

한국어판 · 실무자를 위한 설계·운영 가이드

AI 게이트웨이 플레이북

인증·쿼터·비용 통제·MCP를 갖춘
프로덕션 LLM 게이트웨이 구축

키리스 인증·RBAC

토큰 쿼터·예산

MCP 서버

RAG 어시스턴트

운영 런북

8개 장 · 다이어그램 · 40개 항목 체크리스트

Sanghyeok Y (zerokdevops)

AI 게이트웨이 플레이북

인증·쿼터·비용 통제·MCP를 갖춘 프로덕션 LLM 게이트웨이 구축

한국어판 안내 이 책은 영문판 *The AI Gateway Playbook*의 한국어판이다. 원서는 저자가 사내 LLM 게이트웨이를 설계하고 운영한 경험을 바탕으로 AI 도구의 도움을 받아 집필했고, 한국어판은 원서를 AI 도구로 번역한 뒤 용어를 통일해 편집했다. 책에 나오는 조직, 설정, 수치는 모두 일반화된 예시다.

이 책의 핵심 메시지

LLM을 진지하게 도입하는 회사는 결국 모두 같은 벽에 부딪힌다. 개발자 A는 프런티어 모델 API 키를 노트북에 하드코딩한다. 개발자 B는 스프린트 도중 첫 번째 키가 레이트 리밋에 걸리자 두 번째 키를 새로 발급받는다. 몇 달 뒤 플랫폼 팀은 청구 금액이 이를 책임지는 팀의 규모를 훌쩍 넘어섰다는 사실을, 그리고 "누가 어떤 모델에 어떤 기능을 위해 얼마를 쓰고 있는가"에 아무도 답할 수 없다는 사실을, 나아가 회사의 AI 스택 전체가 지금의 모델 제공사 대부분이 존재하기도 전에 설계된 벤더 추상화 위에 서 있다는 사실을 발견한다.

이 책은 이 문제를 해결하는 시스템, 즉 **사내 LLM 게이트웨이**를 만드는 플레이북이다. 사내 LLM 게이트웨이란 코드베이스와 모든 모델 제공사 사이에 자리 잡은, 얇고 평범하며 감사 가능한 인프라 계층으로, 모델 접근을 성숙한 플랫폼이 시크릿·예산·접근 제어를 다루듯 취급하는 것을 말한다.

이 책을 통해 다음을 구축하는 방법을 배운다.

- 여러 제공사에 걸쳐 모델, 가용성, 가격을 자동으로 동기화하는 **모델 레지스트리**. 이를 통해 코드베이스는 벤더의 모델 ID가 아니라 안정적인 별칭(alias)과 통신한다.
- 채팅, 스트리밍, 임베딩, 톨 호출을 위한 단일 와이어 프로토콜과, 롱테일 기능을 위한 네이티브 패스스루를 갖춘 **통합 API 표면**.
- 사람에게 정적 LLM API 키가 절대 발급되지 않는 **키리스(keyless) 인증 계층**. 자격 증명은 관리형 ID와 워크로드 브로커 안에 존재하며, 접근 권한은 역할별로 부여되고, OBO(대리 호출) 플로우가 모든 토큰을 실제 사용자에게 귀속시킨다.
- LLM 접근에 맞춰 조정된 **6단계 RBAC 모델**. 역할별 요청 레이트 리밋과 일일 토큰 쿼터를 포함한다.
- 모델별·테넌트별·사용자별로 모든 토큰을 계산하는 **미터링 및 비용 파이프라인**. 예산, 소프트·하드 한도, 대시보드, 채널 알림, 쿼터 증설 승인 워크플로, 제공사 인보이스 대비 월별 정산 대사를 포함한다.
- 내부 SaaS 및 사무용 도구를 90개 이상 규모로 Model Context Protocol 톨로 노출하는 **MCP 서버**. OAuth 2.1 + PKCE, 봉투 암호화된 시크릿, PII 마스킹, 셀프서비스 웹 포털로 보호된다.
- 오픈스트레이터와 엔터프라이즈 LLM, MCP, 검색(RAG)을 결합한 **RAG 기반 팀 어시스턴트**. CI/CD 장애 분석이라는 완결된 엔드투엔드 예시와, 이를 정직하게 유지하는 평가 하네스를 함께 다룬다.
- 그리고 이 모든 것을 지속 가능하게 만드는 **운영 계층**. 감사 로그 보존 기간, 정적 키 없는 키 로테이션, 관측성, 장애 런북, 비용 비교 부록, 배포 및 마이그레이션 체크리스트를 포함한다.

이 책은 실무자를 위한 책이다. 각 장은 함정(pitfall) 절로 끝나는데, 이는 현장에서 실제로 일어나는 실수들을 다룬다. 모든 설계에는 다이어그램(Mermaid, ASCII 대체본 포함)과 설정 또는 코드 스케치, 그리고 실제로 선택이 갈리는 의사결정 표가 함께 제공된다.

이 책의 대상 독자

- 서비스 메시, API 게이트웨이, 또는 개발자 플랫폼을 책임지는 시니어 인프라-플랫폼 엔지니어로서 LLM 트래픽을 일급 시민으로 다뤄야 하는 요청을 받은 사람.
- AI 플랫폼 / ML 플랫폼 엔지니어로서 CFO가 분기별 질문을 던지기 전에 모델 접근을 안전하고, 미터링되고, 귀속 가능하게 만들어야 하는 사람.
- 이 시스템을 실제로 운영하게 될 데브옵스와 SRE. SLO, 알림 설계, 런북, 마이그레이션 체크리스트를 얻을 수 있다.
- 직접 구축 vs 구매를 결정하고 이를 방어해야 하는 테크 리드와 엔지니어링 매니저.

다음 사항에 이미 익숙해야 한다. 프로토콜 수준의 HTTP와 gRPC, 서비스 메시 또는 리버스 프록시, 주요 클라우드 하나(어떤 주요 클라우드든 상관없다. 이 책은 의도적으로 클라우드에 중립적이다), 관리형 ID와 키리스 서비스 자격 증명을 갖춘 ID 플랫폼, 기본적인 비용 회계. 프로덕션 LLM 배포를 운영해본 경험이 꼭 필요하지는 않다. 만약 경험이 있다면, 여러 함정 절이 당신이 바빠서 적어두지 못했던 경고처럼 읽힐 것이다.

이 책이 아닌 것

- 모델옵스 가이드가 아니다. 프롬프트 엔지니어링, 파인튜닝, 또는 6장의 어시스턴트에 필요한 실용적인 하네스를 넘어서는 평가 이론은 다루지 않는다.
- 제품 매뉴얼이 아니다. 벤더 이름은 용어상 실질적으로 유용한 경우에만 등장한다(예: "클라우드의 엔터프라이즈 LLM 오퍼링", "Entra ID 키리스", "M365 계열 도구", "Atlassian 계열 도구"). 그 외에는 모두 역할로 서술되므로, 책을 다시 읽지 않고도 자신의 스택으로 치환할 수 있다.
- 특정 회사의 사례 연구가 아니다. 모든 수치는 예시로 표시되어 있다. 여기 담긴 패턴은 일반적인 엔지니어링 관행이며, 당신의 수치는 당신의 수치가 될 것이다.

책의 구성

장	제목	얻을 수 있는 것
01	기업이 관리형 API 게이트웨이를 LLM 트래픽에서 벗어나는 이유	비용·락아웃·통제 논증, 직접 구축 vs 구매 의사결정 표, 그리고 TCO(총소유비용) 프레임: 커스텀 게이트웨이가 관리형 프리미엄 게이트웨이 비용을 약 90% 절감할 수 있는 이유.
02	레퍼런스 아키텍처	전체 아키텍처: 다중 제공사 자동 동기화를 갖춘 모델 레지스트리, 통합 API 표면, 라우팅/한도 계층, 인증, 미터링/비용 파이프라인, 감사, 관측성. 90일 구현 계획 포함.
03	키리스 인증과 6단계 RBAC 모델	정적 LLM 키를 절대 발급하지 않는다: 관리형 ID, OBO 플로우, 6개 역할, 역할별 RPS, 일일 토큰 쿼터, 승인 워크플로 연동.

장	제목	얻을 수 있는 것
04	쿼터, 예산, 비용 통제	모델/테넌트/사용자별 토큰·비용 회계, 예산과 소프트/하드 한도, 실시간 대시보드, 채널 및 웹훅 알림, 쿼터 증설 승인, 월별 정산 대사.
05	엔터프라이즈 규모의 MCP	내부 SaaS 및 사무용 도구 위에 구축한 MCP 서버(90개 이상의 톨), OAuth 2.1 + PKCE 인증, 저장 시크릿을 위한 AES-256-GCM 봉투 암호화, PII 마스킹 파이프라인, 셀프서비스 웹 포털, 톨 호출 감사.
06	RAG 기반 팀 어시스턴트	오케스트레이터 + 엔터프라이즈 LLM + MCP + RAG, CI/CD 장애 분석 어시스턴트 완결 예시, 가드레일, 평가 하네스.
07	운영	감사 로그 기록과 보존 기간, 정적 키 없는 키 로테이션, 관측성 스택, 장애 런북, 비용 비교 부록, 배포 체크리스트, 마이그레이션 체크리스트.
08	프로덕션 준비	전체 체크리스트와 흔한 함정들, 각 항목마다 왜 문제가 되는지 주석으로 설명.
09	찾아보기	참고용 한 줄 장별 지도.

1장과 2장을 먼저 읽는다. 3장부터 6장까지는 대체로 독립적이어서 어느 순서로 읽어도 되지만, 2장에서 나온 아키텍처 용어를 공유한다. 7장은 그 앞의 모든 것에 대한 운영 계약이며, 8장은 시스템이 진짜라고 사람들에게 말하기 전에 실행해보는 것이다.

표기 규칙

- **코드와 설정:** 설정 스니펫은 YAML 또는 JSON이다. 구현 스케치는 Go 스타일 의사코드로, 실제 코드의 형태를 충분히 보여줄 만큼 구체적이되 의도적으로 단순화한 부분은 주석으로 표시한다. 스니펫이 의사코드인 경우 주석으로 그렇다고 밝힌다.
- **다이어그램:** Mermaid 코드 블록은 대부분의 마크다운 도구에서 렌더링된다. 중요한 다이어그램마다 일반 코드 블록 형태의 ASCII 대체본을 함께 제공한다.
- **금액과 물량:** 가격이나 물량처럼 보이는 모든 숫자는 *예시*이며 그렇게 표시되어 있다. 1장의 TCO 모델은 의사결정에 사용하기 전에 반드시 자신의 숫자로 다시 세워봐야 한다.
- **클라우드 용어:** "클라우드"는 사용자의 클라우드 제공사를 뜻한다. "ID 플랫폼"은 클라우드의 ID 서비스(관리형 ID, 키리스 자격 증명, 워크로드 ID 페더레이션)를 뜻한다. "엔터프라이즈 LLM 오퍼링"은 클라우드의 호스팅형 모델 서비스를 뜻한다. "시크릿 스토어"는 KMS 기반의 시크릿 매니저를 뜻한다.
- **시점 표기:** 이 책은 2026년 기술 기준선을 가정한다. Model Context Protocol(MCP)이 표준 톨 프로토콜이고, PKCE를 적용한 OAuth 2.1이 인증 기준선이며, 추론 모델이 정상적인 비용 범주이고, 제공사 가격 체계에 캐시된 토큰과 추론 토큰 항목이 포함되어 있는 시점이다.

기밀성에 관한 안내

이 책에 담긴 내용은 모두 독창적으로 작성되었다. 예시들은 가상이며 일반화되어 있다. 팀원 9명 규모의 "대기업", `standard` 나 `reasoning` 같은 별칭 이름, 그리고 특정 실제 배포가 아니라 패턴을 설명하는 설정 스니펫들이 그렇다. 만약 어떤 패턴이 당신의 스택과 비슷해 보인다면, 바로 그것이 의도한 바다. 패턴은 일반적이며, 구체적인 세부사항은 당신만의 것으로 남는다.

시작하기 전에 한 가지 더

이 책에서 가장 중요한 한 문장은 1장에 있다. **당신은 게이트웨이를 사는 것이 아니라 비용과 통제의 경계를 구축하는 것이다.** 게이트웨이 자체는 쉬운 부분이다. 어려운 부분은 회계, 접근 모델, 그리고 모든 토큰을 귀속 가능하게 만드는 규율이다. 2장의 아키텍처는 이 세 가지를 평범하게 만들기 위해 존재한다. 이 책의 나머지 모든 내용은 이 한 문장을 위해 존재한다.

이제 왜(why)부터 시작해보자.

1장. 기업이 관리형 API 게이트웨이를 LLM 트래픽에서 벗어나는 이유

"당신은 게이트웨이를 사는 것이 아니다. 당신은 비용과 통제의 경계를 구축하는 것이다."

이 장은 그 논거를 제시한다. 누군가 "이미 관리형 API 게이트웨이가 있는데 LLM 트래픽을 위해 왜 또 다른 게이트웨이가 필요한가"라고 묻는 순간을 위해 쓰여졌다. 정직한 답은 이렇다. LLM 트래픽은 관리형 게이트웨이가 세워진 세 가지 전제를 깨뜨리기 때문이다. 요청이 비용의 단위라는 전제, 벤더 추상화 자체가 제품이라는 전제, 그리고 게이트웨이는 중립적인 통로일 뿐이라는 전제다.

1.1 LLM 트래픽은 API 트래픽이 아니다

관리형 API 게이트웨이는 웹 서비스 시대에 설계되었다. 그 비용 모델, 미터링, 보안 모델은 모두 **요청이 원자적 단위**라는 전제를 깔고 있다. LLM 워크로드는 이 전제를 다섯 가지 구체적인 방식으로 깨뜨린다.

비용은 호출 횟수가 아니라 토큰의 함수다. REST 엔드포인트에 보내는 요청은 본문이 10바이트든 100킬로바이트든 같은 비용이 든다. 모델에 보내는 요청의 비용은 입력 토큰, 출력 토큰, 캐시된 입력 토큰, 추론 토큰, 그리고 일부 모달리티의 경우 이미지나 오디오 단위의 함수다. HTTP 형태가 완전히 동일한 두 요청의 비용이 10,000배까지 차이 날 수 있다. 요청 단위로 미터링하는 시스템은 구조적으로 정말 중요한 질문, 즉 "이 기능이, 이 팀이, 이 사용자가 이번 달에 얼마를 썼는가"에 답할 수 없다.

스트림은 수명이 길다. 스트리밍 응답은 서버센트이벤트(SSE) 연결로, 몇 분씩 열려 있을 수 있다. 관리형 게이트웨이는 전통적으로 요청 단위로 과금하고 버퍼링한다. 관리형 계층을 거치는 장시간 SSE는 연결당 오버헤드를 추가하며, 많은 관리형 계층에서 이 연결은 300초짜리 쓰기 작업에 맞게 설계되지 않은 공유 멀티테넌트 엣지 뒤에 놓인다. 이는 테일 레이턴시로 체감되며, 일부 플랫폼에서는 추가 이그레스 비용이나 프리미엄 티어 추가 요금으로도 나타난다.

재시도는 비싸고 먹등적이지 않다. REST 호출을 재시도하면 원래 호출과 같은 비용이 든다. 모델 호출을 재시도하면 *다시 비용이* 청구된다. 그리고 부분적으로 스트리밍되다가 버려진 응답도 이미 출력 토큰 대부분에 대한 비용이 청구된 상태다. 따라서 재시도, 페일오버, 제공사 장애는 가용성 이벤트일 뿐 아니라 *비용 이벤트*이기도 하다. 게이트웨이는 이를 회계에 반영해야 하며, 두 번의 시도와 제공사의 사용량 객체를 모두 볼 수 있는 시스템만이 이를 해낼 수 있다.

프롬프트는 페이로드이고, 페이로드는 데이터다. 모델 요청은 일상적으로 소스 코드, 고객 데이터, 또는 내부 문서를 요청 본문에 담아 보낸다. 요청 본문을 로깅하거나, "인사이드"를 위해 샘플링하거나, 공유 엣지로 라우팅하는 관리형 게이트웨이는 많은 컴플라이언스 상황에서 아예 그 트래픽을 건드려서는 안 되는 존재다. 스트림별로 정확히 어떤 구성 요소가 어떤 바이트를 보았는지 알아야 한다.

모델은 움직이는 타겟이다. 12개월이라는 기간 안에도 모델 ID가 바뀌고, 모델이 단종되고, 가격이 바뀌고, 다른 제공사의 "동일한" 모델이 다른 한도, 다른 토큰 호출 동작, 다른 캐시 시맨틱을 갖게 된다. 모델을 백엔드의 URL 경로처럼 취급하는 게이트웨이는 이런 변화를 전혀 흡수할 수 없고, 그 변화는 고스란히 개발자들에게 떨어진다.

이 다섯 가지 속성 중 어느 것도 특이한 것이 아니다. 데모 단계를 지나 실제 청구서를 받기 시작한 조직이라면 누구나 겪는 정상적인 운영 조건이다.

1.2 관리형 API 게이트웨이가 설계된 목적

관리형 제품을 위해 공정하게 말하자면, 그것이 설계된 목적에는 탁월하게 부합한다.

- 요청 단위 인증(API 키, 엣지에서의 JWT 검증)
- 요청 단위 레이트 리밋과 쿼터
- WAF, IP 필터링, 지역 규칙, 인증서 종료
- 기본적인 사용량 미터링(주로 요청 수, 때로는 바이트 수)
- 라우트 단위 블루/그린 및 카나리 배포
- 관리형 인증서, 관리형 스케일링, 직접 운영할 서버 플릿이 없다는 장점

REST 영역, 즉 모바일 백엔드, 파트너 API, 내부 서비스 간 통신에는 훌륭한 도구다. 문제는 도구가 나빠서가 아니라 LLM 워크로드가 *그 도구의 설계 범위를 벗어나 있다*는 점이며, 관리형 플랫폼의 경제 구조는 바로 그 지점에서 당신을 벌한다.

- **요청 단위 과금**은 하필 프록시하기는 가장 싸고 제대로 미터링하기는 가장 비싼 트래픽에 적용된다.
- **이그레스 및 프리미엄 티어 추가 요금**은 하필 LLM 워크로드의 본질인 장시간·대용량 페이로드 트래픽에 붙는다.
- **토큰 단위 미터링이 없어서**, 게이트웨이가 존재하는 이유인 사용량 기반 통제 기능 자체를 제공하지 못한다.
- **벤더 모델 추상화**는 모델 시장보다 앞서 만들어졌거나 뒤쳐져 있어서, 전환이 곧 재작성이 된다.

1.3 비용 문제

LLM 트래픽을 얇은 커스텀 계층으로 옮기면, 관리형 프리미엄 스타일 API 게이트웨이 대비 게이트웨이 계층 비용이 대체로 약 90% 떨어지는 사례를 여러 팀에서 공통적으로 관찰한다. 왜 이 수치가 현실적인지, 그리고 왜 이것이 반올림 오차가 아닌지 살펴보자.

관리형 프리미엄 스타일 게이트웨이는 당신에게 **플랫폼**에 대한 비용을 청구한다. 멀티테넌트 엣지, 공유 VNet 통합, 요청당 수수료, GB당 이그레스, 프리미엄 티어 인스턴스 최소 요금, 그리고 결정적으로 트래픽이 1,000건이든 1,000,000건이든 상관없이 내야 하는 바닥값이다. 한계 비용은 요청당 거의 고정되어 있지만, 플랫폼에는 최소값이 있다. 트래픽이 적은 달에도 엣지, 콘솔, 인증서 플릿, 지원 티어에 대한 비용을 지불하는 셈이다.

커스텀 게이트웨이 계층은 **컴퓨트**에 대한 비용을 청구한다. 로드 밸런서 뒤에 있는 몇 개의 스테이트리스 프록시 인스턴스, Redis 크기의 상태 저장소, 그리고 메트릭 싱크다. 이 프록시는 한 가지 일, 즉 인증, 라우팅, 미터링, 스트리밍만 하며, 앞단에 멀티테넌트 엣지가 없기 때문에 종종 관리형 계층보다 **더 빠른** 경로로 처리한다. 얇은 프록시의 요청당 한계 비용은 관리형 요청당 수수료의 극히 일부에 불과하고, 플랫폼 바닥값도 없다. 트래픽이 0이면 비용도 거의 0이다.

예시로 든 월별 비교다. 모든 숫자는 형태를 보여주기 위해 만든 것일 뿐, 어떤 회사의 정가를 인용한 것이 아니다.

비용 구성 요소	관리형 APIM 프리미엄 스타일 티어	커스텀 씬 게이트웨이 계층
플랫폼 / 인스턴스 최소 요금	월 \$450(티어 최소값, 예시)	~\$0(컴퓨터만)
요청당 수수료(월 2천만 요청)	\$40(예시 100만 건당 \$2)	\$0(컴퓨터만)
대용량 SSE 페이로드 이그레스 (800GB)	\$96(예시 GB당 \$0.12)	VPC 내 포함 / 미미함
미터링, 토큰 회계	\$0 제공 — 어차피 직접 구축해야 함	내장(4장)
스트리밍 테일 레이턴시 비용	공유 엣지 특성상 종종 프리미엄 추가 요금	없음 — 제공사로 직행하는 경로
토큰 미터링 추가를 위한 엔지니어링	해당 없음 — 어차피 별도로 구축해야 함	게이트웨이 작업에 흡수됨
예시 합계(게이트웨이 계층만)	월 ~\$550 + 어차피 직접 구축해야 하는 미터링	월 \$50150 컴퓨터

이 결정을 방어해야 할 사람을 위해 정직한 주의사항 두 가지를 덧붙인다.

1. 90%라는 수치는 *게이트웨이 계층*의 비용에 관한 것이지 LLM 지출 자체가 아니다. 모델 토큰은 여전히 AI 청구서에서 압도적으로 가장 큰 항목이며, 이 책의 진짜 가치는 그 항목을 *통제*하는 데 있다(4~6장). 줄이는 데 있는 것이 아니다.
2. "커스텀"에는 엔지니어링 시간이 든다. 손익분기점은 실재한다. 관리형 청구서를 작게 소유한 컴포넌트와 맞바꾸는 셈이다. 1.6절은 이 거래가 가치 있는 시점을 가늠하며, 7장의 배포 체크리스트는 그 입장료다.

두 번째 비용 문제는 더 은밀하고, 실제로는 더 비싸다. 바로 **미터링되지 않은 정적 키** 문제다. 관리형 게이트웨이가 모델 제공사 앞에 놓여 있으면, 팀들은 제공사 API 키를 직접 발급받아 쓰게 된다. 관리형 게이트웨이가 사용자별 귀속도, 기능별 예산도 제공하지 않고, 무언가를 거쳐야 할 이유도 주지 않기 때문이다. 청구서는 분해되지 않은 하나의 거대한 숫자로 도착한다. 4장이 그 해법이고, 1장은 그 상처가 드러나는 자리다.

1.4 락아웃 문제

당신에게 팔린 벤더 추상화는 당신에게 필요한 추상화가 아니다.

관리형 게이트웨이와 관리형 부류의 서드파티 "AI 게이트웨이"는 대개 *모델 라우팅*을 제공한다. 지원하는 제공사 목록에서 고르고, 통일된 요청 형태를 받는 식이다. 그 제품은 지금은 낡아버린 전제 위에 세워졌다.

- **모델 ID가 URL이나 벤더 고유 필드 안에 살고 있다.** 기반 제공사가 모델 이름을 바꾸면(그리고 모든 주요 제공사가 12개월 안에 모델 이름을 바꾼 적이 있다) 그 추상화는 벤더에 의해 다시 배포되어야 한다. 벤더의 모델 이름을 바라보도록 되어 있던 당신의 코드베이스도 함께 바뀐다.
- **기능 커버리지는 당신의 일정이 아니라 벤더의 일정을 따른다.** 추론 토큰, 프롬프트 캐싱, 톨 호출 변형, 배치 API, 파인튜닝 엔드포인트 등을 각 제공사가 자기 시계에 맞춰 내놓는다. 관리형 추상화는 몇 달씩

뒤쳐지거나 롱테일 기능을 아예 빠뜨린다. 결국 흔한 경우에는 벤더 추상화를 호출하고 나머지 모든 경우에는 원시 제공사 SDK를 호출하게 되며, 이는 두 개의 코드 경로를 갖게 됨을 의미한다. 그리고 추상화는 둘 중 어느 쪽도 보호해주지 못한다.

- **데이터 경로가 불투명하다.** 어떤 제공사가 실제로 요청을 처리했는지, 재시도가 두 번째 제공사로 갔는지, 제공사의 최종 사용량이 얼마였는지 항상 알 수 있는 것은 아니다. 비용 정산 대사(4장)와 감사(3장)에 있어서, 불투명한 데이터 경로는 불편한 정도가 아니라 자격 미달 사유다.
- **전환은 설정 변경이 아니라 재작성이다.** 추상화가 벤더의 서비스 안에 있다면, 당신의 이식성은 벤더의 변경 이력에 달려 있는 셈이다.

아키텍처적 해법은 작으며, 2장에 담겨 있다. 코드베이스는 **당신이 소유한 별칭**(`standard`, `reasoning`, `embeddings-1`)을 바라보고, 레지스트리가 별칭을 제공사 모델 ID로 매핑하며, 그 매핑은 버전 관리되고 리뷰되는 *당신의* 설정이다. 제공사가 모델 이름을 바꾸면 레지스트리의 한 줄만 바꾸면 되지, 마흔 개 서비스의 한 줄씩을 바꿀 필요가 없다. 그것이 바로 당신이 소유한 추상화와 빌려 쓰는 추상화의 차이다.

1.5 통제 문제

통제에는 세 가지 얼굴이 있다. 컴플라이언스, 귀속, 그리고 폭발 환경이다.

컴플라이언스. 관리형 게이트웨이는 당신이 완전히 들여다볼 수 없는 데이터 처리자다. 어떤 구성 요소가 요청 본문을 로깅하는가? 로그는 어디로 복제되는가? 게이트웨이 자체의 자격 증명 범위를 제한할 수 있는가? LLM 트래픽, 즉 요청 본문이 곧 당신의 데이터인 경우, 많은 컴플라이언스 상황에서는 그 경로가 당신의 경계 안에 있어야 하고, 당신의 키, 당신의 감사, 당신의 보존 규칙을 따라야 한다. 커스텀 계층을 쓰면 당신의 VPC 안에 둘 수 있고, SIEM에 연결할 수 있으며, 벤더가 프라이버시 약관을 바꾸면 아키텍처에서 아예 들어낼 수 있는 구성 요소를 갖게 된다.

귀속. "누가 어떤 모델을, 어떤 기능을 위해 썼고, 비용이 얼마였는가"라는 질문에는 사용자, 팀/테넌트, 기능, 모델이라는 네 가지 축이 있다. 관리형 게이트웨이는 기껏해야 그중 하나(호출한 서비스를 식별한 API 키)만 제공한다. 팀 전체가 공유하는 정적 키를 쓴다면 그 하나의 축마저 거짓이 된다. 통제의 해법은 절차가 아니라 구조여야 한다. 사용자, 팀, 기능 범위를 담은 단명 ID 토큰, 그리고 이 네 축 모두를 기준으로 삼는 미터링이다. 이것이 3장과 4장의 내용이다.

폭발 환경. 범위도, 만료도, 사용자별 바인딩도 없는 정적 제공사 API 키는, 보안 용어로 말하면 당신의 AI 예산에 대한 루트 자격 증명이다. 이 키가 유출되면 공격자는 당신의 데이터를 읽는 것이 아니라 프런티어 모델 가격으로 *당신의 예산을 태운다*. 그리고 이 사건은 보안팀이 아니라 재무팀이 발견한다. 3장의 키리스 설계는 바로 이런 종류의 사고를 구조적으로 불가능하게 만들기 위해 존재한다. 유출될 정적 키 자체가 존재하지 않는 것이다.

1.6 언제 구축하고, 언제 구매할 것인가

멋져 보인다는 이유로 이 시스템을 구축하지 마라. 관리형 옵션은 특정한 범위의 상황에서는 충분히 괜찮다.

다음 경우에는 구매하거나 관리형을 유지한다.

- LLM 지출이 작고 안정적이다(대략 월 모델 토큰 비용이 수천 달러 미만이고 성장 계획이 없는 경우).

- 모델 하나, 제공사 하나, 팀 하나뿐이다.
- 컴플라이언스 상황이 데이터 경로를 제한하지 않는다.
- 작은 상시 운영 컴포넌트와 그 런북을 소유할 여력이 없다.

다음 중 하나라도 해당하면 구축한다.

- 모델 지출이 예산에서 *항목으로 잡힐 만큼* 크다. 반올림 오차 수준이 아니다.
- 한 팀 이상, 한 제품 이상이 모델을 소비하고 있고 "누가 얼마를 썼는가"가 미해결 질문이다.
- 제공사를 하나 이상 쓰거나, 하나에 전념할 수 없다.
- 사용자별 귀속이 필요하다(컴플라이언스, 공정성, 남용 탐지).
- 컴플라이언스 상황이 관리형 데이터 경로를 제한하거나 금지한다.
- MCP와 에이전트형 툴 접근(5장)을 진지하게 고려하고 있다. 관리형 게이트웨이는 일반적으로 툴 호출을 미터링하거나 제어하지 않는다.

구축 범위는 작다. 이 책의 레퍼런스 구현은 얇은 스테이트리스 프록시(Go 서비스 하나 또는 둘), 모델 레지스트리(테이블 하나와 동기화 작업), 미터링 파이프라인(스트림 하나와 롤업), 그리고 ID 통합이다. 2~4명의 엔지니어 팀이 기존 업무와 병행하며 90일 안에 v1을 출시할 수 있다. 2장에 주차별 계획이 있다. 대안은 "영원히 관리형으로 간다"가 아니라 "관리형에 직접 만든 미터링을 덧붙인다"이며, 이는 비용이 더 들면서도 락아웃 문제와 정적 키 문제는 여전히 해결하지 못한다.

1.7 구축함으로써 포기하는 것

이 결정은 언젠가 도전받을 것이므로, 정직하게 말해두자.

- **가용성을 직접 책임진다.** 관리형 계층의 가동 시간은 계약서에 적혀 있지만, 당신의 가동 시간은 페이지에 달려 있다. 7장은 SLO와 런북을 제공하며, 설계는 의도적으로 게이트웨이가 *우아하게 저하*되도록 만든다. 게이트웨이가 다운되면 범위가 제한된 키로 제공사에 직접 호출하는 방식이 통제력은 낮아지지만 계속될 수 있다(7.2절 참고). 완전한 장애는 아니다.
- **보안 표면을 직접 책임진다.** 이제 당신은 자격 증명 브로커다. 이는 특권이자 책임이다. 키 계층 구조, 로테이션, 브레이크글래스까지 모두 3장과 7장에 있다.
- **지루한 부분을 직접 책임진다.** 보존, 정산 대사, 대시보드, 승인. 이 책의 대부분은 지루한 부분이다. 가치는 거기에 있기 때문이다.
- **재무팀에 90% 수치를 설명해야 한다.** 1.3절의 표를 당신의 숫자로 채워서 가져가고, 그것이 게이트웨이 계층이지 토큰 지출이 아니라는 단서도 함께 가져가라.

1.8 앞으로 가져갈 프레임

각 문제당 한 문장씩, 세 문장이다.

- **비용:** LLM 트래픽은 토큰 단위로 미터링되고, 토큰을 볼 수 있는 유일한 구성 요소는 모델 호출 바로 앞에 있는 구성 요소다. 그래서 미터링은 거기에 살아야 하며, 그 미터링을 수행하는 얇은 프록시의 비용은 그렇게 하지 못하는 척하는 관리형 계층 비용의 일부에 불과하다.

- **락아웃:** 당신이 소유하는 추상화는 별칭 레지스트리다. 모델 ID는 코드베이스의 세부사항이 아니라 레지스트리의 세부사항이다.
- **통제:** 정적 키가 없고, 사용자별 귀속이 되며, 데이터 경로가 경계 안에 있고, 모든 1달러마다 사람, 팀, 기능, 모델의 이름을 댈 수 있는 감사 흔적이 있다.

다음 장은 이 세 문장을 현실로 만드는 아키텍처다.

1.9 합정

- **"그냥 관리형 게이트웨이에 넣자"는 회의.** 이에 대한 반박은 정치적이 아니라 구조적이어야 한다. 토큰 미터링, 사용자별 귀속, 데이터 경로 통제는 관리형 제품의 설정 옵션이 아니다. 토큰 단위 사용량이 어디서 미터링되는지 보여달라고 요청하라. 돌아오는 답은 "안 됩니다"일 것이다.
- **90%를 공짜와 혼동하는 것.** 절감은 게이트웨이 계층의 청구서에서 일어난다. 그 앞에 어떤 프록시가 있든 토큰 청구서는 변하지 않으며, 거기에 손을 대는 것이 4~6장이 할 일이다.
- **고통이 실재하기 전에 구축하는 것.** 팀이 하나이고 모델이 하나이고 청구서가 작다면, 관리형 계층이나 심지어 한 서비스 안의 제공사 SDK 정도로도 충분하다. 구축의 명분은 멀티팀 귀속, 멀티 제공사라는 현실, 또는 컴플라이언스지, 아키텍처 취향이 아니다.
- **락아웃은 당신의 URL 안에 있는 벤더의 모델 ID다.** 구축하기 전에 코드베이스에서 원시 제공사 모델 이름을 감사하라. 레지스트리(2.3절)는 바로 그 습관을 은퇴시키는 작업이며, 목록화하지 않은 것은 마이그레이션할 수 없다.
- **관리형 게이트웨이를 그것이 아닌 제공사 추상화로 취급하는 것.** 많은 관리형 티어는 당신의 백엔드로 라우팅하는 옛/제품이지, 모델 제공사로 프록시하는 제품이 전혀 아니다. "기존 APIM을 통해 LLM 호출을 라우팅하자"가 계획이었다면, 그것이 제공사 API로 이그레스할 수 있는지, SSE를 스트리밍할 수 있는지, 토큰을 미터링할 수 있는지 확인하라. 첫 번째는 될지 몰라도 나머지 둘은 안 될 가능성이 크다.

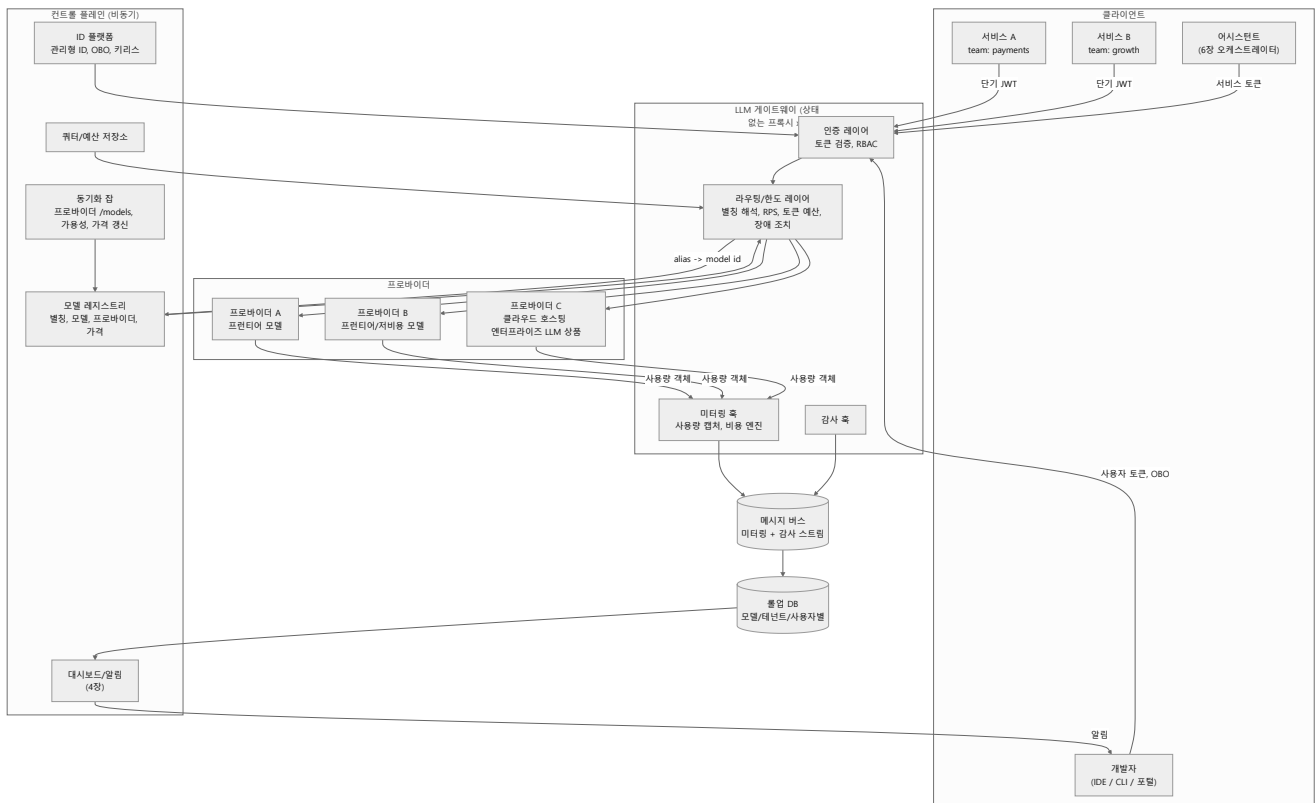
2장. 레퍼런스 아키텍처

이 장은 지도 역할을 한다. 이후 모든 장은 여기 나온 박스 중 하나를 확대해서 들여다보는 셈이다. 전체 모양을 파악하기 위해 이 장을 읽어두고, 나중에 다른 장에서 "미터링 파이프라인"이라는 말을 다시 설명 없이 쓸 때마다 여기로 돌아와 확인하면 된다.

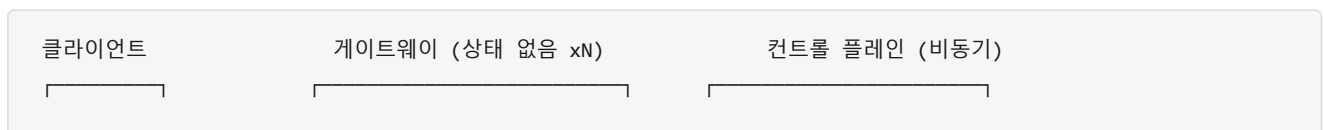
설계 입장을 미리 밝혀둔다. 이것이 게이트웨이와 플랫폼을 가르는 차이다.

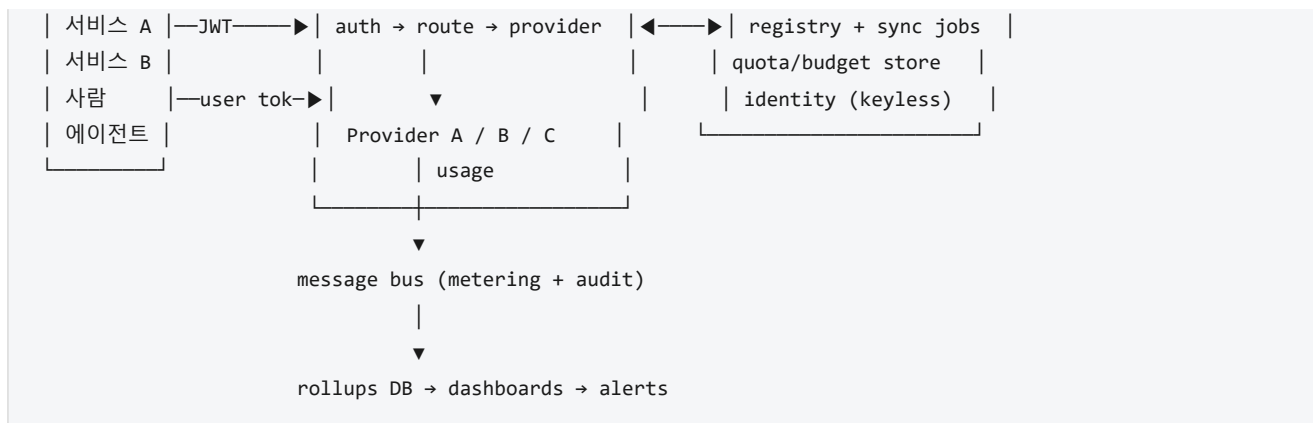
1. **컨트롤 플레인**은 하나, **데이터 플레인**도 하나. 사람이 바꾸는 모든 것은 컨트롤 플레인(레지스트리, 역할, 예산)에 산다. 데이터 플레인(프록시)은 상태가 없고 컨트롤 플레인 상태를 *읽기만* 한다.
2. **모든 것은 미터링된다.** 요청이 미터링 이벤트를 만들어내지 않았다면 그 요청은 일어나지 않은 것이다. 미터링 이벤트는 HTTP 응답만큼이나 실재하는 것이다.
3. **요청 안에는 비밀이 없다.** 네트워크 위를 오가는 자격 증명은 수명이 짧은 ID 토큰뿐이다. 프로바이더 자격 증명은 게이트웨이 프로세스의 비밀 영역 밖으로 절대 나가지 않는다(3장).
4. **지루한 기술을 쓴다.** 언어 하나, 메시지 포맷 하나, 스트림 하나. 게이트웨이의 일은 아키텍처 안에서 가장 재미없는 컴포넌트가 되는 것이다.

2.1 큰 그림



ASCII 대체 그림:





부하 경로는 다음과 같다: 클라이언트 → 인증 → 라우팅 → 프로바이더 → (스트림 반환) → 미터링 → 버스 → 롤업. 컨트롤 플레인도 캐시로부터의 읽기를 제외하면 핫 패스에 절대 끼지 않는다. 상태를 쓰기만 하고, 게이트웨이는 짧은 TTL(1~5초면 충분하다)로 변경 사항을 가져간다.

2.2 컴포넌트 목록

컴포넌트	실행 형태	상태	장애 양상
프록시(게이트웨이)	상태 없는 HTTP/SSE 서비스, N개 인스턴스	없음(레지스트리, 쿼터를 캐시)	인스턴스는 독립적; LB가 분산
모델 레지스트리	작은 Postgres/SQL DB의 테이블	별칭, 모델, 프로바이더, 가격	읽기 캐시; 오래된 읽기는 허용, 레지스트리 자체 부재는 불허(알 수 없는 별칭은 실패 단함)
동기화 잡	스케줄 워커	없음	마지막으로 성공한 스냅샷 유지; 오래된 가격은 플래그만, 치명적이지 않음
쿼터/예산 저장소	Redis(카운터) + SQL(정책)	카운터, 예산, 한도	고부하 시 카운터는 최선 노력(초과 허용 후 알림); 정책은 실패 단함
인증 레이어	프록시 내 인프로세스(라이브러리)	JWKS 캐시	토큰 검증은 실패 단함; JWKS 갱신은 백그라운드
미터링 혹	프록시 내 인프로세스	없음	버스 쓰기는 제한된 재시도와 함께 큐잉; 미터링 유실은 장애로 취급(2.11 참고)
메시지 버스	기존 스트림(Kafka/Event Hubs 등)	영속	표준 내구성
롤업 DB	SQL(v1에서는 레지스트리와 동일 DB도 무방)	사용량 팩트, 롤업	지연은 허용, 유실은 불허(버스에서 재계산)
대시보드/알림	기존 관측성 스택	쿼리	표준
웹 포털(5장)	컨트롤 플레인 앞단의 작은 앱	UI 상태만	비핵심 경로

의도적으로 빠진 것들에 주목하라: 별도의 "AI 플랫폼" 마이크로서비스 숲도 없고, 핫 패스에 ML 오케스트레이션 프레임워크도 없으며, 두 번째 데이터베이스도 없다. v1 시점에는 배포 단위 하나에 스케줄

잡이 전부다. 컴포넌트는 나중에 쪼갤 수 있고, 아래 인터페이스들이 그 이름매다.

2.3 멀티 프로바이더 자동 동기화를 갖춘 모델 레지스트리

1장에서 말한 잠금(lockout) 문제의 해법을 여기서 구체화한다.

테이블 두 개, 잡 하나.

providers — 외부 현실을 담는 테이블:

```
# providers.yaml (providers 테이블로 임포트됨; 검토 편의를 위해 YAML로 표시)
- id: provider-a
  endpoint: https://api.provider-a.example/v1
  auth: managed-identity # 게이트웨이가 해석; 설정 값에는 절대 들어가지 않음
  capabilities: [chat, streaming, tools, reasoning, cache]
  status: active
- id: provider-c-enterprise # "클라우드의 엔터프라이즈 LLM 상품"
  endpoint: https://llm.enterprise.example/openai/v1
  auth: managed-identity
  capabilities: [chat, streaming, tools, reasoning, embeddings, batch]
  status: active
```

models — 카탈로그. 각 행은 (provider, model_id) 하나와 그 *관측된* 상태를 담는다:

```
- alias: standard # 코드베이스가 부를 수 있는 이름
  provider: provider-c-enterprise
  model_id: enterprise-gpt-4o-class-2026-01 # 벤더 상세, 호출자에게는 숨김
  capabilities: [chat, streaming, tools, reasoning, cache]
  context_window: 128000
  price_per_mtok: { input: 2.50, output: 10.00, cached_input: 0.25 } # USD, 주기적으로 갱신
  status: active # active | deprecated | unavailable
  observed:
    last_sync: 2026-01-15T04:00:00Z
    availability: ok # ok | degraded | down (프로브 기반)
    deprecation_notice: null # 또는: "2026-04-30: provider sunset"
```

aliases — 조직이 소유하는 것:

```
- alias: standard
  policy: { min_capability: [chat, tools], max_cost_rank: 2, prefer: provider-c-enterprise }
  binding: standard->provider-c-enterprise/enterprise-gpt-4o-class-2026-01
  canary: { alias: standard-v2, traffic: 5% }
- alias: reasoning
  binding: reasoning->provider-a/frontier-reasoning-2026
- alias: embeddings-1
  binding: embeddings-1->provider-c-enterprise/embed-3-class
```

동기화 잡은 스케줄에 따라 도는 작은 루프 세 개다(기본값: 가용성은 15분마다, 가격은 매일, 지원 종료 공지는 매시간. 프로바이더의 변경 속도에 맞춰 조정한다):

1. **카탈로그 동기화**: 프로바이더의 모델 목록 엔드포인트가 있으면 호출해서 `models` 와 비교한다. 새 모델이 보이면 `status: unreviewed` 행으로 추가한다(포털에는 보이지만 사람이 승인하기 전까지는 라우팅되지 않는다). 모델이 제거되면 행은 유지하되 `status: gone` 으로 표시하고 알림을 보낸다. *바인딩을 자동으로 삭제하지 않는다.*
2. **가용성 프로브**: `active` 상태인 바인딩마다 작고 저렴하며 표시된 프로브 요청을 보낸다(해당 프로바이더의 가장 저렴한 활성 모델에 대한 토큰 1개짜리 완성 요청이며, 미터링에서 `probe: true` 로 태그해 눈에 띄고 비즈니스 비용에서 제외할 수 있게 한다). 실패율을 추적하고, 주 *별칭*을 담당하는 프로바이더가 저하되면 `availability` 를 바꾸고 호출(페이지)한다.
3. **가격 갱신**: 프로바이더가 공개한 가격표(API든 파싱한 페이지든, 프로바이더별로)를 `price_per_mtok` 로 가져온다. 공개 가격이 저장된 값 대비 임계치(예: $\pm 10\%$) 이상 움직였으면 알림을 보낸다. 왜냐하면 *관측된 비용*(사용량 객체에서, 4장)과 *예산에 반영된 가격*(예산 책정 기준)이 서로 벌어질 수 있고, 그 벌어짐은 버그가 아니라 정산 대사(reconciliation) 발견 사항이기 때문이다.
4. **지원 종료 감시**: 프로바이더가 지원 종료 메타데이터를 노출하거나(또는 체인지로그가 기계 판독 가능하면) 영향받는 모델에 `deprecation_notice` 를 기록하고 이슈 트래킹 시스템에 티켓을 연다. 그 티켓의 해결책은 레지스트리 변경이지 코드 변경이 아니다. 그것이 레지스트리를 두는 이유다.

왜 수작업 카탈로그가 아니라 자동 동기화인가: 카탈로그의 가치는 그것이 *사실*이라는 데 있다. 손으로 관리하는 카탈로그는 한 분기 안에 썩고, 사람들이 더 이상 믿지 않게 되며, 그 순간 별칭 레이어는 그냥 느린 거짓말이 된다. 사실(facts)은 자동으로 동기화하고 *바인딩*(호출자가 이름 붙일 수 있는 것)은 사람이 승인하는 것이 올바른 분리다: 사실은 동기화하고, 정책은 검토한다.

2.4 통합 API 표면

흔한 케이스를 위한 단일 와이어 프로토콜, 롱테일을 위한 패스스루.

정규 엔드포인트(의도적으로 OpenAI 호환 형태):

```
POST /v1/chat/completions      # 스트리밍(SSE)과 비스트리밍
POST /v1/embeddings
POST /v1/responses             # 톨 호출/에이전틱 루프 형태(2026년대)
GET  /v1/models               # 프로바이더 카탈로그가 아니라 자체 별칭
GET  /v1/usage/me             # 역할별 자신의 사용량(4장)
```

- 요청 본문은 공통 형태를 따른다: `model` 은 **반드시** 자체 별칭 중 하나여야 하며, 원본 프로바이더 모델 ID는 거부된다(이것이 한곳에서 강제되는 잠금(lockout) 방화벽이다).
- 게이트웨이는 프롬프트에 아무것도 덧붙이지 않는다. 대신 프로바이더로 가는 와이어에 *헤더*를 추가한다: 프로바이더별 모델 ID, 자체 프로바이더 자격 증명(호출자에게는 절대 보이지 않음), 상관관계(correlation) ID.
- **사용량 객체**: 스트림의 마지막 SSE 이벤트를 포함해 모든 응답에는 게이트웨이가 하나의 스키마로 *정규화*한 `usage` 블록이 실린다:

```
{
  "input_tokens": 4120,
```

```

"output_tokens": 356,
"cached_input_tokens": 3800,
"reasoning_tokens": 0,
"modalities": { "images_in": 0, "images_out": 0 },
"provider_model_id": "enterprise-gpt-4o-class-2026-01",
"provider": "provider-c-enterprise",
"latency_ms": { "ttfb": 210, "total": 4820 }
}

```

정규화 레이어는 멀티 프로바이더가 실제로 의미를 갖게 되는 지점이다. 각 프로바이더는 서로 다른 필드 이름으로 사용량을 보고하고, 일부는 캐시된 토큰을 생략하며, 일부는 추론(reasoning) 토큰을 별도로 과금한다. 정규화기는 프로바이더별 어댑터로 위 스키마에 매핑하고, **미터링 파이프라인(2.6)은 오직 정규화된 객체만 소비한다**. 프로바이더 응답에 사용량 블록이 없으면 게이트웨이는 직렬화된 메시지로부터 추정하고(`estimated: true` 로 표시) 정산 대사 작업(4장)은 추정치가 존재한다는 것을 안다.

- **롱테일을 위한 패스스루**: `POST /ptp/{provider}/{any-path}` — 인증, 미터링, 감사는 거치지만 본문은 손대지 않는다. 배치 API, 파인튜닝, 정규 표면에 없는 프로바이더 고유 기능을 위한 것이다. 패스스루도 인증, RPS, 토큰 예산 검사는 거치지만 별칭 규칙은 강제하지 않는다. (그렇다, 이것은 원본 모델 ID를 허용한다. 다만 범위가 좁고, 플래그가 붙으며, 롱테일 전용으로 예약된다. 대안은 통제되지 않는 두 번째 경로를 두는 것인데, 그게 더 나쁘다.)
- **멱등성(idempotency)**: 비스트리밍 호출에는 `Idempotency-Key` 헤더가 붙고, 게이트웨이는 요청 해시 → 응답 매핑을 24시간 보관한다. 스트리밍 호출은 멱등하지 *않다*(절반짜리 스트림을 안전하게 재생할 수 없다). 그래서 이 경우 계약은 최대 1회(at-most-once)이고 재시도 정책은 클라이언트 쪽 책임이며, 미터링 파이프라인은 프로바이더 요청 ID로 중복을 제거한다. 그래서 프로바이더가 두 번 과금한 재시도 요청은 조용히 합쳐지지 않고 두 개의 사용량 객체로 드러난다.

2.5 라우팅/한도 레이어

라우팅 레이어는 순서대로 이렇게 답한다: *누가 할 수 있는가? 얼마나 남았는가? 누가 처리하는가? 처리할 수 없으면 어떻게 하는가?*

1. **별칭 해석**: `model: standard` → 바인딩 → (provider, model_id). 바인딩의 프로바이더가 프로브 결과 `degraded` 상태면 정책에 따라 장애 조치(fail-over)한다. 먼저 같은 별칭의 2차 바인딩(예: `standard` 가 낮은 순위로 프로바이더 B 모델에도 바인딩돼 있는 경우), 그다음엔 `503 model_unavailable` 과 `Retry-After` 로 거부한다. 장애 조치는 *눈에 보인다*. 응답 헤더 `X-LLM-Router: failed-over=1` 과 미터링 이벤트의 `routed_to != requested_binding` 이 그 증거라서, 비용 리포트는 "standard 트래픽의 X% 가 보조 프로바이더에서 실행됐다"고 말할 수 있다.
2. **한도는 두 레이어**:
 - **RPS**(초당 요청 수, ID+역할+별칭별 토큰 버킷): 프로바이더와 자체 큐를 보호한다.
 - **토큰 예산**(4장에서 자세히, 여기서는 요약): 사용자별, 팀별, 별칭 클래스별 일/주/월 토큰 및 비용 예산. 사전 검사는 프로바이더 호출 전에 요청이 선언하거나 추정한 크기로 돈다. 정산은 호출 후 실제 사용량 객체로 돈다. 이 2단계 설계가 중요한 이유: 사전 검사는 추정치로 허용하고 정산은 진실로 교정한다. 그래서 장문 컨텍스트를 반복적으로 추정 초과하는 팀은 청구서에 찍히기 전에 정산 편차 지표로 먼저 드러난다.

3. **우선순위와 대기열**: v1에는 대기열이 없다. 한도 초과는 빠른 429 에 Retry-After 와 명확한 예산 포인터("reasoning 별칭의 월간 예산 91%를 사용 중입니다")를 붙여 보낸다. 큐잉은 v2 기능이다. 미터링 파이프라인은 이미 이를 지원할 모양(버스에 테넌트별 레인이 있음)을 갖췄지만, 압박을 팀으로부터 숨기는 큐는 UX만 좋아진 비용 문제일 뿐이다.
4. **프로바이더별 서킷 브레이커**: 특정 프로바이더에서 N초 동안 에러율이나 p95 지연이 임계치를 넘으면 그 프로바이더의 **신규** 연결에 대해 브레이커가 열린다(진행 중인 스트림은 끝까지 간다). 트래픽은 장애 조치 정책에 따라 옮겨가고, 브레이커 상태는 1급 지표다. 이것이 "프로바이더 C가 오늘 컨디션이 안 좋다"와 "우리 서비스가 녹아내리고 있다"의 차이를 만든다.

하트 패스의 Go 스타일 의사 코드(의도적으로 평탄하다. 이게 루프 전체다):

```
func (g *Gateway) Handle(r *http.Request) {
    id, err := g.auth.Verify(r)          // JWT: user, team, role, scopes
    if err != nil { g.deny(r, err); return }

    req, err := g.decode(r)              // canonical or pass-through
    if err != nil { g.deny(r, err); return }

    if !g.auth.Allows(id, req.Alias, req.Mode) { g.deny(r, ErrRBAC); return }

    if err := g.limits.CheckRPS(id, req.Alias); err != nil { g.deny(r, ErrRPS); return }
    if err := g.limits.CheckBudget(id, req.Alias, req.Estimate()); err != nil {
        g.deny(r, err); return           // err carries budget % and owner contact
    }

    binding, over := g.registry.Resolve(req.Alias)
    resp, usage, err := g.provider.Call(binding, req, id) // streams back to client
    // metering: always, even on error paths (partial usage is still usage)
    g.meter.Emit(MeterEvent{ ID: id, Alias: req.Alias, RoutedTo: binding,
        Usage: usage, Est: req.Estimate(), Over: over, Err: err })
    g.audit.Emit(AuditEvent{ ID: id, Alias: req.Alias, Bytes: r.ContentLength,
        Outcome: outcome(err), TTFB: ... })
}
```

2.6 미터링/비용 파이프라인

파이프라인은 **캡처** → **정규화** → **비용 계산** → **롤업** → **발행** 순서로 흐른다.

- **캡처**: 모든 요청은 정확히 하나의 **MeterEvent** (위 의사 코드)를 만든다. 실패도 포함이다(이미 입력 토큰 4만 개를 소비한 프로바이더의 500 에러는 부분 사용량을 가진 이벤트다. 프로바이더가 그만큼 과금하니 우리도 그래야 한다).
- **정규화**: 프로바이더 어댑터 → 정규 사용량 스키마(2.4). 어댑터는 프로바이더별 파일이고 녹화된 픽스처로 유닛 테스트되며, 테스트 코퍼스에는 프로바이더별 "사용량 블록 없음" 픽스처도 *포함된다*.
- **비용**: $cost = \sum (tokens \times price_per_mtok)$ 를 라인 아이템(입력, 출력, 캐시된 입력, 추론, 모달리티별)에 대해 계산하되, 요청 시점에 **책정된** 가격을 쓴다(가격은 이벤트에 스냅샷되며 다시

유도되지 않는다. 이것이 월간 정산 대사를 가능하게 한다. 어느 날의 이벤트든 그날의 가격으로 재생할 수 있다).

- **롤업**: 스트림 컨슈머는 다음을 유지한다:
 - Redis의 **핫 카운터**: (사용자, 팀, 별칭)별 최근 5분/24시간/월간. 사전 검사와 실시간 대시보드에 쓰인다.
 - SQL의 **팩트**: 요청당 한 행(정리된 형태), 90일간 핫 상태로 보관 후 콜드로 에이징(7장).
 - **롤업 테이블**: 대시보드와 정산 대사를 위한 일별 × (모델, 별칭, 팀, 사용자, 기능) 집계.
- **발행**: v1에서는 별도의 "메트릭 레이크"가 없다. 롤업 DB 자체가 대시보드 데이터 소스다.

기능(feature) 차원: 호출자는 `X-LLM-Feature: code-review` (또는 포털에 등록된 기능 이름)를 선언한다. 기능은 팀 리드가 소유하는 등록된 객체다. 등록되지 않은 기능 이름도 받아들여되 `unregistered` 로 표시되며, 정산 대사 리포트(4.7)는 등록되지 않은 기능별 지출을 나열한다. 이것이 귀속(attribution) 측을 정확하게 유지하는 넋지다.

2.7 감사

감사는 미터링과 분리된 *별도의* 스트림이고, 그 분리 자체가 설계다. 미터링은 "비용이 얼마였나"에 답하고, 감사는 "무엇이 실행됐나"에 답한다. 감사 이벤트:

```
{
  "ts": "2026-01-15T10:22:31.412Z",
  "actor": { "principal": "user:8842", "team": "payments", "role": "contributor",
    "on_behalf_of": null, "client": "service:code-review-bot" },
  "action": "llm.inference",
  "object": { "alias": "reasoning", "feature": "code-review" },
  "outcome": "ok",
  "bytes_in": 48211, "bytes_out": 2104,
  "policy": { "rps_bucket": "contributor:reasoning", "budget_check": "admit" },
  "correlation_id": "req_01J8ZK..."
}
```

중요한 속성들: 추가 전용(append-only)이며 감사 스트림에 한 번만 쓴다. 보존 기간과 접근 규칙은 7장에서 다룬다. **기본적으로 페이로드를 보지 않는다** — 감사는 모양(shape)만 기록하고 내용은 기록하지 않는다 (내용 로깅은 엔드포인트 클래스별로 별도 승인된 명시적 모드이고, 기본 자세는 요청 본문이 절대 감사 스트림에 들어가지 않는 것이다). 감사자 역할(3.5)의 접근 권한은 감사 로그만 읽을 뿐, 미터링의 비용 컬럼이나 내용은 절대 읽지 않는다.

2.8 관측성

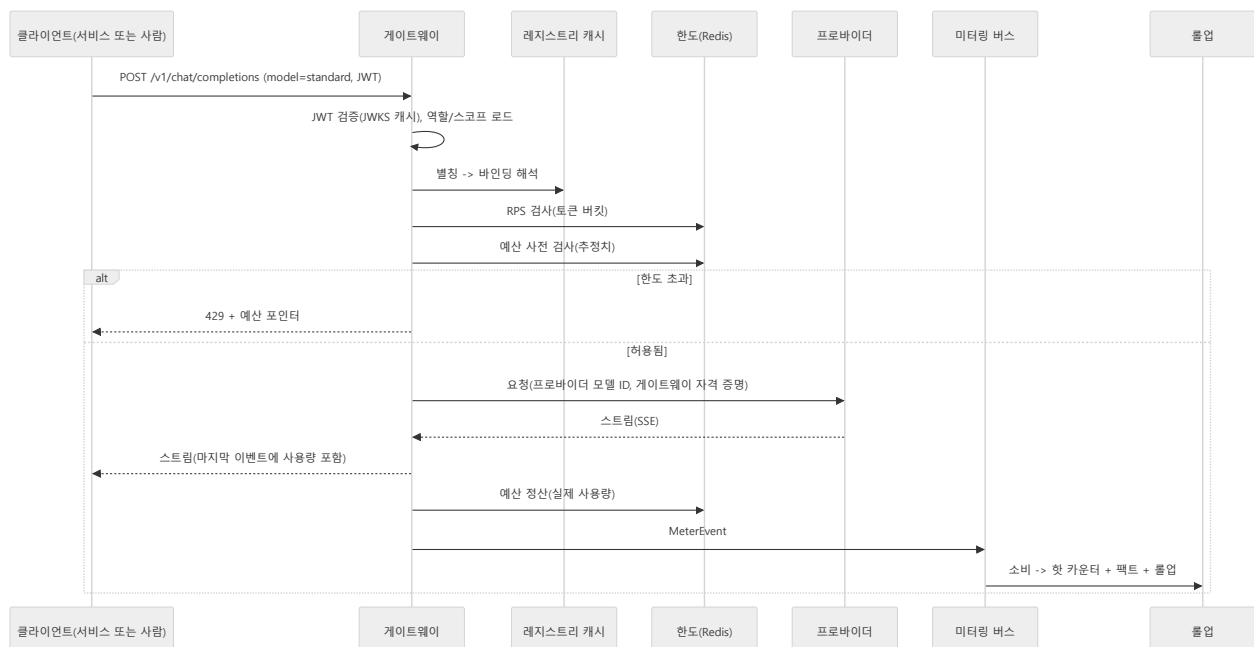
v1에서는 의도적으로 최소한인 세 가지 신호만 둔다:

- **메트릭**(게이트웨이 프로세스에서): (별칭, 프로바이더, 결과)별 요청률/에러율/지연(p50/p95/p99, TTFB, 전체 스트림 시간); 예산 허용/거부율; 브레이커 상태; 레지스트리 캐시 나이; 미터링 버스 지연.

- **트레이스:** 게이트웨이 홉 하나 + 프로바이더 호출당 스펠 하나, `correlation_id` 가 전파된다.
트레이스는 *정규화된 사용량*을 스펠 이벤트로 실어서, 트레이스 하나만 봐도 "이 요청: 입력 토큰 N개, 출력 M개, \$0.0214"라고 답할 수 있다. 트레이스별 비용은 이 시스템에서 가장 쓸모 있는 디버깅 뷰이고, 사용량이 스펠에 올라가는 순간 공짜로 얻어진다.
- **로그:** 요청당 한 줄의 구조화된 JSON(본문 없음). 필드는 미터링 이벤트와 동일해서 로그 쿼리와 사용량 쿼리가 서로 일치한다.

v1의 SLO(조정하되 여기서 시작): 게이트웨이 가용성 99.9%(월간), TTFB p95 < 150ms *게이트웨이가 더한 부분만*(프로바이더 지연 제외 — 둘 다 따로 측정), 미터링 완전성 ≥ 99.9%의 요청(대시보드로 보고, 99.5% 밑으로 떨어지면 알림). 완전성 SLO는 비용 수치를 재무팀이 신뢰할 수 있게 만드는 장치이고, CFO의 질문이 걸려 있는 SLO다.

2.9 요청 생애주기(시퀀스)



데이터를 나르는 모든 화살표에는 대응하는 대시보드 타일이 있다(4장). 관측할 수 없는 단계는 아키텍처에 없는 것이나 마찬가지다.

2.10 배포 형태

- **프록시:** LB(또는 메시) 뒤에 3개 이상의 인스턴스, 수평 확장 가능, 캐시 외에는 로컬 상태 없음. SSE를 인지하는 타임아웃(프로바이더별로 넉넉하게. LB의 유휴 타임아웃은 예상되는 가장 긴 스트림보다 길어야 한다. 전형적인 조용한 장애이니 함정 절 참고).
- **Redis:** 한도/카운터 계층. v1에서는 백업이 있는 단일 관리형 인스턴스로 충분하다(카운터는 재계산 가능하다. 하나를 잃으면 TTL 동안 초과 허용이 생기는 것이지만 데이터 유실이 아니다).
- **SQL:** 레지스트리 + 팩트 + 롤업. v1에서는 DB 하나에 스키마만 분리.

- **버스:** 기존 스트림 토픽 두 개(`llm.metering.v1`, `llm.audit.v1`). 감사 토픽은 기본값이 아니라 7장의 보존 기간 정책을 따른다.
- **ID:** 게이트웨이 자체의 워크로드 ID(관리형, 키리스 — 3장)에 더해 호출자 토큰을 검증할 JWKS 엔드포인트.
- **laC:** 데이터 플레인 전체가 모듈 하나다. 환경 간 동등성은 설정 파일 하나로 해결한다. 7장의 배포 체크리스트가 "내 샌드박스에서는 됨"과 "배포됨" 사이의 관문이다.

2.11 설계된 장애 양상

장애	설계상 대응	남는 위험
프로바이더 장애	프로브 → 브레이커 → 장애 조치 바인딩 또는 빠른 503	모든 바인딩이 죽으면 사용자도 보게 된다. 그게 맞다
레지스트리 DB 다운	캐시가 TTL(5분) 동안 읽기 제공; 알 수 없는 별칭은 실패 닫힘	새 바인딩은 TTL 이하 동안 반영 안 됨
Redis 다운	RPS는 실패 열림 (알림), 예산은 실패 닫힘 (포인터와 함께 거부) — 예산은 지출을 보호하고 RPS는 프로바이더를 보호하기 때문	RPS 쪽은 짧은 초과 허용 구간 발생
버스 다운	제한된 인프로세스 큐(60초) + 알림; 오버플로 시 미터링 유실 지표 증가	유실은 측정되지, 조용히 묻히지 않는다
게이트웨이 자체 다운	디그레이드 모드(7.2): 범위가 제한된 직접 프로바이더 키로 통제 수준을 낮춰 지속	장애 동안 통제 공백 생김; 런북이 메움
가격 데이터 오래됨	책정 가격은 이벤트별로 스냅샷; 드리프트 알림	갱신 전까지는 오래된 가격으로 예산이 돈다 — 허용 가능하며 플래그됨

이 표에서 보이는 설계 목표: **조용히 지나가는 장애는 없다**. 모든 저하는 지표이거나 알림이거나, 둘 다다.

2.12 90일 안에 구현하는 법

전제: 엔지니어 2~4명, 기존 클라우드, 기존 ID 플랫폼, 기존 메시지 버스. 이 계획은 설명된 규모에서 실제로 착륙한 적이 있다. 하나의 모양이지, 약속은 아니다.

1~15일 — 파이프. 프록시 뼈대를 세운다: 정규 `/v1/chat/completions` (스트리밍 + 비스트리밍), 프로바이더 하나, 별칭 하나(`standard`), 기존 ID 플랫폼의 워크로드 토큰을 통한 인증(이 단계에서는 사람용 키가 없어도 괜찮다), 버스로 보내는 미터링 이벤트, 그 프로바이더 하나에 대한 사용량 정규화, 요청당 한 줄 로그. **종료 조건:** 조직 내 실제 서비스 하나가 이걸 통해 `standard` 를 호출하고, 그 사용량이 롤업 DB에 들어간다.

16~30일 — 레지스트리. 모델/프로바이더/별칭 테이블; 첫 프로바이더에 대한 카탈로그 + 가용성 프로브 + 가격 갱신 동기화 잡; `/v1/models` 가 별칭을 반환; 별칭 해석 + 장애 조치 정책(바인딩이 하나뿐이어도 코드 경로는 존재); 프로브 태그가 붙은 미터링 제외 처리. **종료 조건:** 레지스트리에서 모델 이름을 바꾸면 호출자 측 변경 없이 바인딩이 따라간다.

31~45일 — 인증과 RBAC(3장). 완전한 키리스 브로커: 어떤 설정에도 정적 프로바이더 키가 없음; 사람 호출자를 위한 OBO 흐름; 역할별 RPS와 일일 토큰 예산을 강제하는 6개 역할; 2단계 예산 검사. 종료 조건: contributor 토큰은 예산을 429 없이 넘을 수 없고, auditor 토큰은 감사 로그 외에는 아무것도 읽을 수 없다.

46~60일 — 미터링과 대시보드(4장). 팩트 + 롤업, 네 가지 대시보드 뷰(실시간 비용, 예산 소진, 모델별, 팀별), 알림 설계(채널 + 웹훅), 정산 대사 잡 v1(사용량 대 프로바이더 청구서, 프로바이더 하나). 종료 조건: 재무팀이 이번 달 지출을 팀별로 하루 단위까지 볼 수 있고, 지난달 정산 대사 실행 결과가 프로바이더 청구서와 허용 편차 안에서 일치한다.

61~75일 — 두 번째 프로바이더 + 패스스루 + 감사. 두 번째 프로바이더를 엔드투엔드로(정규화기가 제 몫을 하는 지점); /ptp 패스스루; 미터링과 분리된 감사 스트림; 보존 기간 정책; 서킷 브레이커와 장애 조치 훈련(스테이징에서 프로바이더 장애를 유발해보기. 장애 조치는 지루할 만큼 평온해야 한다). 종료 조건: 장애 조치 훈련이 기억이 아니라 운영 위키의 스크린샷으로 남는다.

76~90일 — 경화(hardening), 전환, 첫 마이그레이션. 배포 체크리스트(7.6)를 프로덕션에 맞춰 실행; 마이그레이션 체크리스트(7.7)를 첫 번째 실제 팀을 직접 프로바이더 호출에서 떼어내는 데 실행; SLO 가동; 장애 런북(7.4)을 출력해 첫(계획된) 훈련을 완료. 종료 조건: 팀 하나가 마이그레이션됐고, 그 사용량이 완전히 귀속되며, 떠난 관리형 게이트웨이나 직접 호출 경로는 폐기 절차를 밟고 있다.

90일 안에 의도적으로 빠진 것: MCP 서버(5장 — 그 안에 자체 90일 계획이 있다), RAG 어시스턴트(6장), 웹 포털의 셀프서비스 흐름(읽기 전용 포털은 46~60일에 나오고, 셀프서비스는 그 이후), 큐잉, 멀티 리전. 이것들은 각각 하나의 장(chapter)이지 스트레치 골이 아니다.

2.13 합정

- **게이트웨이가 아니라 플랫폼을 짓는 것.** 레퍼런스는 배포 단위 하나에 잡을 더한 것이다. 설계 문서에 핫 패스에 서비스 일곱 개가 있다면, 이 책이 말하는 게이트웨이가 아니라 회사를 짓고 있는 것이다. 이음대 (2.2)는 실제로 애플 때만 쪼갬다.
- **스트림보다 짧은 LB 유희 타임아웃.** 가장 흔한 조용한 장애다. 긴 스트림이 LB에서 죽고, 클라이언트는 잘린 응답을 보고, 프로바이더는 전체 출력에 대해 과금했는데, 미터링은 "잘림"이라 하고 청구서는 "완료"라고 한다. LB/인그레스 유희 타임아웃을 p99 스트림 길이보다 위로 설정하고, 마지막 사용량 이벤트 유무에 따른 스트림 중간 연결 끊김 대시보드도 추가하라.
- **클라이언트가 선언한 모델을 신뢰하는 것.** 원본 프로바이더 모델 ID를 보내는 호출자는 잠금(lockout) 방화벽을 우회하는 것이다. 거부하고, 로그를 남기고, 그 팀을 포털로 안내하라. 방화벽은 한곳에서만 거부될 때 비로소 방화벽이다.
- **성공만 미터링하는 것.** 입력 토큰 4만 개를 소비한 프로바이더 500 에러는 4만 토큰짜리 이벤트다. 캡처 조건이 `if err == nil` 이라면, 장부는 정확히 프로바이더 에러율만큼 틀린 것이고, 정산 대사 장이 그걸 잡아낸다.
- **리포트 시점에 가격을 다시 유도하는 것.** 비용은 요청 시점에 이벤트별로 스냅샷돼야 한다. 오늘의 가격으로 다시 유도하면 과거를 재생할 수 없게 되고 정산 대사가 불가능해진다. 이건 데이터 모델 결정이고, 잘못 내리면 되돌릴 수 없다.
- *첫날부터 레지스트리에 프로바이더가 하나뿐인 건 괜찮지만, 아키텍처에 프로바이더가 하나뿐인 건 괜찮지 않다.* 정규화기, 장애 조치 코드, 멀티 프로바이더 롤업은 두 번째 프로바이더가 오기 전에

존재해야 한다. "필요할 때 두 번째 프로바이더를 추가하자"는 식이면 두 번째 프로바이더 추가에 한 분기가 걸리는 길이다.

- **프로브의 비용은 0이 아니다.** 프로브 요청도 실제 토큰이다. 태그를 붙이고, 비즈니스 비용에서 제외하고, 예산을 책정하라(사소한 비용이지만, 태그 없는 프로브는 설명 안 되는 정산 대사 편차가 된다).
- **레지스트리는 정책의 진실 공급원, 동기화는 사실의 진실 공급원 — 절대 거꾸로가 아니다.** 누군가 동기화되는 가용성 필드를 손으로 고쳤는데 다음 동기화가 덮어쓴다면, 그 필드는 애초에 정책 필드가 아니었던 것이다. 그런 충돌이 생기면 고칠 것은 정책 컬럼을 추가하는 것이지, 동기화를 멈추는 것이 아니다.