홍지인 · 백엔드 포트폴리오

문제를 정의하고, 선택하고, 검증한 기록

홍지인 Junior Backend Engineer

조회 지연과 장애 상황을 재현하고, 설계의 선택 이유와 검증 결과를 설명합니다. 코드 리뷰와 기술 토론을 통해 서로 다른 해결 방법을 배우는 것을 좋아합니다.

연락처 · 코드 저장소

이메일 andy230@naver.com

전화번호 010-3046-2785

GitHub https://github.com/JIN-076

GitLab https://gitlab.com/jiinhong989

01 **Error Archive** 2인 프로젝트 · 백엔드 전담 pp. 03–12 ↗

배포 서비스 열기 ↗

서비스 경계 → 공개 문서·권한 정책 → 부하·이벤트 복구 → 배포 안정성

담당 범위 5개 서비스·Gateway 설계 및 구현, OCI k3s 배포 환경 구성, 성능·장애 시나리오 검증.

02 **E-Commerce** 개인 프로젝트 · 백엔드 pp. 13–24 ↗

결제·주문 경계 → 재고 예약 → 응답·완료 SLA → 환불·쿠폰

담당 범위 주문·결제와 이벤트 처리 설계 및 구현, 쿠폰 동시성 제어, 부하 테스트를 통한 병목 분석·개선.

03 **JOBPREP** 백엔드 팀 리드 · 총 8명 pp. 25–30 ↗

대량 조회 → 채팅 보존·읽음 상태 → 운영·협업

담당 범위 요구사항·DB 설계 조율, 주요 API·실시간 채팅 구현, 코드 리뷰.

읽는 순서

각 사례는 문제·제약, 선택 이유, 구현 범위, 검증 결과, 남은 한계 순서로 정리했습니다. 측정 조건과 코드·실험 기록은 사례 하단에서 확인할 수 있습니다.

홍지인 · Backend Skill Map

기술별로 다른 경험의 깊이를, 직접 한 작업과 함께

BACKEND ENGINEER · SKILL MAP

정합성·성능·장애를 근거로 설명하는 백엔드 개발자

4단계 기준
기술마다 독립적으로 선택합니다.

읽는 법 각 기술의 직접 적용·검증·설계 경험을 4단계로 표시했습니다. 수준은 아래 경험 범위를 기준으로 한 자기평가입니다.

4단계 기준 아래 기준을 만족하는 경험까지만 선택

1단계	2단계	3단계	4단계
개념을 설명하고 작은 학습·실습으로 확인	프로젝트 기능에 적용하고 테스트·로그로 동작을 확인	병목·실패를 재현하고 전후 지표로 검증	경계·대안·제약·복구 경로를 비교해 설계

기술	수준 · 4단계	직접 적용한 범위 · 프로젝트 근거
BACKEND 언어 · 프레임워크 · 인증 · 실시간 통신		
Java	3단계	EA — 스레드 대기 위치 진단·실행기 제한·독립 호출 병렬화
Kotlin	2단계	EC — Spring Boot 기반 주문·결제 백엔드 기능 구현에 활용
Spring Boot	4단계	EA — 트랜잭션·정책 결합도로 5개 서비스 경계 설계, 장애 시 응답 정책 구현
JPA	3단계	EA·EC — 트랜잭션 경계·조회 구조 조정, DB 집계 이관 효과 검증
QueryDSL	2단계	EC·JP — 조회 API 구현과 조회 쿼리 재구성에 활용
Spring Security	2단계	EA — 인증·권한 검증 경계와 인증·인가 실패 응답 처리
JWT	3단계	EA — 사용자·대상·권한별 내부 JWT 캐싱으로 반복 RS256 서명 감소
OAuth	2단계	EA — GitHub OAuth 로그인 연동과 콜백·토큰 처리 구현
WebSocket	3단계	JP — 전송 실패·접속 캐시 유실 시 메시지 보존과 읽음 상태 회복 검증
STOMP	2단계	JP — 실시간 채팅의 메시지 발행·구독 및 전송 경로 구현
DATA & MESSAGING DB · 캐시 · 이벤트 처리		
SQL	3단계	EC·JP — 실행 계획 분석으로 조회 구조·정렬·집계 비용을 확인하고 쿼리 재구성
PostgreSQL	3단계	EA — 인덱스·표본 조회·DB 집계로 조회 비용 개선, 멱등 키·트랜잭션 검증
MySQL	3단계	EC — 결제 승인 조회의 실행 계획 분석·복합 인덱스 적용 후 응답 지연 검증
Redis	3단계	JP — presence 캐시 삭제 후 SEND 시 재캐싱·읽음 상태 회복 검증
Kafka	4단계	EA — Outbox·멱등 처리·수동 커밋·DLQ 복구 경계 설계, 장애·재전달 검증
INFRA & OBSERVABILITY 배포 · 부하 생성 · 관측		
Kubernetes	2단계	EA — OCI 2노드 k3s에 Gateway·5개 서비스 배치, RollingUpdate 적용
GitHub Actions	2단계	EA — 서비스별 테스트 게이트 구성, 테스트 실패 시 배포 차단
k6	3단계	EA·EC — RPS·VU별 부하를 주입하고 p95·실패율·처리량을 전후 비교
Elastic APM	3단계	EC — 결제 승인 트레이스에서 조회·Outbox 상태 갱신 병목 식별
Prometheus	2단계	JP — CPU·힙·커넥션 풀·HTTP 지표를 수집하고 임계치 관측
Grafana	2단계	JP — 리소스·오류율·응답시간 대시보드와 알림 구성

근거 EA: Error Archive · EC: E-Commerce · JP: JOBPREP. 각 프로젝트 상세 페이지와 저장소에서 구현·측정 근거를 확인할 수 있습니다.

서비스가 해결하는 문제와 맡은 범위

함께 지킬 규칙을 기준으로 서비스 경계 결정

기록·타임라인·솔루션을 Core로 묶고, 인증·발행·알림·인사이트를 분리

진행 기간 2026.06 ~ 09 **구성·역할** 총 2명 · 백엔드 전담

누구의 어떤 문제

오류를 기록하고 시도·해결 과정을 타임라인으로 남겨, 개인과 팀이 해결 경험을 다시 활용하도록 돕는 서비스입니다.

내 책임

5개 서비스와 Gateway의 백엔드 설계·구현, OCI 2노드 k3s 구성, 배포 및 성능·장애 시나리오 검증을 담당했습니다.



그림 01. 해결책만 남고 '왜 그렇게 풀렸는지'는 휘발되는 문제를 해결한다 — 상세 조회 화면은 오류 기록 → 해결 시도 타임라인 → 채택된 솔루션으로 이어지며, **실패한 시도까지 유형·'배운 점'과 함께 기록해 결과가 아닌 맥락과 근거를, 나아가 실패마저 재사용 가능한 자산으로 남긴다.**

다음 · 일관성·변경 기준에 따라 서비스를 나눈 이유

서비스 경계와 데이터 책임

일관성·변경 기준에 따라 서비스를 나눈 이유

서비스 경계 · 논리 구성과 설계의 선택

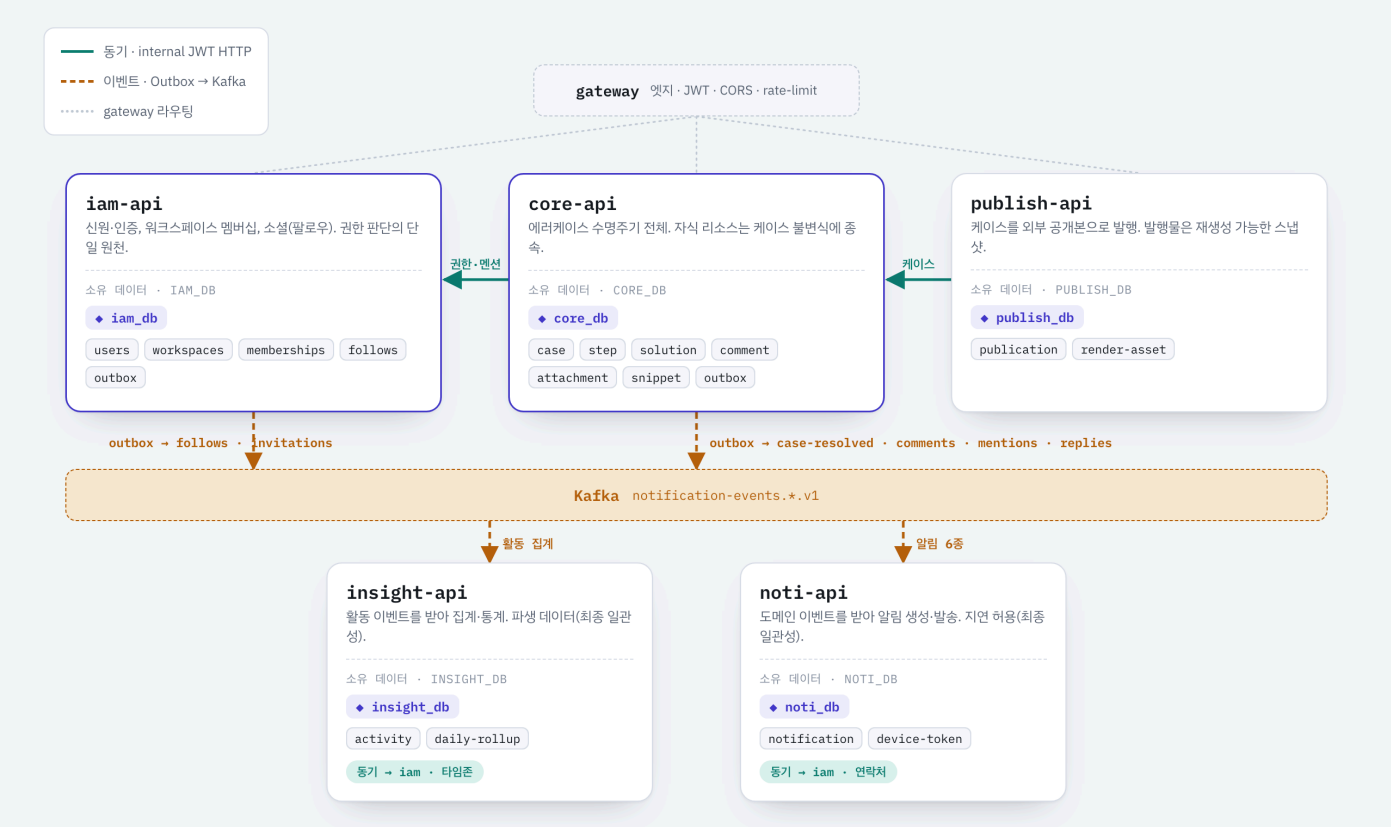


그림 02. 트랜잭션·정책 결합도를 기준으로 core-api에 기록·타임라인·솔루션을 통합하고, iam-api·publish-api·insight-api·noti-api를 서비스별 논리 DB로 분리해 Gateway 라우팅·Internal JWT HTTP·Outbox→Kafka 이벤트로 연결한 MSA 논리 구성도

경계	설계 이유 / 감수한 비용
Core에 기록·타임라인·솔루션 통합	기록·타임라인·솔루션은 "자식은 케이스가 존재·접근 가능할 때만 유효하다"는 같은 트랜잭션 불변식을 공유하므로 Core 한 경계에 통합
인증·발행·알림·인사이트 분리	① **IAM**: "누구"와 "권한"의 단일 원천이자 별개의 일관성/보안 도메인. 모두가 동기 내부 호출로 소비하되, 인증 변경이 케이스 로직으로 번지지 않도록 독립시킨다. ② **PUBLISH**: 발행물은 케이스의 스냅샷 — 재생성 가능하고 원본보다 늦어도 된다. ③ **NOTI · INSIGHT**: 알림은 지연·재시도돼도, 통계는 파생·재계산 가능하므로 즉시 일관성이 불필요하다.

다음 · 논리 경계를 OCI k3s 배치로 연결

실제 배포와 운영 제약

논리 경계를 OCI k3s 배치로 연결

배포 구성 · 소규모 인프라에서 감수한 비용

DEPLOYMENT VIEW

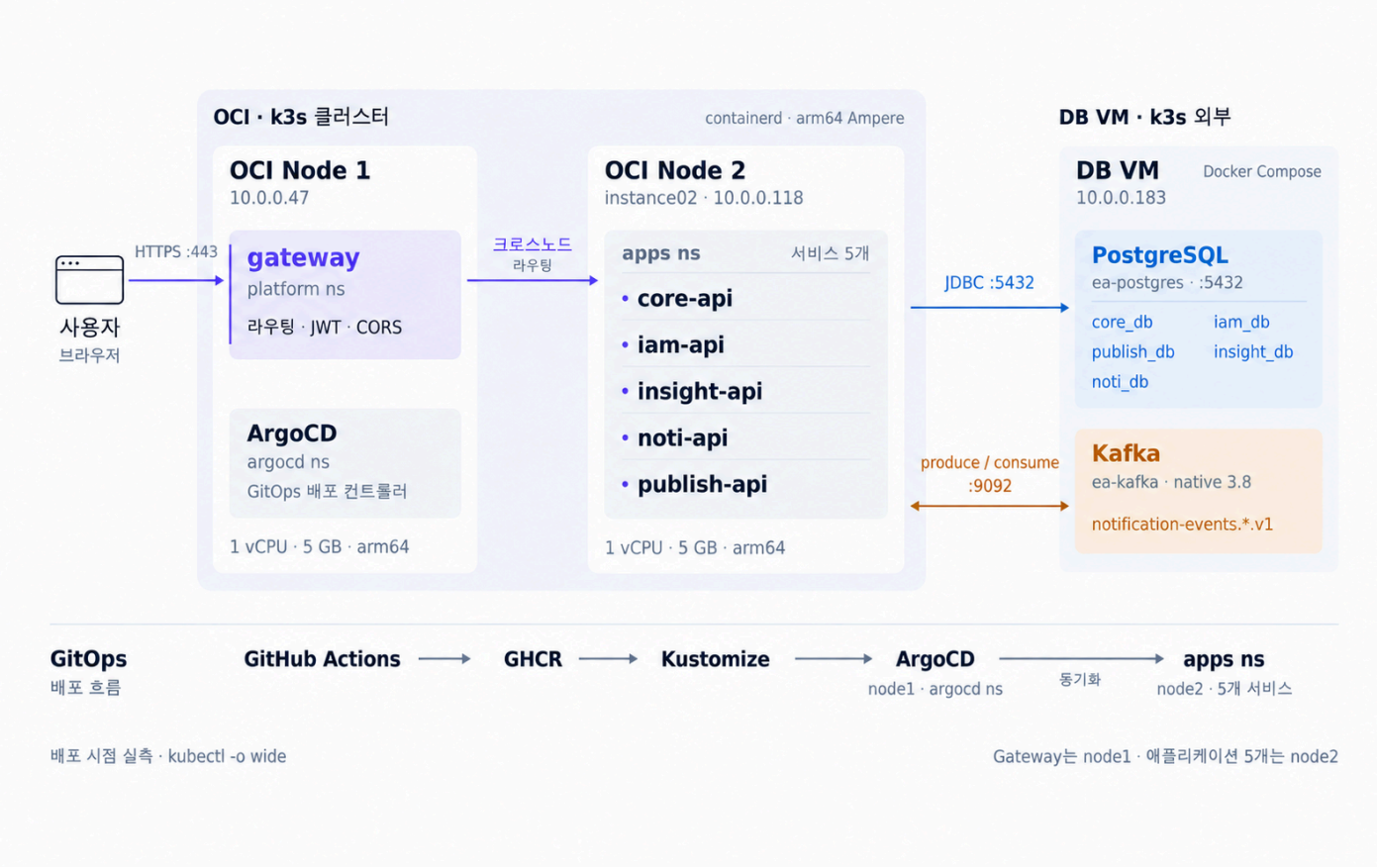


그림 03. 실측 배포 기준으로 OCI 2노드 k3s의 Node 1(platform ns)에 Gateway·ArgoCD, Node 2(apps ns)에 5개 서비스 파드를 배치하고, 별도 DB VM의 Docker Compose PostgreSQL·Kafka와 JDBC :5432·이벤트 :9092 경로 및 GitHub Actions→GHCR→Kustomize→ArgoCD GitOps 흐름을 나타낸 배포도

경계	설계 이유 / 감수한 비용
소규모 인프라	소규모(저비용 OCI·사실상 1인 운영) 단계에서는 다중 브로커·DB 레플리카·멀티 AZ가 주는 가용성보다 그 비용과 운영 부담이 훨씬 커서, HA를 비용·단순성과 의식적으로 맞바꿨다. 대신 비즈니스 사실의 원천을 각 서비스 DB의 Outbox에 남겨 두어, 단일 브로커가 유실돼도 도메인 데이터는 잃지 않고 이벤트를 재발행할 수 있게 함으로써 블라스트 반경을 "영구 손실"이 아닌 "일시 지연"으로 묶어 두었다. 또한 DB-per-service·이벤트 같은 논리 경계를 이미 그어 둔 덕분에, 규모가 커졌을 때 물리 격리로 올리는 일은 재설계가 아니라 브로커 복제·DB 레플리카·노드 분산을 더하는 구성 변경으로 끝난다.

근거 https://github.com/studieo-javry/error-archive

원본 기록과 공개 문서의 생명주기를 분리

편집 중인 기록과 공유한 문서를 스냅샷으로 분리하고, 재발행·공개 중단·복원의 규칙을 설계

문제 · 편집과 공유는 변경 기준이 다르다

원본 오류 기록은 사용자가 계속 수정·삭제하지만, 외부에 공유한 문서는 발행 당시의 내용을 그대로 유지해야 합니다. 원본의 변경·삭제가 이미 공유된 문서에 어떻게 반영돼야 하는지 별도 정책이 필요했습니다.

선택 · 내용과 공개 상태를 분리

발행 시점의 내용을 ContentSnapshot으로 박제해 원본과 분리하고, 발행물의 원본 상태(SOURCE_DELETED)와 공개 상태(UNPUBLISHED)를 직교하는 두 축으로 나눴습니다. 덕분에 원본 삭제와 공개 철회가 서로 다른 사건이 되고, 내용 재발행과 메타데이터 편집도 변경 범위에 따라 구분됩니다.



그림 04. 원본 편집은 공개 스냅샷에 자동 전파되지 않고, 원본 삭제는 SOURCE_DELETED로 표시만 할 뿐 발행물을 지우지 않는다 — 공개 중단(UNPUBLISHED)은 이와 별개의 축이다.

설계 대상	적용한 규칙	지키려는 동작
공개 내용	ContentSnapshot 불변 저장 · 내용 변경 시에만 version++	원본 편집과 공개 문서의 변경 시점을 분리
생명주기	원본 상태(SOURCE_DELETED) / 공개 상태(UNPUBLISHED) 별도 축	원본 삭제 ≠ 공개 철회 · 공개 중단 시 410 · 복원 시 재접근 허용
재요청	멱등키·요청 내용 비교 · 케이스당 발행물 제한 — 멱등키(key,user)·케이스당 1 canonical(original_case_id) UNIQUE	같은 요청은 결과 재생(200) · 다른 내용/동시 레이스는 충돌(409)

검증 범위 · 환경과 관찰 결과

Spring Boot · MockMvc · H2 통합 테스트(기존 9 + 추가 3, publish-api 전체 52개 0 실패). 추가 검증 — ① 동일 멱등키 8-동시발행 → 발행물 1·멱등 1(중복 0 / 분포 201:1, 409:7) ② core 다운 중 공개 HTML 200·core 미호출 ③ 렌더 캐시 워밍 후 메타 변경·unpublish(410)·restore(200) 즉시 반영. H2·단일 JVM이라 실 PostgreSQL·멀티 인스턴스 동시성으로 확대 해석하지 않는다.

선택의 비용 · 원본 삭제와 공개 철회는 별개

완결된 스냅샷으로 유지하고 SOURCE_DELETED로 표시만 한다(재발행만 409) — 공유된 링크를 깨지 않기 위한 선택이다. 다만 민감한 원본이 삭제돼도 발행물을 자동으로 내리지는 않아 작성자가 직접 unpublish해야 하며(원본→발행 실시간 연동은 event-driven 과제로 남김), 캐시 무효화는 공개 중단·메타 변경까지 검증했으나 첨부파일 presign 경로의 캐시 상호작용은 아직 미검증 경계다.

근거 https://github.com/studieo-javry/error-archive · CasePublishment / PublishmentUseCases / PublishmentIntegrationTest · PublishLifecycleVerificationTest (branch verify/publish-lifecycle-separation)

서비스 간 신뢰와 워크스페이스 권한을 분리

내부 토큰으로 신원을 전달하고, 소유자·팀 역할·공개 범위로 데이터 접근 권한을 판단

문제 · 인증된 사용자도 모든 데이터의 권한은 없다

서비스가 나뉜 환경에서 Gateway가 확인한 사용자 신원을 내부 서비스로 전달하면서도, 개인·팀 기록의 접근 범위는 각 서비스가 따로 지켜야 했습니다. 특히 팀 기록을 작성한 사람과 그것을 외부에 공개할 권한이 있는 역할은 구분돼야 했습니다.

선택 · 내부 인증 이후 도메인 정책을 적용

Gateway가 내부 JWT를 발급하고 각 서비스가 서명·발급자·대상 서비스(aud)·만료를 검증해 신원을 신뢰합니다. 그 위에서 Core는 소유자·공개 범위·워크스페이스 역할로 읽기·수정·공개 전환을 다시 판단합니다 — 인증(신원)과 인가(권한)를 분리했습니다.



그림 05. 내부 JWT가 유효하다고 접근을 허용하지 않고, PRIVATE는 소유자·WORKSPACE는 팀 역할·공개 전환은 ADMIN으로 데이터 권한을 별도 판단한다.

검증할 경계	정책 · 기대 동작	실제 확인 결과
내부 토큰	위조·만료·대상 서비스 불일치는 인증 거부	aud 불일치·만료·서명 위조·미지의 발급자 → 전부 401 (4/4)
개인·팀 기록	PRIVATE는 소유자 · WORKSPACE는 팀 역할 확인	비소유자 PRIVATE 403 · 멤버 READ+ 읽기 200 · 비멤버 403
팀 기록의 공개 전환	작성자라도 해당 워크스페이스 ADMIN 필요	owner·ws READ/WRITE → 403 · ADMIN → 200(PUBLIC)
IAM 장애	권한 확인이 필요한 경로는 접근 차단 유지	역할 조회 장애 시 503(fail-closed) · 비인가 허용 0건

검증 조건 · 권한 행렬의 범위

Spring Boot·MockMvc·H2 통합 테스트. 워크스페이스 역할 조회(iam-api) 포트만 목으로 역할/장애 주입. 사용자 2명·워크스페이스 1개·역할 READ/WRITE/ADMIN/비멤버 조합 + 내부 JWT 경계(aud/만료/서명/발급자). core-api 전체 51/0, 내부 토큰 경계 4/4 통과. H2·단일 JVM이라 실 PostgreSQL·멀티 인스턴스로 확대 해석하지 않음

남은 경계 · 권한 변경과 의존 서비스 장애

워크스페이스 역할은 요청마다 iam-api에서 실시간 조회하므로 탈퇴·강등은 다음 요청에 즉시 반영된다(멤버십 회수 시 200→403 확인). 내부 JWT의 ttl(60s)은 신원·전역 역할 전파 주기일 뿐 워크스페이스 역할 판단과는 분리된다. 실제 iam-api 왕복·서킷브레이커 상태 전이·멀티 인스턴스 동시성은 범위 밖(포트 목으로 역할/장애만 주입)

근거 https://github.com/studieo-javry/error-archive · shared-internal-auth / ErrorCaseAccess / WorkspaceAccess / WorkspaceQueryPort · 테스트 WorkspaceAuthorizationVerificationTest · InternalTokenBoundaryVerificationTest (branch verify/auth-workspace-authz)

병목 가설과 실행 계획 개선

추천 조회가 부하에서 실패하는 구간 개선

반복되는 후보 계산을 줄이고, 요청 증가 시의 지연과 실패를 함께 검증

20 RPS · p95

7,450 → 89 ms

동일 단계의 전후 비교

요청 실패율

54.9% → 0%

20 RPS · 35초 구간

측정·검증 조건

OCI · k3s 테스트 환경 · 1 vCPU ARM64 노드 / Core 768Mi · 별도 DB VM · 공개 케이스 100만 건(최근 30일 기록 없는 신규 사용자) · 사용자별 캐시 미스 / 개선 후 공통 후보 캐시 · IAM WireMock · k6 2→5→10→20 RPS · 단계별 35초

문제와 가설

추천 사용자 조회에서 신호가 부족한 신규·비활성 유저에 대해 요청마다 PUBLIC 케이스 전체를 스캔했다. ORDER BY random() (시간 윈도우 없음)과 최근 top-작성자 집계 모두 O(N) 전체 스캔하므로 데이터 규모에 선형 악화가 예상되었다.

가설: ORDER BY random() 전체 Seq Scan이 주 병목, 공통 결과를 매 요청 재계산, 순차 fan-out이 지연을 누적 → 동시 부하에서 1vCPU 포화로 p95 급등·실패 예상

직접 바꾼 부분

① **랜덤 추출 쿼리**: TABLESAMPLE SYSTEM(2): 전체 스캔 대신 페이지 표본만 읽어 O(표본)으로. 단건 574ms→15ms(~37×, EXPLAIN 실측)
② **공통 fallback 결과**: 전역 캐시(@Cacheable, 5분 TTL): 누구에게나 같은 top-작성자 풀을 캐싱해 요청당 full-scan 제거
③ **신호 fan-out 병렬화**: CompletableFuture + 가상 스레드로 신호 계산을 동시 실행(지연 = 합 → max), SecurityContext 전파 유지

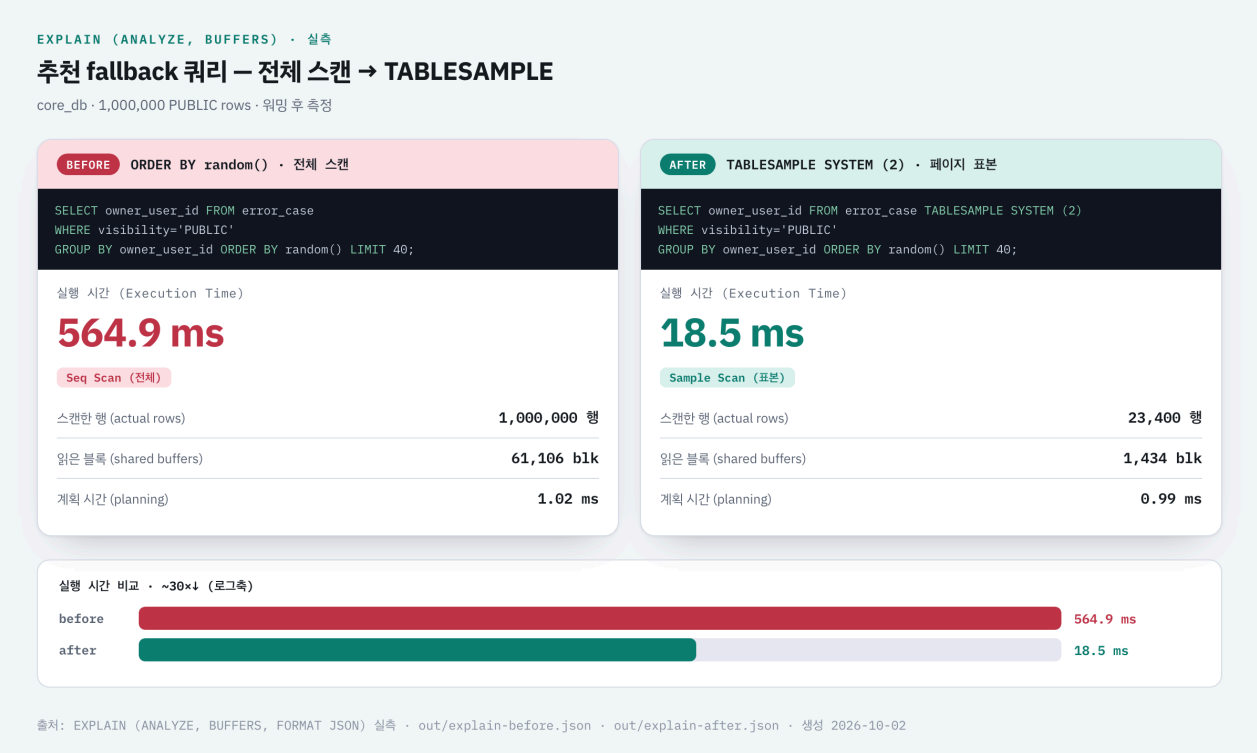


그림 06. ORDER BY random() 전체 스캔 → TABLESAMPLE 표본 스캔으로, 100만 행 → ~2만 행 · 574ms→15ms (~37× ↓, EXPLAIN 실측)

다음 · 추천 조회 개선을 부하 구간별로 검증

동일 부하의 전후 결과와 한계

추천 조회 개선을 부하 구간별로 검증

추천 조회 · 실행 계획 개선 이후의 지연·실패율

LOAD TEST COMPARISON

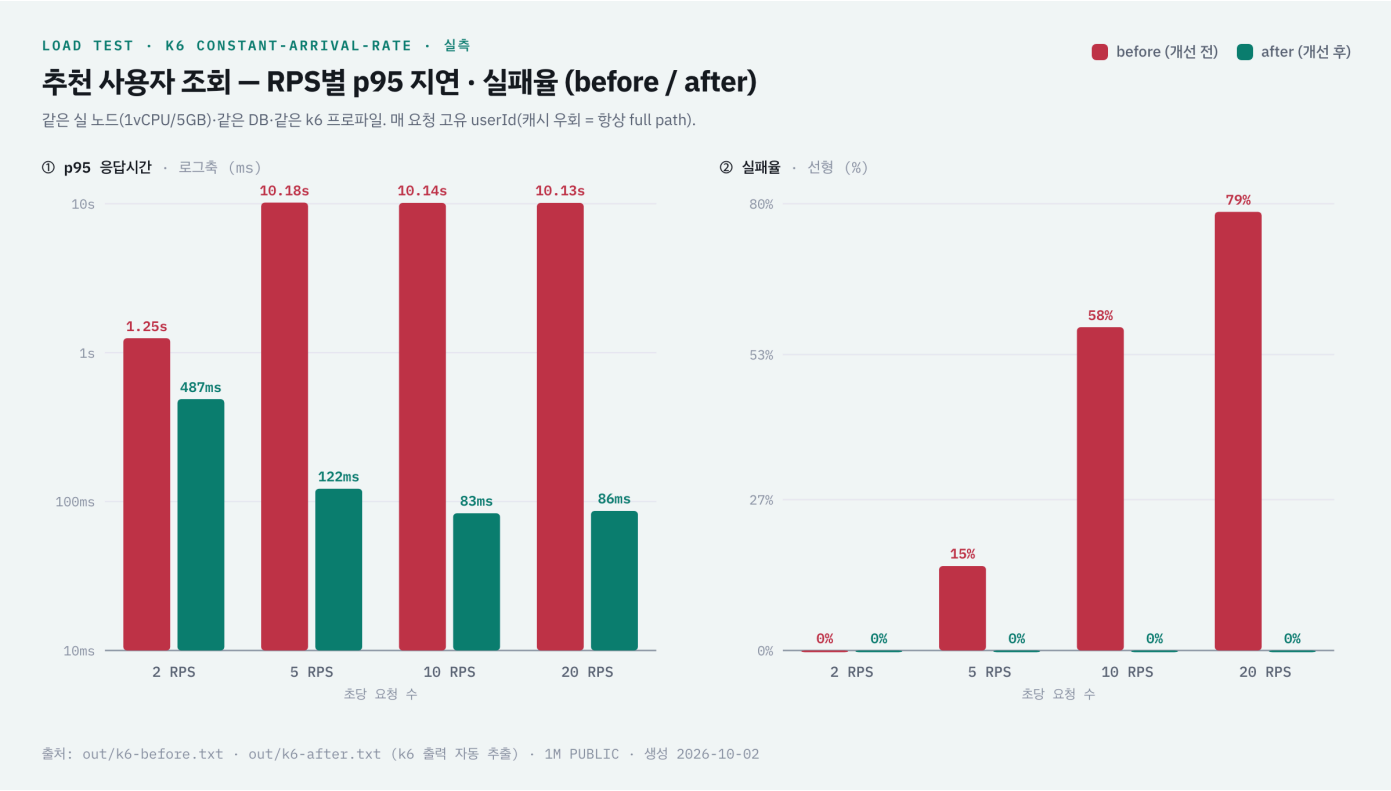


그림 07. RPS 증가 시 개선 전은 5 RPS부터 p95 ~10초·실패율 최대 79%로 붕괴했지만, 개선 후 동일 부하(최악 경로)에서 p95 수십~수백 ms·실패율 0%로 평탄해졌다.

목표 RPS	개선 전 p95 / 실패율	개선 후 p95 / 실패율
2	638 ms / 0%	462 ms / 0%
5	331 ms / 0%	72 ms / 0%
10	7,480 ms / 11.7%	82 ms / 0%
20	7,450 ms / 54.9%	89 ms / 0%

확인한 범위와 남은 한계

IAM 스텁과 합성 데이터의 짧은 부하 구간에서 응답 지연·실패 감소를 확인했습니다. 실제 IAM을 포함한 장시간 처리 한계와 추천 결과의 품질은 별도 검증 대상입니다.

근거 suggested-followees-perf-test.md § 18 · PR #53 https://github.com/studieo-javry/error-archive/pull/53

실패 지점과 처리·복구 설계

재전달을 견디고, 실패한 이벤트를 복구하는 경로

장애 중 처리 위치를 보존하고, 복구 후 집계 결과까지 확인

기존 위험

예상한 위험은 비동기 경계의 전형이었다 — 저장↔발행 사이 유실·중복, 소비 측 재전달 중복, poison 메시지 적체이며, 이는 Outbox 원자성·멱등 upsert·DLQ로 대비해 이벤트 1.2만건 전수 대조에서 누락·중복 0을 확인했다.

반면 격리 하니스에 장애를 주입해 **실제로 발견한 문제는 소비·복구 경계**였다 — insight DB 다운 시 재시도 소진 후 오프셋을 건너뛰어(auto-commit) 유실되고, DLQ 재처리가 배치 단일 트랜잭션이라 1건 실패가 정상 49건까지 롤백해 정체되는 결함 2건을 찾아 고쳤다.

처리와 복구 설계

① **발행 경계** — Transactional Outbox: 도메인 변경과 이벤트 적재를 한 트랜잭션으로 원자화하고 outbox에서 Kafka로 재발행해, "DB는 됐는데 발행 누락 / 발행은 됐는데 롤백"을 차단.
② **소비·집계 경계** — 멱등 upsert + 처리 성공 후 커밋: 집계를 멱등 키 기반 native upsert로 저장해 재전달 중복을 흡수하고, enable.auto.commit=false + 무한 재시도(skip 금지)로 처리에 성공한 뒤에만 오프셋을 커밋 → DB 장애 중에도 유실 없음(#1 수정, 90s 장애·200건 200/200 무손실)
③ **실패 이벤트 복구 경계** — DLQ 행별 독립 트랜잭션: poison은 DLQ로 격리하고, 재처리는 배치 단일 tx가 아니라 행별 독립 트랜잭션으로 돌려 실패 행만 backoff/DEAD 처리 → 1건 실패가 정상 건을 막지 않음(#2 수정), 주기·배치 튜닝으로 1,000건 70→20초

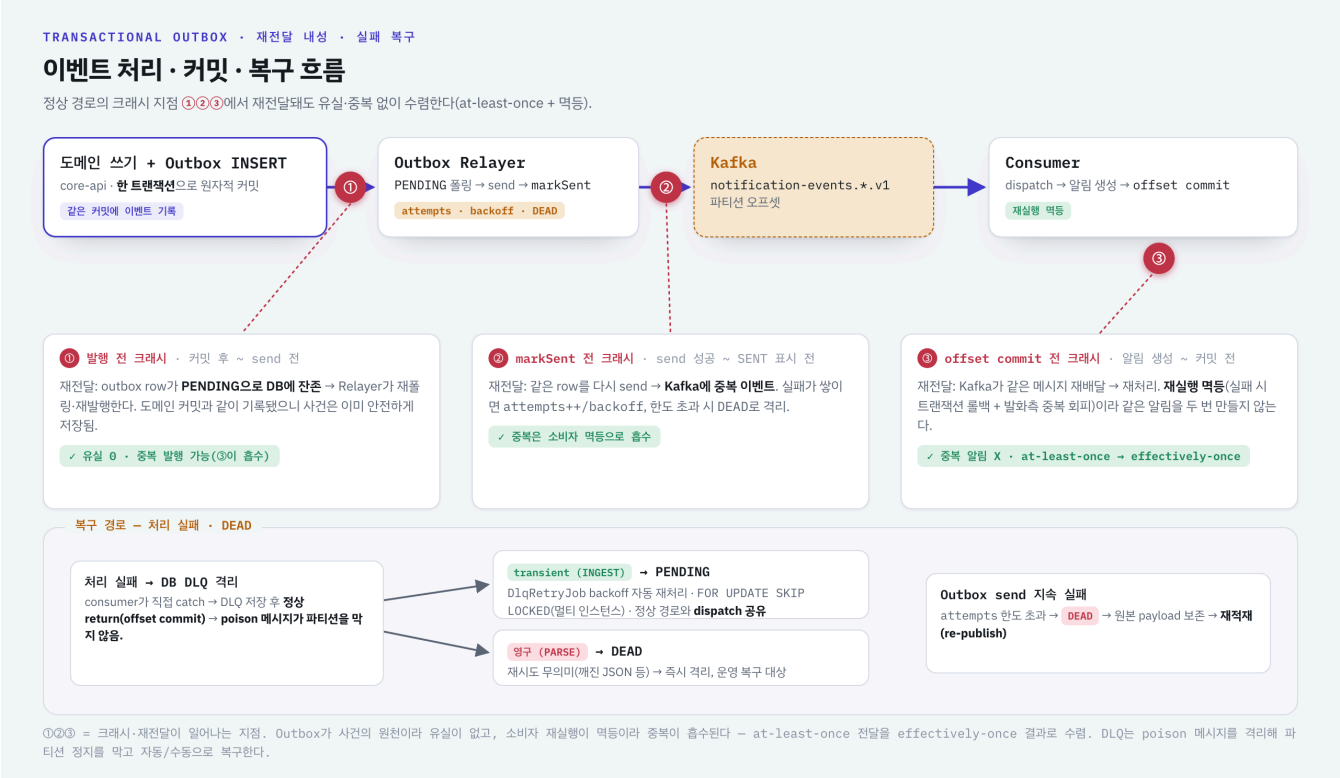


그림 08. Outbox(한 트랜잭션)로 유실을 막고 멱등 재실행으로 중복을 흡수 — ①②③ 어디서 재전달돼도 effectively-once. 실패는 DB DLQ 격리 → 자동 재처리로 복구

다음 · 이벤트 장애·중복 재전달과 복구를 검증

장애 시나리오와 복구 결과

이벤트 장애·중복 재전달과 복구를 검증

이벤트 신뢰성 · 장애 중 보존과 복구 후 수렴

DLQ 복구 · 1,000건

70 → 20초

약 71% 단축 · 별도 복구 실험

DB 장애 검증

200건 처리 / 유실 0건

90초 장애 후 복구한 시나리오

측정·검증 조건

로컬 Docker · insight 1 vCPU/1.5 GB · prod 프로파일(실 Kafka/PG)
DLQ: Hikari 10·실패 원인 제거·1,000건·주기 3초·배치 50→250
DB 장애: 90초 잠금·200건 → 200/200 무손실

검증 시나리오	성공 판정 / 관찰
DB와 DLQ 저장 불가	장애 중 오프셋 전진 없음 → 복구 후 200건 처리
중복 재전달 / 동시 중복	1,000 이벤트 ×5회 재전달·동시 투입 → 1,000건 수렴 (중복 0·누락 0)
DLQ 대량 복구	원인 제거 후 1,000건 처리 완료까지 70→20초

한계와 다음 판단

이번 검증은 단일 브로커·단일 파티션·단일 소비자 범위에서의 중복/누락 수렴(effectively-once)까지만 확인했고, 멀티 파티션·리밸런스·복제는 포함하지 않는다. 특히 장기 DB 장애 시 적체(Outbox PENDING·DLQ 누적과, DLQ sink 자체가 실패하면 2회 재시도 후 skip되는 경로)는 미검증이라, 다음은 지속 장애 하한에서의 백프레셔·보존 한도 측정이 필요하다.

근거 PR #49 github.com/studieo-javry/error-archive/pull/49 · PR #50 https://github.com/studieo-javry/error-archive/pull/50 · 문서 event-resilience.md

비동기 경계의 선택 이유와 책임

결제 응답과 주문 완료의 경계를 분리

PG 승인과 주문 확정을 분리하고, 재고 예약·취소 보상·배송 상태별 환불을 연결한 주문 생애주기를 설계했다.

진행 기간 2025.02 – 05 **구성·역할** 개인 프로젝트 · 주문/결제 백엔드 설계·구현

담당 범위 주문·결제(Toss)·재고·배송·환불 백엔드. Kafka 처리 경계·DB·Saga/Outbox 설계부터 계측·장애·부하 검증까지 담당했다.

왜 주문 확정을 응답 뒤로 옮겼는가

PG 승인이 성공해도 주문 생성·재고 확정은 별도로 지연되거나 실패할 수 있다. 한 요청에서 모두 기다리면 주문 처리 비용이 결제 응답까지 늘어난다. **승인 결과 응답과 주문 확정을 분리**하고, 주문의 완료 여부는 별도 상태로 관리했다.

Kafka·Outbox를 선택한 이유

주문 처리를 응답 경로 밖으로 옮기고 **이벤트 재전달·소비자 분리·파티션별 처리**를 활용하려 Kafka를 선택했다. 승인 기록과 Outbox를 같은 DB 트랜잭션에 남긴 뒤 발행해, 내부 상태만 저장되고 이벤트가 누락될 위험을 보완했다.

비동기화로 추가된 책임과 보완

완료 시점의 차이	승인 후에도 주문은 처리 중일 수 있다. **결제 응답 지연과 주문 완료 지연을 별도로 측정**해, 응답이 빠른 것과 주문이 제때 끝나는 것을 구분했다.
주문·보상 실패	주문 확정 실패는 **결제 취소로 보상**하고, 보상 실패는 재시도·미완료 상태로 추적했다. 재시도 한도 초과 건은 별도 상태로 남겨 수동 개입 대상으로 구분했다.
재전달·중복 반영	이벤트 재전달을 전제로 **처리 기록·상태 검사·DB 제약**을 적용했다. 같은 주문·취소 실행이 재시도돼도 중복 반영되지 않도록 처리했다.

설계 시 목표 **VU100+·누적 수만 건**에서도 **결제 완료 이벤트 발행 → 주문 레코드 생성 완료 p99 ≤ 3초**. 결제 응답과 별개로 검증할 목표이며, 아래 장애 검증 결과와 구분한다.

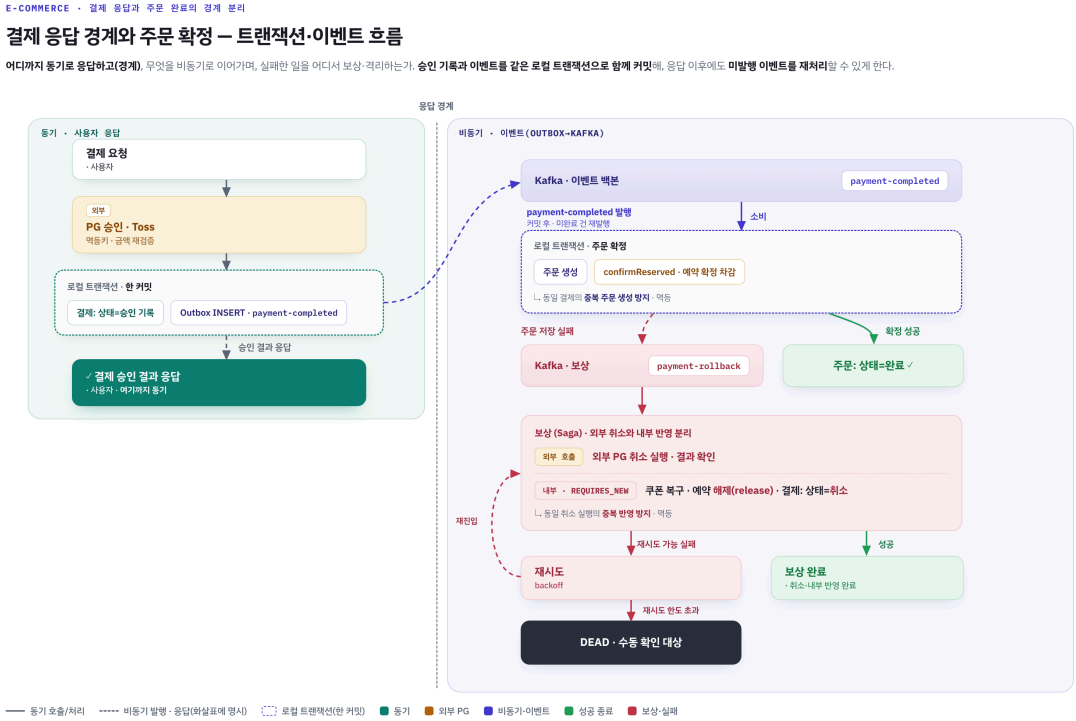


그림 10. 승인 기록·Outbox를 함께 커밋하고 주문 확정·취소 보상을 이벤트로 연결했다. 중복 전달은 멱등 처리와 DB 제약으로 방어한다.

다음 · 주문 생애주기와 장애 검증 결과

전체 도메인 흐름과 검증 범위

주문 생애주기와 장애 검증 결과

결제·주문 경계 · 재고·배송·환불로 이어지는 정책

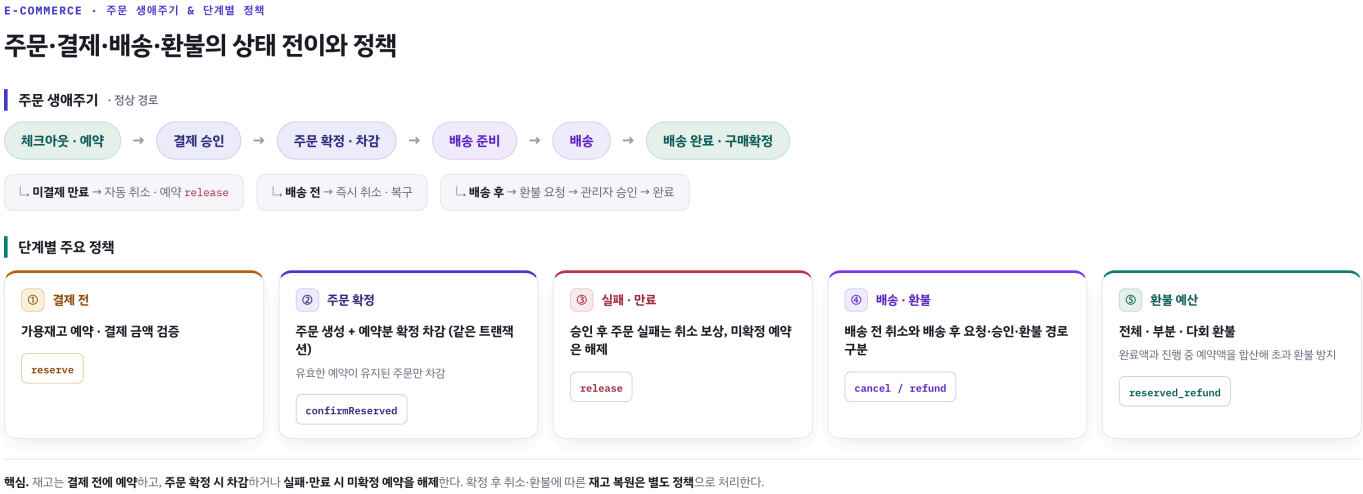


그림 11. 재고 예약 이후 확정·해제를 구분하고, 배송 상태별 취소·환불과 다회 환불 예산을 연결했다. 동일 취소 실행의 재시도는 중복 반영하지 않는다.

장애·재전달 검증

최종 상태 불일치 0건 · 중복 보상 0건

3노드 KRaft · 컨슈머 3개 / 파티션 6개 · 결제 완료 이벤트 1,000건 · 수동 오프셋 커밋
주문 저장 실패·Kafka 재전달을 유도한 조건에서 **1,000건 모두 결제 취소·주문 미생성으로 수렴**함을 확인했다.

이후 검증
주문 완료 p99 ≤ 3초는 별도 개선이 필요했다. **응답·확정 경로를 나눠 계측**하고 조회·Outbox 개선, 재고 교착 해소·확정 배치화로 이어갔다.

근거 결제·주문 상태 전이 및 장애 주입 기록(1,000건) · https://gitlab.com/jiinhong989/e-commerce-server

조회·Outbox의 응답 병목을 계측하고 개선

가주문 조회와 Outbox 동기 대기를 줄이고, 비동기 완료 갱신의 누락 위험을 재발행 경로로 보완했다.

가주문 조회 최적화

p99 671 → 329ms

fetch join으로 N+1 제거 · 커버링 인덱스 적용

로컬 격리 벤치 · VU100

Outbox 완료 갱신 비동기화

호출 경로 p99 736 → 0.2ms

완료 상태 갱신을 별도 executor로 분리

기존 격리 벤치 · VU100 · 2,000건

상단 지표·전후 그래프의 측정 조건

로컬 격리 벤치 · 2vCPU/6GB · 단일 MySQL Testcontainers · VU100 · Micrometer @Timed

조회: QueryIndexTuningIT · Outbox: OutboxAsyncTuningIT / 2,000건

가설과 기준선

3노드 Kafka KRaft E2E의 VU150에서 **주문 완료 p99 3,309ms · 처리량 약 65 req/s 정체**로 3초 목표에 미달했다. **Elastic APM → Micrometer @Timed → 격리 벤치 전후 비교**로 병목을 좁혀 가주문 조회·Outbox 동기 갱신부터 개선했다. 결제 응답의 개선이 주문 완료 SLA까지 바꾸는지는 별도로 검증했다.

개선 대상	실제 변경 내용
가주문 조회	연관 데이터 조회를 **fetch join으로 재구성해 N+1을 제거**하고, **커버링 인덱스**를 적용해 조회 비용을 줄였다.
Outbox 완료	Kafka 발행 성공 이후의 **COMPLETE 갱신을 별도 executor로 분리**해, 호출 경로의 동기 DB 갱신 대기를 줄였다.

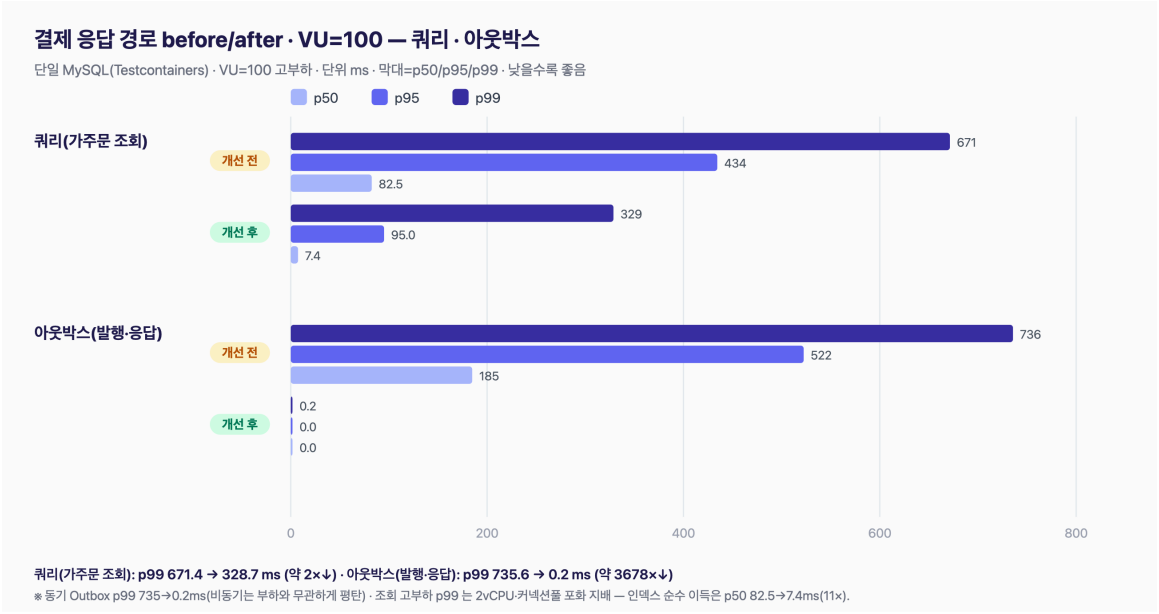


그림 12. VU100 격리 벤치: 조회 p99 671→329ms · Outbox 호출 경로 p99 736→0.2ms

비동기화 후속 보완 · 미완료 상태의 복구

Kafka 발행 성공 후 COMPLETE 갱신이 실패하면 **PENDING이 남을 수 있는 위험**을 확인했다. **CallerRunsPolicy·예외 로깅·이벤트 영속화·재발행 폴러**를 추가하고, OutboxRelayIT로 재발행·FAILED 전환·제외 조건을 검증했다.

다음 개선 대상을 바꾼 관찰

조회·Outbox 개선 뒤에도 주문 완료 **SLA 범위는 늘지 않았다**. 그 두 개선은 '결제 응답' 경로를 빠르게 할 뿐, SLA를 좌우하는 **'컨슈머 확정 처리량'과는 다른 경로**이기 때문이다. 컨슈머를 추적하니 재고 교착과 주문당 약 8회 DB 왕복이 병목이었고, 이 왕복이 단일 DB를 포화시켜 — DB가 벽이라 파티션·컨슈머 확장으론 더 늘지 않는 상황 — 왕복 자체를 줄이는 데드락 해소·주문 확정 배치화로 방향을 옮겼다.

근거 VU100: QueryIndexTuningIT · OutboxAsyncTuningIT / E2E: payment-order-throughput-journey.md §1–4
https://gitlab.com/jiinhong989/e-commerce-server

Outbox 비동기화 뒤 남는 PENDING을 복구

응답 경로에서 분리한 완료 갱신에 실패 감지와 재발행을 더해, 미완료 상태가 방치되는 경로를 보완했다.

> **비동기화 뒤 확인한 빈틈**
>
> Kafka 발행 성공과 DB의 COMPLETE 기록은 별개의 성공 기준이다. 발행 성공 콜백 뒤 실행하는 **completeAsync**가 실패하거나 실행되지 않으면, 메시지는 이미 발행됐어도 **Outbox 행은 PENDING으로 남는다.** 기존에는 이를 회수할 재발행 폴러가 없었다.

실패 지점에 맞춰 추가한 장치

실패 지점	추가한 처리 · 선택의 이유
실행 큐 포화	큐 10,000의 outboxExecutor에 **CallerRunsPolicy**를 적용했다. 실행 중 큐가 포화되면 호출 스레드가 COMPLETE를 수행하도록 해, 작업 거부 대신 **백프레셔**를 주었다.
비동기 갱신 실패	**AsyncUncaughtExceptionHandler**로 void @Async 예외를 명시 로깅하고, 폴러에서 **PENDING 수**를 기록했다. 호출자에게 전파되지 않는 실패와 미완료 상태를 확인할 수 있게 했다.
재발행 정보 부재	**BEFORE_COMMIT**에 OutboxService.persist(topic, event)로 이벤트를 직렬화해 적재했다. **topic·payload·event_type·retry_count**를 추가해 재발행 정보와 재시도 이력을 남겼다.
장기 PENDING	**60초 주기**로 **30초 이상** 지난 PENDING을 재발행하고, 성공 콜백에서 COMPLETE를 갱신한다. **retry_count ≥ 10은 FAILED**로 전환해 자동 재시도를 종료하고 수동 개입 대상으로 구분했다.

OutboxRelayIT · 재발행과 제외 조건 검증

검증 조건	확인한 결과
오래된 PENDING	**재발행 수행 · retry_count 증가** 확인
재시도 한도 도달	**FAILED 전환** 확인 · 수동 개입 대상으로 분리
최근 PENDING / payload 없음	**재발행 대상에서 제외** 확인 · payload 없는 행은 자동 복구 대상이 아님

중복을 허용하는 복구

재발행으로 **중복 전달이 가능**해지므로 멱등 소비를 전제로 한다. 30초 가드는 정상 COMPLETE와의 중첩을 줄일 뿐, 중복을 차단하지는 않는다.

응답 지연과 남은 검증

큐 포화 시 COMPLETE가 호출 경로에서 실행돼 **응답 지연이 늘 수 있다.** 큐 포화·프로세스 강제 종료·중복 소비의 최종 결과는 별도 검증이 필요하다.

근거 AsyncConfig · OutboxService · OutboxRelayScheduler · OutboxRelayIT
https://gitlab.com/jiinhong989/e-commerce-server

결제 전 재고를 예약하고 주문 확정에서 차감

품절 판단을 결제 전으로 옮기고, 확정·해제의 경합과 중복 요청을 상태 전이로 제어했다.

주문 확정의 전제 · 결제 전에 확보한 예약을 소비

문제 · 돈을 받은 뒤 품절을 판단

주문 확정 때 **처음 재고를 확보**하면, 다른 주문이 먼저 소진한 경우 승인된 결제를 취소·환불해야 한다. 결제 성공 응답과 주문 이행이 어긋나는 경로다.

선택 · 품절 판단을 결제 앞으로

체크아웃에서 **stock − reserved ≥ 주문수량**일 때 예약한다. **유효한 예약이 유지된 주문**은 확정 시 가용재고를 다시 경쟁하지 않고, 확보한 예약분을 실제 차감으로 전환한다.

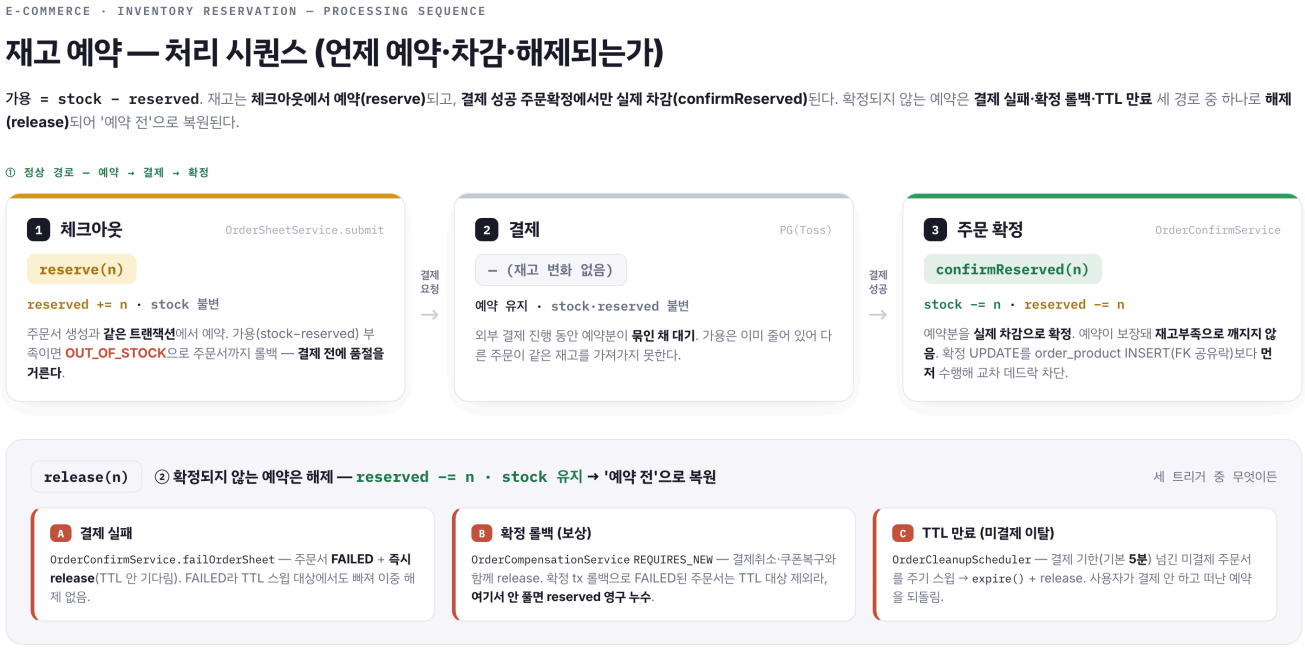


그림 13. 예약은 가용재고를 확보하고, 확정은 stock·reserved를 함께 줄인다. 결제 실패·확정 롤백·미결제 만료에서는 미확정 예약분만 해제한다.

경합·중복에서 확인한 상태 전이

검증 상황	확인한 결과
만료 해제 ↔ 주문 확정	동시에 진입해도 **같은 예약이 확정·해제로 이중 반영되지 않음**을 상태 전이 테스트로 확인
동일 confirm 재전달	멱등 처리로 **stock·reserved의 중복 차감**을 차단
동일 release 재전달	멱등 처리로 **reserved의 중복 해제**를 차단

> 결제 실패는 즉시 해제, 확정 롤백은 **별도 보상 트랜잭션(REQUIRES_NEW)**에서 해제한다. **결제 기한(5분)**이 지난 미결제 주문서는 만료 처리해 예약을 회수한다.

선택의 비용 · 미완료 예약의 회수

결제를 마치지 않은 사용자도 예약 중에는 가용재고를 점유한다. 확정된 판매의 재고 복원과 **미확정 예약의 해제**를 구분하고, 실패·만료 경로에서 예약이 남지 않도록 관리했다.

다음 경계 · 같은 상품의 동시 갱신

예약은 재고 확보 시점을 옮긴다. 그러나 같은 상품 행의 **락 경쟁**은 남으므로, 상품에 주문이 집중되는 조건에서 교착과 행 대기를 따로 확인했다.

다음 사례: 핫 상품 부하에서 잠금 획득 순서를 분석하고 재현한 교착 경로를 개선.

근거 reserve / confirmReserved / release · 만료·확정 경합 및 중복 전이 검증
https://gitlab.com/jiinhong989/e-commerce-server

핫 상품 부하에서 발견한 재고 교착을 해소

재고 처리 교착으로 인한 주문 실패를 줄이고, 단일 상품·다상품 조건에서 개선 결과를 확인했다.

핫 상품 E2E · 실패율

76–97% → 0%

같은 상품에 동시 주문 · VU8–64

다상품 교차 · 교착으로 인한 롤백

480 → 0건

(A,B) / (B,A) · 교차 주문 960회

문제 · 동일 상품의 동시 주문에서 재고 처리 교착 발생

동일 상품의 동시 주문과 다상품 교차 주문을 검증하는 과정에서, **잠금 획득 순서 충돌로 데드락과 주문 처리 실패가 발생하는 문제**를 확인했다. 재고 처리를 **조건부 UPDATE로 전환하고 관련 INSERT보다 먼저 수행**하도록 변경했으며, 다상품 갱신은 **productId 오름차순**으로 통일했다.



그림 14. 단일 상품의 잠금 흐름을 비교한 개념도. 공유락을 선점하던 INSERT보다 재고 UPDATE를 먼저 수행해 상호 대기 경로를 바꿨다.

해결 · 원인에 맞춰 갱신 방식과 실행 순서 변경

조건	기존 잠금 경로	변경한 동작
단일 상품	FK 검사로 **공유락(S)**을 보유한 뒤 **배타락(X)**을 요구해 상호 대기	SELECT FOR UPDATE를 **조건부 UPDATE로 교체** **재고 UPDATE → 관련 INSERT** 순서로 실행
다상품	**A→B / B→A**처럼 트랜잭션마다 상품 잠금 순서가 달라 순환 대기	**productId 오름차순**으로 재고 갱신 순서를 통일

남은 제약 · 같은 상품 행의 잠금 대기

조건부 UPDATE도 배타락을 획득하므로 동일 상품 행의 대기·직렬화는 남는다. 이번 결과는 재현한 교착 경로의 개선이며, 핫 상품의 지연·처리량 상한은 추가 검증이 필요하다.

다음 사례에서는 **서로 다른 상품** 조건으로 주문 확정을 배치화하고 3초 SLA를 검증했다.

근거 payment-order-throughput-journey.md §5 · StockDeductContentionIT · MultiProductDeadlockIT

배치 확정으로 3초 SLA 검증 범위를 확대

응답 경로 개선 후에도 남은 주문 완료 지연을 추적하고, 건별 DB 처리를 묶어 3초 SLA를 VU250까지 검증했다.

3초 SLA를 지키는 부하 범위 확대

VU100 → 250까지 SLA 충족

기존: VU120–130 부근에서 초과
개선 후: VU100·150·200·250 모두 충족

E2E 처리량 · 초기 정체 수준 → 최종 실측

약 65 → 139.3 req/s

약 2.1배 · 최종 값은 VU250에서 측정
격리 벤치 61→162건/s와 별도 비교

동일 VU150 p99 3,309→1,775ms (46.4%↓)
검증 상한 VU250 p99 2,529ms · 실패율 0% · 초과 부하는 미검증

완료 SLA의 정의와 비교 조건
결제 완료 이벤트 발행→주문 레코드 생성 완료 · p99 ≤ 3,000ms
로컬 3노드 KRaft / MySQL Testcontainers · 노드 2 vCPU · closed-model VU100·150·200·250 · 서로 다른 상품 주문

01 병목 — 수신은 배치였지만 DB 처리는 여전히 건별

조회·Outbox 지연은 줄었지만 **완료 SLA의 부하 범위는 늘지 않았다.** 재고 교착 해소 뒤 확정 경로를 추적하니, Kafka는 이미 배치로 수신해도 DB는 **주문마다 별도 트랜잭션·약 8회 왕복**을 반복하고 있었다.

02 선택 — 주문당 DB 왕복·커밋을 배치 단위로 축소

poll 묶음을 한 트랜잭션으로 묶고, **IN 조회·bulk INSERT·상품별 예약분 합산 확정·1회 커밋**으로 반복 비용을 줄였다. 별도 VU4·서로 다른 상품 격리 벤치의 확정 처리량은 **61→162건/s(2.66배)**였다.



그림 15. §6 E2E 실측. VU100·150·200·250에서 p99 3초 이내·실패율 0%를 확인했다. 약 65 req/s는 초기 정체 기준선이며, VU250 초과는 미검증이다.

배치 처리의 비용과 보완

예약 모델로 재고 자원 경쟁을 앞단에 옮겼고, 재고를 체크아웃에서 예약하고 확정은 예약분을 차감만 하므로, 확정 배치가 재고 부족으로 깨지지 않는다. 그러나 **한 건 실패로 배치 전체가 롤백**될 수 있다. ①결제 실패분 사전 분리 · ②배치 실패 시 건별 폴백 · ③멱등 소비와 재전달로 보완하여 최악의 시나리오를 유실·정지가 아니라 **'드문 1건 재처리'**로 제한했다.

남은 제약

원자 UPDATE도 같은 상품 한 행에 X락을 잡으므로, 단일 초인기 상품에 몰리면 그 행에서 직렬화돼 초고부하 p99가 오른다. 예약 모델에선 이 핫 행 UPDATE가 예약·확정·해제 세 지점에서 일어나는데, 특히 예약은 결제 전 '동기' 요청이라 직렬화가 사용자 체크아웃 응답에 바로 드러난다. 개선 방향은 ① Redis 원자 카운터로 DB 핫 행을 크리티컬 패스(체크아웃 예약)에서 제거(비동기 반영) ② 재고 버킷 분할로 락 분산 ③ 대기열로 유입 평탄화 — 플래시 세일 같은 핫 트래픽 분포를 보고 선택한다.

빠른 결제 응답과 빠른 주문 완료는 별개다. 목표 구간의 반복 DB 비용을 줄여 SLA 검증 범위를 넓혔다.

근거 payment-order-throughput-journey.md §1·4·6 · PaymentOrderSagaSlaIT
OrderConfirmBatchThroughputIT · 수신 배치와 DB 처리 배치를 구분

전체·부분·다회 환불을 실행 단위로 설계

결제별 환불 예산과 건별 취소 기록을 분리해, 여러 번의 환불과 같은 실행의 재시도를 구분

지원하는 환불을 실행 기록으로 구분

전체·부분 환불	**cancel_type(FULL/PARTIAL) · cancel_amount**로 취소 유형과 금액을 기록
다회 부분 환불	같은 Payment 아래 **PaymentCancel을 건별로 생성**하고 서로 다른 멱등키 사용
같은 실행의 재시도	**기존 실행 기록과 저장된 멱등키**를 재사용해 새 환불과 구분

배송 전은 즉시 취소, 배송 후는 환불 요청 → 관리자 승인. 실행 기록과 정산 경로는 공통화.

핵심 ERD · 예산은 Payment, 실행은 PaymentCancel

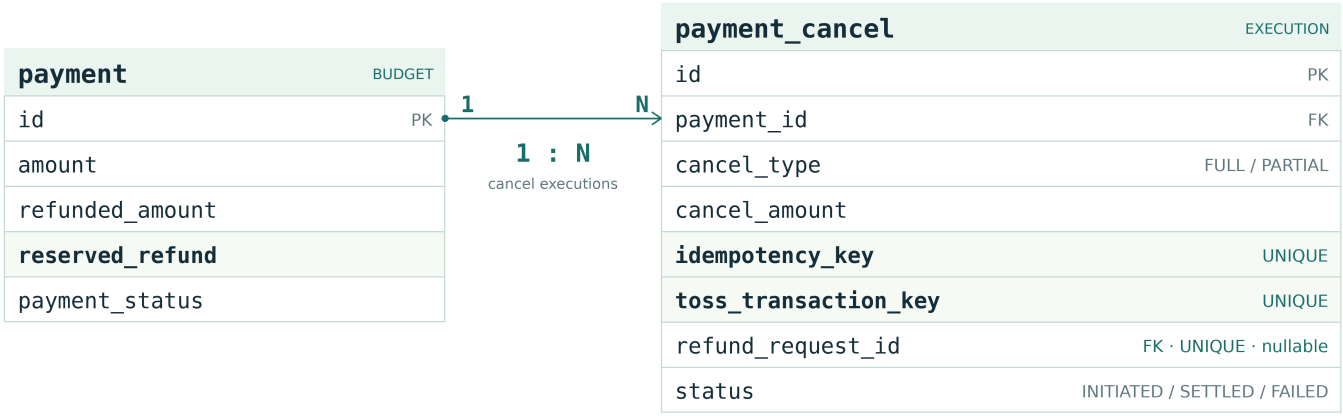


그림 16. Payment는 결제별 환불 예산을 관리하고, PaymentCancel은 전체·부분 환불의 개별 실행과 결과를 누적한다.

핵심 컬럼과 제약이 보호하는 것

컬럼 · 제약	설계 이유와 처리 기준
Payment.reserved_refund 잠금 조회하는 결제 행의 예약액	완료액뿐 아니라 **진행 중 환불액도 예산에 포함**해, 서로 다른 동시 부분 환불이 결제금액을 초과하지 않도록 보호
idempotency_key 실행별 UNIQUE	**같은 실행의 재시도는 같은 키**, 새로운 부분 환불은 다른 키로 구분해 PG 재시도와 별도 취소를 식별
toss_transaction_key PG 취소 거래키 UNIQUE	응답·웹훅 등 여러 경로로 도착한 **같은 PG 취소 결과의 중복 반영**을 DB 제약으로 차단
refund_request_id UNIQUE · NULL 가능	승인 요청 하나에 **실행 기록이 중복 생성되지 않도록** 연결. 환불 요청 없이 즉시 취소하는 실행은 NULL로 구분

예산은 PG 호출 전에 확보하고, 정산은 성공 후 마감

일반 SUM의 과거 스냅샷으로 **진행 중 예약을 놓치는 문제**를 확인했다. 판단 기준을 잠긴 Payment의 예약액으로 옮겨, PG 호출 전에 예산을 확보한다.

환불 가능액 **amount(결제금액) − refunded_amount(완료액) − reserved_refund(예약액)**

① TX1 · 예산 확보	② PG · 외부 호출	③ TX2 · 내부 정산
Payment 잠금 → 예산 확인·예약 INITIATED 실행 기록 저장 후 커밋	저장된 멱등키로 취소 호출 **DB 트랜잭션 밖**에서 응답 대기	예약분 정산·정책상 재고·쿠폰 반영과 **SETTLED 전이**를 함께 커밋

진행 중 금액을 먼저 예약해 동시 요청의 예산을 보호하고, **외부 응답을 기다리는 동안 DB 락을 유지하지 않도록** 경계를 분리했다.

PG 성공 뒤 내부 정산이 끊긴 경우의 복구와 동시 환불 검증은 다음 환불 복구 사례에서 설명합니다.

근거 https://gitlab.com/jiinhong989/e-commerce-server · payment / payment_cancel · TX1 / PG cancel / settle

PG 취소 성공 후 미완료 환불을 복구

저장된 실행 기록으로 미완료 정산을 복구하고, 두 프로세스의 중복·동시 환불을 PG 결과와 내부 장부로 검증

> **검증 환경 · 내부 로직과 멀티 인스턴스 실험**
> **앱 2개 OS 프로세스 · 공유 실 MySQL 8.0 · 상태저장 PG 테스트 서버**
> 별도 **RefundLifecycleIT 14건**(모의 PG) · PG 응답 유실·내부 반영 실패·중복 및 동시 환불 검증

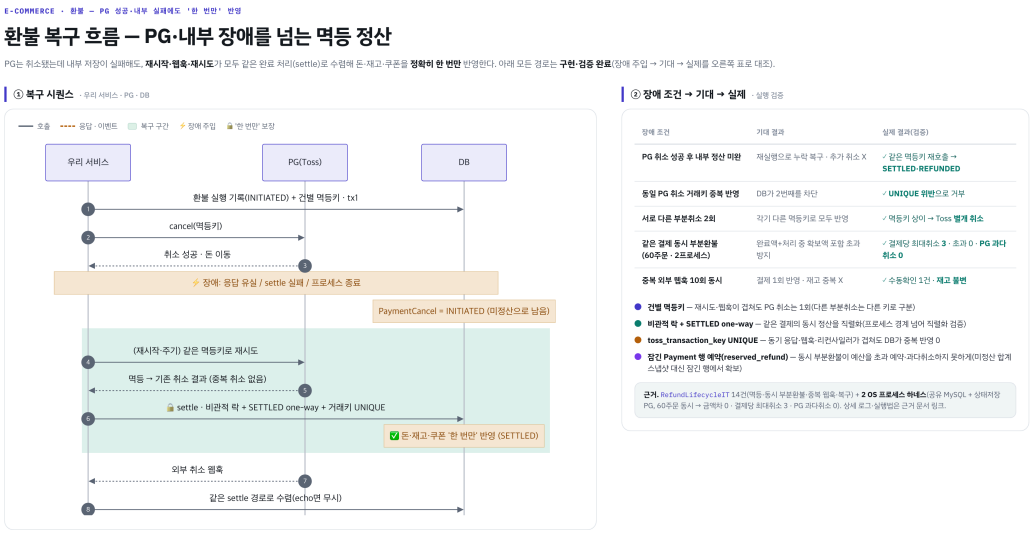


그림 17. PG 성공 뒤 남은 INITIATED를 같은 멱등키로 재처리해 공통 settle로 복구한다. 테스트 서버의 취소 결과와 내부 환불액을 함께 대조했다.

언제·어떻게 다시 처리하는가

① 미완료 실행을 주기적으로 찾기

리컨사일러가 **INITIATED 잔존 건·승인 후 미완료 요청**을 주기적으로 찾아 다시 처리한다. PG 호출 전에 저장한 **PaymentCancel**을 복구 단위로 삼는다.

② 저장한 멱등키로 PG 결과 확인

새 환불로 만들지 않고 **같은 멱등키로 취소를 재호출**한다. 이미 성공했다면 기존 결과를 받아 추가 취소를 막고, 응답·웹훅·재시도를 **공통 settle**로 모은다.

③ 내부 반영을 한 트랜잭션으로 마감

PaymentCancel·Payment를 잠근 뒤 예약분 정산, 환불액과 정책상 재고·쿠폰 반영, **SETTLED 전이**를 함께 커밋한다. 중단된 실행은 저장 기록으로 이어 처리한다.

중복 정산·초과 환불을 막는 경계

같은 실행 · 락 + 완료 상태 검사

동시 정산을 **DB 비관적 락**으로 직렬화하고, 이미 **SETTLED면 no-op**으로 끝낸다. 두 프로세스가 같은 실행을 처리해도 환불액·재고·쿠폰을 다시 반영하지 않는다.

같은 PG 취소 · 거래키 UNIQUE

동기 응답·웹훅·재시도가 같은 취소를 전달해도 **toss_transaction_key UNIQUE**로 중복 저장을 차단한다. 다른 부분 취소는 별도 실행·멱등키로 구분한다.

다른 부분 환불 · 잠긴 행의 예산

잠긴 **Payment의 refunded_amount + reserved_refund**로 예산을 검사하고 PG 호출 전에 예약한다. 다른 동시 요청의 예약액까지 반영해 **결제금액 초과 취소**를 막는다.

장애·경쟁 검증 · 내부 장부와 외부 PG를 함께 확인

검증 조건	확인한 결과
2 프로세스 · 동시 부분 환불 60주문 × 결제당 6건 요청	**PG 과다취소 0 · PG–내부 금액 차이 0 · lost update 0** 해당 테스트 예산에서 결제당 최대 취소 3건. 외부 취소 결과와 내부 장부를 함께 대조.
PG 성공 · 내부 실패	DB 반영 실패·응답 유실·프로세스 종료 후 **같은 멱등키로 추가 취소 없이 정산과 환불 상태 복구**.
승인·웹훅 경쟁 / 중복	공통 settle의 락·SETTLED 검사·거래키 UNIQUE로 **같은 PG 취소의 중복 반영 차단**.
서로 다른 부분 취소 2회	서로 다른 멱등키로 **두 취소 모두 반영**. 동일 실행의 재시도는 기존 결과로 처리.
외부 웹훅 중복 전달	내부 요청 없는 외부 취소는 **수동 확인 1건**으로 모으고 **재고 불변** 확인.

남은 제약

상태저장 PG 테스트 서버 기준이다. **실 Toss·배송사 연동, 회수·검수에 따른 재입고, 환불 반려 중 배송 전이 보호**는 별도 검증 범위로 남겼다.

근거 https://gitlab.com/jiinhong989/e-commerce-server · RefundLifecycleIT · 2 OS 프로세스 하니스

임계 영역과 지켜야 하는 규칙

쿠폰 발급의 임계 영역과 최종 데이터 검증

중복 방지와 재고 보호를 각각 확인하는 동시성 설계

검증 규모

VU 100 · 앱 2 인스턴스
총 3,000 요청 × 5회 반복

실행 로그의 정의를 그대로 사용

최종 결과

중복 0 · 초과 0 · 남은 재고 0 · 5회 일관

서로 다른 정확성 지표로 분리

측정·검증 조건

2 앱 인스턴스(Spring Boot 3.3·JDK 17) · 공유 Redis 1(Redisson 분산 락) + MySQL 8.0.1 (UNIQUE 인덱스 user_id,coupon_id)
초기 재고 100(경합)/5,000(중복) · 사용자 3,000명(또는 동일 1명) · 부하 = VU 100 동시, 2 인스턴스 라운드로빈 · 표본 = 3,000 요청 × 5회 반복 · 장애 주입 없음(로컬 락 대조 포함)

지켜야 하는 규칙

① 동일 사용자 1회 발급(UNIQUE user,coupon)
② 전체 발급 수 ≤ 재고(쿠폰 단위 분산 락)
③ 성공 응답 수 = 발급 내역 수 = 재고 차감 수(실패 응답과 DB 결과 일치)

선택의 이유

발급은 **재고 차감 + 발급 저장(DB)**이 한 임계 영역이라, Redis 원자 연산(Lua/INCR)만으론 외부 DB 트랜잭션 경계를 못 묶음 → 분산 락으로 구간 전체를 직렬화. 비관·낙관 락이 늘 느린 건 아니지만, 멀티 인스턴스의 공유 임계 영역을 앱 레벨로 통제하려 분산 락 선택 (핫 쿠폰 직렬화 병목은 감수)

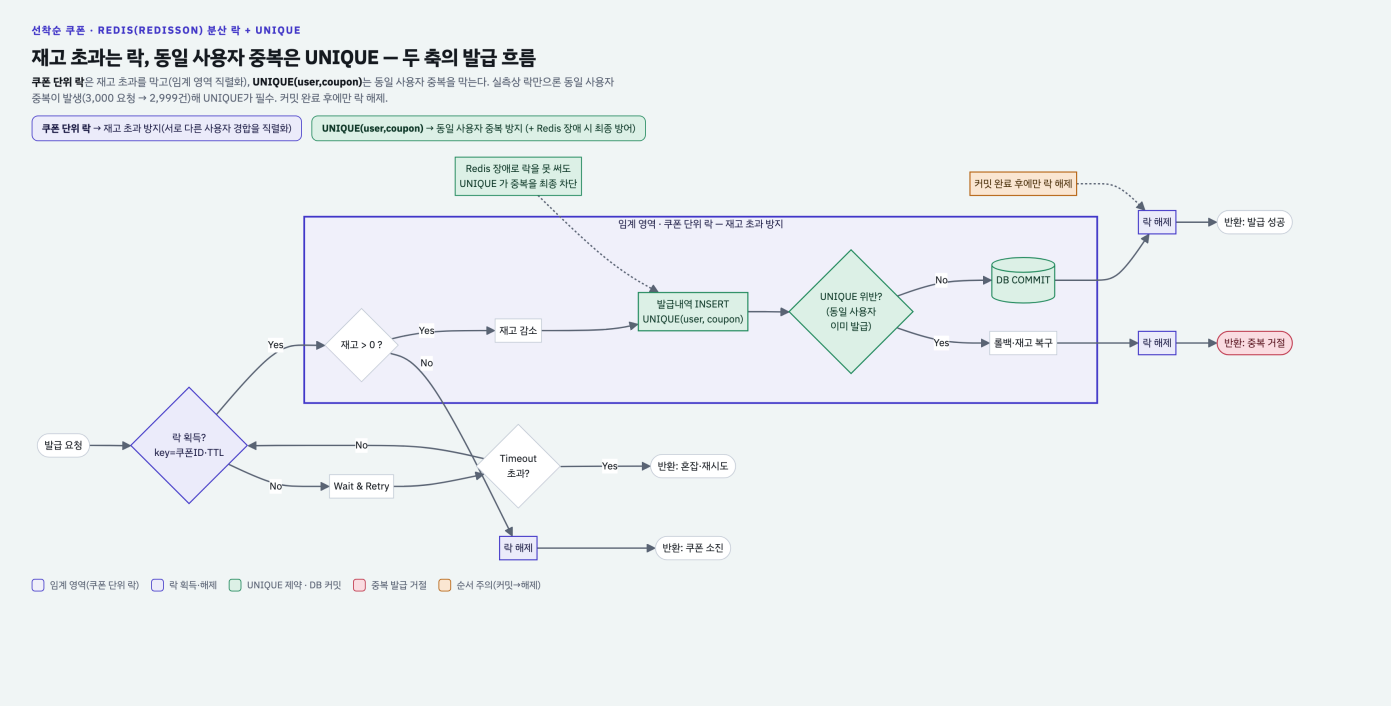


그림 18. 쿠폰ID별 Redisson 분산 락 안에서 **재고 검사 → 재고 감소 → 발급 내역 저장(UNIQUE user,coupon)**까지 한 임계 영역으로 보호하고, 발급 INSERT가 UNIQUE를 위반하면 @Transactional 롤백으로 재고까지 되돌린다. 락은 DB 커밋이 끝난 뒤에만 해제한다(커밋 전 해제 시 미반영 재고로 초과 발급). 재고 초과는 락이, 동일 사용자 중복은 UNIQUE가 담당한다.

다음 · 쿠폰 발급 결과를 실제 DB로 대조

5회 반복 결과와 정합성 확인

쿠폰 발급 결과를 실제 DB로 대조

쿠폰 동시성 · 공유 MySQL·Redis와 2개 인스턴스의 반복 검증

멀티 인스턴스 · 5회 반복 테스트 결과

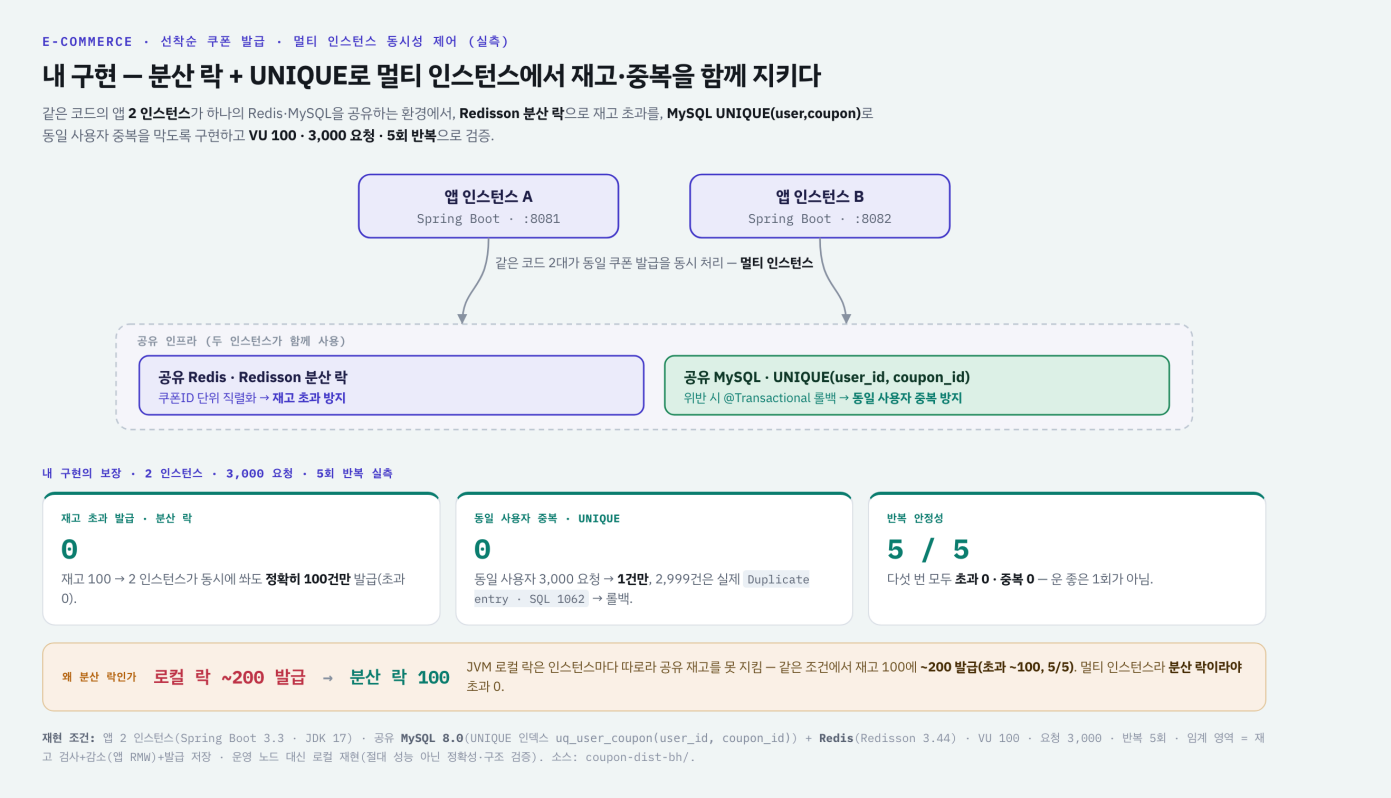


그림 19. 실제 MySQL·Redis(Redisson)를 연결한 **멀티 인스턴스 환경에서 5회 반복 테스트**를 수행해, **중복 발급·초과 발급 0건**을 확인했다.

검증 항목	예상 결과	실제 결과 / 근거
동일 사용자 재요청	발급 내역 1건	1건만 발급됨 (3,000 요청 중 2,999건 UNIQUE 위반 롤백) · DB COUNT=1 · 실제 SQL 1062 Duplicate entry
다수 사용자 경합	총 발급 ≤ 초기 재고	발급 100 = 재고 100 · 남은 재고 0 (2 인스턴스 ·5/5) · 로컬 락이면 ~200(초과 ~100)
개선 전 · 락 획득 실패 / Redis 장애	락 경합 timeout 은 RESOURCE_BUSY 로 거절 ·재고 미차감. 단, Redis 장애 시에는 락 획득 자체가 예외라 발급이 전부 실패(가용성 0). 락을 빼면 UNIQUE가 동일 사용자 중복은 막지만 재고 초과는 못 막음	Redis 실제 종료 주입 — 분산 락은 획득 불가라 락 거는 요청은 전부 실패(발급 0). 락 없이 발급해도 DB UNIQUE가 동일 사용자 중복은 차단(1,000→1건, 999 롤백)하나, 재고 초과는 못 막음(재고 100→1,000, 초과 900)

회고

분산 락은 쿠폰 축에 걸려 전역 발급 수량만 직렬화하므로 재고 초과를 막지만, '한 사용자 1장'은 사용자 축의 유일성이라 락의 보호 범위 밖이다. (락만 쓰면 동일 사용자 중복 2,999건을 실측). 그래서 **UNIQUE(user,coupon) 제약를 더해 중복을 0으로** 교정 — 재고 초과는 락, 동일 사용자 중복은 UNIQUE로 '축을 분리'했다. 또한 소규모라 Redis를 HA(Sentinel)로 못 띄워 분산 락이 단일 Redis SPOF가 되는 한계를 확인하고, **Redis 장애 감지 시 DB 조건부 원자 UPDATE(+UNIQUE)로 전환**하는 fallback을 구현해 Redis 다운·멀티 인스턴스에서도 초과 0·중복 0을 검증했다. — 평상시 분산 락 / 장애 시 DB가 정확성을 떠받치는 2단 구조

다음 · Redis 장애에서도 재고·중복을 보호

장애 시 DB 경로로 전환한 코드

Redis 장애에서도 재고·중복을 보호

쿠폰 동시성 · 조건부 UPDATE와 UNIQUE를 적용한 장애 경로

개선 코드 · Redis 장애 시 발급 경로 보완

제공 코드에서 정합성 보호와 장애 전환에 해당하는 부분만 발췌했다. 락 획득 재시도·해제 로직은 생략했다.

01 · 재고 보호
조건부 원자 차감

```
@Modifying(clearAutomatically = true)
@Query("update CouponInventory ci set ci.stockQuantity = ci.stockQuantity - 1, ci.updatedAt =
current_timestamp "
        + "where ci.id = :id and ci.stockQuantity > 0")
int deductStockAtomic(Long id);
```

재고가 0보다 클 때만 한 번에 차감한다. 영향 행 수 1은 성공, 0은 소진으로 판정하며, 이 UPDATE의 동시성 제어는 DB가 맡는다.

02 · 중복 보호
재고 차감과 발급 저장을 한 트랜잭션으로

```
// UserCouponService — Redis 장애 시 락 없이 발급(원자 차감 + UNIQUE)
@Transactional
public Long downloadAtomic(User user, CouponDownloadRequest request) {
    int affected = couponCommonService.deductCouponStockAtomic(request.getId());
    if (affected == 0) throw new CouponException(OUT_OF_STOCK);         // 재고 소진
    CouponInventory inventory = getCouponInventory(request);            // 차감 후 coupon FK 로드
    return downloadToCouponWallet(user, inventory.getCoupon());
}

@Transactional
protected Long downloadToCouponWallet(User customer, Coupon coupon) {
    UserCoupon userCoupon = UserCoupon.saveWallet(customer, coupon);
    userCouponRepository.saveAndFlush(userCoupon); // UNIQUE(user_id, coupon_id) 위반을 이 지점에서 즉시 표면화
→ 재고 차감까지 롤백
    return userCoupon.getId();
}
```

발급 저장을 saveAndFlush 로 하여 UNIQUE(user_id, coupon_id) 위반을 커밋까지 미루지 않고 그 지점에서 즉시 표면화한다. 위반 예외를 삼키지 않고 전파해 같은 트랜잭션의 원자 재고 차감까지 함께 롤백한다 — 중복 방어는 분산 락이 아니라 DB 제약이 맡으므로, 락 없는 degrade 경로에서도 그대로 유효하다.

03 · 장애 전환
Redis 장애 감지 후 DB 경로로 전환

```
// UserCouponFacadeService — 정상은 분산 락, RedisException(장애)이면 원자 UPDATE 경로로 degrade
try {
    return lockProcessor.tryLockWith(PREFIX + request.getId(), userCouponService::download, user,
request);
} catch (LockTimeoutException e) {              // 락 경합 timeout → degrade 아님
    throw new UserException(RESOURCE_BUSY);
} catch (DataIntegrityViolationException e) {   // 정상 경로 중복(UNIQUE 위반)
    throw new UserException(ALREADY_ISSUED_COUPON);
} catch (RedisException e) {                    // Redis 장애 감지 → DB 원자 UPDATE 경로로 전환
    return downloadAtomicFallback(user, request);
}
```

정상 시 분산 락(tryLockWith) 안에서 download 를 호출하고, Redis 장애로 RedisException 이 잡히면 락 없는 downloadAtomic 경로로 전환한다. 단순 락 경합은 과부하 신호(RESOURCE_BUSY)로 구분해, 'Redis 장애'일 때만 DB 경로로 떨어진다.

근거 coupon-concurrency-report.md

프로젝트 범위와 측정 조건

대량 데이터 조회의 비용을 실행 계획으로 확인

사용자가 조회하는 조건을 기준으로 쿼리와 인덱스를 재구성

진행 기간 2024.11 – 2025.01 **구성·역할** 총 8명 (PM 1 · 기획 1 · BE 3 · FE 3) · 백엔드 팀 리드

서비스와 맡은 범위

취업이 처음인 사용자를 위한 취업 관리 플랫폼입니다. 경험·스터디·면접·지원 현황·필수 자료를 관리하고 실시간 1:1 문의 채팅을 제공합니다.

맡은 범위: 아키텍처·DB 설계와 인프라·배포·모니터링 구축을 주도. 스터디·채팅·인증·내부 API를 구현하고, 스터디 목록 조회의 인덱스·쿼리 최적화 담당

데이터 규모

1,000만 건

해당 테이블·분포를 아래에 기재

조회 지연

API 왕복 p95 927ms → 106ms (약 8.7배↓)

API 전체 / DB 쿼리 구간을 구분

측정·검증 조건

로컬 재현 · MySQL 8.0 · 노드 2vCPU/5GB · study 1,000만행(매칭 250만/비매칭 700만, avg 676B) · offset 200,000·LIMIT 20(깊은 페이지) · warm · API 왕복 클라이언트 end-to-end N=30(p50/p95) + DB 구간 EXPLAIN ANALYZE(actual time·examined)

다음 · 실행 계획으로 조회 비용의 변화를 확인

쿼리 구조·실행 계획과 남은 한계

실행 계획으로 조회 비용의 변화를 확인

대량 조회 · 인덱스·쿼리 재구성과 같은 조건의 전후 비교

문제와 측정

study 목록을 **study_status='RECRUITING' AND deleted_at IS NULL ORDER BY id DESC**로 페이지네이션할 때 뒤 페이지(깊은 offset)로 갈수록 느려졌고, EXPLAIN ANALYZE로 보니 인덱스가 없어 클러스터 PK를 역방향으로 돌며 풀 로우를 800,080건 읽고(비매칭까지) Filter로 걸러 20건만 반환하는 구조였다.

즉, 병목은 정렬·오프셋 때문에 조건에 안 맞는 행까지 전부 풀 로우로 읽는 것이었고, 2vCPU/5GB 노드에서 이 구간이 API 응답의 대부분(p50 881ms 중 DB ~930ms)을 차지했다.

변경과 판단

조회 조건·정렬 **(study_status, deleted_at, id DESC)** 을 그대로 담는 커버링 인덱스를 만들고, 쿼리를 **id만 뽑는 서브쿼리(deferred join)**로 재구성해 정렬·필터·오프셋을 인덱스에서 처리하고 무거운 풀 로우 조회는 최종 20건만 하도록 바꿨다.

examined가 풀로우 80만 → id-only 20만으로 줄어 API 왕복 p50 881→83ms(≈10.6배), 대가로 **쓰기·저장 비용(커버링 인덱스 1개 유지)**이 늘지만 읽기 빈도가 훨씬 높은 목록 조회라 수용 가능하다고 판단했다.

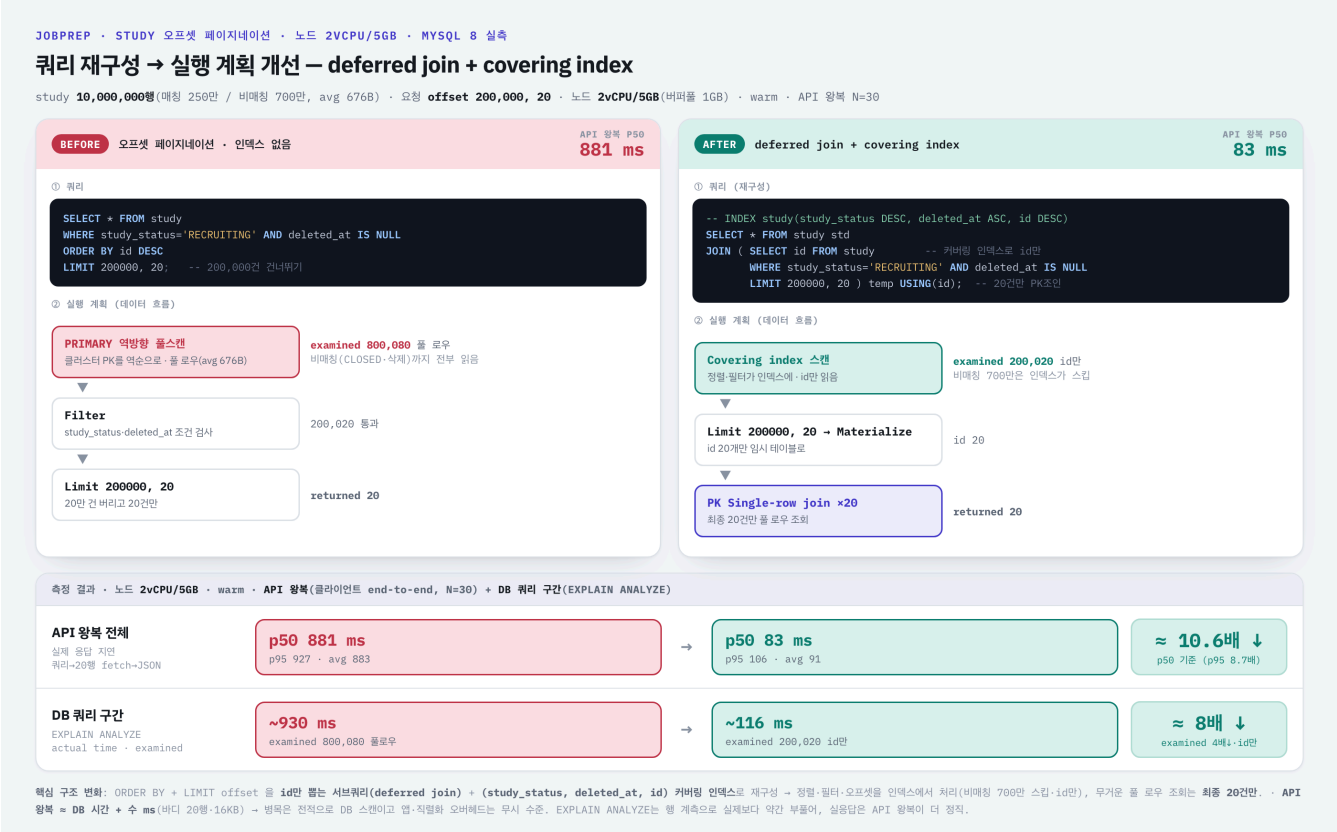


그림 20. BEFORE는 PRIMARY를 역순으로 돌며 풀 로우 800,080건을 읽어 거른다. AFTER는 커버링 인덱스로 매칭 id만 200,020건(비매칭 700만은 스킵) 읽고 최종 20건만 PK로 재조회한다 — 풀 로우 80만 → id 20만

확인한 결과와 한계

같은 warm(버퍼풀 적중) 조건에서, 깊은 페이지(offset 200,000) 는 API 왕복 p50 881ms → 83ms(≈10.6배), 얕은 페이지(offset 수천)는 건너뛸 행이 적어 개선 폭이 작다 — 효과는 **페이지 깊이와 조건 선택도에 비례**한다. 동시 사용자·처리량(TPS) 증가 효과는 단건 쿼리 측정 범위를 벗어나 추정으로만 본다.

근거 https://github.com/JOBPREP-BE/JOBPREP-BE · query-optimization-report.md

메시지 상태와 읽음 판정의 흐름

메시지를 보존하고 읽음 상태를 회복하는 채팅

전송 실패의 영향과 세션 캐시 누락을 서로 다른 경계에서 처리

사용자 문제와 규칙

전송이 실패해도 메시지가 저장에서 소실되지 않아야 하고, 상대가 읽었는지(읽음 상태)를 사용자에게 보여줘야 했다. 이를 위해 저장·전달·읽음의 제품 정의를 보완했다:
① **저장**: 보내는 순간 MongoDB에 먼저 확정(소실 금지)
② **전달**: 성공/실패를 상태로 구분 — WAITING(대기) → COMPLETE(완료), 실패분은 재전송 가능
③ **읽음**: readBy로 수신자별 읽음 표시, 입장·전송 시점에 갱신

직접 구현한 내용

메시지 저장과 후속 처리를 분리하고 전송 상태를 관리했습니다. 읽음은 Redis presence 캐시로 판단(입장 시 일괄 읽음·전송 시점에 상대 접속이면 즉시 읽음)하며, SEND 시점 recover로 유실 캐시를 보완해 신뢰성을 유지했습니다.

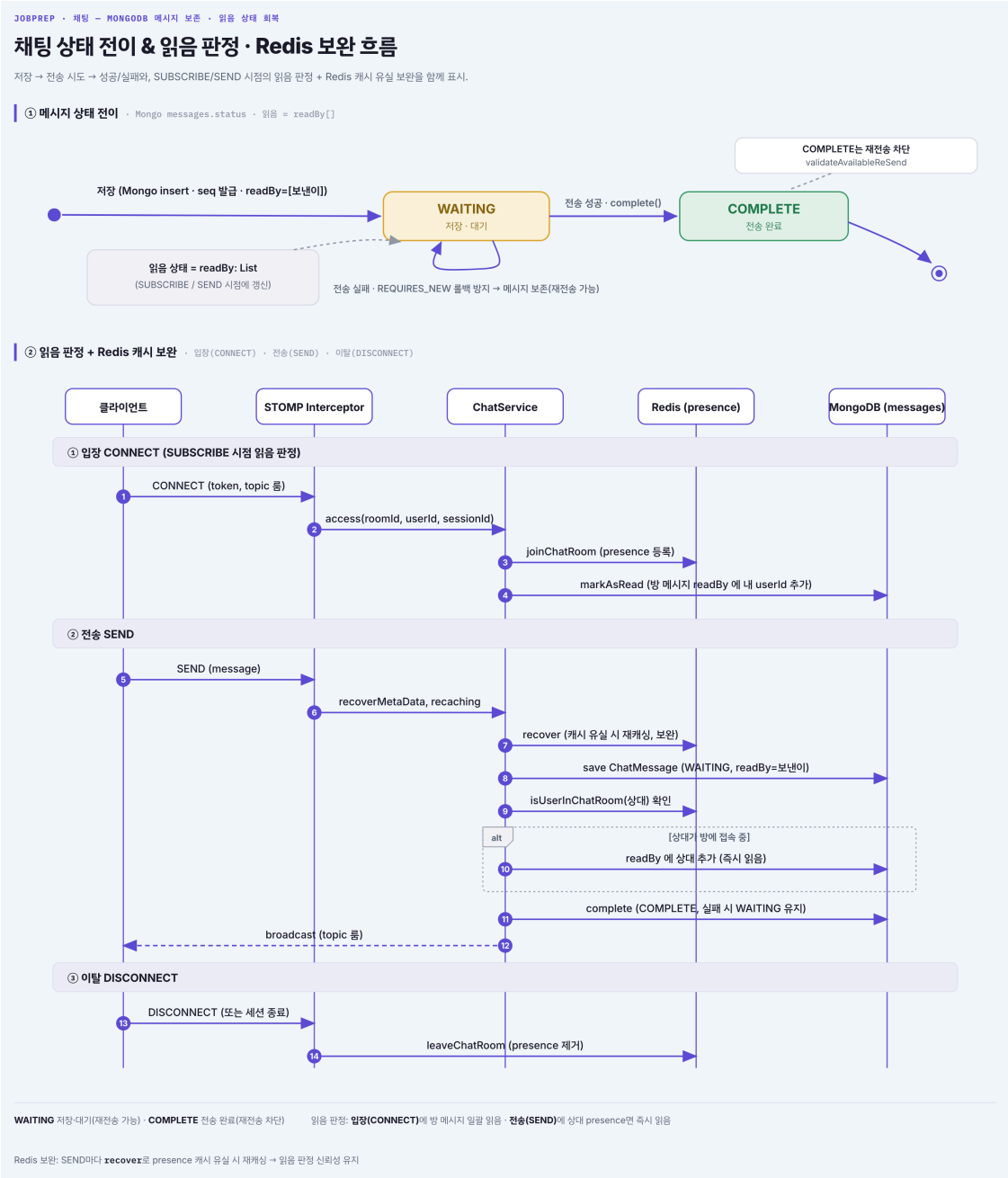


그림 21. 저장(WAITING)은 Mongo에 먼저 확정해 전송 실패와 무관하게 메시지를 보존하고, presence 캐시 유실로 어긋난 읽음 상태는 SEND의 recover·입장 시 일괄 읽음으로 회복한다 — 보존 경계와 회복 시점을 분리

다음 · 메시지 보존과 읽음 복구를 장애로 검증

장애 주입 결과와 남은 경계

메시지 보존과 읽음 복구를 장애로 검증

실시간 채팅 · 저장·전송·접속 상태의 검증 범위

장애 주입 검증과 추가 확인

검증 항목	조건·확인 방법	결과·근거
메시지 보존	저장 커밋 후 handleChatRoom(REQUIRES_NEW)의 완료 단계에 RuntimeException 100건 주입. 저장까지 함께 롤백하는 대조 구현 100건과 비교	**메시지 유실 0/100건** 대조 구현은 100/100건 유실. WAITING 잔존·COMPLETE 미전이, 전송 실패 반환 및 재전송 동작 확인.
읽음 판정 회복	Redis flushDb로 presence 캐시 소실 주입. 캐시 소실 → SEND 시점 recover() 실행 → 읽음 판정 결과 대조	**읽음 판정 실패 → recover() 후 복구** session:*·chat:room:* 재캐싱 후 isUserInChatRoom 정상 인식·상대 readBy 갱신 확인.
로직·동작 검증 재접속 / 다중 세션	presence는 세션 단위(chat:room set에 {userId}:{sid} 다건) → 한 세션 끊겨도 다른 세션 있으면 '접속 중' 유지, 모든 세션 종료 시에만 미접속. 재접속(CONNECT) 시 markAsRead로 누락 읽음 일괄 회복	다중 세션 presence 유지·재접속 시 markAsRead의 읽음 갱신 동작 검증·확인. 단 '접속=실제 열람'가정과 세션 TTL(1d) 만료 경로는 미검증

측정·검증 조건

로컬 재현 · Spring Boot + 실 MongoDB 7 · Redis 7(단일 프로세스) · 사용자 2명(보낸이·상대)·세션 1 · 메시지 보존 100건(완료 단계 실패 주입) + 읽음 회복 3-step
장애 주입: ① 완료/전송 단계 RuntimeException ② Redis presence flushDb(캐시 소실·재시작 모사)
확인 상태: Mongo 메시지 status(WAITING/COMPLETE)·readBy, Redis presence(chat:room:/session:), 읽음 판정 결과
표본: 단일 실행(보존 100 + 대조 naive 100, 읽음 회복 3단계)

남은 경계

presence(접속) ≠ 실제 열람: Redis presence는 방 '접속'만 의미하므로, 접속 중이지만 화면을 보지 않은 경우에도 즉시 읽음 처리될 수 있다.
후속 검증: 클라이언트 포커스/가시성(화면 활성) 신호를 읽음 조건에 추가하고, 재접속·세션 만료 시의 읽음 회복을 재현 테스트로 확인

근거 chat-resilience-report.md

배포·장애 감지 흐름

배포의 실패를 통제하고, 팀의 반복 작업을 줄이기

테스트·헬스체크를 통과한 인스턴스로 트래픽을 전환하고, 배포·장애 감지를 자동화하여 배포 중 서비스 중단 시간 0초를 확인

운영에서 해결한 문제

① **배포 시 중단**: 단일 인스턴스 재배포 때마다 생기던 다운타임을 NGINX Health-Check 기반 Blue-Green 무중단 배포로 제거했다.
② **실패한 배포의 노출**: 빌드·통합 테스트와 /actuator/health 검증을 통과해야만 트래픽을 전환하고 실패 시 기존 LIVE(BLUE)를 유지(롤백)해, 검증 안 된 불량 버전이 사용자에게 노출되지 않게 막았다.
③ **장애 원인 추적**: 모니터링(Prometheus·Grafana)과 Slack webhook 알림(Server Down·Job Failed·Error Log·리소스 사용량)으로 서버 다운·배포 실패를 실시간 감지·추적했다.
④ **팀의 반복 작업**: GitHub 트리거 기반 Jenkins CI/CD 파이프라인 + Google Cloud Secret Manager로 빌드·배포·시크릿 주입을 자동화해 수동 반복을 없앴다.

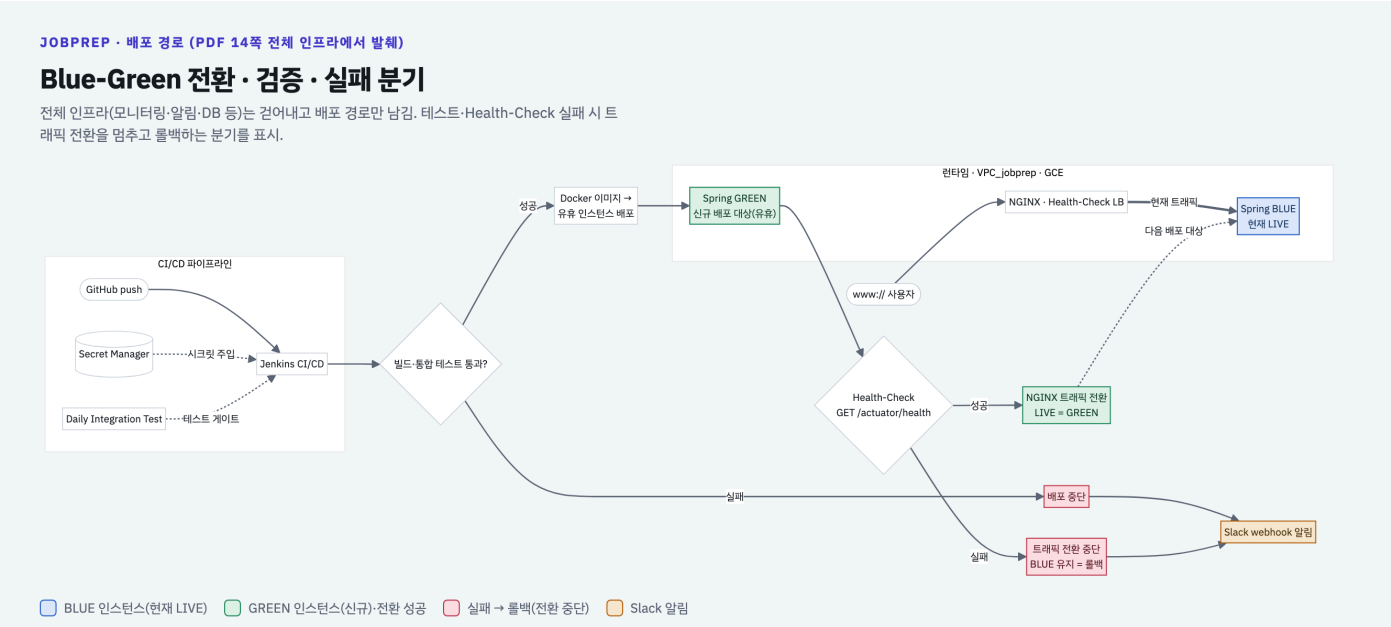


그림 22. 신규 인스턴스(GREEN)는 빌드·통합 테스트 → NGINX Health-Check(/actuator/health) 를 모두 통과해야만 NGINX가 트래픽을 그쪽으로 전환한다. 둘 중 하나라도 실패하면 전환하지 않고 기존 LIVE(BLUE)를 그대로 유지(무중단 롤백) 하며 Slack으로 알린다.

다음 · 배포 분기와 운영 도구의 실제 활용

검증한 동작과 남은 운영 과제

배포 분기와 운영 도구의 실제 활용

운영·협업 · 실패 차단·관측·반복 작업의 개선

직접 만든 것	검증 또는 실제 활용
Jenkins·Nginx 배포 자동화	① **준비 실패**: 유휴 인스턴스 배포 후 통합 테스트 또는 /actuator/health가 실패하도록 유도(잘못된 빌드·의존성) → NGINX 트래픽 전환이 차단되고 기존 LIVE 유지됨을 확인 ② **전환**: health 통과 시 트래픽을 신규(GREEN)로 전환 → 요청 유실·다운타임 0의 무중단 전환 확인 ③ **복귀(롤백)**: 전환 후 이상 감지 시 이전 색(BLUE)으로 즉시 되돌려 복구 → 수동 재배포 대비 복구 시간 단축 ④ **결과**: 재배포 다운타임 제거 + 검증 미통과 배포 자동 차단
MDC·로그·메트릭	① MDC로 요청 단위 상관관계 ID(trace/user/roomId/sessionId)를 로그에 주입 → 분산된 로그를 한 요청 흐름으로 추적 ② **증상별 추적 경로**: 서버 다운·배포 실패 → Slack webhook 알림(Server Down·Job Failed·리소스 사용량)으로 감지, 메시지 전송 실패·지연·읽음 누락 → 에러율·응답시간·커넥션 지표 + Error Log(MDC 상관관계) 로 원인 격리. ③ 모니터링(Prometheus·Grafana) 대시보드로 리소스 기반 임계 알림을 상시 운영 ※ CPU 사용률 > 80% (5분 지속) · 메모리(heap) 사용률 > 85% ※ 디스크 사용률 > 80% ※ HTTP 5xx 에러율 > 5% (5분) · p95 응답시간 > 1s ※ HikariCP 커넥션 풀 사용률 > 80%(active/max) — 커넥션 고갈 선제 감지 ※ actuator /health DOWN → Server Down 즉시 알림
리뷰 기준·스크립트 공유	**리뷰 기준(컨벤션·체크리스트) 문서화·공유** → PR 리뷰에 일관 적용 **더미 데이터 생성 스크립트 / DB 백업·복구 스크립트 공유** → 로컬·테스트 환경을 동일하게 재현하고 팀의 반복 세팅 작업을 제거

한계

Blue-Green 무중단은 HTTP 기준이라 WebSocket은 전환 시 재연결이 별도로 필요하고, @Scheduled가 인스턴스별 실행이라 중복·누락을 수동 backfill로 보정하는 상태 — Quartz(클러스터 스케줄러) 도입을 검토 중이다.

근거 배포: jenkins_cd_pipeline(build→이미지 태깅·push→Deploy to GCE, gcloud ssh) · Blue-Green 코드 ApplicationHealthIndicator(public IP로 BLUE/GREEN 식별 + /internal/health/server·포트 health) · 전체 흐름 문서 deploy-ops-flow.md