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

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, 포맷팅, 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 스핀업, 이미지 풀, 로그 집계 등).
- 반대로 **큰 일을 하나의 태스크**로 몰면 재시도 비용이 커지고, 중간 디버깅 포인트가 사라집니다.
- 경험칙:
  - **외부 호출 1회 ≈ 1 태스크**(뜻있는 재시도/격리가 필요하면)
  - **같은 리소스/이미지/권한**에서 돌아가는 **순수 파이썬 전처리**는 한 태스크 안의 여러 **함수**로 흡수
  - **파티션 다건 처리**는 `TaskFlow + dynamic task mapping`이나 Spark로 넘기고, 결과 수집/커밋을 별도 태스크로

병렬성과 안정성 관점 체크리스트

- **병렬도**: `parallelism`, `dag_concurrency`, `max_active_runs`, `pool`로 상한을 제어.
  작은 태스크를 과도 병렬화하면 외부 시스템을 압사시킬 수 있으니 **풀/레이트리밋**으로 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** → **커밋**처럼 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 모듈로

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/리소스**에서 몇 초 내 끝나는 계산/포맷팅/검증은 **한 태스크**로 합치는 편이 유리.
- **네트워크 호출 1회**는 보통 **1 태스크**(재시도/격리를 위해).
- **파티션 수백 개**는 매핑으로 쪼개되, 외부 시스템 부하를 고려해 **pool**과 **max_active_tis_per_dag**로 제한.
- **스파크/대용량 배치**는 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.