

# 복잡한 시스템을 단순하게, 서비스는 더 안정적으로.

서비스 구조 설계부터 성능 개선, 인증 통합, 배포와 장애 복구까지  
맡아온 3년 8개월차 소프트웨어 엔지니어 고지훈입니다.

|                                     |                              |
|-------------------------------------|------------------------------|
| 성능 최적화<br>쿼리 튜닝 · 병목 분석             | 인증 · 권한 설계<br>SSO 통합 · 접근 제어 |
| 인프라 · 배포 자동화<br>Kubernetes · GitOps | 운영 안정성<br>장애 복구 · 재발 방지      |

01 소개

about

새로운 기술을 배우고 실무에 도입하는 데 거리낌이 없는 개발자입니다.  
습득한 기술을 고도화해 업무 효율을 높이고 비즈니스에 직접 기여하는 솔루션을 만드는 데 주력합니다.  
코드의 효율성과 유지보수성을 높이는 것을 중요하게 생각하고, 클라우드 인프라와 데이터베이스 운영 경험을 함께 갖추고 있습니다.

|                                       |                                                                               |
|---------------------------------------|-------------------------------------------------------------------------------|
| 경력                                    | 활동 · 수상 · 학력                                                                  |
| <b>메디컬아이피</b><br>2025.06 – 현재         | - 토스 Learner's High 1기 (2024.12 – 2025.01) — 토스 개발자 멘토링에서 커리어 점검과 실무 개선 과제 수행 |
| - DB 설계 및 개선                          | - 2022 한이음 공모전 한국정보산업연합회 회장상 수상 (과학기술정보통신부)                                   |
| - 인프라 개선                              | - 2022 신한 스쿼어 브릿지 대학생 부문 우수상 수상                                               |
| - 성능 최적화 및 유지보수                       | - 철도 시설물 성능평가 관련 프로그램 특허 출원 경험                                                |
| - 클라우드 운영 및 관리                        | - 피플(현혈 매칭 플랫폼) 창업 및 2023 예비사회적 기업 선정 (서비스 수익화 경험)                            |
| <b>네모시스</b><br>2023.02 – 2025.05      | - 평택대학교 융합소프트웨어학과 졸업 (4.27 / 4.5)                                             |
| - 소프트웨어 설계 및 개발                       |                                                                               |
| - ERD 및 DB 설계                         |                                                                               |
| - 성능 최적화 및 유지보수                       |                                                                               |
| - 클라우드 운영 및 관리                        |                                                                               |
| <b>피플 · 코파운더</b><br>2022.02 – 2023.11 |                                                                               |
| - 공동 창업 및 서비스 기획                      |                                                                               |
| - DB 설계                               |                                                                               |
| - 앱 개발                                |                                                                               |

## 01 파일 데이터센터 성능 개선 및 비동기 처리 아키텍처 구축

MedicalIP 설계 · 구축 · 성능 검증

소수 사용자의 대량 파일 업로드 속도를 높이고, 검증  
·DB 연결 부하를 제어해 안정적인 처리 구조를 만들었습니다.

|        |                                                                                                                                                                                                                                                                                                            |
|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 문제     | <p>소수 사용자가 수만~수십만 개 파일을 업로드해 총 업로드량이 TB 규모에 이르는 환경으로, 다수 사용자의 동시 접속을 수용하는 것보다 각 사용자의 대량 업로드를 빠르고 안정적으로 완료하는 데 우선순위를 뒀습니다.</p> <p>파일 전송·검증·메타데이터 조회가 같은 자원을 쓰며 스레드·커넥션을 오래 점유해, 업로드 처리량과 일반 API 응답이 함께 영향을 받았습니다.</p>                                                                                       |
| 판단     | <p>Presigned URL로 파일을 스토리지에 직접 전송하도록 구성한 뒤, 업로드 속도를 유지하면서 검증을 자동화하고, 처리량이 늘어도 노드와 DB가 과부하되지 않도록 설계했습니다.</p> <p>파일 전송 구간뿐 아니라 검증과 마무리를 포함한 전체 완료 시간도 함께 살피며 개선 효과를 확인했습니다.</p>                                                                                                                              |
| 비동기 처리 | <p>파일 검증·후처리 이벤트를 Kafka 기반 비동기 워커로 전달해 백엔드 요청 경로에서 검증·후처리를 분리했습니다.</p> <p>RabbitMQ 등과 비교하면서 대량 이벤트 처리 성능과 함께, <b>Kafka Connect·Kafka Streams·CDC</b> 연계로 사내 데이터 파이프라인을 확장할 수 있는 생태계를 주요 선택 기준으로 삼았습니다.</p>                                                                                                 |
| KEDA   | <p>처음에는 Pod를 0개에서 기동했지만, 동시 기동으로 노드 부하가 급증해 <b>Pod 2~3개</b>를 상시 유지하고 작업이 밀릴 때 추가 확장하도록 바꿨습니다.</p> <p>바로 처리할 용량을 확보한 상태에서 KEDA가 Kafka Consumer Lag(소비 지연 메시지 수)를 기준으로 워커를 늘리도록 하고, 워커별 작업 크기와 동시 실행 수도 제한해 확장 시 부하를 관리했습니다.</p>                                                                             |
| 연결 관리  | <p>Pod가 늘면 DB 연결 요청도 함께 증가하므로, <b>PgBouncer</b>로 연결을 풀링·재사용하고 API와 워커의 DB 연결 경로를 분리했습니다.</p> <p>이를 통해 워커 확장에 따른 DB 커넥션 증가 부담을 줄이고, 메타데이터 조회 범위와 인덱스도 조정해 불필요한 DB 부하를 낮췄습니다.</p>                                                                                                                            |
| 결과     | <p>전송·검증·DB 연결의 부하를 분리하고 워커 확장 정책을 조정해 소수 사용자의 대량 업로드 처리 성능과 서비스 운영 안정성을 함께 높였습니다.</p> <p>72.94GB·150,205개 입력 파일을 사용한 AIMD 적용 전후 비교에서 업로더 실행 시간은 고정 동시 요청 100개 대비 <b>22분 18.0초</b>에서 <b>15분 28.3초</b>로 단축되었습니다.</p> <p>같은 비교에서 입력 용량 기준 평균 처리량은 54.5 MB/s에서 78.6 MB/s로 증가하고 재시도는 3,574회에서 187회로 줄었습니다.</p> |
| 검증 성능  | <p>초기 검증 파이프라인의 2026년 7월 <b>5.37GB·10,636개</b> 파일 측정에서 파일 선언은 18.8초, 파일 전송은 34.9초, 검증 구간은 216.9초였습니다.</p> <p>선언·전송·검증을 포함한 <b>전체 완료 시간은 270.7초 (4분 30.7초)</b>였고, 10,636개 파일 모두 검증에 성공했습니다.</p>                                                                                                             |

2026년 8월의 **72.94GB(67.93GiB)·150,205개 입력 파일** 비교에서  
AIMD의 파일 전송 구간은 927.7초, 업로더 전체 실행은 928.3초였습니다.  
측정 구간은 업로더의 스캔·제어 요청·전송·재시도·마무리까지입니다.  
입력 용량을 전체 실행 시간으로 나눈 **평균 처리량은 78.6 MB/s**였으며, 업로드 완료 149,897건·재시도 187회를 기록했습니다.

## TB 규모

수만~수십만 개 파일의 총 업로드량

## Pod 2~3개

상시 유지 · 대기량에 따라 추가 확장

22분 18초 →

15분 28초

72.94GB 입력 ·  
업로더 실행 · 2026.08

3,574 → 187회

동일 데이터 비교 · 재시도 횟수

### 도식 1 파일 검증·후처리를 요청 처리와 어떻게 분리했는가?

KAFKA 기반 비동기 검증·후처리



검증·후처리는 Kafka 이벤트를 받아 워커에서 실행하고, 백엔드 요청 경로와 분리합니다.

Kafka로 파일 처리 이벤트를 전달하고, 비동기 워커에서 검증·후처리를 실행합니다.

## 도식 2 검증 처리량을 늘려도 노드와 DB가 과부하되지 않게 하려면?

### KEDA — 검증 처리 용량 확보 + 노드 부하 제어

Pod 0개에서 동시 기동할 때 발생한 노드 부하 급증을 줄였습니다.

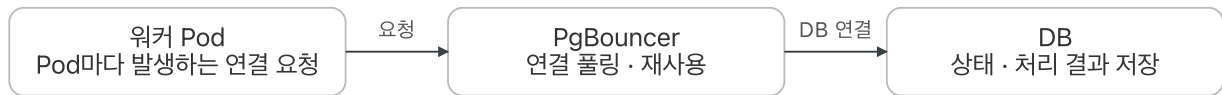


기본 Pod 2~3개를 유지해 파일 수신 이벤트와 대기 작업을 처리합니다.

작업 크기와 워커별 동시 실행 수를 제한해 확장 시 자원 사용을 관리합니다.

### PGBOUNCER — 워커 확장에 따른 DB 연결 부하 제어

Pod가 늘 때 함께 증가하는 DB 연결 요청을 커넥션 풀링으로 완화했습니다.



API와 워커의 DB 연결 경로를 분리하고, 워커 연결은 PgBouncer를 거쳐게 했습니다.

처리 Pod를 유연하게 늘리면서도 DB 커넥션 증가 부담을 함께 줄였습니다.

Kafka 기반 검증·후처리 워커는 KEDA로 처리 용량을 늘리고, PgBouncer로 DB 연결 부담을 줄였습니다.

Presigned URL

S3 호환 스토리지

Kafka

KEDA

PgBouncer

Transactional Outbox

비동기 워커

부하 테스트

## 02 사내 SSO 시스템 구축

MedicalIP 2026.02 - 2026.03 · 약 2개월 설계 · 구축

Keycloak 기반 SSO로 반복 로그인을 줄이고, 사내 시스템의 접근 권한·세션 관리를 중앙화했습니다.

|    |                                                                                                                                                                                                                                                                                                                                        |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 문제 | 시스템마다 로그인과 세션 기준이 따로 있었고, 계정을 만들고 회수하는 일은 각 도구의 관리 화면에서 손으로 처리해야 했습니다.<br>반복 로그인이 불편하다는 건 표면적인 문제였고, 실제 위험은 <b>퇴사자 권한이 남아 있어도 알려주는 사람이 없다는 점</b> 이었습니다.                                                                                                                                                                                |
| 판단 | 자체 인증 모듈도 검토했지만, Sentry·Rancher·VPN 같은 외부 솔루션까지 연동하기에는 제약이 있어 <b>표준 인증 프로토콜로 여러 솔루션을 연결하고 인증 정책을 한곳에서 관리할 수 있는 Keycloak</b> 을 선택했습니다.                                                                                                                                                                                                  |
| 실행 | Naver Works를 IdP로 연동해,<br><b>기존 Naver Works 계정으로 Keycloak에 로그인하도록</b> 구성했습니다.<br>Keycloak 로그인 후에는 SSO를 통해 사내 시스템에 접근하도록 연동했습니다.<br>Keycloak과 Spring Security·Gateway로 사내 시스템의 OAuth2/OIDC 인증 연동을 구성하고, Admin Console·Sentry·Rancher·VPN 등에 공통 접근 권한·세션 정책을 적용했습니다.<br>로그인과 주요 접근에는 MFA를 적용했습니다.<br>Keycloak을 분산 배포해 인증 서비스 가용성을 높였습니다. |
| 결과 | Keycloak 로그인 후 사내 시스템에 SSO로 접근할 수 있게 해,<br><b>업무 도구를 오갈 때 반복 로그인하는 불편을 줄였습니다.</b><br>사내 시스템 접근 권한과 세션 정책을 중앙에서 관리합니다.<br>권한 판단 기준도 역할에서 속성으로 바뀌어(RBAC → ABAC) 역할이 늘어나도 관리 지점이 늘지 않습니다.                                                                                                                                                 |

### 통합 로그인

Keycloak SSO · 사내 시스템 접근

### MFA

로그인 · 주요 접근

### ABAC

역할 → 속성 기반

### 중앙화

접근 권한 · 세션 정책

#### 도식 1 KEYCLOAK 로그인은 사내 시스템 접근으로 어떻게 이어지는가?



권한은 역할이 아니라 속성으로 판단합니다 (RBAC → ABAC). 도구가 늘어나도 권한 판단 기준은 하나입니다.

외부 IdP의 계정 인증으로 Keycloak에 로그인하고, Keycloak의 SSO를 통해 사내 시스템에 접근합니다.

Keycloak

OAuth2/OIDC

Spring Security

ABAC

MFA

### 03 온프레미스 Kubernetes 및 DR 환경 구축

MedicalIP 2025.11 - 2026.02 · 약 4개월 설계 · 구축 · 운영

GitOps·모니터링·트래픽 제어를 Kubernetes로 통합하고,  
온프레미스 운영과 AWS DR로 비용·복구 체계를 개선했습니다.

|        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 문제     | 인프라 운영의 우선순위는 <b>비용 절감</b> 이었으며,<br>트래픽은 빈번한 Scale-out·Scale-in이 필요할 정도로 변동하지 않았고,<br>사내에는 유휴 컴퓨팅 장비 약 4~5대와 NAS 17대가 있었습니다.<br>기존 Docker Swarm 운영 환경에서는 GitOps·모니터링·<br>트래픽 제어를 일관된 방식으로 연결하고 관리할 필요가 있었습니다.                                                                                                                                                                                                                                                                                                                                                                 |
| 판단     | 상시 클라우드 비용을 줄이기 위해 <b>기존 사내 장비로 운영 워크로드를 감당하는 온프레미스 구성</b> 을 선택하고,<br>유휴 장비는 컴퓨팅 자원으로, NAS는 영속 스토리지로 활용하면서 AWS는 DR 중심으로 축소했습니다.<br>작은 인프라 확장보다 현재 트래픽에 필요한 용량과 복구 경로를 경제적으로 확보하는 데 집중했습니다.                                                                                                                                                                                                                                                                                                                                                                                    |
| 전환 이유  | <b>ArgoCD 기반 GitOps, Prometheus·Loki 기반 모니터링, Istio·Gateway API 기반 트래픽 제어</b> 를 통합하기 위해 Kubernetes로 전환하면서,<br>배포 설정과 운영 정책을 리소스로 선언하고 컨트롤러가 일관되게 반영하는 방식을 선택했습니다.                                                                                                                                                                                                                                                                                                                                                                                                             |
| 생태계 활용 | <b>ArgoCD 기반 GitOps</b> 로 배포 변경과 롤백 기준을 Git에 모으고,<br><b>Prometheus 스택과 Loki</b> 로 메트릭·알림·로그를 함께 살펴 장애 원인을 찾도록 구성했습니다.<br>원하는 구성을 리소스로 선언하면 컨트롤러가 반영하는 <b>Operator 패턴</b> 은<br>Prometheus Operator처럼 반복적인 운영 설정을 자동화할 수 있다는 점에서 장점으로 봤습니다.<br>트래픽 제어에는 <b>Istio·Gateway API</b> 를 적용해 헤더 기반 카나리 배포로 beta(스테이징) 환경을 구성했습니다.                                                                                                                                                                                                                                                     |
| 실행     | 유휴 컴퓨팅 장비 약 4~5대로 RKE 클러스터를 구성해 Rancher로<br>관리하면서 AWS의 운영 워크로드를 온프레미스로 이관하고,<br><b>NAS 17대를 Kubernetes 스토리지에 연결해 PV(PersistentVolume)</b><br><b>로 활용</b> 하며 애플리케이션의 영속 데이터를 저장했습니다.<br>Readiness·Liveness Probe와 ReplicaSet·DaemonSet을 운영 기반으로 삼고,<br>ArgoCD App of Apps와 Image Updater로 GitOps 배포를 구성했습니다.<br>Istio와 Gateway API로 <b>헤더 기반 카나리 배포를 도입해 beta(스테이징) 환경</b> 을 구성하고,<br>Kiali로 서비스 간 트래픽 흐름을 확인했습니다.<br>AWS는 DR 중심으로 사용 범위를 줄이고,<br>Route53 HealthCheck 실패를 SQS로 보내 KEDA가 AWS 클러스터의 Pod를 기동하도록 구성했습니다.<br>DB Replication과 복구 경로는 실행 시나리오로 정리해 정기 DR 훈련으로 검증했습니다. |
| 결과     | GitOps 배포·메트릭/로그 모니터링·트래픽 제어를 하나의 운영 체계로 연결하면서,<br>ArgoCD로 배포 이력과 롤백 기준을 관리하고 헤더 기반 카나리 배포로 beta(스테이징) 환경을 구성했습니다.<br>장애가 감지되면 개발자 개입 없이 복구가 시작되며,<br>로그 수집 자동화와 메트릭 모니터링으로 이슈 대응 속도가 두 배가 되었습니다.<br>전환 이후 확인한 <b>AWS 월 비용은 2026년 1월 \$448.19에서 9월 \$46.22로 89.7% 감소</b> 했고,<br>EC2 서비스 요금은 2026년 1월 \$238.98에서 7월 \$9.03로 줄었습니다.                                                                                                                                                                                                                                             |

89.7%↓

AWS 월 비용 · 2026.01 → 09

\$46.22

2026.09 월 비용

약 4~5대

유휴 컴퓨팅 장비 활용

NAS 17대

스토리지 연결 · PV 활용

RKE · Rancher

온프레미스 클러스터

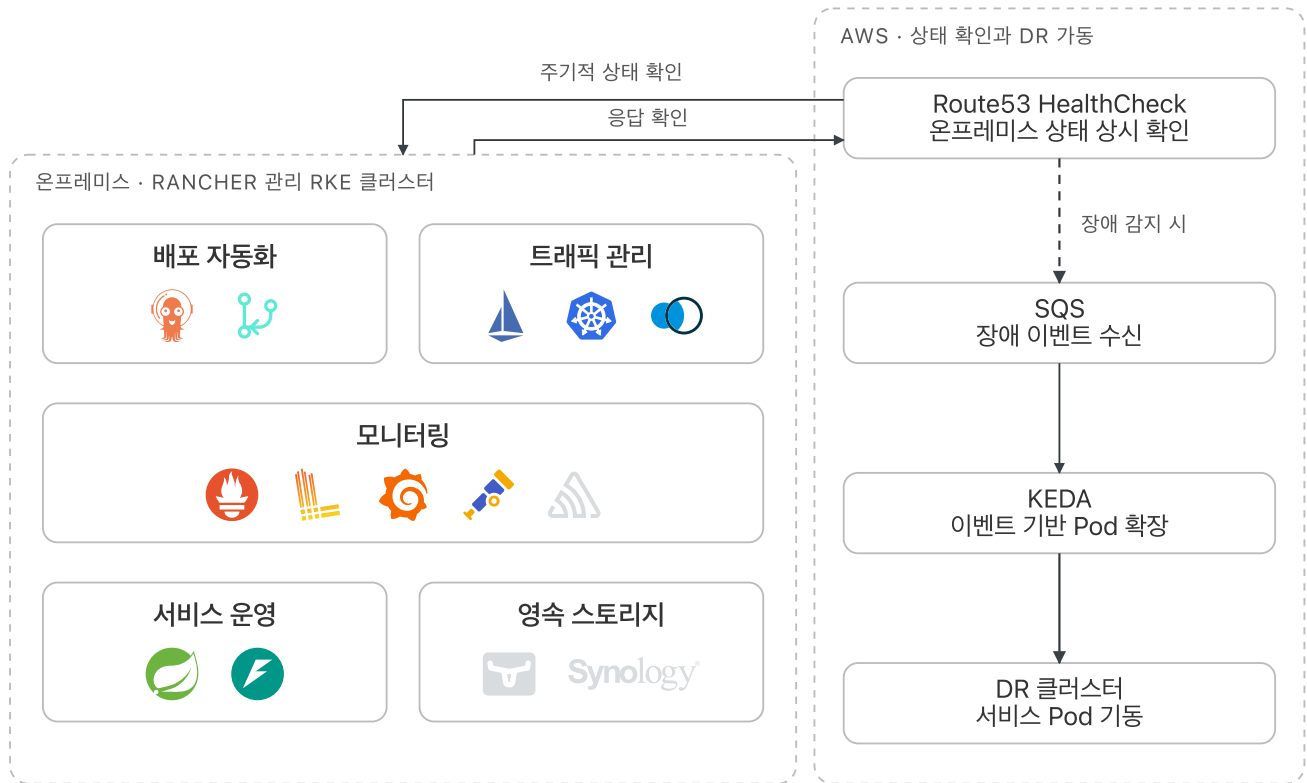
자동

DR Pod 기동

2배

이슈 대응 속도

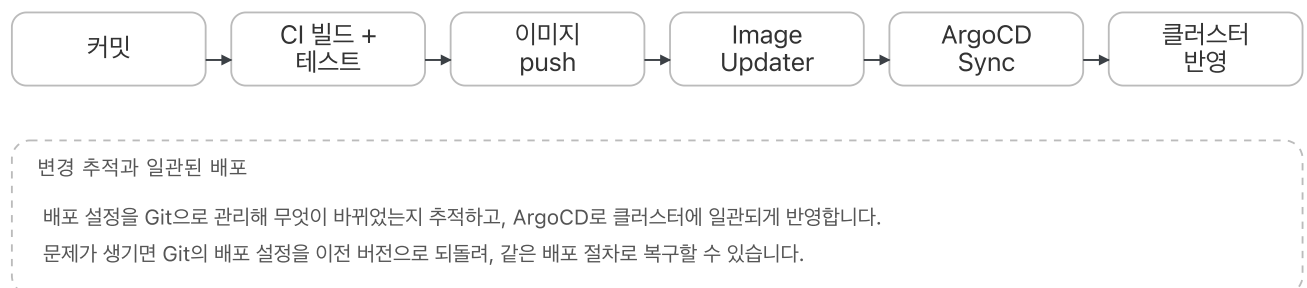
## 도식 1 온프레미스 장애가 나면 DR은 어떻게 가동되는가?



Route53 HealthCheck가 온프레미스 상태를 주기적으로 확인합니다.

장애가 감지되면 SQS 이벤트를 받아 KEDA가 DR 클러스터의 서비스 Pod를 기동합니다.

## 도식 2 배포는 어떤 순서로 흐르는가?



Git을 배포 상태의 기준으로 삼아 변경 이력 추적과 반복 가능한 배포·복구가 가능해졌습니다.

배포 완료·상태 알림은 Teams에 남습니다.

Kubernetes

RKE

Rancher

ArgoCD

Istio

Gateway API

KEDA

Longhorn

AWS

비용 최적화

## 04 라이선스 서버 재설계

MedicalIP 2025.08 - 2025.10 · 약 3개월 설계 · 개발

라이선스 정책을 DB로 분리해 배포 없이 조정하고, 정책 조회 시간을 800ms에서 400ms로 줄였습니다.

|    |                                                                                                                                                                                   |
|----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 문제 | 상품별 라이선스 제한 정책이 코드에 흩어져 있어 정책 하나를 추가할 때도 수정과 배포가 필요했고, 실행 순서와 실패 기준까지 섞여 같은 라이선스인데 사용자마다 제한이 다르게 적용되는 문제가 생겼습니다.                                                                  |
| 판단 | 룰 엔진으로 정책을 외부화하는 방안도 검토했지만, 학습 비용과 운영 주체를 고려하면 정책 규모에 비해 과한 구성이라 정책 데이터는 DB에, 실행 흐름은 코드에 두는 방식을 택했습니다. 운영자가 정책을 읽고 바꾸기 쉬운 단순한 구조를 우선한 선택이었습니다.                                    |
| 실행 | 상품별 활성 정책과 실행 순서를 DB로 옮긴 뒤, 정책 타입별 검증기는 Strategy로 분리하고 공통 실행 순서와 실패 처리는 Template Method로 고정했습니다. 정책이 하나라도 실패하면 403을 반환하도록 하고, Unit·Integration·E2E 테스트로 핵심 API 커버리지를 80%까지 올렸습니다. |
| 결과 | 데이터 변경만으로 정책 조합을 조정하고 신규 정책은 검증기 클래스 하나로 추가할 수 있게 되었으며, DB 설계도 함께 개편해 평균 조회 시간을 800ms에서 400ms로 줄였습니다.                                                                             |

800ms → 400ms

정책 조회

80%

핵심 API 커버리지

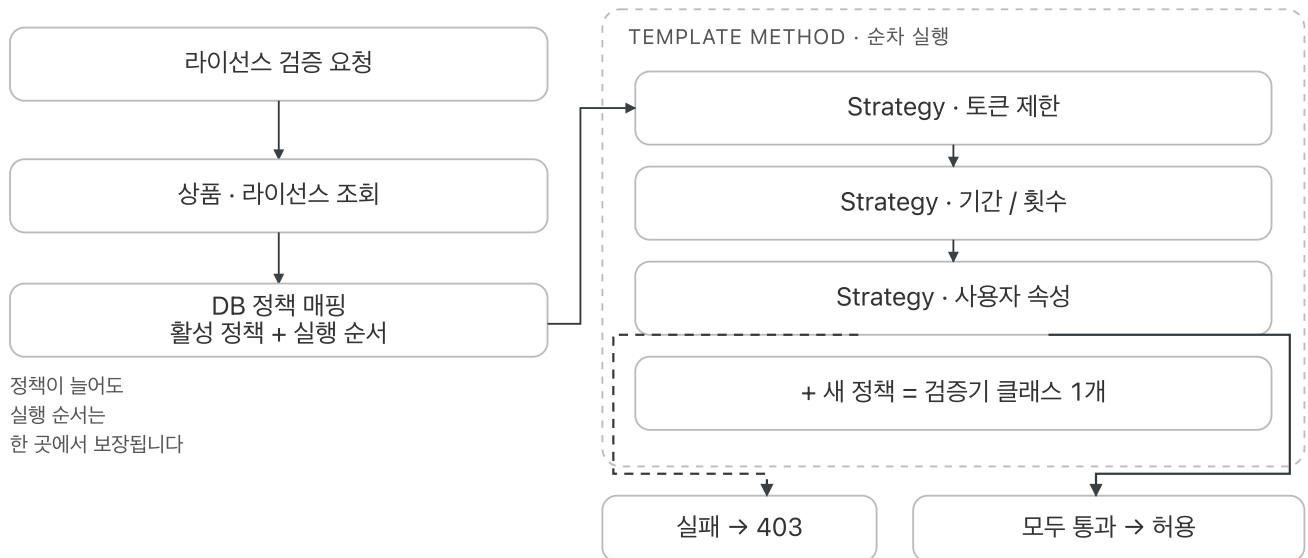
클래스 1개

신규 정책 추가 비용

403

실패 시 즉시 차단

도식 1 어디까지 코드가 하고, 어디부터 데이터가 하는가?



운영자가 바꾸는 영역(정책 데이터)과 개발자가 책임지는 영역(실행 순서 · 실패 처리)을 분리했습니다.

Kotlin

Spring

JPA

MariaDB

Junit5

Strategy Pattern

## 05 NDRS · 재해 복구 시스템

Nemosys 2024.11 - 2025.01 · 약 2개월 설계 · 개발

장애 감지와 복구를 자동화해 복구 시간을 10분에서 3분 이내로 줄였습니다.

|    |                                                                                                                                                                                                                                       |
|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 문제 | 온프레미스 메인 서버가 불안정한 전원 공급으로 간헐적으로 중단되는 데다 복구도 사람이 판단해 처리하다 보니, 대응자가 자리에 있는지에 따라 복구 시간이 달라졌습니다.                                                                                                                                          |
| 판단 | 월 100달러 미만의 비용과 3분 이내 복구를 설계 기준으로 삼아, 온프레미스 운영을 유지하면서 최소 클라우드 자원으로 Active-Standby를 구성하고 감지와 전환을 자동화했습니다.                                                                                                                              |
| 실행 | DB를 AWS EC2 기반 Active-Standby로 구성하고, Route53 HealthCheck와 CloudWatch 이벤트로 이상을 감지하면 EventBridge가 이벤트를 발행하도록 연결했습니다. 이 이벤트를 받은 Lambda와 Step Functions가 중지된 인스턴스를 복구하고 Lambda가 DNS IP를 바꾸도록 했으며, 복구 경로는 코드로 관리하면서 주기적으로 검증하는 절차를 마련했습니다. |
| 결과 | 장애 감지부터 인스턴스 복구와 DNS 전환까지 자동화해, 복구 시간을 10분에서 3분 이내로 줄이고(RTA 70% 감소) 월 100달러 미만으로 재해 복구 체계를 갖췄습니다.                                                                                                                                     |

10분 → 3분

복구 시간

70%↓

RTA 감소

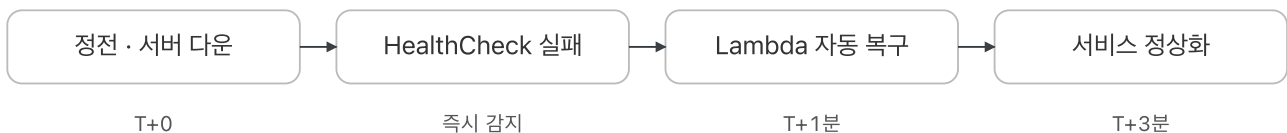
< \$100

월 비용

Active-Standby

DB 구성

도식 1 장애 복구는 몇 분 안에, 누가 하는가?



기존: 사람이 인지하고 수동으로 복구 (평균 10분)

기존 10분

개선 3분 — RTA 70% 감소 · 월 \$100 미만

장애 감지부터 복구까지 자동화해 평균 복구 시간을 10분에서 3분으로 줄였습니다.

AWS

Route53

EventBridge

Lambda

Step Functions

## 06 테스트 코드 문화 도입 및 CI 파이프라인

Nemosys 2024.12 - 2025.01 · 약 2개월 주도 · 세미나

팀 내 테스트 기준을 정립하고, MR·Merge 후 단계별 자동 검증을 CI에 도입했습니다.

|    |                                                                                                                                                                                                                      |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 문제 | 테스트 코드가 없어 기능 하나를 수정하면 다른 기능이 깨졌고,<br>수동 테스트 부담으로 새 기능을 추가하는 비용도 계속 늘어나 배포할 때마다 품질을 장담하기 어려웠습니다.                                                                                                                     |
| 판단 | 팀이 테스트를 개발 과정에 활용할 수 있도록 세미나를 주도해<br>E2E·Integration·Unit의 역할과 회귀 오류 예방 사례를 공유하고,<br>공통 테스트 기준을 정립한 뒤 격리 환경과 CI를 구성했습니다.                                                                                             |
| 실행 | TestContainer와 테스트용 DB로 격리된 환경을 구성하고,<br>GitLab Runner와 Jenkins의 CI를 Merge Request와 Merge 후 단계로 나눴습니다.<br>자주 실행하는 Merge Request 단계는 Unit 위주로 빠르게 확인하고,<br>Merge 후에는 운영과 유사한 환경에서 Unit과 Integration을 함께 실행하도록 구성했습니다. |
| 결과 | MR 단계의 빠른 피드백과 Merge 후 통합 테스트로 변경 영향을 확인하면서,<br>테스트를 코드 리뷰와 배포 전 검증 절차에 정착시켰습니다.<br>메인 솔루션 커버리지는 60%를 목표로 확대하고 있습니다.                                                                                                 |

60%

커버리지 목표

2단계

MR / Merge 후

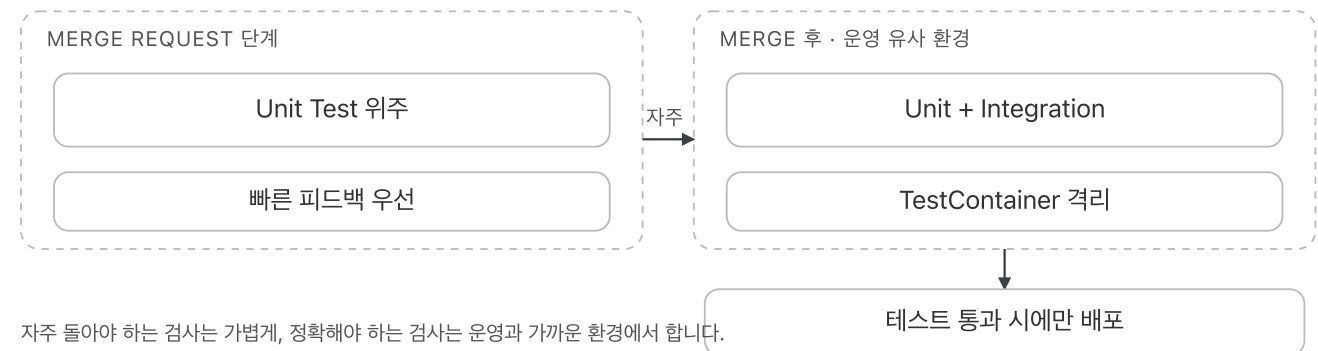
TestContainer

격리 환경

통과 시에만

배포 조건

도식 1 테스트는 언제, 얼마나 돌리는가?



MR에서 무거운 테스트를 돌리면 결국 건너뛰게 됩니다.  
단계를 나눈 이유가 여기 있습니다.

JUnit5

TestContainer

GitLab CI

Jenkins

## 07 철도시설 성능평가 통합 시스템

한국전기철도협회 2023.04 - 현재 개발 · 운영

분석 쿼리를 33,600ms에서 444ms로 개선하고, 문서 생성과 무중단 배포 체계를 구축했습니다.

|    |                                                                                                                                                                                                                                                                   |
|----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 문제 | 모든 철도운영사가 법령에 따라 같은 기준으로 평가받는 시스템으로,<br>평가 결과를 PPT·Excel·PDF·한글로 내보내기 위해 수백 장의 문서를 생성해야 했습니다.<br>분석 API도 30초 이상 걸려 <b>사용자는 화면이 멈춘 줄 알았습니다.</b>                                                                                                                    |
| 판단 | 문서 생성과 조회 성능을 함께 개선하기 위해 서버를 늘리기보다 쿼리의 실행 계획을 먼저 확인했습니다.<br>인덱스를 쓸 수 없는 형태의 쿼리가 원인임을 확인해 수정 방향을 잡고,<br>사용 빈도가 높은 데이터는 캐싱으로 읽기 자체를 줄이기로 했습니다.                                                                                                                      |
| 실행 | ClipReport 5.0으로 양식을 만들고 데이터를 매핑해 여러 포맷으로 내보내는 한편,<br>쿼리는 서브쿼리와 WHERE 절의 함수를 걷어내고 인덱스를 설계했습니다.<br>운영 환경은 온프레미스에서 AWS의 EC2 · AutoScaling · CloudWatch로 이관하고,<br>Redis Look-Aside 캐싱과 Spring Session Data Redis로 세션을<br>유지하면서 Nginx와 Docker로 Blue/Green 배포를 구성했습니다. |
| 결과 | 33,600ms 걸리던 쿼리를 444ms로 줄이고(약 75배) API 응답시간도 평균 30ms 개선했으며,<br>수백 장의 문서를 5~30초 안에 생성하고 배포 중에도 서비스를 유지할 수 있게 됐습니다.                                                                                                                                                 |

33,600ms → 444ms

분석 쿼리 · 약 75배

5~30초

수백 장 문서 생성

30ms ↓

평균 API 응답

Blue/Green

무중단 배포

**도식 1** 같은 질문인데 왜 33초와 0.4초로 같았는가?

AS-IS · 33,600MS

WHERE 절에 CASE · SUBSTRING  
서브쿼리 반복

Seq Scan · 1,200,000 rows

33,600ms

TO-BE · 444MS

인덱스가 동작하는 형태로  
쿼리 재작성

Index Scan · 43 rows

444ms — 읽는 행 수를 1.2M에서 43으로 줄였습니다

같은 데이터에 대한 쿼리를 재작성하고 인덱스를 적용해 조회 시간을 33,600ms에서 444ms로 줄였습니다.

AWS

Redis

쿼리 튜닝

Blue/Green

ClipReport

Nginx

## 08 피플·헌혈 매칭 플랫폼

피플 2022.02 - 2023.11 창업 · 개발 총괄 (BE · App)

환자 사연과 헌혈 참여를 연결하는 앱을 개발해, 700건 이상의 사연에 2,000건의 헌혈 응답을 모았습니다.

|    |                                                                                                                                          |
|----|------------------------------------------------------------------------------------------------------------------------------------------|
| 문제 | 코로나 시기에 혈액 수급이 무너지면서 수혈이 필요한 환자와 가족이 헌혈자를 직접 구해야 했지만, 지정헌혈은 커뮤니티 연결에 의존했고 헌혈자도 자신의 헌혈이 어디에 필요한지 알기 어려웠습니다.                               |
| 판단 | 환자 사연 등록·헌혈 응답을 중심으로 환자는 필요한 헌혈자를 찾고 헌혈자는 참여할 사연을 선택하도록 설계하고, 헌혈증 인증과 리워드를 더해 반복 참여를 유도했습니다.                                             |
| 실행 | 환자 사연은 3일 후 자동 삭제되 응답이 있으면 연장되도록 구성하고, 지정헌혈 응답·헌혈증 인증·캘린더·배지·포인트 샵을 구현했습니다.<br>백엔드와 앱을 직접 개발하면서 AWS 운영 비용도 함께 관리했습니다.                    |
| 결과 | 2024년 11월 기준 5,758명이 앱을 사용했으며,<br>헌혈 참여 3,736회와 지정헌혈 참여 1,250회가 이어졌습니다.<br>700건이 넘는 사연에 2,000건의 헌혈 응답이 모였고,<br>2023년 예비사회적 기업으로 선정되었습니다. |

5,758명

사용자 (2024.11)

3,736회

헌혈 참여

1,250회

지정헌혈 참여

2023

예비사회적 기업 선정

도식 1 헌혈은 어떤 순서로 이어지는가?



700건이 넘는 사연에 2,000건의 응답이 모였습니다. 사연은 3일 뒤 자동으로 내려가고, 응답이 이어지면 연장됩니다.

재고를 채우는 시스템이 아니라, 사연과 응답이 오가는 흐름을 만드는 일이었습니다.

헌혈 매칭

Flutter

Spring Boot

AWS

## 09 Mongle · 이미지 업로드·변환 파이프라인

6인 팀 · 사이드 프로젝트 2024.04 - 2024.05 BE 개발 총괄 · 기여도 90%

이미지 업로드·변환·캐시 갱신을 구현하고, 이미지 응답 시간을 60ms에서 20ms로 줄였습니다.

|    |                                                                                                                                                                                                                                                                                                   |
|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 문제 | 이전 프로젝트에서 큰 원본 이미지를 그대로 제공해 로딩이 느렸고,<br>Mongle에서도 화면마다 필요한 이미지 크기가 달랐습니다.<br>백엔드가 파일을 받아 S3로 다시 보내면서 전송 시간과 서버 자원 부담이 커졌고,<br>이미지를 교체해도 CDN에 이전 이미지가 남는 문제를 함께 다뤄야 했습니다.                                                                                                                          |
| 판단 | 업로드 시 여러 크기를 미리 저장하는 방식과 요청 시 변환하는 방식을 비교한 뒤,<br>다양한 화면 크기에 대응하기 위해 원본은 S3에 저장하고 요청<br>파라미터에 맞춰 변환한 결과를 캐싱하는 방식을 선택했습니다.                                                                                                                                                                          |
| 실행 | Presigned URL을 발급해 클라이언트가 S3로 직접 업로드하도록 하고,<br>CloudFront와 Lambda@Edge·Sharp로 이미지 조회 시 너비·높이·포맷·품질을 조절했습니다.<br>S3 변경 이벤트로 Lambda를 호출해 해당 이미지 경로의 캐시를 무효화하고,<br>업로드 완료 이벤트는 EventBridge·SQS를 통해 NestJS 소비자로 전달했습니다.<br>소비자는 이벤트를 검증하고 이미지 정보를 DB에 저장한 뒤 메시지를 삭제하도록 해,<br>저장 실패 시 재처리할 수 있게 구성했습니다. |
| 결과 | 이미지 리사이징·CDN 적용 전후 비교에서<br>이미지 응답 시간은 60ms에서 20ms로, 콘텐츠 다운로드 시간은 2.1ms에서 0.8ms로 줄었습니다.<br>백엔드는 파일 중계 대신 URL 발급과 업로드 정보 기록을 맡도록 하고,<br>요청별 이미지 변환·결과 캐싱·교체 시 캐시 무효화를 하나의 처리 흐름으로 구현했습니다.                                                                                                           |

[구현 기록 ↗](#)[성능 비교 ↗](#)

60ms → 20ms

이미지 응답 · 전후 비교

2.1ms → 0.8ms

콘텐츠 다운로드 · 전후 비교

S3 직접 업로드

Presigned URL · 파일 중계 제거

DB 저장 후 삭제

SQS 메시지 처리 순서

도식 1 파일을 중계하지 않고 업로드 완료를 어떻게 기록하는가?

업로드 · 완료 이벤트



파일은 클라이언트 → S3로 전송합니다. 소비자는 이벤트 검증 → DB 저장 → 메시지 삭제 순서로 처리합니다.

이미지 교체 · 캐시 갱신



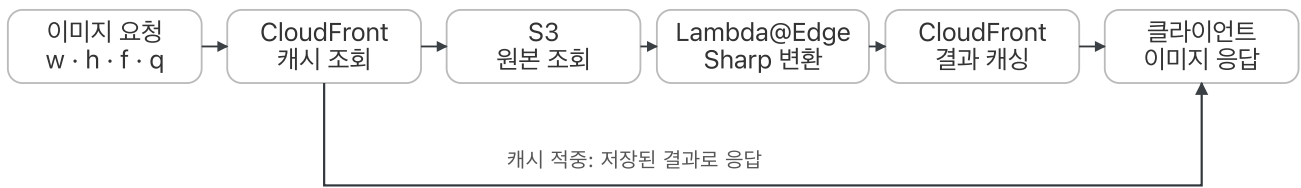
변경된 경로의 캐시를 무효화해, 이후 조회에서 갱신된 원본을 가져오도록 합니다.

파일 전송과 완료 정보 기록을 분리했습니다.

업로드 후 DB 반영은 이벤트 소비자가 맡고, 이미지 교체는 캐시 갱신 흐름으로 처리합니다.

## 도식 2 같은 원본을 화면마다 다른 크기로 어떻게 제공하는가?

캐시 미적중: 원본 조회 → 요청별 변환 → 캐싱



첫 요청은 이미지 변환 시간이 들고, 같은 조건의 후속 요청은 캐시된 결과를 재사용합니다.

크기별 이미지를 미리 생성하는 대신, 다양한 화면 크기에 맞춘 요청 시 변환을 선택했습니다.

캐시 미적중과 적중 경로를 나눴습니다.

Sharp 변환은 Lambda@Edge의 원본 응답 단계에서 수행하고, 결과를 CloudFront에서 캐싱합니다.

S3

Presigned URL

CloudFront

Lambda@Edge

Sharp

EventBridge

SQS

NestJS

배포·장애 복구·쿼리 개선·권한 관리를 팀 공통 절차로 표준화했습니다.

### 새 서비스 배포 온보딩

ArgoCD App of Apps

- 01 이미지를 빌드해 레지스트리에 푸시되는지 CI 로그로 확인한다.
- 02 App of Apps에 애플리케이션 매니페스트를 추가한다 (환경별 values 분리).
- 03 Image Updater가 새 태그를 감지하도록 규칙을 확인한다.
- 04 Sync 후 Pod 상태와 ReplicaSet이 기대값인지 확인한다.
- 05 HealthCheck · 알림 · 로그 수집이 붙었는지 확인하고 Teams 알림을 검증한다.

### DR 발동과 복구 검증

Route53 → SQS → KEDA

- 01 Route53 HealthCheck 실패 알림을 확인한다.
- 02 SQS 이벤트가 발행되었고 KEDA가 DR 클러스터의 Pod를 올렸는지 확인한다.
- 03 DB Replication 상태와 복구 시점(RPO)을 확인한다.
- 04 트래픽이 정상 리전으로 돌아왔는지, 지연이 기준 이내인지 측정한다.
- 05 실제 복구 시간을 기록하고 3분 목표를 넘겼다면 원인을 남긴다.

### 계정 · 권한 발급과 회수

IdP 계정 · Keycloak 권한

- 01 외부 IdP 계정 상태와 Keycloak에 연동된 계정의 소속·역할 속성을 확인한다.
- 02 접근이 필요한 시스템에 속성 기반으로 권한을 매핑한다.
- 03 로그인 및 주요 접근에 MFA가 요구되는지 확인한다.
- 04 퇴사·이동 시 외부 IdP 계정 상태와 연동 시스템의 권한·세션을 점검하고 잔여 접근이 없는지 확인한다.
- 05 변경 이력을 남겨 입사·퇴사 시 누락이 생기지 않게 한다.

### 배포와 롤백

변경 이력은 Git에

- 01 헤더 기반 카나리 배포로 구성된 beta(스테이징) 환경에서 검증하고, Kiali에서 트래픽 흐름을 확인한다.
- 02 에러율·지연이 기준을 넘으면 Git에서 이전 revision으로 되돌린다 (수동 kubectl 금지).
- 03 롤백 후에도 알림과 메트릭이 정상인지 확인한다.
- 04 배포 기록은 Git 커밋과 ArgoCD 이력으로 남기고, 무엇이 문제였는지 한 줄로 기록한다.

### 슬로우 쿼리 대응

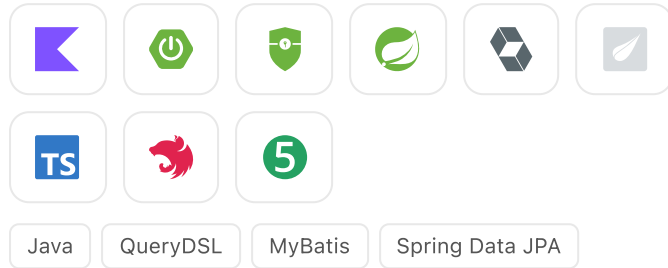
실행 계획부터

- 01 대상 쿼리의 실행 계획을 확인한다 (full scan 여부, rows 추정치).
- 02 WHERE 절의 함수·CASE·SUBSTRING과 불필요한 서브쿼리를 제거해 인덱스가 동작하는 형태로 바꾼다.
- 03 필요한 인덱스를 설계·적용하고 실행 계획을 다시 비교한다.
- 04 반복 조회되는 데이터는 Look-Aside 캐싱 대상인지 검토한다.
- 05 개선 전후 시간을 기록한다 (예: 33,600ms → 444ms).

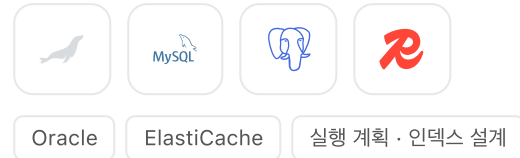
강조된 항목은 실무에서 설계와 운영까지 맡아본 것입니다.

**BACKEND**

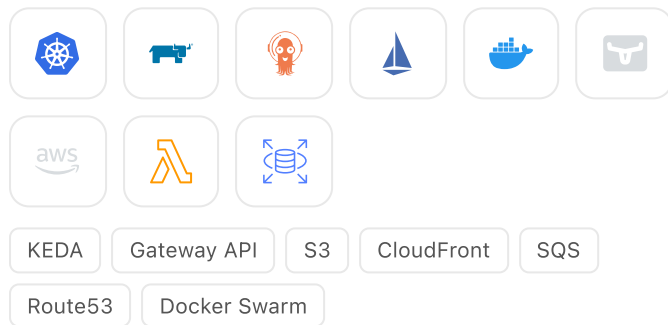
설계 · 운영까지

**DATA**

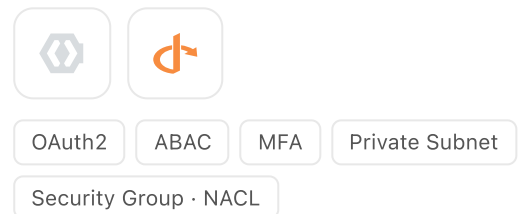
운영 · 튜닝

**INFRA · CLOUD**

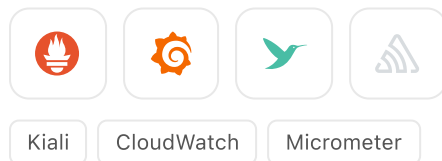
구축 · 운영

**AUTH · SECURITY**

설계 · 구축

**OBSERVABILITY**

구축 · 운영

**DELIVERY**

자동화

