

# 아파트 하자 분류 AI를 운영하는 MLOps 인프라 설계

이음의 하자 관리 서비스 '채들'에서 하자 분류 모델이 실제 운영에서 돌아가도록, AWS EKS 위에 서빙·재학습·배포·모니터링 체계를 설계하고 운영했습니다. 모델 설계는 모델 담당이 맡았고, 저는 **그 모델이 비용과 장애 없이 계속 돌아가게 하는 일**을 맡았습니다.

|           |                         |                                                             |                         |
|-----------|-------------------------|-------------------------------------------------------------|-------------------------|
| 작성<br>하성진 | 역할<br>MLOps 인프라 설계 · 운영 | 스택<br>AWS EKS · Kubeflow ·<br>KServe/Knative ·<br>Karpenter | 기간<br>2024.07 ~ 2025.11 |
|-----------|-------------------------|-------------------------------------------------------------|-------------------------|

**50%↓**

SageMaker 대비 추론 운영 비용  
(최소 절감폭, 동일 요청 기준)

**11 → 5**

상시 가동 서버 수  
온디맨드 → 스팟 전환

**99%**

재학습 파이프라인 성공률  
(스팟 중단 시 자동 재개)

## 01 시스템 구성

서비스별로 진입점을 나누고, **항상 떠 있어야 하는 것과 요청이 있을 때만 뜨면 되는 것**을 레이어로 분리했습니다. 비용 절감과 장애 대응의 대부분이 이 분리에서 나왔습니다.

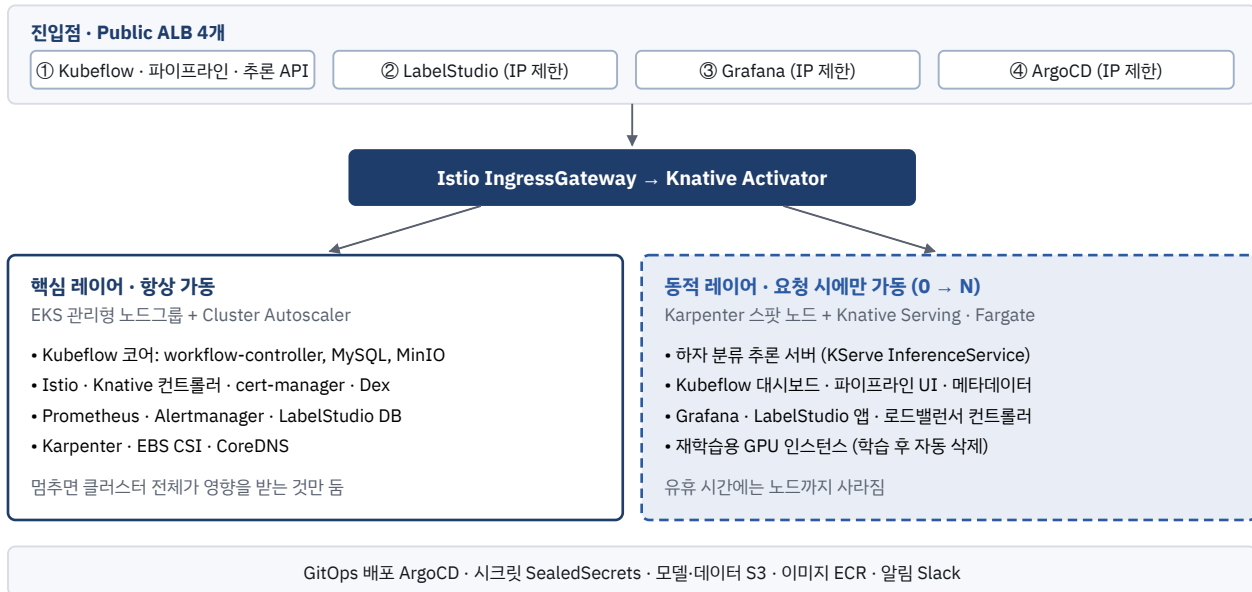


그림 1. 서비스별 ALB 4개와 2계층 노드 구조. 동적 레이어는 트래픽이 없으면 파드와 노드가 모두 0으로 줄어든다.



그림 2. 운영한 하자 분류 모델의 3단 구조. 세 모델이 서로 다른 자원(CPU·GPU)을 쓰기 때문에 서빙과 재학습 인프라를 단계별로 나눠야 했다.

## 02 비용 문제를 구조로 해결한 사례

### 2.1 SageMaker 서버리스의 한계 → 쿠버네티스 기반 서버리스 추론으로 이전

|    |                                                                                                                                                              |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 문제 | SageMaker 서버리스 엔드포인트는 동시 요청 200건, 엔드포인트 50개, 메모리 6GB가 상한이었습니다. 트래픽이 불규칙한데도 유틸 비용이 나갔고, AWS 전용 구성에 묶여 있었습니다.                                                  |
| 해결 | KServe + Knative로 추론 서버를 옮겼습니다. 모델·Transformer·Predictor를 이미지로 만들어 ECR에 올리고, EKS Fargate 위에 InferenceService를 배포했습니다. 요청이 없으면 파드가 0개로 줄어듭니다 (scale-to-zero). |
| 결과 | 같은 요청량 기준 운영 비용 <b>최소 50% 절감</b> . 쿠버네티스 표준 리소스만 쓰게 되어 AWS 종속성도 사라졌습니다.                                                                                      |

### 2.2 쓰지 않는 서비스가 서버를 붙잡고 있던 문제 → 2계층 노드 설계

|    |                                                                                                                             |
|----|-----------------------------------------------------------------------------------------------------------------------------|
| 문제 | 대시보드·라벨링 도구처럼 가끔 쓰는 서비스 때문에 노드가 상시 떠 있었습니다. 노드그룹은 콘솔에서 수동으로 크기를 조정했고, 노드마다 남는 자원이 많았습니다.                                    |
| 해결 | 클러스터의 파드를 "멈추면 클러스터가 영향받는가"로 하나씩 분류했습니다. 핵심 레이어만 관리형 노드그룹에 두고, 나머지는 Knative Serving으로 바꿔 Karpenter가 필요할 때만 스팟 노드를 띄우게 했습니다. |
| 결과 | 상시 서버 <b>11대 → 5대</b> , 구매 방식은 온디맨드에서 <b>스팟</b> 으로 바뀌었습니다.                                                                  |

### 2.3 scale-to-zero의 대가, 콜드 스타트 → 병목 구간을 재서 줄임

|    |                                                                                                                                                                             |
|----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 문제 | 파드를 0개까지 줄이자 첫 요청이 느려졌습니다. 파드 기동 과정을 단계별로 재 보니 TorchServe 이미지 pull이 약 1분 10초로 가장 길었습니다.                                                                                     |
| 해결 | 불필요한 라이브러리를 빼고 베이스 이미지를 바꿨습니다. DockerHub pull 제한을 피하러 이미지를 ECR로 옮기고, 기동을 늦추던 kube-proxy의 자원 할당도 조정했습니다. Fargate 프로필별 기동 시간(Small 약 4분 30초, Large 약 2분 30초)도 실측해 비용과 비교했습니다. |
| 결과 | 모델 서버 이미지 <b>2.9GB → 1.67GB(42% 감소)</b> . Fargate 자체의 프로비저닝 지연은 줄일 수 없는 한계로 기록하고, 설계에 반영했습니다.                                                                               |

### 03 대규모 추론에서 막힌 지점을 찾아낸 사례

약 4만 건을 한 번에 추론하는 배치에서, 요청 수만큼 파드가 늘지 않는 문제가 차례로 나왔습니다. 매번 **설정** → **컴포넌트 격리** → **코드** 순서로 범위를 좁혀 원인을 찾았습니다.

| 증상                                    | 원인을 찾은 방법                                                                                                                                  | 해결과 결과                                                                        |
|---------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| <b>Predictor가 100개에서 멈춤</b>           | Autoscaler 설정을 바뀌도 그대로였습니다.<br>Transformer 없이 Predictor만 200개로 띄우니 정상이어서, 플랫폼이 아니라 호출 코드 문제로 좁혔습니다. KServe 비동기 클라이언트의 동시 요청 기본값이 100이었습니다. | 제한값을 300으로 올림 → 200개 이상 확장,<br>전체 처리 시간 <b>약 30% 감소</b>                       |
| <b>파드 357개 중 230개만 생성</b>             | 실패한 파드가 ProvisioningFailed였고, Fargate의 생성 속도 제한(토른 100개, 초당 20개 충전)에 걸린 것을 확인했습니다.                                                         | 헬스체크로 20개씩 2.5초 간격 사전 기동 →<br><b>357개 100% 생성</b> , 처리 시간 약 15% 단축            |
| <b>요청 10건이 파드 8~9개에만 분배</b>           | Activator 버퍼링과 Queue-Proxy 준비 시점 차이로, 준비 안 된 파드를 건너뛰고 일부 파드에 요청이 몰렸습니다.                                                                    | 사전 기동 후 안정화 대기, 초당 20건씩 점진 전송, 동시성 1·속소 지연 10분 설정 →<br><b>10건이 10개 파드에 분배</b> |
| <b>EKS 1.32 업그레이드 후 파드가 Error로 종료</b> | Transformer는 SIGTERM을 정상 처리했지만, Predictor의 TorchServe는 신호를 받고도 종료하지 못해 강제 종료되고 있었습니다. 업그레이드된 kubelet이 이를 드러낸 것이었습니다.                       | 모델 서버를 KServe 기본 서버로 교체 → 종료 에러 해소, 종료 시간 <b>30초 → 3초</b>                     |

### 04 운영 중 장애를 근본 원인까지 추적한 사례

#### 4.1 제로 스케일 상태에서 요청이 와도 서비스가 깨어나지 않음

**원인** Knative는 Activator가 요청을 감지해 파드를 띄우는데, Activator는 Istio 같은 네트워킹 레이어를 거친 트래픽만 봅니다. ALB가 서비스로 바로 요청을 보내고 있었습니다.

**해결** ALB의 백엔드를 Istio IngressGateway로 바꾸고, oauth2-proxy 인증이 걸리지 않아야 하는 도메인은 AuthorizationPolicy에서 제외했습니다. 이어서 나온 Grafana의 시크릿 접근 실패와 CORS 403도 각각 RBAC 확장과 grafana.ini 설정으로 해결했습니다.

#### 4.2 노드그룹이 늘어나지 않음 → 삭제된 보안그룹을 CloudTrail로 추적

**원인** 노드 수만 늘리려 했는데 "보안그룹 없음" 오류가 났습니다. 시작 템플릿이 가리키는 보안그룹을 CloudTrail로 추적하니, 사용 중이던 보안그룹이 root 계정에서 삭제된 이력이 있었습니다.

**해결** 기존 노드그룹 복구는 불가능하다고 판단하고 새 노드그룹을 만든 뒤 drain으로 파드를 무중단 이전했습니다. 시작 템플릿이 참조하는 보안그룹은 "삭제 금지"로 지정했습니다.

#### 4.3 재학습 파이프라인 타임아웃 → GPU 재고 부족을 찾아냄

|    |                                                                                                                                       |
|----|---------------------------------------------------------------------------------------------------------------------------------------|
| 원인 | Lambda는 정상 응답했는데 파이프라인이 실패했습니다. 애플리케이션 로그가 없어 CloudTrail에서 EC2 API를 확인하니, 해당 가용영역의 GPU 인스턴스 재고 부족(InsufficientInstanceCapacity)이었습니다. |
| 해결 | 재고가 부족할 때마다 코드를 고칠 수 없으므로, 인스턴스 타입을 허용 목록 안에서 파라미터로 받게 바꿨습니다.                                                                         |

#### 4.4 보안을 강화하자 추론 요청이 502로 실패

|    |                                                                                                                 |
|----|-----------------------------------------------------------------------------------------------------------------|
| 원인 | 외부 노출을 막으려 IngressGateway를 LoadBalancer에서 ClusterIP로 바꿨는데, ALB가 여전히 노드를 대상으로 등록(target-type: instance)하고 있었습니다. |
| 해결 | ALB 대상 방식을 파드 IP로 바꿨습니다. 502가 해소되고 별도 ELB 엔드포인트도 사라져 외부 노출 경로가 줄었습니다.                                           |

## 05 재학습 파이프라인 자동화

새 데이터로 추가 학습하자 기존 클래스의 성능이 떨어지는 일을 겪었습니다. 그래서 **재학습은 자동으로 하되, 검증을 통과한 모델만 배포**되게 만들었습니다. 재학습 시점은 두 경우로 나뉘었습니다.

| 경우            | 방식    | 동작                                                                    |
|---------------|-------|-----------------------------------------------------------------------|
| 기존 분류에 데이터 추가 | 완전 자동 | 매달 1일 09시 CronJob이 데이터 증가량을 확인 → 임계값을 넘으면 Slack 알림 후 파이프라인 실행         |
| 새 세부공종 등장     | 반자동   | 공종마다 노이즈 패턴이 달라 도메인 담당이 전처리 규칙을 정의 → 이후 대시보드에서 실행. 이 판단은 자동화하지 않았습니다. |



학습 시작 시 Lambda가 GPU 인스턴스 생성 · 끝나면 삭제 · 스팟 중단 시 S3 체크포인트에서 재개

**그림 3.** Kubeflow Pipelines 기반 재학습 흐름. 파이프라인 코드는 GitHub Actions가 컴파일해 업로드한다.

결과적으로 수작업 대비 **약 70%의 시간을 줄였고**, 스팟 중단 시 자동 재개 덕분에 **파이프라인 성공률 99%**를 유지했습니다. 새 모델은 KServe의 트래픽 분할로 기존 모델과 나란히 띄워 비교한 뒤 전환하는 A/B 테스트 환경도 함께 구성했습니다.

## 06 혼자 운영해도 무너지지 않게 만든 장치

| 영역    | 구성                                                                               |
|-------|----------------------------------------------------------------------------------|
| 배포    | ArgoCD로 클러스터 상태를 Git과 동기화. 시크릿은 SealedSecrets로 암호화해 저장소에서 관리                     |
| 접근 제어 | 운영 도구 ALB는 사내 IP만 허용. 클러스터 API 엔드포인트도 지정 IP와 VPC 내부로 제한                          |
| 모니터링  | Prometheus · Grafana · Alertmanager. Grafana도 Knative로 전환해 쓸 때만 기동               |
| 인수인계  | 추론 API 추가, Karpenter 노드풀 설정, ArgoCD 프로젝트 추가, 시크릿 관리, 노드·파드·라우팅 장애 대응 절차를 런북으로 정리 |

**배운 점** — 처음 LabelStudio를 올릴 때, 불륨 문제를 4일 동안 붙잡고도 풀지 못했습니다. 동적·수동 프로비저닝 개념을 모른 채 답만 찾았기 때문입니다. 이후로는 오류를 만나면 **먼저 동작 원리를 정리하고, 원인 후보를 하나씩 격리**해서 확인합니다. 위의 사례들은 모두 그 방식으로 풀었습니다.

## 07 기술

| 구분        | 내용                                                                                     |
|-----------|----------------------------------------------------------------------------------------|
| 클러스터 · 노드 | AWS EKS (1.30 → 1.32 업그레이드) · Karpenter · Cluster Autoscaler · Fargate · 스팟 인스턴스       |
| 서빙        | KServe · Knative Serving/Eventing · Istio · TorchServe → KServe 기본 모델 서버               |
| 파이프라인     | Kubeflow Pipelines v2 · Kubernetes CronJob · AWS Lambda · GitHub Actions · Optuna      |
| 네트워크 · 보안 | ALB Ingress Controller · Dex/oauth2-proxy · AuthorizationPolicy · RBAC · SealedSecrets |
| 데이터 · 운영  | S3 · ECR · EBS CSI · LabelStudio · ArgoCD · Prometheus · Grafana · Slack SDK           |

비용 절감쪽은 동일 요청량을 기준으로 SageMaker와 Knative 비용을 비교한 값이며, 모델 아키텍처 설계는 모델 담당자가 수행했습니다.