AI 신뢰성 공학: 프로덕션 AI 기능을 만들 때 무조건 지키는 약속들

그레이스풀 디그레이션, 서킷 브레이커, 폴백, 피처 플래그, 장애 복구 런북까지

ThakiCloud (Hyojung Han)

AI 신뢰성 공학: 프로덕션 AI 기능을 만들 때 무조건 지키는 약속들

그레이스풀 디그레이션, 서킷 브레이커, 폴백, 피처 플래그, 장애 복구 런북까지

ThakiCloud (Hyojung Han)

Contents

1 1장. 그레이스풀 디그레이션: 기능이 죽어도 시스템은 살아야 한다

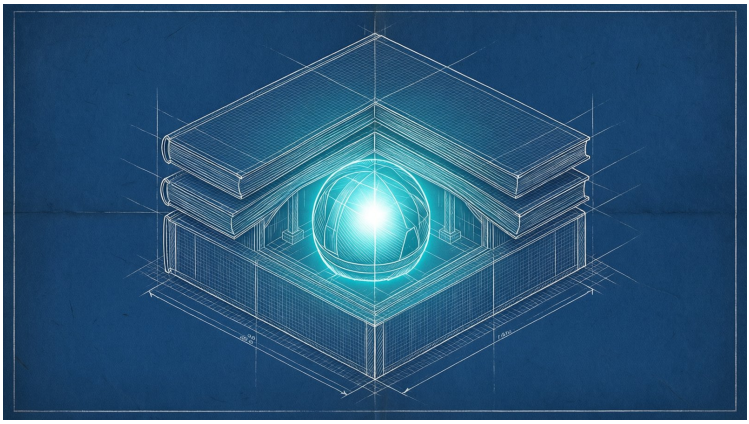

Figure 1.1: 1장. 그레이스풀 디그레이션: 기능이 죽어도 시스템은 살아야 한다

1.1 AI 기능은 왜 다르게 무너지는가

전통적인 소프트웨어는 실패의 경계가 비교적 뚜렷하다. 함수는 값을 반환하거나 예외를 던진다. 예외가 나면 그 사실을 코드가 정확히 알 수 있고, 어디서 무엇이 잘못됐는지도 스택 트레이스로 짚어낼 수 있다. AI 모델을 호출하는 코드는 이 전제가 무너진다. 모델은 예외를 던지지 않고도 틀린 답을 내놓는다. 응답 시간은 입력 길이와 모델 부하에 따라 몇 배씩 흔들리고, 같은 입력을 두 번 넣어도 다른 결과가 나올 수 있다. 실패가 이진적이지 않고 연속적이라는 사실을 AI 기능을 설계할 때 가장 먼저 인정해야 한다.

이 연속성 때문에 AI 기능에는 성공과 실패 사이에 넓은 회색지대가 생긴다.

모델이 응답은 했지만 형식이 깨졌을 수도 있고, 사실관계가 틀렸을 수도 있고, 질문과 무관한 내용을 답했을 수도 있다. 이런 회색지대를 처리하지 못하는 시스템은 실패를 숨긴 채 사용자에게 그럴듯하지만 틀린 결과를 그대로 보여준다. 그래서 AI 기능을 만드는 엔지니어의 첫 번째 책임은 정답을 더 잘 맞히는 것이 아니라, 답이 틀렸거나 늦거나 없을 때 시스템이 어떻게 동작할지를 미리 설계하는 것이다. 이 책 전체를 관통하는 질문은 하나다. 이 기능이 실패하면 사용자는 무엇을 보게 되는가.

1.2 그레이스풀 디그레이션의 세 가지 레벨

그레이스풀 디그레이션은 기능 전체를 끄는 대신 품질을 단계적으로 낮추면서 서비스를 계속 제공하는 설계 원칙이다. AI 기능에서는 이 원칙을 세 가지 레벨로 나누어 적용하면 관리하기 쉬워진다.

표시 레벨은 화면에 무엇을 보여줄지에 관한 것으로, 가장 구현하기 쉽고 가장 먼저 손대야 할 영역이다. AI가 생성한 요약이 실패하면 요약란을 비워두는 대신 원문의 첫 몇 문장을 발췌해서 보여주거나, 최소한 요약 기능이 잠시 쉬고 있다는 상태를 명확히 알려준다.

동작 레벨은 사용자가 할 수 있는 행동의 범위에 관한 것이다. AI 추천 기능이 실패하면 추천 버튼을 숨기는 대신 최근에 본 항목이나 인기 항목처럼 규칙으로 계산 가능한 대안 목록을 보여준다. 사용자가 아무것도 하지 못하는 상태를 만들지 않는 것이 핵심이다.

데이터 레벨은 시스템 내부의 상태와 저장되는 값에 관한 것이다. AI가 생성한 값을 다른 시스템이 참조하고 있다면, 실패했을 때 그 값을 비워두지 않고 마지막으로 성공했던 값이나 명시적인 기본값으로 채워야 한다. 이 레벨을 놓치면 화면은 정상으로 보여도 뒤에서 데이터 정합성이 깨지는, 훨씬 늦게 발견되는 장애로 이어진다.

세 레벨을 동시에 다 설계할 필요는 없다. 다만 기능을 새로 만들 때마다 이

세 가지 레벨마다 실패 시 무엇을 할지 한 문장씩이라도 정해두면, 나중에 실제 장애가 났을 때 즉흥적으로 대응하는 것보다 훨씬 낫다.

1.3 실패를 감지하고 전파를 막는 구조

AI 호출의 실패를 상위 코드까지 그대로 흘려보내면, 그 실패는 호출자를 거치며 점점 커진다. 이를 막으려면 모델 호출을 감싸는 전용 계층을 두고, 모든 호출이 반드시 이 계층을 거치도록 강제해야 한다. 이 계층의 역할은 두 가지다. 하나는 실패를 정확히 분류하는 것이고, 다른 하나는 분류된 실패를 호출자가 다루기 쉬운 형태로 바꿔서 돌려주는 것이다.

실패는 최소한 다음 세 종류로 나누어 다뤄야 한다.

- 타임아웃: 모델이 정해진 시간 안에 응답하지 않는 경우
- 명시적 오류: 모델 서버가 오류 상태나 오류 메시지를 돌려주는 경우
- 형식 오류: 응답은 왔지만 기대한 구조와 다른 경우, 예를 들어 구조화된 데이터를 기대했는데 자유 텍스트가 오거나 필수 필드가 비어 있는 경우

이 세 가지는 서로 다른 복구 전략이 필요하기 때문에 하나로 뭉뚱그려 처리하면 안 된다. 타임아웃은 재시도할 가치가 있지만, 형식 오류는 같은 요청을 반복해도 같은 형식 오류가 날 가능성이 높다.

호출 계층은 실패를 감지하면 예외를 위로 던지는 대신 의미 있는 결과 객체로 바꿔서 반환하는 편이 다루기 쉽다. 성공 여부, 실패 종류, 대체 가능한 기본값을 함께 담은 구조를 쓰면 호출하는 쪽 코드는 매번 예외 처리 블록을 새로 쓰지 않고도 실패를 자연스럽게 다룰 수 있다.

1.4 폴백 화면에서 사용자에게 의미 있는 피드백 주기

모델 호출이 실패했을 때 사용자에게 보여줄 문구는 오류가 발생했습니다처럼 시스템 관점의 표현이면 안 된다. 사용자가 그 기능으로 얻으려 했던 목적을 기준으로 문구를 설계해야 한다. 예를 들어 문서 요약 기능이 실패했다면, 오류가 발생했습니다보다는 지금은 요약을 만들 수 없으니 원문을 직접 확인해달라는 문구가 훨씬 유용하다. 사용자가 다음에 무엇을 하면 되는지까지 알려주는 문구가 좋은 문구다.

동시에 지금 보고 있는 결과가 평소와 다른 경로로 만들어졌다는 사실을 감추지 않는 편이 낫다. 규칙 기반으로 계산된 대체 결과를 마치 AI가 만든 정상 결과처럼 보여주면, 나중에 그 결과가 기대와 다를 때 사용자는 원인을 알 수 없어 더 크게 신뢰를 잃는다. 반대로 지금은 간소화된 방식으로 안내하고 있다는 사실을 작은 배지나 안내 문구로 알려주면, 품질이 조금 낮아도 사용자는 상황을 이해하고 받아들인다. 상태를 정확히 알려주는 것 자체가 신뢰를 지키는 방법이다.

1.5 감지 기준과 복구 시점은 분리해야 한다

실패를 감지하는 로직과 정상 상태로 되돌아갈 시점을 판단하는 로직은 서로 다른 질문에 답한다. 감지 로직은 지금 이 호출이 실패했는가를 판단하고, 복구 로직은 지금 정상 상태로 되돌려도 안전한가를 판단한다. 이 둘을 하나의 로직으로 합쳐 놓으면, 한 번의 성공만으로 바로 정상 상태로 돌아가려다가 다시 실패하는 일이 반복된다.

복구 판단에는 시간 조건과 부하 조건을 함께 쓰는 것이 안정적이다. 시간 조건은 최근 실패가 얼마나 지속되었는지를 보는 것이고, 부하 조건은 지금 시스템과 모델 서버가 여유가 있는지를 보는 것이다. 예를 들어 최근 오 분 동안 연속으로 성공했고 동시에 모델 서버의 응답 지연이 평상시 수준으로 돌아왔다면 그때 복구를 시도하는 식이다. 순간적인 성공 한두 번만으로

복구를 단정하면, 일시적인 흔들림을 완전한 회복으로 오인하게 된다.

1.6 부분 실패를 설계에 반영하기

AI 기능의 실패는 전부 아니면 전무인 경우보다 부분적인 경우가 훨씬 많다. 여러 항목의 순위를 매기는 기능에서 전체 순위 계산은 실패해도 개별 항목의 점수 계산은 성공할 수 있다. 이때 완전한 순위 목록 대신 점수만 나열해서 보여주는 것이 부분 실패를 살리는 방식이다. 텍스트와 이미지를 함께 생성하는 기능이라면 텍스트만 성공하고 이미지가 실패하는 경우가 있다. 이때는 텍스트 결과를 그대로 보여주고 이미지 자리에는 나중에 다시 시도할 수 있다는 안내를 담은 빈 자리를 남겨두는 편이 전체를 실패로 처리하는 것보다 낫다.

부분 실패를 잘 다루려면 기능을 설계하는 단계에서부터 결과를 여러 개의 독립적인 조각으로 나눌 수 있는지 먼저 검토해야 한다. 하나의 큰 호출로 모든 것을 한 번에 받아오는 구조는 구현은 간단하지만, 그중 하나만 실패해도 전체가 무너진다. 반대로 조각을 나누어 병렬로 호출하면 일부가 실패해도 나머지는 살릴 수 있다.

1.7 점진적 복구로 재발을 막는다

실패에서 정상으로 돌아갈 때 모든 트래픽을 한 번에 되돌리면 아직 회복이 덜 된 모델 서버에 다시 부하가 몰려 재발할 위험이 크다. 대신 트래픽 비율을 단계적으로 늘려가는 방식이 안전하다. 처음에는 전체 요청의 일부만 정상 경로로 보내고, 그 구간에서 오류율이 기준 아래로 유지되면 다음 구간으로 비율을 늘린다. 어느 구간에서든 오류율이 다시 기준을 넘으면 그 즉시 이전 비율로 되돌아간다.

이 방식은 완전히 회복되지 않은 상태에서 전체 트래픽을 한꺼번에 받아 다시 무너지는 것을 막아준다. 동시에 모델 서버가 늘어나는 부하에 적응할

여유 시간을 벌어준다.

여기까지 정리한 세 가지 레벨, 실패 감지 계층, 폴백 문구, 감지와 복구의 분리, 부분 실패, 점진적 복구는 서로 독립된 다섯 가지 기법이 아니라 하나의 흐름을 이루는 조각들이다. 실패를 감지하는 순간부터 다시 정상으로 돌아오는 순간까지, 사용자가 그 전 과정에서 계속 무언가를 할 수 있도록 만드는 것이 그레이스풀 디그레이션의 목표다. 다음 장에서 다룰 서킷 브레이커는 바로 이 감지, 차단, 점진적 복구의 흐름을 하나의 재사용 가능한 컴포넌트로 구현하는 방법을 다룬다.

2 2장. AI 서킷 브레이커: 무한 대기열을 막는 보호기

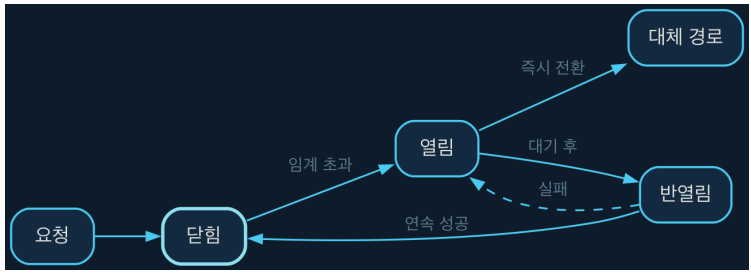

Figure 2.1: 2장. AI 서킷 브레이커: 무한 대기열을 막는 보호기

2.1 재시도만으로는 부족한 이유

모델 호출이 실패했을 때 가장 먼저 떠오르는 대응은 재시도다. 네트워크가 순간적으로 끊기거나 서버가 잠깐 바빴을 뿐이라면 재시도만으로 충분하다. 문제는 모델 서버 자체가 과부하 상태이거나 완전히 죽어 있을 때 생긴다. 이런 상황에서 재시도는 죽어가는 서버에 요청을 더 얹는 꼴이 되어 회복을 오히려 늦춘다. 게다가 요청을 보낸 쪽도 응답을 기다리는 동안 자원을 붙잡고 있게 되므로, 모델 서버 하나의 장애가 그 서버를 호출하는 모든 서비스로 번져나간다. 서킷 브레이커는 이 확산을 끊기 위한 장치다. 실패가 반복되면 아예 호출을 시도조차 하지 않고 즉시 대체 경로로 넘어가도록 만들어서, 죽어가는 서버에 더는 부하를 주지 않으면서 동시에 호출하는 쪽도 무한정 기다리지 않게 한다.

2.2 세 가지 상태: 닫힘, 열림, 반열림

서킷 브레이커는 세 가지 상태를 오가며 동작한다.

- 닫힘: 평소 상태로, 모든 호출이 정상적으로 모델 서버에 전달된다. 실패율을 계속 관찰한다.
- 열림: 실패율이 기준을 넘어서면 이 상태로 전환되며, 이후 들어오는 호출은 모델 서버로 보내지 않고 즉시 대체 경로로 처리한다.
- 반열림: 열림 상태로 일정 시간이 지나면 이 상태로 전환되며, 소수의 요청만 시험 삼아 모델 서버로 보내 회복 여부를 확인한다.

반열림 상태에서 보낸 시험 요청이 성공하면 닫힘 상태로 돌아가고, 실패하면 다시 열림 상태로 되돌아간다. 이 순환 구조 덕분에 서킷 브레이커는 사람이 개입하지 않아도 장애가 발생했을 때 자동으로 격리하고, 회복되면 정상 경로를 스스로 되찾는다.

2.3 응답 시간과 오류율, 두 기준을 함께 본다

서킷을 열지 말지를 판단할 때 오류율 하나만 보면 놓치는 장애 유형이 있다. 모델이 오류를 던지지는 않지만 응답이 극단적으로 느려지는 경우다. 요청을 오류 없이 처리하되 평소보다 열 배 느리게 처리하는 상황은 오류율 지표만으로는 잡히지 않는다. 그래서 오류율과 함께 응답 시간 지표, 특히 상위 구간의 응답 시간을 같이 감시할 필요가 있다.

실무에서는 두 조건에 각각 임계값을 두고 둘 중 하나라도 넘으면 서킷을 여는 방식을 많이 쓴다. 예를 들어 최근 일 분 동안의 오류율이 오십 퍼센트를 넘거나, 상위 오 퍼센트 구간의 응답 시간이 평소 기준의 세 배를 넘으면 서킷을 여는 식이다. 정확한 수치는 서비스마다 다르므로 초기값은 어림잡아 정하고 이후 실제 트래픽 패턴을 관찰하며 조정해나간다. 이 책에서 제시하는 수치는 모두 출발점을 위한 추정치일 뿐, 각자의 트래픽과 모델 특성에 맞춰 재조정하는 과정이 반드시 필요하다.

2.4 무한 재시도가 장애를 키우는 함정

서킷 브레이커 없이 재시도 로직만 있는 시스템에서는 흔히 이런 일이 벌어진다. 모델 서버가 느려지면 요청이 쌓이고, 쌓인 요청 때문에 서버는 더 느려지고, 느려진 서버에 재시도 요청이 계속 더해지면서 상황이 악순환에 빠진다. 이 악순환은 호출하는 쪽 서비스에도 그대로 옮겨붙는다. 응답을 기다리는 스레드나 연결이 쌓이면서 호출하는 쪽 서비스마저 자원이 고갈되어 함께 멈춰버리는 것이다.

이를 막으려면 재시도 횟수에 상한을 두는 것만으로는 부족하다. 재시도 사이의 간격을 점점 늘리는 방식과, 서킷 브레이커로 아예 재시도 자체를 중단시키는 방식을 함께 써야 한다. 서킷이 열린 상태에서는 재시도 로직이 아무리 잘 짜여 있어도 호출을 시도하지 않는 편이 낫다. 이미 죽어 있는 서버에 요청을 보내는 행위 자체가 낭비이자 위험이다.

2.5 반열림 상태에서 트래픽을 조심스럽게 흘려보낸다

열림 상태에서 곧바로 닫힘 상태로 돌아가면 아직 회복이 덜 된 서버에 트래픽이 한꺼번에 몰려 다시 무너질 수 있다. 그래서 반열림 상태에서는 전체 요청 중 일부만, 예를 들어 백 개 중 한두 개 정도만 골라 시험 요청으로 모델 서버에 보낸다. 나머지 요청은 여전히 대체 경로로 처리한다. 시험 요청이 연속으로 몇 번 성공하면 그때 닫힘 상태로 완전히 전환한다.

이 표본 추출 방식은 1장에서 설명한 점진적 복구 원칙을 서킷 브레이커 안에 구현한 것이나 다름없다. 회복 여부를 적은 위험으로 확인하고, 확인이 끝난 뒤에야 전체 트래픽을 되돌린다. 표본의 크기와 성공 판정 기준은 서비스의 위험 허용 수준에 따라 다르게 잡으면 된다.

2.6 서킷 브레이커의 범위를 어디까지 둘 것인가

서킷 브레이커를 시스템 전체에 하나만 두면 관리는 편해도 문제를 정확히 격리하지 못한다. 여러 모델을 함께 쓰는 시스템이라면 모델별로 별도의 서킷을 두는 편이 낫다. 그래야 한 모델의 장애가 다른 모델을 쓰는 기능까지 함께 멈추게 만들지 않는다. 여러 테넌트나 고객사를 서비스하는 경우라면 테넌트별로 서킷을 나누는 것도 고려할 만하다. 특정 고객사의 비정상적인 트래픽 패턴이 다른 고객사의 서비스 품질까지 떨어뜨리는 것을 막을 수 있어서다.

범위를 세분화할수록 격리 효과는 좋아지지만 관리해야 할 서킷의 개수도 늘어난다. 처음에는 모델 단위로 시작해서, 실제 장애 사례를 겪으며 더 세분화할 필요가 있는 지점을 찾아 나가는 편이 현실적이다.

2.7 작은 예시로 임계값 정하는 과정 살펴보기

고객 문의에 자동으로 답하는 기능을 예로 들어보자. 이 기능은 평상시 응답 시간이 대체로 이 초 안팎이고, 하루 평균 오류율은 일 퍼센트 미만이라고 가정한다. 이 상태를 기준선으로 삼아 서킷을 열 조건을 오류율 급등과 응답 시간 급등 두 가지로 나눠 정의한다. 오류율 조건은 최근 일 분 동안의 실패 비율이 평소 기준의 다섯 배를 넘는 경우로 잡고, 응답 시간 조건은 상위 오 퍼센트 구간의 응답 시간이 평소의 세 배인 육 초를 넘는 경우로 잡는다. 둘 중 하나만 충족돼도 서킷이 열린다.

열림 상태로 전환된 뒤에는 삼십 초 동안 모든 요청을 대체 경로로 돌리고, 그 뒤 반열림 상태로 넘어가 전체 요청의 오 퍼센트만 시험 삼아 실제 모델로 보낸다. 이 표본에서 연속으로 열 번 성공하면 닫힘 상태로 돌아가고, 실패가 섞이면 다시 삼십 초 동안 열림 상태를 유지한 뒤 재시도한다. 이렇게 숫자를 구체적으로 정해두면 실제 장애 상황에서 사람이 즉흥적으로 판단할 필요 없이 시스템이 먼저 움직인다. 물론 이

수치들은 어디까지나 출발점을 위한 추정값일 뿐, 실제 서비스의 트래픽 패턴과 사용자의 민감도를 몇 주간 관찰하며 계속 다듬어가야 한다.

다음 장에서는 서킷이 열렸을 때 대체 경로로 무엇을 보여줄지, 그 폴백 자체를 어떻게 설계할지 살펴본다.

3 3장. AI 폴백 설계: 무엇을 대신 보여줄 것인가

Figure 3.1: 3장. AI 폴백 설계: 무엇을 대신 보여줄 것인가

3.1 기능 등급 매기기: 코어, 중요, 보조

모든 AI 기능에 똑같은 수준의 폴백을 준비할 필요는 없다. 자원과 시간은 한정돼 있으니, 기능마다 실패했을 때 사용자가 받는 영향의 크기부터 따져 등급을 나누는 편이 낫다.

- 코어 기능: 없으면 서비스의 핵심 가치 자체가 성립하지 않는 기능. 예를 들어 AI 검색 서비스에서 검색 결과를 만드는 기능이 여기 해당한다. 이 등급은 정교한 폴백을 반드시 갖춰야 한다.
- 중요 기능: 서비스 이용에 상당한 불편을 주지만 다른 방법으로 우회할 수 있는 기능. 예를 들어 AI가 자동으로 채워주는 입력값 추천 기능은 없어도 사용자가 직접 입력할 수 있다.
- 보조 기능: 있으면 좋지만 없어도 핵심 흐름에 영향을 주지 않는 기능. 예를 들어 AI가 생성하는 부가 설명이나 관련 콘텐츠 추천이 여기 해당한다.

등급을 나누는 작업은 개발팀 판단만으로 끝내지 말고 실제 사용자 흐름을 아는 사람과 함께 검토하는 편이 좋다. 개발자 입장에서 사소해 보이는 기능이 실제로는 사용자의 핵심 작업 흐름에 깊이 얽혀 있는 경우가 드물지 않다.

3.2 폴백 사다리: 단계적으로 낮아지는 대안

폴백을 하나의 대체 수단으로만 준비하면 그 대체 수단마저 실패했을 때 다시 막다른 골목에 몰린다. 그래서 코어 기능일수록 여러 단계로 이어지는 폴백 사다리를 준비하는 것이 안전하다. 일반적으로 다음과 같은 순서로 사다리를 구성할 수 있다.

- 일차 폴백: 같은 모델에 더 짧은 시간제한을 두고 한 번 더 시도한다. 순간적인 지연이었다면 이 단계에서 해결된다.

- 이차 폴백: 더 가볍고 빠른 대체 모델로 전환한다. 품질은 낮아지지만 여전히 AI가 생성한 결과를 제공할 수 있다.
- 삼차 폴백: 규칙 기반 로직으로 전환한다. 미리 정의한 조건문이나 통계로 계산 가능한 결과를 대신 내놓는다.
- 사차 폴백: 가장 최근에 성공했던 결과를 캐시에서 꺼내 보여준다. 실시간성은 떨어지지만 완전히 비어 있는 것보다는 낫다.
- 최종 폴백: 정적인 안내 문구나 대체 화면을 보여준다. 모든 자동화된 대안이 실패했을 때의 마지막 방어선이다.

사다리의 각 단계는 이전 단계보다 반드시 더 단순하고 더 안정적이어야 한다. 규칙 기반 로직이 대체 모델보다 복잡하다면 순서가 잘못된 것이다. 사다리를 설계할 때는 아래로 내려갈수록 실패할 가능성이 낮아지는지를 확인해야 한다.

3.3 규칙 기반 폴백과 대체 모델 폴백의 조합

규칙 기반 폴백은 예측 가능하고 실패할 여지가 거의 없다는 장점이 있지만, AI가 만들어내는 맥락 인식 능력을 흉내 내지는 못한다. 반대로 대체 모델 폴백은 어느 정도 맥락을 살릴 수 있지만 그 자체가 또 다른 AI 호출이므로 완전히 실패하지 않는다는 보장이 없다. 그래서 둘을 조합하는 방식이 실무에서 가장 안정적이다.

예를 들어 상품 설명을 자동으로 생성하는 기능이라면, 주력 모델이 실패했을 때 더 작고 빠른 모델로 한 번 더 시도하고, 그마저 실패하면 상품의 속성값을 조합해 만든 정형화된 문장으로 대체하는 식이다. 규칙 기반 폴백을 만들 때는 처음부터 완벽한 문장을 목표로 하지 않는 편이 낫다. 정보가 정확하게 전달되는 것이 매끄러운 문장보다 우선이다.

3.4 폴백 전환을 기록하고 알리기

어떤 요청이 어느 단계의 폴백까지 내려갔는지 기록으로 남기지 않으면, 사다리가 자주 삼차나 사차 단계까지 내려가는데도 아무도 그 사실을 모르는 상황이 벌어진다. 폴백이 전환될 때마다 시점과 원인, 어느 단계로 내려갔는지를 로그로 남겨야 한다. 이 기록이 쌓이면 어떤 기능이 얼마나 자주 폴백에 의존하는지 정량적으로 파악할 수 있고, 이는 근본 원인을 고칠 우선순위를 정하는 데 직접적인 근거가 된다.

내부 기록과 별개로, 사용자에게도 폴백 상태를 적절히 알려야 한다. 다만 모든 사소한 폴백까지 사용자에게 알림을 띄우면 오히려 사용 경험을 해친다. 일차나 이차 폴백처럼 사용자가 체감하기 어려운 수준이라면 조용히 처리하고, 삼차 폴백 이하로 내려가 품질 차이가 눈에 띄는 수준이라면 화면에 상태를 표시하는 기준을 세워두는 것이 좋다.

3.5 폴백 품질을 계속 측정해야 하는 이유

폴백은 한 번 만들어 놓으면 끝나는 것이 아니다. 시간이 지나면서 원래 기능의 데이터나 모델이 바뀌는데 폴백 로직은 그대로 남아 있으면, 폴백이 오히려 최신 상태를 반영하지 못하는 낡은 대안이 되어버린다. 규칙 기반 폴백에 쓰이는 기준값이나 조건문은 주기적으로 다시 검토해서 실제 데이터 분포와 맞는지 확인해야 한다.

또한 폴백이 얼마나 자주 호출되는지, 폴백으로 전환된 뒤 사용자가 그 결과를 실제로 어떻게 이용하는지도 지표로 관리할 필요가 있다. 폴백 호출 빈도가 예상보다 높다면 이는 폴백 설계의 문제가 아니라 주력 기능 자체의 신뢰성 문제일 가능성이 크다. 폴백은 신뢰성을 보완하는 안전망이지, 주력 기능의 불안정을 영구히 감춰주는 도구가 되어서는 안 된다.

3.6 폴백 사다리를 실제 기능에 적용해보기

앞서 등급을 매길 때 코어 기능으로 분류했던 검색 결과 생성 기능을 예로 들어 사다리를 완성해보자. 일차 폴백에서는 시간제한을 이 초에서 오백 밀리초로 줄여 같은 모델에 한 번 더 요청한다. 순간적인 지연이 원인이었다면 대부분 이 단계에서 해결된다. 이차 폴백에서는 응답 속도가 빠른 경량 모델로 전환해 검색어와 가장 관련성이 높은 항목만 추려서 보여준다. 삼차 폴백에서는 검색어에 포함된 핵심 단어를 기존 색인에서 직접 찾는 규칙 기반 검색으로 전환한다. AI가 만든 순위는 아니지만 관련성 있는 결과를 여전히 보여줄 수 있다. 사차 폴백에서는 같은 검색어로 최근에 성공했던 결과가 캐시에 남아 있다면 그 결과를 재사용한다. 모든 단계가 실패하면 최종적으로 인기 검색어나 추천 카테고리를 보여주는 정적 화면으로 넘어간다.

이 사다리를 실제로 구현할 때는 각 단계로 내려갈 때마다 걸리는 시간도 함께 관리해야 한다. 사다리를 다섯 단계까지 다 거치는 데 십 초가 걸린다면, 그 사이 사용자는 이미 화면을 벗어났을 것이다. 단계별로 짧은 시간제한을 두고 전체 사다리가 완료되기까지의 총 시간에도 상한을 정해야 폴백 설계가 실효성을 가진다. 다음 장에서는 이런 폴백 전환과 모델 교체를 배포 없이 즉시 실행할 수 있게 해주는 피처 플래그를 다룬다.

4 4장. AI 피처 플래그: 배포 없이 기능을 통제하는 법

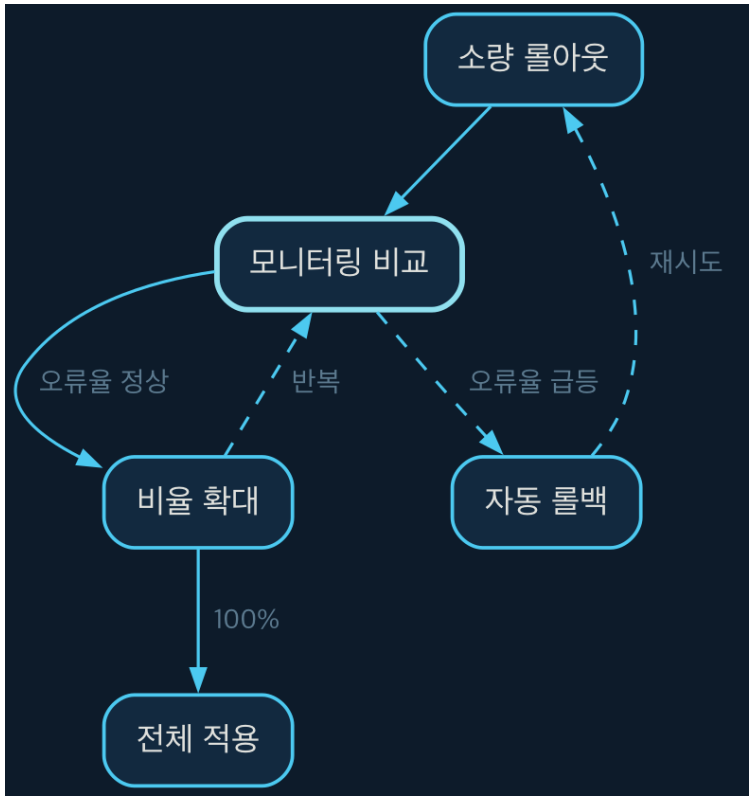

Figure 4.1: 4장. AI 피처 플래그: 배포 없이 기능을 통제하는 법

4.1 왜 AI 기능은 배포 없이 바꿀 수 있어야 하는가

일반적인 소프트웨어 기능은 문제가 생기면 코드를 고치고 다시 배포해서 해결한다. AI 기능은 이 흐름이 너무 느리다. 모델 서버에 이상이 생겼거나, 특정 프롬프트가 유해한 답을 유도하고 있거나, 특정 모델 버전이 예상보다

품질이 낮다는 사실이 밝혀졌을 때, 코드를 고치고 빌드하고 배포하는 절차를 다 거치는 동안 사용자는 계속 나쁜 경험을 겪는다. 피처 플래그는 이 절차를 건너뛰고 운영 중인 시스템의 동작을 즉시 바꿀 수 있게 해주는 장치다. 모델 버전, 프롬프트 내용, 호출 여부 자체를 코드 밖에 있는 설정값으로 빼두면, 그 설정값만 바꿔서 즉시 반영할 수 있다.

4.2 트래픽 비율 분할과 단계적 롤아웃

새 모델이나 새 프롬프트를 도입할 때 전체 트래픽에 한 번에 적용하는 것은 위험하다. 대신 전체 트래픽 중 일부에만 새 버전을 적용하고 나머지는 기존 버전을 유지하는 방식으로 시작해야 한다. 처음에는 극소수의 비율로 시작해서, 문제가 없다는 것이 확인되면 점차 비율을 늘려가는 방식이 표준적인 접근이다.

이 방식의 핵심은 새 버전을 적용받는 대상을 무작위로 고르되, 같은 사용자는 관찰 기간 동안 계속 같은 버전을 받도록 고정하는 데 있다. 사용자가 요청할 때마다 다른 버전을 무작위로 받으면 결과의 일관성이 깨지고, 새 버전의 실제 효과를 측정하기도 어려워진다. 사용자 식별자를 기준으로 안정적으로 그룹을 나누는 방식을 쓰면 이 문제를 피할 수 있다.

4.3 모니터링과 연동한 자동 롤백

피처 플래그로 새 버전을 서서히 늘려가는 과정에서 사람이 매 순간 지표를 지켜보고 있을 수는 없다. 그래서 미리 정한 기준을 벗어나면 자동으로 이전 버전으로 되돌리는 장치를 함께 만들어야 한다. 예를 들어 새 모델을 적용받은 그룹의 오류율이 기존 모델을 적용받은 그룹보다 뚜렷하게 높아지면, 그 즉시 새 모델 적용 비율을 0으로 되돌리고 담당자에게 알림을 보낸다.

자동 롤백을 설계할 때 중요한 점은 비교 대상을 항상 함께 두는 데 있다.

새 버전의 오류율만 절대적인 기준으로 판단하면, 전체 시스템에 원래부터 있던 잡음과 새 버전이 만든 문제를 구분하지 못한다. 기존 버전을 받는 그룹과 새 버전을 받는 그룹을 동시에 관찰하고 그 차이를 기준으로 판단해야 오탐을 줄일 수 있다.

4.4 AI 기능에 특화된 플래그 유형

일반적인 피처 플래그는 켜고 끄는 이진 값이나 단순한 비율 값을 다루지만, AI 기능에는 몇 가지 특화된 플래그 유형이 추가로 필요하다.

- 출력 필터 플래그: 모델 출력에 적용할 검열이나 후처리 규칙의 강도를 조절한다. 특정 유형의 문제가 발견되면 필터 강도를 즉시 높일 수 있다.
- 속도 제한 플래그: 사용자나 테넌트별로 모델 호출 빈도의 상한을 조절한다. 특정 사용자군의 과도한 호출이 전체 서비스 품질을 떨어뜨릴 때 즉시 제한을 걸 수 있다.
- 토큰 예산 플래그: 요청 하나가 소비할 수 있는 최대 토큰 수를 조절한다. 비용이 예상보다 급증할 때 즉시 상한을 낮춰 대응할 수 있다.
- 모델 버전 고정 플래그: 특정 사용자군이나 특정 기능에 대해 모델 버전을 명시적으로 고정한다. 최신 모델이 예상과 다르게 동작할 때 이전 버전으로 즉시 되돌리는 용도로 쓴다.

이 네 가지 유형은 배포 파이프라인을 거치지 않고도 운영 중에 즉시 조절할 수 있어야 진짜 효과가 있다. 코드를 고쳐야만 값을 바꿀 수 있다면 피처 플래그의 존재 이유가 사라진다.

4.5 플래그가 늘어날 때 관리하는 법

피처 플래그는 문제를 빠르게 해결해주지만, 시간이 지나며 개수가 늘어나면 그 자체가 관리 부담이 된다. 어떤 플래그가 지금 어떤 값으로 설정되어 있는지, 왜 그 값으로 정했는지를 한눈에 볼 수 있는 곳이 없으면, 오래된 임시 설정이 원인 모를 문제를 일으키는 상황이 생긴다. 모든 플래그에는 만든 목적과 원래 예정된 정리 시점을 함께 기록해야 한다. 특히 장애 대응 중 급하게 만든 플래그는 장애가 끝난 뒤 정리하지 않으면 다음 사람이 그 존재조차 모른 채 시스템을 운영하게 된다.

플래그가 늘어날수록 조합의 수도 늘어난다는 점도 유의해야 한다. 서로 다른 플래그가 동시에 특정 값으로 설정되었을 때 예상치 못한 상호작용이 생길 수 있다. 플래그를 새로 만들 때는 기존 플래그와 함께 켜졌을 때 무슨 일이 생기는지도 함께 점검하는 습관이 필요하다.

4.6 플래그를 실제 장애 대응에 써보는 순간

새로 배포한 프롬프트가 특정 조건에서 부적절한 문장을 만들어낸다는 신고가 들어왔다고 가정해보자. 배포 없이 대응하는 순서는 이렇다. 먼저 모델 버전 고정 플래그를 이전 프롬프트 버전으로 되돌려 문제가 되는 프롬프트를 즉시 전체 트래픽에서 배제한다. 동시에 출력 필터 플래그의 검사 강도를 한 단계 높여, 혹시 남아 있을지 모르는 유사한 패턴을 한 번 더 걸러낸다. 문제의 범위가 특정 사용자군에 몰려 있다면 그 사용자군에만 속도 제한 플래그를 걸어 추가 피해를 줄인 다음, 원인이 된 프롬프트를 코드 저장소에서 수정하고 검증을 마친 뒤에야 트래픽 비율을 다시 서서히 늘려간다.

이 대응 전체가 코드 배포 없이 설정값 몇 개를 바꾸는 것만으로 몇 분 안에 끝난다는 점이 피처 플래그의 진짜 가치다. 같은 상황을 코드 수정과 재배포로 대응했다면 빌드와 검증 절차를 거치는 동안 문제가 된

프롬프트가 계속 사용자에게 노출되었을 것이다. 다음 장에서는 이런 통제 장치들을 실제 장애 상황에서 어떤 순서로 사용할지, 장애 복구 런북이라는 형태로 정리한다.

5 5장. AI 장애 복구 런북: 그 순간에 무엇을 할 것인가

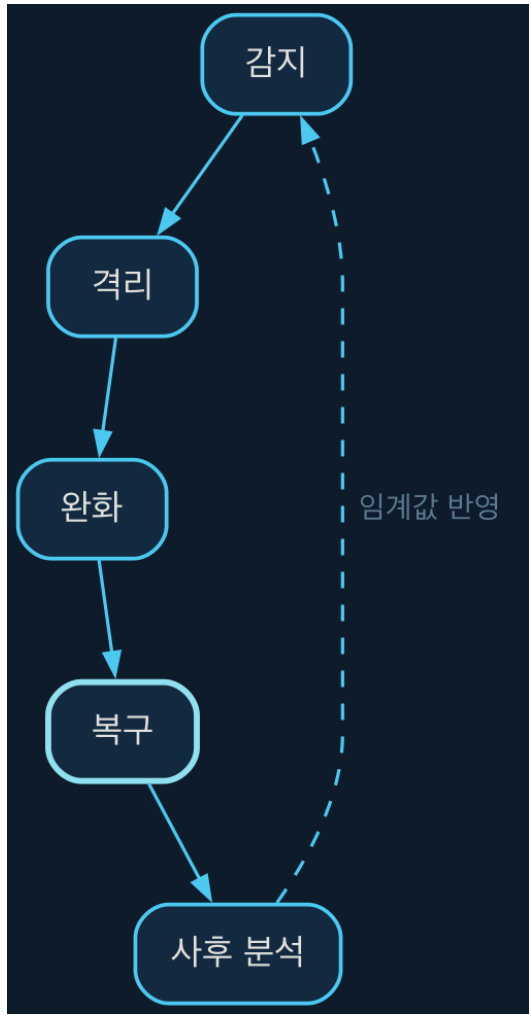

Figure 5.1: 5장. AI 장애 복구 런북: 그 순간에 무엇을 할 것인가

5.1 장애 등급을 미리 나눠두어야 하는 이유

장애가 실제로 터진 순간에 등급을 처음 정하려고 하면 판단이 늦어지고 대응도 늦어진다. 등급은 장애가 나기 전에 미리 정의해 두어야 한다. 일반적으로 다음과 같이 세 단계로 나누면 실무에서 다루기 쉽다.

- 최고 등급: 코어 기능이 완전히 멈추거나 잘못된 결과가 대량으로 사용자에게 노출되는 경우. 즉시 대응 인력을 소집하고 모든 관련자에게 알린다.
- 중간 등급: 중요 기능이 저하되었지만 폴백으로 서비스는 유지되는 경우. 정해진 시간 안에 담당자가 확인하고 대응한다.
- 낮은 등급: 보조 기능에 문제가 있거나 폴백 호출 빈도가 평소보다 높아진 경우. 정기 점검 시점에 함께 검토한다.

등급을 나누는 기준에는 영향을 받는 사용자 규모와 노출되는 결과의 위험성, 지속 시간이 함께 들어가야 한다. 단순히 오류율 수치 하나만으로 등급을 매기면 실제 피해 규모와 어긋나는 경우가 생긴다.

5.2 다섯 단계로 이어지는 대응 흐름

장애 대응은 감지, 격리, 완화, 복구, 사후 분석의 다섯 단계로 이어지는 흐름으로 정리하면 놓치는 부분이 줄어든다.

- 감지: 지표나 알림을 통해 문제를 처음 인지하는 단계다. 앞서 다룬 서킷 브레이커와 자동 롤백 장치가 이 단계의 상당 부분을 자동화해준다.
- 격리: 문제가 다른 정상적인 기능으로 번지지 않도록 경계를 긋는 단계다. 서킷을 강제로 열거나 특정 피처 플래그를 비활성화하는 조치가 여기 해당한다.
- 완화: 사용자가 느끼는 피해를 줄이는 단계다. 폴백 사다리를 강제로 하위 단계로 내리거나, 문제가 된 모델 버전을 이전 버전으로

되돌리는 조치를 취한다.

- 복구: 원래 상태로 안전하게 되돌리는 단계다. 1장에서 다룬 점진적 복구 원칙에 따라 트래픽을 단계적으로 되돌린다.
- 사후 분석: 무엇이 왜 일어났고 재발을 막으려면 무엇을 바꿔야 하는지 정리하는 단계다.

이 다섯 단계를 담당자별로 미리 배정해 두면, 실제 장애 상황에서 누가 무엇을 할지를 그 순간에 새로 정하지 않아도 된다.

5.3 포착하기 어려운 출력을 자동으로 걸러내기

AI 장애 중 가장 다루기 까다로운 유형은 시스템이 형식적으로는 정상 응답을 냈지만 내용이 사실과 다르거나 부적절한 경우다. 이런 출력은 오류 코드로 잡히지 않기 때문에 별도의 검사 절차가 필요하다. 실무에서 쓸 수 있는 방법은 세 가지다. 출력에 포함된 특정 패턴이나 금지어를 검사하는 규칙 기반 검사, 결과의 길이나 구조가 정상 범위를 벗어났는지 확인하는 통계적 검사, 별도의 가벼운 모델로 원래 출력을 다시 검토하는 이중 검토 방식이다.

이런 검사에서 이상이 발견되면 그 결과를 사용자에게 그대로 보여주지 않고 즉시 폴백 경로로 돌리는 자동 차단 절차를 만들어야 한다. 사람이 매 건을 확인할 수 없는 규모의 트래픽에서는 이런 자동 차단이 장애 확산을 막는 마지막 방어선 역할을 한다. 다만 자동 차단 기준이 지나치게 엄격하면 정상적인 결과까지 걸러내므로, 오탐률을 주기적으로 점검하고 기준을 조정하는 작업도 뒤따라야 한다.

5.4 모델 재시작과 데이터 무결성 확인

모델 서버 자체를 재시작하거나 새 버전으로 교체해야 하는 상황에서는 재시작 이후 곧바로 정상 트래픽을 전부 흘려보내지 않는다. 먼저 사전에

준비한 검증용 입력 몇 가지를 보내 응답이 예상 범위 안에 있는지 확인한 뒤에야 실제 트래픽을 점진적으로 늘려간다. 이 검증 단계를 건너뛰면 겉으로는 서버가 살아난 것처럼 보이지만 실제로는 여전히 잘못된 결과를 내는 상태로 트래픽을 받게 될 수 있다.

장애 기간 동안 데이터베이스나 다른 시스템에 잘못된 값이 기록되었을 가능성도 함께 확인해야 한다. 1장에서 언급한 데이터 레벨의 그레이스풀 디그레이션이 제대로 되어 있었다면 이 확인 작업은 훨씬 수월해진다. 만약 잘못된 값이 이미 기록되었다면, 장애 발생 시점과 복구 시점을 기준으로 그 구간에 기록된 데이터를 별도로 추려내 재계산하거나 정정하는 절차를 진행해야 한다.

5.5 사후 분석을 다음 장애를 막는 자산으로 만들기

장애가 끝난 뒤의 사후 분석은 누구를 탓하기 위한 절차가 아니라, 이번 장애에서 얻은 정보를 다음 장애 대응에 재사용하기 위한 절차다. 사후 분석에는 장애가 시작된 시점, 감지까지 걸린 시간, 격리까지 걸린 시간, 완전한 복구까지 걸린 시간을 구체적으로 기록해야 한다. 이 시간들을 장애마다 누적해서 비교하면 어느 단계가 매번 가장 오래 걸리는지가 드러나고, 그 단계를 우선적으로 개선할 근거가 생긴다.

사후 분석의 결과물은 문서로만 남기지 말고, 가능하다면 이번 장애에서 새로 발견한 실패 패턴을 자동 검사 규칙이나 서킷 브레이커의 임계값에 반영해야 한다. 이렇게 하면 장애 하나하나가 단순히 지나가는 사건이 아니라 시스템 전체의 신뢰성을 점진적으로 끌어올리는 데이터로 쌓인다.

5.6 런북은 문서가 아니라 실행 가능한 목록이어야 한다

많은 팀이 런북을 길게 서술된 문서로 작성해두고, 정작 장애가 났을 때는 그 문서를 처음부터 끝까지 읽을 시간이 없어 무용지물이 되는 경험을

한다. 런북은 산문이 아니라 그 순간 누구든 순서대로 따라 할 수 있는 짧은 실행 목록으로 만들어야 한다. 각 항목에는 지금 확인해야 할 지표, 그 지표를 확인하는 구체적인 명령이나 화면 위치, 판단 기준에 따라 다음에 무엇을 할지가 명확히 적혀 있어야 한다. 담당자가 바뀌어도, 심지어 처음 이 서비스를 맡은 사람이라도 그 목록만 따라가면 최소한의 대응은 할 수 있어야 좋은 런북이다.

런북은 실제 장애가 없는 평상시에도 정기적인 모의 훈련으로 점검해야 한다. 서킷을 일부러 열어보고, 피처 플래그를 일부러 이전 상태로 돌려보고, 폴백 사다리가 실제로 예상한 단계까지 내려가는지 눈으로 확인하는 훈련을 거치지 않으면, 런북에 적힌 절차가 실제 시스템의 현재 상태와 어긋나 있어도 아무도 모른 채 지나간다. 이 책에서 다룬 그레이스풀 디그레이션, 서킷 브레이커, 폴백 설계, 피처 플래그, 이 장의 런북은 각각 따로 작동하는 장치가 아니라 실패를 감지하고 격리하고 완화하고 복구하는 하나의 흐름을 이루는 부분들이다. 이 흐름을 처음부터 완벽하게 만들 필요는 없다. 지금 만들고 있는 기능 하나에 이 흐름의 첫 조각 하나를 더하는 것에서부터 시작하면 된다.