airflow dag 실행중에 이런 로그는 어떻게 디버깅을 하는게 좋을까?

x9902167-rnd-data-mart-etl-wt-check-df-lot-id-task-m6bay4cd
*** Found local files:
*** * /opt/airflow/logs/dag_id=X9902167_rnd_data_mart_etl_wt/run_id=scheduled__2025-09-25T23:00:00+00:00/task_id=check_df_lot_id_task/attempt=1.log.SchedulerJob.log
[2025-09-26, 00:01:58 UTC] {scheduler_job_runner.py:769} ERROR - Executor reports task instance <TaskInstance: X9902167_rnd_data_mart_etl_wt.check_df_lot_id_task scheduled__2025-09-25T23:00:00+00:00 [queued]> finished (failed) although the task says it's queued. (Info: None) Was the task killed externally?

이 메시지는 "스케줄러는 아직 queued(대기)라고 보는데, 실행기(Executor)는 이미 failed(실패)로 끝났다고 보고했다. 외부에서 죽었나?)"라는 뜻입니다. **스케줄러-DB 상태와 쿠버네티스 쪽 실제 실행 상태가 어긋난 전형적인 케이스**예요. 원인은 보통 아래 중 하나입니다: (1) K8s에서 파드가 OOM/Preemption/Eviction/강제삭제, (2) 이미 아주 빨리 실패했는데 스케줄러가 반영 전에 봄(레이스), (3) 스케줄러/Executor 하트비트 끊김 또는 DB 락/지연.

바로 확인 루틴 드릴게요. (복붙-점검용)

---

1) Airflow 메타데이터 기준 "사실" 확인

스케줄러가 뭘 보고 있는지 먼저 고정합니다.

```bash

# 해당 run 의 TI 상태 표 (Airflow 2.9 기준)
airflow tasks states-for-dag-run X9902167_rnd_data_mart_etl_wt scheduled__2025-09-25T23:00:00+00:00

# 단건 확인
airflow tasks state X9902167_rnd_data_mart_etl_wt check_df_lot_id_task scheduled__2025-09-25T23:00:00+00:00

# (옵션) 시도 로그 경로 바로 보기
ls -al /opt/airflow/logs/dag_id=X9902167_rnd_data_mart_etl_wt/run_id=scheduled__2025-09-25T23:00:00+00:00/task_id=check_df_lot_id_task/
tail -n 200 .../attempt=1.log
```

DB를 직접 볼 수 있으면 상태/타임스탬프 불일치도 체크하세요.

```sql

-- 이 TI의 생애주기와 타임라인
select dag_id, task_id, run_id, try_number, state, queued_dttm, start_date, end_date, hostname, unixname
from task_instance
where dag_id='X9902167_rnd_data_mart_etl_wt'
  and task_id='check_df_lot_id_task'
  and run_id='scheduled__2025-09-25T23:00:00+00:00'
order by try_number desc;

-- 이 TI를 소유한 scheduler job (있다면)
```

```
select id, state, latest_heartbeat, start_date, end_date, hostname
from job
where job_type='SchedulerJob'
order by start_date desc limit 5;
```

- **queued_dttm 는 있는데 start_date 가 비어 있고 executor 는 failed 라고만 한 경우** → 쿠버네티스 파드 측 즉시 실패/강제 종료 가능성 큼.

---

2) Kubernetes 파드 "실제" 확인 (KubernetesExecutor 전제)

Airflow가 붙여주는 라벨로 바로 찾습니다.

```
bash


# try_number=1 기준으로 파드 찾기
kubectl -n <airflow-ns> get pod \
  -l
"airflow.apache.org/dag_id=X9902167_rnd_data_mart_etl_wt,airflow.apache.org/task_id=check_df_lot
_id_task,airflow.apache.org/try_number=1" \
  -o wide

# 이벤트/종료사유 확인
kubectl -n <airflow-ns> describe pod <pod-name>

# 종료 코드/신호 확인 (있으면)
kubectl -n <airflow-ns> logs <pod-name> --previous
kubectl -n <airflow-ns> get pod <pod-name> -o
jsonpath='{.status.containerStatuses[0].state.terminated.exitCode}{" "}
{.status.containerStatuses[0].state.terminated.reason}{" "}
{.status.containerStatuses[0].state.terminated.signal}{"\n"}'

# 노드 압박/축출 흔적
kubectl -n <airflow-ns> get events --sort-by=.lastTimestamp | tail -n 50
```

- **ExitCode 137 → OOMKilled**, **143 → SIGTERM(축출/스케일인/강제삭제)**, **Evicted/Preempted** 이벤트면 외부 종료가 맞습니다.
- **ImagePullBackOff / CreateContainerError**였으면 파드가 뜨기도 전에 실패 → Executor는 failed, 스케줄러는 잠깐 queued로 봤을 수 있어요.

---

3) 스케줄러/Executor 건강상태 점검

스케줄러가 orphan(고아) 상태를 정리했는지, 하트비트 지연이 있는지 로그를 훑습니다.

```
bash


# 스케줄러/익스큐터 로그 주변 시각(00:01:58 UTC) 스캔
grep -n "orphan" /opt/airflow/logs/scheduler/latest/*.log
grep -n "heartbeat" /opt/airflow/logs/scheduler/latest/*.log
grep -n "Executor reports task instance" /opt/airflow/logs/scheduler/latest/*.log
```

- "**Resetting orphaned tasks**" 같은 라인이 보이면 정리 도중 레이스였을 가능성.
- DB 연결 오류나 **pool deadlock**, **semaphore**, **max_active_tasks_per_dag** 제한으로 대기/해제 타이밍 엇갈린 흔적도 체크.

4) 재시도/정리 액션

원인 파악 뒤 바로 복구 루틴:

```
bash


# 해당 태스크만 클리어(재시도 유도)
airflow tasks clear X9902167_rnd_data_mart_etl_wt --task_regex '^check_df_lot_id_task$' \
  --dag-run-id scheduled__2025-09-25T23:00:00+00:00 --downstream false --y

# (대량 꼬였을 때) run 단위로 안전 클리어
airflow tasks clear X9902167_rnd_data_mart_etl_wt \
  --dag-run-id scheduled__2025-09-25T23:00:00+00:00 --only-running --y
```

K8s 파드가 실패 후에도 남아 있어 원인포렌식이 필요하면, Airflow 설정에서 **delete_worker_pods=False**(혹은 태스크별로 **executor_config**에 **annotations**로 TTL 지정)로 일시 유지해 두세요.

---

5) 재발 방지 체크리스트 (KubernetesExecutor 기준)

- **리소스 한도 현실화**: 태스크 컨테이너에 **resources.requests/limits**를 지정. OOMKilled(137)면 메모리 상향 또는 작업 배치 축소.
  (대량 배치 실행 시에는 이전에 논의했던 **dynamic task mapping + 청크 처리**로 피크 메모리·동시성 완화.)
- **노드 압박 회피**: 클러스터에 Eviction 빈번하면 노드 자원/오버커밋/스케줄 정책 점검. 필요 시 전용 노드풀/노드셀렉터/톨러레이션.
- **이미지/시크릿 안정화**: ImagePullBackOff가 잦으면 레지스트리 접근/이미지 태그 고정/프리풀(daemonset) 고려.
- **하트비트/DB**: 스케줄러/웹서버/트리거러 하트비트 지연이 보이면 DB 성능(커넥션 수, 락, IOPS)과 **sql_alchemy_pool_size**, **max_active_runs**, **parallelism**, **max_active_tasks_per_dag** 튜닝.
- **로그 보존/증거수집**: **on_failure_callback**에서 K8s 파드 이벤트/상태를 자동 덤프해 로그에 남기면, 다음부터는 원인확인이 1단계로 끝납니다.

예) 간단 콜백(요지):

```
python


from kubernetes import client, config
from airflow.models import TaskInstance

def dump_pod_events_on_fail(context):
    ti: TaskInstance = context["ti"]
    dag_id = ti.dag_id; task_id = ti.task_id; try_number = ti.try_number
    # 클러스터 내에서 실행 시 InClusterConfig
    try:
        config.load_incluster_config()
        v1 = client.CoreV1Api()
        lbl = f"airflow.apache.org/dag_id={dag_id},airflow.apache.org/task_id=
{task_id},airflow.apache.org/try_number={try_number}"
        pods = v1.list_namespaced_pod(namespace="<airflow-ns>", label_selector=lbl).items
        for p in pods:
            print(f"[FAILURE POD] {p.metadata.name} phase={p.status.phase} reason=
{p.status.reason}")
            ev = v1.list_namespaced_event("<airflow-ns>", field_selector=f"involvedObject.name=
{p.metadata.name}")
            for e in ev.items:
                print(f"[EVENT] {e.last_timestamp} {e.type} {e.reason} {e.message}")
```

```
    except Exception as e:
        print("[dump_pod_events_on_fail] error:", e)
```

DAG 기본값에 붙이기:

```
python

default_args = {"on_failure_callback": dump_pod_events_on_fail, "retries": 1}
```

---

6) 지금 케이스에서 특히 볼 것

- **00:01:58 UTC 주변 파드 이벤트**에서 Evicted/OOMKilled/Preempted/NodeNotReady 여부.
- 종료 코드 **137/143** 여부.
- **`attempt=1.log.SchedulerJob.log`**만 보인다면, **태스크 컨테이너 로그가 생성되기 전 단계에서 실패**했을 확률이 큽니다(이미지 풀 실패, 스케줄/StartContainer 실패 등).

필요하시면 **`dag_id`**, **`run_id`**, 네임스페이스만 알려주세요. 라벨 셀렉터/쿼리/튜닝 키워드를 바로 그 값들로 맞춰서 더 구체적으로 정리해 드리겠습니다.

다음과 같은 dag 코드에서 어떤걸 추가하면 되지?

```
with DAG(
        dag_id=config.dag_nm,
        schedule_interval=f'0 */{config.coverage_hour} * * *',  # (분, 시간, )
        tags=['rnd', 'wt', 'pyspark'],
        default_args= {
                    'owner': 'a001',
                    'start_date': datetime(2025, 5, 21),
                    'retries': 10,
                    'retry_delay': timedelta(minutes=1)
                },
        max_active_tasks=3,
        max_active_runs=1,
        on_failure_callback = px.AddIn().on_failure_callback(channel_list = ['200393469'], msg='')
        # catchup=True
) as dag:
```

좋아요. 지금 DAG에 “무슨 일이 있었는지 증거를 자동으로 남기는” **실패 콜백**만 추가하면, 방금 같은 **`queued↔failed`** 불일치도 바로 포렌식할 수 있습니다. 아래처럼 **K8s 파드/이벤트/종료코드**를 실패 시 로그로 덤프하는 콜백을 붙이세요.

1) 실패 콜백 함수 추가 (DAG 파일 상단)

```
python

import os, traceback
from kubernetes import client, config
```

```
def dump_pod_evidence(context):
    ti = context["ti"]
    ns = os.getenv("AIRFLOW__KUBERNETES__NAMESPACE") or os.getenv("NAMESPACE") or "airflow"

    try:
        # Airflow on K8s에서는 InCluster 설정이 기본
        try:
            config.load_incluster_config()
        except Exception:
            # 로컬 테스트 대비
            config.load_kube_config()

        v1 = client.CoreV1Api()
        label_selector = (
            f"airflow.apache.org/dag_id={ti.dag_id},"
            f"airflow.apache.org/task_id={ti.task_id},"
            f"airflow.apache.org/try_number={ti.try_number}"
        )

        pods = v1.list_namespaced_pod(namespace=ns, label_selector=label_selector).items
        if not pods:
            print(f"[EVIDENCE] No pod found. ns={ns} labels=[{label_selector}]")
            return

        for p in pods:
            print(f"[EVIDENCE] pod={p.metadata.name} phase={p.status.phase} node=
{p.spec.node_name}")
            cs_list = p.status.container_statuses or []
            for cs in cs_list:
                term = (cs.state and cs.state.terminated) or None
                if term:
                    print(f"[EVIDENCE] container={cs.name} exitCode={term.exit_code} "
                          f"reason={term.reason} signal={term.signal}\n{term.message or ''}")

            # 최근 이벤트 20개만
            evs = v1.list_namespaced_event(ns, field_selector=f"involvedObject.name=
{p.metadata.name}").items
            for e in evs[-20:]:
                print(f"[EVENT] {e.last_timestamp} {e.type} {e.reason} {e.message}")

    except Exception as e:
        print("[EVIDENCE] error:", e)
        print(traceback.format_exc())
```

2) DAG 선언에 연결

지금 쓰신 DAG에 아래 두 줄만 보강하면 됩니다.

- (A) **`default_args`**에 **`on_failure_callback`**을 **함수로** 붙이기
- (B) **`retry_exponential_backoff=True`**로 재시도 간격 점증(군집 압박 완화)

```
python

with DAG(
    dag_id=config.dag_nm,
    schedule_interval=f'0 */{config.coverage_hour} * * *',
    tags=['rnd', 'wt', 'pyspark'],
    default_args={
        'owner': 'a001',
        'start_date': datetime(2025, 5, 21),
        'retries': 10,
        'retry_delay': timedelta(minutes=1),
        'retry_exponential_backoff': True,
        'on_failure_callback': dump_pod_evidence,   # (A) 실패 시 K8s 증거 덤프
    },
    max_active_tasks=3,
    max_active_runs=1,
```

```
    # dag 레벨 콜백을 이미 쓰고 있었다면 -> 위 함수로 교체/병합
) as dag:
    ...
```

> 참고: 기존 **`px.AddIn().on_failure_callback(...)`**을 쓰시던 이유가 알림(Slack 등)이었다면, **알림 + 증거 덤프**를 합친 래퍼를 하나 만들어서 그 래퍼를 **`on_failure_callback`**으로 걸어도 됩니다. (알림 전/후로 **`dump_pod_evidence(context)`** 호출)

3) (선택) 파드 보존으로 현장 보전

원인 포렌식이 더 필요하면, 헬름 값이나 env로 **워커 파드 자동삭제를 잠시 끄는** 것도 유용합니다.

- Helm values (Bitnami/Community 공통)에서 웹서버/스케줄러/트리거러 env에:

```
yaml

env:
  - name: AIRFLOW__KUBERNETES__DELETE_WORKER_PODS
    value: "False"
```

> 이후엔 정리 비용이 늘 수 있으니, 원인 규명 끝나면 다시 True로 돌리세요.

이렇게만 해두면, 다음 같은 로그가 자동으로 남습니다:

- **`phase/exitCode/reason(예: 137=OOMKilled, 143=SIGTERM)`**
- **`Evicted/Preempted/ImagePullBackOff`** 등 **이벤트 메시지**
- 어떤 노드에서 죽었는지

즉, “스케줄러는 queued인데 왜 Executor는 failed라 했는가?”를 **로그 한 장**으로 끝낼 수 있습니다.

> 전역환경 수정은 불가능해. dag수준에서만 변경이 가능하거든

좋아요. “전역 못 건드림” 기준으로, **DAG 파일 안에서만** 넣어 쓸 수 있는 추가사항을 딱 필요한 것만 정리해 드릴게요. 핵심은 두 가지예요:

1. 실패 시 K8s 파드/이벤트/종료코드 자동 덤프 콜백(증거 수집)
2. 재시도·타임아웃·백오프 등 런타임 안전장치

아래 코드는 DAG 한 파일에 그대로 넣어 쓰면 됩니다.

---

1) 실패 콜백: 파드/이벤트/종료코드 로그로 남기기

- 네임스페이스는 env 못쓴다 했으니, **serviceaccount 파일**에서 직접 읽습니다.
- Airflow가 자동 부여한 라벨(dag_id / task_id / try_number)로 파드 매칭.

```
python


# DAG 파일 상단에 추가
import traceback, pathlib
from kubernetes import client, config

def _incluster_namespace() -> str:
    p = pathlib.Path("/var/run/secrets/kubernetes.io/serviceaccount/namespace")
    return p.read_text().strip() if p.exists() else "default"

def dump_pod_evidence(context):
    ti = context["ti"]
    ns = _incluster_namespace()
    try:
        try:
            config.load_incluster_config()
        except Exception:
            config.load_kube_config()
        v1 = client.CoreV1Api()
        ls = (
            f"airflow.apache.org/dag_id={ti.dag_id},"
            f"airflow.apache.org/task_id={ti.task_id},"
            f"airflow.apache.org/try_number={ti.try_number}"
        )
        pods = v1.list_namespaced_pod(namespace=ns, label_selector=ls).items
        if not pods:
            print(f"[EVIDENCE] no pod. ns={ns} labels=[{ls}]")
            return
        for p in pods:
            print(f"[EVIDENCE] pod={p.metadata.name} phase={p.status.phase} node=
{p.spec.node_name}")
            cs_list = p.status.container_statuses or []
            for cs in cs_list:
                term = cs.state.terminated if cs.state and cs.state.terminated else None
                if term:
                    print(f"[EVIDENCE] container={cs.name} exit={term.exit_code} "
                          f"reason={term.reason} signal={term.signal}\n{term.message or ''}")
            evs = v1.list_namespaced_event(
                ns, field_selector=f"involvedObject.name={p.metadata.name}"
            ).items
            for e in evs[-20:]:
                print(f"[EVENT] {e.last_timestamp} {e.type} {e.reason} {e.message}")
    except Exception as e:
        print("[EVIDENCE] error:", e)
        print(traceback.format_exc())
```

2) DAG 선언부: default_args 로 전파 + 백오프/타임아웃

- 전역 말고 **DAG 파일 내부에서** 전부 통제합니다.
- **`default_args['on_failure_callback']`**에 위 콜백을 붙이면 **모든 태스크 실패 시 자동 실행**.
- 과도한 동시 재시도를 눌러주려고 **지수형 백오프**와 **실행 타임아웃**도 같이.

```
python


from datetime import datetime, timedelta
from airflow import DAG

with DAG(
    dag_id=config.dag_nm,
    schedule_interval=f'0 */{config.coverage_hour} * * *',
    tags=['rnd', 'wt', 'pyspark'],
    default_args={
        'owner': 'a001',
        'start_date': datetime(2025, 5, 21),
        'retries': 10,
        'retry_delay': timedelta(minutes=1),
```

```
        'retry_exponential_backoff': True,          # 재시도 간격 점증
        'max_retry_delay': timedelta(minutes=30),   # (선택) 상한
        'on_failure_callback': dump_pod_evidence,   # ★ 실패 시 K8s 증거 수집
        'execution_timeout': timedelta(hours=2),    # ★ 태스크 단위 타임아웃
    },
    max_active_tasks=3,
    max_active_runs=1,
    dagrun_timeout=timedelta(hours=6),              # ★ DAG Run 전체 타임아웃
) as dag:
    ...
```

3) (선택) 태스크별 리소스/스케줄 힌트도 "DAG 코드"에서 지정

전역 values 못 바꾼다 했으니, **오퍼레이터 생성 시 `executor_config`**로 파드 스펙을 보탤 수 있어요.
OOM/Evict가 의심되면 최소한 requests/limits를 박아두면 효과 큽니다.

```
python


from kubernetes.client import V1Pod, V1PodSpec, V1Container, V1ResourceRequirements,
V1SecurityContext

TASK_EXECUTOR_CONFIG = {
  "pod_override": V1Pod(
    spec=V1PodSpec(
      containers=[
        V1Container(
          name="base",
          resources=V1ResourceRequirements(
            requests={"cpu": "500m", "memory": "1Gi"},
            limits={"cpu": "2", "memory": "4Gi"},
          ),
          security_context=V1SecurityContext(run_as_user=1002, run_as_group=0),  # 사용자 선호 반영
        )
      ]
    )
  )
}

# 예시: PythonOperator(..., executor_config=TASK_EXECUTOR_CONFIG)
```

> 이렇게 하면 **특정 태스크만** 리소스를 상향해 재발을 줄일 수 있습니다. (전역 수정 불가 상황에서 현실적인 대응책)

4) (선택) 실패 알림이 이미 있다면, "알림 + 증거덤프" 결합 래퍼

기존 **`px.AddIn().on_failure_callback(...)`**을 쓰고 있다면, 아래처럼 합쳐 한 번에 호출하세요.

```
python


def notify_and_dump(context):
    try:
        dump_pod_evidence(context)   # 먼저 증거 수집
    finally:
        px.AddIn().on_failure_callback(channel_list=['200393469'], msg='')(context)

# 그리고 default_args['on_failure_callback'] = notify_and_dump
```

이 정도면 **전역 손대지 않고도**:

- 실패 원인이 K8s 쪽(137/143/Evicted/ImagePullBackOff 등)인지 즉시 드러나고,
- 재시도 폭주/교착을 완만하게 만들며,
- 필요한 태스크만 리소스를 올려 **실제 장애율을 낮출** 수 있어요.