AI 프로덕션 디버깅

학습되지 않는 에러를 잡아내는 시스템

ThakiCloud (Hyojung Han)

AI 프로덕션 디버깅

학습되지 않는 에러를 잡아내는 시스템

ThakiCloud (Hyojung Han)

Contents

1 1. 프로덕션 AI가 달랐다

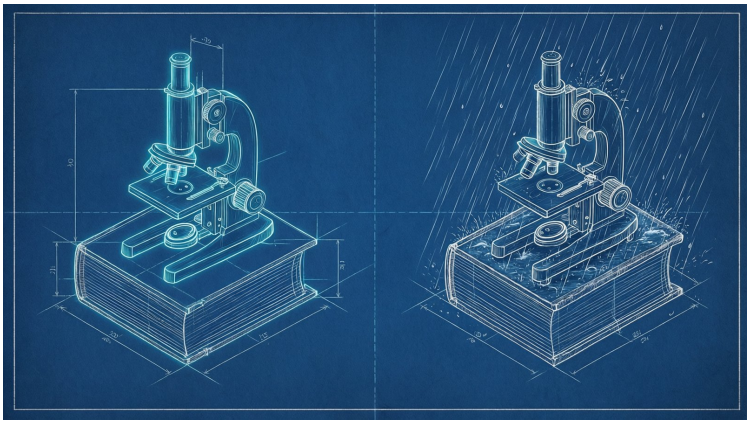

Figure 1.1: 1. 프로덕션 AI가 달랐다

1.1 훈련과 배포 사이의 간극

AI 시스템을 프로덕션에 올려본 엔지니어라면 한 가지를 안다. 훈련 환경에서 완벽히 동작하던 모델이 운영 환경에서는 전혀 다른 결과를 낸다는 사실이다. 이 간극은 기술이 덜 성숙해서가 아니라 두 환경의 본질이 다르기 때문에 생긴다.

훈련 환경은 통제된 조건이다. 데이터셋은 이미 정제됐고 입력 분포는 균등하며 하드웨어 자원은 필요할 때 할당된다. 프로덕션 환경은 다르다. 사용자는 예측 불가능한 입력을 보내고 데이터 분포는 시간에 따라 흔들리며 시스템은 병목과 타임아웃과 메모리 고갈을 수시로 마주친다.

이 차이를 이해하지 못하면 디버깅은 정처 없이 헤매는 일이 된다. 흔히 저지르는 실수 하나. 모델 성능이 떨어지면 모델을 다시 훈련시키자고 나서는 것이다. 그러나 대부분의 문제는 모델에 있지 않다. 데이터

파이프라인이 깨졌거나 입력 전처리가 훈련 때와 달라졌거나 후처리 로직에 버그가 들어간 경우가 훨씬 많다. 모델은 결국 수학적으로 일관된 함수일 뿐이다. 바뀐 건 그 함수가 받는 입력이다.

1.2 모델은 학습하지 않는다

가장 먼저 바꿔야 할 생각이 있다. 배포된 모델은 학습하지 않는다는 것이다. 온프레미스에서 더 많은 데이터를 보며 스스로를 개선하지 않는다. 프로덕션에서 성능이 떨어졌다면 그것은 환경이 변했다는 신호이자 시스템 설계의 문제다.

이건 단순한 말장난이 아니다. 디버깅 전략 자체를 결정한다. 모델 내부에서 원인을 찾으려는 시도는 대부분 실패한다. Grad-CAM이나 SHAP으로 어느 토큰에 주의가 집중되는지 들여다보는 일은 흥미롭지만 서빙 장애를 해결해 주지는 않는다. 실제로 손봐야 할 대상은 모델을 둘러싼 시스템이다.

1.3 AI 시스템 실패의 4가지 유형

프로덕션 AI 시스템의 실패는 크게 4가지로 나뉜다. 유형마다 감지 방법도 해결 접근법도 다르다.

유형 1: 출력 이상. 모델이 잘못된 형식의 출력을 내거나 의미적으로 엉뚱한 결과를 반환하거나 갑자기 null을 내뱉는다. 가장 눈에 띄는 실패이고, 모니터링 시스템에서 즉시 포착되는 경우가 많다.

유형 2: 데이터 드리프트. 입력 데이터의 분포가 훈련 시점과 달라지면서 성능이 서서히 떨어진다. 진행이 점진적이라 며칠, 때로는 몇 주가 지나서야 알아차린다. "작동은 하는데 정확도가 떨어졌다"는 증상은 거의 항상 드리프트다.

유형 3: 파이프라인 실패. 데이터 전처리가 깨지거나 모델 로딩에

실패하거나 서빙 인프라가 다운된다. 모델과는 무관한 곳에서 벌어지며 시스템 장애로 명백하게 드러난다.

유형 4: 에지 케이스. 훈련 데이터에 없던 입력 패턴이 들어오면 시스템이 예측 불가능하게 동작한다. 디버깅이 가장 까다로운 유형이다. 실패가 명백히 드러나지 않고 재현도 어려우며, 흔한 통계적 방법으로는 아예 잡히지 않을 수도 있다.

이 4가지를 구분하는 것이 디버깅의 첫걸음이다. 유형 1의 문제에 유형 3의 해결책을 들이대면 시간만 버린다.

1.4 이 책의 구조

이후 각 장은 위 4가지 유형을 하나씩 깊이 있게 다룬다. 2장에서는 출력 이상 감지와 검증 시스템을, 3장에서는 데이터 드리프트 모니터링과 적응 전략을, 4장에서는 파이프라인 실패 모드와 회복탄력성 설계를, 5장에서는 에지 케이스 수집과 지속적 개선 파이프라인을 다룬다. 마지막 장에서는 4가지 유형을 통합적으로 관리하는 플랫폼적 사고를 정리한다.

2 2. 출력 이상 감지 시스템

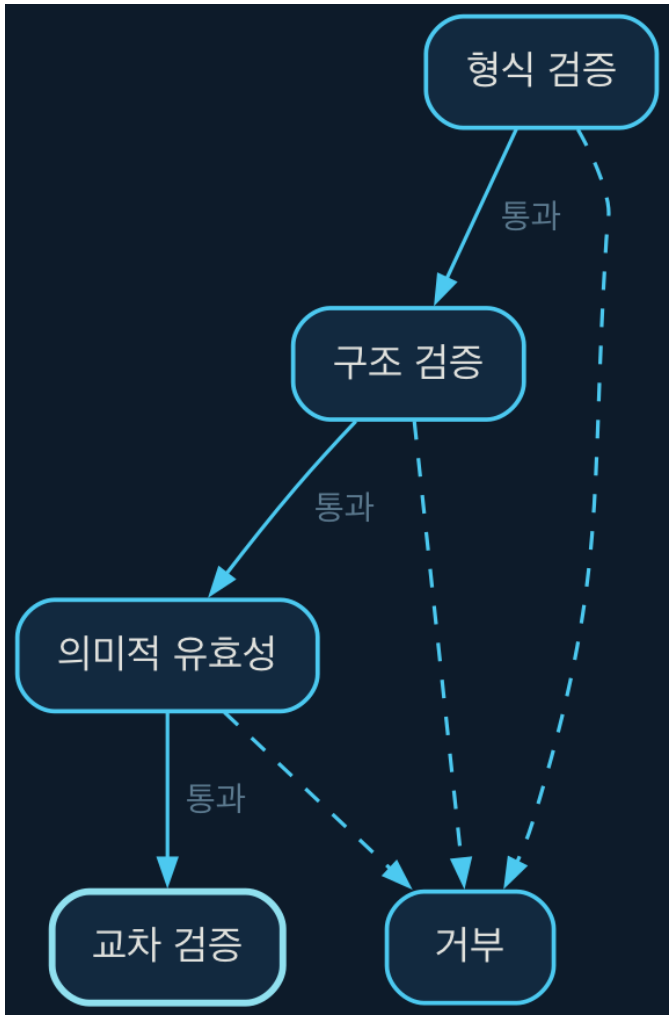

Figure 2.1: 2. 출력 이상 감지 시스템

2.1 출력 검증의 필요성

LLM이나 diffusion 모델이 실시간으로 텍스트를 생성할 때 그 출력은 애플리케이션의 나머지 부분을 좌우한다. 구조화된 데이터로 파싱해야 하면 JSON 스키마를 따라야 하고, 다른 시스템의 입력이 될 때는 정해진 형식이 있다. 그러나 Transformer 기반 모델은 이 형식을 절대 보장하지 않는다. 학습 데이터에 "“json”이 많이 섞여 있었다면 JSON을 뱉어낼 가능성은 높아지지만, 그것이 계약은 아니다.

출력이 한 번 어긋나면 애플리케이션 전체가 깨질 수 있다는 점이 문제다. JSON 파싱 에러, null 참조, 타입 불일치가 연쇄적으로 일어나면 사용자 눈에는 “시스템이 고장났다”로 보인다. 모델의 잘못이 아니다. 검증 없이 출력을 신뢰한 시스템 설계의 잘못이다.

2.2 계층적 검증 구조

출력 이상을 감지하려면 검증도 계층적으로 설계해야 한다. 하위 수준에서 상위 수준으로 올라갈수록 검증 비용은 높아지므로 cheap하고 frequent한 체크를 먼저 통과시킨 뒤 expensive한 deeper 검증을 수행한다.

Level 1: 형식 검증. 출력이 존재하는지, null이나 빈 문자열은 아닌지, 최소 길이를 충족하는지 확인한다. regex 기반이라 밀리초 단위로 끝난다. 실패하면 즉시 reject한다.

Level 2: 구조 검증. JSON 출력이라면 스키마에 맞는지 검증한다. 필수 필드가 존재하는지, 타입이 정확한지, enum 값이 유효한지를 검사한다. JSON Schema 라이브러리를 쓰면 간단히 구현할 수 있다.

Level 3: 의미적 유효성. 구조는 맞지만 의미적으로 이상한 경우를 탐지한다. “age: -5”, “date: 9999-12-31”, “카운트: 문자열” 같은 값들이다. 여기에는 도메인 지식 기반 규칙이 필요하다.

Level 4: 교차 검증. 모델 여러 개의 출력을 비교하거나, 이전 출력과 대조하거나, 외부 기준과의 일관성을 확인한다. 비용이 가장 높으므로 선택적으로만 사용한다.

2.3 구현 패턴

Python에서 이 검증 계층을 구현하면 다음과 같은 구조가 된다.

```
def validate_output(raw_output, config):
 # Level 1: None check, empty check
 if not raw_output or len(raw_output.strip()) == 0:
 return ValidationResult(success=False, level=1, reason="empty")

 # Level 2: Schema validation
 try:
 parsed = json.loads(raw_output)
 except json.JSONDecodeError:
 return ValidationResult(success=False, level=2,
 ↪ reason="invalid_json")

 schema_errors = validate_schema(parsed, config.schema)
 if schema_errors:
 return ValidationResult(success=False, level=2,
 ↪ reason=schema_errors)

 # Level 3: Semantic validation
 semantic_errors = validate_semantics(parsed, config.rules)
 if semantic_errors:
 return ValidationResult(success=False, level=3,
 ↪ reason=semantic_errors)

 return ValidationResult(success=True, parsed=parsed)
```

이 구조의 핵심은 **fail-fast**다. 각 단계에서 실패하면 즉시 반환한다. 전체 출력을 매번 검증하려면 비용이 크니 cheap한 단계에서 먼저 걸러내야 한다.

2.4 이상 감지: 통계적 접근

형식적 검증을 통과해도 출력 품질은 떨어질 수 있다. 이를 포착하려면 통계적 방법이 필요하다. 간단하면서도 효과적인 접근은 **출력 길이 분포**를 모니터링하는 방법이다.

훈련 시점이나 정상 운영 시의 출력 길이 분포를 기록해 두었다가 현재 분포와 큰 차이가 나면 경고한다. 사용자가 갑자기 짧은 답변만 받거나 반대로 불필요하게 긴 출력이 나오기 시작한다면 무언가 잘못됐다는 신호다.

더 정교한 방법으로는 **출력 Embedding 기반 이상 감지**가 있다. 정상 출력을 Embedding 공간에 투영하고, 새로운 출력이 정상 클러스터에서 멀어지면 플래그를 세운다. 이 방법은 계산 비용이 높으므로 샘플링으로 적용한다.

2.5 회귀 테스트로 출력 품질 지키기

새 모델 버전이나 프롬프트 변경을 배포할 때, 이전에 동작하던 출력 패턴이 깨지지 않았는지 확인하는 regression 테스트가 필수다. 이것은 단위 테스트와 다르다. 단위 테스트가 함수의 정확성을 검증한다면, 회귀 테스트는 시스템 전체의 출력 분포가 변하지 않았는지를 검증한다.

```
REGRESSION_CASES = [
 {"input": "서울 날씨 알려줘", "expected_keywords": ["날씨",
↪  "서울"], "forbidden": ["에러"]},
 {"input": "3 + 5는?", "expected_pattern": r"8", "type": "math"},
```

```
 {"input": "안녕", "max_length": 50},
]
```

이런 테스트 케이스를 관리하며 CI/CD 파이프라인에 포함시켜 매 배포마다 실행하면 출력 품질 저하를 자동으로 포착할 수 있다. 출력이 "죄송합니다, 에러가 발생했습니다"로 바뀌는 것 자체가 사용자 경험의 직접적 저하다.

2.6 멀티모달 출력 디버깅

텍스트가 아닌 출력, 예컨대 이미지 생성이나 음성 합성에서는 검증이 더 복잡해진다. 텍스트와 달리 "무엇이 올바른 출력인가"를 자동으로 판단하기 어렵기 때문이다.

이미지 생성에서는 출력 이미지가 존재하는지, 원하는 크기인지부터 확인한다. 그다음 CLIP 기반 유사도 점수를 계산해 참조 이미지와 얼마나 비슷한지를 측정한다. 이 점수가 임계치 이하이면 재생성을 요청한다.

오디오 출력에서는 duration, sample_rate 같은 메타데이터를 확인하고 silence detection으로 너무 짧거나 빈 오디오는 아닌지 검증한다. 최종적으로 Whisper 등으로 다시 전사해 원래 의도한 내용이 유지되는지 확인하는 cross-modal 검증도 효과적이다.

3 3. 데이터 드리프트: 조용한 성능 저하

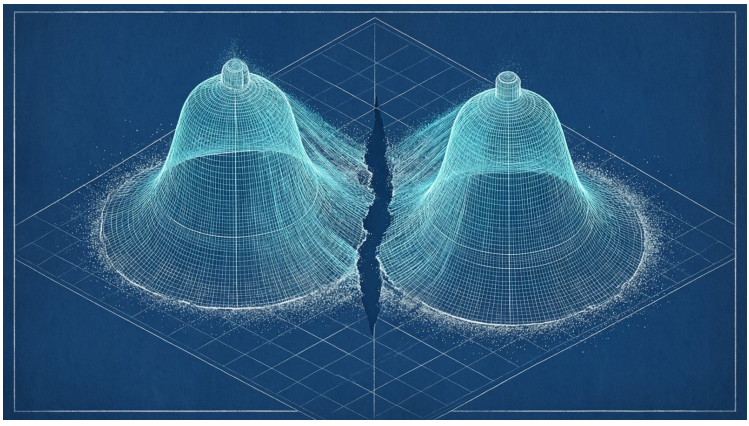

Figure 3.1: 3. 데이터 드리프트: 조용한 성능 저하

3.1 드리프트의 본질

데이터 드리프트는 가장 조용하게 다가오는 실패다. 시스템이 완전히 고장나는 게 아니다. API는 200 OK를 반환하고 모델은 출력을 생성하며 로그에는 에러 하나 찍히지 않는다. 그런데도 predictions의 품질은 서서히 떨어진다. 사용자가 이를 알아채기까지 며칠, 길게는 몇 주가 걸린다. 그때쯤이면 문제는 이미 퍼진 뒤다.

드리프트에는 세 가지 유형이 있다. 유형을 구분해야 각각에 맞는 감지 전략을 세울 수 있다.

Covariate Shift는 입력 변수 X의 분포가 변하는 것이다. 훈련 시 이미지 해상도가 224x224였는데 실제 입력은 380x380으로 들어오는 경우처럼, 입력 자체의 통계적 특성이 달라진다. 데이터 전처리 파이프라인이 바뀌거나 사용자 행동 패턴이 달라지면 이런 일이 생긴다.

Prior Shift는 출력 변수 Y의 분포가 변하는 것이다. 스팸 분류기에서 정상 메일의 비율이 갑자기 늘어난다면 모델이 마주하는 prior distribution이 바뀐 것이다. 대상 도메인 자체가 변했다는 뜻이다.

Concept Drift는 입력과 출력 간 관계, 즉 P(Y|X)가 변하는 것이다. 금리 상승기에 신용카드 이용률이 오르고 카드 사용 패턴이 변화한 것처럼 변수들 사이의 관계 자체가 시간에 따라 바뀐다. 셋 중 가장 다루기 어려운 유형이다. 입출력 분포가 각각 안정적이어도 둘 사이의 관계는 깨질 수 있기 때문이다.

3.2 실시간 모니터링 아키텍처

드리프트를 감지하려면 시스템적으로 데이터를 수집하고 분석해야 한다. 실시간 모니터링 아키텍처의 핵심 컴포넌트는 네 가지다.

데이터 캡처 레이어는 서빙 시점에 입력과 출력을 샘플링한다. 모든 트래픽을 저장하면 비용이 너무 크므로, stratified sampling으로 합리적인 비율로 수집한다. 계층 샘플링이란 특정 조건(새로운 사용자, 특정 카테고리, 이상 패턴)을 만족하는 샘플을 우선 수집하는 것이다.

통계 계산 엔진은 수집된 데이터에서 분포 통계를 계산한다. 입력의 경우 평균, 분산, 분위수를, 출력의 경우 클래스 비율이나 연속 값의 분포를 추적한다. 이 통계량을 시계열로 저장해 시간에 따른 변화를 본다.

드리프트 감지기는 현재 통계량과 기준선(baseline)을 비교한다. 기준선은 훈련 데이터나 직전 안정적 운영 기간의 데이터에서 추출한다. 두 분포 사이의 거리가 임계치를 넘으면 경보를 발생시킨다. 거리 측정에는 Population Stability Index(PSI), Kolmogorov-Smirnov 테스트, KL divergence 등이 쓰인다.

애널리스트 대시보드는 경보를 종합해 팀이 상황을 파악할 수 있게 한다. 단순히 수치를 나열하는 게 아니라 드리프트가 성능에 미친 영향을

추정하고 긴급도를 판단하게 돕는다.

3.3 PSI 기반 감지 구현

PSI는 프로덕션에서 가장 널리 쓰이는 드리프트 지표다. 계산이 간편하고 해석이 명확하다.

```
PSI = Σ ((Actual% - Expected%) * ln(Actual% / Expected%))
```

PSI가 0.1 이하면 significant한 변화 없음, 0.1~0.2는 mild shift, 0.2 이상은 significant drift로 판단한다. 실무에서는 0.25를 임계값으로 삼는 경우가 많다.

```
def calculate_psi(expected, actual, buckets=10):
 """PSI 계산. expected/actual는 각 구간의 비율 리스트."""
 psi = 0
 for exp, act in zip(expected, actual):
 # 0이나 극단적 값 처리
 exp = max(exp, 1e-6)
 act = max(act, 1e-6)
 psi += (act - exp) * math.log(act / exp)
 return psi

def detect_drift(current_stats, baseline_stats, threshold=0.25):
 psi = calculate_psi(baseline_stats['buckets'],
↪  current_stats['buckets'])
 return {
 'drifted': psi > threshold,
 'psi': psi,
 'severity': 'high' if psi > 0.25 else ('medium' if psi > 0.1
 ↪  else 'low')
 }
```

PSI는 단일 수치로 분포 차이를 요약한다는 장점이 있다. 다만 어떤 변수가 drift의 주원인인지는 알려주지 않는다. 원인 파악은 downstream 분석의 몫이다.

3.4 자동 적응 전략

드리프트를 감지만 하고 대응하지 않으면 의미가 없다. 자동 적응 전략은 네 가지 정도로 나뉜다.

Automatic Retraining은 drift가 감지되면 즉시 새 모델을 훈련시킨다. 트리거 조건은 PSI > 0.2에 성능 지표(X_auc, accuracy 등) 저하가 동반되는 경우로 설정한다. 성능 저하 없이 PSI만 높다면 실제로는 문제가 아닐 수 있다. 시즌성 패턴이 원인인 경우가 그렇다.

Ensemble Updating은 기존 모델을 버리지 않고 새 모델과 ensemble을 구성한다. 온라인 학습으로 기존 모델의 가중치를 점진적으로 조절하면 안정성을 유지하면서 새로운 패턴에도 적응한다.

Traffic Routing은 drift가 감지된 입력 그룹만 별도 모델로 라우팅한다. 전체 트래픽을 retraining하는 것은 비용이 크므로 영향을 받는 부분만 분리해 대응한다.

Human-in-the-loop Review는 자동 대응이 위험한 경우 사람 전문가의 판단을 거친다. Concept drift처럼 관계 자체가 크게 변한 경우에는 단순한 재훈련보다 도메인 전문가의 해석이 필요하다.

3.5 드리프트 감지의 실제적 함정

이론적으로는 명확하지만 실전에서는 몇 가지 함정이 있다.

라벨 지연 문제가 먼저다. 성능 지표(accuracy 등)를 추적하려면 정답이 필요한데 레이블이 확보되기까지 시간이 걸린다. 그 사이에 drift가

일어나는지는 알 길이 없다. 대안으로 noisily labeled data나 proxy metric을 쓴다.

정상 variation과 drift의 구분도 까다롭다. 사용자 트래픽에는 요일별, 시간대별, 계절별로 반복되는 주기적 variation이 있다. 이걸 전부 drift로 잡으면 false alarm이 폭증한다. 기준선을 세울 때 이런 정상 variation을 반영해야 한다.

마지막은 **다변량 드리프트**다. 변수가 여러 개면 개별 변수의 PSI가 낮아도 변수 간 관계가 변할 수 있다. 이를 포착하려면 상관행렬이나 공분산 행렬의 변화를 추적해야 한다.

4 4. ML 파이프라인 실패 모드

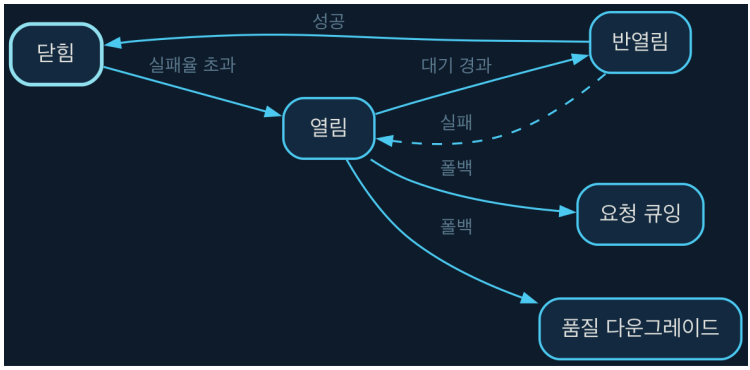

Figure 4.1: 4. ML 파이프라인 실패 모드

4.1 데이터 파이프라인의 취약성

ML 시스템에서 대부분의 실패는 모델 자체가 아니라 데이터 파이프라인에서 발생한다. 데이터가 늦게 도착하거나 포맷이 깨져 들어오거나 스키마가 바뀌면 모델은 손을 쓸 수 없다. 가비지 인 가비지 아웃 원칙은 ML에서 더 가혹하게 작동한다. 모델은 훈련 데이터의 패턴을 학습할 뿐, 유실된 데이터의 패턴은 알아채지 못한다.

전형적인 파이프라인은 데이터 수집 → 전처리 → 특성 추출 → 모델 추론 → 후처리 → 저장 순으로 구성된다. 각 단계마다 다른 종류의 실패가 나타난다.

4.2 단계별 실패 지점

데이터 수집 단계에서는 데이터 소스 자체에 접근할 수 없게 되거나 데이터가 중복되거나 필수 필드가 누락되는 문제가 생긴다. 외부 API가 장애를 일으키거나 Kafka consumer가 뒤처지면 데이터 흐름이 멈춘다.

모니터링에서는 throughput 저하, latency 증가, 에러 레이트 상승으로 이를 포착한다.

전처리 단계의 실패는 조용한 경우가 많다. 전처리가 조금씩 잘못되더라도 모델은 어쨌든 출력을 만들어내기 때문에 시스템은 돌아간다. 다만 출력 품질이 떨어진다. 데이터 타입이 바뀌거나 스케일이 안 맞거나 인코딩이 깨지면 이런 일이 생긴다. 데이터 drift와는 구별해야 한다. drift가 데이터 분포 자체의 변화라면, 전처리 실패는 데이터가 훈련 시와 다른 transformation을 거치는 문제다.

모델 추론 단계에서는 GPU memory 부족, 모델 로딩 실패, 추론 latency timeout이 대표적이다. 모델 크기는 점점 커지는데 GPU 메모리는 제한돼 있어 OOM(Out of Memory)이 흔히 발생한다. 서빙 프레임워크(vLLM, TensorFlow Serving 등)에 장애가 나면 추론 자체가 불가능해지기도 한다.

후처리 단계의 실패는 가장 찾기 어렵다. 후처리가 잘못되어도 추론 자체는 성공하고, 잘못된 출력이 그대로 시스템에 흘러들어간다. JSON 직렬화 실패, 타입 변환 에러, 범위 밖 값 처리가 이 단계에서 발생한다.

4.3 리소스 고갈 감지

GPU 메모리 고갈은 ML 파이프라인에서 가장 흔한 장애 원인이다. 모델 크기가 커지고 동시 요청이 늘고 배치 크기 설정까지 잘못되면 메모리는 금세 바닥난다.

감지는 OOM error 로그 모니터링으로 시작한다. CUDA out of memory, OOMKilled 같은 에러가 찍히면 즉시 알 수 있다. 다만 더 선제적으로 대응하려면 GPU 메모리 사용량을 실시간으로 추적해야 한다.

```
import pynvml
pynvml.nvmlInit()
handle = pynvml.nvmlDeviceGetHandleByIndex(0)
```

```
info = pynvml.nvmlDeviceGetMemoryInfo(handle)
used_ratio = info.used / info.total
if used_ratio > 0.95:
 # 95% 이상 사용 시 경고
 alert("GPU memory critical", used_ratio)
```

이 비율이 95%를 넘으면 새 요청을 즉시 받지 않고 큐에 쌓거나, 배치 크기를 줄이거나, 다른 인스턴스로 라우팅해야 한다.

4.4 서킷 브레이커 패턴

ML 서빙에서 외부 의존성이 실패하면 전체 시스템이 고장 난다. 데이터 소스가 장애를 일으키면 파이프라인 전체가 멈추고, 모델이 응답하지 않으면 사용자 요청이 모두 timeout된다. 서킷 브레이커 패턴은 이런 cascading failure를 막는다.

서킷 브레이커는 세 가지 상태를 가진다. **닫힘(Closed)** 상태에서는 요청이 정상적으로 흐른다. 실패율이 임계치를 넘으면 **열림(Open)** 상태로 전환되고, 이후 요청은 즉시 reject되거나 폴백 응답을 받는다. 일정 시간이 지나면 **반열림(Half-Open)** 상태가 되어 일부 요청을 시도해보고, 성공하면 다시 닫힘 상태로 돌아간다.

ML 시스템에서는 모델 추론 결과를 캐시하는 것도 서킷 브레이커의 일종이다. 모델을 일시적으로 쓸 수 없더라도 유사한 과거 요청의 결과를 내주면 사용자 경험 저하를 막을 수 있다.

4.5 그레이스풀 디그레이션

시스템이 완전히 고장 나지 않더라도 성능이 저하될 수 있다. 이때는 완전히 멈추는 것보다 부분적으로라도 작동하는 편이 낫다. 그레이스풀

디그레이션이 바로 이 발상이다.

요청 queueing은 트래픽이 급증할 때 새 요청을 즉시 reject하지 않고 큐에 넣는 방식이다. 큐가 다 차면 그때서야 reject한다. 사용자는 약간 기다려야 하지만 완전한 실패보다는 낫다.

품질 downgrade는 모델 크기를 줄이거나 더 작은 모델로 교체해 처리량을 늘리는 방식이다. Falcon-7B로 충분한 요청에는 Falcon-7B를 쓰고, 큰 모델이 필요한 요청에만 Falcon-40B를 쓴다.

기능 축소는 일부 기능을 끄고 핵심 기능만 남기는 방식이다. 실시간 번역이 배치 번역으로 바뀌거나 개인화 추천이 범용 추천으로 전환되는 경우가 여기 해당한다.

4.6 장애 격리 설계 원칙

장애 격리의 핵심 원칙은 결합도를 최소화하는 데 있다. ML 파이프라인의 각 컴포넌트는 다른 컴포넌트의 장애에 영향받지 않도록 분리해야 한다.

컴포넌트 간 통신은 asynchronous하게 처리한다. 데이터 파이프라인과 모델 서빙을 직접 연결하지 않고 그 사이에 message queue(Kafka, RabbitMQ 등)를 둔다. 파이프라인이 일시적으로 느려져도 서빙은 방해받지 않고, 서빙을 쓸 수 없게 되어도 파이프라인은 데이터를 계속 쌓아둔다.

각 컴포넌트는 독립적으로 배포할 수 있어야 한다. 모델을 업데이트할 때 서빙 인프라 전체를 내려서는 안 되고, 데이터 전처리 로직을 바꿀 때 모델 서빙을 재시작해서도 안 된다. 이는 컨테이너화(Docker, Kubernetes)와 서비스 메시(service mesh)로 실현한다.

5 5. 에지 케이스 수집과 지속적 개선

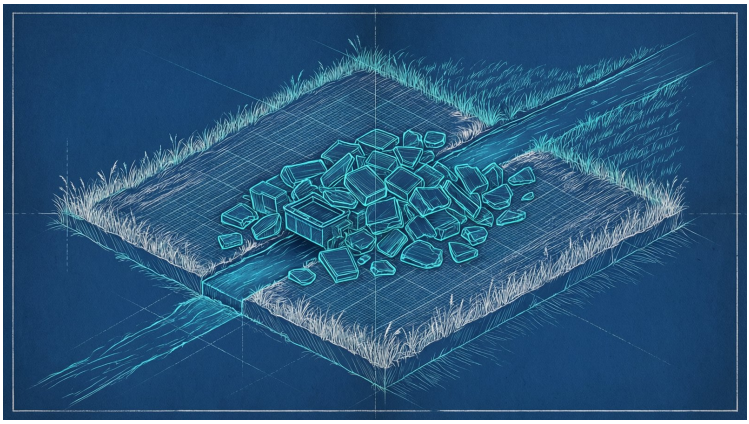

Figure 5.1: 5. 에지 케이스 수집과 지속적 개선

5.1 왜 에지 케이스인가

ML 시스템의 성능은 훈련 데이터가 결정한다. 훈련 데이터에 없는 입력 패턴은 모델도 예측하지 못한다. 당연한 이야기인데도 실무에서는 자주 간과된다. 팀은 모델 아키텍처를 복잡하게 바꾸고 하이퍼파라미터를 튜닝하고 새 프레임워크를 시도한다. 하지만 근본 문제는 데이터에 있다. 아키텍처를 아무리 바꿔도 훈련 데이터에 없는 패턴은 학습하지 못한다.

에지 케이스란 훈련 시나리오에서 만나지 못한 입력 패턴이다. 드물게 발생하기도 하고 전혀 예상하지 못한 조합에서 나타나기도 한다. 에지 케이스를 체계적으로 수집해 재훈련 데이터에 반영하는 일이 프로덕션 AI 시스템을 꾸준히 개선하는 핵심이다.

5.2 실패 케이스 수집 파이프라인

실패 케이스 수집은 자동화해야 한다. 수동으로 하나씩 모으면 속도가 따라가지 못해 결국 방치된다. 자동화된 수집 파이프라인은 네 단계를 거친다.

1단계: 자동 캡처. 모델 출력이 비정상으로 판단되면 해당 입력과 출력을 즉시 저장한다. "비정상"의 판단 기준은 저마다 다르지만 공통으로 쓸 수 있는 조건은 있다. 사용자가 negative 피드백을 보냈거나, 출력이 특정 패턴과 맞지 않거나, 후속 시스템에서 에러가 발생하거나, 예측 confidence가 특정 임계치 이하인 경우다.

2단계: 분류. 캡처된 케이스를 수동으로 분석할 수 있는 수준까지 줄인다. 자동 분류기로 유사한 케이스를 grouping하면 같은 유형의 실패가 여러 건 모여도 하나만 분석하면 된다. 자연어 처리로 텍스트 케이스를 clustering하거나 임베딩 공간에서 가까운 케이스끼리 묶는다.

3단계: 우선순위 결정. 모든 실패 케이스를 한 번에 다 처리할 수는 없다. 영향도와 발생 빈도를 기준으로 우선순위를 매긴다. 영향도는 사용자에게 직접 불이익을 주는지, 시스템 전체에 영향을 미치는지가 기준이고, 빈도는 비슷한 케이스가 얼마나 자주 발생하는지가 기준이다.

4단계: 라벨링. 우선순위가 높은 케이스를 도메인 전문가가 분석해 정답 레이블을 부여한다. 비용이 크므로 정말 필요한 케이스에만 적용한다.

5.3 인간 참여형 라벨링 워크플로우

에지 케이스 분석에는 인간의 판단이 필수다. 모델이 내놓은 출력이 좋은지 나쁜지는 맥락에 따라 다르고, 그것을 판단하려면 도메인 지식이 필요하다. 인간 참여형 라벨링 워크플로우는 효율적으로 운영해야 한다.

핵심 원칙은 **라벨러의 시간을 아끼는 것**이다. 라벨러가 케이스마다 5분씩

들여 분석하게 하면 하루에 처리할 수 있는 케이스 수가 제한된다. 대신 자동화된 분석 결과를 미리 제공하면 라벨러는 그것을 승인하거나 수정하는 데만 시간을 쓰면 된다.

```
# 사전 분석 결과 예시
{
 "case_id": "edge_001",
 "input": "한국의 수도는?",
 "model_output": "서울입니다.",
 "auto_analysis": {
 "predicted_label": "correct",
 "confidence": 0.98,
 "similar_cases": ["edge_023", "edge_047"],
 "keywords": ["지리", "단순 사실"]
 },
 "human_label": None, # 라벨러가 채움
 "priority": "low"
}
```

이렇게 구조화하면 라벨러는 자동 분석 결과를 review하는 형식으로 일할 수 있다. 대부분의 케이스는 automated label을 그대로 승인하면 되고, 불일치하는 케이스에만 집중하면 처리량이 늘어난다.

5.4 재훈련 주기와 데이터 품질

에지 케이스를 수집했으면 재훈련에 반영해야 한다. 다만 언제 재훈련하는 것이 이득인지는 신중히 판단해야 한다.

재훈련에는 비용이 따른다. GPU 자원과 시간이 들고, 재훈련 후 모델 성능이 오히려 떨어질 위험도 있다. 새 버전 모델이 기존에 잘 처리하던 케이스에서 되레 나빠지는 경우도 있는데, 이것이 catastrophic forgetting이다.

정량적 기준으로 재훈련 타이밍을 결정한다. 드리프트 지표(PSI 등)가 일정 임계치를 초과하거나, 사용자 불만 지표가 특정 수준에 도달하거나, 새로운 에지 케이스가 일정 수 이상 누적되면 재훈련을 트리거한다.

정성적 기준으로는 도메인이 바뀌는 경우가 있다. 제품 로드맵이 바뀌어 모델이 다르게 동작해야 하거나, 새로운 규제나 정책이 생겨 기준이 달라지거나, 비즈니스 상황 자체가 바뀌었을 때 재훈련의 가치가 생긴다.

재훈련 후에는 반드시 **A/B 테스트**를 수행한다. 기존 모델과 새 모델을 같은 트래픽으로 비교해 새 모델이 실제로 개선됐는지 확인한다. 성능이 조금이라도 나빠지면 배포를 취소하는 편이 현명하다.

5.5 지속적 개선 문화

기술 시스템보다 중요한 게 있다. 팀이 에지 케이스 수집을 꾸준히 실천하는 문화다. 아무리 기술 인프라가 갖춰져도 팀원이 "실패를 보면 그냥 넘어가자"라고 생각하면 소용없다.

실패 케이스를 수치스러운 것으로 보지 않는 문화가 필요하다. 에지 케이스가 발견된다는 것은 시스템이 사용자와 real world에서 상호작용하고 있다는 증거다. 개선의 기회이지 치욕이 아니다.

에지 케이스 수집을 일상 업무에도 통합한다. 스프린트 리뷰에서 에지 케이스 발견 건을 보고하고, 성공 metric에 "해결된 에지 케이스 수"를 포함하며, 재훈련에 따른 성능 개선을 눈에 보이게 만들어 팀이 성과를 체감하게 한다.

5.6 플랫폼적 사고로 마무리

이 책에서 다룬 4가지 실패 유형, 즉 출력 이상, 데이터 드리프트, 파이프라인 실패, 에지 케이스는 개별적으로 대응할 수도 있다. 하지만 진정한 프로덕션

신뢰성은 이것들을 통합적으로 관리할 때 실현된다.

통합 관리의 핵심은 **관찰 가능성(observability)**이다. 시스템 전체에서 일어나는 일을 unified한 시야로 볼 수 있어야 한다. 출력 검증에서 이상이 감지되면 그것이 드리프트의 신호인지, 파이프라인 문제의 영향인지, 새로운 에지 케이스인지 판단할 수 있어야 한다.

이것은 도구 하나의 문제가 아니다. 모니터링, 로깅, 추적(tracing), 알람이 통합된 플랫폼적 사고가 필요하다. 그 플랫폼 위에 팀의 실무가 자리 잡는다. 이 책이 그 실천의 출발점이 되기를 바란다.