# AI 프로덕션 옵저버빌리티

LLM 애플리케이션의 실행 흐름을 관찰하고 진단하는 체계

ThakiCloud (Hyojung Han)

# AI 프로덕션 옵저버빌리티

LLM 애플리케이션의 실행 흐름을 관찰하고 진단하는 체계

ThakiCloud (Hyojung Han)

# Contents

# 1 옵저버빌리티란 무엇인가, 왜 LLM에 필요한가

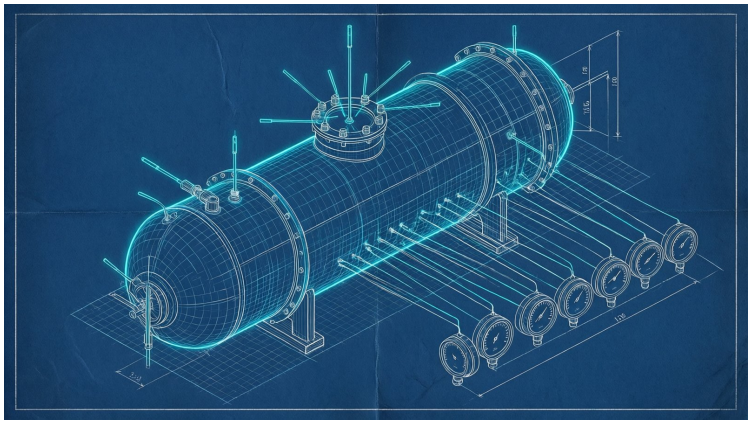

Figure 1.1: 옵저버빌리티란 무엇인가, 왜 LLM에 필요한가

## 1.1 전통 소프트웨어와 LLM의 관찰 가능성 차이

일반적인 백엔드 API를 생각해 보자. 요청이 들어가고 데이터베이스를 조회하고 응답을 반환한다. 각 단계에서 정확히 무슨 일이 일어나는지 메트릭과 로그로 추적할 수 있다. 응답 시간이 300밀리초면 네트워크 50, 쿼리 실행 200, 직렬화 50으로 나눌 수 있다. 불확실성이 거의 없다.

LLM 애플리케이션은 근본적으로 다르다. 입력은 텍스트 프롬프트이고 출력은 텍스트다. 그 사이에는 수십억 개의 파라미터를 가진 신경망이 있으며 같은 프롬프트라도 호출마다 출력이 달라질 수 있다. 전통적인 디버깅 방법인 "입력을 고정하면 출력이 고정된다"는 전제가 깨진다. 출력이 일정하지 않으니 실패의 정의 자체가 흐려진다. 300밀리초 안에 응답이 왔다고 해도 그 안에 담긴 정보가 정확한지는 알 수 없다.

이 차이는 옵저버빌리티 설계 방식에 직접 영향을 미친다. 전통 시스템에서는 "올바른 출력이 왔는가"를 이진적으로 판단할 수 있다. LLM 시스템에서는 "출력이 사실에 부합하는가"를 판단하려면 또 다른 LLM 호출이 필요할 수도 있다. 그래서 LLM 옵저버빌리티는 단순히 모니터링을 더하는 일이 아니라 관찰과 검증 체계를 별도로 구축하는 일이다.

## 1.2 블랙박스에서 회색박스로

LLM을 완전히 불투명한 블랙박스로 취급하면 프로덕션 운영은 불가능에 가깝다. 하지만 완전히 투명하게 만들기도 현실적으로 어렵다. 실용적인 접근은 그 사이 어딘가에 있는 "회색박스"를 목표로 삼는 것이다.

핵심은 불확실성을 없애는 게 아니라, 불확실성을 정량화하고 경계선을 긋는 데 있다. 예를 들어 "이 모델은 95%의 질문에서 사실과 일치하는 응답을 생성한다"는 것이 블랙박스적 인식이라면, "특정 도메인(의료·법률·금융)에서 모델의 hallucination 발생률이 12%이며 이것이 높아지는 입력 패턴은 이렇다"는 것은 회색박스적 인식이다. 후자를 얻으려면 프로덕션에서 지속적으로 출력을 수집하고 검증해야 한다.

실무적으로 회색박스화는 세 축으로 이뤄진다. 모델이 내놓은 결론이 사실인지 추적하는 사실성 메트릭, 입력 조건에 따라 응답 지연이 얼마나 늘어나는지 보는 지연 프로파일, 토큰 소비 패턴이 예상과 맞는지 보는 비용 메트릭이다. 이 세 축을 합치면 LLM 애플리케이션의 상태를 실용적으로 파악할 수 있다.

## 1.3 옵저버빌리티 3대 기둥의 LLM 특화 확장

옵저버빌리티의 전통적 3대 기둥인 메트릭, 로그, 트레이스는 LLM 애플리케이션에도 그대로 적용할 수 있다. 다만 각 기둥이 LLM 고유의

특성을 반영하도록 확장해야 한다.

메트릭은 Prometheus나 CloudWatch 같은 도구로 수집하는 수치형 데이터다. LLM 특화 메트릭으로는 토큰 사용량(입력 토큰 수, 출력 토큰 수), 첫 응답 시간(TTFT), 종단 간 지연, 검증 성공률이 있다. 검증 성공률은 모델 출력이 사전 정의된 사실성 조건을 얼마나 충족하는지 비율로 나타낸 값이다. 이 수치를 시간 축으로 누적하면 모델 업데이트나 입력 분포 변화가 품질에 미치는 영향을 정량적으로 파악할 수 있다.

로그는 요청과 응답의 상세 기록이다. LLM 로그에서는 프롬프트를 저장하되 민감 정보(비밀번호, API 키, 개인 식별 정보)가 포함되지 않도록 마스킹 처리를 자동화하는 게 핵심이다. 또한 같은 프롬프트를 여러 번 호출한 결과를 모두 기록해야 재현 불가능한 출력을 분석할 수 있다. 전통 API와 달리 LLM 로그에서는 어떤 모델 버전이 어떤 프롬프트로 어떤 출력을 냈는지도 함께 기록해야 하므로 로그 스키마가 복잡해진다.

트레이스는 요청 하나가 시스템 안에서 거치는 모든 단계를 시간 순서로 이어붙인 기록이다. LLM 애플리케이션에서 트레이스는 프롬프트 전처리, LLM 호출, 후처리, 외부 도구 호출 등 여러 스팬으로 구성된다. LLM 호출 스팬 안에서 토큰 소비와 응답 생성이 어떻게 진행되는지 구간별로 추적할 수 있다면 지연 최적화의 실마리를 얻는다.

## 1.4 프로덕션 LLM 시스템에서 가장 중요한 4가지 관찰 대상

1년 이상 LLM 시스템을 프로덕션에서 운영한 팀들의 회고와 사고 보고서를 분석하면 문제의 80%가 다음 4가지 관찰 대상 중 하나에서 비롯된다.

먼저 지연 시간이다. 사용자가 체감하는 응답 속도는 첫 토큰이 도착하는 시간과 전체 응답이 완료되는 시간을 분리해서 추적해야 한다. 사용자에게는 첫 토큰이 보이기 시작하는 시점이 실제 응답 시작으로

느껴진다. 네트워크 지연이 200밀리초이고 첫 토큰까지 1초, 전체 생성에 3초가 걸렸다면 사용자가 느끼는 지연은 1.2초다. 네트워크와 생성 시간을 분리하지 않으면 최적화의 방향을 잘못 잡는다.

두 번째 관찰 대상은 토큰 소비 패턴이다. 예상보다 훨씬 많은 토큰을 소비하는 요청이 있다면 프롬프트가 불필요하게 길거나 모델이 같은 내용을 반복해서 생성하고 있는 것일 수 있다. 둘 다 수정 가능한 문제지만 토큰 소비를 요청별로 추적하지 않으면 발견 자체가 불가능하다.

세 번째는 반복되는 실패 패턴이다. LLM이 특정 유형의 입력에서 일관되게 실패한다면 그 패턴을 파악해서 입력 검증 게이트를 두거나, 모델을 교체하거나, 프롬프트를 수정해야 한다. "실패"를 단순한 오류로 정의해서는 안 된다. LLM의 실패는 대개 조용히 일어난다. hallucination이 발생해도 시스템은 정상 응답을 반환하는 것처럼 보일 수 있다.

마지막은 외부 의존성의 상태다. LLM이 도구나 API를 호출하는 구조에서는 그 도구들의 응답 시간과 가용성이 LLM 응답 품질에 직접 영향을 미친다. RAG(Retrieval-Augmented Generation) 시스템에서 벡터 데이터베이스의 검색 지연이 2초 늘어나면 검색 결과를 기반으로 생성하는 LLM의 출력 품질도 함께 떨어진다. LLM 호출뿐만 아니라 LLM이 의존하는 모든 외부 시스템의 상태를 함께 추적해야 한다.

# 2 LLM 실행 추적: 프롬프트 체인과 스팬의 구조화

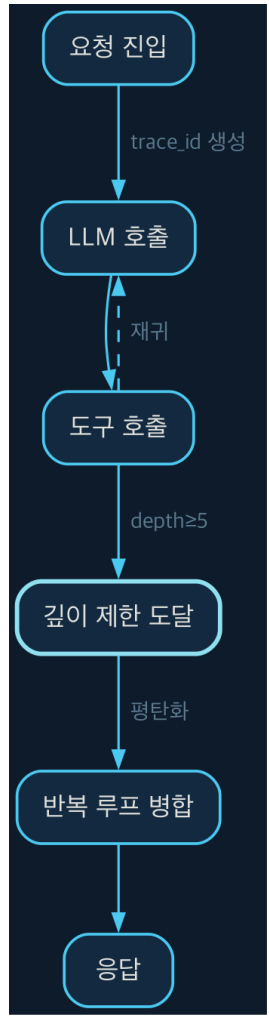


Figure 2.1: LLM 실행 추적: 프롬프트 체인과 스팬의 구조화

## 2.1 LangChain 파이프라인에 추적 IDs 주입하는 3단계

LangChain 기반 애플리케이션에서 실행 추적을 구현하려면 추적 IDs를 명시적으로 주입하고 각 체인 단계를 스팬으로 분리해야 한다. 실제 구현은 세 단계로 나눌 수 있다.

1단계는 추적 컨텍스트를 생성하는 작업이다. 요청이 시스템에 진입하는 지점에서 고유한 추적 ID를 만들고 이를 컨텍스트에 저장한다. Python에서는 uuid나 그에 준하는 라이브러리로 생성하면 된다. 이 ID는 응답이 완료될 때까지 모든 하위 호출에 전달된다. 컨텍스트 저장소는 로컬 변수를 쓰지 말고 threading.local이나 asyncio의 contextvars를 사용해야 비동기 환경에서도 올바르게 전파된다.

2단계에서는 각 체인 구성 요소에 추적 IDs를 전달한다. LangChain의 Runnable 객체는 기본적으로 traceable 데코레이터를 지원하며, 이 데코레이터를 각 체인에 적용하면 상위 추적 ID와 연결된 스팬이 자동으로 생성된다. 직접 구현하는 경우라면 호출 전후에 스팬을 열고 닫는 코드를 명시적으로 작성해야 한다.

3단계는 스팬 속성에 토큰 사용량과 모델 응답 메타데이터를 남기는 작업이다. LLM 호출 스팬에는 모델 이름, 토큰 소비량, 응답 길이, 그리고 가능하면 응답의 첫 토큰까지 걸린 시간(TTFT)을 기록한다. 이런 속성이 갖춰지면 나중에 분석할 때 어떤 요청이 왜 느렸는지, 토큰이 왜 많이 소비되었는지를 스팬 단위로 나눠볼 수 있다.

## 2.2 OpenTelemetry 스팬으로 LLM 호출을 계측화하는 실제 패턴

OpenTelemetry(OTel)는 분산 시스템 추적의 업계 표준이다. LLM 호출을 OTel 스팬으로 계측하면 Jaeger, Zipkin, Tempo 등 다양한 백엔드에서 추적을 조회할 수 있다. 실제 패턴을 살펴보자.

LLM 호출을 위한 스팬을 열 때는 span.start()와 span.end() 사이에 LLM provider의 API 호출을 배치한다. API 호출이 동기적이든 비동기적이든 패턴은 동일하다. 예외가 발생하면 span.set_status()로 상태를 오류로 표시하고 span.record_exception()으로 예외 정보를 기록한다.

스팬 속성으로는 최소한 다음을 포함해야 한다. llm.model에 모델 이름(예: gpt-4o, claude-3-5-sonnet)을, llm.token_usage.input에 입력 토큰 수를, llm.token_usage.output에 출력 토큰 수를, llm.latency.ms에 밀리초 단위 응답 시간을 기재한다. 이렇게 표준화된 속성 이름을 쓰면 여러 모델 제공자를 섞어 쓰는 환경에서도 통일된 대시보드에서 모든 LLM 호출을 집계할 수 있다.

여기서 주의할 점은 API 키나 프롬프트 내용 같은 민감한 정보가 속성에 포함되지 않도록 막는 일이다. 프롬프트는 길이나 토큰 수만 기록하고 내용 자체는 남기지 않는 편이 좋다. 나중에 분석이 필요하면 프롬프트를 별도 저장소(예: S3, 암호화된 DB)에 저장하고 그곳으로의 참조만 스팬에 포함시킨다.

## 2.3 토큰 사용량과 지연 시간을 속성으로 기록하는 스키마

스팬 속성의 스키마 설계는 나중에 데이터를 집계할 때의 편의성을 결정한다. 무작정 많은 속성을 늘어놓기보다 실제로 집계하고 분석할

항목을 선별해서 설계하는 것이 중요하다.

고려해야 할 속성 카테고리는 세 가지다. 첫째는 요청 식별이며, trace_id·span_id·parent_id로 상하 관계를 재구성할 수 있어야 한다. 둘째는 모델 실행 메타데이터로 model_name·token_count_prompt·token_count_comp 포함시킨다. finish_reason은 토큰 생성이 정상 종료(stop)인지 최대 길이 도달(max_tokens)인지 아니면 오류로 끝났는지를 구분해주는 값으로, 생성이 중단된 원인을 파악하는 데 핵심적인 정보가 된다.

셋째는 비즈니스 레벨 정보다. user_id나 session_id처럼 요청자를 식별하는 속성이 있고, request_type처럼 어떤 작업 유형의 요청인지 구분하는 속성도 있다. 예를 들어 고객 지원 챗봇이라면 “환불 요청”과 “상품 문의”를 request_type으로 나눠두면 작업 유형별로 hallucination 발생률이나 평균 지연 시간의 차이를 분석할 수 있다.

스키마 설계에서 흔히 빠지는 함정은 속성을 너무 많이 추가하는 것이다. 처음부터 욕심을 내서 모든 것을 기록하면 속성 이름이 들쭉날쭉해지고 나중에 집계할 때 데이터가 흩어진다. 10개 이하의 핵심 속성을 먼저 정하고, 필요성이 느껴질 때마다 하나씩 추가하는 편이 낫다.

## 2.4 재귀적 호출에서 스팬 연결과 깊이 제한

LLM 에이전트가 도구를 호출하고 그 결과를 다시 LLM에 넣고 다시 도구를 호출하는 구조에서는 재귀적인 스팬 트리가 형성된다. 부모 스팬이 자식 스팬을 만들고 그 자식이 또 다른 자식을 만드는 식이다. 이런 구조가 깊어지면 추적 데이터의 양이 기하급수적으로 늘어나고 시각화도 복잡해진다.

이에 대한 실용적인 해법은 두 가지다. 첫째는 스팬 깊이를 제한하는 방법이다. 예를 들어 깊이가 5 이상이면 더 이상 하위 스팬을 만들지 않고, 그 깊이의 메타데이터만 속성으로 평탄화해서 기록한다. 이렇게 해도

재귀적 에이전트의 전체 경로는 충분히 파악할 수 있다.

둘째는 반복되는 패턴을 하나의 스팬으로 묶는 방법이다. 같은 도구를 세 번째로 호출할 때 본질적으로 같은 작업이 되풀이되고 있다면, 처음 세 번의 호출을 "반복 루프" 스팬 하나로 합쳐서 시각적으로는 하나의 노드로 표현한다. 이렇게 하면 추적 데이터의 양을 줄이면서도 전체 실행 흐름은 그대로 유지할 수 있다.

이 두 기법을 함께 쓰면 100단계짜리 에이전트 루프도 관리 가능한 수준의 추적 데이터로 표현할 수 있다. 깊이 제한과 패턴 통합의 임계값은 애플리케이션의 특성에 맞춰 조정하면 된다. 도구 호출 종류가 다양하고 각 호출의 시간이 중요한 서비스라면 깊이 제한을 낮추고 패턴 통합은 최소화하는 편이 낫다. 반면 비용 최적화가 우선이라면 토큰 소비량 기준의 통합이 더 효과적이다.

# 3 지연 시간 분석: 어디에서 시간이 흐르는가

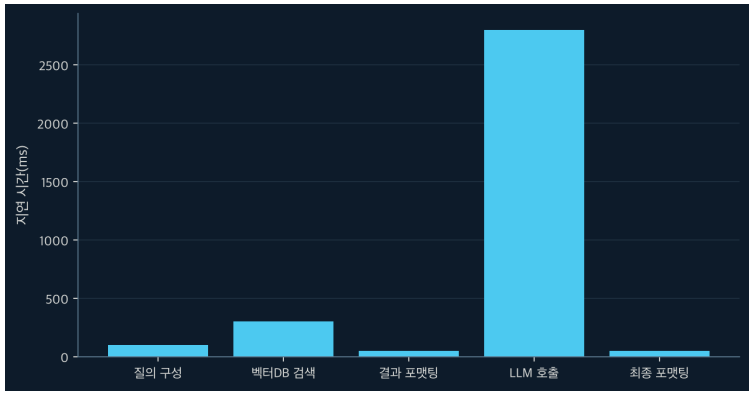


Figure 3.1: 지연 시간 분석: 어디에서 시간이 흐르는가

## 3.1 LLM 응답 지연을 분해하는 산정법

LLM 요청의 종단 간 지연을 하나의 숫자로 뭉뚱그리면 분석이 안 된다. 실제로는 여러 구간이 직렬로 이어진 pipeline이고, 구간마다 지연 원인과 최적화 방법이 다르다. 그래서 지연을 구간별로 쪼개는 것이 분석의 출발점이다.

전체 응답 시간은 세 구간으로 나뉜다. 프롬프트를 LLM provider에 보내고 응답을 받기까지의 네트워크 지연, 모델이 첫 토큰을 생성하기 시작할 때까지 걸리는 첫 토큰 시간(TTFT), 그리고 첫 토큰 이후 전체 응답이 끝날 때까지의 토큰 생성 시간이다. TTFT는 주로 모델의 연산량과 GPU 가용성이 좌우하고, 토큰 생성 시간은 출력 토큰 수에 비례한다.

공식으로 쓰면 전체 지연은 네트워크 왕복 지연 더하기 TTFT 더하기 (출력 토큰 수 × 토큰당 생성 시간)이다. 각 구간의 비중을 실제로 측정해보면 최적화 방향이 뚜렷해진다. 네트워크 지연이 지배적이면 프롬프트

캐싱이나 가까운 edge 위치의 provider를 검토하고, TTFT가 지배적이면 더 작은 모델로 바꾸거나 비동기 streaming으로 응답을 먼저 노출하는 편이 효과적이다.

## 3.2 시리얼 프로파일링으로 지연 상위 기여자 찾기

LLM 애플리케이션에서 지연을 만드는 요인은 LLM 호출 하나만이 아니다. 벡터 데이터베이스 검색, 외부 API 호출, 데이터 전처리, 결과 후처리 등 여러 구간이 함께 작용한다. 시리얼 프로파일링은 전체 요청 처리 경로에서 각 구간이 전체 지연에 얼마나 기여했는지를 순서대로 측정하는 방법이다.

절차는 간단하다. 전체 요청의 시작과 끝에 타임스탬프를 찍고, 그 사이 각 단계의 시작과 끝에도 타임스탬프를 추가한다. 순서대로 기록하면 한 단계가 끝나고 다음 단계가 시작되기까지의 간격을 잴 수 있고, 이 간격을 모으면 전체 처리 시간에서 가장 비중이 큰 구간이 드러난다.

예를 들어보자. 어떤 RAG 기반 질문 응답 시스템에서 전체 응답 시간이 4초였다. 프로파일링 결과는 검색 질의 구성 100밀리초, 벡터DB 검색 300밀리초, 검색 결과 포맷팅 50밀리초, LLM(provider API) 2.8초, 결과 포맷팅 50밀리초였다. 벡터DB 검색보다 LLM provider 응답이 압도적으로 크다. 그러니 최적화 우선순위는 벡터DB가 아니라 LLM 호출 쪽으로 향한다.

반면 벡터DB 검색이 2초, LLM이 1초라면 상황은 뒤바뀐다. 이때는 임베딩 모델 교체나 검색 알고리즘 최적화가 우선이다. 같은 RAG 시스템이라도 입력 문서 크기나 질문 유형에 따라 병목 구간이 달라지므로, 여러 케이스를 반복 측정해보는 편이 좋다.

## 3.3 캐싱으로 지연과 비용 동시 절감

LLM 응답의 상당수는 본질적으로 결정적(deterministic)이다. 같은 프롬프트를 같은 모델에 주면 토큰 단위로 완전히 일치하지는 않아도 의미적으로 비슷한 출력이 나온다. 캐싱 전략은 이 특성을 이용한다.

구현 방식은 두 가지다. 하나는 완전 일치 캐싱(exact-match cache)으로, 프롬프트의 해시값을 키로, LLM 응답을 값으로 저장한다. 같은 프롬프트가 오면 API 호출 없이 캐시된 응답을 그대로 돌려주므로 응답 시간이 수 밀리초까지 줄고 비용은 100% 절감된다. 다만 프롬프트가 조금이라도 다르면 캐시 히트가 나지 않으므로, 프롬프트 안에 날짜나 사용자 이름 같은 동적 변수가 들어 있으면 효과를 기대하기 어렵다.

다른 하나는 유사 캐싱(semantic cache)이다. 임베딩 모델로 프롬프트의 벡터 표현을 만들고, 벡터 유사도로 이전에 본 프롬프트와 얼마나 비슷한지 판단한다. 유사도가 임계값을 넘으면 저장된 응답을 살짝 손봐서 돌려준다. 이 방식은 적용 범위가 넓은 대신, 유사도 계산 비용과 응답 수정 로직의 복잡성이 따라붙는다.

캐싱을 도입할 때 주의할 점은, 응답의 신선도가 중요한 경우에는 캐시를 쓰면 안 된다는 것이다. 환율 정보, 실시간 뉴스, 개인화 추천처럼 입력에 따라 출력이 급격히 바뀌는 작업에는 캐싱이 맞지 않는다. 캐시 히트율 자체도 메트릭으로 계속 추적해야 한다. 히트율이 5% 이하면 캐시 인프라 운영 비용 대비 효과가 미미하므로 다시 검토할 필요가 있다.

## 3.4 비동기 배치 처리와 스트리밍의 지연 프로파일 비교

LLM 시스템에서 응답성을 높이는 아키텍처로는 비동기 처리와 스트리밍이 있다. 두 방식은 지연 프로파일이 근본적으로 다르므로 선택 기준을 분명히

알아둘 필요가 있다.

비동기 배치 처리는 여러 요청을 묶어 한 번에 LLM에 보내고 각 요청의 응답을 따로 받는 방식이다. 개별 요청의 종단 간 지연은 배치 처리를 안 할 때보다 늘어날 수 있다. 대신 GPU 활용 효율이 올라가 처리량이 늘고 (provider 관점에서) 비용은 줄어든다. batch size가 클수록 GPU 효율은 오르지만 개별 요청의 대기 시간도 함께 늘어난다. 이건 tradeoff라서, 서비스 성격에 맞춰 균형점을 잡아야 한다.

스트리밍은 토큰이 생성되는 대로 사용자에게 바로 보여주는 방식이다. 전체 응답이 끝나지 않아도 첫 토큰이 도착하는 순간 화면에 나타나므로 체감 지연이 크게 줄어든다. LLM provider의 TTFT가 같다면 전체 응답 길이와 무관하게 사용자는 거의 즉시 응답이 시작됐음을 확인한다. streaming이 줄이는 건 전체 응답 시간이 아니라 perceived latency, 즉 체감 지연이다.

실무적으로는 이렇게 쓰면 된다. 사용자와 대화하는 인터페이스라면 스트리밍을 기본으로 쓴다. 사용자가 타이핑을 멈추고 기다리는 동안 첫 토큰이 보이기 시작하면 기다림의 체감이 줄어든다. 대시보드나 백그라운드 처리처럼 사용자가 응답을 직접 기다리지 않는 경우에는 batching이 맞다. 여러 요청을 묶어 처리하면 cost-per-token이 낮아지고 GPU 활용률도 올라간다. 두 방식을 섞어 쓸 수도 있다. 초기 응답은 스트리밍으로 빠르게 보여주고 추가 정보는 백그라운드 batch로 생성하는 식이다.

# 4 발동 탐지: 구조적 검증 체계 구축

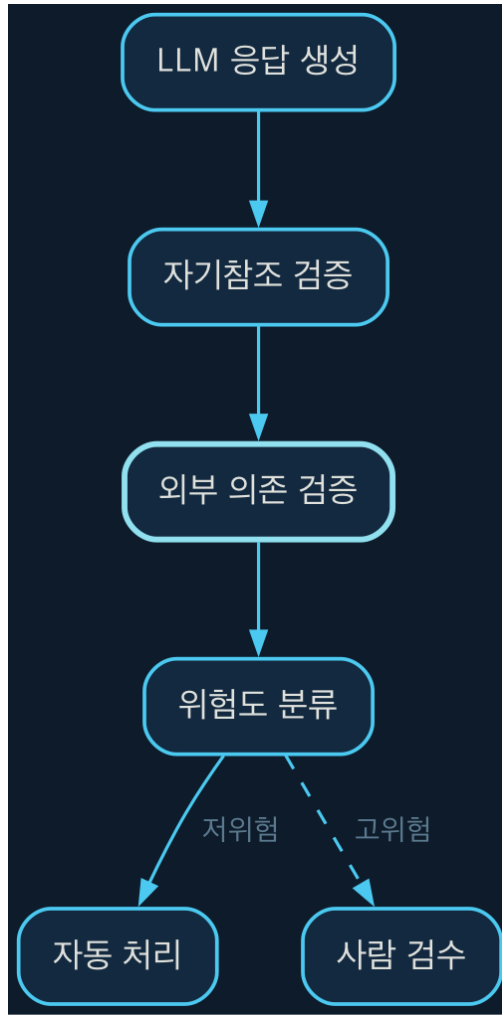


Figure 4.1: 발동 탐지: 구조적 검증 체계 구축

## 4.1 자기참조 검증과 외부 의존 검증의 2단계 게이트

LLM이 사실이 아닌 내용을 만들어내는 현상을 hallucination이라 부른다. 이 용어는 AI 분야에서 나왔지만 오늘날 LLM 운영에서는 흔히 마주치는 문제다. 현재 기술로는 hallucination을 완전히 없앨 수 없다. 그러므로 현실적인 목표는 발생 자체를 줄이고, 발생했을 때 빠르게 탐지해 대응하는 체계를 갖추는 것이다.

효과적인 검증 체계는 2단계 게이트로 구성된다. 1단계는 자기참조 검증(self-verification)이다. 모델이 생성한 응답 자체의 사실 여부를 모델에게 다시 확인시키는 방식이다. 예를 들어 "2024년 한국 GDP 성장률은 2.4%였다"고 생성했다면, 같은 모델(또는 다른 인스턴스)에 "이 문장의 사실 여부를 확인해줘: 2024년 한국 GDP 성장률은 2.4%였다"를 요청한다. 이렇게 하면 생성 단계에 확인 절차를 끼워 넣지 않고도, 만들어진 내용의 사실성을 따로 검증할 수 있다.

2단계는 외부 의존 검증(external verification)이다. 자기참조 검증은 모델의 출력을 모델 스스로 다시 확인하는 방식이라 같은 편향이 반복될 수 있다. 외부 데이터로 대조하는 편이 더 믿을 만하다. 검색 엔진, 사실 데이터베이스, 구조화된 지식 그래프 등을 외부 데이터 소스로 쓸 수 있다. 내부 지식만으로 검증하기보다 실제 외부 데이터를 끌어와 대조하는 쪽이 hallucination 탐지에 효과적이다.

## 4.2 구조화 응답에서 사실성 오류 자동 포착

LLM 출력을 구조화된 형태로 파싱해서 쓰는 시스템에서는 파싱 단계에서 사실성 오류를 잡아낼 기회가 생긴다. 이 방식은 특히 도메인 지식 기반 질의응답에서 효과적이다.

구체적으로는 이렇다. 응답의 구조화된 필드마다 검증 함수를 정의한다. 예컨대 "창립년" 필드가 있다면 해당 값이 실제 회사 설립 연도와

일치하는지 지식 그래프나 위키피디아 API로 확인한다. “직원 수” 필드가 있다면 신뢰할 수 있는 정보원(공시 자료, LinkedIn 등)에서 현재 추정치를 조회해 비교한다. 이렇게 필드별 검증 함수를 구성해 두면 파싱과 동시에 각 필드의 사실성 점수를 계산할 수 있다.

이 방식의 이점은 검증이 필드 단위로 세분화된다는 데 있다. 응답 전체를 사실이라고 판정할 필요 없이 문제가 된 특정 필드에만 플래그를 세울 수 있다. 사용자에게는 문제 있는 부분만 수정하게 하거나 그 부분에 별도 안내 문구를 보여줄 수 있다.

주의할 점은 검증 함수의 오탐(false positive)도 관리해야 한다는 것이다. 필드 값은 맞는데 검증 데이터원이 낡았다면 검증이 실패로 표시될 수 있다. 그래서 검증 결과는 이 단계에서 자동으로 조치하기보다 관리자에게 알림을 보내거나 사용자에게 경고 수준으로 표시하는 편이 적절하다. 자동 조치까지 하려면 검증 함수의 정밀도가 충분히 높아졌다는 확신이 있어야 한다.

## 4.3 도구 호출 결과와 LLM 생성부의 연결 무결성 검증

ReAct 패턴이나 에이전트 구조에서 LLM은 외부 도구를 호출하고 그 결과를 바탕으로 응답을 생성한다. 이때 생길 수 있는 문제가 도구 호출 결과와 LLM 생성부 사이의 불일치다. 예컨대 날씨 API가 “서울 28도, 흐림”이라고 반환했는데 LLM이 “오늘 서울은 맑고 기온이 30도입니다”라고 답하는 경우다. 도구가 준 값과 모델이 만든 문장 사이에 틈이 생긴 것이다.

이 불일치를 잡으려면 도구 호출 결과와 생성부의 연결고리를 추적할 수 있는 체계가 필요하다. 가장 실용적인 방법은 도구 호출 결과를 해당 LLM 호출의 입력에 명시적으로 포함시키는 것이다. LangChain에서는 도구 호출 결과를 messages에 tool_message로 추가해 모델이 이를 참고해서 생성하도록 한다. 이 구조에서는 어떤 도구 결과가 어떤 생성에 쓰였는지 추적할 수 있다.

연결 무결성을 검증하는 한 방법은 모델에게 도구 호출 결과를 요약하게 하는 것이다. 도구 호출이 끝나면 모델에게 "검색 결과: [검색 결과를 간략히 요약]" 같은 형태로 포맷팅하게 한다. 요약 과정에서 검색 결과의 핵심 정보가 제대로 반영됐는지 한 번 더 확인할 수 있다. 이 요약 단계는 LLM 호출 비용을 추가로 발생시키므로 모든 도구 호출에 적용하기보다 응답의 중요도가 높은 경우에만 선택적으로 쓰는 편이 좋다.

## 4.4 사람-루프 선별적 검수와 자동화된 사전 필터의 조합 전략

Hallucination 대응에서 완전히 자동화된 검증 체계만으로 모든 경우를 커버하기는 현실적으로 어렵다. 그래서 자동화와 인간 검수를 적절히 조합하는 편이 실용적인 전략이 된다.

조합 전략의 핵심은 위험도 계층화다. 시스템 응답이 사용자의 판단이나 행동에 직접 영향을 미치는 고위험 영역에서는 자동 검증 뒤에 반드시 사람이 검수하는 단계를 둔다. 의학적 진단 보조, 법적 조언, 투자 추천 등이 여기 해당한다. 이런 영역은 hallucination으로 인한 피해가 너무 크기 때문에, 자동 검증이 아무리 정교해도 최종적으로 인간 전문가의 확인을 거치는 편이 안전하다.

반면 일상적인 정보 조회나 창의적 글쓰기처럼 hallucination의 영향이 제한적인 영역에서는 자동 검증만으로 충분한 경우가 많다. 이런 영역에서는 검증 결과에 따라 신뢰도를 표시하거나, 신뢰도가 낮으면 사용자에게 fact-check 여부를 맡기는 편이 적절하다.

자동화 사전 필터는 구체적 사실이 포함된 응답을 중점적으로 검증하도록 설계하는 편이 효과적이다. 추상적인 설명이나 일반 상식은 hallucination 가능성이 낮다. 반대로 특정 회사명, 날짜, 숫자가 포함된 응답은 자동으로 강화 검증한다. 이렇게 하면 검증 비용을 모든 응답에 균등하게 쓰지 않고

실제로 위험이 높은 응답에 집중 배분할 수 있다.

# 5 토큰 소비와 비용 모니터링: 예산 관리의 실전

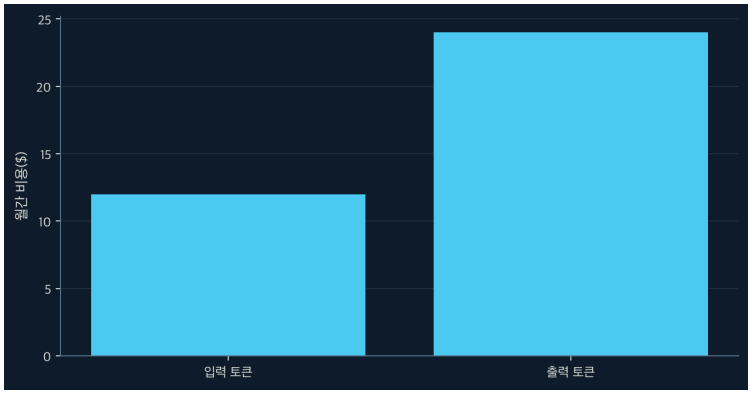


Figure 5.1: 토큰 소비와 비용 모니터링: 예산 관리의 실전

## 5.1 요청 단위와 세션 단위의 3단위 토큰 집계 구조

LLM 사용 비용의 대부분은 토큰 소비에서 나온다. 비용을 제대로 관리하려면 토큰 사용량을 여러 단위 계층으로 나눠 추적해야 한다. 단위를 잘못 잡으면 비용의 실체가 보이지 않는다.

요청 단위는 가장 기본적인 집계 단위다. LLM API 호출 한 번에서 소비된 입력 토큰과 출력 토큰을 각각 기록한다. 여기에 호출 시점의 타임스탬프와 요청 유형을 붙이면 시간대별·유형별 소비 패턴이 드러난다. 이를테면 "오후 6시에 환율 조회 요청이 몰리면 그 시간대 입력 토큰이 급증한다"는 식이다. 요청 단위 데이터는 최적화의 가장 기본이 되는 원료다.

세션 단위는 한 사용자의 대화 흐름 전체를 추적한다. 하나의 대화 안에서 여러 번의 요청이 일어나고 각 요청의 토큰 소비가 누적된다. 세션별로

토큰 총 소비량을 추적하면 특정 사용자가 비정상적으로 많은 토큰을 쓰고 있는지 파악할 수 있다. 세션당 평균 토큰 소비량이 갑자기 늘었다면 프롬프트가 불필요하게 길어졌거나 모델이 장문의 응답을 반복 생성하고 있다는 뜻일 수 있다.

사용자 단위는 세션보다 더 큰 척도다. 한 사용자가 일정 기간(예: 하루, 한 달) 동안 서비스 전체에서 소비한 토큰을 집계한다. 이 단위로 추적하면 사용자별 소비 분포가 드러난다. 상위 20% 사용자가 전체 토큰 소비의 80%를 차지한다면 heavy user를 위한 별도 요금제 도입이나 소비 상한 설정을 검토할 수 있다.

3단위 추적은 단위별로 다른 관리 조치로 이어진다. 요청 단위는 비효율적인 프롬프트를 찾아 최적화하는 데 쓰인다. 세션 단위는 대화 흐름 자체를 효율화하는 데 쓰인다. 사용자 단위는 비즈니스 모델 차원의 의사결정에 쓰인다.

## 5.2 임계치 초과 알람 설계: 동적 임계값과 정적 임계값의 Tradeoff

비용 관리에서 알람은 비용이 통제 불가능해지기 전에 경고하는 핵심 수단이다. 알람 설계에서 가장 중요한 결정은 임계값을 어떻게 잡느냐이다.

정적 임계값은 가장 단순하다. 예컨대 “세션당 토큰 소비가 10만 토큰을 넘으면 알람”처럼 고정된 수치를 정한다. 장점은 구현이 단순하고 임계값을 넘었을 때 의미가 명확하다는 데 있다. 다만 비정상 소비를 놓칠 수 있다. Heavy user의 평균 세션 토큰 소비가 8만 토큰이라면 10만 토큰짜리 임계값은 대부분 정상 범위의 소비에서만 걸린다.

동적 임계값은 과거 데이터를 바탕으로 지금 적정한 임계값을 계산한다. 이동평균이나 지수 평활화 기법으로 최근 N개 세션의 평균 소비량을 구하고 거기에 표준편차의 몇 배를 더해 임계값으로 삼는다. 이렇게 하면 heavy

user의 정상 소비는 임계값을 건드리지 않으면서도 갑자기 소비가 급증한 경우는 포착할 수 있다.

그러나 동적 임계값도 만능은 아니다. 데이터가 충분하지 않은 초기에는 과거 데이터 자체가 형성되지 않아 제대로 작동하지 않는다. 시장 환경이나 비즈니스상의 변화로 정상적인 소비 패턴 자체가 바뀐 경우에도 임계값 갱신이 따라가지 못할 수 있다. 그래서 실용적인 접근은 동적 임계값을 기본으로 쓰되 정적 임계값을 백업으로 두는 것이다. 예컨대 어떤 세션의 토큰 소비가 평소의 3배를 넘으면 절대 임계값인 15만 토큰에 도달했는지와 무관하게 알람이 울리도록 하는 방식이다.

## 5.3 모델별·토큰 유형별 비용 분해로 효율화 기회 포착

LLM 사용 비용을 최적화하려면 "어디에서 비용이 발생하는가"를 세밀하게 뜯어봐야 한다. 대략적인 집계만으로는 최적화 기회가 보이지 않는다.

비용 분해의 첫 번째 차원은 모델별이다. 같은 작업이라도 어떤 모델을 쓰느냐에 따라 비용이 몇 배씩 달라진다. 예컨대 질문 응답 작업에서 GPT-4o-mini는 GPT-4o 대비 토큰당 비용이 약 5분의 1 수준이다. 응답 품질이 허용한다면 더 작은 모델로 바꾸는 것만으로 비용이 크게 줄어든다. 이를 위해서는 먼저 어떤 작업에 어떤 모델을 쓰고 있는지부터 정리해야 한다. 작업별로 모델을 배정하고 각 모델의 소비량을 추적하면 "이 작업은 비싼 모델을 쓰고 있는데 저렴한 모델로 바꿔도 품질 저하가 미미하다"는 기회를 찾아낼 수 있다.

비용 분해의 두 번째 차원은 토큰 유형별이다. 입력 토큰과 출력 토큰의 단가는 다르다. 출력 토큰이 보통 더 비싸므로 입력 프롬프트를 짧게 유지하는 것만으로도 비용을 줄일 수 있다. 예컨대 Retrieval-Augmented Generation에서 검색 결과를 프롬프트에 전부 넣는 대신 핵심 발췌(digest)만 담는 방식으로 입력 토큰을 줄이면 비용이 절감된다. 대화 이력 전체를 입력에 넣는 대신 요약만 유지하는 것도 입력 토큰 비용을

낮추는 방법이다.

실제 분해 산출물의 예를 들어본다. 한 달간 전체 토큰 소비가 1억 2천만 토큰이었다. 이 중 입력 토큰이 8천만, 출력 토큰이 4천만이다. 입력 토큰 비용이 $0.15 per million, 출력 토큰 비용이 $0.60 per million이라면 입력이 $12, 출력이 $24로 합계 $36이고, 출력 토큰이 전체 비용의 약 67%를 차지한다. 토큰 수로는 출력이 3분의 1인데 비용으로는 3분의 2다. 출력 토큰을 줄이는 것이 비용 절감의 우선순위가 된다. 출력 토큰을 줄이는 구체적 방법으로는 max_tokens 제한을 낮추거나 프롬프트에 "답변을 3문장 이내로"와 같은 지시를 추가하는 것이 있다.

## 5.4 LangSmith 대안으로 프로덕션 수준 예산 추적 구축

토큰 소비와 비용을 추적하는 도구 가운데 LangSmith가 가장 널리 알려진 유료 서비스다. 하지만 LangSmith 없이도 비슷한 수준의 추적 체계를 구축할 수 있다. 핵심은 요구 사항을 정확히 파악하고 자체 구축 시 비용 대비 효과를 따져보는 데 있다.

자체 구축 시 필요한 구성 요소는 네 가지다. 첫째는 추적 데이터 수집 계층이다. LLM API 호출을 가로채 토큰 사용량, 지연, 모델 정보를 기록하는 미들웨어나 데코레이터가 여기 해당한다. LangChain, OpenAI SDK, Bedrock SDK 모두 Hook 포인트를 제공한다. 둘째는 시계열 저장소다. 수집된 메트릭을 시계열로 저장해 시간대별·모델별·작업 유형별 소비 추이를 분석할 수 있게 한다. Prometheus나 InfluxDB가 이 역할을 맡는다. 셋째는 대시보드다. 저장된 데이터를 시각화해 이해관계자(개발자, 경영진, 재무팀)가 각자의 관점에서 비용 현황을 파악할 수 있게 한다. Grafana나 Superset로 구현할 수 있다. 넷째는 알람이다. 앞서 설명한 임계치 기반 알람을 대시보드와 연동해 비용이 임계치를 넘으면 Slack이나 Email로 알림받을 수 있게 한다.

자체 구축의 단점도 분명하다. 구축과 운영에 engineering 자원이 든다. 데이터 양이 늘어나면 저장소 비용도 함께 늘어난다. LangSmith는 이런 기능을 별도 구축·운영 없이 제공하므로 engineering 자원이 부족한 조직에서는 유료 서비스 도입이 더 경제적일 수 있다. 반면 자체 구축은 추적 스키마와 데이터 저장 방식을 완전히 통제할 수 있어 민감한 프롬프트 내용을 외부 서비스로 보내지 않아도 된다는 점이 주요한 이점이다.