AI 비용 공학

프로덕션 LLM 앱의 토큰·라우팅·캐시 설계

ThakiCloud (Hyojung Han)

AI 비용 공학

프로덕션 LLM 앱의 토큰·라우팅·캐시 설계

ThakiCloud (Hyojung Han)

Contents

1 비용은 부수물이 아니라 첫 번째 제약이다 1

2 토큰을 세는 기술: 프로덕션 계측 체념 6

3 모델 라우팅: 같은 문제를 싸게 푸는 기술 11

4 캐시 설계: 반복 비용을 구조적으로 줄이는 패턴 17

1 비용은 부수물이 아니라 첫 번째 제약이다

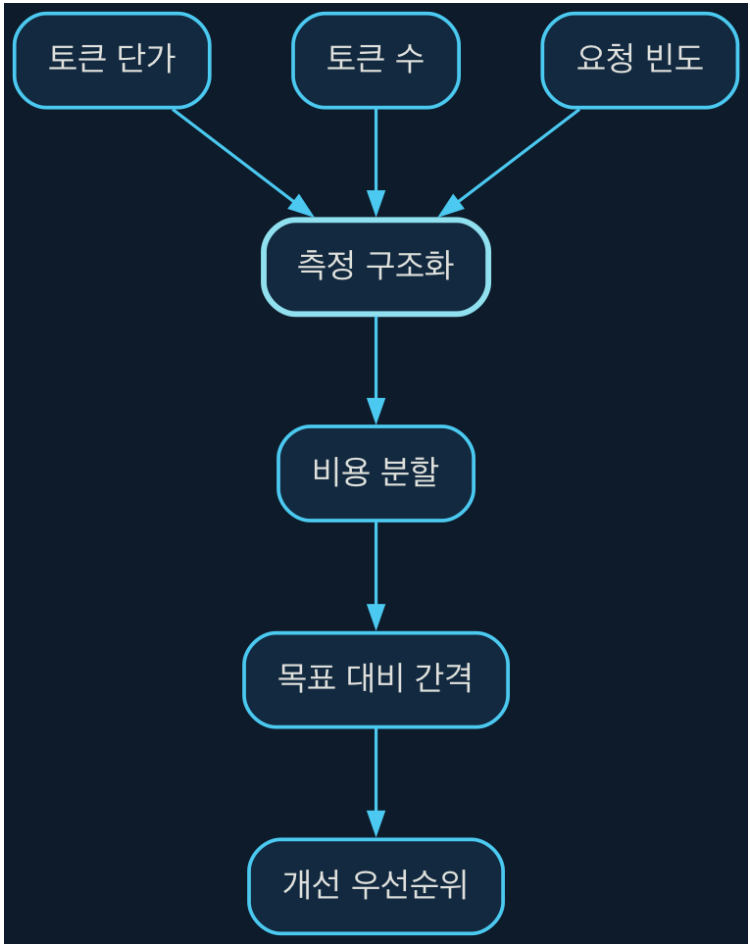

Figure 1.1: 비용은 부수물이 아니라 첫 번째 제약이다

1.1 시작하기 전에: 비용을 외면하는 조직의 공통 패턴

LLM 기반 앱을 프로덕션에 올린 팀 대부분은 비용 문제를 나중으로 미룬다. 프로토타입으로 시작해 사용자가 늘고 청구서가 날아온 뒤에야 비용 구조를 들여다본다. 그 사이 무엇이 잘못되고 있는지는 아무도 재지 않는다.

비용을 부수물로 미뤄두는 동안 팀은 근거 없는 추측에 기댄다. "모델을 바꾸면 비용이 줄겠지"라는 막연한 기대, "캐시를 넣으면 해결될 거야"라는 선입견, "사용자가 늘면 단가가 내려갈 거야"라는 희망 섞인 계산. 셋 다 비용을 측정한 적이 없어서 가능한 판단이다.

비용을 모르면 결정 세 가지가 어긋난다. 어떤 기능이 벌어들이는 돈보다 더 많이 쓰는지 알 수 없고, 기술 선택이 알아채지 못한 비용 함정이 될 수 있으며, 팀이 비용을 문제로 여기지 않으니 개선 노력도 저절로 생기지 않는다.

1.2 비용의 3요소: 토큰 단가, 토큰 수, 요청 빈도

LLM 비용은 세 가지 변수를 곱한 값이다. 이 구조를 먼저 잡아야 어디를 들여다봐야 할지 보인다.

토큰 단가는 모델마다 입력과 출력에서 다르다. 같은 Claude 모델도 Haiku, Sonnet, Opus로 갈수록 1,000토큰당 비용이 올라간다. 같은 작업에 더 비싼 모델을 고르면 단가가 올라간다. 단가는 바꾸기 가장 쉬운 변수이기도 하다.

토큰 수는 하나의 요청에서 소비된 입력 토큰과 출력 토큰의 합계다. 시스템 프롬프트가 길면 입력 토큰이 늘어난다. 응답을 길게 생성하면 출력 토큰이 늘어난다. 프롬프트를 줄이고 불필요한 컨텍스트를 잘라내면 토큰 수가 줄어든다.

요청 빈도는 시간당, 사용자당, 기능당 발생하는 호출 횟수다. 같은 응답을

반복해서 생성하면 그 횟수만큼 비용이 겹쳐 쌓인다. 중복 호출을 줄이거나 결과를 재활용하면 이 요인을 낮출 수 있다.

세 변수 모두 독립적으로 최적화할 수 있다. 그러나 대부분 팀은 단가에만 집중하고 나머지 두 변수를 놓친다.

1.3 비용 곡선: 프로토타입에서 프로덕션으로 전환하는 시점

비용은 이용자 수와 요청 패턴이 바뀌면서 곡선을 그린다. 프로토타입 단계에서는 요청 빈도가 낮고 토큰 수도 적어서 비용이 작아 보인다. 이 구간에서는 "우리 앱은 비용이 적게 든다"는 판단을 내리기 쉽다.

사용자가 늘어난 프로덕션 단계에 접어들면 이야기가 달라진다. 요청 빈도는 10배로 늘고, 사용자가 만들어내는 컨텍스트는 길어지며, 기능이 복잡해지면서 다단계 프롬프트까지 쓰게 된다. 비용 곡선이 가파르게 오르는 구간이다.

이 전환점에서 결정이 뒤집힌다. 비용을 나중으로 미루는 전략은 이미 정해둔 가격 설정과 기능 범위, 기술 스택을 흔든다. 프로토타입에서 고른 모델과 아키텍처가 프로덕션에서 비효율로 드러나면 그걸 고치는 비용이 추가로 든다.

비용 우선 설계는 이 곡선이 오르기 전에 대략적인 규모와 비용 목표부터 정하는 일이다. 1,000사용자당 월 500달러로 잡으면 기능당 토큰 예산이 정해지고, 그 예산 안에서 어떤 모델과 아키텍처가 가능한지 거꾸로 계산한다.

1.4 비용 우선 설계의 3단계 프레임워크

비용을 설계의 첫 번째 제약으로 두는 조직은 대부분 반복 가능한 3단계를 밟는다.

1단계: 측정 가능한 구조 만들기. 비용을 눈에 보이게 만드는 작업이다. 요청마다 입력 토큰 수, 출력 토큰 수, 모델 종류, 요청 시각을 기록한다. 구조는 단순하게 시작해도 된다. 요청마다 모델별 토큰 소비량과 시간을 남겨두면 비용 분석의 기반이 된다.

2단계: 비용을 쪼개 책임을 정하기. 기능별, 사용자 세그먼트별, 모델별로 비용을 나눈다. 전체 비용이 아니라 "어떤 기능이 전체 비용의 60%를 쓰는지"를 알아야 개선 대상이 분명해진다. 뭉뚱그린 전체 비용만으로는 행동으로 이어지지 않는다.

3단계: 목표 대비 간격 측정. 기능별로 감당할 수 있는 비용 상한을 정하고 현재 값과 목표 값 사이의 간격을 잰다. 이 간격이 개선 우선순위를 정한다. 간격이 가장 큰 곳이 가장 먼저 손봐야 할 대상이다.

이 프레임워크가 단순하면서도 효과가 있는 이유는 "측정 없이 개선 없다"는 원칙을 구조로 강제하기 때문이다. 비용 문제 대부분은 측정되지 않은 비용에서 시작된다.

1.5 비용을 모를 때 발생하는 의사결정 왜곡

팀이 비용 구조를 모르면 의사결정 세 가지가 어긋난다.

기능 취사선택. 비용이 보이지 않으면 "이 기능 정도는 넣어도 되겠다"는 판단을 쉽게 내린다. 기능을 넣는 비용이 유지 비용보다 낮게 느껴져 결과적으로 비용이 쌓인다. 모든 기능에 토큰 소비가 따라붙는다는 인식이 없으면 기능 요청은 끝없이 늘어난다.

모델 선택. 더 좋은 모델을 쓰면 결과 품질은 오르지만 같은 기능의 비용도 함께 오른다. 비용을 모르면 "괜찮은 품질의 모델"과 "비싸지만 좋은 모델" 사이의 트레이드오프를 판단할 근거가 없다. 품질을 어느 수준으로 정할지 결정하는 과정이 비합리적으로 흐른다.

스케일링 판단. 사용자가 늘어날 때 비용이 어느 속도로 늘어나는지 모르면 "성장이 곧 수익"이라는 잘못된 등식을 믿게 된다. 실제로는 성장 속도가 비용 증가 속도보다 느리면 오히려 손해다. 비용 증가 속도와 수익 증가 속도를 비교하지 않는 성장은 위험하다.

비용을 "모른다"는 것은 비용이 존재하지 않는다는 뜻이 아니다. 다만 그 비용을 피할 수 없는 결정들을 비용 정보 없이 내리고 있다는 뜻이다.

1.6 정리하며

LLM 비용의 본질은 토큰 단가, 토큰 수, 요청 빈도라는 세 요소를 곱한 값이다. 이 구조를 먼저 잡으면 어디를 최적화해야 할지 보인다. 비용 우선 설계는 비용을 "나중에" 챙기는 게 아니라 처음부터 제약 조건으로 두는 일이다. 다음 장에서는 이 구조를 실제로 측정하는 계측 체계를 만든다.

2 토큰을 세는 기술: 프로덕션 계측 체념

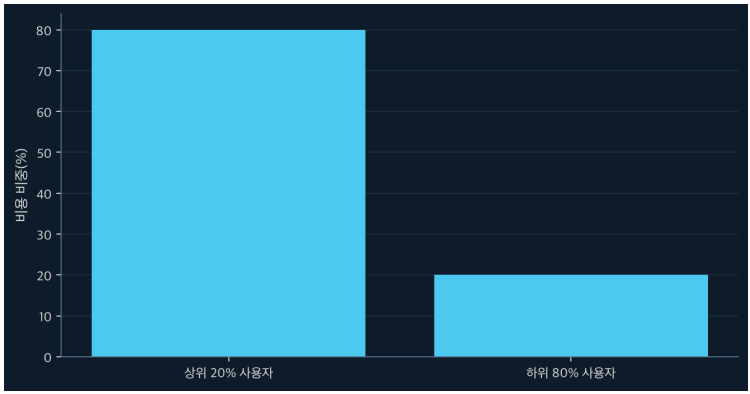

Figure 2.1: 토큰을 세는 기술: 프로덕션 계측 체념

2.1 SDK가 알려주는 토큰 수와 청구 금액의 차이

LLM SDK는 대화가 끝난 후 입력 토큰 수와 출력 토큰 수를 반환한다. 이 숫자를 청구서에 넣으면 “SDK가 세준 대로 지불한다”라고 믿게 된다. 실제로는 차이가 있다.

차이가 생기는 원인은 세 가지로 나뉜다. 우선 청구를 위해 모델 제공자가 내부적으로 토큰화를 다시 수행한다. SDK가 반환하는 토큰 수는 SDK 내부에서 계산한 결과일 뿐이고, 실제 청구 금액은 모델 제공자 서버가 다시 계산한 수치를 기준으로 삼는다. 그래서 둘 사이에 미세한 차이가 생긴다.

다음으로 특수 토큰을 다루는 방식이 모델마다 다르다. 시스템 프롬프트 앞뒤에 붙는 특수 토큰의 종류와 개수가 모델별로 제각각이라, SDK가 “입력 100토큰”이라고 알려줘도 실제 청구에 포함되는 수는 달라질 수 있다.

마지막은 요청 캐싱 할인이다. Anthropic의 Claude 같은 provider는 반복되는 입력 토큰 시퀀스에 할인을 적용하는데, SDK가 반환하는 원래 토큰 수에는 이 할인이 반영되지 않는다. 그래서 청구 금액은 SDK가 알려준 토큰 수보다 낮게 나온다. 할인을 잘 활용하면 비용이 실제로 줄어들지만, SDK 수치만 봐서는 그 효과를 알아챌 수 없다.

SDK 토큰 수는 상대적 비교와 트렌드 분석에는 유용하지만, 실제 청구 금액을 예측하려면 보정 계수가 필요하다는 뜻이다. billing 데이터로 SDK 수치와 청구 금액 사이의 비율을 미리 측정해두면, 이후에는 SDK 수치에 그 계수를 곱해 예상 청구 금액을 빠르게 추정할 수 있다.

2.2 3단위 측정 설계: 요청, 세션, 사용자

비용을 효과적으로 줄이려면 비용이 발생하는 단위부터 정의해야 한다. 대부분의 팀이 여기서 걸린다. "전체 비용"만 보면 개선 대상이 드러나지 않는다. 요청, 세션, 사용자라는 세 단위에서 각각 비용을 측정하면 개선의 실마리가 보인다.

요청 단위. 하나의 API 호출이다. 입력 토큰 수, 출력 토큰 수, 모델 종류, 요청 시작 시각, 요청 종료 시각을 기록한다. 여기에 응답 시간과 성공/실패 여부를 붙이면 비용 대비 성능의 비율도 계산할 수 있다.

요청 단위 계측의 핵심은 모든 요청을 빠짐없이 잡는 것이다. 샘플링이 아니고 전수다. 비용 이상을 감지하려면 100%의 데이터가 필요하다. 샘플링은 트렌드 파악용으로만 쓰고, 정확한 비용 배분은 전수 데이터로 해야 한다.

세션 단위. 하나의 대화 세션에서 발생한 전체 비용이다. 세션 내에서 몇 번의 요청이 오갔고, 누적 입력 토큰과 출력 토큰이 얼마나 되는지 센다. 세션 단위 계측이 중요한 이유는 기능별로 비용을 볼 때 세션이 자연스러운 경계가 되기 때문이다.

예를 들어 요약 기능을 호출하는 세션과 챗봇 기능을 호출하는 세션이 있다고 하자. 세션 단위로 비용을 구분하면 "요약 기능의 1세션 평균 비용"과 "챗봇 기능의 1세션 평균 비용"을 비교할 수 있다. 이 비교가 기능 취사선택의 근거가 된다.

사용자 단위. 특정 사용자가 일정 기간에 소비한 총 토큰 비용이다. 사용자 단위 계측에는 사용자 식별자와 기간이 붙는다. monthly per user, weekly per user 등으로 집계하면 "상위 10%의 사용자가 전체 비용의 몇 %를 쓰고 있는지"가 보인다.

이 비율은 대체로 팔레토 법칙을 따른다. 상위 20%의 사용자가 전체 비용의 80%를 만들어내는 경우가 많다. 이런 패턴이라면 상위 사용자 비용만 관리해도 전체 비용 대부분을 통제할 수 있다.

2.3 3단위 계측을 동시에 구현하는 구조

세 단위의 계측을 개별 시스템으로 만들면 중복과 불일치가 생긴다. 대신 요청 단위에서 출발해 세션, 사용자로 집계하는 구조가 효율적이다.

요청을 받는 순간 토큰 수와 모델 정보를 로그에 찍는다. 이 로그에는 session_id와 user_id를 함께 붙인다. 이후 세션이 종료되면 같은 session_id로 그룹핑해 세션 단위 비용을 집계한다. 주기적으로 사용자별로 집계하면 사용자 단위 비용이 완성된다.

이 구조의 장점은 세분도가 적절하다는 데 있다. 요청 단위 로그에서 session_id를 묶으면 세션 비용이 나오고, user_id를 묶으면 사용자 비용이 나온다. 같은 원본 데이터에서 다른 세분도의 지표가 만들어지므로 불일치가 없다.

구현 시 고려할 점은 로그 볼륨이다. 요청이 많은 시스템에서는 전수 요청 로그 자체가 비용이 될 수 있다. 이때는 요청 샘플링이 아니라 청구 금액 기준의 집계만 전수로 하고, 세분도가 fine한 분석은 샘플링으로 돌리는

것이 현실적이다.

2.4 Hard Cap vs Soft Warning: 토큰 예산제의 구현

비용 목표를 정했으면 그 목표를 강제할 방법이 필요하다. 두 가지 접근이 있다.

Hard Cap은 소비가 한계에 도달하면 더 이상 요청을 허용하지 않는다. 사용자가 할당량을 다 쓰면 "죄송합니다. 이번 달 한도에 도달했습니다"라는 메시지를 반환한다. 구현은 간단하다. 기간별 누적 토큰 수가 임계값을 넘으면 요청을 거절한다.

Hard Cap의 문제점은 완벽한 예측이 어렵다는 데 있다. 월초에 예상치 못한 많은 요청이 오면 중순에 한도가 닿을 수 있다. 그렇게 되면 월말까지 서비스가 정지한다. 이를 피하려고 너무 큰 한도를 설정하면 비용 통제 효과가 약해진다.

Soft Warning은 임계값에 도달해도 요청을 계속 받되 경고를 보낸다. 팀이 Slack 알림을 받고 대응한다. 사용자에게는 "이번 달 사용량이 평균의 150%입니다"라는 안내가 간다. 이 접근은 서비스 정지를 피하지만, 경고를 무시하면 비용이 한계 없이 올라간다.

대부분의 프로덕션 시스템에서는 Soft Warning을 기본으로 하되, 특정 기능에만 Hard Cap을 두는 것이 현실적이다. 중요도가 낮은 기능에 Hard Cap을 걸어두면 비용이 급등하는 것을 방지할 수 있다.

구현 시 주의할 점은 비용이 발생하고 나서 검출까지의 시간 차이다. 토큰 사용량은 요청이 완료된 후에야 정확히 알 수 있다. 따라서 임계값 검사 시점과 검사 기준의 세분도가 맞아야 한다. 예를 들어 "1시간에 10만 토큰"이라는 제한이 있다면, 매 분마다 직전 1시간 누적량을 계산하는 방식이 필요하다.

2.5 정리하며

토큰을 세는 일은 단순해 보이지만 실제 청구 금액과의 차이, 3단위 세분도, 예산 enforcement 구현처럼 놓치기 쉬운 함정이 많다. SDK 수치에 billing 기반 보정 계수를 두고, 요청-세션-사용자 3단위로 집계하는 구조를 만들고, Hard Cap과 Soft Warning을 기능별로 조합한다. 이 세 가지가 프로덕션 계측의 핵심이다. 다음 장에서는 이렇게 측정한 비용으로 모델을 어떻게 선택하고 라우팅할지 다룬다.

3 모델 라우팅: 같은 문제를 싸게 푸는 기술

Figure 3.1: 모델 라우팅: 같은 문제를 싸게 푸는 기술

3.1 모델 계층 설계: 세 개의 레이어

하나의 모델만 쓰는 시스템에서 여러 모델을 조합하는 시스템으로 넘어가는 첫걸음은 모델 계층을 설계하는 것이다. 계층이 없으면 라우팅 결정이 혼란스러워지고, 비용 관리도 불가능하다.

주력 모델. 가장 비싼 모델로, 최종 결과물을 생성하는 역할을 맡는다. 복잡한 추론, 창의적 작업, 높은 품질이 요구되는 응답 생성에 쓴다. 호출 빈도가 가장 낮고 단가가 가장 높다.

라우팅 모델. 요청의 난이도를 먼저 파악해서 주력 모델에 보낼지, 더 싸고 빠른 라우팅 모델에 보낼지를 결정하는 모델이다. 전체 호출의 많은 부분을 받는다. 역할이 "분류"이므로 주력 모델보다 저렴한 모델이 적합하다. 핵심 능력은 "무엇이 복잡하고 무엇이 단순한지 구분하는 것"이다.

경량 모델. 단순한 요청, 반복적인 작업, 사실 조회 등에 쓴다. 호출 빈도가 가장 높고 단가가 가장 낮다. 주력 모델의 비용을 아껴주는 버퍼 역할이다.

이 계층의 핵심 원리는 "비용이 비싼 모델의 소비를 줄이는 것"이다. 주력 모델을 직접 호출하는 대신 라우팅 모델이 먼저 통과시켜서 불필요한 주력 모델 호출을 걸러낸다. 실제로는 대부분의 요청이 경량 모델에서 처리되고, 라우팅 모델이 걸러낸 일부만 주력 모델에 도달한다.

3.2 의사결정 라우팅 vs 품질 라우팅

라우팅을 적용할 때 선택지는 크게 두 가지다. 어느 것을 선택하느냐에 따라 구현 방법과 트레이드오프가 달라진다.

의사결정 라우팅은 "이 요청이 복잡한 판단을 요구하는가?"를 묻는다. 복잡하면 주력 모델로, 단순하면 경량 모델로 보낸다. 복잡도의 판단 기준은 주관적이다. 정해진 규칙이 없어도 될 질문들이 여기 속한다. "이 이메일의 감정을 분류해줘"는 대부분의 경우 경량 모델로 충분하다.

"이 이메일에 대한 답장을 작성하는데, 받는 사람의 지위를 고려해 톤을 조절해줘"는 판단 요소가 많아 주력 모델이 적합하다.

의사결정 라우팅의 구현은 비교적 단순하다. 요청의 성격을 파악하는 규칙을 만들고, 그 규칙에 매칭되면 주력 모델로 보내는 방식이다. 예를 들어 요청 길이가 특정 기준 이상이고, 키워드 집합이 포함되어 있으면 주력 모델이라는 조건식을 만든다.

품질 라우팅은 "얼마만큼의 품질이 필요한가?"를 묻는다. 높은 품질이 필요하면 주력 모델을, 낮으면 경량 모델을 선택한다. 품질 기준은 비즈니스 요구에 의해 정해진다. 예를 들어 고객에게 보내는 응답은 더 높은 품질이 필요하므로 주력 모델, 내부 요약은 경량 모델로 처리할 수 있다.

품질 라우팅에서는 품질 기준을 정의하는 일이 가장 어렵다. "이 응답이 고객에게 적합한 품질인가"를 판단하는 기준이 주관적이기 때문이다. 그래서 품질 라우팅을 성공시킨 팀은 대체로 품질 기준을 명시적으로 문서화해둔다.

3.3 정확도-비용 트레이드오프 곡선

라우팅의 효과를 정당화하려면 "얼마를 절약하면서 품질 저하는 어느 정도인가"를 측정해야 한다. 이 곡선은 라우팅을 도입하기 전에 반드시 그려봐야 한다.

트레이드오프 곡선은 X축에 비용, Y축에 품질 점수를 둔다. 각 모델 계층마다 하나씩 점을 찍으면 세 개의 점이 만들어진다. 경량 모델은 저비용 저품질, 라우팅 모델은 중간 비용 중간 품질, 주력 모델은 고비용 고품질이다.

이 세 점이 직선으로 연결되지 않는 경우가 많다. 실제로는 경량 모델의 품질이 기대보다 높은 구간이 있고, 주력 모델의 추가 비용 대비 품질 향상이 미미한 구간이 있다. 곡선을 그려보면 "중간에 있는 모델이 둘 다 애매한

위치"에 있다는 걸 발견할 수 있다.

품질 요구가 높은 요청은 주력 모델로 보내고 낮은 요청은 경량 모델로 보내면 중간 모델을 통째로 제거할 수 있다는 것이 라우팅의 핵심 통찰이다. 이 곡선에서 "중간 모델" 위치의 모델이 비용만 올리고 품질은 기대만큼 올리지 않는다면, 그 모델을 라우팅 대상에서 제외하는 것이 낫다.

실전에서는 이 곡선을 만들기 위해 과거 요청 데이터를 분석한다. 과거 요청을 주력 모델과 경량 모델 양쪽에서 처리하고, 응답 품질을 비교한다. 차이가 미미한 요청의 비율이 곧 라우팅으로 절약 가능한 비용 비율이다.

3.4 라우팅 엔진 구현: 3가지 접근

라우팅 엔진을 만드는 방법은 크게 세 가지가 있다. 각자의 트레이드오프가 명확하므로 상황에 맞게 선택한다.

규칙 기반 결정 트리. If-else로 요청 속성에 따라 분기한다. 요청 길이, 포함된 키워드, 요청 유형 등을 조건으로 쓴다. 구현이 가장 단순하고 예측 가능성이 높다. 그러나 커버리지에 한계가 있다. 정해진 규칙으로 분류되지 않는 요청이 일정 비율 생긴다.

구현 예시. 요청이 "분류" 또는 "요약"이고 길이가 500토큰 미만이면 경량 모델. "분석" 또는 "작성"이 포함되고 길이가 1,000토큰 이상이면 주력 모델. 나머지는 라우팅 모델로 먼저 보내서 판단을 위임한다.

분류기 기반. 과거 요청 데이터를 라벨링해서 분류기를 훈련한다. "복잡도"를 정답으로 두고, 요청의 텍스트 특성으로 복잡도를 예측하는 모델이다. 규칙보다 커버리지가 넓고 분류도 정확하다. 그러나 훈련 데이터가 필요하고, 분류기 자체의 유지보수 비용이 발생한다.

분류기 기반 라우팅은 규칙으로 설명하기 어려운 "복잡도의 뉘앙스"를 포착하는 데 효과적이다. 예를 들어 "사랑이 무엇인가에 대해 철학적으로

탐구해줘"와 "사랑의 영어 단어를 찾아줘"는 둘 다 "사랑"이라는 단어를 포함하지만 복잡도는 확연히 다르다. 분류기는 이런 뉘앙스를 학습할 수 있다.

임베딩 기반 유사도. 요청의 임베딩 벡터들 사이의 거리를 계산해서 유사한 과거 요청의 처리를 참고한다. 과거에 주력 모델로 처리했던 요청과 현재 요청의 거리가 가깝다면 주력 모델로 보내는 방식이다. 라벨 데이터가 필요 없다는 장점이 있다.

임베딩 기반 접근의 핵심 파라미터는 임계값이다. 과거 주력 모델 처리 이력과 현재 요청의 거리가 임계값 이내면 주력 모델로 보낸다. 임계값이 높으면 더 많은 요청이 주력 모델로 향해 비용이 올라가고, 낮으면 경량 모델로 향해 품질이 저하될 수 있다.

3.5 라우팅의 부작용: 미처리 에지와 품질 불안정

라우팅을 도입할 때 발생하는 부작용 두 가지를 미리 알아둬야 한다.

미처리 에지. 라우팅 엔진이 요청의 복잡도를 "중간"으로 판단해 라우팅 모델로 보냈는데, 정작 라우팅 모델이 잘못된 응답을 생성하는 경우가 있다. 이는 라우팅 판단이 틀린 것인데, 사용자는 그 결과를 주력 모델에서 받은 것으로 인식한다. 결과적으로 품질 저하를 체감하게 된다.

이 부작용을 완화하려면 라우팅 모델의 에러율을 모니터링해서 에러가 특정 유형의 요청에 집중되는 패턴이 있는지 확인하고, "에스컬레이션 경로"를 둬야 한다. 라우팅 모델의 응답에 사용자가 "더 좋은 응답"을 요청하면 주력 모델로 올리는 흐름이다.

품질 불안정. 경량 모델의 품질이 항상 일정하지는 않다. 같은 요청을 두 번 보내도 다른 응답이 나올 수 있다. 특히 출력이 비결정적(non-deterministic)인 모델에서는 품질 차이가 두드러진다. 경량 모델을 쓰는 기능에서 결과의 일관성이 중요하다면 이것이 문제가 된다.

해결책은 간단하다. 경량 모델이라도 온도를 낮추거나 결정적 모드가 있는 모델을 고르면 된다. 응답의 일관성이 중요한 기능에는 경량 모델의 응답을 캐시해서 동일한 요청에 대해서는 동일한 응답을 주는 방식도 고려할 수 있다.

3.6 정리하며

모델 라우팅은 세 개의 모델 계층 설계에서 시작한다. 주력 모델, 라우팅 모델, 경량 모델의 역할을 명확히 하면 비용 흐름이 보이기 시작한다. 의사결정 라우팅과 품질 라우팅 중 무엇을 선택할지는 품질 요구의 명확도에 따라 다르다. 라우팅 엔진은 규칙, 분류기, 임베딩 세 가지 접근 중 상황에 맞는 것을 선택한다. 부작용으로 미처리 에지와 품질 불안정을 미리 설계에 반영해야 한다. 다음 장에서는 캐시로 반복 비용을 구조적으로 줄이는 패턴을 다룬다.

4 캐시 설계: 반복 비용을 구조적으로 줄이는 패턴

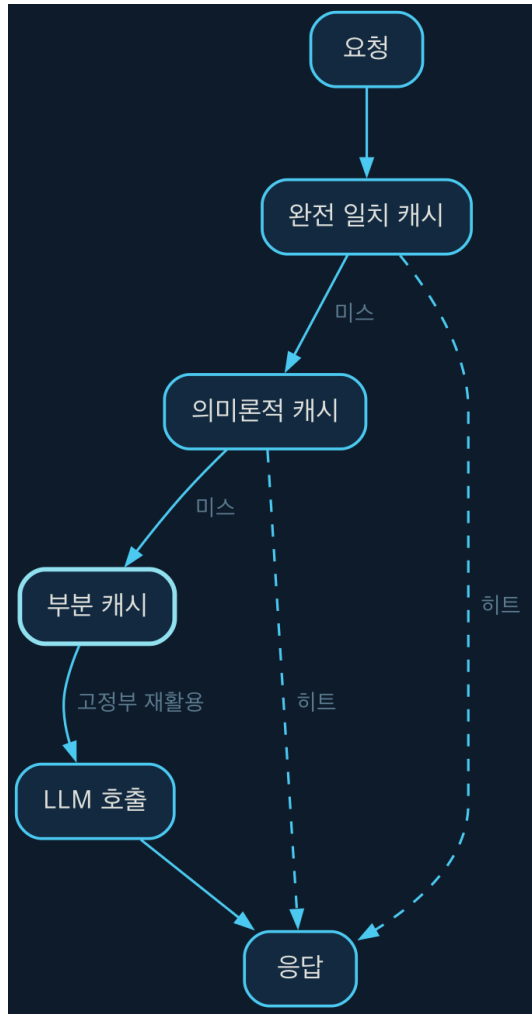

Figure 4.1: 캐시 설계: 반복 비용을 구조적으로 줄이는 패턴

4.1 완전 일치 캐시: 요청 해시의 힘

가장 단순하고 효과적인 캐시 전략은 요청의 해시값을 키로 쓰는 완전 일치 캐시다. 동일한 입력이면 출력도 항상 같을 거라 기대할 수 있을 때 쓴다. 이 캐시의 핵심 전제는 단순하다. 같은 질문에는 같은 답변이 옳다는 것이다.

구현 과정은 네 단계로 나뉜다. 먼저 요청의 입력에서 토큰화된 텍스트를 추출하고, 그 텍스트의 해시를 계산한다. 이 해시를 키로 캐시 저장소를 조회하는데, 히트면 저장된 응답을 그대로 반환하고 미스면 LLM을 호출해 결과를 얻은 뒤 캐시에 저장한다.

이 전략이 강력한 이유는 검사 로직이 없어서다. "이 요청이 캐시 가능한가"를 판단할 필요가 없다. 요청의 텍스트만으로 키가 결정되고, 캐시 여부는 조회 결과가 자연스럽게 정한다.

적용하기 좋은 예시는 문서 요약이다. 같은 문서에 같은 요약을 요청하면 매번 비용을 내지 않아도 된다. FAQ 응답 생성이나 분류 작업, 번역도 동일 입력에 동일 출력이 유효하므로 잘 맞는다.

다만 응답의 시간 민감성은 주의해야 한다. "현재 환율"이나 "오늘의 뉴스"처럼 시간이 지나면 내용이 변하는 요청에 완전 일치 캐시를 쓰면 오래된 응답을 그대로 줄 위험이 있다. 그래서 TTL(Time To Live)을 설정해 일정 시간이 지나면 캐시를 무효화하는 방식이 흔히 쓰인다.

4.2 의미론적 캐시: "거의 같은" 질의의 재활용

완전 일치 캐시의 한계는 "거의 동일한" 요청을 같은 요청으로 인식하지 못한다는 점이다. "서울 날씨 알려줘"와 "서울의 날씨가 어떻게 돼?"는 의미가 같지만 텍스트가 다르므로 완전 일치 캐시로는 잡히지 않는다.

의미론적 캐시는 이 간극을 메운다. 요청을 임베딩 모델로 벡터화한 뒤 벡터들 사이의 코사인 유사도를 계산해서, 일정 유사도를 넘으면 캐시

히트로 간주한다. 텍스트가 다르더라도 의미가 비슷한 요청을 같은 응답으로 처리할 수 있는 이유다.

구현의 핵심 파라미터는 임계값이다. 임계값이 높으면 완전 일치에 가까운 엄격한 매칭이 되고, 낮으면 조금 다른 요청까지 캐시 히트로 인정한다. 임계값을 높이면 품질은 올라가지만 캐시 히트율이 떨어지고, 낮추면 히트율은 오르는 대신 의미적으로 다른 응답을 잘못 재활용하는 위험이 커진다.

임베딩 모델 선택이 결과를 좌우한다. 요청의 의미를 잘 포착하는 모델이 필요하다. 코사인 유사도 비교는 벡터 차원이 클수록 비용이 늘어나므로, 고차원 임베딩을 쓰는 시스템에서는 PCA나 해싱 트릭으로 차원을 줄이는 방법도 고려할 만하다.

적용할 때는 "비슷한 질문"과 "같은 질문"의 차이를 인식해야 한다. 사용자가 처음엔 "서울 날씨", 다음엔 "부산 날씨"라고 물으면 의미가 다르므로 다른 응답이 필요하다. 이런 부분적 유사성을 잘 걸러내려면, 응답을 생성할 때 요청의 핵심 의도를 파악해 캐시 키에 포함시키는 방법이 효과적이다.

4.3 부분 캐시: 컨텍스트를 쪼개서 재활용하는 기술

복잡한 LLM 애플리케이션에서는 요청의 일부만 캐시 가능하고 나머지는 매번 달라진다. 시스템 프롬프트, Few-shot 예제, 사용자 입력, 이 세 부분으로 요청이 구성되는 경우가 대표적이다. 전체를 완전 일치로 캐싱하면 히트율이 너무 낮아지지만, 부분을 나누어 캐싱하면 효율이 달라진다.

시스템 프롬프트 캐시. 애플리케이션의 지시사항이나 역할 정의처럼 프롬프트의 고정된 부분은 매번 같은 텍스트가 전송된다. 이 부분만 따로 캐시해두면 입력 토큰 수를 줄일 수 있다. 다만 모델 제공자가 이미 내부적으로 비슷한 최적화를 지원하는 경우가 있으니, 적용하기 전에 현재

SDK나 API가 캐시를 어떻게 지원하는지부터 확인하는 편이 좋다.

Few-shot 예제 캐시. 작업별로 예제 몇 개를 프롬프트에 넣는 Few-shot 방식에서는 예제 선택만 다르고 예제 자체는 같은 경우가 많다. 예제 세트를 따로 캐시해두면 예제 선택 로직만 독립적으로 동작하게 만들 수 있다. 예제 세트가 자주 바뀌지 않는다면 효과가 크다.

사용자 입력 분리 캐시. 사용자 입력이 길고 복잡하다면, 입력을 구조화해서 필요한 부분만 LLM에 전달하고 나머지는 참조로만 처리할 수 있다. 입력을 파싱해 구조화된 형태로 캐시해두면, 바뀌지 않은 부분에 대해서는 재파싱 비용도 아낄 수 있다.

부분 캐시의 핵심은 무엇이 고정되고 무엇이 변하는지를 파악하는 데 있다. 이 구분이 명확할수록 부분 캐시의 효과도 커진다. 대부분의 애플리케이션에서는 시스템 프롬프트와 Few-shot 예제는 고정되고 사용자 입력만 변하는 패턴이 흔하다.

4.4 캐시 hit rate 80%에서 더 올리기 어려운 이유

캐시를 도입한 초반에는 hit rate가 빠르게 오른다. 하지만 어느 지점을 넘어서면 상승 속도가 눈에 띄게 느려진다. 이 현상의 이유를 알면 다음 단계의 개선 방향도 보인다.

사용자 쿼리의 자연적 다양성. 사용자가 보내는 질문은 생각보다 반복되지 않는다. 완전 일치 캐시는 쿼리가 정확히 같을 때만 잡히므로, 같은 의도라도 다른 표현으로 오면 미스가 난다. 의미론적 캐시가 이를 어느 정도 해소하지만, 유사도 임계값을 높이면 미스가 늘고 낮추면 품질 문제가 생긴다. 이 트레이드오프가 히트율의 상한을 만든다.

시간에 따른 쿼리 분포 변화. 사용자 요구나 트렌드, 계절 같은 요인이 바뀌면 쿼리 분포도 함께 바뀐다. 어제 인기 있던 쿼리가 오늘은 잘 안 보이기도 한다. 캐시 히트율의 분자는 그대로인데 분모인 전체 쿼리 수가

변하니, 상대적 히트율이 줄어드는 것처럼 보인다.

제품 사용 범위 확대. 기능이 늘어나면 이전에는 캐시되지 않던 새로운 유형의 쿼리가 들어온다. 이 쿼리들의 초기 캐시 미스가 전체 분모를 키우면서 히트율을 희석시킨다.

80% 이후 히트율을 더 올리는 방법은 크게 두 갈래다. 하나는 안 되는 부분은 그대로 인정하고 다른 차원의 최적화를 보는 것이다. 캐시 미스가 났을 때의 비용을 줄이는 방법, 즉 응답 생성을 더 빠르고 싸게 만들 방법을 찾는 식이다. 다른 하나는 히트율 자체가 아니라 히트율과 평균 비용의 곱을 최적화 대상으로 삼는 것이다. 캐시로 절감하는 비용은 히트율에 평균 응답 비용을 곱한 값이므로, 히트율이 낮더라도 히트 대상의 평균 비용이 높으면 절감액은 오히려 커진다. 결국 전체 비용 절감이라는 목표 아래에서 히트율과 평균 비용의 곱을 최적화하면 된다.

4.5 정리하며

캐시는 반복 비용을 줄이는 가장 강력한 도구다. 완전 일치 캐시로 정확히 같은 요청을 잡고, 의미론적 캐시로 거의 같은 질의를 재활용한다. 부분 캐시로 컨텍스트를 쪼개 고정된 부분을 재활용하면 효율이 한 단계 더 올라간다. 히트율이 80%를 넘어서면 더 올리기 어려운 이유는 사용자 쿼리의 자연적 다양성과 시간에 따른 분포 변화 때문이다. 다음 단계는 캐시 미스 시의 비용을 줄이거나, 히트율과 평균 비용의 곱을 최적화하는 쪽으로 방향을 트는 것이다.