# LLM 평가 엔지니어링

프로덕션 시스템을 위한 신뢰할 수 있는 AI 평가 설계

ThakiCloud (Hyojung Han)

# LLM 평가 엔지니어링

프로덕션 시스템을 위한 신뢰할 수 있는 AI 평가 설계

ThakiCloud (Hyojung Han)

# Contents

# 1 평가의 본질: 왜 '출력 좋아 보인다'는 신뢰성의 근거가 되는가

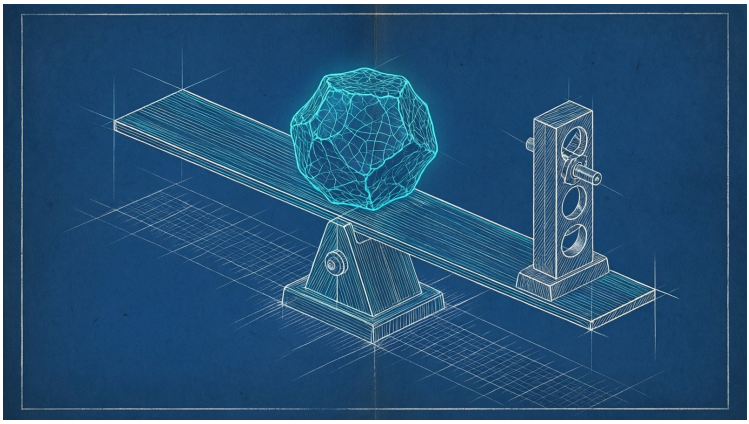

Figure 1.1: 평가의 본질: 왜 '출력 좋아 보인다'는 신뢰성의 근거가 되는가

## 1.1 인상과 시스템적 신뢰의 차이

LLM을 프로덕션에 배포한 팀이 가장 자주 받는 질문은 "이 모델, 실제로 괜찮은 거야?"이다. 돌아오는 대답은 대개 "테스트해봤는데 잘 됩니다" 아니면 "비슷한 입력을 사람이 검토했거든요" 정도다. 언뜻 합리적으로 들리지만, 평가 엔지니어링 관점에서 보면 이 대답들 사이에는 핵심적인 격차가 있다.

첫 번째 격차는 인상의 국소성 문제다. 평가자가 실제로 테스트해본 입력은 전체 입력 분포 중 극히 일부일 뿐이다. 게다가 평가자는 자신이 테스트한 입력에 대해 확증 편향을 갖기 쉽다. "내가 넣어서 성공한 케이스니까 아마 잘 될 거야"라는 추론은 통계적으로 정당화되지 않는다. 성공 케이스를 보면 성공 가능성을 과대평가하고 실패 케이스를 보면 과소평가하는 경향이

있는데, 이 편향은 표본 크기와 무관하게 나타난다.

두 번째 격차는 인간 평가자의 일관성 문제다. 같은 텍스트를 같은 입력으로 두 번 보여줘도 평가자는 매번 같은 점수를 주지 않는다. 연구에 따르면 텍스트 유사도 작업에서 인간 평가자 간 일치도는 0.4~0.6 수준에 머무는 경우가 많다[추정]. 이 정도의 일치도라면 개별 점수 하나만으로는 신뢰하기 어렵다.

세 번째 격차는 맥락 의존성 문제다. 평가자가 "좋다"고 판단한 출력이 실제 배포 환경의 맥락과는 다를 수 있다는 점을 생각해보자. 낮에 테스트한 출력이 야간 배치에서는 같은 프롬프트에도 다르게 반응할 수 있다. 토큰 생성의 확률적 특성 때문에 완전히 동일한 출력을 재현할 수 없기 때문이다. 결론은 명확하다. 인상 기반 평가는 인상 관리에 지나지 않으며, 시스템적 신뢰를 구축하려면 구조화된 평가 프레임워크가 필요하다.

## 1.2 평가의 세 가지 계층

구조화된 평가를 설계할 때는 세 가지 계층을 고려해야 한다. 정밀성(accuracy), 영향성(impact), 안정성(consistency)이다.

정밀성은 출력이 사실적으로 정확한지를 묻는다. 검색 증강 생성(RAG) 시스템이라면 검색 결과와 생성된 텍스트 사이의 사실적 일관성(factual consistency)이 여기에 해당한다. 영향성은 출력이 실제로 유용한 결과를 만드는지를 묻는다. 자동 요약 시스템이라면 그 요약이 의사결정에 실제로 도움이 되는가라는 질문이다. 안정성은 같은 입력을 반복 호출했을 때 출력이 얼마나 일관되는지를 묻는다. 이 세 계층을 모두 충족해야 비로소 "신뢰할 수 있는 시스템"이라 부를 수 있다.

정밀성만 측정하고 영향성은 무시하는 경우가 실무에서 흔하다. 모델이 사실적으로 정확한 답변을 내놓아도 그 답변이 실용적이지 않으면 시스템은 결국 쓰이지 않는다. 고객 서비스 챗봇이 제품 사양을 정확히

인용하더라도 응답 시간이 10초 이상 걸린다면 실제 서비스에서는 외면받는다. 영향성을 고려한다는 것은 기술적 정확성 너머의 사용자 경험까지 설계에 포함시킨다는 뜻이다.

안정성은 특히 프로덕션 환경에서 중요하다. LLM의 확률적 생성 특성 때문에 동일한 프롬프트에도 매번 다른 출력이 나올 수 있다. 이 중 일부는 명백히 틀렸고, 일부는 정답이지만 표현만 다르다. 안정성을 측정한다는 것은 이런 변동의 폭을 정량화하고, 중요한 영역에서는 그 변동폭이 허용 범위 안에 있는지 확인하는 작업이다.

## 1.3 샘플링 바이어스와 적응형 표집 문제

평가 데이터셋을 구축할 때 가장 주의해야 할 것이 샘플링 바이어스다. 가장 흔한 형태는 목표 샘플링(goal sampling)이다. 개발자가 "이 모델이 잘 해야 하는 영역"을 중심으로 테스트 케이스를 만들기 시작하면 해당 영역에서는 높은 성능이 나오지만, 나머지 영역의 성능 격차는 드러나지 않는다.

적응형 표집은 더 교묘한 문제다. 모델이 특정 입력 타입에서 실패하면 개발자는 그 타입에 대한 테스트 케이스를 더 추가한다. 그러면 다음 평가에서 그 타입의 성능이 좋아지고, 이를 보고 "모델이 발전했다"고 오해하게 된다. 실제로는 특정 타입에 과적합됐을 뿐이다. 이를 평가의 적응적 과적합(adaptive overfitting)이라고 부른다.

이 문제를 막으려면 평가 데이터셋과 모델 개발 사이의 독립성을 보장해야 한다. 테스트셋을 먼저 고정한 뒤 그 안에서 성능이 개선되었는지만 측정할 게 아니라, 평가 데이터셋의 분포가 실제 입력 분포를 얼마나 대표하는지부터 확인해야 한다.

실무에서 쓰는 방법 하나는 입력 빈도를 가중한 샘플링이다. 실제 트래픽에서 각 입력 카테고리가 차지하는 비중을 분석하고 그 비중에 비례해 테스트 케이스를 뽑는다. 그러면 평가 결과가 실제 서비스 성능을

더 정확히 예측하게 된다.

## 1.4 평가자 효과 통제하기

인간 평가자의 일관성 문제를 해결하는 데는 몇 가지 구조적 접근이 있다.

첫 번째는 명시적 기준을 세우는 것이다. "이 답변은 좋다"라는 추상적 지시보다 "이 답변은 관련 없는 정보를 포함하고 있다"처럼 구체적인 체크리스트 항목이 평가자 간 일관성을 높인다. 평가 전에 평가자에게 기준을 교육하고, 그 효과가 유지되는지 주기적인 캘리브레이션 검사로 확인하는 편이 좋다.

두 번째는 다중 평가자의 투표를 집계하는 방식을 바꾸는 것이다. 단순 다수결 대신, 평가자 간 불일치가 클 때 그것을 불확실성의 신호로 활용할 수 있다. 불일치가 큰 케이스는 재검토하고 일치도가 높은 케이스만 통과시키는 방식으로 평가의 정밀도를 높인다.

세 번째는 평가 프로토콜의 표준화다. 블라인드 평가(blind evaluation), 즉 평가자가 어떤 모델의 출력인지 모른 채 동일한 기준으로 평가하게 하는 것이 기본이다. 모델 이름이나 버전 정보가 평가에 영향을 준다는 사실은 실험으로 확인됐다. "OpenAI의 새 모델"이라는 정보 없이 같은 출력을 평가하면 점수가 달라진다.

이 세 계층과 편향 통제 방법을 프레임으로 삼으면 "좋아 보인다"는 주관적 인상에서 벗어나 체계적인 신뢰성 평가로 나아갈 수 있다. 다음 장에서는 이 평가 프레임워크를 단일 프롬프트 수준에서 시스템 전체로 확장하는 방법을 다룬다.

# 2 프롬프트 레벨 평가에서 시스템 레벨 평가로: 평가 스코프의 확장

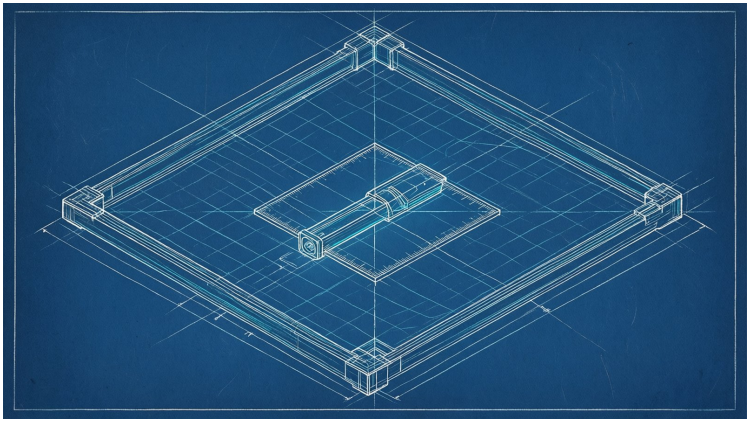

Figure 2.1: 프롬프트 레벨 평가에서 시스템 레벨 평가로: 평가 스코프의 확장

## 2.1 단일 프롬프트 테스트의 한계

개별 프롬프트를 테스트하는 일은 소프트웨어 공학의 유닛 테스트에 견줄 만하다. 유닛 테스트가 없으면 코드의 신뢰성을 가늠할 수 없듯이, 프롬프트 테스트가 없으면 특정 케이스에서 시스템이 어떻게 동작하는지 알 수 없다. 다만 유닛 테스트만으로 시스템 전체의 신뢰성을 보장하지 못하듯, 프롬프트 테스트만으로도 프로덕션 시스템의 동작을 보장할 수는 없다.

한계를 구체적으로 짚어보자. few-shot 예제 5개로 구성된 프롬프트를 테스트한다고 해보자. 5개가 모두 성공하면 개발자는 흔히 프롬프트가 잘 작동한다고 판단한다. 하지만 실제로는 그 5개의 예제가 전체 입력 분포를

대표하지 못하는 경우가 대부분이다. 등장 빈도가 낮은 edge case는 few-shot 예제에 포함될 확률 자체가 낮고, 그래서 애초에 테스트 대상에서 빠진다.

예를 들어 고객 서비스 챗봇에서 "배송 지연 됐어요"라는 입력은 자주 등장하니 테스트하기도 쉽다. 반면 "배송이 2개월 전이었는데 이제 와서 취소하고 싶어요"라는 입력은 드물지만, 시스템의 처리 능력을 검증하는 데 오히려 더 중요한 edge case다. 이런 케이스일수록 시스템이 망가질 가능성이 높은데, 정작 few-shot 예제에서는 좀처럼 찾아보기 어렵다.

이 문제의 핵심은 "내가 테스트한 입력"과 "실제 사용자가 입력하는 입력" 사이의 분포 차이에 있다. 개발자가 테스트하는 입력은 대개 자신의 상상력으로 만들어낸 것이라, 실제 입력 분포에서 보면 편향된 표본에 지나지 않는다.

## 2.2 입력 분포의 계층적 모델링

이 간극을 좁히는 한 가지 접근은 입력 분포를 계층적으로 모델링하는 것이다. 입력 공간을 빈도-중요도 매트릭스로 나누면 평가 자원을 훨씬 합리적으로 배분할 수 있다.

| | 자주 등장 | 드물게 등장 |
|---|---|---|
| **중요도 높음** | Quadrant A: 빈도 높고 영향도 높음 - 가장 우선 평가 | Quadrant C: 드물지만 영향도 높음 - 반드시 평가 |
| **중요도 낮음** | Quadrant B: 자주 보이지만 영향도 낮음 - 가벼운 평가 | Quadrant D: 드물고 영향도 낮음 - 우선순위 최저 |

Quadrant A에 속한 케이스는 실제 트래픽에서 가장 많이 등장하고 사용자 경험에도 직결되므로, 평가의 중심축이 되어야 한다. Quadrant C에 속한

케이스는 드물게 발생하지만 한 번 터지면 시스템 신뢰성에 큰 타격을 준다. 그래서 평가 데이터셋에는 반드시 포함시켜야 한다.

입력 분포를 이렇게 모델링하려면 먼저 실제 프로덕션 로그를 분석해야 한다. 과거 트래픽에서 입력을 카테고리별로 분류하고, 각 카테고리의 빈도와 그 입력이 만들어낸 결과의 영향도를 추정한다. 영향도는 그 지점 이후 사용자가 어떤 행동을 했는지로 측정할 수 있다. 시스템의 출력이 사용자의 다음 행동을 결정하는 데 실제로 기여했는가?

## 2.3 A/B 테스트와 shadow 모드

새로운 모델이나 프롬프트를 프로덕션에 배포하기 전에 성능을 확인하는 방법은 크게 두 가지다. A/B 테스트와 shadow 모드다.

shadow 모드는 기존 프로덕션 시스템 옆에 새 시스템을 나란히 배치하고, 동일한 입력을 두 시스템에 동시에 흘려보내 추론하게 하는 방식이다. 다만 새 시스템의 출력은 사용자에게 보여주지 않고 로그로만 남긴다. 기존 시스템을 건드리지 않으면서 새 시스템의 동작을 검증할 수 있는 안전한 방법인 셈이다.

A/B 테스트는 실제 사용자를 대상으로 두 버전을 비교하는 방식이다. 일정 비율의 사용자에게는 기존 버전을, 나머지에게는 새 버전을 보여주고 그 차이를 관찰한다. shadow 모드보다 더 현실에 가까운 성능을 측정할 수 있지만, 새 버전의 출력이 사용자에게 직접 영향을 미치는 만큼 위험 부담도 따른다.

실무에서 권장하는 패턴은 이렇다. 먼저 shadow 모드로 새 시스템의 성능을 일정 기간 지켜본다. 주요 지표가 기준치를 넘어선다는 것이 확인되면 소규모 A/B 테스트로 전환한다. 소규모에서 별다른 문제가 없다면 A/B 비율을 점진적으로 늘려간다. 문제가 발견되는 순간 즉시 원래 버전으로 rollback한다.

이 패턴은 평가 결과를 production 배포의 트리거로 삼는 구조, 다시 말해 평가가 “deploy or not”을 가르는 게이트가 되는 구조를 만든다. 이것이 3장에서 다룰 자동화 평가 파이프라인의 핵심 설계 원칙이다.

## 2.4 회귀 테스트로 평가 체계를 전환하기

기존 시스템의 동작을 회귀 테스트로 확보해두면, 새로운 변경 때문에 생기는 의도치 않은 성능 저하를 자동으로 잡아낼 수 있다. 다만 LLM 시스템에서 회귀 테스트를 구축하려면 몇 가지 짚어야 할 점이 있다.

회귀 테스트의 기준이 되는 케이스 집합은 시간이 지나면서 계속 갱신되어야 한다. 프로덕션에서 새로운 failure 케이스가 발견되면 그때마다 회귀 테스트 스위트에 추가한다. 그래야 매 배포마다 그 케이스에 대한 최소한의 성능이 보장된다.

두 번째로 짚어야 할 것은 통과와 실패를 가르는 기준이다. 기존 시스템 대비 새 시스템의 성능이 1퍼센트만 개선돼도 통과로 볼 것인가, 아니면 통계적으로 유의미한 차이가 없어도 통과로 볼 것인가? 전자를 택하면 개선 방향이 옳다는 뜻이 되고, 후자를 택하면 기존 수준을 유지하면서 응답 속도나 메모리 사용량 같은 다른 측면을 개선했다는 뜻이 된다. 어느 쪽을 고르든 팀 내 합의가 먼저다.

회귀 테스트를 구축할 때 흔히 저지르는 실수가 있다. 모든 지표를 동시에 통과해야 비로소 통과로 인정하는 방식이다. 이러면 응답 시간 개선과 내용 품질 개선을 한꺼번에 달성해야 하는 부담이 고스란히 팀에게 돌아간다. 실무에서는 지표를 필수 지표와 바람직 지표로 나누는 편이 효과적이다. 필수 지표만 통과하면 회귀 테스트를 통과한 것으로 보고, 바람직 지표는 점진적으로 개선해나가는 방식이다.

이 장에서 살펴본 스코프 확장 원칙은 다음 장에서 다룰 자동화 평가 파이프라인의 설계로 곧장 이어진다. 파이프라인을 구축할 때 이 원칙들을

반영하면, 평가는 단순한 “점검”을 넘어 “배포 결정의 근거”로 기능하게 된다.

# 3 자동화 평가 파이프라인의 설계 원칙

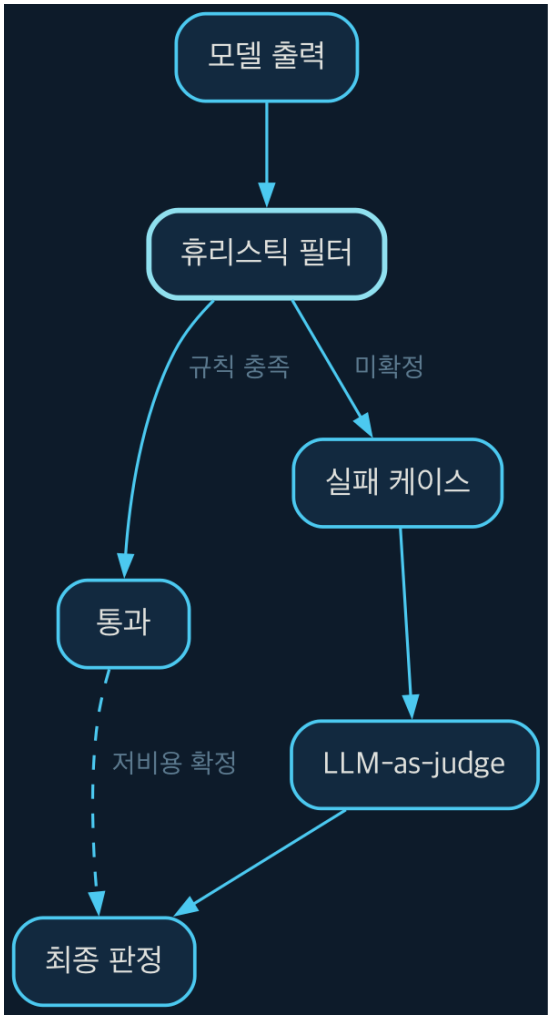


Figure 3.1: 자동화 평가 파이프라인의 설계 원칙

## 3.1 휴리스틱 게이트와 LLM-as-judge의 적절한 분배

자동화 평가 파이프라인을 설계할 때 가장 기본적인 판단은 어느 단계에서 휴리스틱 규칙을 쓰고 어느 단계에서 LLM-as-judge를 쓰느냐이다. 이 분배가 파이프라인의 속도와 품질을 동시에 좌우한다.

휴리스틱 게이트는 빠르지만 Coverage가 제한적이다. 예를 들어 출력에 특정 금지어가 포함됐는지는 간단한 문자열 탐색으로 검사할 수 있다. 응답 시간이 임계값을 초과했는지도 상수 시간 비교로 판단할 수 있다. 이런 검사는 결정론적이고 재현 가능하므로 평가 결과의 신뢰도가 높다. 그러나 이런 유형의 검사는 전체 평가 차원 중 작은 부분만 차지할 뿐이다.

LLM-as-judge는 더 넓은 Coverage를 갖지만 비용이 크고 결과의 일관성에 문제가 있다. 출력의 유창성이나 서로 다른 두 입력에 걸친 응답의 일관성처럼 규칙으로 표현하기 어려운 항목은 LLM이 판단한다. 다만 LLM의 판단이 매번 같지는 않다는 점을 감안해야 한다.

파이프라인에서 이 둘의 조합을 설계할 때 효과적인 패턴은 2단계 구조다. 첫 번째 단계에서 휴리스틱으로 대부분의 케이스를 빠르게 필터링한다. 이 단계에서 실패한 케이스만 두 번째 단계의 LLM-as-judge로 넘어간다. 이렇게 하면 LLM-as-judge의 호출 횟수를 줄이면서도 정밀한 평가를 유지할 수 있다.

구체적인 예시를 보자. 고객 서비스 챗봇의 자동 평가 파이프라인을 만든다고 하자. 첫 번째 단계에서 휴리스틱으로 출력의 길이가 10단어 이하인 경우, 특정 금지어 목록이 포함된 경우, 응답 시간이 3초를 초과한 경우를 걸러낸다. 이 세 가지 검사는 모두 상수 시간 연산이므로 빠르게 동작한다. 통과한 케이스만 두 번째 단계에서 LLM-as-judge가 판단한다. 이때 LLM은 “응답이 고객의 질문을 적절히 해결했는가”를 판단한다.

## 3.2 결정론성 보장

평가 파이프라인의 결과를 믿으려면 같은 코드를 같은 입력으로 반복 실행했을 때 같은 결과가 나와야 한다. 이것을 결정론성이라고 부른다. LLM 시스템에서는 결정론성을 보장하기가 특히 어렵다. 확률적으로 생성하는 특성 때문이다.

LLM은 동일한 프롬프트에도 매번 다른 토큰을 생성할 수 있다. temperature를 0으로 설정해도 하드웨어의 부동소수점 연산 차이나 CUDA 커널의 비결정론 때문에 출력이 달라질 수 있다. 즉 같은 테스트를 두 번 돌렸는데 결과가 다르다고 해서 반드시 모델 탓은 아니다. 평가 환경 자체의 문제일 수 있다.

결정론성을 보장하기 위한 실용적인 방법들을 정리한다.

첫 번째는 시드 고정이다. 테스트를 실행할 때 환경 변수 PYTHONHASHSEED, CUDA_LAUNCH_BLOCKING 등을 특정 값으로 고정한다. 이것만으로 완전한 결정론을 보장하지는 못해도 재현 불가능한 차이는 줄일 수 있다.

두 번째는 Monte Carlo 샘플링을 쓰는 방법이다. 같은 입력을 여러 번 추론해 그 결과 분포를 보는 방식이다. 단일 실행 결과를 신뢰하기보다는 여러 번 실행했을 때 결과가 특정 범위 안에 모이는지를 확인한다. 이것은 특히 안정성을 테스트할 때 유용하다.

세 번째는 평가 결과의 신뢰구간을 함께 보고하는 것이다. 점 추정치만 보고하면 그 값의 불확실성을 알 수 없다. 95% 신뢰구간을 함께 보고하면 결과가 실제로 의미 있는 차이인지 아닌지를 판단할 수 있다.

## 3.3 지표 간 상관관계 분석

평가 파이프라인에서 지표를 여러 개 추적한다면 그 지표들 사이의 관계를 분석해야 한다. 상관관계가 높은 지표끼리는 하나가 오르면 다른 것도 따라 오르는 경향이 있어 따로 추적할 이유가 약하다. 반대로 상관관계가 낮거나 음의 상관관계를 갖는 지표는 서로 다른 차원을 측정하는 셈이라 모두 중요하다.

예를 들어 응답의 factual accuracy와 유창성 사이에는 일반적으로 양의 상관관계가 있다. 사실관계가 정확한 답변일수록 표현도 자연스러운 경향이 있다는 뜻이다. 그러나 때로는 이 둘 사이에 트레이드오프가 있을 수 있다. 매우 정확한 응답이 조금 어색한 표현을 사용할 수 있다. 이런 경우라면 둘을 별도 지표로 계속 추적할 필요가 있다.

상관관계 분석을 하는 또 다른 이유는 지표 선택의 합리성을 검사하기 위해서다. "새 프롬프트를 적용한 뒤 모든 지표가 개선되었다"고 주장해도 그 지표들이 서로 고도로 상관한다면 실제로는 하나의 현상만 여러 번 측정했을 가능성이 있다. 그러면 개선의 폭이 과대평가될 수 있다.

실무에서는 평가 파이프라인을 구축한 뒤 적어도 한 번은 지표 간 상관관계를 분석하고 그 결과로 지표 수를 합리적인 규모로 줄이는 작업을 권장한다. 지표가 20개를 넘으면 결과의 해석이 어려워지고, 팀이 실제로 활용하지 않는 지표는 유지할 이유가 없다.

## 3.4 파이프라인 슬로우 포인트 제거

평가 파이프라인이 실용적이려면 실행 시간이 합리적이어야 한다. 평가에 1시간이 걸리면 커밋마다 실행하기 어렵고, 그러면 파이프라인의 가치가 반감한다. 흔히 발생하는 슬로우 포인트를 확인하고 처리한다.

가장 흔한 슬로우 포인트는 LLM-as-judge의 호출 횟수다. 휴리스틱

필터 없이 모든 출력을 LLM-as-judge에 통과시키면 평가 시간이 급격히 증가한다. 앞서 설명한 2단계 구조로 이 문제를 완화할 수 있다.

두 번째 슬로우 포인트는 네트워크 I/O다. 평가 과정에서 외부 API를 호출할 때 그 호출 횟수와 네트워크 지연을 최소화해야 한다. 예를 들어 여러 입력의 출처 확인을 각각 별도 API 호출로 처리하면 호출 횟수가 늘어나 네트워크 지연이 누적된다. 가능하면 batch API를 활용하고 그 결과를 캐싱해 반복 호출을 막는다.

세 번째 슬로우 포인트는 평가 결과의 저장소 접근이다. 평가할 때마다 전체 결과를 데이터베이스에 기록하면 그 쓰기 비용이 쌓인다. 평가가 완료된 뒤 한 번에 기록하는 배치 쓰기 방식을 적용하면 이 비용을 줄일 수 있다.

슬로우 포인트 분석에는 프로파일링이 필수다. 파이프라인의 각 단계에서 걸린 시간을 로깅하고 그 비율을 시각화하면 병목이 어디인지 명확히 드러난다. 추측으로 최적화하지 말고 데이터를 기반으로 개선 대상을 골라야 한다.

이 장에서 설명한 자동화 평가 파이프라인의 설계 원칙들은 다음 장에서 설명하는 인간 피드백 통합과 맞물려 작동한다. 자동화 파이프라인이 평가의 속도와 재현성을 제공하고, 인간 피드백이 자동화 파이프라인의 품질을 교정하는 구조를 만들어야 한다.

# 4 인간 피드백의 전략적 통합: 라벨링 비용을 최소화하는 설계

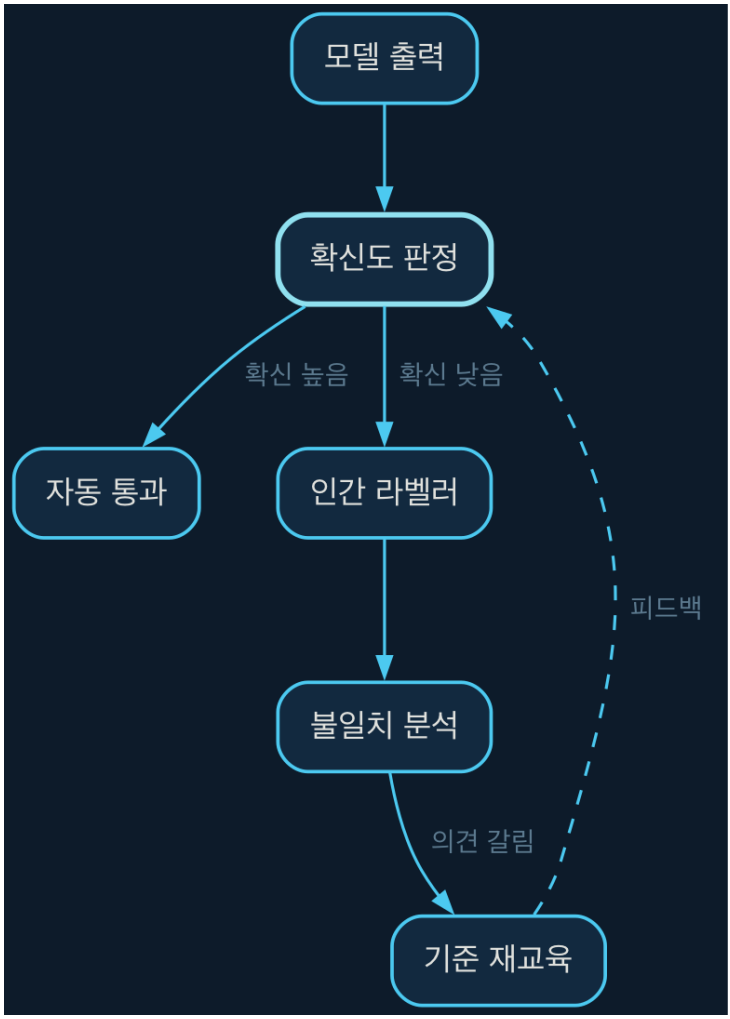


Figure 4.1: 인간 피드백의 전략적 통합: 라벨링 비용을 최소화하는 설계

## 4.1 라벨링 예산의 최적 배분

인간 피드백은 비용이든 시간이든 유한한 자원이다. 라벨을 무제한으로 모을 수 없기에, 이를 어떻게 배분하느냐가 관건이다. 무작정 모으기보다 전략적으로 나눠 써야 한다.

가장 기본적인 원칙은 불확실성이 큰 케이스에만 라벨을 요청하는 것으로, 이를 uncertainty sampling이라 부른다. 모델이 이미 자신 있어 하는 케이스까지 라벨을 모으면 리소스 낭비다. 확신이 없는 출력, 또는 여러 해석 중 하나만 골라야 하는 edge case에 라벨을 요청할 때 정보 가치가 가장 높다.

구체적인 구현 방법은 이렇다. 자동화 평가 파이프라인에서 각 케이스마다 모델의 confidence 점수를 기록해두고, confidence가 높으면 그대로 통과시키고 낮은 케이스만 인간 평가자에게 넘긴다. confidence를 무엇으로 정의할지가 관건인데, 모델의 token probability를 쓰거나 복수 모델의 출력을 비교하거나 여러 turn에 걸친 출력의 일관성을 검사하는 방법 등이 쓰인다.

예를 들어 질문 응답 시스템에서 top-1 응답과 top-2 응답의 확률 차이가 크면, 모델이 그 질문에 확신을 갖고 있다고 본다. 반대로 차이가 작으면 모델이 여러 답 사이에서 망설이고 있다는 뜻이므로, 그 케이스를 인간에게 넘긴다.

이 접근은 라벨링 비용을 크게 줄이면서도 모델이 실제로 놓치는 부분에만 집중해서 확인할 수 있다는 장점이 있다. confidence가 높은 케이스는 라벨을 모아봐야 대부분 기존과 같은 판단이 나오므로, 그 라벨은 새로운 정보를 주지 못한다.

## 4.2 절대 평가보다 상대 평가의 효율성

절대 평가, 즉 “이 출력은 5점 만점에 몇 점인가”라는 질문 형태는 직관적이지만 통계적으로는 비효율적이다. 절대 점수는 평가자 간 일관성을 확보하기가 어렵다. 어떤 평가자는 엄격하고 어떤 평가자는 관대한 경향을 보이는데, 같은 출력을 두고 한 사람은 4점을 주고 다른 사람은 2점을 줄 수도 있다.

상대 평가, 즉 두 출력 중 어느 쪽이 더 나은지 묻는 방식은 이 문제를 완화해준다. pairwise 비교에서는 평가자가 직접 점수를 매기는 대신 상대적 순서만 정하면 되므로, 평가자마다 다른 척도를 쓰는 데서 오는 영향이 줄어든다. “A와 B 중 어느 쪽이 더 나은가요”라는 질문에는 평가자 A와 평가자 B가 서로 다른 기준을 쓰더라도, 같은 쌍을 두고 같은 선택을 할 확률이 절대 평가보다 높다.

랭킹 기반 평가도 비슷한 효과를 낸다. N개의 출력을 순서대로 정렬하게 하면 0에서 5까지 같은 척도를 따로 정할 필요 없이 순서만 매기면 된다. 평가자의 인지적 부담이 줄어들고, 그만큼 더 일관된 데이터가 모인다.

실무에서는 두 방식을 조합하는 편이 효과적이다. 먼저 모든 출력을 pairwise 비교로 대략적인 우위부터 가리고, 그렇게 좁혀진 하위 집단만 절대 평가로 다시 살펴 차이를 정밀하게 측정한다. 이렇게 하면 평가의 거칠기와 세밀함을 상황에 맞게 조절할 수 있다.

## 4.3 레이블러 간 agreement 관리

여러 레이블러에게 같은 데이터의 라벨을 맡기면 그 결과가 반드시 일치하지는 않는다. 이 disagreement를 어떻게 처리하느냐에 따라 데이터 품질이 갈린다.

단순 다수결 방식은 간단하지만 그 뒤에 숨은 정보를 살리지 못한다. 3명의

레이블러 중 2명이 같은 답을 했다면 그 2명의 선택을 따르면 그만이다. 다만 2명이 같은 선택을 했더라도 그 둘 사이의 의견 강도는 다를 수 있다. 한 명은 A가 확실히 낫다고 봤고 다른 한 명은 망설이다가 A를 골랐을 수도 있는데, 다수결 방식은 이런 정보를 놓친다.

더 정밀한 방법은 disagreement의 크기 자체를 불확실성 신호로 쓰는 것이다. 평가자들의 의견이 크게 갈린 케이스는 출력이 모호하거나 평가 기준이 불명확하거나, 아니면 모델 성능을 개선해야 할 영역일 가능성이 있다. 이런 케이스를 따로 모아 분석하면 라벨링 기준의 문제를 찾아내거나 모델의 약점을 짚어낼 수 있다.

실무에서는 전체 평가 케이스 중 disagreement 비율이 높은 상위 10퍼센트를 추려 분석한다. 그 결과에 따라 대응이 갈리는데, 라벨링 기준이 모호해서 disagreement가 생긴 것이라면 평가자에게 구체적인 기준을 다시 교육하고, 모델이 실제로 edge case에서 약한 것이라면 그 영역을 중점적으로 개선한다.

레이블러별로 동의율을 추적하는 것도 도움이 된다. 다른 레이블러들과 체계적으로 다른 의견을 내는 레이블러가 있다면, 그 판단을 의심해볼 필요가 있다. 그에게 피드백을 주거나, 그의 라벨에 낮은 가중치를 적용하는 방법을 쓸 수 있다.

## 4.4 비동기 피드백 수집 시스템

실시간 시스템에서 인간 피드백을 쓰려면 수집과 반영 사이의 지연 시간을 관리해야 한다. 사용자가 시스템 출력을 접한 뒤 피드백을 남기는 일은 대개 비동기로 일어난다. 그 자리에서 바로 남기기보다 나중에 앱을 다시 열어 남기는 경우가 흔하다.

시스템을 설계할 때 고려할 점이 있다. 피드백을 사용자에게 대뜸 요구하면 응답률이 낮지만, 시스템이 요청 타이밍을 잘 설계하면 응답률이 오른다.

예컨대 시스템 출력이 사용자의 기대와 어긋났을 법한 순간에 피드백을 요청하면 효과적이다. 모델의 confidence가 낮은 출력을 보여줬을 때나 edge case로 분류된 입력이었을 때 사용자에게 확인을 구하는 편이 좋다.

피드백 데이터의 품질 관리도 중요하다. 사용자에게서 모은 피드백에는 노이즈가 섞이기 마련이다. 의도적으로 악의를 담은 피드백, 무심코 클릭한 피드백, 맥락을 놓쳐 잘못 남긴 피드백이 뒤섞일 수 있어서 이를 걸러내는 전처리 단계를 자동화 파이프라인에 넣어야 한다.

이 장에서 다룬 인간 피드백의 전략적 통합은 다음 장에서 다룰 모델 iteration 개선과 곧바로 이어진다. 인간 피드백은 그저 "더 많은 데이터"를 모으는 일이 아니라, 모델이 실제로 개선되어야 할 영역을 찾아내는 데 초점을 맞춰야 한다.

# 5 모델 iteration 개선의 실효성 검증: 개선이 실제로 개선인지 증명하는 방법

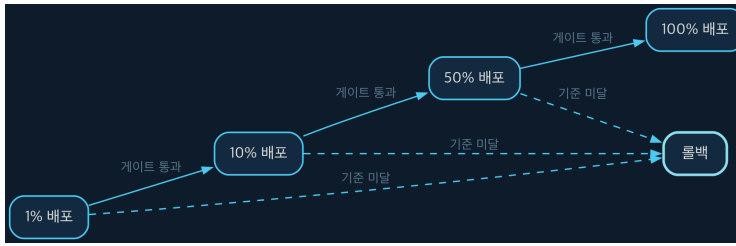


Figure 5.1: 모델 iteration 개선의 실효성 검증: 개선이 실제로 개선인지 증명하는 방법

## 5.1 사전-사후 분석의 표준 절차

새로운 프롬프트나 모델 버전을 배포한 뒤 “개선되었다”고 주장하려면 증거를 대야 한다. 증명 없는 개선 선언은 믿음일 뿐이다. 사전-사후 분석은 개선 여부를 객관적으로 입증하는 기본 프레임워크다.

절차부터 정리해보자. 먼저 평가 기준을 확정한다. 어떤 지표로 개선을 판단할 것인가는 모델 iteration을 시작하기 전에 이미 합의되어 있어야 한다. “응답이 더 정확해졌다”는 모호한 표현이고 “factuality 지표가 85점에서 89점으로 올랐다”는 구체적인 표현이다. 지표를 구체적으로 고르는 일이 사전-사후 분석의 출발점이 된다.

두 번째로 동일한 테스트 셋을 정의한다. 사전 테스트와 사후 테스트가 서로 다른 입력으로 돌아갔다면 차이의 원인이 모델 변경인지 입력 차이인지 구분할 수 없다. 그래서 두 테스트는 반드시 같은 테스트 셋으로 실행해야 한다. 테스트 셋은 실제 입력 분포를 대표하도록 구성해야 하며, 이는 앞선

장에서 다룬 입력 분포 모델링과 이어진다.

세 번째로 측정 환경을 동일하게 맞춘다. temperature, 시드, 하드웨어 사양, 라이브러리 버전 등이 사전 테스트와 사후 테스트에서 같아야 한다. 환경 차이로 생긴 variation을 개선으로 착각할 수 있다.

네 번째로 결과를 기록하고 공유한다. 개선 폭과 통계적 신뢰구간을 함께 보고해야 한다. "89점이 되었다"만으로는 정보가 부족하다. "89점(95% CI: 87~91점)"이라고 쓰면 훨씬 많은 것을 말해준다. 신뢰구간이 넓다는 것은 개선 여부가 그만큼 불확실하다는 뜻이므로 해석도 달라져야 한다.

## 5.2 통계적 유의성 versus 실질적 유의성

사후 테스트 결과가 사전보다 높게 나왔다고 해서 곧바로 "개선"이라 부르기 전에 두 가지를 구분해야 한다. 통계적 유의성과 실질적 유의성이다.

통계적 유의성은 이 차이가 우연히 나왔을 경우 관찰될 확률이 기준치 이하라는 주장이다. 보통 p-value가 0.05 이하면 통계적으로 유의미하다고 본다. 다만 이는 순수하게 수학적인 판단 기준일 뿐이다. p-value 0.04가 실제로 무엇을 의미하는지는 따로 따져봐야 한다.

실질적 유의성은 그 차이가 사용자 경험이나 업무 성과에 실제로 영향을 미칠 만큼 크다는 뜻이다. 예를 들어보자. 광고 문구 생성 모델에서 CTR(클릭률)이 2.1%에서 2.15%로 올랐다. 통계적으로는 유의미한 차이일 수 있다. 하지만 이 차이가 업무적으로도 의미가 있을까? 0.05%p 차이는 연간 목표를 좌우할 정도는 아닐 수 있다.

실무에서는 이 두 유의성을 함께 판단하는 프레임워크를 쓴다. 개선이라고 선언하려면 먼저 통계적으로 유의미한 차이가 있어야 하고, 그 위에서 실질적 유의성 기준도 충족해야 한다. 실질적 유의성의 기준은 팀과 업무마다 다르므로 사전에 합의한 임계치가 필요하다.

## 5.3 개선과 함께한 회귀의 탐지

모델 iteration의 주요 위험 중 하나는 목표로 삼지 않은 지표에서 성능이 떨어지는 것이다. 이를 회귀라 부른다. 사전-사후 분석을 설계할 때는 이 회귀를 탐지하는 일이 핵심 과제가 된다.

예를 들어 모델의 유창성을 개선하려고 학습 데이터를 바꿨다고 하자. 그 결과 새 모델은 더 자연스러운 응답을 내놓지만 사실적 정확성은 떨어졌다. 이것이 의도하지 않은 회귀다. 평가에서 유창성 지표만 추적했다면 이 회귀는 눈에 띄지 않는다.

이 문제를 막으려면 다차원 지표 체계를 미리 갖춰둬야 한다. 주요 지표들 사이에 상관관계가 없다면 한 지표의 개선이 다른 지표의 저하와 trade-off 관계에 놓일 수 있다. 그러므로 목표 지표뿐 아니라 관련 지표도 함께 모니터링해야 한다.

구체적인 모니터링 구조는 다음과 같다. 먼저 모델을 바꾸기 전에 모든 지표의 기준선을 설정한다. 모델을 바꾼 뒤에는 같은 테스트 셋으로 모든 지표를 다시 측정한다. 사전에 설정한 기준선과 비교해 목표 지표의 개선 폭과 다른 지표들의 변화량을 함께 살핀다. 목표하지 않은 지표에서 유의미한 저하가 나타났다면 그것을 회귀로 분류하고 원인을 분석한다.

## 5.4 세마포어 배포 패턴

개선을 확인했다고 해도 곧바로 전체에 배포하는 것은 위험하다. 미처 살피지 못한 구석에서 실패가 터질 수 있기 때문이다. 세마포어 배포는 변경을 전체에 적용하기 전에 소규모로 먼저 효과를 확인하는 단계적 접근법이다.

단계를 설명하면 이렇다. 먼저 1% 배포로 시작한다. 전체 트래픽의 1%에만 새 버전을 적용하고 나머지 99%는 기존 버전을 유지한다. 이 단계에서

에러율, 지연 시간, 주요 지표를 모니터링한다. 문제가 없으면 10% 배포로 넓히고, 이어서 50%, 100% 순서로 확대한다. 각 단계에서 정해진 기간 동안 지켜보고 문제가 발견되면 이전 단계로 곧바로 rollback한다.

이 패턴의 핵심은 단계마다 있는 Gate다. 1%에서 10%로 넓히기 전에 1% 단계 데이터가 기준을 충족했는지 반드시 확인해야 한다. 기준을 충족하지 못하면 rollback하고 원인을 분석한 뒤 다음 iteration을 시작한다.

이 구조를 갖추면 개선이 실제로 개선인지 판단하는 신뢰도가 크게 올라간다. 사전-사후 분석으로 개선 여부를 판단하고, 세마포어 배포로 실제 환경에서의 안정성을 확인하며, 회귀 탐지로 의도하지 않은 부작용을 조기에 발견한다. 이 세 가지가 함께 작동할 때 모델 iteration은 막연한 희망이 아니라 데이터에 근거한 의사결정 과정이 된다.