멈추지 않고 옮기기

레거시 마이그레이션의 기술

ThakiCloud (Hyojung Han)

멈추지 않고 옮기기

레거시 마이그레이션의 기술

ThakiCloud (Hyojung Han)

Contents

1 경계부터 긋는다: 무엇을 옮기고 무엇을 남길 것인가

마이그레이션이 실패하는 자리는 대부분 새 기술이 아닙니다. 범위입니다. "결제 시스템을 새 스택으로 옮기자"는 문장에는 옮길 대상이 없습니다. 결제 시스템이라는 이름 아래 API 서버, 정산 배치, 관리자 화면, 세 개의 데이터베이스 테이블 그룹, 그리고 아무도 기억하지 못하는 크론 잡 열두 개가 함께 들어 있기 때문이죠. 이 덩어리를 통째로 들면 무중단은 불가능합니다. 들 수 있는 크기로 쪼개는 일이 첫 번째 작업입니다.

1.1 이전 단위는 기능이 아니라 이음새로 정한다

옮길 단위를 고를 때 흔한 실수는 조직도나 기능 명세를 기준으로 자르는 것입니다. "쿠폰 기능을 먼저 옮기자"는 말은 그럴듯하지만, 쿠폰 로직이 주문 테이블을 직접 갱신하고 있다면 그 경계는 존재하지 않습니다. 자를 수 있는 자리는 이미 코드 안에 있는 이음새(seam)뿐입니다.

이음새란 호출자를 고치지 않고 구현을 바꿔 끼울 수 있는 지점을 뜻합니다. 현실에서는 다음 네 곳 중 하나입니다.

- 네트워크 경계: HTTP 엔드포인트, gRPC 서비스, 큐 토픽
- 모듈 경계: 인터페이스 하나로 감싸인 패키지, 리포지터리 계층
- 데이터 경계: 다른 컴포넌트가 조인하지 않는 테이블 묶음
- 배치 경계: 입력과 출력이 파일이나 테이블로 끊긴 잡

이 넷 중 어디에도 걸리지 않는다면, 그 대상은 아직 이전 단위가 아닙니다. 옮기기 전에 이음새를 만드는 작업이 선행되어야 합니다. 인터페이스를 하나 뽑고 기존 구현을 그 뒤로 밀어 넣는 리팩터링은 이전이 아니라

준비이며, 이 준비를 건너뛰면 나중에 롤백이 불가능해집니다. 되돌릴 곳이 없으니까요.

1.2 소유권 지도: 쓰기와 읽기를 따로 그린다

경계를 그으려면 지금 누가 무엇을 만지고 있는지 알아야 합니다. 여기서 반드시 지켜야 할 규칙이 하나 있습니다. 읽기와 쓰기를 같은 표에 뭉뚱그리지 마십시오. 마이그레이션의 난이도는 거의 전적으로 쓰기 쪽에서 결정됩니다.

테이블 하나를 예로 들면 이렇게 정리됩니다.

테이블	쓰는 주체	읽는 주체
orders	주문 API	주문 API, 정산 배치, 관리자
coupons	주문 API, 이벤트 배치	주문 API, 마케팅 대시보드
user_grade	야간 배치	주문 API, 추천 서비스

이 표만 그려도 판단이 나옵니다. `orders`는 쓰는 곳이 하나라서 이전 단위로 적합합니다. `coupons`는 쓰는 주체가 둘이라 이중 쓰기 설계가 두 배로 복잡해지죠. `user_grade`는 쓰기가 배치 한 곳이므로, 읽는 곳이 많더라도 의외로 옮기기 쉬운 편에 속합니다.

지도를 그릴 때는 코드 검색만 믿지 마세요. 실제로 어떤 접속 계정이 어떤 테이블에 쓰기를 하는지는 데이터베이스 쪽 감사 로그나 접속 통계에서 확인하는 편이 정확합니다. 사내 관리 도구, 운영자가 쓰던 임시 스크립트, 데이터 분석팀이 붙여 놓은 리드 리플리카가 코드 저장소 밖에서 살아 있는 경우가 흔합니다.

1.3 옮기지 않을 것을 먼저 확정한다

범위를 줄이는 가장 빠른 방법은 옮길 것을 고르는 게 아니라 옮기지 않을 것을 못 박는 일입니다. 대상 목록을 네 칸으로 나눠 보세요.

- 이전: 새 기반으로 그대로 옮긴다
- 재작성: 옮기면서 구조를 바꾼다
- 동결: 옛 기반에 남기고 더 이상 고치지 않는다
- 폐기: 아무 데도 옮기지 않고 없앤다

경험상 폐기 칸이 예상보다 훨씬 큽니다. 몇 년 동안 아무도 열지 않은 리포트 화면, 후속 상품이 나오며 의미를 잃은 통계 배치, 이미 다른 경로가 생겼는데 남아 있는 중복 API가 그렇습니다. 이런 항목을 하나 폐기할 때마다 이전 작업과 검증 작업이 함께 사라집니다. 옮기는 비용보다 확인하는 비용이 더 크다는 점을 기억하면, 폐기 판정에 시간을 쓰는 편이 훨씬 남는 장사입니다.

폐기 여부를 판정할 때는 "쓰는 사람이 있나요"라고 묻지 마십시오. 그렇게 물으면 모두가 그렇다고 답합니다. 대신 접근 로그를 90일치 뽑아 호출 건수를 보여 주고 "이 숫자가 0인데 없애도 될까요"라고 물어야 합니다. 데이터를 앞에 놓으면 대화가 빨리 끝납니다.

그리고 동결 칸을 부끄러워할 필요가 없습니다. 레거시를 전부 없애야 성공이라는 생각은 계획을 비대하게 만들 뿐입니다. 잘 돌아가고 변경 요구가 없는 배치 하나를 옛 서버에 그대로 두는 선택은, 그 서버를 언제까지 유지할지와 누가 그 결정을 다시 검토할지만 문서에 남긴다면 충분히 합리적입니다.

1.4 되돌릴 수 있는 단위인가: 네 가지 질문

단위 하나를 정했다면 실행 전에 다음 네 질문에 전부 답할 수 있어야 합니다. 하나라도 막히면 단위가 너무 큽니다.

첫째, 이 단위를 새 경로로 보냈다가 5분 안에 옛 경로로 되돌릴 수 있습니까. 되돌리는 방법이 배포라면 답은 아니오입니다. 배포는 몇 분에서 몇십 분이 걸리고, 장애 중에 배포 파이프라인이 정상이라는 보장도 없습니다. 되돌리기는 설정 값 하나를 바꾸는 수준이어야 합니다.

둘째, 되돌린 뒤 데이터가 어긋난 채로 남습니까. 새 경로가 처리한 요청이 옛 저장소에 반영되지 않았다면, 롤백은 데이터 유실이 됩니다. 이 질문에 걸리는 단위는 3장의 이중 쓰기 설계를 먼저 통과해야 합니다.

셋째, 이 단위가 잘못됐다는 것을 사용자보다 먼저 알 수 있습니까. 새 경로에만 걸리는 지표가 없다면 전환 여부를 알아차리는 사람은 고객센터입니다. 전환 전에 신규 경로 전용 성공률과 지연 지표를 먼저 만들어야 합니다.

넷째, 이 단위 하나가 실패해도 나머지 서비스가 살아 있습니까. 옮기려는 컴포넌트가 다른 모든 요청의 앞단에 있다면, 그 단위는 첫 번째 순서로 부적합합니다.

1.5 짧은 사례: 알림 시스템 하나를 자르는 과정

추상적으로만 보면 잘 와닿지 않으니 흔한 예를 하나 따라가 보겠습니다. 사내에 알림 발송 모듈이 있고, 이메일과 문자와 앱 푸시를 모두 처리하며, 주문 서비스와 회원 서비스가 함수 호출로 직접 씁니다. 발송 이력은 주문 데이터베이스의 테이블 하나에 남습니다.

첫 관찰은 이렇습니다. 함수 호출로 붙어 있으니 네트워크 경계가 없고, 이력 테이블이 주문 데이터베이스 안에 있으니 데이터 경계도 없습니다. 이

상태로는 이전 단위가 아닙니다.

그래서 준비 작업이 먼저 들어갑니다. 발송 인터페이스를 하나 뽑아 두 서비스가 그것만 호출하게 바꾸고, 이력 조회는 관리자 화면 한 곳만 쓴다는 걸 확인한 뒤 조회 함수를 인터페이스 뒤로 넣습니다. 여기까지가 이음새 만들기이며, 아직 아무것도 옮기지 않았습니다.

다음은 쪼개기입니다. 이메일과 문자와 푸시를 한 덩어리로 보지 않고 채널별로 나눕니다. 푸시는 실패해도 재발송이 쉽고 법적 보관 요건이 약하니 첫 번째 후보가 됩니다. 문자는 비용이 발생하고 중복 발송이 곧바로 항의로 이어지니 뒤로 미룹니다. 이력 테이블은 마지막입니다.

결과적으로 "알림 시스템 이전"이라는 한 문장은 준비 두 건과 이전 세 건으로 늘어납니다. 일이 늘어난 것처럼 보이지만, 각 항목이 독립적으로 되돌릴 수 있게 되었다는 점이 이득입니다.

1.6 순서는 위험이 낮고 학습이 큰 것부터

경계를 다 그었다면 순서를 정할 차례입니다. 가장 중요한 것부터 옮기고 싶은 유혹이 크지만, 첫 단위의 목적은 성과가 아니라 학습입니다. 배포 파이프라인, 모니터링, 롤백 절차, 데이터 검증 도구를 실제로 한 바퀴 돌려 보는 것이 첫 이전의 실제 산출물입니다.

그래서 첫 대상으로는 트래픽이 있으면서(그래야 문제가 드러납니다) 실패해도 사업이 멈추지 않는 읽기 위주 경로가 적당합니다. 조회 API 하나, 알림 발송 한 종류, 내부용 리포트 하나가 좋은 후보입니다. 여기서 얻은 도구와 절차를 그대로 두 번째, 세 번째 단위에 재사용하면 속도가 붙습니다.

반대로 마지막까지 미뤄야 할 것은 쓰기 주체가 여럿인 핵심 테이블, 트랜잭션 경계가 여러 서비스에 걸친 흐름, 그리고 금액을 확정하는 정산 경로입니다. 이 셋은 앞선 단위들에서 검증 도구가 충분히 다듬어진 뒤에

손대야 합니다.

이 장의 결론은 단순합니다. 계획서 첫 장에 적을 문장은 "무엇을 새 스택으로 옮긴다"가 아니라 "이번 분기에 옮길 단위는 이것 하나이고, 나머지는 남기거나 없앤다"여야 합니다. 다음 장에서는 그 하나를 실제로 옮길 때 호출자를 건드리지 않는 방법, 즉 스트랭글러 배치를 다룹니다.

2 스트랭글러: 라우팅 한 겹으로 점진 전환하기

옛 시스템을 새 시스템으로 바꾸는 방법은 크게 둘입니다. 새것을 다 만든 다음 어느 주말에 갈아 끼우거나, 앞에 얇은 층을 하나 세우고 그 층 뒤에서 조금씩 바꿔 끼우거나. 앞의 방식은 계획이 단순한 대신 실패했을 때 돌아갈 곳이 없습니다. 뒤의 방식이 스트랭글러입니다. 덩굴이 나무를 감싸며 자라다가 결국 나무 자리를 대신 차지하듯, 새 구현이 옛 구현을 조금씩 감싸며 대체합니다.

핵심은 새 시스템을 잘 만드는 데 있지 않습니다. 호출자가 어느 쪽이 응답했는지 몰라도 되게 만드는 데 있습니다.

2.1 파사드부터 세운다

스트랭글러의 첫 작업은 새 코드를 짜는 일이 아니라, 호출자와 옛 구현 사이에 층 하나를 끼워 넣는 일입니다. 이 층은 처음에는 아무 일도 하지 않습니다. 받은 요청을 그대로 옛 구현에 넘기고 응답을 그대로 돌려줄 뿐이죠.

이 단계가 지루해 보여서 건너뛰고 싶겠지만, 여기서 세 가지를 얻습니다. 호출자 목록이 확정되고(파사드를 거치지 않는 호출자가 남아 있다면 그건 곧 사고입니다), 요청과 응답의 실제 모양이 로그로 드러나며, 나중에 라우팅을 넣을 자리가 생깁니다. 명세 문서에 적힌 인터페이스와 실제로 오가는 데이터가 다른 경우는 놀랄 만큼 흔합니다. 파사드는 그 차이를 조용히 알려주는 장치입니다.

파사드를 놓을 위치는 시스템 형태에 따라 다릅니다.

- HTTP 서비스라면 리버스 프록시나 API 게이트웨이의 경로 규칙
- 라이브러리 호출이라면 인터페이스 하나와 위임 구현
- 메시지 기반이라면 토픽을 하나 더 두고 중계하는 소비자
- 데이터베이스 접근이라면 리포지터리 계층 또는 뷰

여기서 하지 말아야 할 일이 있습니다. 파사드를 세우면서 동시에 인터페이스를 개선하지 마십시오. 필드 이름이 마음에 안 들어도, 응답 구조가 어색해도 그대로 통과시켜야 합니다. 파사드 단계의 성공 기준은 "배포했는데 아무 일도 일어나지 않음"입니다. 이 단계에서 동작이 바뀌면, 이후에 문제가 생겼을 때 원인이 파사드인지 새 구현인지 구분할 수 없게 됩니다.

2.2 트래픽 비율이 아니라 기능으로 쪼갠다

파사드가 자리를 잡으면 다음은 분기입니다. 여기서 많은 팀이 트래픽의 몇 퍼센트를 새 경로로 보내는 방식을 먼저 떠올립니다. 카나리 배포에 익숙하니 자연스러운 발상이지만, 마이그레이션에서는 대개 나쁜 첫 선택입니다.

이유는 상태 때문입니다. 같은 사용자의 요청이 어떤 때는 새 경로로, 어떤 때는 옛 경로로 가면 두 저장소에 반쯤씩 흔적이 남습니다. 문제를 재현할 수도 없고, 롤백해도 데이터가 뒤섞인 채 남습니다. 무작위 비율 분기는 완전히 무상태인 읽기 경로에서만 안전합니다.

대신 이렇게 쪼개세요.

- 기능 단위: 조회는 새 경로, 생성과 수정은 옛 경로
- 엔티티 단위: 특정 상품군, 특정 지역, 특정 테넌트만 새 경로
- 사용자 단위: 사번 기반 내부 계정부터, 그다음 베타 그룹
- 시간 단위: 신규 생성 건만 새 경로, 과거 건은 옛 경로

이 축들은 공통점이 있습니다. 하나의 개체가 항상 같은 경로로 간다는

점입니다. 이걸 지키면 문제가 생겼을 때 "어제 오후에 들어온 A 테넌트 주문"처럼 범위를 특정할 수 있고, 되돌릴 때도 그 범위만 되돌리면 됩니다.

가장 자주 쓰이는 조합은 "읽기 먼저, 그다음 신규 쓰기, 마지막에 과거 데이터"입니다. 읽기는 되돌리기 쉽고, 신규 쓰기는 영향 범위가 시점으로 잘리며, 과거 데이터는 백필이 끝난 뒤에 손대면 됩니다.

2.3 섀도 트래픽으로 미리 달군다

새 경로를 실제 사용자에게 노출하기 전에, 응답을 버리는 방식으로 먼저 태워 볼 수 있습니다. 파사드가 요청을 옛 경로로 보내 응답을 돌려주면서, 같은 요청 사본을 새 경로로도 보냅니다. 새 경로의 응답은 사용자에게 가지 않고 비교와 기록에만 쓰입니다.

섀도 트래픽으로 잡히는 문제는 단위 테스트가 절대 못 잡는 종류입니다. 실제 트래픽에만 존재하는 이상한 입력값, 응답 시간의 꼬리 구간, 커넥션 풀 고갈, 캐시가 비었을 때의 동작 같은 것들이죠. 예상 부하가 아니라 실제 부하로 달구는 일이 핵심입니다.

주의할 점 두 가지가 있습니다.

첫째, 부수 효과를 반드시 차단해야 합니다. 섀도로 흘린 요청이 알림을 보내거나 외부 결제를 호출하거나 재고를 차감하면 그것은 사고입니다. 새 경로에 섀도 모드 플래그를 두고, 이 모드에서 외부 호출은 기록만 하고 실제로 나가지 않게 막아야 합니다. 쓰기 계열을 섀도로 돌릴 때는 별도의 검증용 저장소를 쓰는 편이 안전합니다.

둘째, 응답 비교는 자동화해야 합니다. 사람이 로그를 눈으로 보는 방식은 하루도 못 갑니다. 두 응답을 정규화한 뒤 차이를 세는 작은 비교기를 만들고, 필드별 불일치 건수를 지표로 뽑으세요. 이때 타임스탬프나 생성 아이디처럼 원래 다를 수밖에 없는 필드는 비교 대상에서 제외하는 목록을 명시적으로 관리해야 합니다. 이 제외 목록이 슬금슬금 길어진다면 그

자체가 경고입니다.

2.4 라우팅 규칙은 설정이 소유한다

전환 비율과 분기 조건을 코드에 박아 두면, 되돌리는 데 배포가 필요해집니다. 1장에서 정한 “5분 안에 되돌리기” 기준이 여기서 깨집니다.

라우팅 규칙은 런타임에 바꿀 수 있는 곳에 두어야 합니다. 기능 플래그 저장소, 설정 서비스, 최소한 컨피그맵 정도면 됩니다. 그리고 다음 세 가지를 함께 만들어 두세요.

첫째는 전체 되돌리기 스위치입니다. 조건이 아무리 복잡해도 값 하나로 모든 트래픽을 옛 경로로 돌릴 수 있어야 합니다. 이 스위치는 장애 중에 당황한 사람이 조작하게 되므로, 이름은 헷갈리지 않게 짓고 조작 방법은 런북 첫 줄에 적어 둡니다.

둘째는 현재 상태를 보여 주는 화면입니다. “지금 어느 조건이 새 경로로 가고 있는가”를 누구나 3초 안에 확인할 수 있어야 합니다. 이게 없으면 새벽에 깨어난 사람이 설정 저장소를 뒤지게 됩니다.

셋째는 변경 이력입니다. 누가 언제 어떤 값을 바꿨는지 남지 않으면, 지표가 흔들렸을 때 원인 후보에서 라우팅 변경을 배제할 수 없습니다.

2.5 계약 차이는 파사드가 흡수한다

새 구현이 옛 구현과 완전히 같은 응답을 내는 경우는 드뭅니다. 필드 이름이 조금 다르고, 오류 코드 체계가 바뀌었고, 페이지네이션 방식이 커서 기반으로 옮겨 갔을 수 있죠. 이 차이를 호출자에게 떠넘기면 스트랭글러의 전제가 무너집니다. 호출자를 고쳐야 한다면 그건 점진 전환이 아니라 동시 변경입니다.

그래서 차이는 파사드가 흡수합니다. 새 경로의 응답을 옛 형식으로 되돌려

주는 변환 계층을 파사드 안에 두는 것이죠. 코드가 지저분해 보여도 괜찮습니다. 이 변환기는 임시 구조물이고, 전환이 끝난 뒤 호출자를 하나씩 새 계약으로 옮기면서 제거하면 됩니다.

다만 변환기에 로직이 들어가는 순간을 경계해야 합니다. 필드 이름을 바꾸고 형식을 맞추는 정도는 변환이지만, 없는 값을 계산해 채우거나 조건에 따라 다른 값을 만들어 내기 시작하면 그것은 세 번째 구현이 생긴 것입니다. 세 번째 구현은 아무도 테스트하지 않고 아무도 소유하지 않습니다.

변환 규칙은 목록으로 관리하고 각 항목에 제거 조건을 함께 적어 두세요. "호출자 A가 새 형식을 지원하면 삭제"처럼 적혀 있으면 나중에 정리할 때 판단이 빠릅니다.

2.6 진행 상황은 코드 줄 수가 아니라 트래픽으로 센다

스트랭글러의 진척도를 "새 서비스에 구현된 엔드포인트 개수"로 보고하면 착시가 생깁니다. 만들어 놓고 아무도 안 쓰는 엔드포인트는 진척이 아니라 재고입니다.

세어야 할 숫자는 이쪽입니다.

지표	의미
신규 경로 비중	실제 이전된 정도
옛 경로 잔여 호출	남은 작업량
되돌리기 횟수	안정화 수준

주간 보고에는 이 세 숫자만 있으면 충분합니다. 특히 되돌리기 횟수는 숨기지 말고 그대로 보고하세요. 되돌리기가 잦다는 것은 실패가 아니라 안전장치가 작동하고 있다는 뜻이고, 되돌리기가 한 번도 없다는 것은 안전한 게 아니라 아직 충분히 밀어붙이지 않았다는 뜻일 때가 많습니다.

여기까지 오면 요청 경로는 새 시스템으로 옮겨졌습니다. 그러나 데이터는 아직 두 곳에 나뉘어 있습니다. 다음 장은 그 두 곳을 어떻게 같은 상태로 유지하고, 정말 같은지 어떻게 증명하는지를 다룹니다.

3 이중 쓰기와 정합성 검증: 데이터가 어긋나는 지점 찾기

요청 경로를 옮기는 일은 되돌릴 수 있습니다. 설정 값 하나면 되니까요. 데이터는 다릅니다. 새 저장소에만 기록된 주문 한 건은 라우팅을 되돌린다고 옛 저장소로 돌아오지 않습니다. 그래서 무중단 이전의 진짜 난이도는 전부 이 장에 몰려 있습니다.

목표는 명확합니다. 어느 시점에 전환하든, 어느 시점에 되돌리든 두 저장소가 같은 사실을 담고 있게 만드는 것. 그리고 같다는 걸 감이 아니라 숫자로 말할 수 있게 만드는 것입니다.

3.1 이중 쓰기는 백필과 짝을 이룰 때만 성립한다

이중 쓰기는 이름 그대로 한 번의 쓰기 요청을 두 저장소에 반영하는 방식입니다. 그런데 이중 쓰기만 켜 놓으면 새 저장소에는 오늘 이후 데이터만 쌓입니다. 어제까지의 데이터는 없죠. 이 상태로 읽기를 새 저장소로 돌리면 과거 조회가 전부 빈 결과를 냅니다.

그래서 두 작업은 항상 짝입니다.

- 이중 쓰기: 지금 이후 발생하는 변경을 양쪽에 반영
- 백필: 이중 쓰기 시작 시점 이전의 데이터를 새 저장소로 복사

순서가 중요합니다. 이중 쓰기를 먼저 켜고, 그다음 백필을 돌려야 합니다. 반대로 하면 백필이 도는 동안 들어온 변경이 새 저장소에 반영되지 않아 구멍이 생깁니다. 이중 쓰기를 먼저 켜면 백필은 과거 구간만 채우면 되고, 겹치는 구간은 같은 값을 두 번 쓰는 것이라 결과가 같습니다.

이 성질을 멱등성이라고 부르며, 백필 설계의 유일한 필수 조건입니다. 백필

잡은 몇 번을 다시 돌려도 결과가 같아야 합니다. 그러려면 삽입은 무조건 삽입이 아니라 "있으면 갱신, 없으면 삽입"이어야 하고, 갱신 기준은 실행 시각이 아니라 원본의 변경 시각이어야 합니다. 백필이 뒤늦게 도착해 최신 값을 과거 값으로 덮어쓰는 사고는 이 조건 하나로 막을 수 있습니다.

백필은 한 번에 다 돌리지 말고 잘라서 돌리세요. 기본 키 범위나 생성 시각 구간으로 자르고, 어디까지 처리했는지를 별도 테이블에 기록합니다. 그래야 중간에 멈춰도 이어서 재개할 수 있고, 부하가 높은 시간대에는 잠시 세울 수도 있습니다. 커서를 남기지 않는 백필 스크립트는 반드시 한 번은 처음부터 다시 돌게 됩니다.

3.2 부분 실패를 어떻게 흡수할 것인가

이중 쓰기의 본질적 문제는 두 저장소에 걸친 트랜잭션이 없다는 점입니다. 옛 저장소에 쓰기는 성공했는데 새 저장소 쓰기가 실패하는 순간이 반드시 옵니다. 네트워크가 끊기고, 배포 중이고, 커넥션이 모자랍니다.

선택지는 셋입니다.

동기 이중 쓰기는 둘 다 성공해야 응답을 돌려줍니다. 정합성은 가장 좋지만 가용성이 나빠집니다. 새 저장소가 흔들리면 아직 사용자도 없는 시스템 때문에 서비스가 멈춥니다. 전환 초기에 이 방식을 쓰는 것은 위험을 반대로 옮기는 셈입니다.

비동기 이중 쓰기는 옛 저장소에 쓰고 응답한 뒤 새 저장소 쓰기를 큐를 통해 처리합니다. 가용성은 지키지만 지연이 생기고 큐가 밀리면 차이도 벌어집니다. 실패한 메시지를 다시 시도하는 장치와 끝내 실패한 것을 모아 두는 자리가 함께 있어야 합니다.

변경 데이터 캡처는 애플리케이션이 아니라 데이터베이스의 변경 로그를 읽어 새 저장소에 반영하는 방식입니다. 애플리케이션 코드를 거의 건드리지 않아도 되고 누락 가능성이 낮은 대신, 운영할 구성 요소가 하나

늘고 스키마 변경에 민감합니다.

어느 쪽을 고르든 지켜야 할 원칙이 있습니다. 새 저장소 쓰기 실패가 사용자 요청을 실패시키면 안 됩니다. 전환이 끝나기 전까지 진실의 원본은 옛 저장소입니다. 새 저장소는 아직 따라오는 쪽이고, 따라오는 쪽의 사정으로 원본이 흔들려서는 곤란합니다.

그리고 실패를 삼키지도 마십시오. 실패는 조용히 무시하는 게 아니라 세어야 합니다. 실패 건수와 그 식별자를 남겨 두면 다음에 이야기할 리컨실러가 정확히 그 지점을 다시 맞춰 줍니다.

3.3 정합성 검증은 상시로 돌린다

"몇 건 뽑아서 비교해 봤는데 같더라"는 검증이 아닙니다. 어긋남은 보통 특정 조건에서만 발생하고, 무작위 샘플은 그 조건을 잘 피해 갑니다.

필요한 것은 계속 도는 리컨실러입니다. 구조는 단순합니다.

1. 비교 대상 구간을 정한다(예: 최근 30분에 변경된 건)
2. 양쪽에서 같은 구간을 읽는다
3. 키 기준으로 맞춰 세 가지로 분류한다
4. 결과를 지표로 내보내고, 필요하면 자동 교정한다

분류는 이 세 가지면 충분합니다.

분류	뜻
누락	옛쪽에만 존재
잉여	새쪽에만 존재
불일치	양쪽 값이 다름

각 분류는 원인이 다릅니다. 누락은 이중 쓰기 실패나 백필 구멍을 뜻합니다. 잉여는 대개 삭제가 전파되지 않은 경우이며, 소프트 삭제를

쓰는 시스템에서 특히 자주 나옵니다. 불일치는 가장 흥미로운 신호인데, 두 구현이 같은 입력을 다르게 해석한다는 뜻이기 때문입니다. 반올림 규칙, 시간대 처리, 문자열 정규화, 기본값 채우기가 단골 원인입니다.

비교할 때는 값을 그대로 대조하지 말고 정규화한 뒤 대조하세요. 소수점 자릿수를 맞추고, 시각은 표준 시간대로 변환하고, 널과 빈 문자열의 취급을 통일합니다. 이 정규화 규칙 자체가 이전 명세의 일부라는 점을 기억하면 좋습니다. 정규화 규칙이 늘어난다는 것은 두 시스템의 의미 차이가 그만큼 있다는 뜻이니까요.

자동 교정은 신중하게 켜세요. 방향은 항상 원본에서 사본으로 한 방향이어야 합니다. 양방향 교정은 두 시스템이 서로를 덮어쓰며 값이 진동하는 상황을 만들 수 있습니다. 그리고 교정 건수 자체를 지표로 남겨야 합니다. 조용히 고쳐 주는 리컨실러는 문제를 숨기는 장치가 되기 쉽습니다.

3.4 순서와 중복이라는 두 가지 함정

비동기 경로를 쓰는 순간 두 가지 문제가 따라옵니다. 메시지가 순서대로 오지 않을 수 있고, 같은 메시지가 두 번 올 수 있습니다.

순서 문제는 이렇게 나타납니다. 어떤 주문이 생성된 뒤 곧바로 취소되었는데, 취소 메시지가 생성 메시지보다 먼저 도착합니다. 새 저장소에서는 취소가 대상 없음으로 실패하고, 그다음 생성이 처리되어 결국 취소되지 않은 주문이 남습니다. 해결책은 마지막 쓰기 승리 방식을 버전으로 판정하는 것입니다. 모든 레코드에 원본의 변경 시각이나 증가하는 버전 번호를 함께 실어 보내고, 새 저장소는 자기가 가진 값보다 오래된 메시지를 무시합니다.

중복 문제는 재시도가 만들어 냅니다. 새 저장소에 쓰기는 성공했는데 응답을 받기 전에 연결이 끊기면, 발신 쪽은 실패로 판단하고 다시 보냅니다.

같은 메시지를 두 번 처리해도 결과가 같아야 하는 이유가 여기 있습니다. 삽입을 갱신 가능한 형태로 쓰거나 처리한 메시지 식별자를 일정 기간 기록해 두는 방법이 흔합니다.

이 두 가지를 나중에 붙이려면 이미 쌓인 데이터를 다시 손봐야 합니다. 처음부터 넣고 시작하는 편이 훨씬 쌉니다.

3.5 드리프트 지표를 컷오버 게이트로 승격한다

리컨실러가 돌기 시작하면 숫자가 생깁니다. 이 숫자를 대시보드에 두고 가끔 쳐다보는 정도로 끝내지 마시고, 다음 단계로 넘어갈 자격을 판정하는 기준으로 승격시키세요.

예를 들면 이런 식입니다. 최근 7일 동안 불일치율이 만 건당 1건 미만이고, 24시간 넘게 해소되지 않은 누락이 0건이며, 자동 교정 건수가 감소 추세일 때만 읽기 전환을 진행한다. 숫자는 시스템마다 다르지만, 이 기준은 전환 당일이 아니라 미리 정해 두어야 합니다. 당일에 정하면 사람은 통과하는 쪽으로 기준을 맞추게 됩니다.

또 하나 유용한 습관은 어긋남이 발견될 때마다 그 원인을 유형으로 기록해 두는 것입니다. 시간대 처리, 삭제 전파 누락, 큐 지연, 백필 커서 오류처럼 이름을 붙여 두면 몇 주 뒤에 같은 유형이 반복되는지 아니면 매번 새로운 유형인지 보입니다. 반복되면 근본 원인이 남아 있는 것이고, 매번 새롭다면 이제 꼬리 구간에 들어섰다는 신호입니다.

데이터가 두 곳에서 같아졌다는 확신이 생겼다면 이제 전환을 실행할 차례입니다. 다음 장은 전환 그 자체, 그리고 전환과 같은 무게로 준비해야 하는 되돌리기를 다룹니다.

4 되돌릴 수 있는 컷오버: 전환과 롤백을 같은 문장으로 쓴다

컷오버는 마이그레이션에서 가장 짧은 순간이면서 가장 많이 준비해야 하는 순간입니다. 앞의 세 장을 제대로 밟아 왔다면 컷오버 자체는 설정 값 하나를 바꾸는 일에 불과합니다. 그 한 번의 조작이 지루하게 느껴진다면 준비가 잘된 것입니다. 긴장되고 극적으로 느껴진다면 무언가를 건너뛰었다는 뜻입니다.

4.1 계획서에 롤백을 같은 분량으로 적는다

컷오버 계획서를 쓸 때 대부분의 팀은 전진 절차만 상세히 씁니다. 몇 시에 무엇을 하고, 누가 확인하고, 다음 단계로 언제 넘어가는지. 그리고 마지막 줄에 "문제 발생 시 롤백"이라고 한 줄 적습니다.

그 한 줄이 새벽 세 시에 아무 도움이 되지 않습니다. 롤백도 전진과 같은 수준으로 써야 합니다. 어떤 명령을, 어떤 순서로, 누가 실행하는지. 실행 후 무엇을 확인해야 되돌아왔다고 판단할 수 있는지. 되돌린 뒤 새 경로가 처리한 데이터는 어떻게 처리할 것인지.

특히 마지막 항목이 자주 빠집니다. 라우팅만 되돌리면 요청 흐름은 옛 경로로 돌아오지만, 그사이 새 경로가 만든 데이터는 어딘가에 남아 있습니다. 3장의 이중 쓰기를 켜 둔 상태라면 양쪽에 다 있으니 문제가 없지만, 이중 쓰기를 이미 껐다면 그 구간은 새 저장소에만 존재합니다. 그래서 이중 쓰기를 끄는 시점은 컷오버 직후가 아니라 관측 기간이 끝난 뒤여야 합니다.

계획서에 넣을 최소 항목을 정리하면 이렇습니다.

- 전진 절차: 단계별 조작과 각 단계의 확인 항목
- 롤백 절차: 같은 형식으로, 역순으로
- 중단 기준: 어떤 숫자가 어떻게 되면 멈추는가
- 관측 기간: 언제까지 지켜보고 언제 다음 단계로 가는가
- 역할: 조작하는 사람, 지표를 보는 사람, 판단하는 사람
- 연락 체계: 판단을 내릴 사람이 자리에 없을 때의 대안

역할을 세 명으로 나누는 이유가 있습니다. 조작하는 사람이 동시에 지표를 보고 판단까지 하면, 방금 자기가 한 일이 잘됐기를 바라는 마음이 판단을 흐립니다. 한 명이라도 지표만 보는 사람을 두면 중단 결정이 훨씬 빨라집니다. 작은 팀이라 사람이 없다면 최소한 판단 기준을 미리 숫자로 못 박아 두어야 합니다.

4.2 되돌릴 수 없는 지점을 미리 표시한다

모든 단계가 되돌릴 수 있는 것은 아닙니다. 어떤 작업은 실행하는 순간 과거로 돌아갈 방법이 사라집니다.

전형적인 예는 이렇습니다.

- 옛 저장소의 테이블을 삭제하거나 컬럼을 드롭하는 작업
- 외부 시스템에 새 주소를 통보해 옛 주소를 폐기하는 작업
- 아이디 채번 방식을 바꿔 두 체계의 값이 충돌하게 되는 작업
- 되돌릴 수 없는 스키마 변경이 섞인 데이터 변환

이런 작업은 컷오버 계획서에서 별도로 표시하고, 가능한 한 뒤로 미뤄야 합니다. 원칙은 단순합니다. 되돌릴 수 있는 작업을 전부 먼저 하고, 되돌릴 수 없는 작업은 관측 기간을 충분히 지난 뒤에 별도 일정으로 진행합니다.

스키마 변경에서는 확장 후 축소 방식이 잘 통합니다. 컬럼 이름을 바꿔야 한다면 새 컬럼을 추가해 양쪽에 쓰고, 읽기를 새 컬럼으로 옮기고, 한참 뒤에 옛 컬럼을 지웁니다. 세 번의 배포로 나뉘지만 각 단계가 되돌릴 수

있습니다. 한 번에 이름을 바꾸는 편이 깔끔해 보여도, 그 깔끔함의 대가는 되돌릴 수 없음입니다.

외부와 연결된 부분은 특히 조심해야 합니다. 우리 쪽 설정은 1분이면 되돌리지만, 파트너사에 통보한 주소나 발급한 인증 정보는 되돌리는 데 며칠이 걸립니다. 이런 항목은 새것과 옛것을 한동안 병행 지원하는 쪽으로 설계하고, 옛것을 닫는 시점은 5장의 소등 절차에서 다룹니다.

4.3 중단 기준을 숫자로 정한다

"이상하면 되돌린다"는 기준은 기준이 아닙니다. 무엇이 이상한지 사람마다 판단이 다르고, 전환을 준비한 사람일수록 이상 신호를 정상 범위로 해석하는 경향이 있습니다.

중단 기준은 전환 전에 숫자로 적어야 합니다. 예를 들면 이런 형태입니다.

지표	중단 기준
오류율	기준선의 2배 초과
응답 지연	상위 구간 1.5배 초과
정합성 불일치	새 건 발생

여기서 두 가지를 강조하고 싶습니다.

먼저 기준선을 미리 재 두어야 합니다. 전환 후 오류율이 0.3퍼센트라는 숫자만 보면 이게 나쁜 건지 알 수 없습니다. 전환 전 일주일 같은 요일, 같은 시간대의 값을 기록해 두면 비교할 수 있습니다. 요일과 시간대를 맞추는 이유는 트래픽 성격이 다르면 지표도 자연히 달라지기 때문입니다.

그리고 지표는 전체 평균이 아니라 새 경로만 따로 봐야 합니다. 트래픽의 10퍼센트만 새 경로로 보낸 상태에서 전체 오류율은 거의 움직이지 않습니다. 새 경로에서 오류율이 열 배로 뛰어도 전체 지표는 조용합니다.

그래서 2장에서 라우팅을 넣을 때 경로 구분 라벨을 함께 붙여야 한다고 했던 것입니다.

관측 기간도 미리 정하세요. 흔한 실수는 전환 후 10분 동안 문제가 없으면 성공으로 간주하고 다음 단계로 넘어가는 것입니다. 배치가 도는 시간대, 트래픽이 몰리는 시간대, 하루에 한 번 실행되는 정산이 지나가야 비로소 드러나는 문제가 있습니다. 그래서 관측 기간의 하한은 보통 하루이고, 정산이나 마감 같은 주기 작업이 걸려 있다면 그 주기를 최소 한 번은 통과해야 합니다.

4.4 당일 타임라인은 촘촘하지 않아야 한다

컷오버 당일 일정을 분 단위로 빽빽하게 짜 놓으면, 한 단계가 밀리는 순간 나머지가 전부 압박을 받습니다. 그러면 확인해야 할 것을 건너뛰게 되고, 정확히 그 지점에서 사고가 납니다.

좋은 타임라인은 오히려 헐렁합니다. 단계와 단계 사이에 지켜보는 시간이 들어가 있고, 그 시간에 무엇을 볼지가 적혀 있습니다. 대략 이런 모양이 됩니다.

- 시작 전: 기준선 지표 기록, 담당자 확인, 되돌리기 명령 준비
- 1단계: 내부 계정만 새 경로로, 30분 관측
- 2단계: 특정 테넌트 하나 추가, 2시간 관측
- 3단계: 전체 전환, 하루 관측
- 다음 날: 정산 배치 결과 대조 후 이중 쓰기 유지 여부 결정

각 관측 구간이 끝날 때는 “이상 없으니 다음”이 아니라 미리 정한 숫자를 실제로 읽고 기록해야 합니다. 기록을 남기면 나중에 문제가 드러났을 때 언제부터 시작됐는지 역추적할 수 있습니다.

그리고 마지막 단계 뒤에 아무 일도 하지 않는 시간을 일부러 남겨 두세요. 전환한 사람이 그날 다른 작업을 잡아 두지 않는 것만으로도 대응 속도가

달라집니다.

4.5 리허설 없이 실행하지 않는다

컷오버 계획서를 다 썼다면 읽어 보는 것으로 끝내지 말고 실제로 돌려 봐야 합니다. 리허설의 목적은 절차가 맞는지 확인하는 게 아니라, 절차에 적혀 있지 않은 것을 발견하는 데 있습니다.

리허설에서 자주 드러나는 것들이 있습니다. 설정 변경 권한이 특정 한 사람에게만 있어서 그 사람이 없으면 아무것도 못 한다는 사실, 롤백 명령에 오타가 있다는 사실, 대시보드에 필요한 지표가 아직 없다는 사실, 그리고 전환 단계 사이의 대기 시간이 실제로는 문서에 적힌 것보다 훨씬 길다는 사실 같은 것이죠.

리허설은 두 종류로 나눠 하면 좋습니다. 하나는 스테이징에서 전체 절차를 실제 명령까지 실행하는 것이고, 다른 하나는 운영 환경에서 롤백 경로만 검증하는 것입니다. 후자는 실제로 소량의 트래픽을 새 경로로 보냈다가 즉시 되돌리는 방식으로 할 수 있습니다. 되돌리기가 작동한다는 것을 실제로 확인한 팀과 문서상으로만 확인한 팀은 전환 당일의 자신감이 다릅니다.

마지막으로 시점 선택 이야기를 한마디 덧붙이면, 무중단 전환의 장점은 심야에 할 필요가 없다는 것입니다. 사람이 가장 또렷한 시간, 필요한 사람들이 모두 자리에 있는 시간, 그리고 문제가 생겼을 때 여유 있게 대응할 수 있도록 하루가 아직 남아 있는 시간을 고르세요. 평일 오전이 대체로 가장 좋은 선택입니다. 새벽 전환을 고집한다면 그것은 아직 되돌리기를 신뢰하지 못한다는 뜻이고, 그렇다면 전환보다 되돌리기를 먼저 손봐야 합니다.

전환이 안정되고 관측 기간을 통과했다면 남은 일은 하나입니다. 옛 시스템을 실제로 끄는 것.

5 끄기: 옛 시스템을 실제로 내리는 순서

마이그레이션 프로젝트의 90퍼센트는 여기서 멈춥니다. 트래픽은 새 시스템으로 넘어갔고, 지표도 좋고, 팀은 다음 일로 옮겨 갑니다. 옛 시스템은 "혹시 모르니까" 켜 둔 채로 남습니다. 그리고 3년 뒤, 아무도 그게 뭘 하는지 모르는 서버가 여전히 청구서에 찍혀 있습니다.

꺼지지 않은 레거시는 이전이 끝난 상태가 아닙니다. 두 시스템을 동시에 운영하는 상태입니다. 비용도 두 배, 보안 패치 대상도 두 배, 온콜 담당자가 알아야 할 것도 두 배입니다. 게다가 옛 시스템이 살아 있는 한 누군가는 그쪽에 새 기능을 붙이고, 그러면 이전은 사실상 원점으로 돌아갑니다.

이 장은 그 마지막 한 걸음을 위한 절차입니다.

5.1 숨은 소비자를 찾아내는 트래픽 고고학

끄기 전에 답해야 할 질문은 하나입니다. 아직 이걸 쓰는 사람이 있는가.

코드 검색으로는 답이 나오지 않습니다. 저장소 밖에서 호출하는 것들이 있기 때문입니다. 운영자의 로컬 스크립트, 사내 위키에 적힌 컬 명령, 데이터 분석팀의 예약 쿼리, 파트너사 시스템, 그리고 누군가 몇 년 전에 설정해 둔 모니터링 프로브 같은 것들이죠.

실제 호출을 봐야 합니다. 확인할 곳은 대개 이렇습니다.

- 액세스 로그: 최근 90일간의 호출자 아이피와 사용자 에이전트
- 데이터베이스 접속 통계: 어떤 계정이 언제 붙었는가
- 방화벽과 보안 그룹 로그: 어디에서 들어오는 연결이 있는가
- 인증 서버 로그: 어떤 토큰과 키가 아직 발급되어 쓰이는가
- 배치 스케줄러: 아직 등록된 잡이 남아 있는가

90일을 보는 이유는 월 단위와 분기 단위 작업 때문입니다. 30일만 보고 껐다가 분기 마감 배치가 죽는 사고는 드물지 않습니다. 회계나 정산이 걸려 있다면 400일 정도를 보는 편이 안전합니다. 연간 작업이 한 번은 지나가야 하니까요.

호출자를 찾았다면 목록을 만들고 각각에 담당자를 붙입니다. "누군지 모르는 아이피"는 그대로 두면 영원히 모르는 채로 남습니다. 네트워크 대역으로 조직을 추정하고, 사내 공지를 올리고, 그래도 안 나오면 다음 단계인 차단으로 밝혀냅니다.

5.2 소등은 차단, 대기, 삭제 세 단계로 나눈다

한 번에 끄지 않습니다. 되돌릴 수 있는 단계부터 밟습니다.

1단계는 차단입니다. 서비스는 살아 있지만 요청을 거절합니다. 완전히 막기 전에 예고 단계를 두면 더 좋습니다. 응답에 폐기 예정 헤더를 붙이거나, 응답을 의도적으로 몇 초 늦추거나, 하루 중 특정 시간대에만 막아 보는 방식이 있습니다. 이 기법은 조용히 쓰던 소비자를 스스로 나타나게 만드는 데 효과적입니다. 아무도 항의하지 않는다면 정말 아무도 안 쓰는 것이고, 항의가 들어오면 그 사람이 곧 담당자입니다.

차단 단계의 핵심은 즉시 되돌릴 수 있다는 점입니다. 설정 하나로 다시 열 수 있어야 하고, 되돌리는 방법을 차단 공지에 함께 적어 두어야 합니다.

2단계는 대기입니다. 차단한 상태로 충분한 기간을 둡니다. 서버는 켜져 있고 데이터도 그대로지만 아무도 쓰지 않는 상태입니다. 기간은 앞서 로그를 본 주기와 같아야 합니다. 월간 작업이 있으면 최소 한 달, 분기 작업이 있으면 한 분기입니다.

이 기간에 데이터를 백업해 둡니다. 옛 저장소의 스냅샷을 만들어 별도 보관소에 넣고, 복원 절차를 문서로 남기고, 실제로 한 번 복원해 봅니다. 복원해 본 적 없는 백업은 백업이 아니라 위안입니다. 보관 기간은 법적

요건과 내부 정책에 맞춰 정하되, 백업이 있다는 사실 자체가 삭제 단계의 심리적 장벽을 크게 낮춰 줍니다.

3단계는 삭제입니다. 인스턴스를 내리고, 데이터베이스를 지우고, 디엔에스 레코드를 없애고, 인증 정보를 폐기하고, 파이프라인과 대시보드와 알림 규칙을 제거합니다. 이 단계는 되돌릴 수 없으므로 체크리스트로 진행하고, 각 항목에 실행자와 실행 시각을 남깁니다.

삭제할 때 자주 잊는 것들이 있습니다. 모니터링 알림 규칙(꺼진 서버 때문에 새벽에 울립니다), 시크릿 저장소에 남은 자격 증명, 방화벽 규칙, 로드밸런서의 빈 대상 그룹, 그리고 문서에 남아 있는 옛 주소입니다. 마지막 항목이 특히 성가신데, 문서를 안 고치면 반년 뒤 신규 입사자가 존재하지 않는 서버로 연결을 시도하게 됩니다.

5.3 끄지 못하게 만드는 것은 대개 기술이 아니다

소등이 미뤄지는 이유를 파고들면 기술적 장애물은 의외로 적습니다. 대부분은 사람과 일정의 문제입니다.

가장 흔한 형태는 책임의 공백입니다. 이전을 수행한 팀은 새 시스템의 성과로 평가받고, 옛 시스템을 끄는 일은 아무의 성과도 아닙니다. 그래서 계획서 마지막 줄에 “레거시 폐기”라고 적혀 있어도 아무도 그 줄을 집어 가지 않습니다. 해법은 단순합니다. 소등을 프로젝트의 마지막 단계가 아니라 완료 조건으로 정의하고, 담당자와 기한을 이전 착수 시점에 함께 못 박는 것입니다.

다음으로 흔한 것은 막연한 불안입니다. “혹시 문제가 생기면 돌아갈 곳이 필요하지 않을까”라는 말은 합리적으로 들리지만, 기한이 없으면 영원히 유효합니다. 이 불안에는 날짜로 답해야 합니다. 전환 후 몇 주가 지나고 어떤 조건을 만족하면 되돌아갈 필요가 없다고 판단할지를 미리 적어 두면, 나중에 감정이 아니라 문서를 근거로 결정할 수 있습니다.

마지막은 남은 소비자가 옮길 시간을 못 내는 경우입니다. 이때는 차단 예고와 기한을 함께 통보하고, 그 기한을 지키는 편이 낫습니다. 기한을 한 번 연장하면 두 번째 연장은 훨씬 쉬워집니다.

5.4 폐기 완료 기준

무엇을 만족하면 끝났다고 말할 수 있는지 미리 정해 두면 프로젝트가 흐지부지되지 않습니다. 다음 정도면 충분합니다.

항목	완료 조건
트래픽	옛 경로 호출 0건
인프라	인스턴스와 저장소 삭제
비용	청구서에서 항목 사라짐
문서	옛 주소 참조 0건

비용 항목을 넣은 이유가 있습니다. 이것만이 유일하게 거짓말을 하지 않는 지표이기 때문입니다. 다 껐다고 생각했는데 다음 달 청구서에 여전히 찍혀 있다면, 어딘가에 남아 있는 것입니다. 다음 달 명세서를 확인하는 일까지를 프로젝트 범위에 포함시키세요.

5.5 회고에 남길 것

마지막으로 남는 작업은 배운 것을 다음 이전에 쓸 수 있는 형태로 적어 두는 일입니다. 대부분의 조직에서 마이그레이션은 한 번으로 끝나지 않습니다. 이번에 만든 파사드 패턴, 리컨실러, 컷오버 계획서 양식, 소등 체크리스트는 다음 대상에 거의 그대로 재사용됩니다.

회고에 꼭 남겼으면 하는 항목은 세 가지입니다.

먼저 어긋남의 유형 목록을 남겨 둡니다. 3장에서 기록해 둔 정합성 문제의 원인들을 정리해 두면, 다음 이전에서는 같은 항목을 미리 점검하고 시작할 수 있습니다. 시간대 처리와 반올림 규칙은 거의 매번 등장하는 단골입니다.

여기에 실제로 걸린 시간과 처음 예상한 시간의 차이도 적어 둘 만합니다. 어느 단계가 예상보다 길었는지를 남기면 다음 계획의 정확도가 올라가는데, 경험상 백필과 소등이 가장 크게 빗나가는 구간입니다. 앞의 것은 데이터 양 때문에, 뒤의 것은 사람 때문에 그렇습니다.

끝으로 되돌리기를 실제로 쓴 횟수와 그때의 상황을 기록해 둡니다. 되돌릴 수 있게 만든 설계가 값을 했는지를 보여 주는 증거이고, 다음 프로젝트에서 준비 단계에 시간을 쓰자고 설득할 때 쓰이는 근거가 됩니다.

여기까지가 전부입니다. 다섯 장을 한 문장으로 줄이면 이렇게 됩니다. 옮길 것을 작게 자르고, 앞에 층을 하나 세워 조금씩 바꿔 끼우고, 데이터가 같다는 것을 상시로 증명하고, 되돌릴 수 있는 방식으로 전환하고, 마지막에 옛것을 실제로 끕니다. 새로운 기술은 하나도 필요하지 않습니다. 필요한 것은 매 단계마다 돌아갈 자리를 남겨 두는 습관뿐입니다.