TECHNICAL PORTFOLIO · SELECTED

김현욱 기술 포트폴리오

지원 직무와 관련된 작업 8건을 한 문서로 묶었습니다. 각 항목은 문제 상황과 판단, 조치, 확인된 결과 순으로 정리했고, 항목마다 상세 페이지 URL을 함께 실었습니다.

Data Engineer | 수록 항목 8개 | portfolio.hwlabs.dev | hyunwook711@naver.com | github.com/hyunwook711

구성
지원 직무 관련 항목 선별

내용
문제 → 판단 → 조치 → 결과

링크
항목별 상세 페이지 URL 포함

포함 항목

01
OS 패키지 취약점 수집 정확도 개선
2022 · 래브라도랩스(LabradorLabs)
portfolio.hwlabs.dev/items/os-vuln-collection-accuracy.html

02
라이브러리 컴포넌트 테이블 재설계 (DB 부하 개선)
2026 · 래브라도랩스(LabradorLabs)
portfolio.hwlabs.dev/items/library-table-redesign.html

03
DB 인덱스 최적화 & 용량 53% 절감
2025 · 래브라도랩스(LabradorLabs)
portfolio.hwlabs.dev/items/db-index-optimization.html

04
ETL 데이터 수집 플로우 재정립
2026 (3월~ 진행) · 래브라도랩스(LabradorLabs)
portfolio.hwlabs.dev/items/etl-flow-redesign.html

05
K8s · Airflow 데이터 수집 플랫폼 운영 & 장애 대응
2024–2026 · 래브라도랩스(LabradorLabs)
portfolio.hwlabs.dev/items/infra-k8s-airflow-ops.html

06
크롤러 DB 부하를 줄인 DML Broker 설계
2026 · 래브라도랩스(LabradorLabs)
portfolio.hwlabs.dev/items/dml-broker-db-stability.html

07
AWS → IDC/In-house 인프라 이전 & 클라우드 비용 절감
2024–2026 (AWS 자원 종료 ~2026-04) · 래브라도랩스(LabradorLabs)
portfolio.hwlabs.dev/items/aws-to-idc-migration.html

08
Grafana 기반 데이터·DB 모니터링 구축
2024–2025 · 래브라도랩스(LabradorLabs)
portfolio.hwlabs.dev/items/grafana-monitoring.html

01 · SELECTED PORTFOLIO

OS 패키지 취약점 수집 정확도 개선

12종 주요 OS 배포판의 보안 권고안을 소스 특성에 맞게 파싱해, 패치 버전까지 정확히 담은 취약점 데이터를 수집

기간	소속	역할
2022	래브라도랩스(LabradorLabs)	데이터 엔지니어 · 수집 정확도 개선 주도

`Vulnerability` `OS Package` `OVAL` `Security Tracker` `Crawler` `Data Pipeline`

작업자의 메모

OS 취약점 데이터는 CVE ID만 맞아도 되는 문제가 아닙니다. 실제 고객사 검증에서는 어떤 패키지 버전이 패치됐는지까지 맞아야 합니다.

12종 배포판의 API, OVAL, Security Tracker를 따로 다룬 이유는 그 정확도를 확보하기 위해서였습니다. 이 작업은 데이터 수집이라기보다 오탐을 줄이는 기반 작업이었습니다.

배경

클라우드 네이티브가 빠르게 확산되면서 **OS 패키지 취약점 정보의 정확성** 요구가 커졌습니다. 그러나 기존 크롤러는 각 OS 배포판이 제공하는 보안 권고안을 정확히 파싱하지 못해 **패치된 버전 정보가 누락되거나 부정확**한 문제가 있었습니다.

- "취약하다/아니다"만이 아니라 **어느 버전에서 고쳐졌는지(패치 버전)**가 정확해야 스캐너가 신뢰할 만한 판정을 내릴 수 있음
- 배포판마다 보안 데이터 제공 방식이 달라(**API · OVAL · Security Tracker**), 단일 파싱 방식으로는 정확도 확보가 어려움

역할

데이터 엔지니어로서 **소스 분석 → 파서 재설계 → 패치 버전 추출 → 신뢰도 검증**의 전 과정을 직접 수행했습니다. 수집 방안은 팀 리뷰를 거쳐 확정했고, 12종 OS의 서로 다른 보안 데이터 제공 방식을 각각 분석하고 소스에 가장 맞는 수집 방식(예: Ubuntu의 Git 직접 클론 파싱)을 선택해 데이터 정확성과 신뢰도를 끌어올렸습니다.

데이터 수집 파이프라인의 정확도·신뢰도 개선에 초점을 둔 작업.

접근

12종 주요 OS 배포판(RedHat · Debian · Ubuntu 등)이 보안 데이터를 어떻게 제공하는지 **심층 분석**한 뒤, 소스 특성에 맞춰 크롤러를 수정·재개발했습니다.

① 소스 분석 — 12종 OS의 제공 방식(API/OVAL/Tracker) 파악 → **② 파서 재설계** — 소스별 특성에 맞춰 크롤러 수정·재개발 → **③ 패치버전 추출** — 패치 버전 포함 정확한 데이터 수집 → **④ 신뢰도 검증** — 스캐너 분석 결과 정확성 확인

대표 사례로 **Ubuntu**는 기존의 웹 페이지 파싱 방식에서 **Git 리포지토리를 직접 클론해 파싱**하는 방식으로 변경해 데이터 신뢰도를 높였습니다.

다종 보안 소스 — API · OVAL · Security Tracker · Git → **소스별 파서** — 12종 OS 맞춤 수집기 → **정규화** — 패치 버전 포함 취약점 데이터 → **취약점 스캐너** — 신뢰도 높은 판정

결과

커버리지

12종 배포판 × 4가지 제공 방식 (API·OVAL·Tracker·Git)을 소스별 파서로 흡수

수집 방식 전환

Ubuntu는 웹 페이지 파싱에서 Git 저장소 클론 파싱으로 바꿔 패치 버전 누락을 제거

검증

수집한 패치 버전이 스캐너 판정에 그대로 쓰이므로, 스캐너 분석 결과로 정확성을 확인

웹 상세: https://portfolio.hwlabs.dev/items/os-vuln-collection-accuracy.html

02 · SELECTED PORTFOLIO

라이브러리 컴포넌트 테이블 재설계 (DB 부하 개선)

고객사 환경의 중앙 운영 DB CPU 이상을 슬로우 쿼리와 실행계획으로 추적해, 라이브러리 컴포넌트 테이블의 구조·식별자·데이터 정합성 문제를 근본 원인부터 재설계

기간	소속	역할
2026	래브라도랩스(LabradorLabs)	데이터 엔지니어

`MySQL` `EXPLAIN` `Collation` `Schema` `Index` `JSON` `varchar` `npm / PyPI / Maven`

작업자의 메모

CPU 부하가 보이면 서버를 키우고 싶은 유혹이 있지만, 슬로우 쿼리와 실행계획을 따라가 보니 문제는 테이블 구조와 식별자 정책에 있었습니다. 부하가 데이터 정합성과 스키마 문제에서 비롯된 경우였습니다.

라이브러리 컴포넌트 테이블은 제품의 여러 기능이 기대는 중심 테이블이라, 단순 튜닝보다 데이터 모델을 다시 보는 쪽이 더 오래 가는 해결책이었습니다.

배경

고객사 환경에서 **중앙 운영 DB의 CPU 이상(부하)**이 관측됐습니다. 슬로우 쿼리에서 시작해 실행계획으로 좁혀 보니, 다수 패키지 생태계의 컴포넌트 정보를 담는 **라이브러리 컴포넌트 테이블의 구조**가 부하·정합성 문제의 근원이었습니다.

- **collation 불일치 조인**: 같은 값을 저장하는 두 테이블의 collation이 달라(한쪽만 대소문자 구별) 뷰를 통한 조인은 인덱스를 타지 못하고 **약 130만 건을 전체 스캔**하는데, 테이블을 직접 조인하면 인덱스를 타고 **약 6,000건**에 그친다는 것을 실행계획으로 확인
- 문자열(`varchar`) 타입 컬럼에 **`JSON`을 그대로 저장**하던 케이스(예: 라이선스 컬럼)가 존재 — 조회·가공 시 비효율과 데이터 깨짐을 유발
- **식별자 문제**: 서로 다른 패키지가 같은 키로 충돌하거나, 대소문자 차이로 동일 컴포넌트가 누락되는 케이스가 발생

역할

데이터 엔지니어로서 **부하 원인 추적 → 진단 → 전수조사 → 스키마·인덱스·식별자 재설계**를 주도했습니다. 원인 추적과 전수조사, 구현은 직접 수행했고, 여러 기능이 기대는 중심 테이블이라 재설계 방안은 여러 안을 만들어 팀장·CTO 리뷰를 거쳐 확정했습니다. 증상(CPU 부하)에 대증하지 않고 구조적 결함을 근본 원인까지 추적해 성능과 데이터 정합성을 함께 개선했습니다.

collation 통일, 식별자 정책 정비와 스키마·인덱스 재설계가 핵심 변경점. varchar-JSON 컬럼의 타입 정규화는 이 조사에서 발견해 별도 작업으로 진행.

접근

① 추적 슬로우 쿼리 확인 → 실행계획으로 원인 쿼리 분석 → **② 진단** collation 불일치로 뷰 조인이 전체 스캔되는 것을 확인 → **③ 전수조사** varchar-JSON 저장·식별자 충돌 케이스를 전체 테이블 대상으로 조사 → **④ 재설계** 스키마·인덱스·식별자 정책 재설계

슬로우 쿼리에서 원인 쿼리를 찾아 실행계획을 비교했습니다. 뷰를 통한 조인은 collation 차이 때문에 인덱스를 타지 못해 약 130만 건을 스캔했고, 같은 조건으로 테이블을 직접 조인하면 약 6,000건에 그쳤습니다. 이어 `varchar`에 `JSON`을 저장하던 케이스를 **전체 테이블 대상 전수조사**해 영향 범위와 깨진 데이터 비율을 파악하고, 식별자 충돌·대소문자 누락 같은 구조적 문제를 정리한 뒤, 어긋나 있던 collation을 통일하고 **스키마·인덱스·식별자 정책을 재설계**해 부하와 정합성을 동시에 잡았습니다.

결과

근본 원인 분석
중앙 운영 DB CPU 부하의 원인을 라이브러리 테이블 구조로 규명

스캔 감소
뷰 조인 약 130만 건 전체 스캔 → 테이블 직접 조인 약 6,000건 (실행계획 기준)

정합성 개선
식별자 정책 정비로 충돌·누락을 줄이고 데이터 정합성 향상

웹 상세: https://portfolio.hwlabs.dev/items/library-table-redesign.html

03 · SELECTED PORTFOLIO

DB 인덱스 최적화 & 용량 53% 절감

수년간 운영되며 비대해진 수집 DB와 고객사 배포용 DB의 인덱스를, 사용 주체별 쿼리 전수 조사를 근거로 걷어내고 SQL을 튜닝해 용량과 성능을 동시에 개선

기간	소속	역할
2025	래브라도랩스(LabradorLabs)	데이터 엔지니어 · 인덱스 진단 및 최적화 주도

`MySQL` `EXPLAIN` `Index Tuning` `SQL Optimization` `Query Analysis`

작업자의 메모

이 작업에서 제일 조심한 부분은 "인덱스를 많이 지웠다"가 아니라 "지워도 되는지 증명했다"였습니다. 운영 DB에서 감으로 인덱스를 건드리는 순간 성능 문제는 더 커질 수 있기 때문입니다.

수집 DB는 크롤러가 실행하는 쿼리를, 배포용 DB는 분석 엔진이 실행하는 쿼리 95개를 전수로 모아 EXPLAIN으로 사용 여부를 확인한 뒤에야 삭제와 재구성을 진행했습니다. 줄어든 용량보다, 그 판단 과정을 남긴 것이 더 중요했습니다.

배경

수년간 운영되어 온 수집 DB와 고객사 배포용 DB는 누적된 **미사용 인덱스**와 **비효율 쿼리**로 저장 용량이 비대해지고 전반적인 성능이 저하된 상태였습니다. 두 DB는 사용 주체가 달라(크롤러 / 분석 엔진) 정리 기준도 나눠서 세웠습니다.

- 인덱스는 한 번 만들면 계속 쌓이기만 할 뿐, 정말 쓰이는지 검증된 적이 없었음
- 불필요한 인덱스는 **저장 공간 낭비**와 함께 쓰기 성능·DB 초기 구성 시간에 부담을 가중
- 감(感)이 아니라 **실제 사용 데이터**에 근거한 정리 기준이 필요

역할

데이터 엔지니어로서 **쿼리 수집 → 실행 계획 분석 → 인덱스 정리 → SQL 튜닝**의 전 과정을 직접 수행했습니다. 운영 DB 변경이라 적용 범위와 순서는 팀 리뷰를 거쳤고, 운영 중인 중앙 운영 DB를 다루는 만큼, 실제 사용 패턴을 근거로 안전하게 변경 범위를 좁혀가며 진행했습니다.

결과적으로 용량·비용·성능을 동시에 개선했고 이후 정기적인 인덱스 점검 기준으로 남겼습니다.

접근

① 수집 사용 주체별 쿼리 전수 확보 → **② 분석** EXPLAIN으로 인덱스 사용 여부 검증 → **③ 정리** 불필요 인덱스 삭제·복합 인덱스 재구성 → **④ 튜닝** 쿼리 재작성으로 인덱스를 타게 재작성

- 배포용 DB는 분석 엔진이 실행하는 쿼리 **95개**를 `EXPLAIN`으로 전수 분석해 인덱스 **311개 중 224개(72%)**가 실행 계획에서 전혀 쓰이지 않음을 확인
- 수집 DB는 크롤러가 실행하는 쿼리를 전수 조사해 미사용 인덱스와 더 이상 쓰지 않는 레거시 테이블을 정리
- 불필요 인덱스를 삭제하고 복합 인덱스의 **컬럼 순서**를 실제 실행 계획에 맞게 재구성
- SQL 튜닝: `OR` 조건을 `UNION`으로 전환하고 `LIKE` 와일드카드 패턴을 인덱스가 타도록 최적화

추측 대신 실측 실행 계획을 근거로 삭제와 재구성을 결정했습니다.

결과

수집 DB 9.6TB → 4TB
크롤러 쿼리 전수 조사로 미사용 인덱스·레거시 테이블 정리

배포용 DB 2.6TB → 1.2TB
분석 엔진 쿼리 95개 전수 분석으로 53.8% 절감

초기 설치 시간 절반
고객사 온프레미스 DB 생성 12시간 → 6시간

정리 기준 존치
실측 실행 계획 기반의 정기 인덱스 점검 기준으로 남김

웹 상세: https://portfolio.hwlabs.dev/items/db-index-optimization.html

04 · SELECTED PORTFOLIO

ETL 데이터 수집 플로우 재정립

재수집 때마다 1,000만~1억 건의 외부 API 요청이 필요하던 원테이크 수집을, RAW 보존·재파싱 구조의 오브젝트 스토리지 기반 파이프라인으로 재설계

기간	소속	역할
2026 (3월~ 진행)	래브라도랩스(LabradorLabs)	데이터 엔지니어 · 파이프라인 재정립 주도

`ETL` `ELT` `Data Pipeline` `Object Storage` `SeaweedFS` `DB / Infra`

작업자의 메모

원테이크 수집은 처음에는 단순하지만 재수집이 필요해지는 순간 외부 API 비용과 실패 위험이 그대로 다시 옵니다. 1,000만~1억 건 요청을 반복해야 하는 구조라면 파이프라인부터 바꾸는 편이 맞았습니다.

RAW를 남기고 재파싱할 수 있게 만든 것은 저장소를 하나 더 둔다는 뜻이 아니라, 장애와 품질 개선을 나중에 더 낮은 비용으로 처리할 수 있게 만든 결정이었습니다. 업계 용어로 치면 ETL에서 ELT로의 전환입니다 — 일단 원본을 적재해 두고, 변환은 필요할 때 다시 할 수 있게 만든 구조니까요.

배경

제품 데이터는 **수집(Gathering) → 정제(Curation) → 배포(Distribution)**로 이어지는 파이프라인으로 운영됩니다. 단계별로 처리 방식과 저장 위치가 제각각이라, 흐름을 한눈에 파악하기 어렵고 데이터 규모가 늘수록 확장에 부담이 있었습니다.

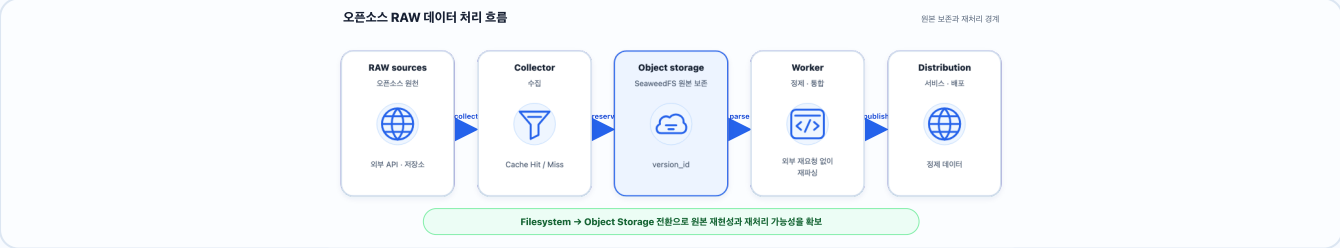
- RAW를 보존하지 않는 **원테이크 수집**(API 호출 → 즉시 파싱 → DB 저장) 구조라, 재수집·추가 수집 때마다 **매번 1,000만~1억 건의 외부 API 요청**이 필요했고 rate limit에 걸려 수집이 장기화
- 파싱 버그·이상 데이터 발견 시 원본(RAW)이 이미 사라져 **원인 추적·재현 불가** — 유일한 해결책이 전체 재수집
- 오픈소스 RAW 데이터가 늘어나면서 기존 **파일시스템 기반 저장**으로는 용량·확장성·관리에 한계
- 수집부터 배포까지의 흐름이 표준화되어 있지 않아 신규 데이터 소스 편입과 운영 인계가 어려움

역할

데이터 엔지니어로서 **파이프라인 재정의 → 스토리지 설계 → 마이그레이션 전략 수립**을 주도했습니다. 오브젝트 스토리지 방안은 세 가지 안을 비교 문서로 만들어 팀장·CTO 리뷰로 확정했고, 수집부터 배포까지 전체 흐름을 표준화하고 파일시스템에서 오브젝트 스토리지로 옮기는 로드맵을 세워 저장 구조를 단계적으로 옮기고 있습니다.

현재 진행 중인 항목으로, 단계적으로 마이그레이션을 적용하고 있습니다.

접근



수집과 정제를 분리하고 RAW 원본을 먼저 보존해 외부 재요청 없이 다시 파싱할 수 있게 설계했습니다.

파이프라인 전체를 다시 정의하면서 RAW 데이터 저장을 **파일시스템에서 오브젝트 스토리지(SeaweedFS)**로 옮기는 마이그레이션 전략과 로드맵을 수립했습니다. 단계별 책임과 데이터 흐름을 명확히 해 수집 플로우를 표준화하는 것이 핵심입니다.

- RAW 데이터를 오브젝트 스토리지에 **원본 보존**하고 상태 관리 DB로 수집(Collector)과 정제(Worker)를 분리 — 재처리는 외부 요청 없이 **재파싱**으로 해결
- Cache Hit/Miss 패턴(이미 있으면 즉시 반환)과 **version_id 기반 수정 이력 추적**으로 데이터 재현 가능성 확보
- 오픈소스 RAW 데이터 수집용 **오브젝트 스토리지 설계**
- 파일시스템 → 오브젝트 스토리지(SeaweedFS) **마이그레이션 전략·로드맵** 수립

결과

외부 요청 제거

재수집 때마다 반복되던 1,000만~1억 건의 API 호출이 재처리 시 0건으로 (RAW 재파싱)

재처리 시간 단축

rate limit에 묶여 수일씩 걸리던 기간 재수집을 내부 재파싱으로 대체 (설계 산정 예: 2일 → 5분)

추적·재현 가능

version_id 이력 추적과 원본 보존으로, 이상 데이터의 원인 규명·재현이 가능

수집 플로우 표준화

수집→정제→배포 책임 명확화, 신규 소스 편입·운영 인계 용이

웹 상세: https://portfolio.hwlabs.dev/items/etl-flow-redesign.html

05 · SELECTED PORTFOLIO

K8s · Airflow 데이터 수집 플랫폼 운영 & 장애 대응

데이터 수집 파이프라인을 Kubernetes 위 Airflow로 운영하며, 반복 장애를 근본 원인 분석으로 해소하고 안정성·가용성을 확보

기간	소속	역할
2024–2026	래브라도랩스(LabradorLabs)	데이터 엔지니어 · 인프라 운영

`Kubernetes` `Apache Airflow` `KubernetesExecutor` `Helm` `Git-Sync` `PostgreSQL` `WireGuard`

운영해야 했던 규모

인턴과 팀원이 만든 크롤러가 80개를 넘으면서, 각자 서버에서 실행하던 방식으로는 실행 시간과 자원 사용을 통제하기 어려웠습니다. 외부 API의 rate limit, DB 연결 수, CPU·메모리, 네트워크 대역폭을 한꺼번에 봐야 했습니다.

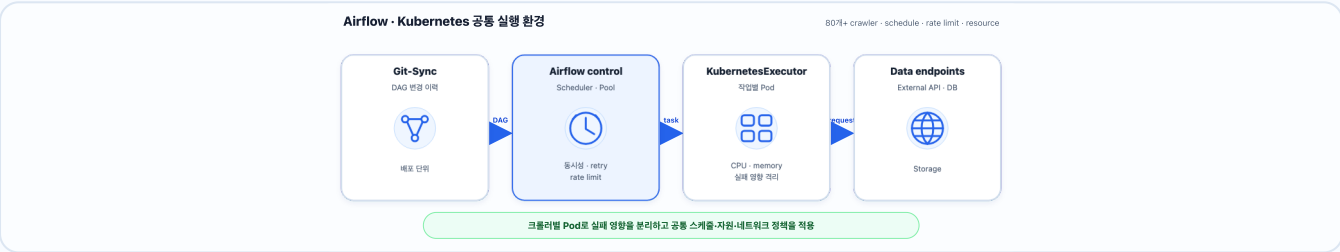
운영 중 해결한 문제

Pod가 생성되지 않거나 Init 단계에서 멈출 때 단순 재시작으로 넘기지 않았습니다. 노드 inode와 DiskPressure, CoreDNS, kube-proxy, etcd 응답 지연, Git-Sync 권한과 재시도 설정을 순서대로 확인했습니다.

토큰과 키 만료는 사전 알림을 달아 장애가 난 뒤 교체하는 일을 줄였습니다. 특정 서버에 작업이 몰리는 경우에는 스케줄과 자원 요청량을 다시 조정했습니다.

공통 실행 환경을 만든 방식

Airflow를 공통 스케줄러로 두고 KubernetesExecutor로 작업마다 Pod를 띄웠습니다. 크롤러별 실행 시간과 동시 실행 수를 나눴고, Airflow pool과 재시도 정책으로 외부 API 호출이 한 번에 몰리지 않도록 했습니다.



Git-Sync로 DAG를 배포하고 Airflow가 스케줄·동시성·재시도를 관리하며 KubernetesExecutor가 작업별 Pod를 생성했습니다.

- 수집 주기는 데이터의 시급성으로 결정 — 취약점 정보는 고객사가 빨리 대비할수록 가치가 있어 20분 주기로 수집하고 1시간 배치와 맞물려 약 1시간 안에 고객사까지 도달, 라이브러리 버전은 새로 나와도 기업이 바로 코드를 고치지 않으므로 4시간 주기로 완화
- 크롤러별 DAG 분리와 Git-Sync를 통한 변경 이력 관리
- CPU·메모리 요구량에 따라 Pod 자원 할당
- rate limit에 맞춘 pool·backoff·재시도 정책
- IDC 자원이 부족할 때 WireGuard로 연결한 In-house 워커까지 사용

역할을 나눈 방식

각 크롤러의 수집 로직과 데이터 오류는 해당 담당자가 수정했습니다. 저는 여러 크롤러가 함께 쓰는 **스케줄링, Kubernetes 자원, 네트워크, DB 연결, 모니터링**을 설계하고 운영했습니다. 공통 장애가 생기면 원인을 찾아 담당자가 다시 같은 문제를 겪지 않도록 설정과 런북에 반영했습니다.

결과

80개+ 크롤러
실행 시간과 동시성을 한 곳에서 관리

데이터 도달 시간
취약점 정보는 수집부터 고객사 반영까지 약 1시간

작업 격리
크롤러별 Pod로 자원 과사용과 실패 영향 분리

장애 대응
만료·DNS·디스크·스케줄 문제를 알림과 런북으로 관리

웹 상세: https://portfolio.hwlabs.dev/items/infra-k8s-airflow-ops.html

06 · SELECTED PORTFOLIO

크롤러 DB 부하를 줄인 DML Broker 설계

80개가 넘는 크롤러의 DB 직접 연결을 공통 Broker와 제한된 queue로 바꿔 lock·connection 오류를 없앤 작업

기간	소속	역할
2026	래브라도랩스(LabradorLabs)	데이터 엔지니어 · 공통 DB 접근 구조 설계

`MySQL` `PostgreSQL` `Kubernetes` `HAProxy` `Connection Pool` `Bounded Queue`

문제

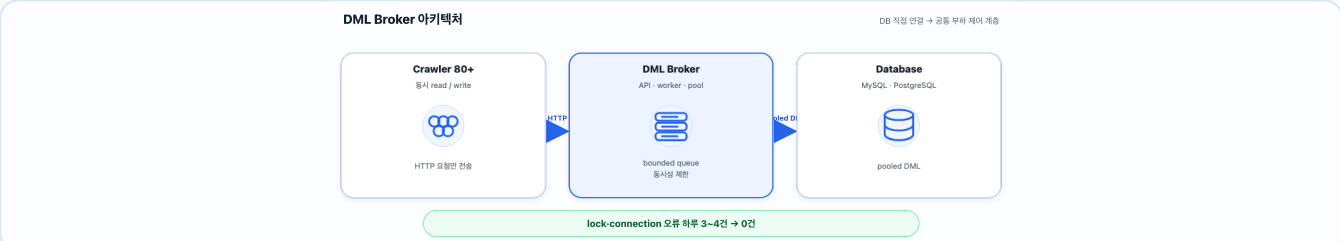
80개가 넘는 크롤러가 같은 DB에 직접 연결해 읽기와 쓰기를 병렬로 처리했습니다. 부하가 몰리면 lock과 connection 오류가 하루 3~4건씩 발생했고, 크롤러마다 접속 방식과 재시도 로직도 달랐습니다.

운영에서 정한 규칙

- queue가 차면 429와 Retry-After를 반환하고 client는 backoff 후 재시도
- write 요청은 idempotency key와 DB unique key/upsert로 중복 반영 방지
- Kubernetes 노드마다 Broker pod를 배치해 특정 노드로 요청이 쏠리지 않게 분산
- IDC와 In-house는 같은 host명을 쓰되 site-local worker로 연결
- request id, 처리 시간, active/pending 수를 로그로 남겨 병목 추적

DB 앞에 공통 경계를 둔 이유

크롤러가 DB 계정과 connection을 직접 들고 있지 않도록 중간 Broker API와 worker를 설계했습니다. 요청을 worker의 bounded queue에 넣고, 정해진 동시성 안에서 connection pool을 사용하게 했습니다.



크롤러는 HTTP 요청만 보내고, queue와 connection pool에서 대기열과 DB 동시성을 제한했습니다.

결과

DB lock·connection 오류

하루 3~4건에서 적용 후 4개월간 0건

접속 정보

크롤러마다 흩어져 있던 DB credential을 공통 계층으로 이동

부하 제어

동시에 처리할 요청 수와 대기열을 한 곳에서 관리

제가 맡은 부분

문제 정리, Broker API와 worker 구조, queue·connection pool 정책, 실패·재시도 경계를 설계했습니다. 각 수집기 담당자는 자신의 크롤러를 Broker 호출 방식으로 바꾸고, 공통 DB 접근 구조와 운영 기준은 제가 맡았습니다.

웹 상세: https://portfolio.hwlabs.dev/items/dml-broker-db-stability.html

07 · SELECTED PORTFOLIO

AWS → IDC/In-house 인프라 이전 & 클라우드 비용 절감

AWS에서 운영하던 데이터 백엔드·수집 인프라를 자체 IDC/In-house 서버로 이전하고 검증을 거쳐 전환한 뒤 AWS 자원을 종료해 반복 과금을 끊은 작업

기간	소속	역할
2024–2026 (AWS 자원 종료 ~2026-04)	래브라도랩스(LabradorLabs)	데이터 엔지니어 · 인프라

`AWS` `EC2 / EFS / RDS` `S3` `IDC / In-house` `MySQL Replication`

처음 마주한 문제

시작은 AWS 비용이었습니다. 수집·배포·운영(prod)이 AWS의 DB 인스턴스 하나에 통으로 들어 있어 데이터가 늘수록 과금이 따라 늘었고, 비용 문제로 그 구성 그대로 사내 범용 PC의 M.2 SSD로 옮겼습니다. 그러자 다른 문제가 생겼습니다. 크롤러가 몰리는 시간대에 DB 쓰기와 디스크 I/O가 겹쳤고, 발열이 올라가면 서버 전체가 멈춰 한 달에 10~20번씩 운영 이슈가 생겼습니다.

단순히 DB가 느리다고 보지 않고 **syslog, 실행 시간대, 네트워크 상태**를 같이 확인했습니다. 장애가 특정 시간대의 수집 집중과 맞물린다는 점을 찾았습니다.

제가 맡은 부분

증상 확인과 스케줄 조정, IDC 서버 도입 요청, 데이터 이관과 검증, DB·백업·모니터링 재구성을 맡았습니다. 비용 절감보다 먼저 서비스가 멈추지 않는 순서를 잡는 데 집중했습니다.

먼저 멈추지 않게 만들기

새 서버를 바로 들일 수는 없었기 때문에 수집 스케줄을 먼저 분산했습니다. 동시에 돌던 작업을 시간대별로 나누고, 무거운 작업이 한 장비에 몰리지 않도록 실행 순서를 조정했습니다.

조정 전	스케줄 분산 후
월 10~20건 수준의 프리징·DB 운영 이슈	월 5건 미만으로 감소

IDC로 옮긴 과정

스케줄 조정은 임시방편이라고 판단했습니다. 서버 도입을 요청하고, 인스턴스 하나가 다 감당하던 구조를 수집 DB·배포 DB·운영 DB로 분리해 수집 부하가 배포·운영 조회를 방해하지 않게 만들었습니다. IDC 환경에 DB primary/replica, 수집 서버, 백업·복구 경로와 모니터링을 다시 구성했습니다.

현황 확인 syslog·네트워크·부하 시간대 점검 → **임시 안정화** 수집 스케줄 분산 → **IDC 구축** 서버·DB·백업·모니터링 구성 → **이관 검증** 데이터 정합성 확인 후 전환

이관을 마친 뒤에는 운영(prod) DB만 AWS에 남기고 나머지 자원을 정리해 월 AWS 비용을 약 94% 줄였습니다.

결과

DB 운영 장애
스케줄 조정으로 월 5건 미만, IDC 전환(2024) 이후 1~2건

부하 간섭 제거
수집·배포·운영 DB를 분리해 수집 부하가 조회를 방해하지 않는 구조

반복 비용
운영 DB 외 AWS 자원 정리로 월 비용 약 94% 절감

복구 가능성
백업·복원과 정합성 확인을 거친 뒤 전환

웹 상세: https://portfolio.hwlabs.dev/items/aws-to-idc-migration.html

08 · SELECTED PORTFOLIO

Grafana 기반 데이터·DB 모니터링 구축

DB 안에만 쌓이던 데이터·운영 지표를 대시보드로 가시화하고 여러 메트릭을 한 화면에서 비교 분석하도록 모니터링 체계를 구축

기간	소속	역할
2024–2025	래브라도랩스(LabradorLabs)	데이터 엔지니어

`Prometheus` `Grafana` `PostgreSQL` `MySQL` `Exporter` `cAdvisor` `Alerting`

확인해야 할 시스템이 늘어난 문제

팀에서 운영하는 서버가 10~20대로 늘면서, 장애가 생길 때마다 서버에 접속해 DB와 로그를 따로 확인하는 방식이 한계에 닿았습니다. 수집 누락, 수집량 급증, DB replica 지연, 고객사로 나가는 데이터 상태가 서로 다른 화면에 흩어져 있었습니다.

알림과 운영

대시보드만 만들어 두지 않고, 임계값을 넘으면 메신저와 이슈 관리 도구로 알림이 가도록 연결했습니다. 토큰 만료, 백업 실패, 데이터 동기화 지연처럼 미리 조치할 수 있는 항목은 장애 전에 알려 주도록 했습니다.

무엇을 한 화면에 모았나

Prometheus와 Grafana를 사용해 서버와 DB 상태를 공통 대시보드로 묶었습니다. 데이터 현황은 PostgreSQL의 시계열 데이터와 연결해 같은 시간축으로 비교했습니다.

서버·DB·수집·고객사 데이터 지표를 모아 같은 시간축으로 비교하고 이상 상태는 알림으로 연결했습니다.

- **서버** — CPU, 메모리, 디스크, 컨테이너 사용량
- **DB** — read/write, connection, replica 상태와 지연
- **수집** — 크롤러 실행 여부, 누락, 시간대별 수집량 증감
- **고객사 데이터** — 배포·동기화 진행률, 오류와 지연

결과

10~20대 서버

공통 상태를 한 화면에서 확인

DB·수집·배포

서로 다른 지표를 같은 시간축으로 비교

선제 대응

만료·지연·실패를 알림으로 먼저 확인

제가 맡은 부분

공통으로 볼 지표를 정하고 Prometheus 수집, Grafana 대시보드, PostgreSQL 시계열 조회와 알림을 구성했습니다. 개별 크롤러의 내부 로직은 담당자가 관리했고, 저는 전체 수집량과 누락, DB·서버, 고객사 데이터 흐름을 함께 볼 수 있는 관측 구조를 맡았습니다.

웹 상세: https://portfolio.hwlabs.dev/items/grafana-monitoring.html