큐의 기술: 배경 작업

한꺼번에 처리하지 않는 규율

ThakiCloud (Hyojung Han)

큐의 기술: 배경 작업

한꺼번에 처리하지 않는 규율

ThakiCloud (Hyojung Han)

Contents

1 어떤 일을 큐에 넘기는가

보고서 버튼을 누르면 이메일을 보내고 이미지를 렌더링하고 LLM을 호출한다. 전부 하나의 요청 안에서. 사용자는 30초를 기다렸다가 브라우저가 포기하면, 다시 누른다. 이메일 두 통, 이미지 두 장, LLM 두 번. 배경 작업 문제가 시작되는 모습은 대개 이런 형태. 원인은 거의 언제나 경계가 잘못된 데 있다.

큐는 빠르게 만드는 도구가 아니다. 지금 당장 하지 않아도 되는 일을 요청 밖으로 내보내는 장치다. 그래서 첫 번째 질문은 언제나 같다. 어떤 일이 동기 경로를 타고 어떤 일이 큐로 가는가.

1.1 두 개의 경로

동기 경로는 단순하다. 요청이 들어와, 처리되고 결과가 돌아간다. 그 사이에 사용자의 화면이 막혀 있다. 조회, 검색, 결제 승인은 이 경로에 잘 맞는 일이다. 사용자가 그 자리에서 결과를 기다리고 처리가 빠르고 결정을 내리기 쉽다.

비동기 경로는 "받았다"는 사실만 즉시 돌려준다. 일은 나중에 일어나고 결과는 이메일이나 상태 변경, 아니면 푸시로 도착한다. 사용자의 대기 시간은 거의 0이 되지만 새로운 계약이 생긴다. 언제 끝나는가, 실패하면 어떻게 되는가. 큐가 지켜야 할 계약은 바로 이 둘.

둘 중 어느 쪽이 더 모던하다는 문제가 아니다. 사용자가 얼마나 기다려줄 수 있는가, 그리고 서비스가 일을 늦게 해도 되는가가 문제다.

1.2 작업을 나누는 다섯 가지 질문

새로운 일이 생겼을 때, 경로에 넣기 전에 질문을 다섯 가지 한다.

첫째, 사용자가 지금 결과를 봐야 하는가. 화면을 보며 기다리고 있다면 동기다. 2초가 걸려도. 비동기는 사용자가 자리를 뜬 뒤 알아서 처리해도 되는 일에만 쓰는 것이다.

둘째, 얼마나 걸리는가. 대략적인 가이드라인은 이렇다. 100ms 이하는 동기여도 문제없고 1초를 넘으면 큐 후보다 [추정]. 그 라인 자체보다 중요한 것은, 처리 시간이 내 손에서 벗어난다는 사실이다. DB 조회나 메모리 계산이라면 동기다. SMTP나 이미지 서비스, LLM API의 속도에 의존한다면 큐다. 걸리는 시간이 외부 서비스의 그날 컨디션에 따라 흔들리면, 그 일은 요청 경로에 두어선 안 된다.

셋째, 실패해도 다시 해도 되는가. 가장 중요한 질문이다. 안전하게 재실행할 수 있는 작업이 큐에 갈 수 있는 작업이다. 다시 하면 두 번 청구가 되거나 두 통의 영수증이 나가는 작업이라면, 멱등하게 만들거나, 아니면 동기에서 처리하고 실패를 다른 방식으로 감당한다.

넷째, 외부 한도가 있는가. LLM API, 이메일 발송, 결제 제공자에는 모두 할당이 있다. 동기 경로에서 할당에 걸리면 사용자의 요청이 그 자리에서 죽는다. 큐에서 걸리면 그 작업만 늦어진다. 한도에 부딪히는 일은 큐의 일이다.

다섯째, 지금 안 하면 값이 아끼는가. LLM 추론은 대표적이다. “있으면 좋은” 요약이라면, 밤에, 혹은 배치로 돌릴 수 있다. 큐는 비용을 협상할 시간을 준다. 같은 출력을 더 싼 시간에 산출하는 것은 동기 경로에서는 불가능하다.

1.3 유형별 적용

1인 개발자가 자주 만나는 일들을 이 질문으로 돌려보자.

- 결제: 승인은 동기다. 사용자가 카드 결과를 그 자리에서 받아야 한다. 그런데 승인이 성공한 뒤의 "영수증 발급, 알람, 정산"은 큐다. 재시도해도 안전하고 10분 늦어져도 아무 손해가 없다.
- 이메일: 항상 큐다. SMTP는 느리고 발송 서비스에는 시간당, 하루 할량이 있으며 실패가 잦다. 재시도가 본업인 작업이다.
- 이미지 처리: 큐다. 리사이즈, 썸네일 생성, OCR은 프로세스를 오래 점유한다. 요청 경로에서 돌리면 다른 사용자가 전부 막힌다.
- LLM 추론: 큐다. 한도와 비용과 배치가 모두 해당되고 걸리는 시간이 프롬프트에 따라 크게 흔들린다.
- 조회와 검색: 동기다. 캐시를 붙여도 동기다. 사용자의 대기는 용인되지 않으며 처리는 빠르고 결정적이다.

작업	기본 경로	이유
결제 승인	동기	결과를 지금 받아야 함
영수증 발급	큐	재시도가 안전함
가입 이메일	큐	느리고 한도 있음
아바타 처리	큐	프로세스를 오래 점유
LLM 요약	큐	한도와 비용
목록 조회	동기	기다림이 불가함

1.4 반대편의 함정

함정은 두 반대편에 있다. 그리고 둘 다 흔하다.

동기 체인은 하나의 요청 안에서 외부 서비스를 연달아 부르는 모양이다. 이메일 1초, 이미지 2초, LLM 5초. 합은 8초이고 타임아웃이 먼저 온다.

사용자가 다시 누르면, 이미 성공한 부분이 또 실행된다. 실전 규칙은 하나다. 하나의 요청은 느린 외부 호출을 한 번까지. 그 넘은 부분은 전부 큐다.

큐 만능주의는 반대쪽이다. 단순한 조회까지 큐에 넣고 사용자로 하여금 100ms면 끝날 일을 "처리 중" 화면에서 기다리게 한다. 비동기는 일을 미루는 기술이지, 만능의 미덕이 아니다. 사용자가 기다리는 일은 돌려주면 된다. 기다리는 일을 큐에 보내면 사용자 경험의 버그가 된다.

그리고 하나 더. 침묵하는 비동기다. 일을 큐에 넣고 사용자에게 아무것도 말하지 않는다. 사용자가 받은 건지 잃어버린 건지 모른다. 비동기로 보내면 "받았다"와 "끝났다"를 어딘가에서 보여줘야 한다. 이메일, 상태 변경, 단순한 폴링. 침묵은 비동기 경로에서 가장 큰 버그다.

1.5 전달의 형태

"받았다"와 "끝났다"를 어떻게 보여줄 것인가는 옵션이 아니라, 비동기 경로에 붙이면 생기는 필수 비용이다. 흔한 선택지는 네 개인데, 각각의 성격이 다르다.

이메일은 가장 단단하다. 사용자가 자리를 떠도 도달하고 구현 비용이 적다. "끝났습니다" 알림의 기본값이다.

상태 필드는 사용자가 돌아와서 다시 확인하는 작업에 쓰는 용도다. DB에 "처리 중", "완료"를 적어 두고 페이지에서 보여준다. 짧은 폴링을 붙이면 몇 분 안에 다시 볼 작업은 여기로.

WebSocket은 사용자가 그 자리에 머물면서 과정을 실시간으로 보고 싶어 할 때다. LLM 생성을 토큰 단위로 보여 주는 화면이 대표 예다. 다만 1인 팀에는 상태 관리, 재연결, 타임아웃까지 움직이는 부분이 가장 많은 선택지다. 처음부터 여기로 가지 않는다.

원칙은 이렇다. "끝남"은 이메일이나 상태로 충분하다. "그 자리의 진행"이 필요하면 그때서야 WebSocket을 고려한다. 그리고 어느 방식을 쓰든 "언제 받았고, 언제 끝났는가"를 로그에 적는다. 그것이 "왜 이렇게 걸렸지?"에 나중에 답할 수 있는 전부다.

1.6 미니 사례: 요약 기능을 붙일 때

구체적인 상황에 적용해 보자. 기존 SaaS에 "문서 상단에 3문장 요약"을 추가하고 싶다.

사용자가 지금 결과를 봐야 하는가. 페이지는 요약을 보여줘야 한다. 그래서 읽기는 동기다. 그런데 요약을 만드는 것은 다르다. 5초에서 20초, LLM API에는 한도가 있다. 그래서 생성은 큐다.

형상이 이렇게 된다. 사용자가 페이지를 열면, DB에 요약이 아직 없으면 "요약 생성 중"을 보여준다. 요약 작업이 끝나면 DB를 갱신한다. 다음 페이지 조회, 혹은 짧은 폴링에서 요약이 나온다. 처음 한 번은 약간 늦지만 그 뒤는 즉시다. 그리고 LLM API가 막혀도 페이지가 죽지 않는다. "요약 생성 중"이 조금 더 오래 보일 뿐이다. 실패가 배경 안에 머무는 것이다.

그리고 실패 시의 모양도 정해 둔다. LLM API가 한동안 죽어 있으면, 요약은 영원히 "생성 중"이 되어선 안 된다. 페이지는 요약 없이도 완전히 동작한다. 요약이 없는 문서를 보여주는 것이, "요약 생성 중"이 영원히 붙어 있는 것보다 낫다. 그래서 요약 작업에는 "최대 N일" 같은 유효기간을 두거나, 실패가 쌓이면 그 문서의 요약 요청 자체를 중단하고, 나중에 다시 시도한다. 비동기 경로에도 종료 조건이 필요하다. 영원히 진행 중인 작업은, 멈춘 작업의 UX 버전이다.

이것이 전형적인 모양이다. 읽기는 동기, 쓰기는 비동기. 사용자는 읽기만 기다리고 쓰기에는 절대 기다리지 않는다. "이건 큐를 넣어야 하나"라는 질문의 대부분은 이 형태로 다시 쓰면 풀린다.

1.7 두 번째 미니 사례: 구독 갱신

두 번째 예로 구독 갱신을 보자. 매달 카드 결제를 해야 하는 제품이다.

갱신은 사용자가 요청을 보내는 일이 아니다. 시간에 의해 발생하는 일이다. 즉 "사용자가 결과를 기다리는 화면" 자체가 없다. 경로는 처음부터 큐다.

정교한 지점은 여기다. 청구는 멱등하지 않은 사이드 이펙트다. 같은 갱신을 두 번 청구하면 안 된다. 그래서 순서가 중요하다. 먼저 갱신 기록을 만든다. 사이클당 하나, 유니크하게. 그 다음 "이 갱신을 청구하라"는 작업을 큐에 넣는다. 멱등성 키는 갱신 기록의 id. 청구가 성공한 뒤 워커가 죽어도, 회수된 작업은 키 확인에서 "이미 했다"로 막힌다.

실패하면? 카드가 만료되었을 수 있다. 이것은 재시도 스케줄의 클래식한 경우다. 바로, 1시간 뒤, 익일, 3일 뒤. 고정 간격으로 N회까지 시도하고 그래도 안 되면 "정기결제 실패" 상태로 옮겨, 이메일을 보내고 구독을 일시 정지한다. 이 "N시간 뒤에 다시"도 전부 큐 작업이다. 지연 엔큐, 그것뿐이다. 재시도 스케줄은 큐 위에서 자연스럽게 생기는 구조이지, 코드 속의 while이 아니다.

이 사례가 주는 교훈은, 경계가 "사용자의 화면"이 아니라 "시간"으로 정해지는 일도 있다는 점이다. 그리고 그 일에서 진짜 문제는 경로 선택이 아니라, 실패를 어떻게 계급화하고 스케줄에 얹을 것인가다.

1.8 경계는 정한 뒤에도 확인한다

경계를 정하면 끝이 아니다. 실제로 잘 맞는지 확인하는 단계를 하나 두자. 확인은 세 가지다.

첫째, 요청 시간을 재본다. 동기 경로에 남긴 작업들이 여전히 목표 시간 안에 들어오는가. "결제는 3초 이내"라고 정했다면, 실제로 p95가 2.8초인가. 경계는 시간의 문제다. 시간 데이터 없이는 경계가 추측이다.

둘째, “처리 중” 화면의 체류 시간을 본다. 비동기로 넘긴 작업에서 사용자가 “처리 중”을 보는 시간이 얼마나 되는가. 몇 초라면 괜찮고 몇 분이라면, 전달 수단이 모자란 것이다. 상태 폴링을 붙이거나, 이메일 알림을 추가하는 타이밍이다.

셋째, 샘플로 끝에서 끝을 따라간다. 실제 사용자의 한 요청을 골라서, 동기 부분이 어디에서 끝나고 큐 부분이 어디에서 시작했는지, 언제 끝났는지 시간 순서로 적어 본다. 한 번 해 보면, 설계도랑 다른 지점이 대개 하나쯤 나온다.

이 확인을 한 달에 한 번, 혹은 기능 변경 후에 한다. 제품의 모양이 바뀌면, 경계도 다시 보아야 한다.

1.9 흔한 질문 두 개

결정 뒤에는 대개 같은 질문이 두 개인다.

“웹후크는?” 웹후크는 큐의 대안이 아니라, 제공자 쪽에서 “일이 끝났습니다”를 내 쪽으로 보내주는 장치다. 웹후크로 들어온 일은 중복과 위조를 검증하고, 실제 처리는 큐에서 한다. “웹후크를 받았으니 요청 안에서 처리하겠다”는 생각은, 동기 체인 문제와 똑같은 방식으로 돌아온다.

“타이머가 필요한 일(매일 3시 정산, 매시간 리포트)은?” 타이머는 트리거일 뿐이다. 타이머가 울리면 큐에 작업 하나를 넣는 것까지가 타이머의 몫이고, 그 뒤는 전부 큐의 문제다. 타이머 코드 안에서 정산 로직을 돌리는 것은, 동기 체인 문제와 똑같은 방식으로 돌아온다.

1.10 경계 정하기 체크리스트

끝으로 짧은 목록이다. 하나하나 대답할 수 있으면, 경계는 제자리에 있다.

- 사용자가 화면을 보며 기다리는가. 그렇다면 동기다.

- 처리 시간이 외부 서비스에 좌우되는가. 그렇다면 큐다.
- 한도에 부딪히는가. 그렇다면 큐다. 가능하면 배치도.
- 실패해도 재실행이 안전한가. 아니라면, 먼저 멱등하게 만들고 나서 큐다.
- 늦어져도 되는가. 안 되면 동기다.
- 사용자가 "받았다"를 확인할 수 있는가. 아니라면, 전달 수단을 추가한다.

경계를 정하는 것이 운영의 절반이다. 경계가 정해지면, 그다음에는 작업의 모양과 실패의 처리가 순서대로 따라온다. 그것이 다음 장의 이야기다.

2 등록에서 실행까지, 파이프라인의 형태

경계가 정해졌으면 다음 질문은 모양이다. 큐는 결국 목록인데, 배경 작업 파이프라인의 품질은 두 가지가 정한다. 들어가는 작업의 모양, 그리고 그 작업을 꺼내서 돌리는 워커의 루프.

2.1 작업은 재시도의 단위

작업의 크기가 실패의 파급 범위를 정한다. 작업은 안전하게 다시 실행할 수 있는 한 단위의 조작이어야 한다. 세 가지 조건이 있다. 입력이 명확하고 조작이 하나이며 완료를 검증할 수 있어야 한다.

흔한 반례는 묶어 넣기다. "주문 처리"라는 작업에 결제 확인, 발송, 이메일, 적립을 모두 담는 경우. 마지막에 이메일이 실패하면 재시도 시 전체가 다시 돌고, 발송은 두 번 일어난다. 작게 쪼개자. 사이에 의존성이 있으면, 조정은 별도의 부모 작업이나 상태 기계를 맡기고, 자식들은 각자 재시도 가능하게 만든다.

다른 하나의 원칙도 중요하다. 페이로드에는 참조를 넣는다. 작업에 사용자 기록의 복사본을 담아 두면 안 된다. user_id만 넣고 실행할 때 DB에서 최신으로 가져온다. 큐 속에 고인 복사본과 DB의 상태가 다른 것은 찾기 힘든 버그의 원천이다. 큐는 "무엇을 할 일"을 전하는 곳이고 "지금은 어떤 상태"는 DB가 담당한다.

좋은 작업	나쁜 작업
이메일 1통 전송	주문 전체 처리
이미지 리사이즈 1개	결제 + 발송 + 적립
LLM 호출 1회	보고서 전체 생성
해지 절차 1단계	고객에게 연락하기

"고객에게 연락하기"는 나쁜 작업이다. 무엇을, 어떤 식으로, 그리고 어떻게 완료가 되는지 알 수 없다. 좋은 작업은 목적어가 하나인 동사다.

2.2 등록, 즉 엔큐

엔큐는 작업을 큐에 쓰는 행위다. 가볍게 만든다. 엔큐 시점에 파일을 가져오거나 API를 부르는 일이 필요하다면, 그 일 자체를 별도 작업으로 만든다. 엔큐는 빠른 기록이어야 한다. 요청 경로에서 오래 걸리는 순간, 그것은 다시 동기 체인이 된다.

페이로드의 크기에도 선을 긋는다. 큐는 작은 구조화된 데이터를 실어 나르는 곳이지, 파일 저장소가 아니다. 이미지 바이트나 수십 KB 되는 JSON 덩어리를 페이로드에 담아두면, 세 가지 손해가 생긴다. 큐 저장소가 부풀고 로그에 페이로드는 찍히고 중복 실행할 때마다 같은 큰 몸체를 다시 실어 나른다. 큰 몸체는 스토리지에 두고 페이로드는 그 참조(id, URL, 버전)만 넣는다. 경험적인 한계로, 페이로드는 몇 KB 안에 머무는 것을 목표로 한다. 그보다 커진 작업은 "작업을 크게 만들었다"는 신호다.

작업의 외곽, 즉 컨베이어에 들어가는 기본 필드는 대개 이렇게 생긴다.

- id: 작업 자체의 식별자.
- type: 어떤 핸들러를 돌릴지.
- payload: 입력. ID와 최소한의 파라미터.
- created_at: 등록 시각. 최고령 대기 지표의 기준.
- attempts: 몇 번 실행되었는지.
- priority: 배치 구분용. 선택 사항.
- idempotency_key: 업무적인 중복 제거 키.

두 가지를 빼먹지 않는다. 하나는 시크릿을 페이로드에 넣지 않는 것. API 키를 큐 속에 하루 묵히면, 언젠가 로그 덤프에 찍힌다. 다른 하나는 중복 제거 키를 업무 쪽에서 만든다는 것. "주문 123번의 영수증 이메일"의 키는

order:123:receipt 이런 식이다. 큐 시스템이 만들어 주지 않는다. 이 키가 있어야 중복 엔큐와 중복 실행이 "이미 했다"로 판정된다.

검증의 시점도 정해 둘 일이다. 엔큐 시점에 무거운 검증을 하지 않는다. 엔큐는 빠르게. 검증은 실행 시점에 한다. 이유는, 실행 시점의 세계가 지금의 세계이기 때문이다. 엔큐 때는 통과했던 데이터가 실행 때쯤이면 삭제되었거나, 상태가 바뀔 수 있다. "직전에 확인"하는 것만이 의미가 있는 검사다. 둘 다 할 수는 있지만 둘 다 하면 엔큐가 느려져서 요청 경로가 끌려간다. 그 대가를 알았을 때만 둘 다 하라는 뜻이다.

2.3 작업의 이름과 버전

type 필드는 계약이다. 큐에 적힌 이름은, 다른 날, 어쩌면 다른 버전의 코드에 의해 실행된다. 그래서 작업을 바꿀 때는, 기존 핸들러를 그 자리에서 바꾸지 않는다. 새로운 type을 추가한다.

예를 들어 email:v1의 로직을 바꾸고 싶다면, email:v2를 쓰고 엔큐 쪽을 v2로 넘긴다. 큐에 남은 v1 작업들은 옛 핸들러가 끝마치고 새 작업은 전부 v2로 들어온다. 중간에 섞일 일 없다. v1이 완전히 비었음을 확인한 뒤, 옛 핸들러를 지운다.

이것은 돌아가는 길처럼 보이지만 배포 날의 가장 덜 아픈 방법이다. 핸들러를 그 자리에서 바꾸면, 큐에 남은 작업이 "바꿈 이전"인지 "이후"인지 구분할 수 없게 된다. 같은 이름의 작업이 배포 시각에 따라 다른 행동을 하는 파이프라인은, 고장 났을 때 추리부터 시작해야 한다. type 버전화는 그 추리를 없애는 답이다.

2.4 워커 루프

워커는 작업을 꺼내서 돌리는 반복이다. 가져오기(클레임)를 하고 실행하고 확인(ack)한다. 여기서 핵심은 클레임의 의미다. 워커가 작업을 꺼내면,

실행 중에는 다른 워커에게는 보이지 않는다. 가시성 타임아웃이 이 "보이지 않는 시간"을 결정한다. 워커가 ack 전에 죽으면, 타임아웃이 지나자 그 작업이 다시 돌아오고 다른 워커가 다시 실행한다.

즉 실행 보장 의미는 "최소 한 번"이다. 같은 작업은 두 번 돌 수 있다. 이것은 피해야 할 결함이 아니라, 설계 전제다. 모든 사이드 이펙트는 두 번 와도 무해해야 한다. 다음 절의 멱등성이 이 전제를 지탱한다.

뼈대만 추리면 이 모양이다.

```
loop:
 job = queue.claim(visibility_timeout)
 if job == null:
 sleep(0.5)
 continue
 try:
 if done(job.idempotency_key):
 ack(job)
 continue
 result = handlers[job.type](job.payload)
 mark_done(job.idempotency_key, result)
 ack(job)
 except RetryableError as e:
 if job.attempts >= MAX_ATTEMPTS:
 move_to_dlq(job, e)
 else:
 requeue(job, delay=backoff(job.attempts))
 except FatalError as e:
 move_to_dlq(job, e)
 finally:
 log(job.id, job.type, job.attempts, duration, outcome)
```

동시성은 워커당 동시에 돌리는 작업 수다. 작게 시작한다. LLM 호출이

메인인 워커는 동시성 1에서 2로 충분할 때가 많다 [추정]. 동시성을 올리는 것은 쉽지만 한도에 부딪히고 나서 낮추는 것은 고통스럽다. 작업별 타임아웃도 필요하다. 한도를 넘긴 작업은 실패로 처리해서 재시도 경로로 넘긴다. 멈춘 작업의 문제는 다음 장에서 자세히 다룬다.

2.5 재시도와 백오프

재시도하기 전에 오류를 분류한다. 재시도 가능한 쪽은 5xx, 네트워크, 타임아웃, 일시적인 할당 고갈이다. 재시도 불가능한 쪽은 400, 잘못된 입력, 권한 부족, 데이터 누락이다. 둘째 쪽은 10번을 돌려도 10번 똑같이 실패한다. 재시도 슬롯을 먹이지 말고 곧바로 데드레터로 보내자.

재시도 가능한 쪽은 지수 백오프에 자기를 얹는다. 1초, 4초, 16초, 64초, 그리고 그 위에 무작위 값을 더한다. 자기가 없으면, 한꺼번에 실패한 작업이 같은 순간에 다시 몰려가서 외부 서비스를 또 때린다. 무리가 붕괴하고 재실행이 또 무리가 되는 악순환이다. 자기는 그것을 퍼뜨리기 위한 것뿐이다.

구체적으로는 이렇게 한다. 지연 시간은 base에 2의 시도 횟수 거듭제곱을 곱한 값이다. 그 위에, 지연 시간의 절반만큼 무작위 값을 얕는다. 1초, 2초, 4초, 8초가 기본 궤도이고 각 단계에서 절반 이하의 랜덤이 붙는다. 상한(cap)도 둔다. 재시도 간격이 10분을 넘어서는 것은, 대개 이미 "지연 작업"이지 "재시도"가 아니다. 그 순간에는 DLQ에 두는 쪽이 더 정직하다.

base는 작업의 성격에 따라 다르다. 외부 서비스 장애를 기다리는 작업은 base를 크게(10초부터), 순간적인 과부하를 피하는 작업은 작게(1초부터) 시작한다. 둘 다 0부터 시작하면, 전자는 서비스 복구 전에 여러 번 허탕을 치고 후자는 같은 순간에 다시 몰려간다.

MAX_ATTEMPTS에는 상한을 반드시 둔다. 재시도로 고쳐지지 않는 작업은 재시도 문제가 아니다. 데이터나 코드 문제이고 그 해결 장소는

데드레터다. 시도 횟수가 높다는 것은 성실한 것이 아니라, 자원을 낭비하고 있는 신호다.

2.6 순서

배경 파이프라인에서 전역 순서를 쫓지 말자. 비쌈은 물론이고 대부분의 제품은 그것을 필요로 하지 않는다. 필요한 것은 같은 키 안의 순서다. user_id나 order_id를 파티션 키로 쓰면, 같은 키의 작업은 차례대로, 다른 키는 병렬로 돈다. "이 주문의 결제, 발송, 이메일"은 순서가 보장되고 다른 주문은 기다리지 않는다.

주의는 두 가지다. 순서가 있어도 중복 보호는 필요하다. 순서는 순서 뒤집힘을 막을 뿐, 두 번 실행을 막지는 못한다. 멱등성 키는 여전히 있어야 한다. 또, 순서가 있는 파티션에 독소 작업, 즉 항상 실패하는 작업이 들어가면 파티션 전체가 멈춘다. 그래서 시도 횟수 상한과 데드레터는 옵션이 아니다. 순서를 쓰는 순간 필수 조건이 된다.

2.7 우선순위의 첫 결정

우선순위는 나중에 붙이는 기능이 아니라, 처음에 두 줄로 정하는 것이 좋다. "긴급"과 "보통" 두 단계. 긴급은 사용자가 지금 화면에서 기다리는 작업(영수증, 승인 알림)이고 보통은 나머진다.

동작 방식은 간단하다. 워커는 먼저 긴급 큐를 비우고 비면 보통 큐로 간다. 같은 단계 안에서는 도착 순서다. 이렇게 하면, 두 가지 문제가 함께 해결된다. 중요한 일이 밀리지 않고 복잡하지도 않다.

삼가야 할 것은 단계 수를 늘리는 것이다. 5단계, 10단계로 만들면, 두 가지 손해가 생긴다. 한 단계의 양이 워커 용량보다 작으면, 그 단계의 작업은 다음 단계로 내려오지 못해 영원히 대기한다(기아 현상). 또, "어떤 단계인지"를 정하는 판단이 코드에 흩어지면서, 경계 결정이 매일 다시

일어나게 된다. 1인 팀에는 2단계가 실전이다. 3단계가 필요해지는 순간, 그것은 이 책의 범위 밖인 성장 신호다.

그리고 우선순위를 붙였으면, 지연 엔큐도 함께 쓸 수 있다. "30분 뒤에 다시"는 별도 스케줄러가 아니라, 30분 뒤의 큐에 넣는 작업일 뿐이다. 구독 갱신의 재시도 스케줄, 밤 배치 시작, 타이머 트리거. 전부 같은 메커니즘으로 돌아간다.

2.8 중복과 멱등성

중복은 세 곳에서 온다. 워커의 재시도, 중복 엔큐, 그리고 가시성 타임아웃에 의한 회수(워커 사망). 셋 다 막을 수 없다. 큐 시스템은 본질적으로 "중복"과 "처음"을 구별하지 못한다. 그래서 사이드 이펙트를 두 번 와도 무해하게 만드는 쪽으로 간다.

보통 쓰는 보호 장치는 세 가지다. 첫 번째는 DB의 유니크 제약. 주문당 영수증은 하나이므로, 두 번째 삽입은 그냥 실패한다. 두 번째는 상태 확인. 이미 발송이라면 다시 발송하지 않는다. 세 번째는 idempotency_key 테이블. 첫 실행에 키를 적어 두고 두 번째가 오면 건너뛴다. 멱등적으로 만들 수 없는 외부 호출, 예를 들어 카드 청구라면, "진행 중"을 먼저 기록하고 나중에 검증하거나, 제공자의 멱등성 기능을 쓴다.

외부 호출마다 제공자 쪽 멱등성이 따로 있다. 결제 제공자는 idempotency key를 받고, 이메일 서비스는 메시지 식별자를 중복 판단의 기준으로 쓴다. 이것을 쓰면 "두 번 청구" 문제를 제공자에게 맡길 수 있다. 기능이 없는 곳은, 결국 "진행 중" 기록이 전부다. 그래서 멱등성 키를 업무 쪽에서 만들어야 한다. 제공자에게 넘길 값이 되어야 하니까.

끝으로 이 장의 요약을 한 줄씩 남긴다. 작업의 모양이 실패의 파급 범위를 정하고 워커 루프가 죽음의 순간에 무엇이 일어나는지를 정하며, 중복 제거 키가 중복이 해롭지 않은지를 정한다. 이 셋을 기억하고 있으면, 혼자

짠 파이프라인은 쉽게 무너지지 않는다. 그리고 정말로 무너지는 순간의 이야기는, 다음 장에서 한다.

3 멈추고 죽는 작업, 데드레터까지

배경 파이프라인은 조용하다. 아무 일이 없으면 존재를 느끼지 못한다. 그래서 진짜 실력은 무언가 죽는 순간에 드러난다. 일은 네 가지 사각지대에서 사라진다. 워커가 죽는 것, 작업이 실패하지 않고 멈추는 것, 작업이 영원히 재시도되는 것, 그리고 ack과 사이드 이펙트가 어긋나는 것. 하나하나 보고 각각을 어디다 걸고넘여 잡는지 정해 둔다.

3.1 워커의 죽음

배포, OOM, 패닉. 워커가 작업을 실행하다가 죽는다. 그 작업은 ack되지 못했고 가시성 타임아웃이 지나면 다시 돌아와, 다른 워커가 다시 실행한다. 이 회복은 필수이며 대가는 중복 실행이다. 사이드 이펙트가 두 번 일어날 수 있다.

바로 그래서 앞 장의 "최소 한 번"과 "멱등성"이 파이프라인의 핵심이 된다. 재실행을 막을 수 있는 방법은 없다. 재실행을 무해하게 만들 뿐이다.

가시성 타임아웃은 범위를 가진 값이다. "작업의 최대 실행 시간보다 길어야" 한다. 타임아웃이 30초이고 작업이 40초가 걸리면, 멀쩡한 워커의 작업이 다른 워커한테 탈취당한다. 동시에 "잃을 수 있는 시간보다 짧아야" 한다. 타임아웃이 1시간이면, 죽은 작업을 1시간 동안 찾지 못한다. 합리적인 시작점은 p95 실행 시간의 2에서 3배다. 관측하면서 점차 조인다 [추정].

워커가 하나뿐인 팀에는 한 줄을 더 보태자. 재시작을 누가 할 것인가. 워커 하나가 죽으면, 그 워커를 살리는 것도 누군가의 몫이다. 프로세스 매니저(systemd 같은 것)가 죽은 워커를 다시 띄워 주도록 해 두는 것은, 큐 운영의 기본 장비다. 매니저가 없다면, 일정 간격으로 "워커가

살아있는가"를 확인하는 워치독 크론 하나다. 둘 다 없으면, 워커 사망은 사람 눈으로 알아차리는 것뿐이다. 그리고 사람들은 새벽에 깨어 있지 않다.

사망에는 또 다른 종류가 있다. 메모리다. 큰 이미지를 다루거나, LLM 응답을 스트리밍으로 받아 쌓는 워커는, 실행 중반에 OOM으로 죽을 수 있다. 이런 죽음은 프로세스가 막 살던 바로 직전에 일어난다 보니, 더 찾기 어렵다. 처방은 설계 단계에서 해야 한다. 입력에 크기를 제한하고 큰 출력을 스트리밍으로 흘려 보내고 워커가 N개 작업을 다룬 뒤 스스로 재시작하게(recycling) 둔다. 워커의 메모리 사용이 오르기만 한다면, 그것은 언젠가 죽을 신호다. 스스로가 죽기 전에 원인을 찾아야 한다.

3.2 멈춘 작업

사각지대 중 가장 교활한 것이 이것이다. 예외가 없고 실패도 없고 ack도 없다. 그냥 멈춰 있다. 데드락, 아주 느린 외부 응답, 잠금 대기. 워커는 멀쩡해 보이고 프로세스는 죽지 않았으며 작업은 그저 움직이지 않는다.

탐지는 두 종류다. 첫 번째는 작업별 타임아웃이다. 실행에 시한을 주고 넘기면 실패로 처리해서 재시도 경로로 넘긴다. 외부 호출에 타임아웃을 두는 것은 기본 위생이고 그 부재는 버그다. HTTP에도, DB에도, 파일 읽기에도. 두 번째는 오래된 작업 스캔이다. claimed_at이 오래되고 ack이 안 온 작업을 백그라운드 프로세스가 정해진 간격으로 찾는다. 찾으면 로그를 남기고, 필요하면 워커를 재시작한다.

처리는 단순하다. 워커를 죽인다. 멈춘 작업을 품고 있는 워커는 어차피 건강하다고 보기 어렵다. 깔끔한 재시작은 안전하다. 그 작업은 가시성 타임아웃으로 회수되니까. 중요한 것은 알아차리는 일이다. 알아차리지 못한 멈춤은 파티션 전체 정지와 같은 효과를 내며, 큐는 그저 "느리다"는 모습으로 남는다.

멈춤의 원인은 크게 두 갈래로 나뉜다. 내 쪽의 멈춤(데드락, 걸린 쿼리, 연결 풀 고갈)과 바깥쪽의 멈춤(응답이 늦어지는 외부 서비스). 전자는 타임아웃으로 잘라 내면 되지만 후자는 더 신중하다. 외부 서비스가 "느린" 것과 "죽은" 것을 구별 못 하면, 타임아웃을 짧게 두느라 멀쩡한 큰 파일도 자꾸 실패로 만들게 된다. 그래서 외부 호출은 두 값으로 둔다. 응답 헤더를 받을 때까지의 시간(connect), 그리고 본체를 다 받을 때까지의 시간(read). 둘 다 타임아웃이 필요하고 둘 다 로그에 남는다.

3.3 무한 재시도, 독소 메시지

영원히 성공하지 않는 작업이 있다. 잘못된 페이로드, 외부 필드 누락, 코드 버그. 실패하고 재시도하고 다시 실패하고 다시. 상한이 없으면 이 루프는 계속되고 워커의 시간을 태운다. 순서가 있는 파티션이라면 파티션 전체를 막는다.

대응은 앞 장에서 세운 시도 횟수 상한과 데드레터다. 독소 메시지는 그것이 필수인 이유다. MAX_ATTEMPTS는 관대하게 잡을 파라미터가 아니라, 퓨즈다. 퓨즈가 끊기면 그 일은 데드레터로 옮겨지고, 큐는 정상 흐름으로 돌아온다.

3.4 ack과 사이드 이펙트의 간극

순서가 문제다. 사이드 이펙트를 끝낸 뒤, 그때서야 ack한다. ack을 먼저 하면, 사이드 이펙트가 끝나는 사이에 프로세스가 죽으면 일은 사라진다. 재시도도, 데드레터도 없다. 그냥 증발이다. 가시성 타임아웃이 보호하는 것은 ack 이전의 시간뿐이기 때문이다.

사이드 이펙트가 길다면, 외부 API 호출 같은, DB에 "진행 중"을 먼저 적어 둔다. ack은 "진행 중"을 "완료"로 바꾸는 행위다. 이렇게 하면 워커가 죽어도, 진행 중 기록으로 일을 찾아내서 보상할 수 있다. 짧게 말하면,

ack은 “이 일은 안전하게 기록되었다”는 완료 신호다. 일을 끝낸 신호가 아니다.

3.5 실패를 로그에 남긴다

작업이 실패하면 남는 것은 로그 한 줄이다. 그 줄의 내용이, 일주일 뒤에 디버깅이 가능할지를 정한다. 최소 필드는 이렇게 된다. 작업 id, type, 시도 횟수, 오류 종류(재시도 가능 여부), 오류 메시지, 실행 시간, 그리고 correlation id.

오류 종류가 특히 중요하다. DLQ를 볼 때 묻는 질문은 “이건 내 버그인가, 외부의 사고인가”다. 그 분류가 로그에 이미 적혀 있으면, 답은 즉시 나온다. 없으면 메시지에서 추리해야 하고 디버깅은 바로 거기서 시간을 잃는다.

correlation id는 엔큐부터 ack까지를 하나로 잇는 값이다. 요청 헤더에 담아 두고 외부 호출에도 붙여 보낸다면, 값 하나로 전체 이야기를 찾을 수 있다. 1인 팀에서 이것은 “조사 가능한 시스템”과 “기대밖에 할 수 없는 시스템”의 차이다.

3.6 네 사각지대를 한 장에

사각지대	증상	대응
워커 사망	작업이 두 번 실행	멱등성 키 + 가시성 타임아웃
멈춤	오래 claim된 작업	작업별 타임아웃 + stale 스캔
독소 메시지	재시도만 쌓임	시도 상한 + 데드레터
ack 후 낙마	이펙트 불완전	이펙트 완료 후 ack

3.7 데드레터는 대기실이다

데드레터 큐, 즉 DLQ는 실행하지 못한 일이 내려앉는 곳이다. 흔한 오해는 그것을 무덤으로 다루는 것이다. "실패했으니 뭐." 그렇게 두면 실패는 조용히 묻히고 몇 달 뒤에 "고객이 이메일을 못 받았다"는 형태로 다시 나온다. DLQ는 대기실이다. 판단이 내려지면 다시 받을 작업이 잠깐 머무는 곳.

그러면 저장 요건이 생긴다. "왜 실패했는지"와 "지금 리플레이해도 되는지"를 판단할 수 있어야 한다.

- 원래 페이로드, 또는 그것의 참조.
- 마지막 오류와 그 종류. 재시도 가능했거나, 치명적이었거나.
- 시도 횟수와 마지막 실행 시각.
- idempotency_key와 trace id. 로그와 이어 주기 위해서.
- DLQ에 들어온 시각.

"실패했습니다" 한 줄만 있는 것은 DLQ가 아니라 쓰레기통이다.

DLQ를 보는 화면은 단순해도 된다. 큰 도구가 아니라, 위 필드를 목록으로 보여주는 표와, type과 시간대 필터면 충분하다. 중요한 행위는 그 화면을 정기적으로 여는 것이다. 주간 점검에서든, 매일 아침이든. 안 여는 DLQ는 없는 것과 같다.

3.8 리플레이 절차

DLQ의 일은 절차로 받을 뿐, 대충 전부 다시 넣어서는 안 된다.

먼저 진단. 오류 종류로 분류한다. 코드 버그, 나쁜 데이터, 외부 서비스 장애. 그 다음 원인을 고친다. 코드 버그라면 고쳐 배포하고 나쁜 데이터라면 처리 방식을 정한다. 셋째로 해당 작업만 골라낸다. 원인이 1시간짜리 외부 장애였다면, 그 시간대의 작업만. 넷째로 다시 엔큐한다. 멱등성 키가

있으니, 이미 끝난 사이드 이펙트는 두 번 일어나지 않는다. 다섯째로 확인. 몇 개의 결과를 눈으로 보고 DLQ가 비어 가는지를 본다.

리플레이 자체에도 순서와 묶음이 있다. 가장 오래된 것부터, 그리고 100개 단위로. 한꺼번에 1000개를 다시 넣으면, 그것 자체가 외부 서비스에 대한 스파이크가 된다. 방금 장애에서 회복한 제공자에게 1000개 청구가 동시에 떨어지는 셈이다. 100개를 넣고 재시도율과 성공을 5분간 살핀 뒤, 다음 묶음으로 간다. "천천히 다시"도 재시도의 한 종류다.

리플레이할 필요가 없는 작업도 판단이 필요하다. 나쁜 데이터, 유효기간이 지어진 업무. "처리하지 않고 닫는다"는 결정과 이유를 기록한다. 방치된 DLQ는 불안이 자라는 곳이고 처리된 DLQ는 기록이다.

3.9 실전 시나리오: SMTP 장애 2시간

지금까지 세운 것들이 실제로 어떻게 움직이는지, 하나의 시나리오로 돌려본다. 아침 9시에 이메일 제공자가 장애를 낸다. 모든 발송 작업이 5xx로 실패한다.

첫 5분: 재시도. 작업들은 재시도 가능한 오류로 실패하고, 백오프와 함께 큐로 돌아간다. 재시도율 지표가 뛴다. 알람이 여기에서 울린다면, 그 알람이 가리키는 곳은 내 코드가 아니라 제공자의 상태 페이지다.

30분: 큐가 늘어난다. 워커는 멀쩡하고 성공만 없는 상태다. 최고령 작업의 대기 시간이 오르기 시작한다. 그런데 시스템은 고장난 것이 아니라, 늦은 것이다. 이 구분이 큐의 전부이기도 하다. 장애는 이메일 경로 안에 갇혀 있고 결제, 이미지, LLM 경로는 평소처럼 돈다.

2시간: 제공자가 복구된다. 백오프를 기다리던 작업들이 차례로 다시 실행되고, 성공한다. 큐는 가장 오래된 것부터 비어 간다. 장애 동안 MAX_ATTEMPTS를 넘겨 DLQ에 떨어진 작업이 있다면, 그것들은 이제 성공할 것이다. 확인한 뒤 리플레이한다.

이 과정에서 사람이 하는 일은 네 가지다. 알아차리기, 원인 확인하기, 기다리기, 리플레이하기. 패닉도, 수작업도 없다. 파이프라인이 장애를 흡수하고 그것이 설계가 의도대로 일어난다는 뜻이다.

3.10 배포 날의 연습

배포는 워커가 정례적으로 죽는 시간이다. graceful shutdown은 큐 운영의 일부다. 순서대로. 새 작업 수신을 중단하고 진행 중인 작업의 완료를 제한 시간까지 기다리고, 시간을 넘기면 가시성 타임아웃의 회수에 맡긴다. 배포 중에 프로세스를 강제 종료하지 않는다. SIGTERM 처리는 소량의 코드다. 하지만 없는 배포는 매번 중복 실행의 도박이다.

이 장의 메시지는 짧다. 죽음과 멈춤과 실패를 막을 수는 없다. 미리 그 귀추를 정할 수 있을 뿐이다. 가시성 타임아웃이 죽음을 받고 타임아웃과 스캔이 멈춤을 받고 시도 상한이 독소를 받고 DLQ가 해결 불가능한 일을 받는다. 넷이 제자리에 있으면, “고장 난” 파이프라인은 추측해야 하는 대상이 아니라, 보고 고칠 수 있는 대상이 된다.

4 혼자서도 무너지지 않게: 관측과 운영

1인 팀에는 24시간 온콜이 없다. 새벽 3시에 큐가 막히면, 일어날지 말지를 그때서야 결정하게 된다. 그래서 관측의 설계 목표는 명확하다. 조치할 일이 있을 때만 깨우는 것, 그리고 낮에 5분이면 건강을 확인할 수 있는 것.

4.1 핵심 지표는 네 개

지표는 많이 필요 없다. 네 개면 충분하고 각자의 역할이 다르다.

첫째, 최고령 작업의 대기 시간. 정체의 진짜 계기다. 길이가 500개여도 1분 안에 비워진다면 괜찮고, 5개여도 3시간을 기다린 작업이 있다면 사고다. 그래서 "몇 개가 쌓였는가"보다 "쌓인 것 중 가장 오래된 것이 몇 시간을 기다렸는가"를 본다.

둘째, 처리 시간의 p95. 열화의 징조다. DB가 느려지고 외부 서비스가 느려지고 워커가 늙으면 여기부터 커진다. 작업 유형별로 보는 것이 적절하다. 이메일의 p95와 LLM의 p95를 합쳐서 보면 아무 의미가 없다.

셋째, 재시도율. 외부 불안정의 징조다. LLM API나 SMTP가 그날 컨디션이 나쁘면, 여기가 가장 먼저 반응한다. 재시도율이 갑작스레 오르면 대개 "외부 쪽이 아픈" 신호다. 내 코드보다는 그쪽을 먼저 확인하게 된다.

넷째, 데드레터 수. 조용한 실패다. 1이 의미 있는 유일한 지표다. 사용자가 요청한 일이 일어나지 않았다는 증거가 하나 생겼기 때문이다.

지표	알려주는 것	경계 예시 [추정]
최고령 대기	정체의 시작	1시간 초과
처리 p95	성능 열화	통상의 2배
재시도율	외부 불안정	5퍼센트 초과

지표	알려주는 것	경계 예시 [추정]
데드레터 수	조용한 실패	1건 이상

경계값은 시작점이다. 제품 리듬에 맞게 조이는 것. 밤에만 도는 LLM 배치와 분 단위 이메일의 "통상"은 같은 지표라도 다르다.

지표를 어디에 두느냐가 다음 문제다. 1인 팀은 도구에 과감히 적게 투자해도 된다. 네 수치를 한 화면에서 볼 수 있으면 충분하다. 대시보드 한 장, 혹은 주 1회 리포트를 뱉는 페이지, 더 단순하게는 노트의 표. 정해진 날에 보는 습관이 핵심이다. 네 수치를 10초 안에 못 찾으면, 보지 않게 되고 안 보는 지표는 장식이 된다.

4.2 알람을 적게, 정확하게

임계값을 정하는 사람이 나이고 그 값에 시달리는 사람은 미래의 나다. 그래서 기준은 단순하다. 깨어났을 때, 뭔가 할 수 있는가. "보다 생각해 보자"는 답이라면 그것은 낮에 볼 티켓이다.

1인 팀의 깨움 알람은 최소한으로 둔다. 실전적인 방법은 최악의 경우를 가려 보는 것이다. 제품이 망가졌는데, 하나의 지표만 알 수 있다고 하자. 무엇에 대해 깨우고 싶겠는가. 보통 2, 3개로 수렴한다. "최고령 대기가 X를 넘었다", "데드레터가 0이 아니다", "워커가 죽었다(하트비트 부재)". 나머지는 주간 리뷰로 보내자.

또 하나의 습관: 제품의 모양이 바뀌면 임계값을 다시 쓴다. 새 작업 유형을 넣으면, 재시도율과 p95의 통상값이 바뀐다. 한 달 전에 둔 값은 이제 노이즈다. 임계값은 대화다.

알람 피로도 진지하게 다룬다. 매일 울리는데 아무도(즉 나) 조치를 안 취하는 알람은, 다음 진짜 알람을 무시하게 훈련시킨다. 2주일 연속

노이즈였던 알람에는 세 갈래가 있다. 원인을 고치거나, 임계값을 올리거나, 지운다. 지워진 알람이 무시된 알람보다 낫다.

4.3 용량: 얼마나 버틸 수 있는가

용량은 간단한 곱셈과 나눗셈으로 산다. 1분당 처리 속도는 (워커 수 x 동시성)을 평균 처리 시간으로 나눈 값이다. 워커 하나가 4개를 동시에 돌리고 평균 작업이 30초면, 대략 1분에 8개, 1시간에 500개쯤 [추정].

이 숫자를 두 방향으로 쓴다. 하나는 "쌓인 것을 언제 비우는가"에 답하는 것. 큐 길이를 처리 속도로 나눈다. 2000개 쌓였고 1시간에 500개면 4시간. 그 4시간이 사용자 입장에서 허용되는가. 제품 질문이다. 다른 하나는 "급증이 오면 어떻게 되는가"에 답하는 것. 평소 트래픽의 두 배가 오면, 처리 속도가 그대로라면 대기 시간은 두 배다.

과부하가 올 때 레버는 세 개인데, 순서가 있다. 먼저 워커 수를 올린다. 제일 싸고 제일 빠르다. 그다음 배치를 줄인다. 밤 배치의 우선순위를 낮추거나, 잠시 멈추고 실시간 경로에 runway를 내준다. 마지막으로 용량을 사 온다. 외부 한도를 올리거나, 제공자를 하나 더 붙이는 것. 오늘 할 수 있는 것부터 시작하는 게 원칙이고, 용량 문제가 지표로 먼저 보이는 곳은 최고령 대기다.

4.4 LLM 부하만의 운영

LLM 작업에는 따로 관리해야 할 세 가지가 있다.

레이트 리밋 여력. 급증이 왔을 때 큐가 바로 쌓이지 않도록 여유를 둔다. 할당의 100퍼센트를 쓰면, 조금만 증가해도 지연이 되고 지연은 클레임이 된다. 실제로는 70에서 80퍼센트 이용률이 안전한 선이다 [추정].

한도는 숫자로 읽어야 한다. 제공자가 1분에 600회라고 하면, 초당 10회다. 내 작업 하나가 호출을 두 번 한다면, 초당 5개까지 안전하다. 이 숫자를

알아야, "동시성을 몇까지 올리면 되는가"에 답할 수 있다. 워커 2개, 동시성 4면 초당 8개까지 돌리려 하니, 한도를 넘는다. 그래서 LLM 큐의 동시성은 호출 수에서 역산한 값으로 정한다. 감으로 올린 동시성은, 급증 날에 큐를 쌓게 만드는 값이다.

비용. 하루 토큰, 작업당 비용. 상한을 정하고 넘기면 알람을 받는다. 돌연변이는 재시도되는 거대 프롬프트, 끝없는 생성 루프. 그것은 큐 사고이기 전에 비용 사고다. 상한이 퓨즈다.

실시간과 배치의 분리. "지금 바로 요약이 필요한" 것과 "밤에 돌리면 되는" 것은 다른 큐에 넣는다. 다른 우선순위, 다른 동시성, 다른 한도 배분. 섞어두면 배치가 할량을 잡아 먹어서 실시간 경로가 느려진다. 이것은 처음에 한 번 정하면 되는 결정이고, 안 정하면 매일 고통을 사는 결정이다.

모델이나 프롬프트 변경은 릴리스다. 실패의 모양이 바뀐다. 새로운 오류, 느려진 응답, 늘어난 재시도. 변경 이후 24시간, DLQ와 재시도율을 유심히 본다. 테스트에서는 괜찮았던 프롬프트가 프로덕션에서는 독소 메시지가 될 수 있다.

4.5 15분 주간 점검

알람은 비상용이다. 나머지는 일정으로 한다. 15분 주간 점검이 1인 팀의 운영 전체다. 순서에는 이유가 있다. DLQ를 가장 먼저 보는 것은, 그것이 이미 "결정이 필요한 일"이기 때문이다. 나머지 네 가지는 상태 확인이고 DLQ만 행동을 요구한다. 행동을 요구하는 것을 먼저 처리해 두면, 나머지 확인이 빨라진다. 그리고 반대도 그렇다. DLQ가 비어 있으면, 이 팀의 조용한 실패는 이번 주에도 없었다는 뜻이다.

첫째, DLQ. 새 항목이 있나. 있다면 각자에게 판단. 리플레이하거나, 닫거나. DLQ가 비어 있으면, 그것이 최고의 결과다.

둘째, 최고령 작업. 오래 기다린 것이 있나. 있다면 이유를 찾는다. 동시성이

부족한가, 외부 한도인가, 멈춘 파티션인가.

셋째, 비용. 이상한 튀기가 있나. 작업 실행 수와 대조해서, "일이 많아진" 건지 "단가가 비싸진" 건지 구분한다.

넷째, 재시도율. 갑작스러운 변화가 있나. 있다면 어떤 작업 유형, 어떤 오류인지 본다.

다섯째, 로그 샘플링. 몇 개의 작업을 골라, 엔큐부터 ack까지 쫓아간다. 지표는 요약이고 요약은 버그를 숨길 수 있다. 샘플링은 끝에서 끝을 검증하는 유일한 수단이다.

점검에는 한 줄의 기록을 남긴다. "날짜, DLQ 0, 최고령 12분, 비용 정상." 끝이다. 한 달 뒤 이 한 줄들이 추세 그래프를 만든다. 최고령 대기가 매주 조금씩 오르고 있는지, 비용이 매주 조금씩 비싸지고 있는지. 한 번의 측정에서는 안 보이고 한 줄 기록에서는 보이는 문제들이 있다.

4.6 비상 스위치

운영의 끝자락에는 "지금은 끄겠다"는 선택지가 준비되어 있어야 한다. 평소에 두어 두고 쓸 일이 없기를 바라는 장치가 비상 스위치다. 1인 팀에 필요한 것은 세 개면 충분하다.

첫째, 큐별 일시 정지. 배치 큐를 멈추는 스위치 하나. 실시간이 막혔을 때, 배치를 꺼서 runway를 내주는 것이 가장 빠른 조치다. DB 플래그 한 칸이면 된다. "이 큐의 작업은 클레임하지 않는다"는 조건을 워커 루프에 넣는 것뿐.

둘째, 워커 정지. 워커 자체를 멈추는 것. 배포 전에, 혹은 "지금 뭔가가 이상하다"는 직감이 들 때, 새 작업을 받지 않게 하는 스위치다. 이미 받은 작업은 끝내게 두고 새 것은 받지 않게 하는 것.

셋째, 기능 끄기. 큐 뒤의 기능을 아예 끄는 것. "요약 생성" 버튼 자체를 잠깐

숨기는 것. 큐가 아무리 잘 만들어져도, 그 뒤의 기능이 잘못된 데이터로 넘쳐나는 상황이라면, 입력을 막는 것이 먼저다.

스위치의 원칙은 하나다. 누르면 되돌릴 수 있어야 한다. 정지한 큐는 나중에 다시 돌리면 되고 꺼둔 기능은 나중에 다시 열리면 된다. 되돌릴 수 없는 스위치는 폭발이다.

4.7 종합 사례: 작은 SaaS 하나

끝으로, 이 책의 모든 것을 하나의 제품에 올려 보자. 이메일, 아바타 이미지, LLM 요약, 결제를 하는 제품이다.

경로(1장): 결제 승인은 동기. 목록 조회는 캐시와 함께 동기. 결제 성공 이후의 "영수증 이메일", "아바타 처리", "LLM 요약"은 큐.

작업 모양(2장): 각 작업은 하나의 동사. 페이로드는 ID. 멱등성 키는 업무 키에서 생성. 영수증과 요액은 order_id를 파티션 키로, 아바타는 user_id를 쓴다.

정책: 이메일은 3회 재시도, 백오프. 아바타도 3회. LLM 요약은 5회, 별도 한도, 밤에는 배치. 상한 초과면 DLQ. 작업별 타임아웃은 이미지 60초, LLM 5분 [추정].

실패(3장): 가시성 타임아웃은 p95 실행 시간의 3배. 오래된 작업 스캔은 1분 간격. 배포는 graceful shutdown. DLQ는 주간 리플레이 절차.

관측(이 장): 지표 네 개. 깨움 알람은 2개(최고령 대기, DLQ 0 아님). 15분 주간 점검. 하루 토큰 비용 상한.

고장이 났을 때, 증상은 각기 다른 행동을 가리킨다. 최고령 대기가 솟으면, LLM API 상태와 동시성을 확인한다. 재시도율이 오르면, 이메일 제공자를 본다. DLQ가 쌓이면, 오류 종류로 분류한다. 비용 알람이 울리면, 돌연변이를 찾는다. 이것이 파이프라인을 "운영"한다는 뜻이다. 소리가

나면 무엇을 해야 하는지를 아는 것이 핵심이다.

4.8 결론: 규율

큐는 빠르게 만드는 기술이 아니다. 실패를 감당할 수 있게 하는 경계다. 한꺼번에 처리하지 않으면, 재시도가 있다. 재시도가 있으면, 매 순간 거기에 있을 필요가 없다. 죽은 것은 데드레터가 받아 주므로, 실패는 조용히 사라지지 않는다.

1인 개발자의 규율은 다섯 가지다. 경계를 정하고 작게 자르고 중복을 무해하게 만들고 해결 불가능한 것은 데드레터에 받고 지표 네 개를 본다. 많지 않다. 배경 작업의 운영이란, 결국 이 다섯 가지를 처음에 세우고 그 뒤로는 지키는 일이다.