불확실한 AI를 제어가능하게

프로덕션 AI 신뢰성 패턴 네 가지

ThakiCloud (Hyojung Han)

불확실한 AI를 제어가능하게

프로덕션 AI 신뢰성 패턴 네 가지

ThakiCloud (Hyojung Han)

Contents

1 출력을 예측 불가능하게 만드는 세 가지 원인

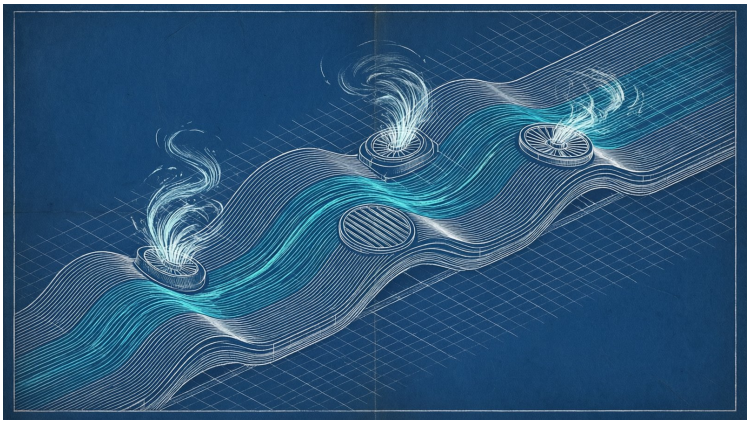

Figure 1.1: 출력을 예측 불가능하게 만드는 세 가지 원인

1.1 확률의 본질

LLM은 다음 토큰을 확률적으로 선택하는 기계다. 같은 질문에 같은 모델이 매번 같은 답을 주지는 않는다. temperature를 0으로 고정해도 아키텍처 수준의 비결정성은 완전히 사라지지 않는다. 토큰 선택이 단어 단위가 아니라 하위 토큰 단위에서 이루어지기 때문이다. 문자열 "안"은 하나의 토큰이 아니라 바이트 쌍으로 분해될 수 있고, 그 분해 경로는 모델마다 다르다.

이 비결정성이 프로덕션 시스템에서 출력을 예측 불가능하게 만드는 첫 번째 근본 원인이다. 소프트웨어는 입력과 출력이 명확해야 안전하게 운영할 수 있는데, AI 모델은 같은 입력에도 다른 출력을 내놓을 수 있다. 이 차이를 수용하지 못하면 시스템 전체가 비정상적으로 동작한다.

1.2 프롬프트 민감도

프롬프트를 거의 그대로 두고 단어 하나만 바꿔도 결과가 극단적으로 달라지는 경우가 놀라울 만큼 흔하다. 한 단어를 바꾸거나 문장 하나를 추가하는 것만으로 모델의 답변이 완전히 뒤바뀐다. 이것이 프롬프트 민감도이며, 엔지니어 입장에서는 예측 불가능성의 또 다른 원천이다.

이 특성이 위험한 이유는 시스템이 의도하지 않은 방향으로 동작할 수 있어서다. 프롬프트를 검증할 때 모든 가능한 변형을 테스트하지 않으면 운영 중 예기치 않은 출력을 만나게 된다. 입력의 미세한 차이가 출력의 극적인 차이를 만들기 때문에 디버깅도 그만큼 까다로워진다.

1.3 문맥 경계 밖의 거짓말

모델은 학습 데이터에 없는 지식을 그럴듯하지만 사실이 아닌 내용으로 만들어 낼 수 있다. 이런 현상을 거짓말 생성이라고 부른다. 모델이 모르는 것을 모른다고 말하는 대신 그럴듯하지만 잘못된 정보를 지어내는 것이다.

이 현상은 모델이 문맥에서 제공한 정보를 사용하지 못하거나 학습 데이터의 특정 패턴에 과적합될 때 나타난다. 프로덕션에서는 사용자가 의도치 않은 질문을 던졌을 때 거짓말이 만들어질 수 있다. 그 정보를 그대로 신뢰하고 시스템을 운영하면 사용자에게 잘못된 안내를 제공하게 된다.

확률적 본질, 프롬프트 민감도, 거짓말 생성이라는 이 세 원인은 서로 독립적이지 않다. 한 원인이 다른 원인을 촉발할 수 있다. 예를 들어 프롬프트 민감도 때문에 거짓말이 더 자주 발생할 수 있다. 이 상호작용을 이해해야 예측 가능한 AI 시스템을 만드는 첫걸음을 뗄 수 있다.

1.4 시스템을 설계할 때 받아들여야 할 전제

이 세 가지 원인을 인식하면 시스템 설계의 전제가 명확해진다. AI 출력이 항상 올바를 것이라고 가정해서는 안 된다. 출력이 유효한지 검증하는 계층이 반드시 필요하다. 재시도가 항상 유익한 것도 아니다. 거짓말은 완전히 없앨 수 없지만 감지하고 대응할 수는 있다.

기존 소프트웨어 공학 관습 중 일부는 AI 시스템에도 유효하지만, 완전히 동일하게 적용되지는 않는다. AI는 확률적 요소이며 그 특성을 고려한 설계가 필요하다.

2 구조화된 출력으로 불확실성을 계약으로 만들기

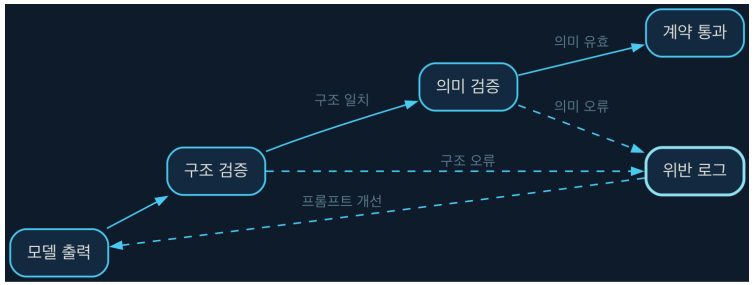

Figure 2.1: 구조화된 출력으로 불확실성을 계약으로 만들기

2.1 왜 자유 형식이 위험한가

AI가 자유 형식으로 답을 내놓으면 프로그램이 그 답을 자동으로 처리하기 어렵다. 자연어로 적힌 답변은 파싱이 까다롭고, 출력이 매번 같은 구조를 갖는다는 보장도 없다. 시스템의 다른 부분이 이 출력을 쓰려면 일정한 계약이 있어야 한다. 계약이 없으면 출력이 바뀔 때마다 다음 단계 시스템이 멈추거나 엉뚱하게 동작한다.

구조화된 출력은 이 문제를 근본적으로 해결한다. 모델이 반환하는 정보의 형태를 미리 정의해두면 시스템은 그 구조를 전제로 다음 처리를 설계할 수 있다. 출력이 계약과 어긋나면 오류로 처리하면 그만이다.

2.2 세 가지 강제 기법

2.2.1 JSON 모드

대부분의 LLM API는 JSON 모드를 지원한다. 응답 형식을 JSON으로 지정하면 모델은 그 구조에 맞는 텍스트를 만들어낸다. 이것만으로도 자유 형식이 지닌 위험은 크게 줄어든다. 다만 모델이 완벽한 JSON을 항상 만들어내는 것은 아니다. 구조는 따르지만 내용 일부가 틀리거나, 문법적으로 유효하지 않은 JSON을 내놓는 경우도 있다.

그래서 JSON 모드만으로는 부족하다. 검증 계층이 반드시 뒤따라야 한다.

2.2.2 도구 호출

모델에게 명시적으로 정의된 도구를 호출하도록 하는 방식이다. 도구의 입력과 출력 스키마가 곧 계약이 된다. 모델은 스키마에 맞는 파라미터를 만들어야만 도구를 호출할 수 있다. JSON 모드보다 강제력이 더 세다. 도구의 스키마가 바뀌면 모델의 호출 방식도 자동으로 따라 바뀌기 때문이다.

도구 호출은 에이전트 시스템에서 특히 쓸모가 크다. 여러 도구를 순서대로 호출하며 작업을 완성하는 구조에서는, 각 도구의 계약이 명확할수록 전체 흐름도 예측하기 쉬워진다.

2.2.3 검증을 동반한 파싱

어떤 기법을 쓰든 검증은 빠질 수 없다. 모델이 돌려준 출력을 검증 라이브러리로 확인하고, 계약에 어긋나면 곧바로 오류로 처리한다. 이 검증 계층이 없으면 잘못된 출력이 소리 없이 시스템에 스며든다.

검증 라이브러리로는 Python의 pydantic, JavaScript의 zod, Go의 gojsonschema 등을 꼽을 수 있다. 스키마를 먼저 정의한 뒤, 그 스키마에

맞춰 출력을 검증한다.

2.3 계약 위반을 잡아내는 구조

검증은 한 계층이 아니라 두 계층으로 나눠 운영하는 편이 효과적이다. 첫 번째 계층은 구조를 확인한다. 예상한 키가 있는지, 값의 타입이 맞는지를 본다. 두 번째 계층은 의미적 유효성을 확인한다. 값이 합리적인 범위 안에 있는지, 다른 필드와 충돌하지는 않는지를 살핀다.

이렇게 나누면 검증이 실패한 원인을 더 정확히 짚어낼 수 있다. 구조적 오류는 모델에게 주는 출력 형식 지시를 강화하면 바로잡을 수 있지만, 의미적 오류는 프롬프트를 바꾸거나 검증 로직을 추가해야 고칠 수 있다.

2.4 시스템의 다른 부분과 연결하기

계약 위반을 그저 오류로 처리하고 버리면 개선할 기회를 놓친다. 위반이 발생했을 때 이를 기록하고 분석하는 체계가 필요하다. 위반이 생긴 프롬프트와 출력을 저장해두고, 어떤 유형의 위반이 자주 일어나는지를 추적한다.

이런 데이터가 쌓이면 패턴이 드러난다. 특정 프롬프트 패턴이 계약 위반을 자주 낳는다면 그 프롬프트를 손보고, 특정 필드에서 검증이 자주 실패한다면 그 필드의 정의를 다시 검토한다. 이 피드백 루프가 없으면 같은 종류의 오류가 계속 반복된다.

2.5 단계별 구현 순서

구현은 작은 단계부터 시작하는 편이 좋다. 먼저 응답의 핵심 필드 세 개를 정하고 구조화된 출력을 요청한다. 그 필드에 대한 검증만 먼저 구현해본다. 이것이 제대로 동작하는지 확인한 뒤 필드를 하나씩 늘려간다.

이 순서를 따르면 검증 로직과 시스템 다른 부분의 통합을 점진적으로 진행할 수 있다. 한 번에 모든 필드를 정의하고 검증하면 문제가 어디서 생기는지 파악하기 어렵다. 작은 계약부터 시작해 점차 넓혀가는 편이 실패를 줄이는 길이다.

3 재시도와 완전 실패 사이의 전략적 선택

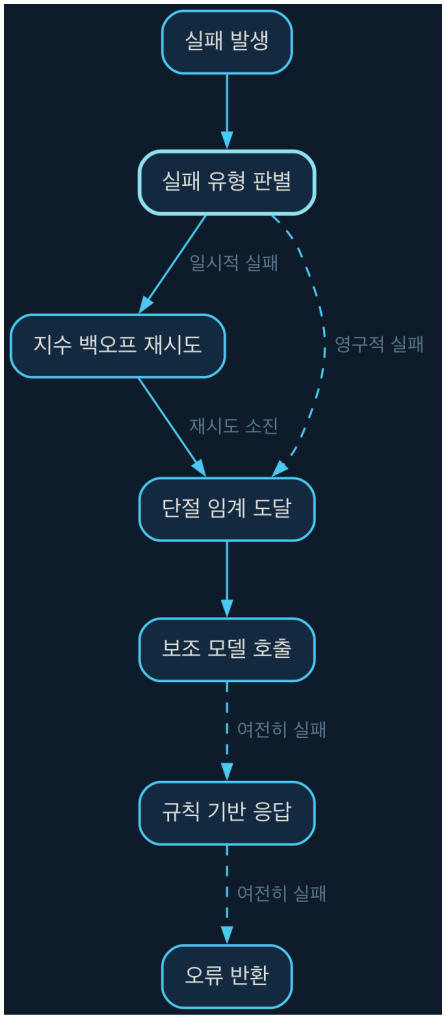

Figure 3.1: 재시도와 완전 실패 사이의 전략적 선택

3.1 재시도가 항상 답은 아닌 이유

AI 모델 호출이 실패하면 재시도부터 하고 싶어진다. 네트워크 오류나 일시적 문제라면 재시도로 풀리는 경우가 많기 때문이다. 하지만 모든 실패가 재시도로 고쳐지지는 않는다. 모델이 거짓 정보를 만들어내거나 잘못된 판단을 내렸다면, 다시 호출한다고 그 문제가 사라지는 것은 아니다. 오히려 다른 방식으로 틀린 답이 돌아올 수도 있다.

여기에 재시도의 함정이 있다. 재시도가 문제를 해결하지 못했는데도 시스템은 마치 해결된 것처럼 계속 동작한다. 사용자 눈에는 정상 답변이 온 것처럼 보이지만, 실제로는 또 다른 유형의 잘못된 답이 온 것뿐이다.

그래서 재시도를 설계할 때는 의미 있는 실패와 의미 없는 실패를 구분해야 한다. 의미 있는 실패만 재시도하고, 의미 없는 실패는 곧바로 완전 실패로 처리한다.

3.2 실패 유형 분류

AI 모델 호출의 실패는 크게 세 갈래로 나뉜다.

먼저 일시적 실패가 있다. 네트워크 오류, 모델 서버의 순간적 과부하, Rate Limit 등으로 생기는 실패이며 시간이 지나면 풀리는 경우가 많아 재시도가 효과적이다.

다음은 영구적 실패다. API 키가 잘못됐거나 모델이 해당 작업을 지원하지 않을 때 일어난다. 이런 실패는 재시도해도 결과가 똑같이 나오므로 재시도의 의미가 없다.

마지막은 결과 실패다. 모델이 응답은 돌려줬지만 그 응답이 기대와 다른 경우다. 구조화된 출력을 요청했는데 형식이 틀렸거나, 거짓 정보가 섞였거나, 판단 자체가 잘못됐을 때 여기 해당한다. 이 경우 재시도하면 다른 결과가 나올 수 있지만, 같은 문제가 그대로 반복될 수도 있다.

3.3 지수적 백오프와 잔여 실패율

재시도를 할 때는 지수적 백오프가 표준으로 쓰인다. 첫 번째 재시도는 1초 후, 두 번째는 2초 후, 세 번째는 4초 후처럼 대기 시간을 늘려가는 방식이다. 이렇게 하면 모델 서버에 부담을 주지 않으면서 재시도가 한꺼번에 몰리는 것도 막을 수 있다.

문제는 재시도를 몇 번 거치고도 실패할 확률이 여전히 남는다는 점이다. 첫 시도의 실패율이 10퍼센트라면 세 번 재시도해도 세 번 모두 실패할 확률은 0.1퍼센트다. 낮아 보이지만, 그 0.1퍼센트가 운영 현장에서는 큰 문제로 번지기도 한다.

재시도를 설계할 때는 이 잔여 실패율까지 염두에 둬야 한다. 재시도를 전부 소진하고도 실패한다면 그 다음을 어떻게 처리할지 미리 정해둬야 한다. 사용자에게 오류를 보여줄지, 기본값을 돌려줄지, 다른 경로로 대안 처리할지를 말이다.

3.4 단절로 전환하는 판단 기준

재시도를 계속할지 여기서 끊을지를 판단하는 기준은 미리 정해둬야 한다. 기준이 없으면 실패가 이어질 때마다 재시도만 반복하게 되고, 그 사이 사용자는 아무 응답도 받지 못한다.

한 가지 방법은 시도 횟수와 경과 시간의 합계에 상한을 두는 것이다. 예컨대 재시도 3회, 총 경과 시간 10초를 상한으로 잡고 둘 중 하나라도 먼저 도달하면 재시도를 멈추고 단절 처리한다.

다른 방법은 실패 유형을 그때그때 판단하는 것이다. 일시적 실패라면 재시도를 이어가고, 영구적 실패나 결과 실패라면 곧바로 끊는다. 이 방식이 더 정교하지만 판단 로직은 그만큼 복잡해진다.

어느 방법을 쓰든 단절 이후의 동작은 반드시 미리 정의해둬야 한다.

단절이 일어났을 때 시스템이 멈추는 것보다는, 미리 정해둔 안전한 응답을 돌려주는 편이 낫다.

3.5 대안 처리 설계

단절 뒤에 돌려줄 안전한 응답을 정의하는 작업을 대안 처리 설계라고 부른다. 보통 세 단계로 구성한다.

가장 먼저 시도하는 것은 모델을 바꾸는 방법이다. 주 모델에서 문제가 생기면 보조 모델을 호출한다. 보조 모델은 주 모델과 다르므로 같은 문제가 반복되지 않을 가능성이 있다. 다만 보조 모델의 응답 품질이 다를 수 있으므로 그 차이가 허용 범위인지 먼저 따져봐야 한다.

그다음은 규칙 기반 응답을 돌려주는 방법이다. AI 모델을 거치지 않고 미리 정의한 로직으로 응답을 만든다. 품질은 떨어지지만 적어도 시스템이 멈추지는 않는다.

마지막은 오류를 반환하는 것이다. 앞의 두 단계가 모두 실패했을 때 시스템이 오류를 돌려준다. 최악의 경우이긴 해도, 잘못된 정보를 돌려주는 것보다는 낫다.

세 단계를 전부 구현하려면 부담이 될 수 있다. 시스템의 중요도와 사용 빈도에 맞춰 필요한 수준만 구현해도 무방하다.

4 관측 가능한 AI 시스템으로 건전하게 운영하기

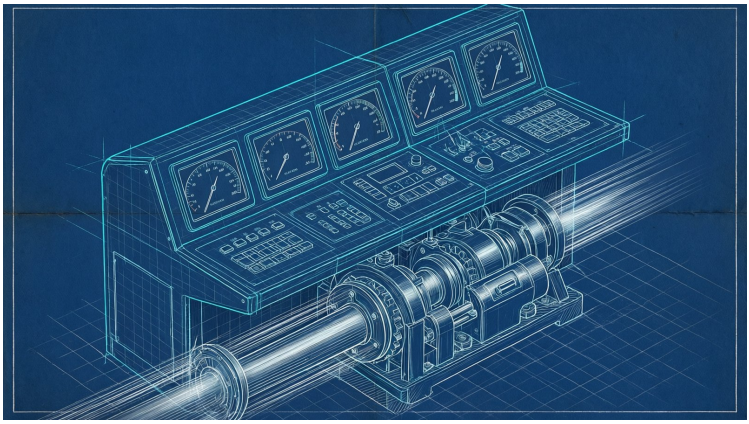

Figure 4.1: 관측 가능한 AI 시스템으로 건전하게 운영하기

4.1 왜 옵저버빌리티가 중요한가

AI 시스템은 전통적인 소프트웨어와 다르다. 코드의 논리가 명확하고 출력이 보장되는 시스템에서는 로그와 메트릭만으로 충분할 때가 많다. 하지만 AI 모델은 확률적 요소가 작용해서 출력이 언제나 보장되지는 않는다. 이 불안정성을 감지하고 대응하려면 시스템의 행위를 자세히 관측할 수 있어야 한다.

옵저버빌리티는 시스템의 내부 상태를 외부 출력에서 추측할 수 있는 능력을 뜻한다. 전통적인 모니터링이 시스템의 정적인 상태를 확인하는 일이라면, 옵저버빌리티는 시스템의 동적인 행위 패턴을 읽어내는 일에 가깝다. AI 시스템에서는 특히 이 능력이 중요한데, 왜 실패가 발생했는지 왜 특정 출력이 돌아왔는지를 이해해야만 같은 실패가 반복되는 것을 막을

수 있기 때문이다.

4.2 출력 품질을 수치화하는 지표

출력 품질을 관리하려면 먼저 품질을 측정할 수 있어야 한다. AI 시스템의 품질 지표는 전통적인 소프트웨어와는 성격이 다르다. 정확도나 처리 속도만으로는 AI 출력의 적합성을 판단하기 어렵다.

4.2.1 구조적 유효성 비율

전체 응답 중에서 사전에 정의한 구조에 맞던 응답의 비율이다. JSON 모드를 사용한다면 모델이 돌려준 응답 중에서 유효한 JSON이던 비율을 측정한다. 이 비율이 낮으면 모델의 출력 형식 강제가 제대로 동작하지 않는다는 신호이다.

이 비율이 급격히 떨어지면 모델의 동작이 변했거나 프롬프트가 의도치 않게 바뀐 경우다. 추이를 계속 지켜보면 시스템의 건전성을 판단하는 근거로 삼을 수 있다.

4.2.2 의미적 유효성 비율

구조적으로 유효한 응답 중에서 내용까지 적합했던 응답의 비율이다. 이것은 검증 로직으로 자동으로 측정할 수 있다. 예를 들어 특정 필드의 값이 허용 가능한 범위 안에 있는지 검증하고 그 통과 비율을 지표로 삼는 식이다.

이 비율이 낮으면 구조는 맞지만 내용이 부적합한 응답이 많다는 뜻이다. 이럴 때는 프롬프트를 바꾸거나 검증 로직을 재정의해야 한다.

4.2.3 거짓말 발생 빈도

모델이 사실이 아닌 정보를 생성한 빈도를 측정한다. 자동으로 재기는 어렵지만 사용자로부터 피드백을 수집하거나 정답이 있는 태스크에서 모델의 출력을 평가해서 근사적으로 측정할 수 있다. 완전히 자동화하기는 어려워도 정기적으로 표본을 뽑아 추적할 가치는 충분하다.

4.2.4 지연 시간 분포

요청부터 응답까지의 시간을 분포로 측정한다. 특히 신뢰성이 중요한 시스템에서는 평균보다 분산이 더 눈여겨볼 대상이다. 지연 시간이 갑자기 늘어난다면 모델 서버에 문제가 생겼거나 Rate Limit에 걸린 경우다.

4.3 모델 비결정성에 대응하는 테스트 전략

AI 모델의 비결정성은 테스트를 어렵게 만든다. 같은 입력으로 테스트해도 매번 다른 출력이 나올 수 있기 때문이다. 그래도 비결정성 안에서 테스트할 수 있는 부분과 그렇지 않은 부분은 나눠서 다뤄야 한다.

4.3.1 결정적 요소 테스트

구조화된 출력의 존재 여부, 응답 시간, 오류 유형 등은 테스트할 수 있다. 이 요소들은 모델의 확률적 출력과는 무관하게 측정된다. 회귀 테스트로 삼기에 적합하다.

4.3.2 확률적 요소 테스트

출력의 내용까지 테스트하려면 확률적 접근이 필요하다. 같은 입력을 여러 번 보내고 출력의 분포를 측정하는 방식이다. 예를 들어 100번의 요청 중 95번 이상이 특정 구조를 갖는다면 이를 품질 기준의 하나로 삼을 수 있다.

이 방법은 비용이 많이 든다. 테스트를 실행할 때마다 API 호출 비용이 발생하기 때문이다. 그래서 핵심 시나리오에만 적용하고 나머지는 결정적 요소 테스트로 처리한다.

4.3.3 합성 데이터로 검증

모델의 출력을 미리 알려진 정답과 비교할 수 있는 합성 태스크를 만든다. 정답이 명확한 질문들을 모아 테스트 세트를 구성하고 모델의 응답을 자동으로 채점한다. 이렇게 하면 사람이 일일이 판단하지 않아도 품질을 지속적으로 모니터링할 수 있다.

4.4 건전성 프로토콜과 점진적 개선 순환

위의 지표들을 지속적으로 모아 분석하면 시스템의 건전성 프로토콜을 만들 수 있다. 프로토콜은 지표의 정상 범위를 정의하고 그 범위를 벗어나면 알림을 보내거나 자동으로 대응하는 규칙을 말한다.

예를 들어 구조적 유효성 비율이 95퍼센트 아래로 떨어지면 알림을 보내고, 90퍼센트 아래로 떨어지면 자동으로 재시도 횟수를 늘리는 식으로 대응한다. 이 규칙들을 미리 정의해두면 시스템이 알아서 품질 저하를 감지하고 반응한다.

이 프로토콜의 핵심은 점진적 개선 순환이다. 지표에서 이상 패턴을 발견하면 원인을 분석해 시스템 개선에 반영한다. 개선 뒤에는 다시 지표를 모니터링해서 효과를 확인한다. 이 순환을 반복하면 시간이 지나도 시스템이 일정한 품질을 유지할 수 있다.

4.5 운영에서 지켜야 할 것

옵저버빌리티와 테스트 전략을 갖추었다 하더라도 운영에서는 몇 가지를 추가로 주의해야 한다.

모델의 버전이 바뀌면 출력 품질이 달라질 수 있다. 새로운 버전으로 업데이트할 때는 반드시 테스트를 실행하고 품질 기준을 통과한 것을 확인한 뒤에 배포한다. 모델 공급자가 버전을 올리면 변경 로그를 확인하고 영향도를 분석해야 한다.

시스템의 입력 분포가 변하면 출력 품질도 따라 변한다. 사용자들의 질문 패턴이 바뀌면 모델이 익숙하지 않은 입력을 만나는 경우가 늘어난다. 입력 분포의 변화를 주기적으로 확인하고 분포가 크게 변했을 때는 시스템의 품질 기준을 재검토해야 한다.