이원석 포트폴리오

Introduction

PoC 검증부터 프로덕션 최적화까지 전체 개발 사이클을 주도하며, 클라이언트 환경에 대한 깊은 이해를 바탕으로 백엔드 중심의 엔드투엔드 오너십을 실천합니다.

Achievements

- ONOL: 실시간 패킷 분석 및 스마트 로깅 파이프라인 구축 (TPS 6.7배 향상, 스토리지 90% 절감)
- Auto Throttle: TCP Vegas 기반 적응형 동시성 제어 오픈소스 구현 및 Maven Central 배포 (6.1ns 오버헤드, 지연 시간 6.7배 단축)
- MUTR: gRPC 및 Redis 코디네이터 기반 AI 분석 파이프라인 구축 (계보 무결성 보장 및 주제 변이 정밀 포착)

Technical Stack

- Language: Java, Kotlin
- Framework: Spring, Android
- Data & Messaging: SQL, Redis, Apache Kafka
- Infrastructure: Docker, AWS

Contact

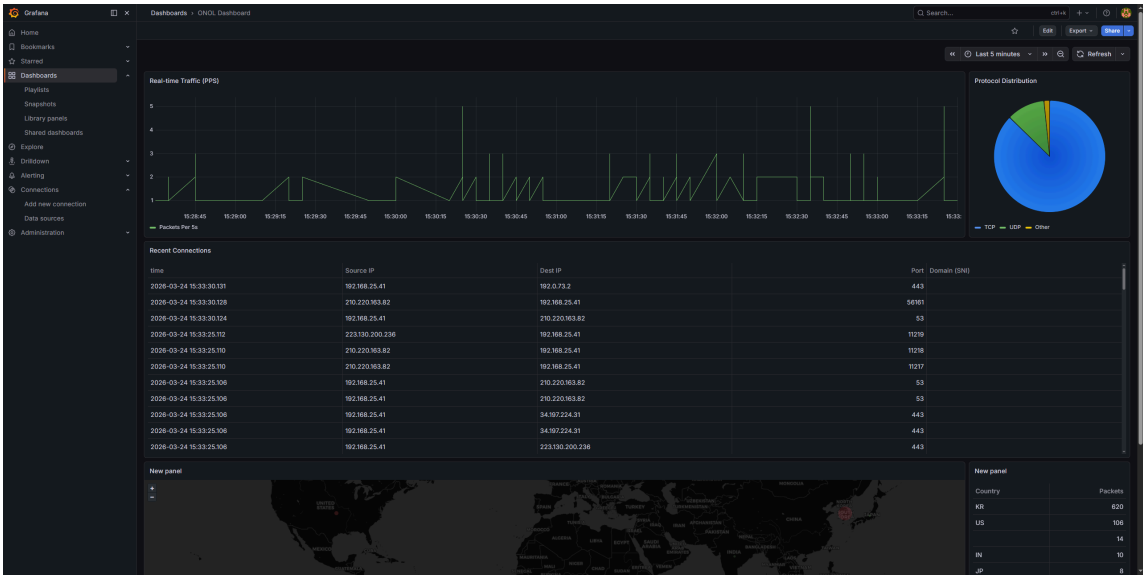
- GitHub: https://github.com/tenoenc
- Blog: https://tenoenc.github.io
- Email: tenoenc@gmail.com

ONOL

ONOL은 로컬 네트워크의 패킷을 실시간으로 수집하고 분석하여 위협 탐지와 통계 정보를 제공하는 네트워크 분석 시스템입니다. 단순히 모든 트래픽을 기록하는 기존 방식에서 벗어나, 데이터의 문맥을 파악해 가치 있는 정보만을 선별함으로써 스토리지 효율과 분석 정합성을 동시에 확보했습니다.

대용량 트래픽이 쏟아지는 환경에서도 시스템의 실시간성을 유지하기 위해 수집(Collector)과 분석(Analyzer) 레이어를 Kafka로 물리적 분리했으며, Redis 파이프라이닝과 비동기 논블로킹 I/O를 활용해 데이터 파이프라인의 병목을 제거했습니다. 또한 헥사고날 아키텍처를 적용하여 인프라의 변화에도 비즈니스 로직의 순수성을 보존할 수 있는 유연한 구조를 지향합니다.

Preview (전체 대시보드)

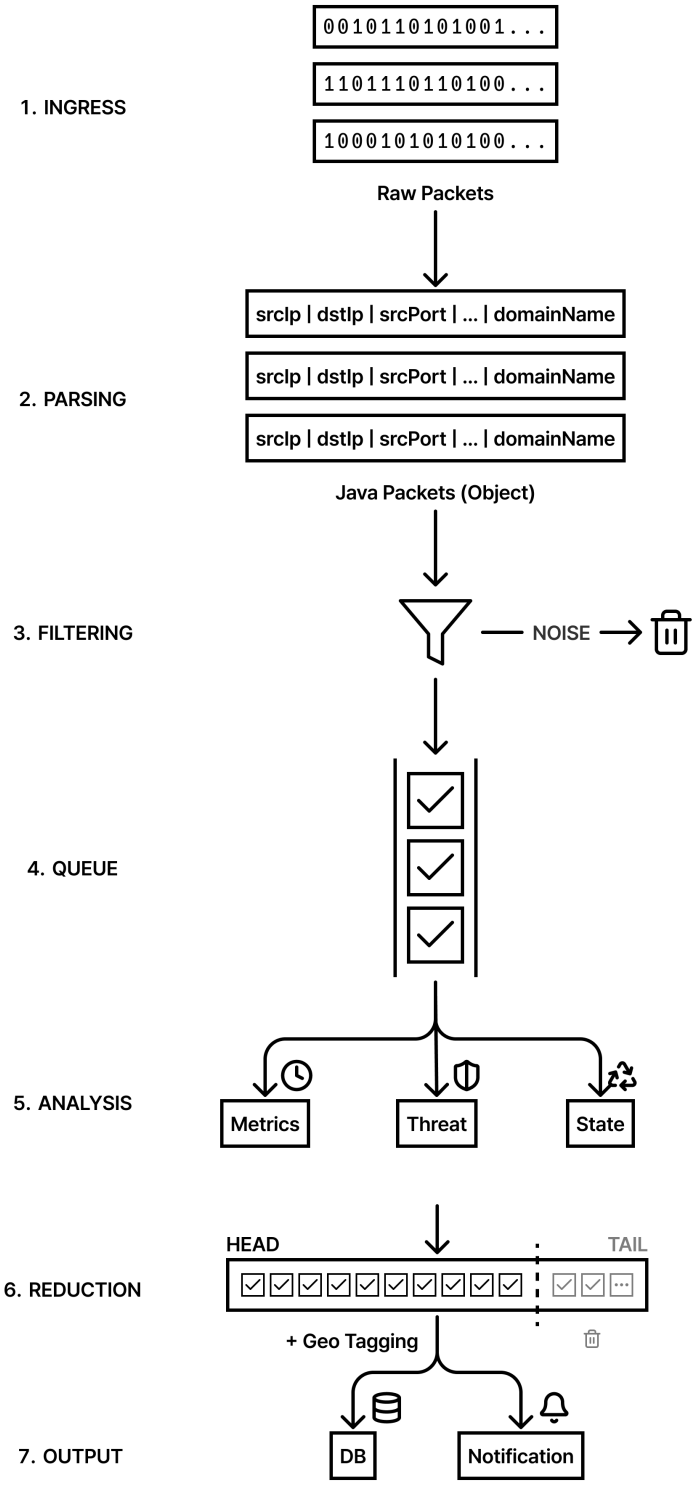


Key Features

- **Context-Aware Smart Logging**: 모든 패킷을 저장하지 않고 연결 제어 패킷과 세션 초반 10개의 핵심 문맥만 선별 저장하여, **스토리지 점유율을 90% 이상 절감**하고 DB 쓰기 성능을 개선합니다.
- **Bulk-Optimized Pipeline**: Redis Pipelining 및 Bulk 연산을 도입해 네트워크 RTT 부하를 최소화하고, **초당 처리량(TPS)을 6.7배 향상**시켜 8분 규모의 처리 지연을 해소했습니다.
- **Atomic Threat Detection**: 외부 공격자의 포트 스캔 행위를 실시간으로 식별합니다. Redis Set 구조를 통해 오탐을 방지하며, 분석 파이프라인과 독립된 비동기 알림 체계로 즉각적인 대응을 지원합니다.
- **Zero-Copy SNI Parser**: TLS Handshake의 Client Hello 영역을 바이트 단위로 직접 탐색하여, 트래픽 복호화 없이 접속 대상 도메인(SNI)을 실시간으로 추출합니다.
- **Self-Sustaining Backpressure**: `CallerRunsPolicy`를 적용하여 DB 저장 부하가 임계치를 넘을 경우 컨슈머가 직접 처리하게 함으로써, **시스템 붕괴 리스크를 방지하고 처리 속도를 자생적으로 조절**합니다.

System Architecture

데이터의 정합성과 효율적인 처리를 보장하기 위해 **이벤트 구동형 헥사고날 아키텍처**를 기반으로 설계되었습니다.



수집과 분석의 물리적 분리 (Collector-Analyzer)

- **Collector**: `Pcap4j` 기반의 수집기가 패킷을 캡처한 후 최소한의 노이즈 필터링을 거쳐 Kafka로 전송함으로써 수집 스레드의 부하를 최소화합니다.
- **Analyzer**: Kafka 배치를 소비하여 위협 탐지 및 세션 추적을 수행합니다. 처리 실패 데이터는 에러 메시지와 함께 DLQ(Dead Letter Queue)로 격리하여 유실 없는 재처리 환경을 구축했습니다.

도메인 중심의 상태 관리

- **TcpSessionTracker**: 패킷의 방향에 관계없이 동일한 Flow Key를 생성하고, Redis를 활용해 TCP 연결 상태를 실시간으로 관리하는 상태 머신 역할을 수행합니다.
- **PortScanDetector**: 특정 IP로부터 유입되는 서로 다른 목적지 포트의 개수를 Redis Set을 통해 원자적으로 카운팅하여 수직적 포트 스캔 공격을 즉각 탐지합니다.

Engineering Challenges

Redis N+1 문제 해결을 통한 처리량 확보

초기 설계에서는 패킷당 개별적인 Redis RTT가 발생하여 대용량 트래픽 환경에서 실시간성이 무너지는 현상이 발생했습니다. 배치 처리 시 1,250ms가 소요되어 약 8분의 데이터 처리 지연이 발생했습니다.

- **해결책**: **'Bulk Read → In-Memory 상태 전이 계산 → Bulk Write'** 패턴으로 로직을 리팩토링했습니다. `MultiGet` 과 Redis Pipelining을 도입하여 수천 개의 명령을 단 한 번의 네트워크 통신으로 처리하도록 최적화했습니다.
- **결과**: 배치 처리 시간을 **약 83% 단축**하고, TPS를 **6.7배(390 → 2,600)** 향상시켜 파이프라인의 실시간성을 확보했습니다.

스마트 로깅 정책을 통한 Elephant Flow 제어 및 스토리지 최적화

4K 영상 스트리밍 등 분석 가치가 낮은 대용량 UDP 트래픽이 무제한 저장되어 스토리지 낭비와 DB I/O 병목을 유발했습니다.

- **해결책**: Redis Atomic Increment를 활용해 세션별 패킷 누적 수를 추적하고, **초반 10개 패킷 및 제어 패킷만 저장**하는 선별적 로깅 정책을 도입했습니다. DNS와 같은 필수 식별 패킷은 유지하되 단순 데이터 패킷은 과감히 폐기했습니다.
- **결과**: 분석에 필요한 핵심 문맥은 100% 보존하면서 **스토리지 사용량을 90% 이상 절감**했으며, 처리 속도를 **9.8배 개선**했습니다.

Redis Set 기반의 원자적 위협 탐지 로직 설계

외부 공격자가 시스템의 취약점을 탐색하기 위해 수행하는 포트 스캔 행위를 실시간으로 식별해야 했습니다. 단순히 패킷의 양을 측정하는 방식은 정상적인 다중 연결과 악의적인 스캔 행위를 구분하지 못해 탐지 정합성이 낮아지는 결함이 있었습니다.

- **해결책**: Redis의 **Set 자료구조**를 활용해 특정 IP가 접속한 목적지 포트를 중복 없이 수집하고, 5분 단위의 슬라이딩 윈도우 내에서 고유 포트 개수를 확인하는 로직을 구현했습니다. `SADD` 와 `SCARD` 명령을 **Pipelining**으로 결합하여 원자적 카운팅을 수행하고 통신 오버헤드를 관리했습니다.
- **결과**: 위협 감지 시에도 분석 파이프라인의 성능을 유지하며 식별이 가능해졌으며, **이벤트 기반 아키텍처**를 통해 분석 로직과 알림 전파 로직(Discord)을 격리하여 시스템 안정성을 확보했습니다.

네트워크 환경 변화에 대응하는 Supervisor 루프 설계

물리 네트워크 인터페이스 변경이나 가상 인터페이스(Docker 등) 혼재 시 수집 프로세스가 중단되는 결함을 해결해야 했습니다.

- **해결책**: 5초 주기로 시스템의 모든 장치를 검색하여 최적의 물리 인터페이스를 자동 식별하는 **Supervisor 루프**를 구현했습니다.
- **결과**: 루프백이나 가상 장치를 제외한 실제 트래픽 장치를 우선 선택하며, 네트워크 단절이나 환경 변화 시에도 별도의 재시작 없이 **무중단 스위칭**이 가능한 안정성을 확보했습니다.

Performance Metrics

벤치마킹 브랜치를 통해 측정한 최적화 전(Legacy)과 후(Optimized)의 성능 지표입니다.

측정 지표	최적화 전 (Legacy)	최적화 후 (Optimized)	개선 결과
평균 처리 시간 (Batch 500)	~1,250 ms	**~220 ms**	**83.2% 단축**
초당 처리량 (TPS)	~390 TPS	**~2,600 TPS**	**6.7배 향상**
스토리지 사용량 (Streaming)	100% 저장	**< 10% 저장**	**90% 이상 절감**
시스템 지연 (Lag)	지속적 증가 (8분+)	**0초 (실시간 유지)**	**완전 해소**

Technical Stack

Core & Infrastructure

- **Java 21 & Spring Boot 3.3.2**: 가상 스레드 및 최신 Spring 생태계 활용
- **Apache Kafka 3.7**: 수집/분석 모듈 간 물리적 분리 및 메시지 버퍼링
- **Redis 7.2**: 실시간 세션 상태 관리 및 위협 탐지 카운터
- **TimescaleDB (PostgreSQL 17)**: 시계열 패킷 로그 저장 및 최적화

Packet Analysis

- **Pcap4j 1.8.2**: Native 네트워크 인터페이스 제어 및 패킷 캡처
- **Netty ByteBuf**: 바이트 조작 및 TLS SNI 직접 파싱
- **MaxMind GeoIP2**: IP 기반 전 세계 국가 코드 해소

Auto Throttle

Auto Throttle is a lightweight, adaptive concurrency control library designed for Spring Boot applications. Unlike traditional rate limiters that require static configuration, Auto Throttle dynamically adjusts concurrency limits in real-time based on the server's response time (RTT), protecting the application from overload while maximizing throughput.

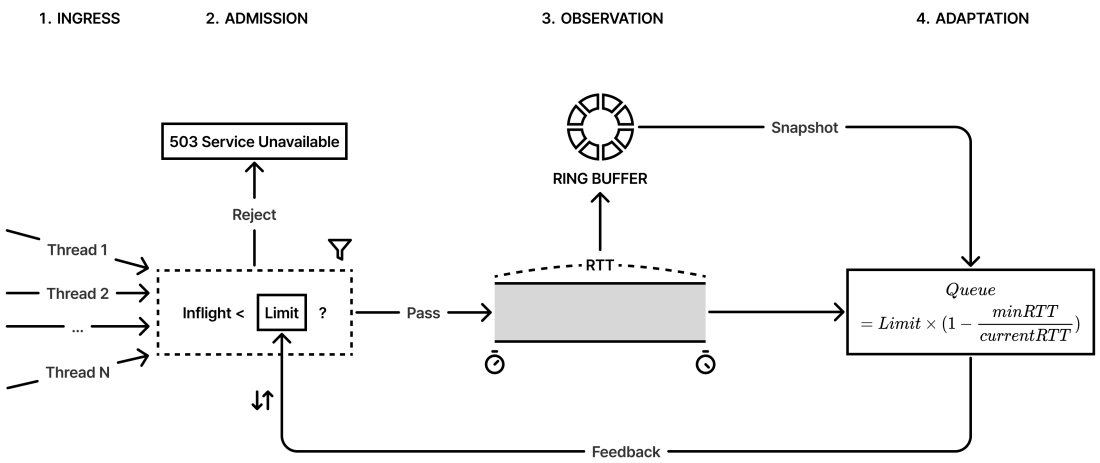
Overview

In distributed systems, static rate limiting is often insufficient because the capacity of a service fluctuates depending on downstream dependencies, database performance, and garbage collection pauses.

Auto Throttle implements the **TCP Vegas congestion avoidance algorithm**, adapted for application-layer concurrency control. It treats the service as a network pipe, measuring the round-trip time of requests to detect queuing delay. When latency increases, it gracefully reduces the concurrency limit; when latency recovers, it explores higher limits.

Key Features

- **Adaptive Control:** Automatically finds the optimal concurrency limit without manual tuning.
- **TCP Vegas Algorithm:** Uses queue size estimation based on minimum RTT vs. current RTT.
- **Zero-Overhead:** Built on Java 21 Virtual Threads and lock-free atomic primitives for nanosecond-level performance.
- **Fail-Fast:** Immediately rejects excess traffic with `HTTP 503 Service Unavailable` to prevent cascading failures.
- **Observability:** Seamless integration with Spring Boot Actuator and Micrometer.



Performance Benchmarks

Auto Throttle is designed to be **Zero-Overhead**. We verified the performance using two different methods: **Microbenchmark (JMH)** and **Load Testing (k6)**.

1. Microbenchmark (Internal Overhead)

How much time does it take to make a decision? **Less than 20 nanoseconds.**

We measured the core logic performance using JMH (Java Microbenchmark Harness).

Operation	Throughput (ops/s)	Average Time (ns/op)	Note
Acquire (Decision)	~160,000,000	**~6.1 ns**	Lock-Free / Zero-Allocation
Release (Feedback)	~75,000,000	**~13.3 ns**	High-Performance RingBuffer

> ***Result:*** *The overhead is negligible compared to typical HTTP request processing times (10ms+).*

2. Load Testing (Protection Capability)

Does it actually protect the server under heavy load?

We verified the effectiveness of Auto Throttle using **k6** load testing.

Test Scenario

- **Hardware:** Local Dev Machine (Intel i7-8700, 32GB RAM)
- **Environment:** Spring Boot 3.2 + Java 21 (Virtual Threads)
- **Traffic:** Ramp up to **3,000 concurrent users** (VUs)
- **Endpoint:** Simulated slow processing (100ms delay)

Results

The table below compares the server performance under extreme load.

Metric	Without Auto-Throttle	With Auto-Throttle	Impact
P95 Latency	**995.64 ms** (Severe Lag)	**148.66 ms** (Stable)	**6.7x Faster Response**
System State	Overloaded (Queue Buildup)	Healthy (Fast Failure)	**Prevented Cascading Failure**
Load Shedding	0 requests rejected	**~46,000 requests rejected**	**Effective Protection**

> ***Conclusion:*** *Without Auto Throttle, the server suffered from a backlog, causing response times to skyrocket to ~1 second. With Auto Throttle enabled, the server maintained its optimal response time (~150ms) by intelligently shedding excess load (HTTP 503), protecting existing users from degradation.*

How It Works

1. **Measurement:** The library measures the Round-Trip Time (RTT) of every request using a high-performance ring buffer.
2. **Aggregation:** Every 100ms (configurable), it calculates the average RTT and tracks the minimum RTT (minRTT) seen so far.
3. **Estimation:** It calculates the estimated queue size using Little's Law principles derived from TCP Vegas:

   > *QueueSize = CurrentLimit * (1 - minRTT / CurrentRTT)*

4. **Adjustment:**

- If `QueueSize < Alpha`: The system is underutilized. Increase the limit.
- If `QueueSize > Beta`: The system is congested. Decrease the limit.
- Otherwise: Maintain the current limit.

Caveats

1. Low Traffic Environments

Auto Throttle is optimized for high-concurrency scenarios. If your traffic is very low (e.g., < 50 RPS), the algorithm might not collect enough samples (`MIN_SAMPLES`) within the default window (100ms).

- **Solution:** Increase the update interval window size in configuration.

2. JVM Warm-up Phase

During the initial startup (Cold Start), request latencies may be higher due to JVM warmup (JIT compilation). The limiter might behave conservatively during this phase but will **automatically adapt** as the application stabilizes and latencies decrease.

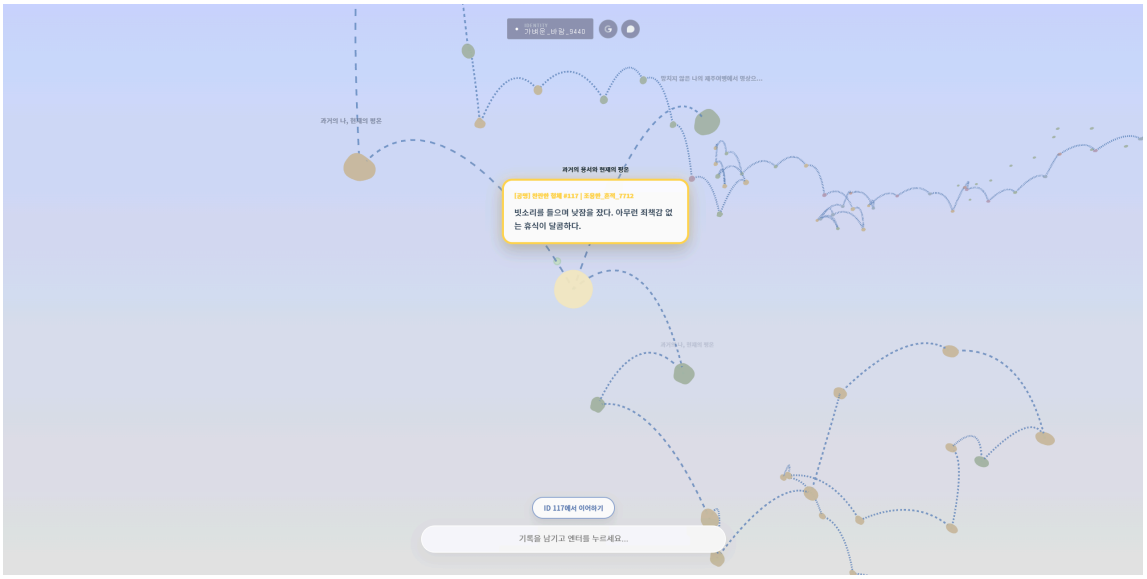
MUTR

MUTR은 텍스트 데이터의 의미적 변이와 계보를 3D 공간에 시각화하는 AI 기반 인터랙티브 아카이브입니다. 기존의 선형적인 타임라인 방식에서 벗어나 정보가 전파되며 파생되는 디지털 나비효과를 입체적으로 추적합니다.

사용자가 생성한 노드는 AI 엔진의 맥락 및 감정 분석을 통해 고유한 시각적 속성을 부여받습니다. 분석된 데이터는 사용자의 시선 방향과 확률적 편향 알고리즘에 따라 3D 좌표계 상의 최적의 위치에 배치되어 유기적인 군집을 형성합니다. 이러한 대규모 데이터의 실시간 분석 환경에서도 안정적인 동시성을 보장하기 위해 Java 21의 가상 스레드를 활용한 Non-blocking I/O 처리와 gRPC 통신 기반 아키텍처를 구축했습니다.

Live Demo: https://mutr.cloud

Preview (3D 공간 계보 탐색)

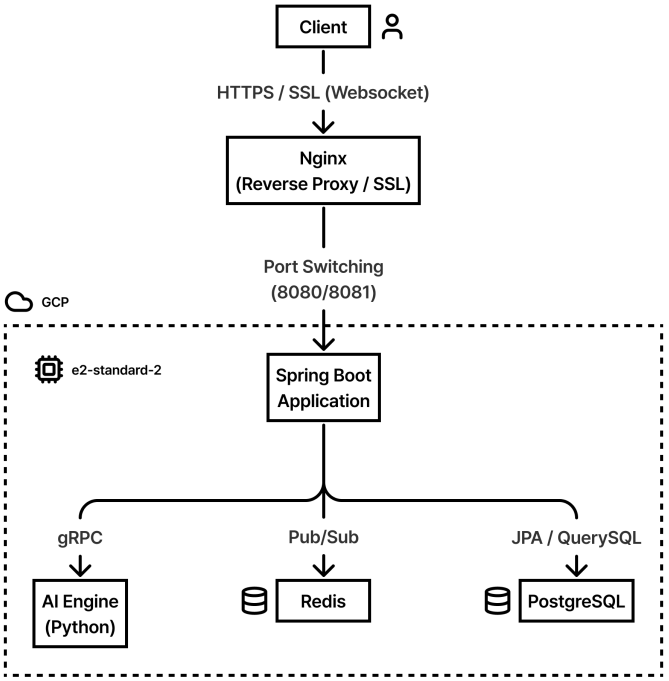


Key Features

- **Optimized AI Analysis Pipeline**: **gRPC와 Protobuf**를 적용하여 이기종 서버 간 통신 효율을 높이고, **SBERT 및 Llama 3.2** 모델을 연계해 정밀한 의미 유사도와 문맥을 도출합니다.
- **Gaze-Aware 3D Mapping**: 사용자의 실시간 **시선 방향과 확률적 편향 알고리즘**을 서버 도메인 로직에 적용하여, 3D 좌표계 상에서 데이터가 입체적인 군집을 형성하도록 제어합니다.
- **Atomic Lineage Coordinator**: **Redis 기반 대기열**을 구축하여 비동기 분석 환경의 경쟁 상태를 해결하고, 부모-자식 노드 간의 순차적 정합성과 데이터 계보의 무결성을 보장합니다.
- **Event-Driven Real-time Sync**: **트랜잭션 기반 이벤트 아키텍처**와 **STOMP 프로토콜**을 결합하여, 분석 완료 시점에 맞춰 별도의 새로고침 없는 실시간 3D 렌더링 동기화를 구현했습니다.

System Architecture

데이터의 정합성과 실시간성을 확보하기 위해 **이벤트 구동형 설계**를 기반으로 시스템을 구축했습니다. 물리적으로는 단일 인스턴스에 배포되지만, 논리적으로 모듈 간 결합도를 엄격히 제어하여 향후 MSA 전환이 용이한 구조를 지향합니다.



도메인 중심의 Modular Monolith 설계

복잡한 비즈니스 로직을 체계적으로 관리하기 위해 기능 단위로 모듈을 격리하고 응집도를 높였습니다.

- 엔티티가 좌표 계산과 상태 검증을 직접 수행하는 **Rich Domain Model**을 설계하여 데이터와 로직이 분리되는 것을 방지했습니다.
- 좌표나 변이 정보와 같은 핵심 개념을 **불변 객체**로 정의하여, 연산 과정에서의 사이드 이펙트를 차단하고 도메인의 무결성을 확보했습니다.
- 서비스 간의 복잡한 상호작용은 별도의 도메인 서비스로 분리하여 각 객체의 역할과 책임을 명확히 정의했습니다.

통신 프로토콜 이원화

서비스 요구사항에 맞춰 내부 엔진 통신과 클라이언트 동기화 전략을 최적화했습니다.

- **Internal (gRPC):** Spring Boot와 Python 서버 간에는 gRPC를 사용합니다. 바이너리 직렬화를 통해 데이터 크기를 줄이고, HTTP/2 기반의 멀티플렉싱으로 대량의 분석 요청을 지연 없이 처리합니다.
- **External (WebSocket):** 클라이언트와의 실시간 동기화에는 STOMP 프로토콜을 채택했습니다. 분석 완료 즉시 서버가 메시지를 발행하는 구조로 설계하여 불필요한 폴링 부하를 제거했습니다.

Redis를 통한 비동기 동시성 제어

비동기 분산 환경에서 부모 노드의 분석 완료 전 자식 노드가 생성될 때 발생하는 데이터 경합을 제어합니다.

- 분석 중인 부모를 참조하는 자식 노드 요청은 즉시 DB에 반영하지 않고 **Redis 대기열**에 격리하여 데이터 인과 관계를 보호합니다.
- 부모 노드의 분석 완료 이벤트가 발행되면 코디네이터가 이를 감지하여, 대기 중이던 자식 노드들을 **분산 락**을 통해 원자적으로 해제하고 파이프라인에 재투입합니다.

리소스 최적화 및 안정적인 배포 전략

제한된 리소스 환경(GCP e2-standard-2)에서 최대의 가용성을 확보하기 위한 인프라를 구성했습니다.

- **무중단 배포:** Docker Compose와 Nginx를 연계한 Blue/Green 배포를 자동화하여, 헬스 체크 통과 시에만 트래픽을 전환하는 안정적인 운영 환경을 구축했습니다.
- **SSL Offloading:** 호스트 레벨의 Nginx에서 SSL 종료를 전담하게 하여 애플리케이션 컨테이너의 부하를 줄이고, 배포 시 인증서 관련 다운타임을 차단했습니다.

Engineering Challenges

비동기 재귀 분석 환경에서의 데이터 경합 제어

부모 노드의 AI 분석이 완료되지 않은 시점에 자식 노드가 생성되면, 참조할 데이터가 부재하여 계보가 끊기는 경쟁 상태가 발생했습니다. 단순히 데이터베이스에 락을 거는 방식은 시스템 전체의 처리량을 저하시키는 병목이 될 것으로 판단했습니다.

- **해결책:** Redis 기반의 **상태 의존적 코디네이터**를 설계하여 문제를 해결했습니다. 분석 중인 부모를 둔 자식 노드 요청을 즉시 DB에 저장하지 않고 Redis 대기열에 격리했습니다. 이후 부모의 분석 완료 이벤트가 발행되면 코디네이터가 대기 중인 자식들을 감지하여 원자적으로 해제하고 분석 파이프라인에 재투입하는 구조를 구축했습니다.
- **결과:** 사용자 경험을 저해하지 않으면서도 복잡한 인과 관계를 가진 데이터의 순차적 무결성을 보장했습니다.

재귀 CTE를 통한 문맥 추출 최적화 및 주제 고착화 방지

초기에는 최근 텍스트와 직전 요약본을 단순 결합하여 AI에게 전달했으나, 중복된 문맥 정보로 인해 AI가 새로운 주제 변이를 포착하지 못하고 기존 주제를 답습하는 현상이 나타났습니다.

- **해결책:** PostgreSQL의 **재귀적 공통 테이블 식(Recursive CTE)** 을 도입하여 문맥 추출 로직을 재설계했습니다. 단일 쿼리로 노드의 계보를 역순 순회하며 텍스트 길이를 동적으로 합산하고, 정확히 500자 단위가 채워지는 시점의 조상 노드를 식별했습니다. 이를 통해 조상의 요약본과 이후 파생된 순수 변동분만을 결합하여 프롬프트를 재구성했습니다.
- **결과:** AI에게 중복 없는 고밀도 문맥을 제공함으로써 주제 분석의 정확도를 높이고 흐름의 미세한 변이를 정밀하게 포착하도록 개선했습니다.

이기종 시스템 간 저지연 데이터 통신 파이프라인 구축

Java 기반 웹 서버와 Python AI 엔진이 대규모 텍스트 데이터를 교환하는 환경에서, 기존 REST API 방식은 직렬화 과정의 높은 CPU 부하와 페이로드 크기 문제로 인해 성능 병목이 예상되었습니다.

- **해결책:** 두 서버 간 통신 프로토콜을 **gRPC**로 전환하여 I/O 효율을 개선했습니다. Protocol Buffers를 사용하여 데이터를 바이너리 형식으로 압축 전송함으로써 데이터 크기를 줄이고 파싱 비용을 최소화했습니다. 또한 인터페이스를 엄격하게 정의하여 런타임에 발생할 수 있는 데이터 타입 불일치 오류를 사전에 차단했습니다.
- **결과:** HTTP/2 멀티플렉싱을 통해 대량의 분석 요청을 지연 없이 처리하는 고성능 파이프라인을 실현했습니다.

AI 서버 가용성 대응을 위한 자동 복구 서비스 구현

AI 서버의 일시적인 장애나 재시작 시 진행 중이던 분석 작업이 유실되어 데이터 상태가 불일치하는 결함을 보완해야 했습니다.

- **해결책:** gRPC 헬스체크 기반의 **복구 서비스**를 구현했습니다. 서버의 가용 상태를 실시간으로 감지하고, 서버가 정상화되는 즉시 데이터베이스에서 미완료 상태로 남은 노드들을 식별합니다. 해당 노드들은 코디네이터에 의해 분석 파이프라인으로 자동으로 재투입됩니다.
- **결과:** 장애 상황에서도 데이터 유실 없이 분석을 재개할 수 있는 시스템의 회복 탄력성을 확보했습니다.

Technical Stack

Backend Core & Framework

시스템의 안정성과 확장성을 위해 최신 Java 생태계의 기술을 적극적으로 채택했습니다.

- **Java 21 & Spring Boot 3.5.9**: 가상 스레드 기반의 전담 실행자(Executor)를 적용하여 동시 처리량을 확보하고, Modular Monolith 구조를 통해 모듈 격리를 구현했습니다.
- **Spring Security & OAuth2:** JWT 기반 인증과 더불어 UUID 기반의 **Guest Token** 필터를 직접 구현하여, 로그인 없이도 모든 기능을 즉시 체험할 수 있는 환경을 제공합니다.
- **Spring Events & STOMP:** 트랜잭션 커밋 시점과 외부 로직을 분리하기 위해 이벤트를 활용하며, 분석 완료 데이터는 **웹소켓**을 통해 클라이언트에 실시간으로 푸시됩니다.

Database & Persistence Strategy

데이터의 특성에 맞춰 저장소의 기능을 최적화하여 사용했습니다.

- **PostgreSQL 17:** 계층형 데이터인 노드 계보를 효율적으로 조회하기 위해 **재귀적 공통 테이블 식 (Recursive CTE)** 을 사용하며, 가변적인 AI 분석 결과는 **JSONB** 타입을 활용해 유연하게 저장합니다.
- **Redis 7:** Redisson 기반의 **분산 락**과 대기열을 구축하여, 비동기 환경에서 발생하는 데이터 경합을 제어하고 분석 파이프라인의 순차적 무결성을 보장하는 코디네이터로 활용합니다.

AI Interface & Communication

이기종 시스템 간의 데이터 전송 효율과 분석 모델의 성능을 고려했습니다.

- **gRPC & Protobuf:** Java 웹 서버와 Python AI 엔진 간의 통신에 도입했습니다. 바이너리 직렬화를 통해 대량의 텍스트와 벡터 데이터 전송 시 발생하는 오버헤드를 최소화했습니다.
- **Python 3.10 & PyTorch:** AI 모델 서빙을 위한 런타임으로 사용하며, **Llama 3.2** 및 **KR-SBERT** 모델을 통해 텍스트의 맥락 분석과 변이 측정을 수행합니다.

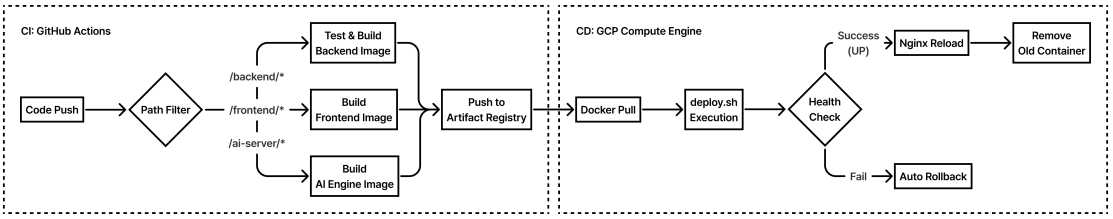
Infrastructure & DevOps

제한된 서버 리소스 내에서 서비스 가용성을 유지하기 위한 환경을 구축했습니다.

- **GCP & Nginx:** 호스트 레벨의 Nginx에서 **SSL Offloading**을 수행하여 백엔드 부하를 줄이고, Docker Compose를 연계하여 단일 인스턴스 내에서 **Blue/Green 무중단 배포**를 실현했습니다.
- **GitHub Actions:** 코드 변경 감지 기반의 **선택적 빌드** 파이프라인을 구축하여 CI 시간을 단축하고, 배포 전 과정을 자동화했습니다.

Deployment Strategy

GitHub Actions와 Google Artifact Registry를 연계하여 빌드부터 배포까지 전 과정을 자동화했습니다. 모노레포 구조의 운영 효율을 높이고 제한된 리소스 환경에서 서비스 연속성을 보장하는 데 중점을 두었습니다.



효율적인 빌드를 위한 선택적 CI 파이프라인

단일 저장소 내에서 여러 모듈을 관리함에 따라 발생하는 빌드 비효율을 해결하기 위해 변경 감지 기반의 파이프라인을 구축했습니다. `dorny/paths-filter` 를 도입하여 커밋된 코드 중 실제로 수정된 모듈만 식별하고 필요한 서비스만 선별적으로 빌드함으로써 CI 소요 시간을 단축했습니다. 또한 빌드 전 단계에서 테스트 통과 여부를 검증하는 품질 게이트를 배치하여 안정성이 확보된 코드만 배포 프로세스에 진입하도록 강제했습니다.

단일 인스턴스 기반의 무중단 배포 전략

별도의 로드밸런서가 없는 제한된 서버 리소스 환경에서 서비스 중단 없는 배포를 위해 독자적인 Blue/Green 전략을 수립했습니다. Nginx가 참조하는 포트 설정 파일을 원자적으로 교체하고 리로드하는 방식을 적용하여 트래픽 유실 없는 서비스 전환을 실현했습니다. 배포 스크립트는 신규 컨테이너의 헬스체크를 일정 시간 폴링하며, 정상 응답이 없을 경우 즉시 배포를 중단하고 자동 롤백을 수행하여 시스템의 가용성을 보호합니다.

보안 강화 및 운영 가시성 확보

배포 과정의 모든 민감 정보는 GitHub Secrets를 통해 안전하게 관리하며, 모든 명령은 SSH 암호화 채널을 통해 수행됩니다. 배포 결과와 커밋 메타데이터는 Discord Webhook을 통해 실시간으로 공유됩니다. 이를 통해 개발자가 운영 상황을 즉각적으로 인지할 수 있는 피드백 루프를 구축하고 장애 발생 시 신속한 대응이 가능하도록 했습니다.