`ARCHITECTURE DESIGN SPECIFICATION (SAD) · ISO/IEC/IEEE 42010 & 4+1 VIEW STANDARD`

[아키텍처 설계 명세서] Spring Boot · React 풀스택 다계층 캐시 레이어 아키텍처 설계서 (SAD)

문서 관리 항목	명세 내용	문서 관리 항목	명세 내용
문서 번호 (Doc ID)	`SAD-CACHE-2026-001`	**버전 (Version)**	`v1.0.0`
대상 시스템 / 범위	전사 React SPA + Spring Boot 백엔드 통합 캐시 계층	**문서 상태 (Status)**	`APPROVED (표준 규격 확정)`
설계 책임 (Architect)	플랫폼 아키텍처 & 프론트엔드/백엔드 코어 TF	**준거 표준 (Standard)**	`ISO/IEC/IEEE 42010 · 4+1 View · RFC 9111 (HTTP Caching)`
런타임 기준 스택	`React 18 (TanStack Query v5) + Spring Boot 3.3 + Caffeine 3.1 + Redis 7.2`	**보안 등급**	`대외비 (Internal Engineering Standard)`

문서 제·개정 이력 (Revision History)

버전	일자	변경 구분	작성 / 승인자	주요 설계 제·개정 내역
`v0.5.0`	2026-10-16	초안 작성	코어 아키텍처 파트	L0(Browser) ~ L3(Redis) 다계층 캐시 토폴로지 및 표준 캐시 키 네이밍 규격 초안 정의
`v0.9.0`	2026-10-22	검토 반영	백엔드 & FE 플랫폼 파트	`TieredCacheManager`, Redis Pub/Sub L1 동기화, SingleFlight 스탬피드 방어, TanStack Query 규격 반영
`v1.0.0`	2026-10-26	표준 승인	아키텍처 심의 위원회	4대 아키텍처 뷰(논리·프로세스·데이터·배포) 및 ArchUnit/ESLint 캐시 적합성 규칙 최종 확정

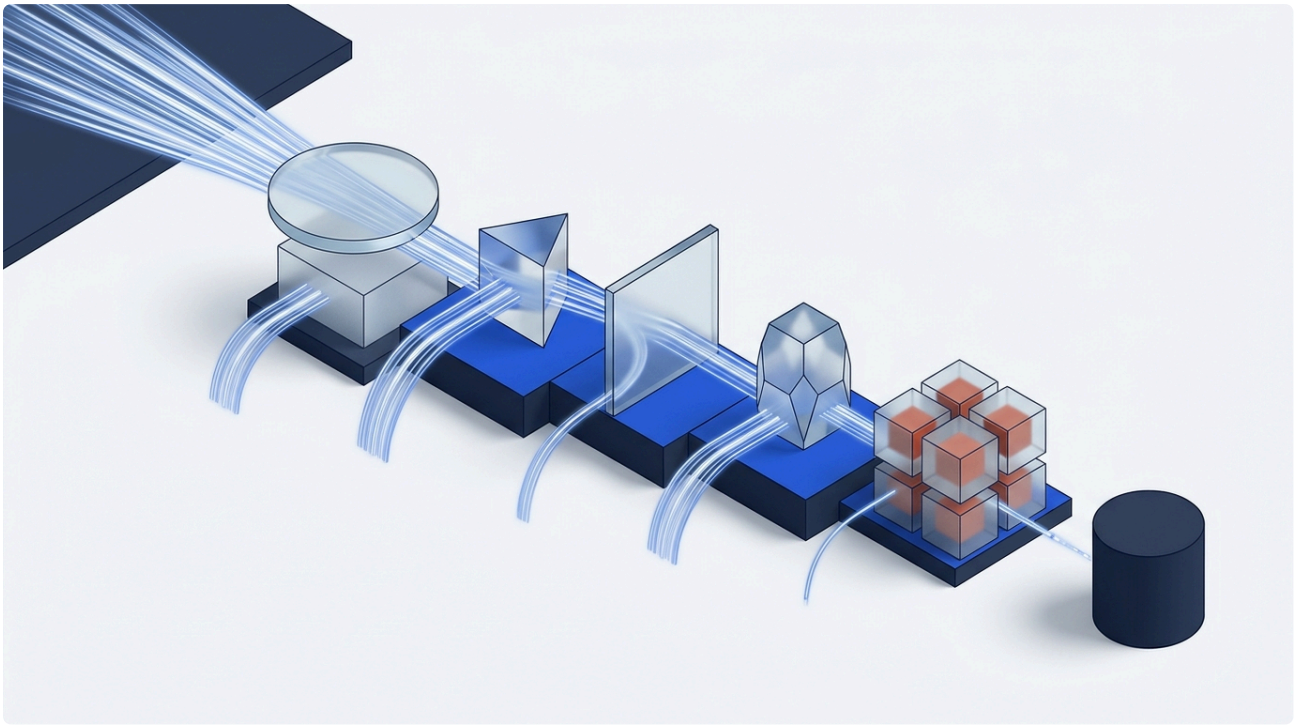
제1장. 설계 개요 및 아키텍처 원칙 (1. Architecture Overview & Principles)

> **[설계 핵심 정의 (Architecture Blueprint Summary)]**: 본 설계서는 **React 18 프론트엔드(`L0 Browser Memory Cache`) → CDN 및 HTTP 프로토콜(`L1 Edge/ETag Cache`) → Spring Boot 로컬 힙(`L2 Caffeine In-Memory Cache`) → 분산 메모리 그리드(`L3 Redis 7.2 Cluster`)**로 이어지는 **풀스택 4단계(End-to-End Multi-Tier) 캐시 레이어 참조 아키텍처**와 실시간 정합성 무효화(Invalidation) 및 장애 격리 메커니즘을 명세함.

1.1 시스템 컨텍스트 및 풀스택 다계층 캐시 흡수 구조 (System Context View)

- **시스템 경계 및 단계별 트래픽 필터링 메커니즘**:

- 사용자 브라우저에서 발생하는 전체 조회 요청의 **60~70%는 React `TanStack Query` 인메모리 캐시(`L0`)와 CDN/HTTP `304 Not Modified` (`L1`) 계층에서 네트워크 비용 없이 즉시 흡수**됨.
- 백엔드 WAS로 진입한 요청의 **90% 이상은 JVM 내부 `Caffeine` (`L2` , 응답 `≤ 0.5ms`) 및 `Redis Cluster` (`L3` , 응답 `≤ 3ms`)에서 처리**되어, 최종 RDBMS까지는 전체 트래픽의 **3~5% 미만**만 도달하도록 데이터베이스 부하를 원천 차단함.



React 브라우저부터 CDN, Spring Boot L1 Caffeine, L2 Redis Cluster까지 이어지는 풀스택 다계층 캐시 필터링 개념도

1.2 캐시 아키텍처 핵심 설계 원칙 명세 (Architecture Principles Specification)

원칙 ID	아키텍처 설계 원칙명	상세 설계 규격 및 강제 사항 (Design Rule Specification)	검증 방법 (Fitness Check)
`AP-01`	**엔티티 직접 캐싱 금지 및 불변 DTO 전용 캐싱 (Immutable DTO Only)**	• JPA `@Entity` 객체는 프록시 초기화 (`LazyInitializationException`), 양방향 순환 참조, 더티 체킹 오염을 유발하므로 캐시 저장을 전면 금지함 • 오직 직렬화 안전성이 보장된 **불변** `Java Record` **또는 전용** `CacheDTO` 만 캐싱 대상으로 허용함	ArchUnit `CACHE-RULE-01`
`AP-02`	**트랜잭션 커밋 이후 캐시 무효화 강제 (Post-Commit Invalidation)**	• `@Transactional` 내부에서 직접 Redis 키를 삭제/갱신하면 DB 롤백 시 데이터 불일치(Phantom Cache)가 발생함 • 모든 캐시 무효화는 반드시 `@TransactionalEventListener(phase = AFTER_COMMIT)` 또는 Outbox 릴레이를 통해 물리 커밋 완료 직후에 실행함	ArchUnit `CACHE-RULE-02`
`AP-03`	**L1(로컬) + L2(분산) 계층형 조합 및 Pub/Sub 동기화 (Tiered Sync)**	• 초고빈도 조회 데이터는 `Caffeine(L1)` + `Redis(L2)` 2단 조합(`TieredCacheManager`)으로 구성하여 네트워크 I/O 병목을 제거함 • 데이터 변경 시 `Redis Pub/Sub` 토픽 (`cache:evict`)을 브로드캐스트하여 **클러스터 내 전 WAS 파드의 L1 로컬 캐시를 10ms 이내에 동시 무효화**함	통합 테스트 / PubSub 지연 계측
`AP-04`	**3대 캐시 이상 현상 (Stampede·Avalanche·Penetration) 원천 방어**	• **Stampede(동시 재계산 폭주)**: 핫 키 만료 시 단일 스레드만 DB를 조회하는 `SingleFlight(분산/로컬락)` 강제 • **Avalanche(대량 동시 만료)**: 모든 TTL에 **±15% 랜덤 Jitter** 필수 부여 • **Penetration(미존재 키 공격)**: `BloomFilter` 또는 `Null Object 단기 캐싱(TTL 30s)` 적용	부하 테스트 (k6 / Gatling)
`AP-05`	**캐시 장애 시 Graceful Degradation (Fail-Open Fallback)**	• Redis 클러스터 순단 또는 타임아웃(`> 200ms`) 발생 시 전체 서비스가 마비되지 않도록 `Resilience4j CircuitBreaker` 가 즉시 작동함 • 서킷 Open 시 예외를 삼키고 **L1 로컬 캐시 또는 Read Replica DB(Rate Limit 적용)로 투명하게 Fallback** 처리함	Chaos Redis Kill 테스트

1.3 비기능 품질 속성 및 목표 SLA/SLO 명세 (Quality Attribute Specification)

요구사항 ID	품질 속성 분류	정량 목표 지표 (Target SLA / SLO)	아키텍처 실현 메커니즘 (Architectural Tactics)
`NFR-01`	**계층별 응답 지연 (Latency)**	• React L0 Hit: **0ms** / HTTP 304 Hit: **≤ 15ms** • Spring L1(Caffeine) Hit: **P99 ≤ 0.8ms** • Spring L2(Redis) Hit: **P99 ≤ 4.5ms**	• 브라우저 `staleTime` 메모리 즉시 반환 및 `ShallowEtagHeaderFilter` 페이로드 생략 • JVM Heap 내부 `W-TinyLFU` 참조 및 Lettuce Netty 논블로킹 커넥션 풀링
`NFR-02`	**캐시 적중률 (Hit Ratio)**	• 풀스택 통합 캐시 적중률 **≥ 92.0%** • 마스터/카테고리/공통코드 L1 적중률 **≥ 99.0%**	• 파드 기동 시 `ApplicationReadyEvent` 기반 핵심 도메인 캐시 선적재(Warm-up) • React Router Hover 시점 `queryClient.prefetchQuery` 선제적 프리페칭
`NFR-03`	**정합성 전파 지연 (Consistency)**	• 백엔드 L2 Evict + 전 노드 L1 Pub/Sub 무효화: **P99 ≤ 20ms** • React 클라이언트 화면 Stale 갱신: **≤ 500ms**	• `AFTER_COMMIT` 이벤트 기반 Redis `UNLINK` + `PUBLISH` 파이프라이닝 • Mutation `onSuccess` 즉시 무효화 + 타 사용자 변경분 SSE(`Server-Sent Events`) 푸시
`NFR-04`	**DB 부하 절감율 (DB Offloading)**	• 피크 타임(10,000 TPS) 기준 RDBMS 쿼리 유입률 **≤ 5% (500 QPS 이하)** • 핫 키 만료 시 DB 동시 쿼리 수 **노드당 최대 1건**	• `Caffeine.refreshAfterWrite` 비동기 백그라운드 갱신 및 `SingleFlight` 락 • 대용량 목록 조회 시 HTTP `stale-while-revalidate` 헤더 결합
`NFR-05`	**가용성 및 장애 격리 (Resilience)**	• Redis 전체 장애 시에도 조회 API 가용성 **≥ 99.95% 유지** • 캐시 타임아웃에 의한 WAS 스레드 고갈 **0건**	• Lettuce 커맨드 타임아웃 `200ms` 칼차단 + `SafeTieredCache` 예외 흡수 래퍼 • Redis 장애 시 L1 Caffeine 만료 시간을 일시 연장(Grace Mode)하여 DB 폭주 방어

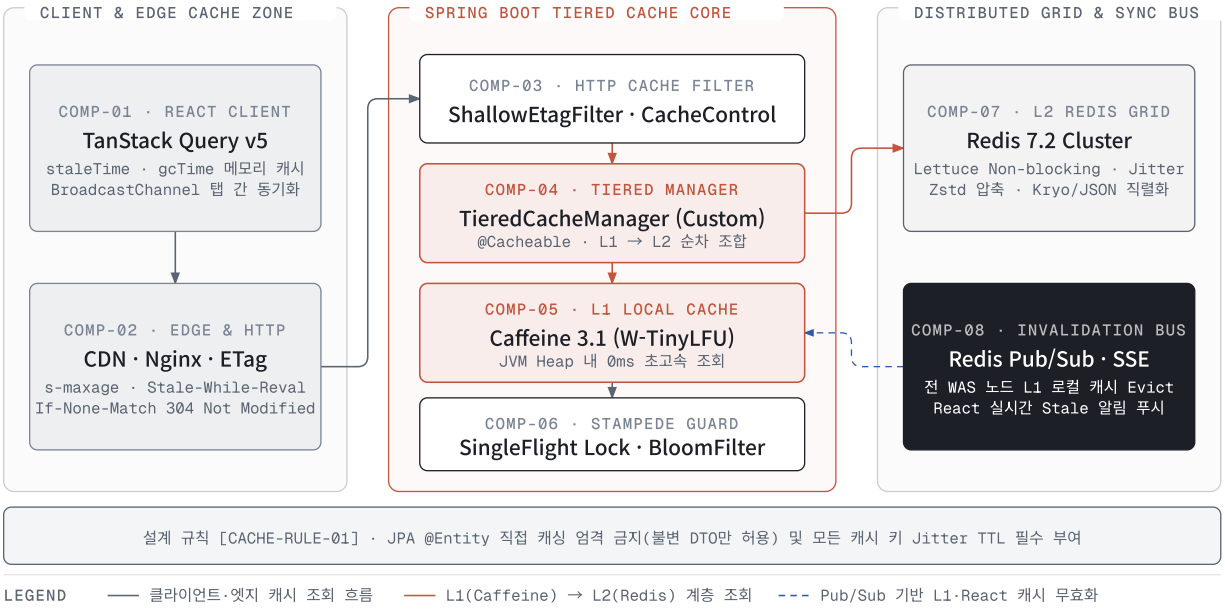
제2장. 논리 아키텍처 및 컴포넌트 설계 (2. Logical & Component View)

2.1 React · CDN · Spring Boot(L1/L2) 5계층 캐시 논리 아키텍처 구조도

- **구조 명세**: 전체 캐시 아키텍처는 데이터가 위치하는 물리적 거리에 따라 `Client & Edge Cache Zone (React/CDN/HTTP)` → `Spring Boot Tiered Cache Core (L1 Caffeine & SingleFlight)` → `Distributed Grid & Sync Bus (L2 Redis Cluster & Pub/Sub)` 의 3대 영역·8개 핵심 컴포넌트(`COMP-01` ~ `COMP-08`)로 구성됨.

VIEW 01 · LOGICAL & COMPONENT ARCHITECTURE (5-TIER FULL-STACK CACHE HIERARCHY)

[논리 뷰] React · Spring Boot 5계층 캐시 레이어 컴포넌트 구조도



View 1 논리 뷰 — React · Spring Boot 5계층 캐시 레이어 컴포넌트 구조도

2.2 컴포넌트 카탈로그 및 책임 명세표 (Component Specification Catalog)

컴포넌트 ID	컴포넌트명	소속 계층 / 패키지	핵심 설계 책임 (Core Responsibilities)	입출력 인터
`COMP-01`	**React Client Cache (`TanStack Query v5`)**	`Frontend (Browser Memory / IndexedDB)`	• API 응답 상태의 인메모리 캐싱 (`staleTime` , `gcTime`), 동일 쿼리 키 중복 요청 병합(Request Deduplication) • `useMutation` 성공 시 낙관적 업데이트(Optimistic UI) 및 관련 `QueryKey` 즉시 무효화	• 키 규격: `'detail` • 탭 동기화 우저 다중
`COMP-02`	**Edge CDN & HTTP Cache Gateway**	`Perimeter Edge (CloudFront / Nginx)`	• 해시된 정적 번들 (`/assets/*.js` , `*.css`) 1년 불변(`immutable`) 캐싱 • 공통 조회 API에 대해 `Cache-Control: public, s-maxage=30, stale-while-revalidate=60` 엣지 캐싱 수행	• 조건부 요 백엔드 `ET` 시 응답
`COMP-03`	**HTTP ETag & CacheHeader Filter**	`adapter.in.web.filter` (`Spring Web`)	• 응답 본문(JSON Byte)의 MD5/Murmur3 해시를 계산하여 `ETag` 응답 헤더 자동 주입 (`ShallowEtagHeaderFilter`) • 도메인별 어노테이션 또는 인터셉터 기반으로 표준 `Cache-Control` 헤더 부여	• 적용 대상 (`GET`) A • 효과: 모 송량 최대
`COMP-04`	**Tiered Cache Manager (`TieredCacheManager`)**	`common.cache.core` (`Spring Cache SPI`)	• Spring 표준 `CacheManager` 및 `Cache` 인터페이스를 커스텀 구현하여 **단일 `@Cacheable` 선언만으로 `L1(Caffeine)` → `L2(Redis)` 순차 조회 및 역채움 (Backfill)** 수행 • Redis 장애 시 예외를 포획하고 L1 단독 모드로 자동 전환 (`SafeFallback`)	• 구현: `org.spr` • 설정: 도 L1/L2 활성
`COMP-05`	**L1 Local Cache (`Caffeine 3.1`)**	`common.cache.l1` (`JVM Heap Memory`)	• `W-TinyLFU` 알고리즘 기반 최고 적중률 로컬 인메모리 캐시 운영 (`maximumSize` 및 `expireAfterWrite` 강제) • 직렬화/역직렬화 및 네트워크 비용이 **0ms**이므로 초당 수만 건 호출되는 핫 키를 1차 방어함	• 제약: 파. `maximum` 장 • 통계: `r` 적중률 노출
`COMP-06`	**Stampede & Penetration Guard**	`common.cache.guard` (`SingleFlight / Bloom`)	• **SingleFlight**: 캐시 미스 발생 시 `ConcurrentHashMap<Key,`	• 타임아웃 캐시 반환

컴포넌트 ID	컴포넌트명	소속 계층 / 패키지	핵심 설계 책임 (Core Responsibilities)	입출력 인터
			`CompletableFuture>` 및 Redisson 분산 락을 결합하여 단 1개의 스레드만 DB 조회 수행 • **Penetration Guard**: 존재하지 않는 ID 조회 시 `NullValue` 마커를 30초간 캐싱	• 효과: 핫 100% 차
`COMP-07`	**L2 Distributed Cache (`Redis 7.2 Cluster`)**	`adapter.out.cache` (`Lettuce / Redisson`)	• 다중 WAS 파드가 공유하는 단일 진실 캐시 그리드 운영 (`3 Master + 3 Replica`) • 기본 TTL에 **랜덤 Jitter(±15%)** 를 자동 가산하여 저장하며, 대형 JSON(`> 10KB`)은 `Zstd` 압축 저장	• 클라이언 킹 파이프 • 명령어 지 `DEL` (대형
`COMP-08`	**Cache Invalidation Bus (`Pub/Sub + SSE`)**	`common.cache.sync` (`RedisMessageListener`)	• 트랜잭션 커밋 직후 (`AFTER_COMMIT`) L2 Redis 키를 `UNLINK` 하고 `cache:invalidation` 채널로 무효화 메시지 발행 • 전 WAS 파드가 이를 수신해 로컬 `Caffeine(L1)` 에서 즉시 제거하며, 필요 시 SSE로 React 클라이언트에 Stale 이벤트 푸시	• 메시지 포 `key, ver` • 자기 자신

2.3 백엔드(Spring Boot) 및 프론트엔드(React) 캐시 모듈 표준 패키지 구조

```TEXT
[1] Spring Boot 백엔드 캐시 레이어 표준 패키지 구조
com.company.platform
├── common.cache
│   ├── config
│   │   ├── CacheSpec.java                  # [정책 정의] 캐시 이름별 L1/L2 TTL, MaxSize, Jitter 비율
│   │   ├── TieredCacheConfig.java          # [COMP-04] Caffeine(L1) + Redis(L2) + CircuitBrea
│   │   └── HttpCacheHeaderConfig.java      # [COMP-03] ShallowEtagHeaderFilter 및 Cache-Contro
│   ├── core
│   │   ├── TieredCacheManager.java         # [COMP-04] Spring CacheManager 커스텀 구현체
│   │   ├── TieredCache.java                # [COMP-04] L1 조회 -> L2 조회(L1 역채움) -> SingleFli
│   │   └── JitterTtlGenerator.java         # [COMP-07] Base TTL ± 15% 랜덤 Jitter 계산기
│   ├── guard
│   │   ├── SingleFlightExecutor.java       # [COMP-06] 동일 키 동시 요청 병합 (Local Future + Dist
│   │   └── BloomFilterGuard.java           # [COMP-06] 미존재 ID 캐시 관통(Penetration) 방어 필터
│   └── sync
│       ├── CacheEvictEvent.java            # [COMP-08] 도메인 상태 변경 시 발행되는 내부/PubSub 이벤트
│       ├── PostCommitCacheEvictor.java     # [COMP-08] @TransactionalEventListener(AFTER_COMM
│       └── RedisCacheEvictSubscriber.java # [COMP-08] 전 WAS 파드 L1 Caffeine 동시 무효화 리스너
└── domain.product
    ├── application.dto
    │   └── ProductCacheDto.java            # [AP-01] 캐시 저장 전용 불변 Java Record (JPA @Entity
    └── application.service
        └── ProductQueryService.java        # @Cacheable(cacheNames = "product:detail", key =

[2] React 프론트엔드 캐시 레이어 표준 디렉토리 구조 (TanStack Query v5)
src/
├── app/providers/
│   └── QueryProvider.tsx                   # [COMP-01] QueryClient 전역 설정 (staleTime, gcTime
├── shared/api/
│   ├── httpClient.ts                       # Axios/Fetch 인스턴스 (ETag / If-None-Match 조건부 요
│   └── sseCacheSync.ts                     # [COMP-08] 백엔드 SSE 캐시 무효화 이벤트 수신 -> queryCl
└── entities/product/api/
    ├── productKeys.ts                      # [Query Key Factory] 계층형 쿼리 키 중앙 관리 (오타 및 누
    ├── useProductDetailQuery.ts            # 조회 훅 (staleTime: 60_000, placeholderData, prefe
    └── useUpdateProductMutation.ts         # 변경 훅 (onMutate 낙관적 업데이트 + onSuccess 캐시 무효화
```

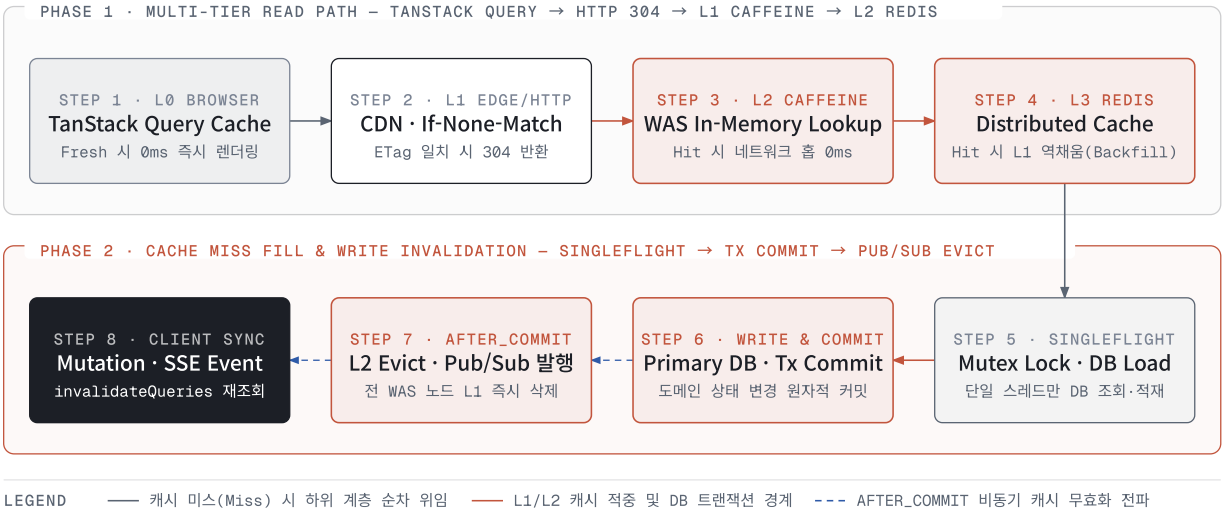
제3장. 프로세스 및 런타임 파이프라인 설계 (3. Process & Runtime Sequence View)

3.1 풀스택 캐시 조회(Read Look-aside) 및 쓰기 무효화(Write Invalidation) 8단계 시퀀스

- **동적 실행 흐름**:
  - **Phase 1 (Read Path: `STEP 1` ~ `STEP 5`)**: React 컴포넌트가 데이터를 요청하면 `L0(TanStack Query)` → `L1(CDN/HTTP 304)` → `L2(WAS Caffeine)` → `L3(Redis Cluster)` 순서로 캐시를 확인하며, 전 계층 Miss 시에만 `STEP 5(SingleFlight Lock)` 를 통해 단 1개의 스레드가 DB를 조회하여 상위 캐시를 역순으로 채움(Backfill).
  - **Phase 2 (Write & Invalidation Path: `STEP 6` ~ `STEP 8`)**: 상태 변경(`POST` / `PUT` / `DELETE`) 요청은 DB 트랜잭션이 **물리적으로 커밋된 직후(`AFTER_COMMIT`)**에만 L2 Redis 키를 삭제(`UNLINK`)하고 `Redis Pub/Sub` 으로 전 WAS의 L1 캐시를 즉시 비우며, 클라이언트 측 TanStack Query 캐시를 동기화함.

VIEW 02 · PROCESS & RUNTIME CACHE SEQUENCE VIEW (READ & WRITE PIPELINE)

[프로세스 뷰] 풀스택 다계층 캐시 조회 및 실시간 무효화 8단계 파이프라인



View 2 프로세스 뷰 — 풀스택 다계층 캐시 조회 및 실시간 무효화 8단계 파이프라인

3.2 단계별 런타임 캐시 처리 규격 및 타임아웃·예외 계약 명세표

단계 ID	단계명 (Stage)	처리 컴포넌트	입력 계약 (Key / Header)	런타임 처리 규칙 및 타
`STEP 1`	**L0 Browser Cache**	React `TanStack Query v5`	`QueryKey` (`['product','detail',id]`)	• 캐시 엔트리가 `sta` 경우 네트워크 요청을 즉시 반환 • `staleTime` 경과 링하면서 백그라운드
`STEP 2`	**L1 Edge & HTTP 304**	CDN + `ShallowEtagFilter`	`GET /api/v1/...` + `If-None-Match: "W/xyz"`	• 공개 API의 경우 CD 에서 즉시 응답 • 백엔드 도달 시 응답 `If-None-Match` 와 송 생략
`STEP 3`	**L2 Local In-Memory**	`COMP-05` (`Caffeine Cache`)	`{cacheName}::{key}`	• WAS JVM 힙 내 `C` `O(1)` 조회 (네트워크 • `NullValue` 마커 DB 조회 없이 즉시 빈 환
`STEP 4`	**L3 Distributed Grid**	`COMP-07` (`Redis 7.2 Cluster`)	`{env}:{svc}:v1:{cacheName}:{key}`	• Lettuce 비동기 커 웃 **`200ms` 엄격 제한** • **L2 Hit 시**: 역직렬화 **컬 `Caffeine(L1)`** 후속 호출 가속
`STEP 5`	**SingleFlight DB Load**	`COMP-06` (`SingleFlight`) + Read DB	Cache Miss Key + Mutex Lock	• 동일 키에 대해 수백 더라도 **`Completabl` 락**으로 단 1개 스레드만 회 • 조회 결과에 **Base T**

단계 ID	단계명 (Stage)	처리 컴포넌트	입력 계약 (Key / Header)	런타임 처리 규칙 및 티
				적용하여 L2 및 L1에 `TTL 30s` 로 Null 마
`STEP 6`	**Atomic DB Write**	Command Service + Primary DB	`@Transactional` 상태 변경 명령	• 비즈니스 불변식 검증 사항 반영 • 트랜잭션 내부에서는 않고 `ApplicationE` `CacheEvictEvent`
`STEP 7`	**Post-Commit Evict & Pub/Sub**	`COMP-08` (`PostCommitCacheEvictor`)	`CacheEvictEvent(cacheName, key)`	• `@TransactionalE = AFTER_COMMIT)` • ① Redis `UNLINK` ② `PUBLISH cache` 클러스터 전 WAS 노드
`STEP 8`	**Client Cache Sync**	React `useMutation` / `SSE Listener`	Mutation Response / SSE Event	• **요청 당사자**: `onSu` `queryClient.inv` `queryKey })` 또는 갱신 • **타 접속 클라이언트**: `queryKey` 를 Stale 시 재조회 수행

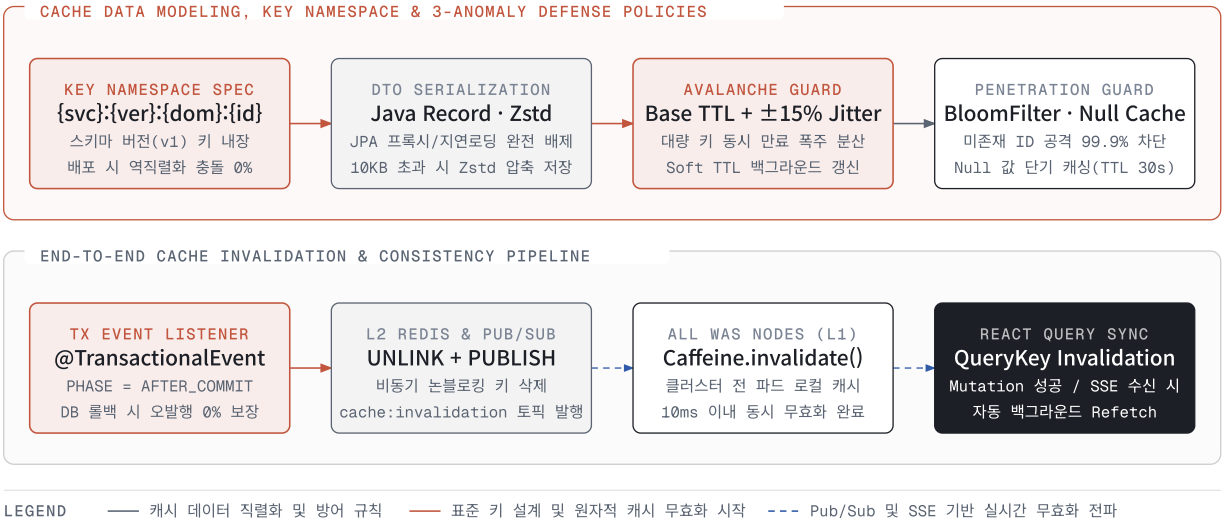
제4장. 데이터 및 영속성 아키텍처 설계 (4. Data & Persistence View)

4.1 캐시 키 스키마·직렬화·TTL Jitter 설계 및 전 계층 정합성 동기화 구조도

- **캐시 데이터 뷰 명세**: 다계층 캐시 환경에서 가장 치명적인 장애 원인인 **배포 시 클래스 스키마 호환성 깨짐(Serialization Conflict)**, **동시 만료 폭주(Cache Avalanche)**, **WAS 노드 간 로컬 캐시 불일치(Stale L1 Cache)**를 해결하기 위한 표준 데이터 스키마 및 동기화 파이프라인을 명세함.

VIEW 03 · CACHE KEY SCHEMA, SERIALIZATION, JITTER TTL & CONSISTENCY ARCHITECTURE

[데이터 뷰] 캐시 키·직렬화·TTL 정책 및 전 계층 정합성 동기화 구조도



View 3 데이터 뷰 — 캐시 키·직렬화·TTL 정책 및 전 계층 정합성 동기화 구조도

4.2 도메인별 다계층 캐시 정책 및 TTL·크기·무효화 규격 명세표 (Cache Policy Matrix)

정책 ID	대상 데이터 도메인 (예시)	캐시 계층 구성	표준 캐시 키 패턴 (Namespace Spec)	계층별 TTL 및 Jitter / 크기 규격	무효화(정합성
`CACHE-01`	**마스터 / 공통 코드 / 카테고리** (변경 극소, 초고빈도 조회)	`L0(React)` + `L1(CDN)` + `L2(Caffeine)` + `L3(Redis)`	`{env}:{svc}:v1:common:code:{groupCode}`	• **L0(`staleTime`)**: `30분` • **L1(`s-maxage`)**: `10분` • **L2(Caffeine)**: `TTL 10분` , `Max 5,000건` • **L3(Redis)**: `TTL 6시간 (±15% Jitter)`	• 관리자 정 시 A Redis `Pub/S` 초기화
`CACHE-02`	**상품 상세 / 콘텐츠 본문** (읽기:쓰기 = 95:5, 트래픽 집중)	`L0(React)` + `L2(Caffeine)` + `L3(Redis)` + `SingleFlight`	`{env}:{svc}:v1:product:detail:{productId}`	• **L0(`staleTime`)**: `1분` (`gcTime: 10분`) • **L2(Caffeine)**: `TTL 15초` , `Max 10,000건` • **L3(Redis)**: `TTL 30분 (±15% Jitter)`	• 가격/ `AFTER` 효화 • L2(C TTL(15 안전장
`CACHE-03`	**개인화 프로필 / 권한 (RBAC)** (사용자별 격리, 보안 민감)	`L0(React)` + `L2(Caffeine)` + `L3(Redis)` *(CDN 공용 캐싱 절대 금지)*	`{env}:{svc}:v1:user:perm:{userId}`	• **HTTP 헤더**: `Cache-Control: private, no-store` • **L2(Caffeine)**: `TTL 10초` , `Max 20,000건` • **L3(Redis)**: `TTL 15분 (±10% Jitter)`	• 권한 시 즉시 `{user` `query` 호출
`CACHE-04`	**복합 검색 / 필터링 목록** (조합 다양, 페이로드 큼)	`L0(React)` + `HTTP ETag` + `L3(Redis Zstd 압축)`	`{env}:{svc}:v1:search:list:{sha256(params)}`	• **L0(`staleTime`)**: `30초` , `keepPreviousData` • **L2(Caffeine)**: 미사용 (힙 메모리 오염 방지) • **L3(Redis)**: `TTL 3분 (±15% Jitter)`	• 쿼리 SHA-2 • 10KB `Zstd` 트워크
`CACHE-05`	**집계 대시보드 / 랭킹 뷰** (고비용 연산, 최종 일관성 허용)	`L0(React)` + `L2(Caffeine Refresh)` + `L3(Redis SortedSet)`	`{env}:{svc}:v1:stats:ranking:{period}`	• **L0(`staleTime`)**: `1분` • **L2(Caffeine)**: `refreshAfterWrite=30s` , `expire=2m` • **L3(Redis)**: `TTL 10분` (배치/스트림 선적재)	• 사용자 집계 쿼 백그라 (`Writ` `Refre` 적 갱신

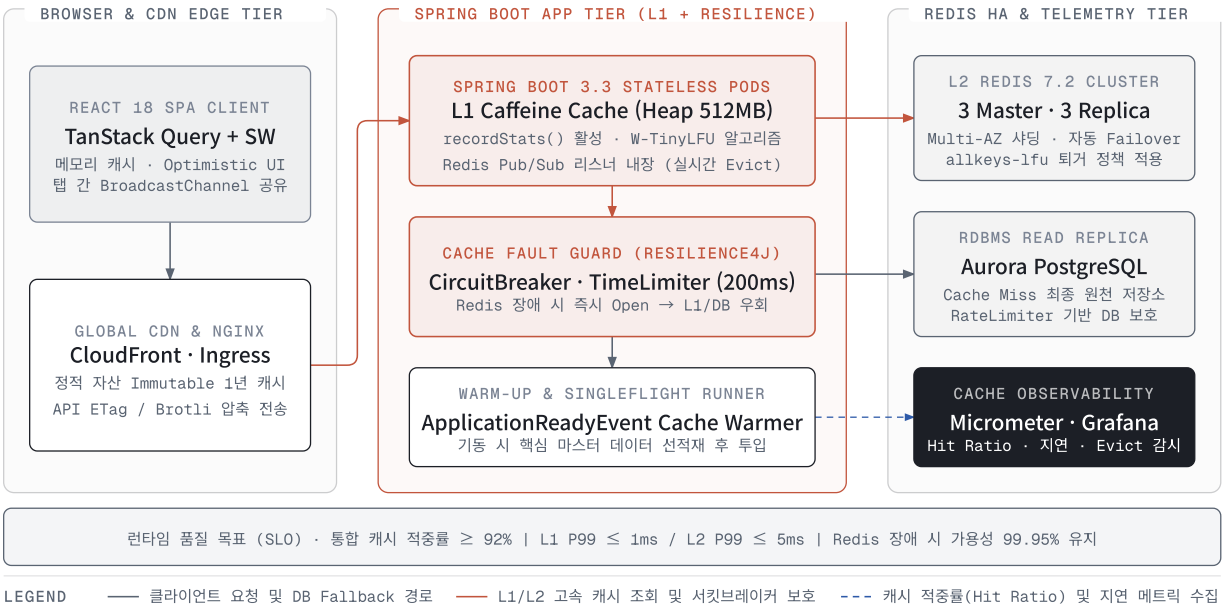
제5장. 물리 배포 및 회복탄력성·관측성 설계 (5. Deployment, Resilience & Observability View)

5.1 풀스택 캐시 런타임 토폴로지 및 서킷브레이커·관측성 구조도

- **물리 배치 및 고가용성 명세**: 프론트엔드 React SPA와 CloudFront CDN을 거쳐 진입한 트래픽은 Kubernetes/WAS 클러스터 내 **Stateless Spring Boot 파드(L1 Caffeine 내장)**에서 처리되며, 백엔드 분산 캐시는 **3개 가용영역(Multi-AZ)에 걸친 Redis 7.2 Cluster(3 Master + 3 Replica)**로 구성되어 단일 노드/AZ 장애 시 자동 Failover를 수행함.

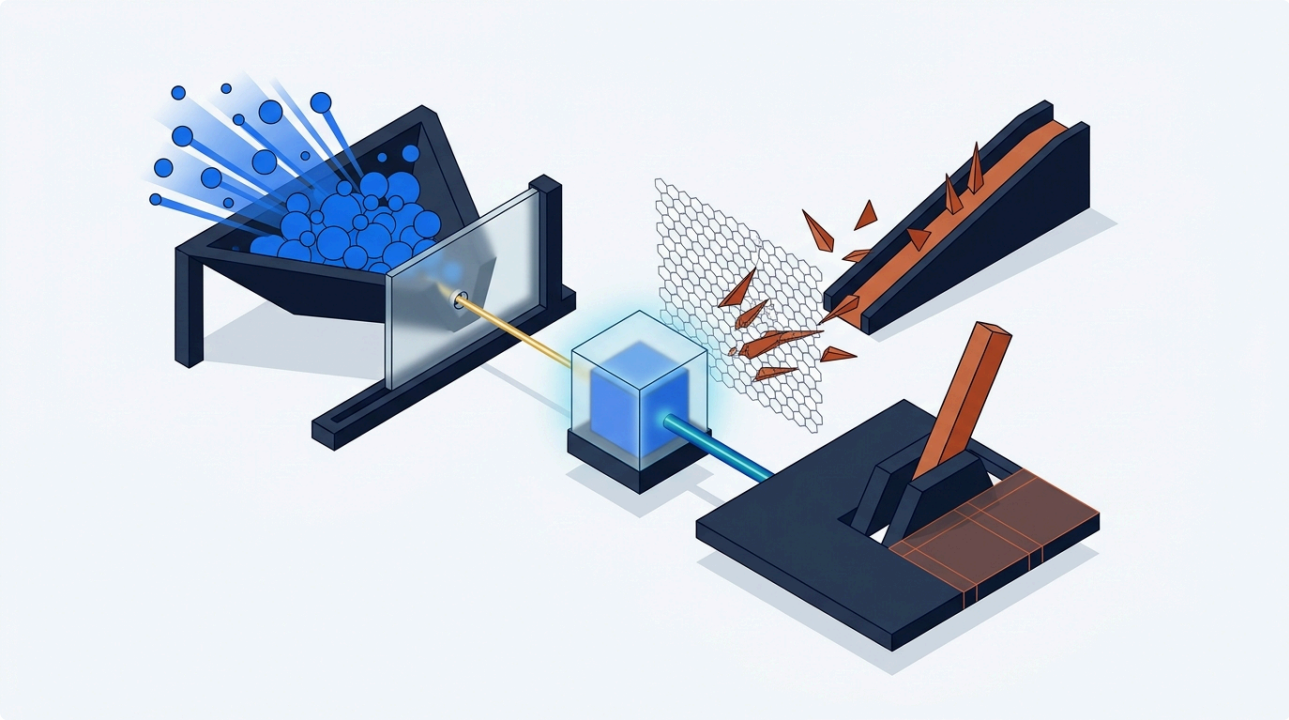
VIEW 04 · DEPLOYMENT TOPOLOGY, REDIS HA CLUSTER, CIRCUIT BREAKER & OBSERVABILITY

[배포 뷰] 풀스택 캐시 런타임 토폴로지 및 장애 격리·관측성 설계도



View 4 배포 뷰 — 풀스택 캐시 런타임 토폴로지 및 장애 격리·관측성 설계도

5.2 캐시 스탬피드(Stampede) 방어, 서킷브레이커(Fail-Open) 및 관측성 상세 설계



SingleFlight 뮤텍스 락, 블룸 필터 캐시 관통 방어 및 서킷브레이커 장애 격리 개념도

- **캐시 회복탄력성·메모리 보호·관측성 엔지니어링 파라미터 명세표 (Cache Engineering Specification)**

규격 ID	설계 영역	적용 대상 컴포넌트	표준 설정 파라미터 및 임계치 명세 (Parameter Spe
`CACHE-RES-01`	**Redis 클라이언트 타임아웃 및 토폴로지 자동 갱신**	Spring Data Redis (`LettuceConnectionFactory`)	• `command-timeout: 200ms`, `connect-time` • `ClusterTopologyRefreshOptions.enablePe` • `enableAllAdaptiveRefreshTriggers()` (`PERSISTENT_RECONNECTS`) • `SocketOptions.keepAlive(true)`
`CACHE-RES-02`	**캐시 장애 격리 서킷브레이커 (Fail-Open Fallback)**	Resilience4j `CircuitBreaker` + `TieredCache`	• `slidingWindowSize: 50`, `minimumNumber` • `failureRateThreshold: 40%`, `slowCallD` `150ms` • `waitDurationInOpenState: 10s`
`CACHE-RES-03`	**Redis 메모리 상한 및 퇴거 정책 (Eviction Policy)**	Redis 7.2 Cluster (`redis.conf`)	• `maxmemory: 75% of Node RAM` (Fork Copy 확보) • `maxmemory-policy: allkeys-lfu` (빈도 기 • `lazyfree-lazy-eviction: yes`, `lazyfre` `yes`, `lazyfree-lazy-server-del: yes`
`CACHE-RES-04`	**캐시 관측성 (Observability) 및 이상 징후 알람**	Micrometer + Prometheus + Grafana	• **필수 수집 메트릭**: `cache_gets_total(hit/mis` `cache_puts_total`, `cache_evictions_tot` `lettuce_command_completion_seconds` • **Redis 지표**: `used_memory_rss`, `evicted_k` `connected_clients`, `keyspace_hits`

제6장. 표준 인터페이스 및 코드 템플릿 명세 (6. Standard Interface & Code Specification)

6.1 백엔드(Spring Boot) 및 프론트엔드(React) 캐시 레이어 표준 구현 코드 템플릿

- **[Template 1] 도메인별 캐시 정책 레지스트리 및 Jitter TTL 계산기 (`CacheSpec.java` — `COMP-04`, `COMP-07`)**

JAVA

```
package com.company.platform.common.cache.config;

import java.time.Duration;
import java.util.concurrent.ThreadLocalRandom;
import lombok.Getter;
import lombok.RequiredArgsConstructor;

@Getter
@RequiredArgsConstructor
public enum CacheSpec {
    // 캐시명(버전 포함), L1 최대 개수, L1 TTL, L2 기본 TTL, Jitter 비율(0.15 = ±15%)
    COMMON_CODE("v1:common:code", 5_000, Duration.ofMinutes(10), Duration.ofHours(6), 0.15
    PRODUCT_DETAIL("v1:product:detail", 10_000, Duration.ofSeconds(15), Duration.ofMinutes
    USER_PERMISSION("v1:user:perm", 20_000, Duration.ofSeconds(10), Duration.ofMinutes(15)
    SEARCH_LIST("v1:search:list", 0, Duration.ZERO, Duration.ofMinutes(3), 0.15); // L1 미사

    private final String cacheName;
    private final int l1MaxSize;
    private final Duration l1Ttl;
    private final Duration l2BaseTtl;
    private final double jitterRatio;

    public Duration computeL2JitterTtl() {
        long baseMillis = l2BaseTtl.toMillis();
        double factor = 1.0 + ThreadLocalRandom.current().nextDouble(-jitterRatio, jitterR
        return Duration.ofMillis(Math.max(1000L, (long) (baseMillis * factor)));
    }
}
```

- **[Template 2] L1(Caffeine) + L2(Redis) + SingleFlight 통합 계층형 캐시 (`TieredCache.java` — `COMP-04`, `COMP-05`, `COMP-06`)**

JAVA

```java
package com.company.platform.common.cache.core;

import com.github.benmanes.caffeine.cache.Cache;
import java.util.concurrent.Callable;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ConcurrentHashMap;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.cache.support.AbstractValueAdaptingCache;
import org.springframework.data.redis.core.RedisTemplate;

@Slf4j
@RequiredArgsConstructor
public class TieredCache extends AbstractValueAdaptingCache {

    private final String name;
    private final CacheSpec spec;
    private final Cache<Object, Object> l1CaffeineCache; // null if L1 disabled
    private final RedisTemplate<String, Object> l2RedisTemplate;
    private final ConcurrentHashMap<String, CompletableFuture<Object>> inFlightLocks = new

    @Override
    protected Object lookup(Object key) {
        String redisKey = buildRedisKey(key);
        // 1. L1 (Caffeine) 로컬 메모리 조회 (0ms)
        if (l1CaffeineCache != null) {
            Object l1Val = l1CaffeineCache.getIfPresent(key);
            if (l1Val != null) {
                return l1Val;
            }
        }
        // 2. L2 (Redis Cluster) 분산 캐시 조회 + L1 역채움(Backfill)
        try {
            Object l2Val = l2RedisTemplate.opsForValue().get(redisKey);
            if (l2Val != null) {
                if (l1CaffeineCache != null) {
                    l1CaffeineCache.put(key, l2Val);
                }
                return l2Val;
            }
        } catch (Exception ex) {
            // [AP-05 Fail-Open] Redis 장애 시 예외를 전파하지 않고 DB 조회로 Fallback
            log.warn("[TieredCache] L2 Redis lookup failed for key={}. Falling back to ori
        }
        return null;
    }

    @Override
    @SuppressWarnings("unchecked")
    public <T> T get(Object key, Callable<T> valueLoader) {
        ValueWrapper cached = get(key);
        if (cached != null) {
            return (T) cached.get();
        }
```

```
        // 3. [COMP-06 SingleFlight] 핫 키 동시 만료 시 단일 스레드만 DB ValueLoader 실행
        String lockKey = buildRedisKey(key);
        CompletableFuture<Object> future = inFlightLocks.computeIfAbsent(lockKey, k ->
            CompletableFuture.supplyAsync(() -> {
                try {
                    T loaded = valueLoader.call();
                    put(key, loaded);
                    return toStoreValue(loaded);
                } catch (Exception e) {
                    throw new ValueRetrievalException(key, valueLoader, e);
                } finally {
                    inFlightLocks.remove(k);
                }
            })
        );
        return (T) fromStoreValue(future.join());
    }

    @Override
    public void put(Object key, Object value) {
        Object storeVal = toStoreValue(value);
        if (l1CaffeineCache != null) {
            l1CaffeineCache.put(key, storeVal);
        }
        try {
            l2RedisTemplate.opsForValue().set(buildRedisKey(key), storeVal, spec.computeL2
        } catch (Exception ex) {
            log.warn("[TieredCache] L2 Redis put failed for key={}", key, ex);
        }
    }

    public void evictLocalOnly(Object key) {
        if (l1CaffeineCache != null) {
            l1CaffeineCache.invalidate(key);
        }
    }

    private String buildRedisKey(Object key) {
        return "prod:commerce:" + name + ":" + key;
    }
}
```

- **[Template 3] 트랜잭션 커밋 이후(`AFTER_COMMIT`) L2 삭제 및 Pub/Sub 전 노드 L1 무효화 (`PostCommitCacheEvictor.java` — `COMP-08`)**

JAVA

```java
package com.company.platform.common.cache.sync;

import com.company.platform.common.cache.core.TieredCache;
import com.company.platform.common.cache.core.TieredCacheManager;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Component;
import org.springframework.transaction.event.TransactionPhase;
import org.springframework.transaction.event.TransactionalEventListener;

@Slf4j
@Component
@RequiredArgsConstructor
public class PostCommitCacheEvictor {

    public static final String INVALIDATION_TOPIC = "cache:invalidation:topic";
    private final StringRedisTemplate stringRedisTemplate;
    private final TieredCacheManager tieredCacheManager;

    public record CacheEvictEvent(String cacheName, String key) {}

    // [AP-02] 반드시 물리 DB 트랜잭션이 성공적으로 커밋된 직후에만 캐시 무효화 수행
    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    public void onAfterCommitEvict(CacheEvictEvent event) {
        String redisKey = "prod:commerce:" + event.cacheName() + ":" + event.key();
        try {
            // 1. L2 Redis 비동기 논블로킹 삭제 (DEL 대신 UNLINK 사용)
            stringRedisTemplate.unlink(redisKey);
            // 2. 클러스터 내 모든 WAS 파드의 L1 Caffeine 캐시 동시 제거를 위해 Pub/Sub 브로드캐스트
            String payload = event.cacheName() + "::" + event.key();
            stringRedisTemplate.convertAndSend(INVALIDATION_TOPIC, payload);
        } catch (Exception ex) {
            log.error("[CacheEvictor] Failed to evict L2/PubSub for {}", redisKey, ex);
            // Redis 장애 시에도 현재 노드의 L1 캐시는 즉시 비움
            TieredCache cache = tieredCacheManager.getTieredCache(event.cacheName());
            if (cache != null) cache.evictLocalOnly(event.key());
        }
    }
}
```

- **[Template 4] React 프론트엔드 `TanStack Query v5` 계층형 키 팩토리 및 낙관적 업데이트·무효화 훅 (`productQueries.ts` — `COMP-01`)**

TYPESCRIPT

```typescript
// File: src/entities/product/api/productQueries.ts
import { QueryClient, useQuery, useMutation, useQueryClient } from '@tanstack/react-query'
import { httpClient } from '@/shared/api/httpClient';

// 1. [Query Key Factory] 계층적 쿼리 키 표준화 (부분 무효화 및 자동완성 보장)
export const productKeys = {
  all: ['products'] as const,
  lists: () => [...productKeys.all, 'list'] as const,
  list: (filters: Record<string, unknown>) => [...productKeys.lists(), filters] as const,
  details: () => [...productKeys.all, 'detail'] as const,
  detail: (productId: string) => [...productKeys.details(), productId] as const,
};

// 2. [L0 Read Hook] staleTime(1분) 동안 네트워크 호출 0회 보장 + ETag 조건부 요청 연동
export function useProductDetailQuery(productId: string) {
  return useQuery({
    queryKey: productKeys.detail(productId),
    queryFn: async ({ signal }) => {
      const { data } = await httpClient.get(`/api/v1/products/${productId}`, { signal });
      return data.data;
    },
    staleTime: 60 * 1000,       // 1분간 Fresh 상태 유지 (중복 API 호출 원천 차단)
    gcTime: 10 * 60 * 1000,     // 언마운트 후 10분간 가비지 컬렉션 보류
    retry: 1,
  });
}

// 3. [Write & Invalidation Hook] 낙관적 업데이트(Optimistic UI) 및 관련 캐시 즉시 무효화
export function useUpdateProductMutation(productId: string) {
  const queryClient = useQueryClient();

  return useMutation({
    mutationFn: (payload: { name: string; price: number }) =>
      httpClient.put(`/api/v1/products/${productId}`, payload),
    onMutate: async (newProduct) => {
      await queryClient.cancelQueries({ queryKey: productKeys.detail(productId) });
      const previous = queryClient.getQueryData(productKeys.detail(productId));
      queryClient.setQueryData(productKeys.detail(productId), (old: any) => ({ ...old, ...
      return { previous };
    },
    onError: (_err, _newProduct, context) => {
      if (context?.previous) {
        queryClient.setQueryData(productKeys.detail(productId), context.previous);
      }
    },
    onSettled: () => {
      // 백엔드 AFTER_COMMIT 무효화와 동기화하여 상세 및 목록 캐시 재검증 트리거
      queryClient.invalidateQueries({ queryKey: productKeys.detail(productId) });
      queryClient.invalidateQueries({ queryKey: productKeys.lists() });
    },
  });
}
```

제7장. 아키텍처 적합성 검증 및 기술 스택 명세 (7. Architecture Fitness & Tech Stack BOM)

7.1 ArchUnit & ESLint 기반 캐시 아키텍처 적합성 자동 검증 규칙 (Fitness Functions)

모든 백엔드(Spring Boot) 및 프론트엔드(React) 코드는 CI 빌드 단계에서 아래 5대 캐시 아키텍처 적합성 규칙(`CACHE-RULE-01` ~ `CACHE-RULE-05`)을 자동 검증하며, 위반 시 PR 병합이 차단됨.

규칙 ID	캐시 적합성 검증 규칙명	검증 도구 / 대상	위반 판정 조건 (Violation Condition)	강제 수준
`CACHE-RULE-01`	**JPA `@Entity` 직접 캐싱 원천 금지**	ArchUnit (`..application..` , `..domain..`)	`@Cacheable` 또는 `@CachePut` 이 선언된 메서드의 반환 타입이 `@jakarta.persistence.Entity` 어노테이션을 보유할 시	**Blocker (CI 차단)**
`CACHE-RULE-02`	**트랜잭션 내부 Redis 직접 삭제/쓰기 금지**	ArchUnit (`..application.service..`)	`@Transactional` 메서드에서 `RedisTemplate.delete/unlink/set` 을 직접 호출할 시 (`AFTER_COMMIT` 리스너만 허용)	**Blocker (CI 차단)**
`CACHE-RULE-03`	**미등록 캐시 이름 (`CacheSpec` 외) 임의 사용 금지**	ArchUnit + Unit Test	`@Cacheable(cacheNames = "...")` 에 지정된 문자열이 `CacheSpec` Enum에 정의되지 않은 임의 문자열일 시 (무제한 TTL 사고 방지)	**Blocker (CI 차단)**
`CACHE-RULE-04`	**Redis 블로킹 명령어 (`KEYS` , `FLUSHALL`) 코드 사용 금지**	ArchUnit / Semgrep (`..adapter.out.cache..`)	`RedisConnection.keys()` , `flushAll()` , `flushDb()` 메서드를 호출하는 코드가 존재할 시	**Blocker (CI 차단)**
`CACHE-RULE-05`	**React Query 배열 키 하드코딩 금지 및 `staleTime` 명시**	ESLint (`@tanstack/eslint-plugin-query`)	`useQuery` 호출 시 `*Keys` 팩토리 함수를 사용하지 않거나, 전역/로컬 `staleTime` 설정이 누락될 시	**Critical (PR 차단)**

JAVA

```java
package com.company.platform.architecture;

import static com.tngtech.archunit.lang.syntax.ArchRuleDefinition.methods;
import static com.tngtech.archunit.lang.syntax.ArchRuleDefinition.noClasses;

import com.tngtech.archunit.core.importer.ImportOption;
import com.tngtech.archunit.junit.AnalyzeClasses;
import com.tngtech.archunit.junit.ArchTest;
import com.tngtech.archunit.lang.ArchRule;
import jakarta.persistence.Entity;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.transaction.annotation.Transactional;

@AnalyzeClasses(packages = "com.company", importOptions = ImportOption.DoNotIncludeTests.clas
public class CacheArchitectureFitnessTest {

    // [CACHE-RULE-01] @Cacheable / @CachePut 메서드는 절대 JPA @Entity를 직접 반환할 수 없음 (불변 DT
    @ArchTest
    static final ArchRule CACHE_RULE_01_forbid_caching_jpa_entities_directly =
        methods().that().areAnnotatedWith(Cacheable.class).or().areAnnotatedWith(CachePut.cla
            .should().haveRawReturnType(
                com.tngtech.archunit.base.DescribedPredicate.not(
                    com.tngtech.archunit.core.domain.properties.CanBeAnnotated.Predicates.ann
                )
            )
            .because("JPA @Entity 직접 캐싱은 LazyInitializationException 및 프록시 직렬화 오염을 유발

    // [CACHE-RULE-02] @Transactional 서비스는 RedisTemplate을 직접 참조하여 동기 삭제/갱신할 수 없음 (AF
    @ArchTest
    static final ArchRule CACHE_RULE_02_transactional_services_must_not_call_redis_directly =
        noClasses().that().resideInAPackage("..application.service..")
            .and().areAnnotatedWith(Transactional.class)
            .should().dependOnClassesThat().haveFullyQualifiedName("org.springframework.data.
            .because("트랜잭션 롤백 시 캐시 불일치를 막기 위해 캐시 무효화는 반드시 @TransactionalEventListe
}
```

7.2 전사 표준 풀스택 캐시 기술 스택 명세서 (Standard Cache Layer BOM)

기술 분류	표준 채택 기술 (Library / Engine)	표준 버전 (BOM)	아키텍처 역할 및 필수 설정 규격
Frontend State & Cache (L0)	React 18 + `@tanstack/react-query`	**v5.56.x**	`staleTime` / `gcTime` 분리, `broadcastQueryClient` (탭 동기화), `QueryKey` 팩토리 표준화
Edge & HTTP Cache (L1)	CloudFront / Nginx + `ShallowEtagHeaderFilter`	**RFC 9111**	`Cache-Control: s-maxage, stale-while-revalidate` , `If-None-Match` (`HTTP 304`)
Backend Local Cache (L2)	Caffeine Cache (`com.github.ben-manes.caffeine`)	**v3.1.8**	`W-TinyLFU` 퇴거 알고리즘, `recordStats()` , `maximumSize` 상한 강제, 0ms 힙 조회
Distributed Cache Grid (L3)	Redis Cluster (또는 Valkey)	**v7.2.x LTS**	3 Master / 3 Replica Multi-AZ, `allkeys-lfu` , `lazyfree-lazy-*=yes` , `UNLINK` 비동기 삭제
Redis Client & Lock	Spring Data Redis (`Lettuce`) + `Redisson`	**Boot 3.3 / 3.30.x**	Netty 논블로킹 I/O, `command-timeout=200ms` , `AdaptiveRefreshTriggers` , 분산 뮤텍스 락
Serialization & Compression	Jackson `JavaTimeModule` (Record) + `Zstd-jni`	**2.17.x / 1.5.x**	불변 Record 직렬화, 10KB 초과 페이로드 `Zstd` 압축 저장
Resilience & Fault Guard	Resilience4j (`CircuitBreaker` / `TimeLimiter`)	**v2.2.x**	Redis 장애 시 0ms Fail-Open 우회, `SingleFlight` 스탬피드 락, `BloomFilter` 관통 방어
Observability & Fitness	Micrometer + Prometheus + ArchUnit + ESLint	**1.13.x / 1.3.x**	계층별 Hit/Miss/Evict 메트릭 수집, `@Entity` 캐싱 차단(`CACHE-RULE-01~05`)