVES ✓
worker evicted (task deferred)	DEFERRED	no	LIVES ✓
user clicks "Mark Failed"	FAILED	YES	cancelled ✓
user marks Dag run failed	FAILED	YES	cancelled ✓
job genuinely fails	FAILED	YES	no-op ✓
job succeeds	SUCCESS	YES	no-op ✓

A signal handler fires on process events. A teardown fires on task-state events.

Only task state carries intent.

Start from the anti-pattern.

finetune_operator.py — before

```python
class SubmitJobOperator(BaseOperator):

    def execute(self, context):
        self.job_id = client.submit(self.spec)
        client.wait_for_completion(self.job_id)

    def on_kill(self):
        client.cancel(self.job_id)      # ← the line everyone ships
```

One operator. One process. One hook deciding life or death.

Make the watch deferrable.

finetune_operator.py — building the pattern

```python
class SubmitJobOperator(BaseOperator):

    def execute(self, context):
        job_id = client.submit(self.spec)
        self.defer(trigger=WatchJob(job_id=job_id), method_name="complete")

    def on_kill(self):
        client.cancel(self.job_id)     # ← unreachable while DEFERRED
```

The subprocess exits between polls, so the hook has no process to fire in.
This step buys **kill discrimination**, not the worker slot everyone assumes.

Add the @teardown for cancellation.

dags/finetune_dag.py — building the pattern

```
submit = SubmitJob(
    task_id="submit",
    job_name=name,
    spec=spec,
    deferrable=True,
    wait_for_termination=True,
).as_setup()
cancel = CancelJob(task_id="cancel", job_name=name)

submit >> cancel.as_teardown(setups=submit)
```

Cancellation gets a home. as_teardown() fires on **task state**, not signals.

End the task in the trigger.

END_FROM_TRIGGER=TRUE · AIRFLOW.TRIGGERS.BASE · AIRFLOW 2.10+

finetune_trigger.py — how the watch hands back

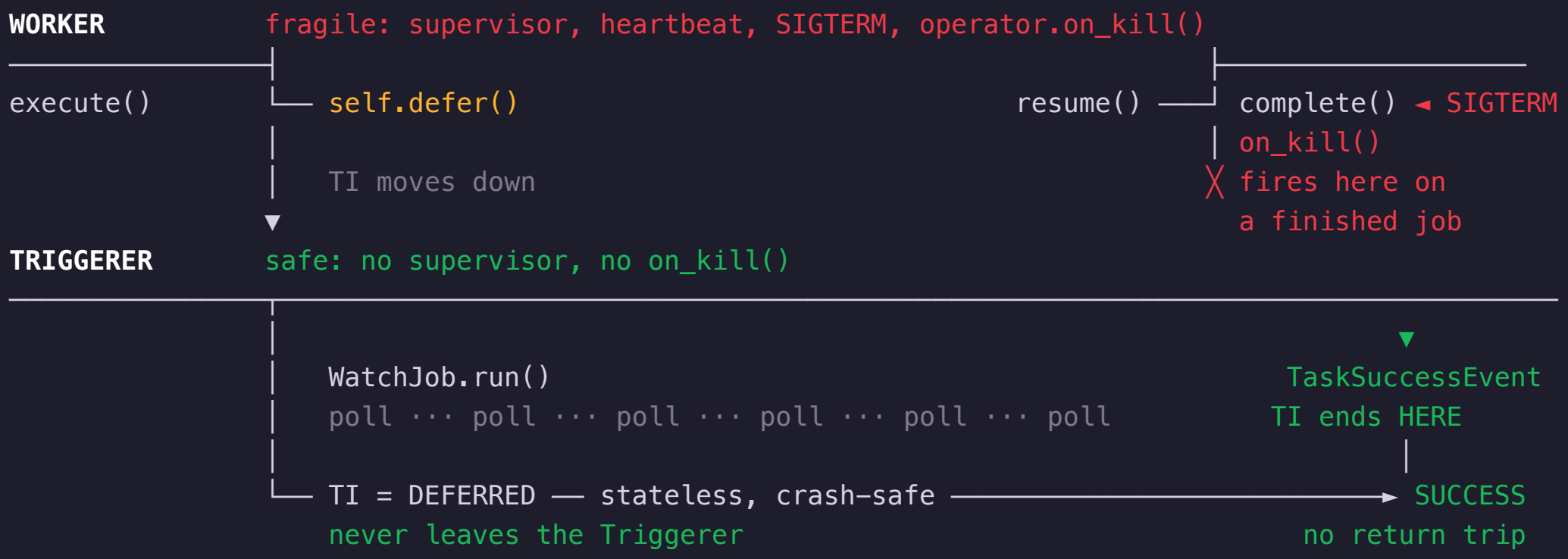
```python
    async def run(self):
        status = await watch(self.job_id)

        if self.end_from_trigger:
            yield TaskSuccessEvent()                # ends here
        else:
            yield TriggerEvent({"status": status})  # goes back to a worker
```

TriggerEvent ⟶ worker resumes ⟶ operator.complete() ⟶ SUCCESS
✗ a fresh subprocess: supervisor · heartbeat · SIGTERM
✗ operator.on_kill is reachable again in this window

TaskSuccessEvent ⟶ the triggerer sets the final state ⟶ SUCCESS
✓ no worker, no subprocess, no hook that can fire

Skip the last mile entirely.

END_FROM_TRIGGER=TRUE ELIMINATES THE RETURN TRIP TO THE WORKER



Group the three into one unit.

dags/finetune_dag.py — building the pattern

```
with TaskGroup(group_id=group_id) as g:
    name = deterministic_name(group_id)

    submit = SubmitJob(
        task_id="submit",
        job_name=name,
        spec=spec,
        deferrable=True,
        wait_for_termination=True,
    ).as_setup()
    cancel = CancelJob(task_id="cancel", job_name=name)

    submit >> cancel.as_teardown(setups=submit)
```

One group_id in, one deterministic name computed once, shared by all three tasks.

The factory. 2.7 through 3.2 — and after.

dags/finetune_dag.py — building the pattern

```python
def resilient_ai_job(*, group_id, spec):
    with TaskGroup(group_id=group_id) as g:
        name = deterministic_name(group_id)

        submit = SubmitJob(
            task_id="submit",
            job_name=name,
            spec=spec,
            deferrable=True,
            wait_for_termination=True,
        ).as_setup()
        cancel = CancelJob(task_id="cancel", job_name=name)

        submit >> cancel.as_teardown(setups=submit)
    return g
```

dags/finetune_dag.py — usage

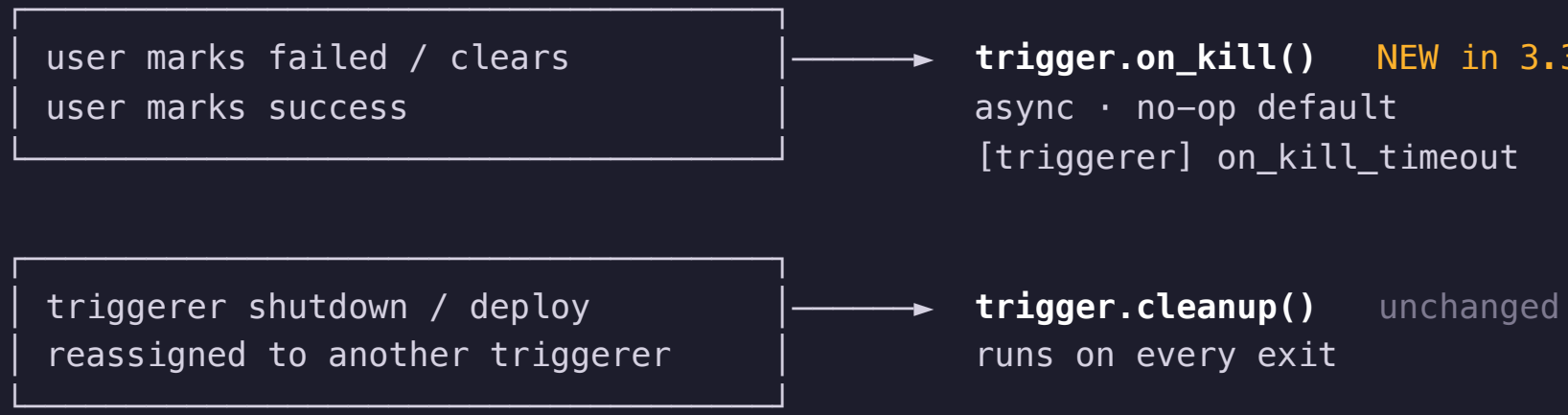
```python
resilient_ai_job(group_id="finetune", spec=FINETUNE) >> evaluate
```

3.3 collapses it all

One operator, one trigger, trigger.on_kill does the rest.

3.3.0 introduces the trigger.on_kill hook.

PR #65590 · MILESTONE 3.3.0



The trigger itself can now tell a real kill from infrastructure noise.
So submit, watch and cancel collapse into **one operator call**.

Authoring the whole pattern in one operator, since 3.3.0.

ONLY WHEN THE PROVIDER'S TRIGGER IMPLEMENTS TRIGGER.ON_KILL

dags/finetune_dag.py — the whole task

```
SubmitJob(
    task_id="finetune",
    job_name=DETERMINISTIC_NAME,
    deferrable=True,
    wait_for_termination=True,
)
```

- All a Dag author writes. Nobody wrote a CancelJob task.
- An **explicit kill** — Mark Failed, clear, mark success — is handled inside the provider's trigger by trigger.on_kill().

Inside execute().

finetune_operator.py — shipped by the provider

```python
class SubmitJobOperator(BaseOperator):

    def execute(self, context):
        job_id = client.submit(self.spec)
        self.defer(trigger=WatchJob(job_id=job_id))
```

No method_name, no complete() — the trigger ends the task directly.

Inside the trigger, both hooks.

finetune_trigger.py — shipped by the provider

```python
class WatchJob(BaseTrigger):

    def __init__(self, job_id, *, end_from_trigger=True):
        ...

    async def run(self):
        while True:
            status = await check(self.job_id)
            if status.is_terminal():
                if self.end_from_trigger:
                    yield TaskSuccessEvent()
                else:
                    yield TriggerEvent({"status": status})
                return
            await asyncio.sleep(30)

    async def on_kill(self) -> None:
        await self.client.stop_job(self.job_id)
```

trigger.cleanup() vs trigger.on_kill().

RECAP — EVERY TRIGGER EXIT, MAPPED TO THE HOOK IT ACTUALLY REACHES

trigger.cleanup()

- triggerer shutdown or deploy
- reassigned to another triggerer

trigger.on_kill() 3.3+

- user marks failed / clears

THE HONEST CONCESSION THAT EARNS THE CONCLUSION

trigger.on_kill() is the only place where infrastructure instability and user-intended kill are distinguishable.

06 / 06

Summary

2.7 to 3.0, 3.0 to 3.2, 3.3!

Every version since 2.7 can do this. Newer versions need less code.

VERSION	IDENTITY	WATCH	USER-KILL
2.7 – 3.0	deterministic name	deferrable	teardown task
3.0 – 3.2	deterministic name + trigger_kwargs	deferrable	teardown task
3.3!	deterministic name + task_state_store (AIP-103)	deferrable	teardown + trigger.on_kill()

Every trick, and where you can use it.

FINAL SUMMARY — THE LAST TWO ROWS ARE THE ONLY ONES NEEDING A RECENT AIRFLOW

TRICK	WHAT IT BUYS	WHERE
deterministic job name	every attempt re-finds the same job	any
AlreadyExists = success	submit is a safe no-op on retry	any
deferrable=True	no subprocess for SIGTERM to reach	2.2+
cancel.as_teardown(setups=submit)	cancel fires on task state, not signals	2.7+
wait_for_termination=True	submit + watch collapse into one task	provider
end_from_trigger=True	no resume, so no on_kill at completion	2.10+
trigger.on_kill()	prompt cancel on real user intent	3.3.0

Questions?

- SIGTERM carries no reason, so operator.on_kill cannot carry a decision.
- A signal handler fires on process events. A teardown fires on task-state events.
- The triggerer is the only place those two are distinguishable.

Jason(Zhe-You) Liu • Astronomer

jason.liu@astronomer.io

github.com/jason810496

linkedin.com/in/zhe-you-liu