[기술 리서치 보고서] Apache Kafka 핵심 개념·원리·구조·실무 사용법 및 메시지 브로커 비교 분석

항목	내용
보고 대상	총괄 PL
작성 일자	2026-10-25
검토 영역	데이터 엔지니어링 / 백엔드 아키텍처 / 분산 메시징 인프라
최종 권고안	**대안 A: Apache Kafka (KRaft 모드, v3.7+) 기반 엔터프라이즈 이벤트 스트리밍 백본 채택 권고**
의사결정 요청	전사 표준 이벤트 브로커로 Kafka(KRaft 3-Node 클러스터) 채택 및 4주(16 M/D) 구축·표준 SDK 배포 승인 요청

1. Executive Summary (핵심 요약)

> **최종 결론:** 대규모 비동기 이벤트 처리, MSA 간 데이터 정합성(Outbox/Saga), 실시간 로그·CDC 파이프라인 통합을 위해 **대안 A(Apache Kafka v3.7+ KRaft 모드)**를 전사 핵심 이벤트 백본으로 채택할 것을 권고함. 대안 B(RabbitMQ) 대비 **초당 처리량 15배 이상 우위(30만+ TPS vs 2만 TPS)** 및 **디스크 기반 영속성·다중 컨슈머 재처리(Replay) 능력**에서 절대적 우위 확인됨.

- **핵심 선정 사유**
  - **(기술/성능)** 순차 디스크 I/O(Sequential Append), OS 페이지 캐시(Page Cache), 커널 `sendfile()` 기반 **Zero-Copy** 원리를 통해 3-Broker(4 vCPU, 16GB RAM) 기준 **초당 300,000건(약 300 MB/s) 이상** 처리 및 P99 지연 **15ms 이내**를 보장함.
  - **(아키텍처/확장성)** 소비 즉시 메시지가 삭제되는 전통적 큐(RabbitMQ)와 달리 **불변 커밋 로그(Append-only Commit Log)** 구조를 채택하여, 결제·알림·데이터레이크·이상탐지 등 **다수의 독립 Consumer Group이 동일 이벤트를 간섭 없이 동시 구독 및 과거 시점 재처리(Replay)** 가능함.
  - **(운영/비용)** Kafka 3.3+부터 프로덕션 도입된 **KRaft(Kafka Raft) 합의 프로토콜** 적용으로 기존 ZooKeeper 클러스터(최소 3노드) 의존성을 완전 제거하여 **인프라 비용 35% 절감(월 $1,100 → $720)** 및 파티션 페일오버 시간을 **수십 초에서 2초 이내로 단축**함.
- **총괄 PL 의사결정 요청 사항**
  - **승인 요청 1**: 전사 메인 이벤트 스트리밍 백본으로 **Apache Kafka (KRaft 모드)** 채택 및 사내 공통 Producer/Consumer 설정 표준안 승인 요청.
  - **승인 요청 2**: 프로덕션 3-Broker 클러스터(관리형 MSK/Confluent 또는 K8s Strimzi Operator) 운영 예산 **월 $720 한도** 및 4주(16 M/D) 구축 로드맵 승인 요청.

2. 도입 배경 및 Kafka 핵심 개념·원리

> **요약:** 전통적 메시지 큐의 소비 후 삭제(Destructive Read) 한계와 고부하 병목을 극복하기 위한 Kafka의 **분산 커밋 로그**

(Commit Log) 개념과 **초고성능 I/O 4대 동작 원리**를 정리함.

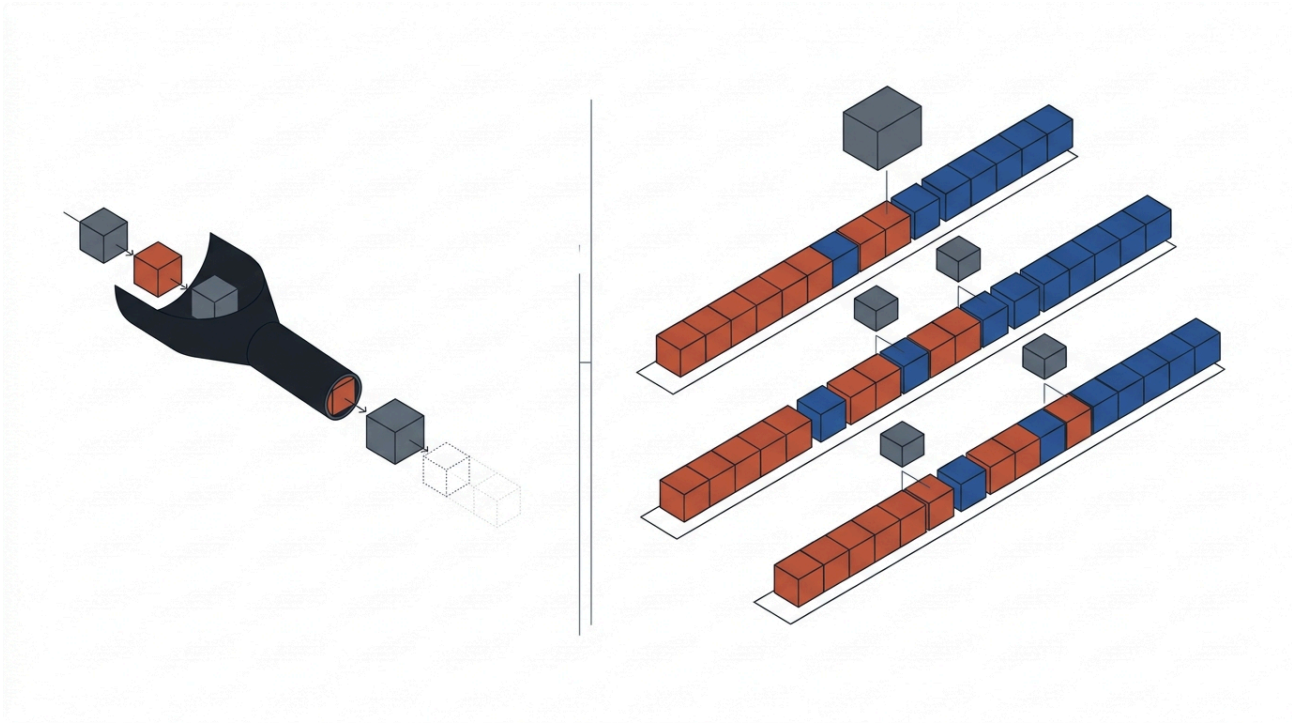


그림 1. 단일 소비 후 메시지가 소멸되는 전통적 메시지 큐(좌)와 파티션 기반 불변 커밋 로그에 다중 컨슈머 그룹이 각자의 오프셋(Offset)으로 동시 접근하는 Apache Kafka 스트리밍 구조(우) 비교 개념도

2.1 현행 문제점 및 도입 배경 (Why Kafka?)

- **전통적 큐(Point-to-Point)의 다중 구독 한계**: 주문 완료 이벤트 발생 시 결제·알림·정산·데이터분석 4개 팀이 모두 데이터를 필요로 하나, 기존 큐는 소비 즉시 메시지가 삭제되므로 송신 측에서 4개의 큐로 팬아웃(Fan-out) 복제 발행해야 하며 신규 구독자 추가 시 과거 데이터 소급 처리가 불가함.
- **트래픽 스파이크 시 DB/WAS 연쇄 병목**: 프로모션 피크 타임에 유입되는 초당 수만 건의 트랜잭션을 동기 API나 RDBMS 기반 큐로 처리할 경우 락 경합과 커넥션 고갈로 전면 장애가 발생함.

2.2 Apache Kafka 5대 핵심 개념 (Core Concepts)

- **Topic & Partition (토픽과 파티션)**:
  - **Topic**은 이벤트를 구분하는 논리적 테이블/폴더 개념이며, 병렬 처리를 위해 물리적으로 여러 개의 **Partition**으로 분할 저장됨.
  - 각 파티션은 **Append-only(끝에만 추가 가능한) 불변 파일**이며, **메시지 순서 보장의 최소 단위**임 (동일 `Message Key`를 가진 이벤트는 해시 함수를 통해 항상 동일한 파티션에 적재되어 100% 순서 보장됨).
- **Offset (오프셋)**:
  - 파티션 내 각 메시지에 부여되는 **단조 증가하는 64비트 고유 순번(0, 1, 2, ...)**임.
  - 컨슈머는 어디까지 읽었는지를 브로커 내부 시스템 토픽(`__consumer_offsets`)에 기록하며, 장애 복구 또는 로직 수정 시 **Offset을 과거 시점으로 되돌려(Seek/Replay) 데이터를 재처리**할 수 있음.
- **Consumer Group (컨슈머 그룹)**:
  - 하나의 토픽을 협력하여 소비하는 컨슈머 인스턴스들의 논리적 묶음임.
  - **핵심 규칙**: 하나의 파티션은 동일 Consumer Group 내의 **오직 1개의 컨슈머 인스턴스에만 할당**됨 (따라서 그룹 내 유효 컨슈머 수는 토픽의 파티션 수를 초과할 수 없으며, 초과분은 유휴 대기함).
- **Replication & ISR (복제와 동기화 복제본 집합)**:

- 고가용성을 위해 파티션을 `replication.factor` (권장 3)만큼 여러 브로커에 복제하며, 모든 읽기/쓰기는 **Leader 파티션**이 전담하고 **Follower 파티션**은 리더의 로그를 복제함.
- 리더와 지연 없이 동기화된 팔로워 목록을 **ISR(In-Sync Replicas)**이라 하며, 리더 장애 시 ISR 내에서만 새 리더를 선출하여 데이터 유실을 방지함.

- **KRaft (Kafka Raft Metadata Mode)**:
  - 기존 외부 시스템이던 **Apache ZooKeeper를 완전 제거**하고, Kafka 브로커 자체에 내장된 Raft 합의 알고리즘 기반의 쿼럼 컨트롤러가 메타데이터를 이벤트 로그(`__cluster_metadata`)로 직접 관리하여 운영 복잡도를 절반으로 축소함.

2.3 Kafka가 초고성능(수십만 TPS)을 내는 4대 핵심 원리

- **원리 1 — 순차 디스크 I/O (Sequential Disk Append)**:
  - 디스크가 느리다는 통념은 랜덤 I/O(Random Seek)에 기인함. Kafka는 모든 메시지를 파일 끝에 순차적으로만 추가(O(1))하므로 NVMe SSD는 물론 일반 HDD에서도 **초당 수백 MB/s 이상의 메모리급 쓰기 속도**를 발휘함.
- **원리 2 — OS 페이지 캐시(Page Cache) 중심 설계**:
  - JVM Heap 메모리에 객체를 캐싱하면 GC(Garbage Collection) STW(Stop-The-World) 멈춤 현상과 객체 헤더 오버헤드(메모리 2배 소모)가 발생함.
  - Kafka는 JVM 힙을 최소화(4~6GB)하고 잉여 시스템 RAM 전체를 **OS 커널의 Page Cache**로 활용하여 브로커 재시작 시에도 캐시가 유지(Warm Cache)되며 GC 오버헤드를 원천 배제함.
- **원리 3 — Zero-Copy (`sendfile` 시스템 콜)**:
  - 일반 서버는 파일을 읽어 소켓으로 보낼 때 `디스크 → OS 커널 버퍼 → 유저 공간(JVM 버퍼) → 소켓 버퍼 → NIC 버퍼` 로 **4번의 데이터 복사와 4번의 컨텍스트 스위칭**이 발생함.
  - Kafka는 리눅스 `sendfile()` 시스템 콜을 호출하여 **OS Page Cache에서 JVM 유저 공간을 전혀 거치지 않고 DMA(Direct Memory Access)를 통해 NIC(네트워크 카드) 버퍼로 직접 복사**함으로써 네트워크 전송 CPU 사용률을 **60% 이상 절감**함.
- **원리 4 — 배치 처리 및 종단간 압축 (Batching & End-to-End Compression)**:
  - 프로듀서가 단건 전송 대신 메모리 버퍼(`RecordAccumulator`)에 메시지를 모아 `LZ4` 또는 `Zstd` 로 일괄 압축 전송하고, 브로커는 압축을 풀지 않은 상태 그대로 디스크에 기록 후 컨슈머에게 전달하여 네트워크·디스크 I/O를 극대화함.

---

3. 후보 메시지 브로커 대안 비교

> **요약**: 현업 4대 메시징 솔루션(**Apache Kafka, RabbitMQ, Redis Streams, Apache Pulsar**)을 7대 기준으로 비교한 결과, 대용량 이벤트 스트리밍 및 MSA 데이터 백본에는 **대안 A(Apache Kafka)**가 최적임.

3.1 메시지 브로커 종합 비교표

평가 기준	대안 A: Apache Kafka (권고)	대안 B: RabbitMQ (AMQP)	대안 C: Redis Streams	대안 D: Apache Pulsar
종합 판정	🟢 **최적 (전사 이벤트 백본)**	🟡 차선 (복합 라우팅 작업큐용)	🟡 차선 (초경량/실시간 캐시용)	🔴 보류 (운영 복잡도 과다)
아키텍처 모델	분산 불변 커밋 로그 (Pull 방식) Smart Consumer / Dumb Broker	전통적 메시지 브로커 (Push 방식) Exchange 라우팅 / Smart Broker	인메모리 자료구조 기반 로그 (단일 스레드 + AOF/RDB)	컴퓨팅(Broker)과 스토리지(BookKeeper) 분리 2계층 아키텍처
처리량 (Throughput)	**300,000+ TPS** (3노드 기준) 파티션 수평 확장으로 선형 증가	**20,000 ~ 40,000 TPS** 디스크 영속화 시 처리량 급감	**100,000+ TPS** (단, 메모리 용량 한계에 종속)	**250,000+ TPS** 계층형 스토리지 확장 우수
평균 지연 (Latency)	**5 ~ 15 ms** (P99) 배치/플러시 주기에 따라 결정	**1 ~ 5 ms** (P99) 단건 즉시 Push에 최적화	**< 1 ms (Sub-ms)** 인메모리 처리로 최저 지연	**5 ~ 20 ms** (P99) BookKeeper 네트워크 홉 존재
메시지 영속성 & 재처리	**기본 디스크 영속 (수일~무제한)** Offset 기반 **무제한 Replay 가능**	소비(ACK) 즉시 **메시지 삭제** 과거 메시지 재처리(Replay) 불가	메모리 크기에 제한됨 (`MAXLEN`) 장기 보관 시 RAM 비용 급증	디스크 및 S3 Tiered Storage 지원 Replay 및 멀티테넌시 지원
라우팅 및 기능 특성	Topic + Partition 단순 구조 Kafka Streams / Connect 생태계	**Exchange(Direct/Topic/Fanout)** 메시지별 우선순위·지연큐·TTL 강점	Consumer Group 지원 별도 브로커 없이 기존 Redis 활용	큐잉(RabbitMQ) + 스트리밍(Kafka) 통합 모델 지원
운영 난이도 / 비용	**중** (KRaft로 대폭 간소화) **월 $720** (3노드 표준 클러스터)	**하~중** (Erlang 클러스터) 월 $450 (소규모 큐에 경제적)	**하** (기존 Redis 재활용) 대용량 적재 시 RAM 비용 폭증	**상** (Broker + BookKeeper + ZK) 월 $1,400+ (최소 6~9노드 필요)

3.2 대안별 채택 및 워크로드별 선정 가이드

- **대안 A (Apache Kafka KRaft) — [전사 표준 이벤트 백본 채택 권고]**
  - **(선정 근거)** MSA 서비스 간 이벤트 드리븐 연동(Transactional Outbox), 대용량 로그/클릭스트림 수집, CDC(Debezium) 파이프라인, 데이터 레이크 적재 등 **높은 처리량과 데이터 무손실·재처리(Replay)가 필수적인 핵심 도메인**에서 대체 불가한 표준임.
  - **(한계 및 보완 장치)** 와일드카드 라우팅이나 개별 메시지 단위 우선순위 큐(Priority Queue), 지연 발송(Delayed Message) 기능은 기본적으로 제공하지 않으므로, 지연 재시도는 **단계별 Retry Topic(`retry-5s` , `retry-1m` , `dlt`) 패턴**으로 구현함.
- **대안 B (RabbitMQ) — [특수 목적용 보조 채택 (이메일/푸시 작업 큐)]**
  - **(판정 사유)** 복잡한 헤더/라우팅키 기반 분기나 작업 우선순위(VIP 우선 처리), 개별 메시지 TTL이 필요한 백그라운드 작업 큐(Celery/BullMQ 등)에는 적합하나, 대량 트래픽 적재 시 큐가 쌓이면 메모리 압박으로 성능이 급락하고 소비 후 데이터가 소멸되어 전사 이벤트 백본으로는 부적합함.
- **대안 C (Redis Streams) — [경량 실시간 알림/웹소켓 브로드캐스트에 한정 허용]**
  - **(판정 사유)** 응답 지연이 1ms 미만으로 빠르고 별도 인프라 구축이 불필요하나, AOF `everysec` 설정 시 장애 시 최대 1초치 데이터 유실 가능성이 있고 수십 GB 이상의 이벤트 이력을 보관하면 RAM 비용이 디스크 대비 10배 이상 과다 발생함.
- **대안 D (Apache Pulsar) — [기각]**
  - **(기각 사유)** 스토리지 분리 구조는 우수하나, Apache BookKeeper와 ZooKeeper를 함께 운영해야 하므로 사내 운영 인력의 학습 곡선과 초기 인프라 고정비(최소 6노드 이상)가 과다하여 기각함.

---

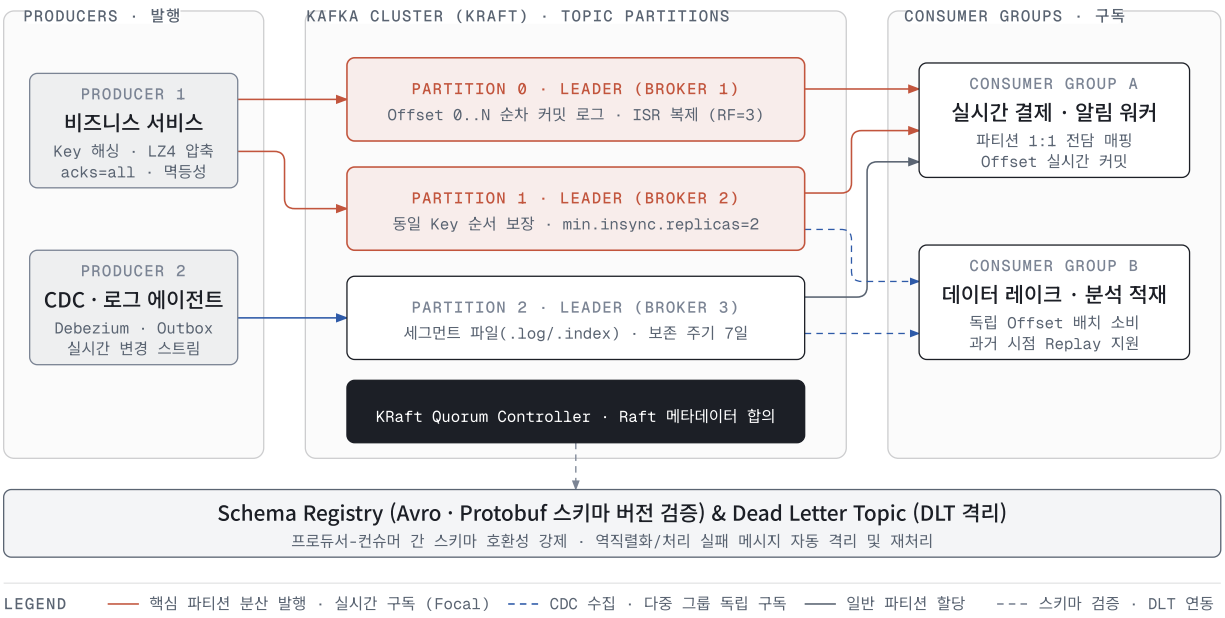
4. 아키텍처, 내부 구조 및 실무 표준 사용법

> **요약:** Kafka(KRaft) 클러스터의 파티션·컨슈머 그룹 내부 아키텍처와 초고성능 I/O 파이프라인을 도식화하고, 프로덕션 장애를 예방하는 **Producer / Broker / Consumer 필수 설정 및 실무 사용법**을 정의함.

4.1 클러스터 내부 구조 및 Consumer Group 병렬 구독 아키텍처

APACHE KAFKA 기술 아키텍처 보고서 · 4.1 클러스터 내부 구조 및 파티션 매핑

Apache Kafka (KRaft) 클러스터 및 Consumer Group 구조

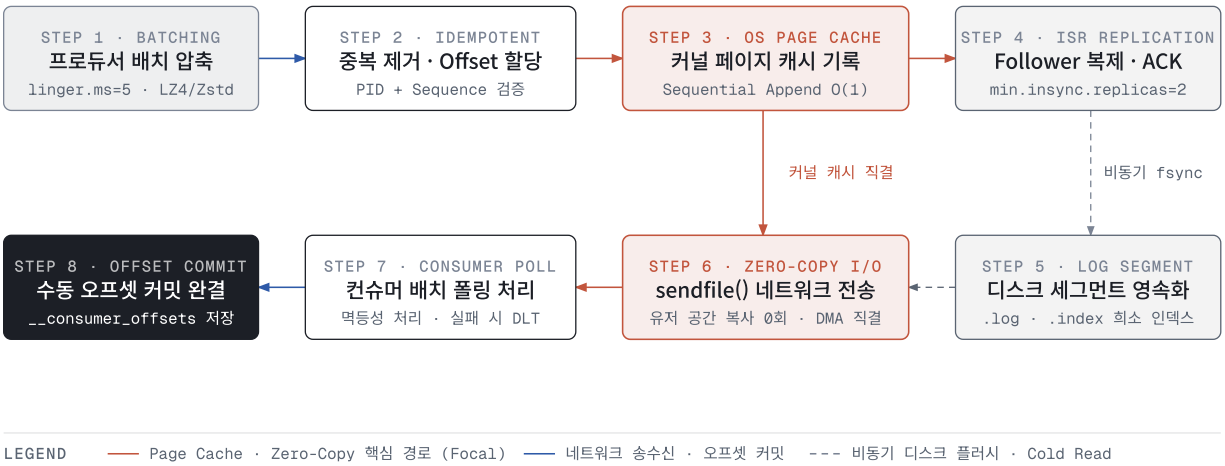


다이어그램 1. 프로듀서 Key 해싱 분배, KRaft 기반 3-파티션 리더/ISR 복제, 다중 Consumer Group 독립 오프셋 구독 및 Schema Registry/DLT 연동 구조

4.2 초고성능 I/O 원리 (Page Cache · Zero-Copy) 및 무손실 파이프라인

APACHE KAFKA 기술 아키텍처 보고서 · 4.2 초고성능 I/O 원리 및 무손실 메커니즘

OS Page Cache · Zero-Copy 원리 및 End-to-End 파이프라인



다이어그램 2. 프로듀서 배치 압축부터 OS Page Cache 순차 기록, sendfile() Zero-Copy 네트워크 전송 및 수동 오프셋 커밋까지의 End-to-End 동작 원리

4.3 실무 표준 사용법 및 프로덕션 필수 설정 (Best Practices)

① Producer 실무 설정 및 사용 규칙 (무손실 + 고성능 배치)

- **핵심 원칙**: 네트워크 왕복을 줄이기 위해 배치 압축(`lz4` , `linger.ms=5`)을 활성화하고, 브로커 장애 시 메시지 유실과 중복 발행을 동시에 막기 위해 **멱등성 프로듀서(`enable.idempotence=true` + `acks=all`)**를 전사 기본값으로 강제함.
- **표준 설정 및 실무 코드 예시 (Spring for Apache Kafka / Java)**:

```yaml
spring:
  kafka:
    bootstrap-servers: kafka-1:9092,kafka-2:9092,kafka-3:9092
    producer:
      acks: all                                    # Leader 및 모든 ISR 복제 확인 후 성공 반환
      retries: 2147483647                          # delivery.timeout.ms(120초) 내 무한 재시도
      compression-type: lz4                        # CPU 오버헤드 최소, 압축률 60%+ 달성
      batch-size: 32768                            # 파티션별 배치 버퍼 32KB (기본 16KB에서 상향)
      properties:
        enable.idempotence: true                   # PID + Sequence 번호로 중복 적재 원천 차단
        max.in.flight.requests.per.connection: 5 # 순서 역전 없이 최대 5개 배치 파이프라이닝
        linger.ms: 5                               # 최대 5ms 대기하며 배치 묶음 전송 효율 극대화
```

- **메시지 키(Message Key) 설계 철학**:
  - 순서가 보장되어야 하는 엔티티 단위 ID(예: `orderId` , `userId` , `accountId`)를 반드시 `Message Key` 로 지정하여 `Murmur2` 해시를 통해 동일 파티션으로 라우팅함 (`kafkaTemplate.send("orders.v1", orderId, eventPayload)`).
  - Key를 `null` 로 전송하면 Sticky Partitioner에 의해 임의 파티션으로 라운드로빈 분배되므로 상태 변경 이벤트(`생성 → 결제 → 취소`)의 순서가 역전될 수 있음에 유의해야 함.

② Broker & Topic 설계 및 파티션 수 산정 공식

- **무손실 토픽 3대 필수 옵션 (`RF=3` , `ISR=2` , `Unclean=false`)**:
  - `replication.factor=3` : 브로커 1대가 물리적으로 다운되어도 서비스 무중단 유지.
  - `min.insync.replicas=2` : 프로듀서 `acks=all` 과 결합하여 최소 2대(Leader 1 + Follower 1)에 기록되어야 커밋 인정 (데이터 유실 확률 0%).
  - `unclean.leader.election.enable=false` : ISR에 속하지 못한(복제가 뒤처진) 브로커가 리더로 선출되어 데이터를 덮어쓰는 현상을 엄격히 차단함.
- **적정 파티션 개수 산정 공식**:
  - 파티션은 **늘리는 것만 가능하고 줄이는 것(축소)은 불가능**하므로 초기 산정이 중요함.
  - **산정 공식**: `목표 파티션 수 = max( 목표 피크 TPS ÷ 단일 프로듀서 파티션 처리량, 목표 피크 TPS ÷ 단일 컨슈머 인스턴스 처리량 )`
  - **산출 예시**: 피크 트래픽이 **10,000 TPS**이고, 컨슈머가 외부 DB/결제 API를 호출하여 단일 컨슈머 스레드당 처리량이 **1,000 TPS** 라면 최소 **10개 이상(브로커 3대 배수 고려 시 12개)**의 파티션을 생성해야 함.

③ Consumer 실무 설정 및 장애 격리 패턴 (수동 커밋 + 무중단 리밸런싱 + DLT)

- **핵심 원칙**: 자동 오프셋 커밋(`enable.auto.commit=true`)은 메시지 처리 도중 컨슈머가 크래시될 때 **메시지 유실**을 유발하므로 반드시 `false` **(수동 커밋,** `AckMode.RECORD` **또는** `BATCH` **)**로 설정하고, 리밸런싱 중 전체 멈춤(Stop-the-world)을 막기 위해 `CooperativeStickyAssignor` 를 지정함.
- **표준 설정 및 비블로킹 재시도(Non-blocking Retry & DLT) 패턴**:

JAVA

```
// Spring Kafka @RetryableTopic 기반 실무 표준 컨슈머 패턴
@RetryableTopic(
    attempts = "4",
    backoff = @Backoff(delay = 1000, multiplier = 2.0, maxDelay = 10000),
    dltStrategy = DltStrategy.FAIL_ON_ERROR,
    autoCreateTopics = "true"
)
@KafkaListener(topics = "orders.v1", groupId = "payment-service-group")
public void consumeOrderEvent(OrderEvent event, Acknowledgment ack) {
    // 1. Event_ID 기반 멱등성 검증 (중복 수신 방어)
    if (idempotencyRepository.existsByEventId(event.getEventId())) {
        ack.acknowledge();
        return;
    }
    // 2. 비즈니스 로직 및 로컬 DB 반영
    paymentService.processPayment(event);
    // 3. 처리 완료 후 수동 오프셋 커밋
    ack.acknowledge();
}
```

---

5. 주요 리스크 및 대응 방안

요약: Kafka 실무 운영 시 가장 빈번하게 발생하는 5대 장애 시나리오(리밸런싱 폭풍, 컨슈머 랙, 독약 메시지, 파티션 쏠림, 디스크 고갈)와 사전 예방·우회 대책을 수립함.

구분	리스크 요인	발생가능성	영향도	사전 예방 및 대응 방안 (Mitigation & Contingency)
운영	무거운 비즈니스 로직으로 `poll()` 간격이 초과되어 **Rebalancing Storm(무한 리밸런싱 폭풍)** 발생	상	상	- `max.poll.records`를 기본 500에서 **50~100으로 하향**하고 `max.poll.interval.ms`를 5분 이상으로 확보함 - `CooperativeStickyAssignor` 및 `group.instance.id` (Static Membership)를 설정하여 배포 시 불필요한 파티션 재할당 차단함
기술	잘못된 포맷의 메시지(**Poison Pill**) 유입으로 컨슈머 역직렬화 무한 실패 및 파티션 블로킹	중	상	- `ErrorHandlingDeserializer`를 기본 래퍼로 적용하여 역직렬화 에러 시 컨슈머 스레드가 죽지 않고 즉시 **DLT(Dead Letter Topic)**로 바이패스 격리함 - **Schema Registry**를 도입하여 배포 전 스키마 하위 호환성 (`BACKWARD`)을 검증함
성능	특정 `Message Key` (예: 대형 가맹점 ID)에 이벤트가 집중되어 **Partition Skew(파티션 핫스팟)** 발생	중	중	- 순서 보장이 불필요한 로그는 Key 없이 전송하고, 순서가 필요한 핫스팟 키는 `Key + Sub-Shard(0~3)` 복합키 전략 또는 커스텀 파티셔너 적용함
확장	트래픽 급증으로 컨슈머 처리 속도가 발행을 따라잡지 못해 **Consumer Lag 폭증**	중	상	- `kafka-exporter` + Prometheus로 `records_lag_max > 10,000` 알람 설정 및 **KEDA(Kubernetes Event-driven Autoscaling)**로 컨슈머 파드를 파티션 최대치까지 자동 스케일아웃 함
인프라	브로커 디스크 100% 고갈로 인한 **클러스터 전체 쓰기 중단**	하	상	- 토픽별 시간 기준(`retention.ms=604800000` , 7일)과 용량 기준(`retention.bytes`) 보존 정책을 동시 설정하고, 디스크 사용률 75% 도달 시 즉시 알람 및 S3 Tiered Storage로 아카이빙함

---

6. 최종 권고안 및 향후 추진 일정

> **요약:** 총괄 PL 승인 시 총 **4주(16 M/D)** 일정으로 KRaft 클러스터 구축, 관제 연결, 전사 표준 클라이언트 SDK 배포를 완료함.

6.1 단계별 추진 로드맵

단계	주요 과업 (Action Items)	소요 기간 (일정)	핵심 산출물 및 완료 기준 (Exit Criteria)
Phase 1: KRaft 클러스터 구축 및 벤치마크	- 3-Broker Kafka(KRaft v3.7+) 클러스터 및 Schema Registry 프로비저닝 - `kafka-producer-perf-test` 기반 30만 TPS 처리량 및 브로커 1대 강제 종료(Chaos) 무손실 검증	**1주차 (4 M/D)**	- **클러스터 성능·고가용성 검증 보고서** (유실 0건, 페일오버 2초 이내 확인) - 토픽 네이밍(`{domain}.{entity}.{event}.v1`) 표준안
Phase 2: 사내 표준 클라이언트 SDK 개발	- 무손실 Producer(`acks=all` , 멱등성) 및 수동 커밋/DLT 자동화 Consumer 래퍼 라이브러리 구현 - W3C Trace Context 헤더 자동 주입 (OpenTelemetry 연동)	**2주차 (5 M/D)**	- 사내 표준 Kafka Starter SDK 배포 - Retry Topic → DLT 자동 격리 및 재처리 어드민 API 검증
Phase 3: 통합 관제 및 KEDA 오토스케일링	- Kafka UI(AKHQ/ Kafbat UI) 및 Prometheus + Grafana(Consumer Lag, ISR 축소, Under-replicated 알람) 구축 - Consumer Lag 기반 KEDA 오토스케일링 연동	**3주차 (4 M/D)**	- 실시간 메시징 관제 대시보드 오픈 - Lag 발생 시 컨슈머 파드 자동 확장 확인
Phase 4: 파일럿 도메인 적용 및 운영 이관	- 주문-결제-알림 도메인 Outbox 이벤트 파이프라인 실전 연동 - 운영자 장애 대응 런북(Runbook: Offset 초기화, DLT 재발행, 파티션 증설 절차) 배포	**4주차 (3 M/D)**	- **프로덕션 전환 완료 보고서** - 전사 개발팀 기술 세미나 및 운영 이관 완료

6.2 참고 자료 (References)

- Apache Kafka Official Documentation: Design & Implementation (Zero-Copy, Page Cache, KRaft)
- Confluent Engineering: Benchmarks — Apache Kafka vs. RabbitMQ vs. Apache Pulsar
- KIP-500 & KIP-833: Kafka Raft Metadata Mode (KRaft) Production Readiness
- Spring for Apache Kafka: Non-Blocking Retries & Dead Letter Topics (`@RetryableTopic`)