



airflow dag 코딩시에 task로 처리 또는 task내에서 function 을 적절히 활용등의 어떤 관점에서 하는게 좋을 까?

좋은 관점이에요. 정리하면 이렇습니다—“**태스크는 스케줄링/장애복구의 단위, 함수는 로직의 단위.**”
Airflow에서는 이 두 개를 깔끔히 분리할수록 유지보수·성능·디버깅이 편해집니다.

어떤 걸 태스크로? 어떤 걸 함수로?

- **태스크로 쪼갤 것**
 - 외부 시스템 I/O가 있고 실패/재시도가 의미 있는 단계 (DB 적재, S3 업로드, StarRocks DDL/DML, Spark/KPO 제출)
 - 리소스 요구량/권한/컨테이너 이미지가 다른 단계 (CPU/GPU/메모리, UID·GID, 이미지 태그가 다를 때)
 - 병렬도/스케줄 제약이 다른 단계 (rate limit, pool, priority_weight, queue)
 - 체크포인트가 필요한 지점 (중간 산출물 고정, 부분 성공 인정)
 - 다른 담당자/오너십이 갈리는 경계
- **함수로 묶을 것 (태스크 내부)**
 - 순수 계산, 포매팅, 검증, 파싱, 작은 변환 등 **빠르고 결정적(deterministic)** 로직
 - 재사용되는 유틸리티 (예: SQL 빌더, 스키마 검증, S3 key 생성)
 - 여러 태스크에서 동일하게 쓰이는 전처리/후처리

K8s(특히 KPO/KE)에서의 현실적인 기준

- 작은 일을 너무 많은 태스크로 쪼개면 오버헤드↑ (스케줄러·DB 메타 업데이트, K8s Pod 스펙업, 이미지 풀, 로그 집계 등).
- 반대로 큰 일을 하나의 태스크로 몰면 재시도 비용이 커지고, 중간 디버깅 포인트가 사라집니다.
- 경험칙:
 - 외부 호출 1회 ≈ 1 태스크(뜻있는 재시도/격리가 필요하면)
 - 같은 리소스/이미지/권한에서 돌아가는 순수 파이썬 전처리는 한 태스크 안의 여러 함수로 흡수
 - 파티션 다건 처리는 TaskFlow + dynamic task mapping이나 Spark로 넘기고, 결과 수집/커밋을 별도 태스크로

병렬성과 안정성 관점 체크리스트

- 병렬도: parallelism, dag_concurrency, max_active_runs, pool로 상한을 제어.
작은 태스크를 과도 병렬화하면 외부 시스템을 압사시킬 수 있으니 풀/레이트리밋으로 조절.
- 상태/아티팩트: XCom에는 경량 메타만. 데이터는 S3/MinIO에 두고 **키만 XCom**으로.

- 리소스: KPO/KE라면 태스크 경계마다 `resources`, `nodeSelector`, `tolerations`, `securityContext(UID=1002)` 등을 바꿀 수 있음 → **경계 = 태스크**가 맞습니다.
- 재시도 비용: 오래 걸리는 작업은 **준비/검증** → **본작업** → 커밋처럼 2~3단으로 쪼개어 실패 범위를 좁히기.

구조 패턴 (권장)

1. 얇은 태스크 + 두터운 함수(모듈)

- `dags/etl_foo.py`에는 태스크 선언만, 비즈니스 로직은 `libs/etl/foo.py` 등에.
- 함수는 순수/테스트가능/재사용, 태스크는 스케줄·권한·리소스의 데코레이터 역할.

2. 입·출력 분리

- `extract(I/O)` → `transform`(함수 내부에서 메모리/파일 변환) → `load(I/O)` 3단.
- `transform`은 가능하면 하나의 태스크(내부 함수 분기)로 묶어 오버헤드 절감.

3. 다이내믹 매핑

- 날짜·파티션·장비ID 등 리스트를 매핑해 **N개의 동일 태스크** 생성.
- 후속 `reduce` 또는 `aggregate` 태스크로 종합.

4. 무상태(Stateless) 기본

- 로컬 디스크 의존 X, 외부 상태는 항상 명시적으로 읽고 쓰기.
- 체크포인트는 오브젝트 스토리지와 idempotent 키 설계로.

예시 스케치 (사용자 환경 반영: UID=1002, KPO 혼용)

```
python

# dags/example_etl.py (Airflow 2.9 TaskFlow API)
from airflow import DAG
from airflow.decorators import task
from airflow.utils.dates import days_ago
from kubernetes import client as k8s
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import KubernetesPodOperator
from libs.etl import build_sql, validate_rows, write_parquet # <-- 함수는 외부 모듈

default_args = dict(retries=2)

UID_1002 = k8s.V1SecurityContext(run_as_user=1002, run_as_group=0)

with DAG(
    "example_etl",
    start_date=days_ago(1),
    schedule="@daily",
    default_args=default_args,
    max_active_runs=1,
    catchup=False,
) as dag:

    @task
    def extract(ds: str) -> dict:
        # 가벼운 메타 로직은 함수로
        sql = build_sql(ds) # 순수 함수
        # 실제 데이터는 DB→S3로 덤프하고, S3 키만 리턴
        s3_key = f"raw/mytbl/dt={ds}/part-000.parquet"
        # ... (DB → S3 I/O 수행) ...
        return {"s3_key": s3_key, "rows": 12345}
```

```
@task
def transform(meta: dict) -> dict:
    # S3에서 읽어 변환 → S3에 다시 기록 (함수들을 조합)
    rows = validate_rows(meta["s3_key"]) # 순수 검증 로직
    out_key = write_parquet(rows) # 순수/경량 함수들로 구성
    return {"out_key": out_key}

load_heavy = KubernetesPodOperator(
    task_id="load_heavy",
    name="load-heavy",
    image="dlops/airflow:2.9.2-0.3-python3.11-spark3.5.3",
    cmds=["bash", "-lc"],
    arguments=["python /app/jobs/load_to_starrocks.py --s3-key {{ ti.xcom_pull('transform')
['out_key'] }}"],
    namespace="airflow",
    is_delete_operator_pod=True,
    container_resources=k8s.V1ResourceRequirements(
        requests={"cpu": "1", "memory": "2Gi"},
        limits={"cpu": "2", "memory": "4Gi"},
    ),
    container_security_context=UID_1002, # ← 사용자 선호 반영
)

meta = extract()
out = transform(meta)
meta >> out >> load_heavy
```

태스크 개수에 대한 감(KE/KPO 기준)

- 동일 이미지/권한/리소스에서 몇 초 내 끝나는 계산/포매팅/검증은 **한 태스크**로 합치는 편이 유리.
- 네트워크 호출 1회는 보통 1 태스크(재시도/격리를 위해).
- 파티션 수백 개는 매핑으로 쪼개되, 외부 시스템 부하를 고려해 **pool**과 **max_active_tis_per_dag**로 제한.
- 스파크/대용량 배치는 KPO 하나에 작업을 몰고, 내부 단계는 스파크 쪽 DAG로 관리.

작은 결론

- 경계(리소스/권한/재시도/체크포인트)가 다르면 → 태스크.
- 동일 컨텍스트에서 굴러가는 순수 로직은 → 함수.
- 데이터는 오브젝트 스토리지로 흘리고, XCom은 경량 메타만.
- 매핑·풀·리소스 한도를 적극 활용해 **“적은 수의 의미 있는 태스크 + 두터운 함수 재사용”**을 기본값으로 잡으시면 안정적입니다.

원하시면 지금 운영 중인 대표 DAG 하나 골라서, “태스크 경계 재설계” 버전으로 리팩터링 예시 만들어드릴게요.



KubernetesExecutor환경에서 오라클 쿼리 한개 단위로 task를 나누는건 너무 비효율적인것 같은데

맞아요. **KubernetesExecutor(KE)**에서 “오라클 쿼리 1개 = 태스크 1개”는 손해가 큼니다.

KE는 태스크마다 새 파드가 뜨기 때문에, **파드 스핀업·이미지 풀·연결수립 오버헤드**가 쿼리 실행 시간보다 길어지기 쉽습니다. 실무에선 아래 원칙으로 정리하면 깔끔합니다.

권장 원칙 (요지)

- **태스크 경계 = 의미 있는 트랜잭션/체크포인트 경계.**
쿼리 1개가 아니라, **동일 리소스·권한·롤백 단위로 묶어서** 한 태스크에 담으세요.
- **쿼리 묶음 실행 + 세이프포인트(부분 롤백) + 애플리케이션 재시도.**
오류가 나도 전체가 아닌 **해당 구간만 롤백**하고 이어갈 수 있게 만듭니다.
- **다이내믹 매핑은 “파티션/테이블 단위” 정도로만.**
“쿼리당 1태스크” 대신 ****논리 배치(예: 테이블 하나, 파티션 하루)****를 한 태스크로.

안전한 묶음 전략 (Transaction & Checkpointing)

한 태스크 안에서 **N개의 SQL**을 실행하되:

1. **사전 검증:** 대상 오브젝트 존재, 권한, 세션 파라미터 설정(예: `ALTER SESSION SET nls_date_format=...`).
2. **실행 구간:** `SAVEPOINT sp_k`마다 소구간으로 끊고 진행.
3. **오류 처리:** 재시도 가능한 에러(네트워크, 잠금)만 **지수 백오프**로 재시도, 영구 에러는 즉시 실패/스킵 기록.
4. **커밋 정책:** 소구간마다 커밋(대량 적재 시) 또는 끝에 일괄 커밋(DDL/일관성 중요 시).
5. **체크포인트 기록:** 컨트롤 테이블(Oracle) 또는 **S3/MinIO의 상태 파일**에 진행상황 저장 → 재실행 시 이어서.

이렇게 하면 “태스크 수는 적게, 실패 반경은 좁게”가 됩니다.

매핑 기준 (언제 태스크를 나눌까)

- **나누기 좋을 때**
 - 테이블/스키마/파티션 등 **업무 의미가 분명한 경계.**
 - 리소스/권한/세션 세팅이 달라지는 지점.
 - 재시도 비용이 커서 중간 저장이 꼭 필요한 지점.
- **합치기 좋을 때**
 - 동일 스키마, 동일 세션 세팅에서 돌아가는 짧은 DML/DDDL 여러 개.
 - 전·후처리가 거의 없고, **I/O 왕복보다 실행시간이 짧은** 쿼리들.

성능/안정성 팁 (cx_Oracle/oracledb)

- **세션 풀?** KE에선 태스크마다 파드가 바뀌니 프로세스 생존 간 풀 재사용이 사실상 불가. 대신 **실행 안에서 최대한 재사용**(1 커넥션으로 N개 쿼리).
- **배치 DML:** `executemany()` + `batcherrors=True`로 행 단위 오류만 골라 처리.
- **페치 성능:** `cursor.arraysize`, `prefetchrows` 조정. LOB는 필요 시에만 가져오기.
- **Idempotent 설계:** MERGE/UPSERT, 중복 키 무해화, **재시도 시 부작용 최소화.**

- **에러 등급화:** 재시도 가능(예: ORA-00054 lock busy, ORA-12170 timeout, ORA-12541 listener) vs 영구 오류(ORA-00942 table not exist 등) 분리.

코드 예시 (한 태스크에서 다수 SQL을 안전하게 실행)

```
python

# PythonOperator/TaskFlow 내부에서 사용할 헬퍼 (oracledb 권장)
import time, oracledb
from typing import List

RETRYABLE_ORA_PREFIX = ("ORA-00054", "ORA-12170", "ORA-12514", "ORA-12541", "ORA-03135")

def run_sql_batch(
    dsn: str, user: str, password: str,
    sql_list: List[str],
    commit_mode: str = "chunk", # "chunk" | "end"
    chunk_size: int = 5,
    max_retries: int = 3,
    retry_backoff: float = 2.0,
):
    """
    - 동일 세션에서 sql_list를 순서대로 실행
    - chunk_size마다 SAVEPOINT/커밋
    - 재시도 가능한 오류는 소구간 단위로 백오프 재시도
    """
    conn = oracledb.connect(user=user, password=password, dsn=dsn)
    try:
        cur = conn.cursor()
        # 세션 파라미터 예시
        cur.execute("ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD'")

        i = 0
        while i < len(sql_list):
            chunk = sql_list[i:i+chunk_size]
            cur.execute("SAVEPOINT sp_chunk")
            attempt = 0
            while True:
                try:
                    for sql in chunk:
                        cur.execute(sql)
                    if commit_mode == "chunk":
                        conn.commit()
                    break # chunk 성공
                except oracledb.Error as e:
                    msg = str(e)
                    if any(pref in msg for pref in RETRYABLE_ORA_PREFIX) and attempt <
max_retries:
                        # 롤백 후 재시도
                        cur.execute("ROLLBACK TO SAVEPOINT sp_chunk")
                        sleep_s = (retry_backoff ** attempt)
                        time.sleep(sleep_s)
                        attempt += 1
                        continue
                    else:
                        # 영구 오류 또는 재시도 초과
                        raise
            i += len(chunk)

        if commit_mode == "end":
            conn.commit()
    finally:
        conn.close()
```

Airflow 태스크 쪽(예: TaskFlow)에서는 쿼리 묶음만 넘겨 실행하면 됩니다.

python

```
from airflow.decorators import task

@task(retries=1)
def apply_schema_changes(ds: str):
    sqls = [
        "CREATE TABLE IF NOT EXISTS T1 (ID NUMBER, DT DATE)",
        "CREATE INDEX IF NOT EXISTS IDX_T1_DT ON T1(DT)",
        "COMMENT ON TABLE T1 IS 'daily load'",
        # ...
    ]
    run_sql_batch(
        dsn="HOST:PORT/SVC",
        user="APP",
        password="{{ conn.oracle.password }}", # Airflow Connection/Jinja 사용 권장
        sql_list=sqls,
        commit_mode="chunk",
        chunk_size=3
    )
```

태스크 설계 패턴 (오라클 중심)

- (준비) 프리체크: 권한/오브젝트/세션 파라미터 확인 → 빠른 실패
- (본작업) 다수 SQL 묶음 실행(위 헬퍼)
- (사후) 통계 갱신/가시성: 컨트롤 테이블에 결과·실패 원인 기록, AWR/ASH 포인트 링크 로그 남기기
- **대량 적재**는 별도 태스크에서 **SQL*Loader**/외부테이블/**INSERT /*+ APPEND */** 병렬 옵션 등으로 처리

언제 다이내믹 매핑?

- 하루 파티션 200개처럼 병렬이 의미 있을 때만.
외부 DB를 과부하시키지 않도록 **pool**과 **rate limit**로 제한하세요.
- 매핑 단위 예: [{table: T1, dt: 2024-01-01}, ...] → 테이블×파티션 = 1 태스크
"SQL 1개 = 1 태스크"는 지양.

요약

- KE 환경에선 쿼리 단위 태스크 분할은 비효율.
- 업무·트랜잭션 의미 단위로 묶고, 세이프포인트 + 부분 커밋 + 재시도를 한 태스크에서 처리.
- 병렬화는 파티션/테이블 단위까지만, DB 부하를 **pool/레이트리밋**으로 제어.
- 대량 작업은 배치 DML·병렬 옵션·외부 테이블 등 **오라클 친화적 기법**을 활용.

원하시면 지금 돌리는 DAG 하나 가져와서, **쿼리 1개형 다건 태스크** → "의미 단위 묶음형 소수 태스크"로 즉시 리팩터링해드릴게요.



(KubernetesExecutor환경)

bulk로 대량(수천개)으로 작업을 실행하는데, 이럴때 각 작업별로 dag를 파라미터 처리해서 실행하는데, 이때 각각의 dag는 task를 20개 정도 가지고 있거든, 내가 생각할때는 참 잘못된 로직 같거든.

맞는 감각이세요. ***수천 건 × (DAG를 건별로 생성/트리거) × (DAG당 20태스크)***는 KubernetesExecutor에선 거의 최악의 조합입니다.

파드 스핀업/이미지 풀/메타데이터 갱신 비용이 폭발하고, 스케줄러 DB도 괴롭힙니다. 더 낮게 가려면 아래처럼 바꾸세요.

방향성 한 줄 요약

- DAG 수를 줄이고, 한 DAG Run 안에서 *dynamic task mapping* + *체크 처리*로 팬아웃.
- 작업 경계는 “비즈니스 단위/리소스 단위/롤백 단위”로, 쿼리 1개·작업 1개씩 쪼개지 말 것.
- KE 오버헤드 최소화: 짧은 로직은 함수 내부로 흡수, I/O 큰 단계만 태스크로 남기기.

권장 아키텍처 패턴

1) “컨트롤러 DAG 1개 + 매핑” 패턴

- 단일 DAG가 “오늘 처리할 작업목록(work items)”을 조회(예: Oracle/MinIO의 메타테이블).
- @task → 리스트 반환 → `expand()`로 동일 태스크의 팬아웃 생성.
- 체크(chunk) 단위로 묶어서 1태스크가 N개를 처리(예: 1태스크 = 50건).
→ Pod 수, 커넥션 수, 로그 스트림 수가 급감.

python

```
from airflow import DAG
from airflow.decorators import task
from airflow.utils.dates import days_ago

CHUNK = 50 # 1 태스크가 처리할 건수

with DAG("bulk_controller", start_date=days_ago(1), schedule="@hourly", catchup=False,
max_active_runs=1) as dag:

    @task
    def plan() -> list[dict]:
        # 메타테이블/큐에서 작업목록 조회 → [{"id":..., "table": "T1", ...}, ...]
        items = fetch_work_items(limit=5000)
        # 체크로 묶기
        return [ {"batch": items[i:i+CHUNK]} for i in range(0, len(items), CHUNK) ]

    @task.pool("oracle_dml", pool_slots=1, retries=2)
    def run_batch(batch: dict):
        # 한 세션에서 SAVEPOINT로 부분 커밋/롤백, 재시도 가능한 에러만 백오프
        exec_oracle_batch(batch["batch"]) # 내부는 함수(응답 이전에 드린 run_sql_batch 유사)

    @task
    def finalize():
        mark_run_complete()
```

```
chunks = plan()
run_batch.expand(batch=chunks) >> finalize()
```

핵심

- `pool('oracle_dml')`로 동시 커넥션/부하 상한을 강제.
- **CHUNK**를 조절해 **Pod** 개수와 세션 수를 통제.
- 짧은 파싱/검증은 함수로 내부 처리(태스크 늘리지 않기).

2) “작업 수천 건 → KPO 1~N개에 몰아넣기” 패턴

- 정말 초대용량이면, 파이썬 태스크 대신 **KubernetesPodOperator** 1~수개가 내부에서 멀티프로세스/스레드로 처리(컨테이너 내부 워커).
- 장점: Airflow 태스크 수 최소화, Pod만 몇 개.
- 단점: 세밀한 재시도/가시성은 컨테이너 쪽 로직에서 구현해야 함.

3) 큐(메시지/테이블) 기반 워커 패턴

- 컨트롤러 **DAG**는 큐에 적재만, 워커 **DAG**는 주기적으로 큐를 폴링해 한 번에 **K**개씩 소비.
- Airflow가 스케줄러 역할만 하고, 세밀한 상태관리는 큐/테이블로.

안티패턴 (지금 구조의 문제점)

- 작업별 **DAG** 생성/트리거: 스케줄러 메타DB 트래픽·락 경합·UI 성능 모두 악화.
- **DAG당 20태스크 × 수천 DAG Run**: KE에선 Pod 수가 기하급수적으로 증가 → 이미지 풀/CPU/네트워크 병목.
- 짧은 단계까지 태스크화: “파드 띄우는 비용 > 로직 실행 시간”이 되면 손해.

가드레일(필수 한도 설정)

- 풀(Pools): 외부 DB(오라클) 보호용 상한. 예: `oracle_dml` 풀 10슬롯.
- `max_active_runs`(DAG별): 컨트롤러 1로 고정 권장.
- 전역 상한: `parallelism`, `max_active_tis_per_dag`(전역)으로 폭주 방지.
- 재시도 정책: 재시도 가능한 ORA 코드만 백오프, 나머지는 즉시 실패 기록.
- **Idempotent** 키: 작업 재시도 시 중복 반영 방지(머지/업서트/체크포인트).

전환(리팩터) 플랜, 짧고 굵게

1. 작업메타 테이블/파일 표준화: `work_id`, `state`(PENDING/RUNNING/DONE/ERR), `payload`, `updated_at`.
2. 컨트롤러 **DAG** 1개로 통합. 기존 수천 DAG 트리거 → 컨트롤러 단일 **Run**으로 대체.
3. **dynamic mapping** + 청크 적용. 최초엔 `CHUNK=50`, 풀슬롯=10으로 시작.

4. **에러 테이블/로그 구조 정리**: 어떤 work_id가 왜 실패했는지 한눈에.
5. **부하 테스트**: 개발/스테이징에서 이미지 캐시를 맞추고, Pod 수/세션 수/AWR 확인.
6. **점진 전환**: 일부 작업군만 먼저 이관 → 문제 없으면 전량 전환.

간단 체크리스트

- ☐ “짧은 전처리/검증”은 **함수**로 흡수했는가
- ☐ 한 세션에서 **N개 처리 + SAVEPOINT**로 실패 반경을 좁혔는가
- ☐ **Pool/청크**로 DB/Pod 수 상한을 걸었는가
- ☐ **컨트롤 테이블**로 재시작 가능성을 확보했는가
- ☐ **KPO로 몰아넣기**가 더 유리한 구간은 없는가(초대용량/CPU 바운드)

원하시면 현재 “작업별 DAG + 20태스크” 중 하나를 샘플로 주시면, 위 패턴으로 **컨트롤러 + 매핑 + 청크** 구조로 바로 바꿔 드릴게요. KE 환경(Uid=1002, KPO 혼용) 기준으로 리소스/풀/리트라이까지 세팅해드립니다.



현재의 챗방의 답변내용들 기억해줘

네, 지금 대화에서 정리된 관점들(작업별 DAG 다수 생성은 비효율 → 컨트롤러 DAG + 다이내믹 매핑 + 청크 처리 등)은 기억해둘게요.