

LiteLLM Proxy로 세우는 LLM Gateway: AI 에이전트 모델 호출 길목을 하나로 모으는 방법

호출 경로 통합에서 키 분리와 모니터링까지

AI 에이전트를 운영하는 조직이 먼저 정리할 대상은 모델보다 모델로 가는 길목입니다. 에이전트, 보조 스크립트, 독립 서비스가 각자 프로바이더를 호출하면 키 관리, 비용 집계, 모델 교체에서 통제가 비게 되고, 호출 주체가 늘수록 이 공백도 커집니다.

목차

LiteLLM Proxy로 세우는 LLM Gateway: AI 에이전트 모델 호출 길목을 하나로 모으는 방법

- 1장: LLM 호출 경로가 흩어지면 생기는 일: 키·비용·모델 교체의 통제 공백
- 2장: LiteLLM 이 만들어진 이유와 LiteLLM Proxy 가 필요해진 배경
- 3장: LiteLLM SDK 와 LiteLLM Proxy 의 차이: 앱 안의 라이브러리와 조직의 게이트웨이 서버
- 4장: LLM Gateway 가 갖춰야 할 조건과 LiteLLM Proxy 의 충족 범위
- 5장: 오픈소스 LLM Gateway 지형과 LiteLLM 의 위치: 상위 5개와 국내 공급 유형
- 6장: 도커에서 운영하는 LiteLLM Proxy 구성: 모델 별칭 9개, 풀백 3건, 소비자 6종
- 7장: AI 에이전트 토폴로지와 각 에이전트의 역할: 스택 2개와 프로파일 16개의 요청 흐름
- 8장: LLM Gateway 모니터링: 관측 대상 다섯 묶음과 현재 구성의 공백
- 9장: 한계와 다음 단계 그리고 기술적 우위 요약: 의존 관계에 따른 조치 순서
- Appendix A. References
- Appendix B. Glossary

LiteLLM Proxy로 세우는 LLM Gateway: AI 에이전트 모델 호출 길목을 하나로 모으는 방법

AI 에이전트를 운영하는 조직이 먼저 정리할 대상은 모델보다 모델로 가는 길목입니다. 에이전트, 보조 스크립트, 독립 서비스가 각자 프로바이더를 호출하면 키 관리, 비용 집계, 모델 교체에서 통제가 비게 되고, 호출 주체가 늘수록 이 공백도 커집니다. LiteLLM Proxy는 100개가 넘는 프로바이더를 OpenAI 형식 하나로 호출하게 해 주는 오픈소스 LLM Gateway입니다[S01][S02]. 호출 경로를 이 한 곳으로 모으면 에이전트가 늘어도 모델 운영에서 고칠 곳은 프록시 설정 파일 한 곳으로 유지됩니다. 오픈마루의 운영 구성에서는 에이전트 프로파일 16개가 같은 모델 별칭과 같은 프록시 주소를 가리키므로, 기본 모델을 교체할 때 별칭 정의 한 곳만 수정합니다. 이 구성을 사례로 드는 이유는 길목을 모아서 얻은 이점과 아직 남은 공백을 운영 중인 환경에서 함께 확인할 수 있기 때문입니다.

다만 길목을 모으는 것만으로 통제와 관측이 함께 갖춰지지는 않습니다. 같은 구성을 LLM Gateway 요건 평가표의 여덟 항목으로 판정하면 충족이 하나, 부분 충족이 넷, 공백이 셋입니다. 소비자 6종이 마스터 키 하나를 공유하고, 지표 수집은 켜져 있지 않으며, 프록시를 거치지 않는 경로도 3건 남아 있습니다. 그래서 이 문서가 제시하는 순서는 다음과 같습니다. 먼저 흩어진 LLM 호출 경로를 조사해 LiteLLM Proxy 한 곳으로 모읍니다. 이어 소비자별 가상 키로 키를 분리하고, 지연과 첫 토큰 시간, 폴백 발생을 보는 모니터링 지표 수집을 다음 단계로 정합니다.

엔지니어는 구성과 설정을 다루는 3장, 6장, 7장, 8장에서, 의사결정권자는 요건과 대안, 한계를 다루는 4장, 5장, 9장에서 판단 근거를 얻을 수 있습니다.

1장: LLM 호출 경로가 흩어지면 생기는 일: 키·비용·모델 교체의 통제 공백

AI 에이전트 운영의 병목은 모델이 아니라 모델로 가는 길목에서 생깁니다. LLM(Large Language Model, 대규모 언어 모델)을 호출하는 코드가 에이전트, 보조 스크립트, 독립 서비스마다 따로 있으면 모델의 성능과 무관하게 키 관리, 비용 집계, 모델 교체 세 가지 업무에서 통제가 비게 됩니다. LiteLLM Proxy는 이렇게 흩어진 호출 경로를 한 곳으로 모으는 오픈소스 LLM Gateway입니다. LLM Gateway는 호출 주체와 모델 사이에서 인증, 라우팅, 기록을 한 곳에서 처리하는 중계 서버를 가리키며, LiteLLM은 100개가 넘는 프로바이더(모델 API를 제공하는 사업자)를 OpenAI 형식 하나로 호출하게 해 줍니다[S01][S02].

호출 경로가 지금 문제가 되는 이유는 에이전트가 사람보다 훨씬 자주, 그리고 사람이 지켜보지 않는 시간에도 모델을 호출하기 때문입니다. 에이전트는 모델이 도구 호출을 요청할 때마다 도구 결과를 가지고 모델을 다시 호출하므로 대화 한 번이 요청 여러 건으로 늘어납니다. 여기에 예약 작업이 더해지면 호출 주체의 수와 호출 횟수가 함께 커집니다. 오픈마루는 오픈소스 에이전트 프레임워크 Hermes Agent로 프로파일 16개를 운영하고 있고, 등록된 예약 작업 94건 가운데 48건이 LLM을 사용합니다. 프로파일은 설정 파일, 메모리, 예약 작업을 따로 갖는 에이전트 단위입니다[S10]. 이 구성을 예로 드는 이유는 에이전트가 열 개를 넘은 환경에서 호출 경로가 실제로 몇 갈래로 나뉘는지를 숫자로 보여 주기 때문입니다.

통제가 비는 업무는 키, 비용, 모델 교체 세 가지이며, 그 가운데 키에서 가장 먼저 문제가 드러납니다. 여러 소비자가 키 하나를 함께 쓰면 누가 얼마나 호출했는지 구분할 수 없고, 키가 유출되었을 때 모든 소비자의 키를 한꺼

번에 교체해야 합니다. 비용도 같은 이유로 소비자별로 나뉘지 않으며, LLM Gateway를 거치지 않는 호출은 합계에서조차 빠집니다. 모델 이름과 접속 주소가 호출 코드마다 적혀 있으면 모델 하나를 교체할 때 그 코드를 모두 찾아 고쳐야 합니다.

LLM Gateway를 설치하는 것만으로 세 가지가 모두 해결되지는 않으며, 오픈마루의 LiteLLM Proxy 구성에서도 이미 해결된 것은 프록시 설정 파일 한 곳의 수정으로 끝나는 모델 교체뿐입니다. 반면 모든 소비자가 마스터 키 하나를 공유하고 있고, 프록시를 거치지 않고 프로바이더를 직접 호출하는 경로도 3건 남아 있습니다. 이 사례는 LLM Gateway를 운영하는 조직에서도 호출 경로 조사가 먼저 필요하다는 점을 보여 주기 때문에 1장부터 다룹니다.

도입 검토는 제품 비교보다 자기 환경의 호출 경로를 적어 보는 작업에서 시작하는 편이 순서에 맞습니다. LLM Gateway가 없는 구성에서는 호출 주체마다 프로바이더에 직접 연결하고 키도 호출 주체마다 따로 보관합니다. 단일 길목 구성에서는 모든 호출 주체가 LiteLLM Proxy 하나로 들어오고 프로바이더 키는 프록시에만 남습니다. 이 장의 끝에는 두 구성 사이에서 자기 환경이 어디에 있는지 판정하는 데 쓰는 호출 경로 조사표 양식이 있습니다.

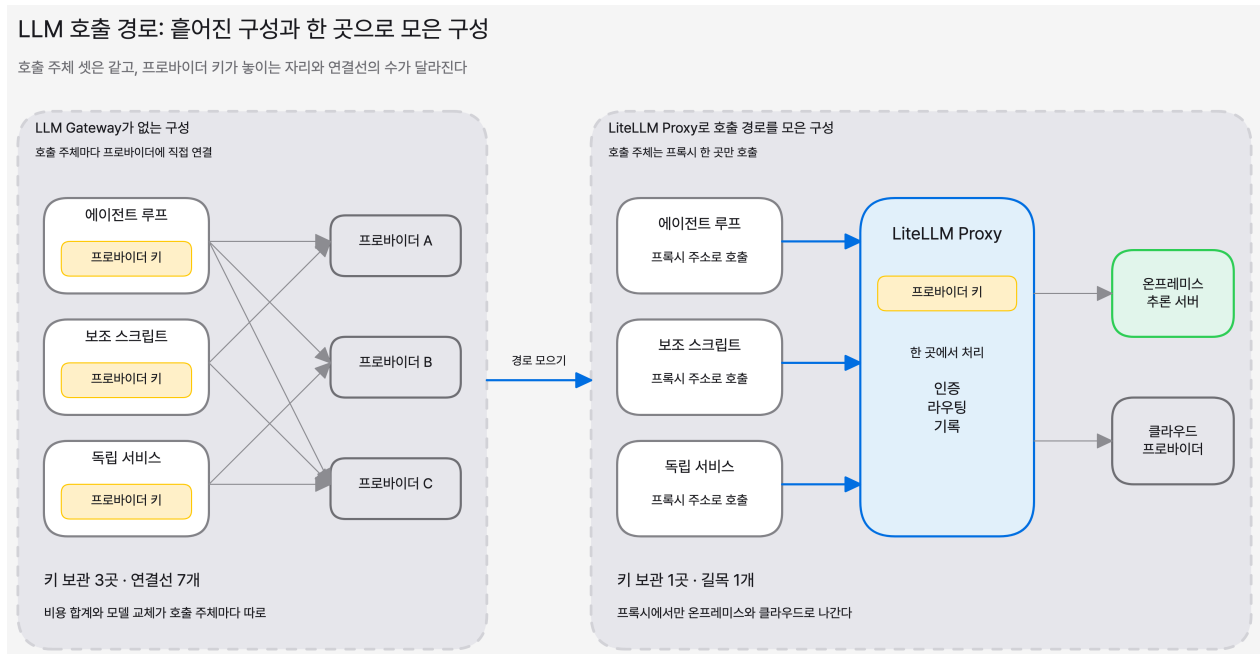


그림. LLM Gateway가 없는 구성과 LiteLLM Proxy로 호출 경로를 모은 구성의 대비

1.1 에이전트·스크립트·서비스로 흩어진 LLM 호출 경로의 세 형태

LLM을 호출하는 주체는 에이전트 루프, 에이전트 밖 보조 스크립트, 독립 서비스 세 형태로 나눌 수 있습니다. 에이전트 루프는 에이전트 프레임워크가 대화와 도구 호출을 반복하면서 모델을 호출하는 경우이고, 호출 설정은 프레임워크의 설정 파일에 있습니다. 보조 스크립트는 에이전트가 도구로 실행하거나 예약 작업이 실행하는 프로그램으로, 호출 코드와 키가 스크립트 안이나 실행 환경의 환경변수에 있습니다. 독립 서비스는 에이전트와 별개로 운영되는 검색, 제작, 관리 도구이며 서비스마다 자체 설정에 모델 주소와 키를 갖습니다.

세 형태를 구분하는 이유는 형태마다 호출 코드를 고치는 사람과 키가 보관되는 위치가 다르기 때문입니다. 에이전트 루프의 설정은 에이전트 운영자가 관리하지만 보조 스크립트는 그 스크립트를 작성한 개발자가, 독립 서비스는 서비스 담당자가 관리합니다. 에이전트 프레임워크의 모델 설정만 점검해서는 나머지 두 형태의 호출이 조

사에서 빠집니다. 호출 형식의 차이가 코드에 미치는 영향과 오픈마루 구성에서 세 형태가 나타나는 모습을 차례로 살펴보면 이 구분의 쓸모가 분명해집니다.

1.1.1 프로바이더 추가와 함께 늘어나는 분기 코드: 호출마다 100줄 안팎의 if/else

프로바이더를 직접 호출하는 코드는 프로바이더가 하나 늘 때마다 분기가 늘어납니다. 요청을 만드는 형식, 응답에서 결과를 꺼내는 형식, 오류를 알리는 예외의 종류가 프로바이더마다 다르기 때문입니다. LiteLLM을 만든 BerriAI의 창업자 Ishaan Jaffer는 자사 챗봇에 Azure와 Cohere를 추가하면서 이 문제를 겪었습니다. LLM API를 호출하는 곳마다 100줄 안팎의 if/else 분기가 여러 개 필요했고, 그 결과 디버깅해야 할 문제가 급격히 늘었다고 설명합니다[S08].

분기 코드의 부담은 호출 주체가 여럿일 때 더 커지는데, 애플리케이션 하나에서는 분기를 공통 함수로 묶어 관리할 수 있지만 에이전트와 스크립트와 서비스가 각자 프로바이더를 호출하면 같은 분기가 여러 코드베이스에 복제됩니다. 이때 프로바이더 한 곳이 요청 형식을 바꾸면 복제된 분기를 모두 찾아 고쳐야 하고, 한 곳이라도 빠지면 그 호출 주체에서만 오류가 납니다. 즉 호출 형식의 차이는 코드 분량의 문제로 끝나지 않고 장애 원인을 찾는 범위를 넓힙니다.

LiteLLM은 입출력 형식을 하나로 맞춰 이 분기를 없앴으며, 어느 프로바이더를 호출하든 요청은 OpenAI 형식으로 보내고, 응답은 같은 형식으로 받으며, 예외도 프로바이더와 무관하게 같은 종류로 돌려받습니다[S02][S08]. 다음 표는 프로바이더를 직접 호출할 때와 LiteLLM으로 형식을 맞췄을 때 호출 코드가 달라지는 세 부분을 나란히 놓은 것입니다.

구분	프로바이더를 직접 호출할 때	호출 코드에 생기는 일	LiteLLM으로 형식을 맞췄을 때
요청 형식	프로바이더마다 다름	프로바이더별로 요청을 만드는 분기	OpenAI 형식 하나
응답 형식	프로바이더마다 다름	응답에서 결과를 꺼내는 분기	프로바이더와 무관하게 같은 형식
예외	프로바이더마다 종류가 다름	오류 처리와 재시도를 나누는 분기	프로바이더와 무관하게 같은 종류

다만 형식을 맞추는 것과 호출 경로를 모으는 것은 서로 다른 일이어서, 형식만 맞춘 상태에서는 분기 코드가 사라져도 프로바이더 키가 여전히 호출 주체마다 남고, 누가 얼마나 호출했는지도 한 곳에서 볼 수 없습니다. 그래서 LiteLLM은 애플리케이션 안에 넣는 라이브러리인 LiteLLM SDK와 조직이 함께 쓰는 서버인 LiteLLM Proxy 두 형태로 제공됩니다[S02]. 형식 문제가 조직 단위의 통제 문제로 넓어진 과정은 2장의 주제입니다.

1.1.2 에이전트 루프, 보조 스크립트, 독립 서비스로 나뉘는 호출 주체

오픈마루의 구성에서 LiteLLM Proxy를 호출하는 소비자는 6종이고, 프록시를 거치지 않는 경로가 3건 있습니다. 이 구성을 미리 소개하는 이유는 LLM Gateway를 이미 운영하는 환경에서도 호출 주체가 세 형태에 고르게 퍼져 있다는 점을 확인할 수 있기 때문입니다. 에이전트 루프에 해당하는 소비자는 Hermes Agent 컨테이너 두 개이며, 업무 에이전트 스택과 콘텐츠 공장 스택으로 나뉩니다. 두 스택의 프로파일은 모두 OpenAI 호환 방식으로 LiteLLM Proxy 주소를 호출하도록 설정되어 있습니다.

보조 스크립트는 에이전트 컨테이너 안에 있지만 에이전트 루프 밖에서 모델을 호출합니다. PDF 읽기, 명함 인식, 이력서 평가, 콘텐츠 요약, 영상 첫 댓글 작성 스크립트가 여기에 속하고, 이들은 프록시를 직접 호출합니다. 독립 서비스로는 임베딩과 대화 호출을 함께 쓰는 슬라이드 검색 서비스, 영상 제작 도구 일부, 그리고 모델 목록 조회와 점검 호출만 하는 에이전트 콘솔이 있습니다. LiteLLM Proxy는 OpenAI 호환 클라이언트를 그대로 연결할 수 있는 엔드포인트를 제공하므로, 형태가 다른 소비자들도 같은 호출 방식으로 프록시에 연결됩니다 [S02].

다음 표는 호출 주체의 세 형태에 오픈마루의 소비자와 프록시 밖 경로를 대응시킨 것입니다.

호출 형태	LiteLLM Proxy를 거치는 소비자	프록시를 거치지 않는 경로
에이전트 루프	업무 에이전트 스택, 콘텐츠 공장 스택	없음
에이전트 밖 보조 스크립트	PDF 읽기, 명함 인식, 이력서 평가, 콘텐츠 요약, 영상 첫 댓글 스크립트	AI 검색 인용 주간 점검 스크립트, 위키 정리 스크립트
독립 서비스	에이전트 콘솔, 슬라이드 검색 서비스, 영상 제작 도구 일부	이미지 생성 백엔드 한 개

이 표에서 프록시 밖 경로 3건은 모두 에이전트 루프가 아닌 형태에 있습니다. 에이전트 프레임워크는 모델 설정이 한 곳에 있어 프록시로 옮기기 쉽지만, 스크립트와 서비스는 작성자가 각자 프로바이더 키를 넣어 만들기 때문에 조사하지 않으면 드러나지 않습니다. 따라서 에이전트 프레임워크만 프록시에 연결해서는 길목이 하나가 되지 않습니다. 모델 별칭과 풀백의 구성은 6장에서, 에이전트 스택의 배치와 요청 흐름은 7장에서 자세히 설명합니다.

세 형태 가운데 에이전트 루프는 호출 횟수를 미리 정하기 어렵다는 점에서 나머지 둘과 다릅니다. 보조 스크립트와 독립 서비스는 작성자가 코드에 적어 둔 횟수만큼 모델을 호출하지만, 에이전트는 목표를 받은 뒤 어떤 도구를 부르고 언제 멈출지를 스스로 정하면서 단계마다 모델을 다시 호출합니다. 사용자 메시지 한 건이 모델 호출 몇 건으로 이어질지는 실행해 봐야 알 수 있고, 호출량은 사용자 수보다 에이전트가 맡은 작업의 단계 수에 따라 달라집니다. 에이전트가 늘어날수록 호출을 한 곳에서 세고 제한하는 길목이 필요해지는 이유가 여기에 있습니다. AI 에이전트의 정의와 구조를 따로 다룬 글은 함께 읽을 자료 [R2]에 있습니다.

1.2 카·비용·모델 교체에서 드러나는 통제 공백

호출 경로가 흩어졌을 때의 문제는 사고가 날 가능성보다 지금 할 수 없는 일의 목록으로 적어야 영향 범위가 분명해집니다. 키 하나를 여러 소비자가 함께 쓰는 구성에서는 소비자별 지출 추적, 모델 접근 통제, 예산 설정 세 가지를 할 수 없습니다. LiteLLM Proxy에서 이 세 가지를 제공하는 기능이 가상 키입니다[S03]. 마스터 키는 프록시 관리자가 쓰는 키로 다른 키를 만드는 역할을 하고, 가상 키는 마스터 키로 소비자마다 따로 발급하는 키입니다[S03].

모델 교체에서는 수정해야 하는 설정 위치의 수가 통제 수준을 나타냅니다. 모델 이름과 접속 주소가 호출 코드마다 적혀 있으면 교체 작업의 크기가 호출 주체 수에 비례하고, 한 곳에 모여 있으면 수정 한 번으로 끝납니다. 반면 LLM Gateway 밖에서 프로바이더를 직접 호출하는 경로에는 키 통제도 모델 교체의 이점도 적용되지 않습니다. 다음 표는 통제 대상 세 가지에서 나타나는 증상과 그 상태에서 할 수 없는 일을 정리한 것입니다.

통제 대상	호출 경로가 흩어졌을 때의 증상	그 상태에서 할 수 없는 일
키	여러 소비자가 키 하나를 공유하거나 프로바이더 키가 스크립트마다 따로 있음	소비자 한 곳만 사용 중지, 유출된 키 만 골라 교체
비용	사용량이 소비자별로 나뉘지 않고, 프 록시 밖 호출은 합계에서 빠짐	소비자별 비용 귀속, 예산 한도 설정
모델 교체	모델 이름과 접속 주소가 호출 코드마 다 적혀 있음	설정 한 곳의 수정으로 모든 호출 주 체에 반영

1.2.1 마스터 키 하나를 공유할 때 가려지는 소비자별 사용량과 키 유출 범위

오픈마루의 소비자 6종은 모두 같은 마스터 키로 LiteLLM Proxy를 호출합니다. 이 구성을 드는 이유는 호출 경로를 프록시 한 곳으로 모은 뒤에도 키를 나누지 않으면 무엇이 보이지 않는지를 구체적으로 확인할 수 있기 때문입니다. 프록시는 사흘 동안 대화 완성 요청 약 2,500건을 처리했지만, 요청에 붙은 키가 모두 같아서 그 가운데 어느 에이전트가 몇 건을 호출했는지는 프록시에서 구분할 수 없습니다. 게다가 컨테이너 로그에는 요청별 모델 별칭이 남지 않아 어느 모델에 요청이 물렸는지도 프록시 쪽 기록만으로는 알 수 없습니다.

키 유출에 대응하는 범위도 같은 이유로 넓어져, 마스터 키가 유출되면 에이전트 스택 두 개, 보조 스크립트, 독립 서비스가 쓰는 키를 모두 교체해야 하고, 교체하는 동안 모든 소비자의 호출이 영향을 받습니다. 마스터 키는 다른 키를 만드는 관리자 키 역할을 합니다[S03]. 따라서 호출용으로 여러 소비자에 나누어 두기에는 권한이 지나치게 큰 키입니다. 소비자별 키가 있다면 유출된 키 하나만 폐기하고 나머지 소비자는 그대로 운영할 수 있습니다.

이 공백에 대응하는 기능이 가상 키이며, LiteLLM Proxy의 가상 키는 키 단위 지출 추적, 키별로 호출할 수 있는 모델을 제한하는 접근 통제, 키·사용자·팀 단위 예산 세 가지를 제공합니다[S03]. 이 기능들은 별도 계약이 필요한 유료판 전용 기능에 속하지 않고 누구나 쓸 수 있는 공개판에 포함되어 있습니다[S04]. 다만 가상 키를 쓰려면 Postgres 데이터베이스를 프록시에 연결해야 합니다[S03]. 오픈마루의 구성은 데이터베이스 없이 설정 파일 한 개로 운영하므로 가상 키를 쓰지 않습니다. 다음 표는 두 구성에서 가능한 기능을 대조한 것입니다.

기능	마스터 키 공유 구성	소비자별 가상 키 구성
프록시 인증	가능	가능
소비자별 지출 추적	불가	가능
키별 모델 접근 통제	불가	가능
키·사용자·팀 단위 예산	불가	가능
키 유출 때 교체 범위	모든 소비자	해당 키를 쓰는 소비자
필요한 구성 요소	설정 파일	설정 파일과 Postgres

비용 귀속과 사고 대응 범위는 모두 소비자를 식별하는 수단이 있어야 좁힐 수 있습니다. 모니터링도 마찬가지여서 LiteLLM Proxy의 키·팀·사용자별 비용 지표와 예산 지표는 가상 키가 있어야 값이 나뉩니다[S05]. 키 분리

는 보안 조치이면서 비용 관리와 관측의 전제 조건입니다. 9장의 다음 단계는 Postgres 연결과 에이전트·서비스별 가상 키 발급을 이 문제의 해법으로 다룹니다.

1.2.2 모델 교체 범위의 차이: 호출 지점 전체 수정과 별칭 정의 한 곳 수정

LLM Gateway가 없는 구성에서 모델을 교체하려면 모델 이름과 접속 주소가 적힌 설정을 호출 주체마다 고쳐야 합니다. 에이전트 프레임워크의 설정 파일, 보조 스크립트의 코드나 환경변수, 독립 서비스의 설정이 모두 수정 대상입니다. 수정 대상이 여러 곳이면 일부만 바뀐 상태가 생기고, 그동안 호출 주체마다 다른 모델이 응답합니다. 교체 작업에는 수정뿐 아니라 빠진 곳이 없는지 확인하는 작업이 함께 따릅니다.

LiteLLM Proxy는 이 수정 위치를 별칭 정의 한 곳으로 줄이며, 여기서 별칭은 호출 주체가 요청에 적는 모델 이름을 실제 백엔드에 연결하는 프록시 설정 항목입니다. 오픈마루의 에이전트 프로파일 16개는 모두 같은 기본 별칭과 같은 프록시 주소를 가리키도록 설정되어 있습니다. 기본 대화 모델을 다른 모델로 교체할 때 프로파일 16개의 설정은 그대로 두고 프록시 설정 파일의 별칭 정의만 수정합니다. 이 사례는 키 분리나 지표 수집을 하기 전 단계에서도 길목을 하나로 모으면 얻는 이점을 보여 주기 때문에 문제 정의와 함께 소개합니다.

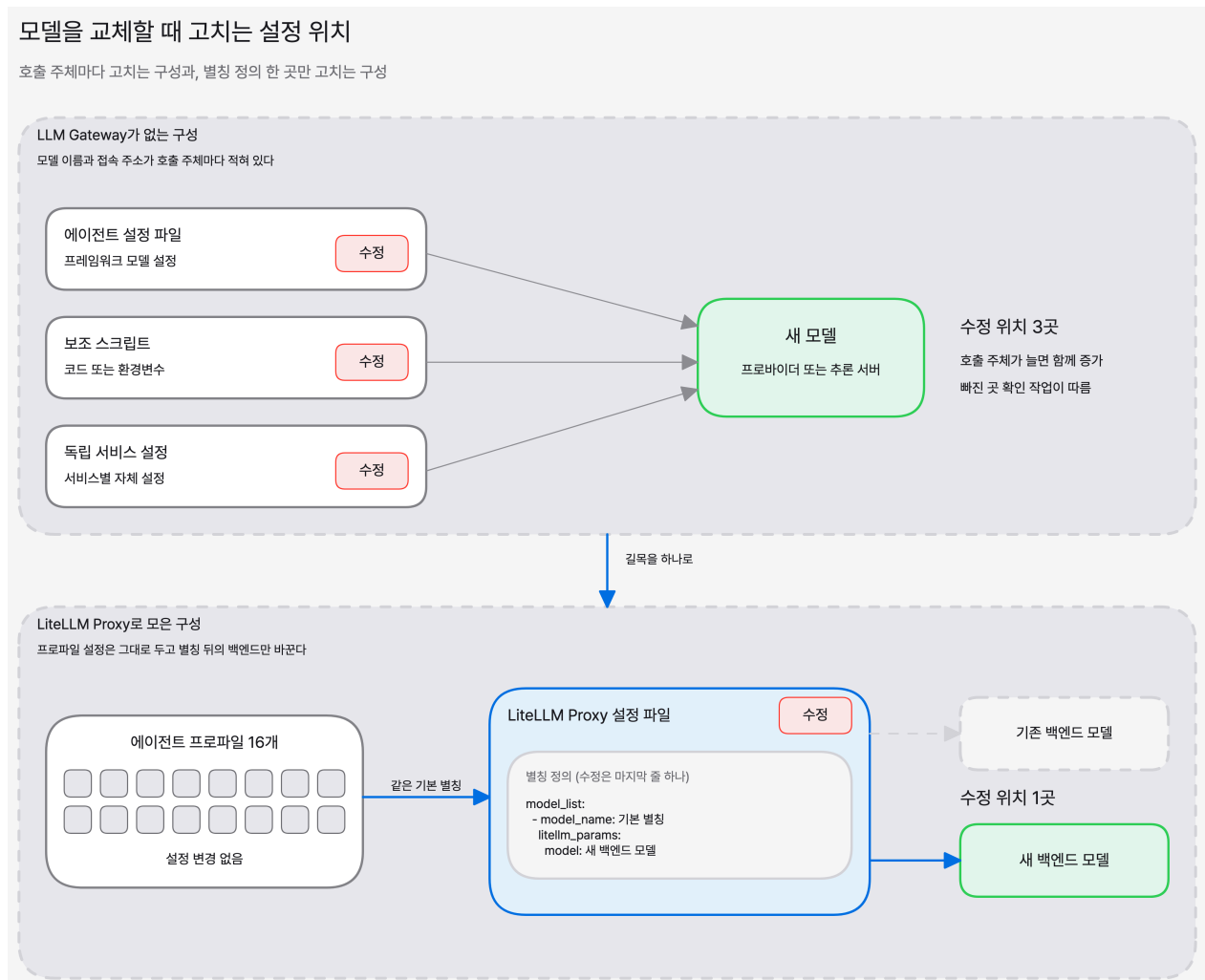


그림. 모델을 교체할 때 수정하는 설정 위치의 수 대비

별칭은 모델 교체뿐 아니라 장애 대응의 단위이기도 합니다. 폴백은 한 모델이 응답하지 않을 때 요청을 다른 모델로 넘기는 동작이며, LiteLLM Proxy에서는 어느 별칭이 실패하면 어느 별칭으로 넘길지를 목록으로 정의합

니다[S06]. 폴백, 재시도, 쿨다운은 데이터베이스 없이 설정 파일만으로 동작합니다[S06]. 오픈마루의 구성에는 폴백 3건이 정의되어 있어 온프레미스 추론 서버가 응답하지 않으면 요청이 다른 모델로 넘어가고, 이 동작도 에이전트 쪽 설정을 바꾸지 않고 프록시 설정 파일에서 관리합니다.

모델 교체는 조직이 정한 시점에만 일어나지 않습니다. 클라우드 프로바이더는 자체 일정에 따라 모델을 새 버전으로 바꾸거나 제공을 끝내며, 그때 호출 지점마다 모델 이름이 적혀 있으면 조직이 정하지 않은 일정에 맞춰 호출하는 코드와 설정을 모두 고쳐야 합니다. 별칭을 쓰는 구성에서는 이런 변경도 별칭 정의에서 처리하므로, 소비자가 부르는 이름은 그대로 두고 그 뒤에 연결된 프로바이더 모델만 바꿉니다. 온프레미스 모델을 교체할 때와 같은 절차가 외부 요인으로 생긴 교체에도 그대로 적용됩니다. 오픈마루의 구성을 예로 들면 프로바이더의 일정에 영향을 받는 경로는 클라우드 별칭 하나이고, 고칠 위치도 그 별칭의 정의 한 곳입니다.

1.2.3 게이트웨이 밖 호출의 통제·집계 공백: 프로바이더 직접 호출 스크립트

LiteLLM Proxy를 거치지 않는 호출은 프록시가 제공하는 통제와 집계의 대상에서 빠지므로, 그 호출에는 프록시의 인증이 적용되지 않고, 별칭과 폴백 정의도 쓰이지 않으며, 프록시 쪽 기록에도 남지 않습니다. 프로바이더 키도 스크립트의 실행 환경에 따로 보관되므로 키 관리 대상이 프록시 밖에 하나씩 더 생깁니다. 즉 LLM Gateway를 운영하더라도 프록시 밖 경로가 남아 있으면 그만큼 통제 범위가 줄어듭니다.

오픈마루의 구성에는 이런 경로가 3건 있습니다. 이 3건을 밝히는 이유는 프록시를 운영하는 조직에서도 프록시 경유 여부를 따로 조사해야 한다는 판단의 근거가 되기 때문입니다. AI 검색 인용을 주간으로 점검하는 스크립트는 클라우드 프로바이더 세 곳의 API를 직접 호출하고, 위키 문서를 정리하는 스크립트도 클라우드 프로바이더 API를 직접 호출합니다. 이미지 생성 백엔드 한 개는 여러 모델을 중개하는 외부 서비스를 직접 호출합니다. 다음 표는 프록시를 거치는 경로와 거치지 않는 경로를 구분한 것입니다.

구분	호출 주체	호출 대상	프록시의 통제와 집계
프록시 경유	에이전트 스택 두 개, 에이전트 콘솔	LiteLLM Proxy의 모델 별칭	적용
프록시 경유	보조 스크립트 5종, 슬라이드 검색 서비스, 영상 제작 도구 일부	LiteLLM Proxy의 모델 별칭	적용
프록시 비경유	AI 검색 인용 주간 점검 스크립트	클라우드 프로바이더 세 곳의 API	적용되지 않음
프록시 비경유	위키 정리 스크립트	클라우드 프로바이더 API	적용되지 않음
프록시 비경유	이미지 생성 백엔드 한 개	모델 중개 서비스	적용되지 않음

프록시 밖 경로는 프록시 설정 파일에 나타나지 않습니다. 그래서 LLM Gateway의 설정 파일만 검토해서는 찾을 수 없고, 프로바이더 키가 보관된 위치와 프로바이더 API 주소를 호출하는 코드를 따로 조사해야 합니다. 반면 음성 인식처럼 LLM을 호출하지 않는 처리는 조사 대상에서 제외합니다. 오픈마루의 구성에서 음성 메시지는 호스트의 로컬 음성 인식 서버가 처리하므로 LLM 프록시와 무관합니다.

호출 경로를 모으면 조직 밖으로 나가는 요청의 목록도 함께 정리됩니다. 프록시 설정 파일의 별칭 정의에는 각 별칭이 온프레미스 추론 서버로 가는지 클라우드 프로바이더로 가는지 적혀 있으므로, 이 파일이 곧 외부로 향하는 경로의 목록이 됩니다. 오픈마루의 구성에서 프록시를 거치는 클라우드 경로는 별칭 9개 가운데 하나뿐입

니다. 이 구성을 예로 드는 이유는 어떤 요청이 조직 밖으로 나가는지를 설정 파일 한 개로 답할 수 있는 범위와 답할 수 없는 범위를 함께 보여 주기 때문입니다.

프록시 밖 경로 3건은 모두 외부 API로 향하는 호출입니다. 외부로 나가는 경로를 세어 보면 프록시 안에 한 건, 프록시 밖에 세 건이 있으므로, 프록시 설정 파일만으로는 외부 전송 경로의 일부만 설명할 수 있습니다. 데이터 등급을 판정하거나 감사 질의에 답해야 하는 조직에서는 이 차이가 그대로 답변에서 빠지는 부분이 됩니다. 외부로 나가는 경로가 몇 건인지부터 프록시 설정 파일과 호출 경로 조사표를 함께 놓고 세어야 합니다. LLM Gateway는 호출을 중계하는 구성 요소인 동시에 무엇이 조직 밖으로 나가는지를 한 곳에서 보여 주는 근거 자료이며, 이 역할은 프록시 밖 경로를 줄일수록 완전해집니다.

1.3 호출 경로 조사 항목: 소비자, 키, 프록시 경유 여부

호출 경로를 LiteLLM Proxy 한 곳으로 모으려면 먼저 자기 환경에서 LLM을 호출하는 주체를 빠짐없이 적어야 합니다. 조사할 항목은 소비자 목록, 소비자가 쓰는 키, 프록시 경유 여부 세 가지입니다. 소비자 목록은 에이전트 루프, 보조 스크립트, 독립 서비스 세 형태로 나누어 작성하면 에이전트 밖의 호출이 빠지지 않습니다. 키 항목에는 키 값을 제외하고 키의 종류와 다른 소비자와의 공유 여부만 적습니다.

세 항목이 채워지면 조치의 우선순위를 정할 수 있는데, 프록시를 거치지 않는 소비자는 경로를 옮기는 대상이 되고, 키를 공유하는 소비자는 가상 키 발급 대상이 됩니다. 조사표는 도입 검토의 첫 산출물이며, 9장이 제시하는 다음 단계의 첫 단계도 이 조사입니다. LLM Gateway를 아직 운영하지 않는 조직도 같은 표를 쓸 수 있고, 그 경우 프록시 경유 여부 열은 모두 비경유로 채워집니다.

조사 범위는 LLM 호출로 한정하되, 임베딩 호출도 대화 호출과 같은 프록시와 같은 키를 거칠 수 있으므로 조사 대상에 포함합니다. 반면 LLM을 사용하지 않는 예약 작업은 제외합니다. 예를 들어 오픈마루의 예약 작업 94건 가운데 30건은 스크립트만 실행하고 LLM을 호출하지 않으므로 조사표에 적을 대상이 아닙니다. 이 수치를 드는 이유는 예약 작업 수와 LLM 호출 주체 수가 같지 않다는 점을 보여 주기 때문입니다.

1.3.1 LLM 호출 주체별 조사표: 소비자, 사용 키, 대상 모델, 프록시 경유 여부

조사표는 호출 주체 하나를 한 행으로 적고 다섯 개 열을 채우는 양식이며, 각 열은 뒤에서 내릴 결정 하나에 대응하므로 열을 줄이지 않는 편이 좋습니다. 소비자 열과 사용 키 열은 가상 키를 몇 개 발급할지를 정하는 근거가 되고, 대상 모델 열은 프록시에 등록할 모델 별칭의 목록이 됩니다. 프록시 경유 여부 열은 경로를 옮길 대상, 호출 빈도 열은 옮기는 순서를 정하는 데 씁니다. 다음 표는 열마다 적는 내용과 그 열을 쓰는 결정을 정리한 조사표 양식입니다.

열	적는 내용	이 열로 정하는 것
소비자	호출 주체의 역할 이름과 호출 형태 (에이전트 루프, 보조 스크립트, 독립 서비스)	가상 키를 발급할 단위
사용 키	키의 종류(프록시 마스터 키, 프록시 가상 키, 프로바이더 키)와 다른 소비자와의 공유 여부	키 분리 대상과 유출 때 교체 범위

열	적는 내용	이 열로 정하는 것
대상 모델	호출하는 모델 또는 모델 별칭, 대화·임베딩 구분	프록시에 등록할 별칭과 키별 모델 접근 범위
프록시 경유 여부	경유 또는 비경유, 비경유이면 직접 호출하는 프로바이더	프록시 경유로 옮길 경로
호출 빈도의 규모감	하루 또는 일주일 단위의 대략적인 요청 수, 예약 실행 여부	옮기는 순서와 예산 한도의 출발값

사용 키 열을 적을 때는 키 값을 조사표에 옮기지 않는데, 조사표는 여러 사람이 함께 보는 문서이므로 키의 종류와 보관 위치만 적어도 판단에 충분합니다. 호출 빈도는 정확한 수치가 아니어도 됩니다. 소비자별 사용량을 프록시에서 나누어 집계하려면 가상 키가 있어야 하는데[S03], 조사는 가상 키를 발급하기 전에 하는 작업이기 때문입니다. 그래서 이 열에는 예약 작업의 실행 주기나 에이전트의 세션 수처럼 이미 알고 있는 값으로 규모를 적습니다.

대상 모델 열에는 호출하는 모델과 함께 그 소비자가 맡은 일을 한 줄로 적어 두면 좋습니다. 에이전트는 받은 권한의 범위에 따라 필요한 통제가 달라지므로, 문서를 읽기만 하는 소비자와 외부 시스템에 쓰는 소비자는 같은 양식에 적더라도 맡은 일을 구분해 두어야 합니다. 이 한 줄은 가상 키를 발급할 때 키마다 허용할 모델을 고르는 근거가 됩니다. 예를 들어 임베딩만 호출하는 검색 서비스의 키에는 임베딩 별칭만 허용하고, 비전 작업을 맡은 스크립트의 키에만 클라우드 별칭을 허용하는 식입니다.

이 양식으로 오픈마루의 구성을 조사한 결과 가운데 소비자 목록과 호출 형태는 6장의 소비자 구성에 있고, 조사에서 나온 소비자 6종과 프록시 밖 경로 3건을 다음 단계의 입력으로 쓰는 방법은 9장에 있습니다. 양식과 결과를 나누어 실는 이유는 독자가 자기 환경의 표를 먼저 작성한 뒤 다른 조직의 결과와 비교하는 순서가 판단에 더 유용하기 때문입니다. 조사표의 행 수가 예상보다 많다면, 특히 보조 스크립트와 독립 서비스의 행이 에이전트의 행보다 많다면 호출 경로가 이미 흩어져 있다는 신호로 볼 수 있습니다.

1.3.2 문제 정의에서 운영 구성과 다음 단계까지 9개 장의 읽는 순서

2장부터 9장까지는 조사표를 채운 뒤에 내릴 결정의 근거를 순서대로 제공합니다. 2장과 3장은 LiteLLM이 만들어진 배경과 LiteLLM SDK, LiteLLM Proxy 두 형태의 차이를 설명합니다[S02]. 4장과 5장은 LLM Gateway의 요건과 대안을 다루며, 공개판과 유료판의 기능 경계도 4장에 있습니다[S04]. 6장부터 8장까지는 오픈마루의 운영 구성과 모니터링을, 9장은 한계와 다음 단계를 다룹니다.

LiteLLM Proxy의 채택 여부는 리스크 수용 가능성, 운영 부담, 대안 대비 우위 세 가지 기준으로 판단할 수 있습니다. 리스크 수용 가능성은 프록시가 멈추면 모든 소비자가 멈춘다는 점과 2026년 3월 LiteLLM의 PyPI 패키지에서 일어난 공급망 사고 같은 위험을 조직이 감당할 수 있는지를 묻습니다[S07]. 운영 부담은 Postgres 연결과 지표 수집처럼 통제 수준을 높일 때 함께 늘어나는 운영 대상을 가리킵니다. 대안 대비 우위는 다른 오픈소스 LLM Gateway나 국내 공급 솔루션과 비교했을 때 LiteLLM Proxy가 자기 조직의 조건에 더 맞는지를 확인하는 기준입니다.

다음 표는 장마다 답하는 질문과 관련된 판단 기준을 대응시킨 것입니다. 엔지니어는 구성과 설정을 다루는 3장, 6장, 7장, 8장을 먼저 읽고, 의사결정권자는 요건과 대안, 한계를 다루는 4장, 5장, 9장을 먼저 읽어도 내용이 이어집니다.

장	답하는 질문	관련 판단 기준
1장	LLM Gateway가 없을 때 키, 비용, 모델 교체에서 막히는 일	조사 항목
2장	프로바이더마다 다른 호출 형식의 문제와 LiteLLM의 풀이	대안 대비 우위
3장	LiteLLM SDK와 LiteLLM Proxy를 쓰는 주체와 시점	운영 부담
4장	LLM Gateway의 기본 요건, 상세 요건과 LiteLLM Proxy의 충족 범위	대안 대비 우위
5장	오픈소스 LLM Gateway 상위 5개의 선정 기준과 국내 공급 유형	대안 대비 우위
6장	모델 별칭, 폴백, 온프레미스 추론 서버와 클라우드 경로의 구성	운영 부담
7장	에이전트 스택의 구분과 요청이 모델에 닿는 경로	운영 부담
8장	모니터링 대상과 현재 구성에서 비어 있는 지표	리스크 수용 가능성
9장	키 분리, 지표 수집, 우회 경로 정리의 순서	리스크 수용 가능성

8장과 9장은 1장에서 정의한 통제 공백이 관측과 조치로 이어지는 부분입니다. 예를 들어 첫 토큰 시간(TTFT, Time To First Token)은 요청을 보낸 뒤 응답의 첫 토큰이 도착하기까지 걸린 시간이며, LiteLLM Proxy는 이 값과 폴백 발생 횟수를 Prometheus 지표로 제공합니다[S05]. 오픈마루의 구성에서는 지표 수집이 켜져 있지 않으므로 이 값들을 현재 볼 수 없고, 9장은 호출 경로 조사 다음에 지표 수집과 키 분리를 어떤 순서로 진행할지를 의존 관계에 따라 제시합니다. 조사표를 작성한 독자는 9장의 첫 단계를 이미 마친 상태에서 나머지 단계를 검토하게 됩니다.

2장: LiteLLM 이 만들어진 이유와 LiteLLM Proxy 가 필요해진 배경

LiteLLM은 BerriAI가 2023년에 공개한 오픈소스 프로젝트이며, 100개가 넘는 LLM 프로바이더(모델 API를 제공하는 사업자)를 OpenAI 형식 하나로 호출하게 해 줍니다[S01]. 창업자가 밝힌 출발점은 챗봇에 프로바이더를 추가하는 작업이었으며, 프로바이더가 늘 때마다 호출 코드에 프로바이더별 분기가 늘어 디버깅 부담이 커졌습니다[S08].

AI 에이전트를 운영하는 조직은 온프레미스 추론 서버와 클라우드 모델을 함께 쓰고 더 나은 모델이 나오면 기본 모델을 교체하므로 같은 문제를 더 큰 규모로 겪습니다. 호출 형식이 프로바이더마다 다르면 모델을 바꿀 때마다 호출 코드를 고쳐야 하고, 모델을 호출하는 에이전트와 스크립트가 늘어나면 키와 사용 기록을 호출 주체마다 따로 관리해야 합니다.

두 문제는 원인이 서로 다르므로 해결 수단도 다르며, 호출 형식의 차이는 애플리케이션 안에 넣는 라이브러리인 LiteLLM SDK가 해결합니다. 반면 키 보관, 비용 귀속, 모델 접근 통제는 호출 주체가 여럿일 때 생기는 조직 단위의 문제이므로, 여러 호출 주체가 함께 거치는 서버인 LiteLLM Proxy가 필요합니다[S02].

LiteLLM을 호출 형식 변환 도구로만 이해하면 LiteLLM Proxy를 두는 이유를 놓치게 됩니다. 이 장에서는 프로바이더별 형식 차이가 호출 코드의 어느 부분에 부담을 주었는지, LiteLLM이 그 가운데 무엇을 통일했는지, 형식을 통일한 뒤에도 남는 문제가 왜 게이트웨이 서버를 요구하는지를 순서대로 확인할 수 있습니다.

2.1 프로바이더마다 다른 호출 형식과 100줄 안팎 if/else 분기의 누적

LLM API는 프로바이더마다 요청 파라미터, 응답 구조, 오류 표현이 다르게 설계되어 있습니다. 애플리케이션이 프로바이더 한 곳만 호출하는 동안에는 이 차이가 드러나지 않지만, 두 번째 프로바이더를 추가하면 호출 코드에 프로바이더를 가르는 분기가 생깁니다. LiteLLM 창업자는 LLM API를 호출할 때마다 100줄 안팎의 if/else 분기가 여러 개 필요했다고 설명했습니다[S08].

이 분기는 요청을 만드는 코드, 응답을 읽는 코드, 오류를 처리하는 코드에 각각 생기므로 프로바이더가 하나 늘어나면 수정할 코드도 세 층에서 함께 늘어납니다. 형식 차이를 요청·응답·예외 세 층으로 나누어 보면 LiteLLM이 통일한 대상이 무엇인지 정확히 구분할 수 있습니다.

2.1.1 요청·응답·예외 세 층에서 발생하는 프로바이더별 형식 차이

요청 층의 차이는 같은 대화 완성 요청을 프로바이더마다 다른 형식으로 작성해야 한다는 데서 나옵니다. 예를 들어 파라미터의 이름, 대화 메시지를 담는 구조, 인증 정보를 전달하는 방식이 프로바이더에 따라 다를 수 있습니다. 한 프로바이더가 받는 파라미터를 다른 프로바이더가 받지 않는 경우도 있으므로, 호출 코드는 프로바이더별로 요청을 다시 조립해야 합니다.

응답 층에서는 응답의 구조가 프로바이더마다 다르므로 결과를 읽는 코드가 프로바이더 수만큼 필요합니다. 예를 들어 생성된 문장이 응답 본문의 어느 필드에 들어 있는지, 토큰 사용량을 어떤 이름으로 돌려주는지가 프로바이더에 따라 다를 수 있습니다. 그 경우 화면에 답을 표시하는 코드뿐 아니라 사용량을 집계하는 코드까지 프로바이더별로 따로 유지해야 합니다.

예외 층의 차이는 장애 대응 코드에 영향을 줍니다. 속도 제한 초과, 인증 실패, 시간 초과, 컨텍스트 길이 초과 같은 상황은 프로바이더를 가리지 않고 일어날 수 있지만, 이를 알리는 오류의 종류는 프로바이더마다 다릅니다.

잠시 뒤에 다시 시도할 오류와 즉시 중단할 오류를 구분하는 판단 코드 역시 프로바이더 수만큼 작성해야 합니다. 아래 표는 세 층에서 차이가 생기는 항목의 예와 그 차이가 호출 코드에 주는 부담을 정리한 것입니다.

층	프로바이더마다 달라질 수 있는 항목의 예	호출 코드에 생기는 부담
요청	파라미터 이름, 메시지 구조, 인증 방식	프로바이더별 요청 조립 코드
응답	결과 문장의 필드 위치, 토큰 사용량 필드	프로바이더별 응답 해석 코드와 사용량 집계 코드
예외	속도 제한, 인증 실패, 시간 초과, 컨텍스트 초과를 알리는 오류의 종류	프로바이더별 재시도·중단 판단 코드

이 구조에서는 모델을 다른 프로바이더의 모델로 교체하는 작업이 설정 변경으로 끝나지 않습니다. 요청 조립, 응답 해석, 오류 판단 코드를 모두 수정하고 다시 시험해야 하기 때문입니다. 즉 프로바이더별 형식 차이는 모델 교체 비용이 커지는 근본 원인이며, LiteLLM이 처음 해결하려 한 대상도 이 세 층의 차이였습니다[S02][S08].

2.1.2 BerriAI의 2023년 출발: Azure와 Cohere 추가에서 드러난 디버깅 부담

LiteLLM을 만든 회사는 BerriAI입니다. BerriAI는 2023년에 창업해 Y Combinator Winter 2023 배치에 참여했고, 창업자는 Krrish Dholakia와 Ishaan Jaffer입니다[S08]. BerriAI는 2023년 7월 27일에 공개 저장소를 만들었고, 저장소의 LICENSE 파일도 2023년 Berri AI를 저작권자로 적고 있습니다[S01][S11].

LiteLLM의 출발점은 챗봇에 Azure와 Cohere를 추가하던 작업이었습니다. 창업자 Ishaan Jaffer는 그 과정에서 LLM API 호출마다 100줄 안팎의 if/else 분기가 여러 개 필요해졌고, 이 때문에 디버깅 문제가 급격히 늘었다고 설명했습니다[S08]. 프로바이더를 한 곳 더할 때마다 요청, 응답, 예외를 처리하는 코드가 각각 한 벌씩 더 필요해지기 때문입니다.

분기가 디버깅 부담으로 이어지는 이유는 오류의 원인 범위가 넓어지기 때문입니다. 응답이 비정상일 때 원인이 모델인지, 프로바이더별 요청 조립 코드인지, 응답 해석 코드인지를 먼저 가려내야 하고, 이 확인을 프로바이더별 분기마다 반복해야 합니다. 또한 한 프로바이더용 분기만 수정하고 다른 분기를 그대로 두면 같은 기능이 프로바이더에 따라 다르게 동작하는 결함이 남습니다.

아래 표는 창업부터 최신 릴리스까지 LiteLLM의 연혁을 정리한 것입니다. 스타와 포크 수는 2026년 10월 6일 GitHub API 조회값입니다[S11].

시기	내용
2023년	BerriAI 창업, Y Combinator Winter 2023 배치 참여
2023년 7월 27일	공개 저장소 생성
2026년 10월 3일	최신 릴리스 v1.104.0
2026년 10월 6일	저장소 스타 60,173개, 포크 12,046개

LiteLLM이 챗봇을 만들면서 실제로 겪은 디버깅 문제에서 출발했다는 사실은 이 도구의 설계 방향을 설명해 줍니다. LiteLLM은 새로운 호출 규격을 만드는 대신 OpenAI 클라이언트를 쓰던 코드가 그대로 동작하도록 OpenAI 형식에 다른 프로바이더를 맞추는 방식을 택했으므로, 기존 코드에서 호출 부분만 교체해도 프로바이더별 분기를 줄일 수 있습니다[S02].

2.2 OpenAI 형식 하나로 맞춘 입력·응답·예외와 100개 이상 프로바이더 지원

LiteLLM은 입출력 형식을 하나로 맞춰 프로바이더별 분기를 없애는 방법으로 이 문제를 해결했습니다[S08]. 100개가 넘는 프로바이더를 OpenAI 형식으로 호출하고, 어느 프로바이더를 쓰든 응답 형식이 같으며, 예외도 프로바이더와 무관하게 같은 종류로 반환합니다[S02]. 앞에서 구분한 요청·응답·예외 세 층이 모두 통일 대상에 들어 있습니다.

다만 형식 통일이 해결하는 범위는 호출 코드까지입니다. 프로바이더 키를 어디에 보관할지, 사용량을 누구에게 귀속할지, 어떤 호출 주체에 어떤 모델을 허용할지는 형식이 같아져도 그대로 남습니다. 형식 통일로 얻는 것과 얻지 못하는 것을 구분해 두면 LiteLLM Proxy가 추가로 맡는 역할이 분명해집니다.

2.2.1 OpenAI 호환 호출 형식으로 통일한 100개 이상 프로바이더의 요청과 응답

LiteLLM SDK를 쓰는 애플리케이션은 어느 프로바이더를 호출하든 OpenAI 형식의 요청을 작성하고 OpenAI 형식의 응답을 받습니다. 공식 문서는 LiteLLM SDK를 파이썬 애플리케이션에 직접 통합하는 라이브러리이자 OpenAI 클라이언트를 그대로 대체할 수 있는 구성 요소로 소개합니다[S02]. OpenAI 클라이언트로 작성한 기존 코드는 호출 구조를 유지한 채 다른 프로바이더의 모델을 사용할 수 있습니다.

형식 통일의 직접적인 효과는 호출 코드를 한 벌로 유지할 수 있다는 것입니다. 요청 조립과 응답 해석이 프로바이더와 무관해지므로 운영자는 모델 선택을 코드 변경 없이 설정 변경으로 처리합니다. 지원 범위는 100개 이상의 프로바이더이며, 저장소 README도 LiteLLM을 여러 프로바이더를 단일 인터페이스로 호출하는 오픈소스 AI 게이트웨이로 소개합니다[S01].

프로바이더별 호출 형식이 OpenAI 형식 하나로 통일되는 구조

요청, 응답, 예외 세 층이 모두 통일 대상이다

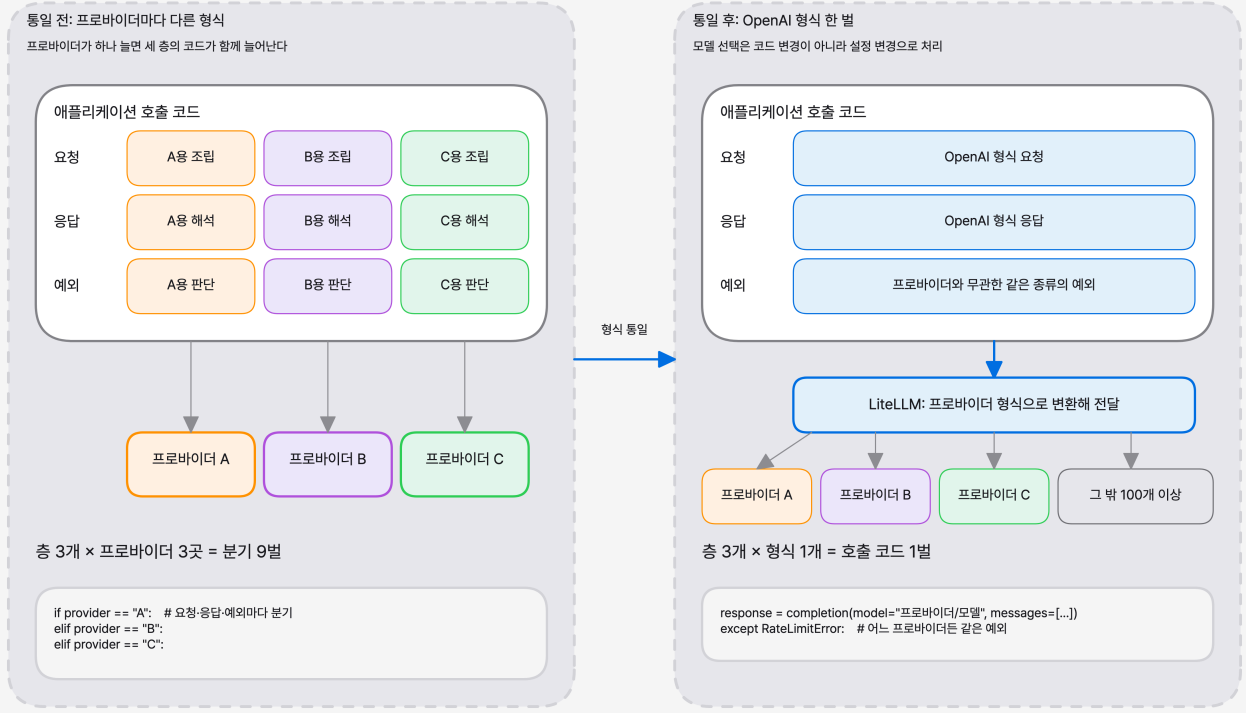


그림. 프로바이더별 호출 형식이 OpenAI 형식 하나로 통일되는 구조

형식을 통일하더라도 프로바이더가 받지 않는 파라미터는 여전히 존재하므로 호환 처리가 필요합니다. 오픈마루의 LiteLLM Proxy 구성을 예로 드는 이유는 운영 환경에서 이 호환 처리를 설정 항목 하나로 다루고 있기 때문입니다. 오픈마루는 프로바이더가 받지 않는 파라미터를 요청에서 제거하도록 설정해 두었고, 그 덕분에 에이전트는 백엔드가 온프레미스 추론 서버인지 클라우드 모델인지 구분하지 않고 같은 요청을 보냅니다.

형식이 같아진다고 모델의 능력까지 같아지는 것은 아닙니다. 오픈마루의 구성을 다시 예로 드는 이유는 형식 통일 뒤에도 모델 선택이 운영자의 일로 남는다는 점을 보여 주기 때문입니다. 오픈마루는 이미지가 포함된 PDF를 읽는 비전 작업에 클라우드 모델로 연결되는 별칭을 따로 지정해 두었습니다. 여기서 별칭은 호출 주체가 요청에 적는 모델 이름을 실제 백엔드에 연결하는 LiteLLM Proxy의 설정 항목입니다. 즉 형식 통일은 호출 방법을 하나로 만들 뿐이며, 어떤 작업에 어떤 모델을 쓸지는 운영자가 별도로 정해야 합니다.

2.2.2 프로바이더와 무관하게 같은 종류로 반환되는 예외와 재시도 판단

LiteLLM은 예외를 프로바이더와 무관하게 같은 종류로 반환합니다[S02]. 속도 제한 초과나 시간 초과가 어느 프로바이더에서 발생했든 호출 코드는 같은 종류의 예외를 받으므로, 재시도 여부를 판단하는 코드를 프로바이더별로 따로 작성할 필요가 없습니다. 예외 층에 있던 프로바이더별 분기는 이 방식으로 없어집니다.

예외 통일은 재시도와 폴백(fallback, 한 모델이 실패했을 때 다른 모델로 요청을 전환하는 동작)이 성립하는 전제이기도 합니다. LiteLLM의 안정성 설정에는 재시도 횟수, 허용 실패 횟수, 실패한 배포를 호출 대상에서 제외하는 쿨다운 시간이 있고, 폴백 설정에는 일반 폴백과 별도로 컨텍스트 초과와 콘텐츠 정책 위반에 각각 대응하는 항목이 있습니다[S06]. 이러한 설정은 오류의 종류를 프로바이더와 무관하게 구분할 수 있을 때 의미를 갖습니다.

오픈마루의 구성을 예로 드는 이유는 예외 통일을 전제로 한 설정이 서로 다른 백엔드 사이에서 어떻게 쓰이는지 보여 주기 때문입니다. 이 구성은 재시도 1회와 쿨다운 300초를 설정하고, 온프레미스 추론 서버의 Qwen 계열 27B급 모델이 응답하지 않으면 35B급 모델로 전환하는 폴백 3건을 정의해 두었습니다(6장). 35B급 모델은 오픈마루가 쿠버네티스에서 따로 운영하는 또 하나의 LiteLLM 게이트웨이, 곧 2차 게이트웨이 뒤에 있습니다. LiteLLM이 두 백엔드의 오류를 같은 종류로 해석하지 못한다면 이런 전환 규칙을 백엔드별 조건 없이 설정 파일의 목록으로 적기 어렵습니다.

하지만 재시도와 폴백은 LiteLLM Proxy에만 있는 기능이 아닙니다. 두 기능은 Router라는 공통 구성 요소가 제공하므로 LiteLLM SDK에서도 동일하게 동작합니다[S02][S06]. 폴백이 필요하다는 이유만으로는 LiteLLM Proxy를 두어야 한다는 결론이 나오지 않으며, LiteLLM Proxy를 두는 근거는 형식 통일 뒤에도 남는 키와 비용의 문제에 있습니다.

2.3 라이브러리에서 조직 게이트웨이로 넓어진 요구: 키·비용·접근 통제의 중앙화

호출 형식을 통일한 뒤에도 조직에는 해결되지 않은 문제가 남습니다. LLM을 호출하는 주체가 여럿이 되면 프로바이더 키, 사용량 기록, 모델 접근 권한, 요청 로그가 호출 주체마다 따로 남기 때문입니다. 이 문제는 호출 코드의 품질과 무관하며, 호출 주체가 하나일 때는 나타나지 않습니다.

LiteLLM은 이 요구를 별도 제품으로 분리하지 않고 처음부터 라이브러리와 서버 두 형태로 대응했습니다[S08]. 라이브러리가 형식 문제를 맡고 서버가 조직 통제 문제를 맡는 구분이 초기부터 있었던 것입니다.

2.3.1 Python SDK와 FastAPI 서버 두 형태로 출발한 LiteLLM

Y Combinator 회사 소개는 LiteLLM을 오픈소스 Python SDK와 Python FastAPI 서버 두 가지로 설명합니다[S08]. 서버 형태는 라이브러리가 널리 쓰인 뒤에 추가한 부가 기능으로 볼 수 없으며, LiteLLM을 만든 쪽은 호출 형식 문제와 서버가 필요한 문제를 초기부터 함께 다루었습니다.

현재 공식 문서도 같은 구분을 유지합니다. LiteLLM SDK는 파이썬 애플리케이션에 직접 통합하는 라이브러리이고, LiteLLM Proxy는 조직 전체의 LLM 접근을 관리하는 플랫폼 팀이 자체 구축하는 게이트웨이입니다[S02]. 아래 표는 두 형태가 초기에 소개된 방식과 현재 공식 문서의 설명을 나란히 정리한 것입니다.

형태	초기 소개	현재 공식 문서의 설명	해결하는 문제
LiteLLM SDK	오픈소스 Python SDK	파이썬 애플리케이션에 직접 통합하는 라이브러리	프로바이더별 호출 형식 차이
LiteLLM Proxy	Python FastAPI 서버	플랫폼 팀이 자체 구축하는 조직 단위 게이트웨이	키·비용·접근 통제의 중앙 관리

저장소의 표기는 그 뒤로 달라졌습니다. 저장소 설명 문구는 현재 Rust 코어와 Python SDK를 함께 적고 있으며 저장소의 주 언어 표시는 Python이고, README는 LiteLLM을 오픈소스 AI 게이트웨이로 소개합니다[S01][S11].

두 형태가 초기부터 함께 있었다는 사실은 LiteLLM SDK와 LiteLLM Proxy를 같은 기능의 두 가지 배포 방식으로 보면 안 된다는 점을 알려 줍니다. LiteLLM SDK는 애플리케이션 한 개의 호출 코드를 단순하게 만들고, LiteLLM Proxy는 여러 애플리케이션의 호출을 한 곳에서 관리합니다. 어느 형태를 쓰지는 조직이 지금 겪는 문제가 형식 문제인지 통제 문제인지에 따라 정해집니다.

2.3.2 호출 주체가 늘 때 라이브러리만으로 해결되지 않는 키 보관과 비용 귀속

애플리케이션마다 LiteLLM SDK를 넣으면 호출 코드는 단순해지지만 프로바이더 키를 애플리케이션 수만큼 나누어 보관해야 합니다. 에이전트, 보조 스크립트, 독립 서비스가 각자 프로바이더를 직접 호출하려면 각자의 설정에 키가 있어야 하기 때문입니다. 이 상태에서 키를 교체하려면 키가 들어 있는 설정을 모두 찾아 수정해야 하고, 한 곳이라도 빠뜨리면 그 호출 주체는 인증 실패로 멈춥니다.

사용 기록도 각 애플리케이션이 자기 호출만 남기므로 조직 전체에서 어느 에이전트가 어느 모델을 얼마나 썼는지를 한 곳에서 확인할 수 없습니다. 또한 특정 호출 주체에만 특정 모델을 허용하거나 사용 한도를 두는 통제는 호출 주체의 코드에 맡길 수 없으며, 호출 경로 중간에서 요청을 검사하는 구성 요소가 있어야 적용할 수 있습니다.

LiteLLM Proxy는 이 통제를 가상 키로 제공합니다. 가상 키는 LiteLLM Proxy가 호출 주체별로 발급하는 키이며, 이 키로 지출 추적, 모델 접근 통제, 예산 설정을 할 수 있습니다[S03]. 공개판은 OpenAI 호환 게이트웨이, 가상 키, 지출 추적, 예산, 폴백, 요청·응답 로깅을 포함합니다[S04]. 다만 가상 키를 쓰려면 Postgres 데이터베이스를 연결해야 합니다[S03].

호출 주체가 늘 때 프로바이더 키와 사용 기록이 놓이는 자리

라이브러리만 쓰는 구성과 LiteLLM Proxy를 둔 구성

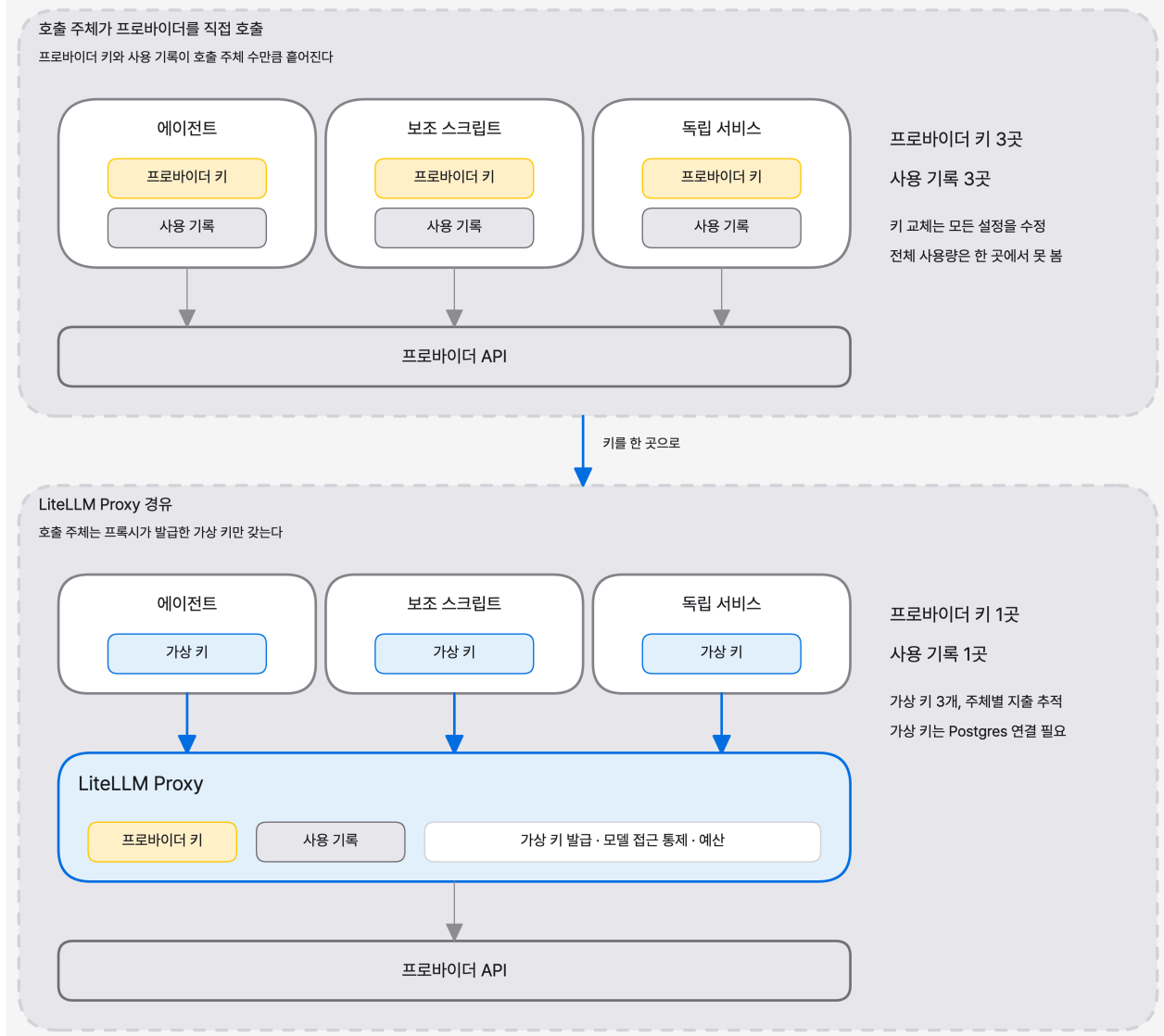


그림. 호출 주체 수에 따른 프로바이더 키와 사용 기록의 분포 대비

오픈마루의 구성을 예로 드는 이유는 호출 주체가 늘어난 조직에서 키가 어디에 놓이는지를 보여 주기 때문입니다. 오픈마루에서는 에이전트 스택 2개, 에이전트 콘솔, 에이전트 컨테이너 안의 보조 스크립트, 호스트의 슬라이드 검색 서비스와 영상 제작 도구 일부까지 소비자 6종이 LiteLLM Proxy를 호출합니다. 여기서 소비자는 LiteLLM Proxy를 호출하는 쪽을 가리킵니다. 이들이 사용하는 온프레미스 추론 서버와 클라우드 모델의 접속 정보는 LiteLLM Proxy 설정 한 곳에만 있습니다. 다만 이 구성은 데이터베이스를 연결하지 않아 가상 키를 쓰지 않고, 모든 소비자가 관리자 키인 마스터 키 하나를 공유합니다. 따라서 프로바이더 접속 정보의 보관은 한 곳으로 모았지만 소비자별 비용 귀속은 해결하지 못한 상태입니다.

LiteLLM Proxy를 도입할지는 조직 규모보다 두 가지 조건으로 판단하는 편이 정확합니다. 첫째 조건은 LLM을 호출하는 주체가 둘 이상인가이고, 둘째 조건은 키, 예산, 로그를 책임지는 통제 주체가 있는가입니다. 두 조건이 모두 해당하면 라이브러리만으로는 통제 수단이 부족하므로 게이트웨이 서버를 검토해야 합니다.

2.3.3 OpenAI 호환 클라이언트를 그대로 연결하는 단일 엔드포인트의 의미

LiteLLM Proxy는 OpenAI 호환 클라이언트를 그대로 연결할 수 있는 엔드포인트를 제공합니다[S02]. OpenAI 클라이언트를 쓰는 에이전트와 스크립트는 호출 코드를 유지한 채 접속 주소와 키만 LiteLLM Proxy의 것으로 바꾸면 연결할 수 있습니다. 이 점이 도입 부담을 결정합니다. 게이트웨이를 도입하기 위해 기존 에이전트와 도구를 다시 개발해야 한다면 호출 경로를 한 곳으로 모으는 작업은 시작하기 어렵기 때문입니다.

에이전트 프레임워크도 같은 방식으로 연결됩니다. 오픈소스 에이전트 프레임워크 Hermes Agent는 사용자가 지정한 엔드포인트를 포함해 원하는 모델을 쓸 수 있고, 모델을 코드 변경 없이 교체할 수 있습니다[S10]. Hermes Agent에 LiteLLM Proxy의 주소를 사용자 지정 엔드포인트로 설정하면 그 에이전트의 모델 호출은 모두 LiteLLM Proxy를 거칩니다.

오픈마루의 구성을 예로 드는 이유는 단일 엔드포인트가 에이전트 여러 개의 설정을 얼마나 단순하게 만드는지 보여 주기 때문입니다. 오픈마루가 운영하는 Hermes Agent 프로파일 16개는 모델 설정이 모두 같습니다. 프로파일은 설정 파일과 메모리를 따로 갖는 에이전트 단위입니다[S10]. 기본 모델 별칭, OpenAI 호환 방식, LiteLLM Proxy 주소 세 가지를 동일하게 갖기 때문입니다(7장). 기본 모델을 교체할 때는 프로파일 16개의 설정을 그대로 두고 LiteLLM Proxy 설정 파일의 별칭 정의만 수정합니다.

단일 엔드포인트가 있다고 해서 모든 호출이 저절로 모아지는 않습니다. 접속 주소를 바꾸지 않은 스크립트는 계속 프로바이더를 직접 호출하며, 오픈마루의 구성에도 LiteLLM Proxy를 거치지 않는 경로가 3건 남아 있습니다. 결국 호출 형식 문제는 LiteLLM SDK가 해결하고 키·비용·접근 통제 문제는 LiteLLM Proxy가 해결하지만, LiteLLM Proxy의 통제는 실제로 그 엔드포인트를 거치는 호출에만 적용됩니다.

3장: LiteLLM SDK 와 LiteLLM Proxy 의 차이: 앱 안의 라이브러리와 조직의 게이트웨이 서버

BerriAI는 LiteLLM을 두 가지 형태로 제공합니다. 하나는 파이썬 애플리케이션 안에 넣어 쓰는 라이브러리인 LiteLLM SDK이고, 다른 하나는 조직의 여러 애플리케이션이 함께 호출하는 게이트웨이 서버인 LiteLLM Proxy입니다[S02]. 두 형태는 이름이 비슷하고 프로바이더(모델 API를 제공하는 사업자)마다 다른 호출 형식을 OpenAI 형식 하나로 맞춘다는 목적도 같아서, 처음 접하는 독자는 한 제품의 설치 옵션 두 가지로 받아들이기 쉽습니다. 하지만 두 형태는 설치하는 위치가 다르고, 그 차이 때문에 사용 주체, 키 보관 위치, 장애 영향 범위가 모두 다릅니다.

AI 에이전트를 여러 개 운영하는 조직에서는 이 구분을 정확히 알아야 도입 범위를 정할 수 있습니다. 예를 들어 폴백(fallback, 한 모델이 실패했을 때 다른 모델로 요청을 넘기는 동작)이 필요하다는 이유만으로 LiteLLM Proxy를 도입하면 그 결정은 근거가 틀립니다. 재시도, 폴백, 부하 분산은 Router라는 공통 구성 요소의 기능이며 LiteLLM SDK에서도 그대로 동작하기 때문입니다[S02][S06]. 반면 소비자마다 키를 따로 발급하고 예산을 정하는 기능은 LiteLLM Proxy에만 있습니다. 여기서 소비자는 LiteLLM Proxy를 호출하는 애플리케이션, 에이전트, 스크립트를 가리킵니다. 따라서 서버를 따로 두는 이유는 키와 예산, 그리고 한 곳에 모이는 로그에서 찾아야 합니다.

LiteLLM Proxy 안에도 기능의 경계가 하나 더 있습니다. 폴백, 재시도, 쿨다운(실패가 이어진 백엔드를 정해진 시간 동안 호출 대상에서 빼는 동작), 타임아웃, 부하 분산은 설정 파일만으로 동작하지만, 가상 키(virtual key, 소비자별로 발급해 사용량과 권한을 나누는 키)는 Postgres 데이터베이스를 연결해야 쓸 수 있습니다[S03][S06]. LiteLLM Proxy를 도입했다는 사실만으로는 소비자별 통제가 가능한지 알 수 없고, 데이터베이스 연결 여부까지 확인해야 합니다.

이 장에서는 두 형태의 차이를 세 층으로 나누어 확인할 수 있습니다. 첫째 층은 설치 위치와 사용 주체이고, 둘째 층은 Router 공통 기능과 LiteLLM Proxy 전용 기능의 경계이며, 셋째 층은 데이터베이스 유무에 따른 LiteLLM Proxy의 기능 범위입니다. 세 층을 차례로 확인하면 독자는 자기 환경에 LiteLLM SDK로 충분한지, LiteLLM Proxy가 필요한지, 필요하다면 데이터베이스까지 연결해야 하는지를 판정할 수 있습니다. 4장의 LLM Gateway 요건 평가표도 공개판 포함 여부와 데이터베이스 필요 여부를 열로 두어 같은 구분을 씁니다.

3.1 LiteLLM SDK 와 LiteLLM Proxy 의 위치 대비: 애플리케이션 내부와 조직 공용 서버

LiteLLM SDK와 LiteLLM Proxy를 가르는 첫 번째 기준은 설치 위치입니다. LiteLLM SDK는 파이썬 애플리케이션의 프로세스 안에 들어가는 라이브러리이고, LiteLLM Proxy는 애플리케이션 밖에 따로 띄워 여러 애플리케이션의 요청을 받는 서버입니다[S02]. 위치가 다르면 프로바이더 키가 놓이는 곳과 장애가 미치는 범위, 그리고 그 구성 요소를 관리하는 사람이 다릅니다. 뒤에서 다루는 차이는 대부분 이 위치 차이에서 나옵니다.

3.1.1 휴대용 변환 어댑터와 건물 배전반 비유로 본 LiteLLM SDK 와 LiteLLM Proxy

LiteLLM SDK는 휴대용 변환 어댑터에, LiteLLM Proxy는 건물의 배전반에 건줄 수 있습니다. 변환 어댑터는 사용자가 각자 가지고 다니면서 콘센트 모양이 다른 곳에서도 같은 플러그를 꽂게 해 주는 도구이고, 어댑터를 누가 몇 개 쓰는지 건물 관리자가 알지 못합니다. 반면 배전반은 건물로 들어온 전기를 층과 회로별로 나누어 보내는 설비이며, 회로마다 차단 스위치와 계량기를 두어 어느 회로가 얼마나 쓰는지 확인하고 필요하다면 그 회로

만 꾸을 수 있습니다. LiteLLM SDK는 애플리케이션 하나가 여러 프로바이더를 같은 형식으로 호출하게 해 주고, LiteLLM Proxy는 조직의 모든 호출을 한 곳에서 나누어 보내면서 소비자별 사용량과 권한을 관리합니다.

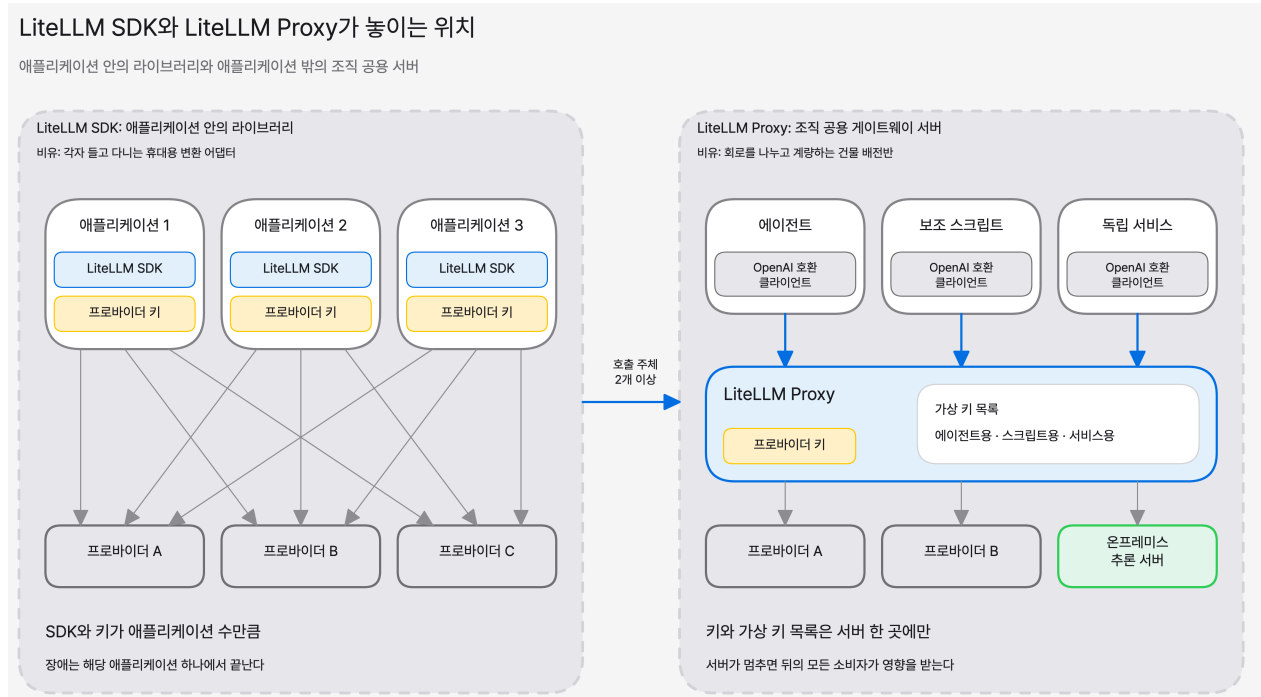


그림. 애플리케이션 안의 LiteLLM SDK와 조직 공용 서버인 LiteLLM Proxy의 위치 대비

이 비유는 설치 위치와 관리 범위를 설명할 때만 유효하고, 세 가지 부분에서 실제와 다릅니다. 첫째, 변환 어댑터는 모양만 바꾸지만 LiteLLM SDK는 Router를 통해 재시도, 폴백, 부하 분산까지 수행합니다[S02]. 차단 스위치에 해당하는 보호 기능이 배전반 쪽에만 있다고 이해하면 틀립니다. 둘째, 배전반이 있어도 어댑터는 여전히 필요하지만, LiteLLM Proxy 뒤의 애플리케이션은 OpenAI 호환 클라이언트를 그대로 연결하므로 형식 변환 라이브러리를 따로 넣지 않아도 됩니다[S02]. 셋째, 배전반은 모든 건물에 있어야 하는 설비인 데 비해 LiteLLM Proxy는 호출 주체가 늘고 통제가 필요했었을 때 선택하는 구성입니다.

비유가 다른 부분을 먼저 밝히는 이유는 기능 소속을 잘못 이해하면 도입 판단이 틀어지기 때문입니다. 재시도와 폴백이 서버에만 있다고 가정하면 애플리케이션 하나를 운영하는 팀도 서버를 세우게 되고, 운영할 대상만 늘어 납니다. 반대로 LiteLLM SDK로 형식 문제를 해결했으니 조직 통제도 해결됐다고 보면 키와 비용의 공백이 그대로 남습니다. 아래 두 항에서는 비유를 내려놓고 공식 문서가 밝힌 구성 요소를 기준으로 두 형태를 설명합니다.

3.1.2 LiteLLM SDK: 파이썬 애플리케이션 안에 넣어 쓰는 호출 라이브러리

LiteLLM SDK는 파이썬 애플리케이션에 직접 넣어 쓰는 호출 라이브러리이며, 사용 주체는 LLM 프로젝트를 만드는 개발자입니다[S02]. 공식 문서는 LiteLLM SDK를 OpenAI 클라이언트를 그대로 대체할 수 있는 구성 요소로 소개하므로, 기존에 OpenAI 클라이언트를 쓰던 코드는 호출부를 LiteLLM SDK의 함수로 바꾸는 방식으로 적용하고 그 뒤로는 같은 함수로 다른 프로바이더의 모델을 호출합니다[S02]. LiteLLM SDK는 애플리케이션과 같은 프로세스에서 실행되므로 따로 띄울 서버가 없고, 프로바이더 키는 그 애플리케이션의 환경 설정에 들어갑니다.

LiteLLM SDK의 구성 요소는 네 가지입니다[S01][S02].

구성 요소	하는 일
호출 함수	completion(), embedding(), image_generation() 같은 함수로 100개 이상 프로바이더를 OpenAI 형식으로 호출합니다
Router	재시도, 폴백, 여러 배포 사이의 부하 분산을 수행합니다
공통 예외	프로바이더가 달라도 같은 종류의 예외로 돌려주어 오류 처리 코드를 하나로 유지합니다
관측 콜백	호출 결과를 외부 관측 도구로 보내는 연결 기능을 제공합니다

이 표에서 주목할 구성 요소는 Router입니다. Router가 LiteLLM SDK에 들어 있으므로 애플리케이션 하나가 프로바이더 여러 곳을 호출하면서 실패에 대비해야 하는 경우에는 서버 없이도 요건을 채울 수 있습니다. 예를 들어 사내 문서 요약 서비스 하나가 클라우드 모델을 기본으로 쓰고 실패했을 때 다른 프로바이더로 넘기는 구성은 LiteLLM SDK만으로 구현할 수 있습니다. 호출 주체가 하나이고 키를 그 애플리케이션 담당자가 직접 관리해도 되는 상황이라면 LiteLLM SDK가 더 단순한 선택입니다.

다만 LiteLLM SDK가 해결하는 범위는 그 애플리케이션 하나까지입니다. 재시도 횟수와 폴백 순서는 애플리케이션 코드나 설정에 들어 있고, 호출 기록도 그 애플리케이션이 연결한 콜백으로만 나갑니다. 두 번째 애플리케이션이 생기면 같은 설정과 같은 키를 한 벌 더 두어야 하고, 두 애플리케이션의 사용량을 합쳐서 보는 일은 LiteLLM SDK 밖에서 따로 해결해야 합니다.

3.1.3 LiteLLM Proxy: 플랫폼 팀이 자체 구축하는 OpenAI 호환 게이트웨이 서버

LiteLLM Proxy는 조직 전체의 LLM 접근을 관리하는 플랫폼 팀이 자체 인프라에 구축해 운영하는 게이트웨이 서버입니다[S02]. 애플리케이션은 프로바이더를 직접 호출하지 않고 LiteLLM Proxy의 주소로 요청을 보내며, LiteLLM Proxy가 요청에 담긴 모델 이름을 실제 프로바이더나 추론 서버로 연결합니다. 애플리케이션 쪽에는 OpenAI 호환 클라이언트만 있으면 되므로, 파이썬으로 작성하지 않은 서비스나 기성 에이전트 프레임워크도 주소와 키만 바꾸어 연결할 수 있습니다.

LiteLLM Proxy가 제공하는 구성 요소는 다음 네 가지입니다[S01][S02].

구성 요소	하는 일
가상 키	소비자별로 키를 발급하고 키, 팀, 사용자 단위 예산을 연결합니다
중앙 로깅·가드레일·캐싱	모든 소비자의 요청을 한 곳에서 기록하고 검사하며 응답을 재사용합니다
관리 화면	가상 키와 예산 같은 관리 대상을 웹 화면에서 다룹니다
OpenAI 호환 엔드포인트	기존 OpenAI 호환 클라이언트를 코드 변경 없이 연결합니다

네 구성 요소는 모두 호출 주체가 여럿이라는 전제에서 의미가 생깁니다. 가상 키는 소비자를 구분하려는 기능이고, 중앙 로깅은 여러 소비자의 기록을 한 곳에 모으려는 기능입니다. LiteLLM Proxy의 대상은 여러 에이전트와 서비스가 같은 모델 자원을 함께 쓰는 조직이며, 애플리케이션 하나만 운영하는 팀에는 서버 한 대를 더 운영하는 부담이 이점보다 클 수 있습니다.

LiteLLM Proxy가 하지 않는 일도 함께 구분해 두어야 합니다. LiteLLM Proxy는 요청을 받아 형식을 맞추고 알맞은 백엔드로 넘기는 게이트웨이 계층이며, 토큰을 실제로 생성하는 일은 그 뒤에 있는 추론 서버나 프로바이더가 맡습니다. LiteLLM Proxy를 도입해도 모델의 생성 속도 자체는 달라지지 않습니다. 응답이 느릴 때 조치할 대상이 프록시인지 추론 서버인지를 나눠 보는 방법은 8장의 지연 지표에서 다룹니다.

두 계층은 함께 놓아 쓰는 구성 요소이며, 한쪽이 다른 쪽의 일을 대신하지 않습니다. 온프레미스에서 오픈 모델을 서빙하는 조직이라면 클라이언트, LiteLLM Proxy, vLLM 같은 추론 서버 순으로 구성 요소를 놓게 되고, 오픈마루의 구성도 기본 대화 모델을 이 순서로 호출합니다. 이 사례를 드는 이유는 게이트웨이 계층과 추론 계층을 나눠 두면 한쪽을 바꿀 때 다른 쪽을 고치지 않아도 된다는 점을 운영 중인 구성에서 확인할 수 있기 때문입니다. 추론 서버를 교체하거나 늘릴 때는 별칭 뒤의 배포 정의만 바꾸고, 게이트웨이의 키와 폴백 정의는 그대로 유지됩니다. LLM Gateway 도입 절차를 따로 정리한 글은 함께 읽을 자료 [R1]에 있습니다.

이 문서에는 게이트웨이라는 말이 세 가지 뜻으로 등장하기 때문에 수식어로 구분해 씁니다. LLM Gateway는 LiteLLM Proxy처럼 모델 호출을 한 곳에서 받는 서버를 가리키고, 에이전트의 게이트웨이 프로세스는 메신저 계정을 에이전트에 연결하는 프로세스를 가리킵니다. 2차 게이트웨이는 오픈마루가 쿠버네티스에서 따로 운영하는 또 하나의 LiteLLM 게이트웨이를 뜻합니다. 수식어 없이 게이트웨이라고 쓴 곳은 모두 LLM Gateway를 가리킵니다.

3.2 LiteLLM SDK와 LiteLLM Proxy 대조표: 사용 주체, 선택 시점, 키 관리, 장애 범위

LiteLLM SDK와 LiteLLM Proxy 가운데 어느 쪽이 맞는지는 조직의 크기보다 호출 주체의 수와 통제 필요를 기준으로 정합니다. 직원이 많은 조직이라도 LLM을 호출하는 애플리케이션이 하나뿐이면 LiteLLM SDK로 충분하고, 작은 팀이라도 에이전트와 스크립트 여러 개가 같은 키를 나누어 쓰기 시작하면 LiteLLM Proxy가 필요합니다. 두 형태를 같은 항목으로 나란히 놓고 사용 주체, 선택 시점, 키 관리, 장애 범위를 차례로 비교하면 독자의 환경이 어느 쪽에 해당하는지 판정할 수 있습니다.

3.2.1 단일 애플리케이션 개발자와 플랫폼 팀으로 나뉘는 사용 주체와 선택 시점

LiteLLM SDK의 사용 주체는 애플리케이션 하나를 만드는 개발자이고, LiteLLM Proxy의 사용 주체는 여러 팀과 에이전트의 LLM 접근을 관리하는 플랫폼 팀입니다[S02]. 개발자는 자기 코드 안에서 호출 형식을 통일하려고 LiteLLM SDK를 쓰고, 플랫폼 팀은 다른 사람이 만든 여러 애플리케이션의 호출을 한 곳에서 받으려고 LiteLLM Proxy를 씁니다. 두 형태는 같은 사람이 상황에 따라 고르는 선택지이기도 하지만, 조직 안에서 서로 다른 역할이 쓰는 도구이기도 합니다.

LiteLLM SDK에서 LiteLLM Proxy로 넘어가는 시점은 두 조건이 함께 충족될 때입니다. 하나는 LLM을 호출하는 주체가 둘 이상이 되는 것이고, 다른 하나는 프로바이더 키를 한 곳에서 관리해야 할 필요가 생기는 것입니다. 호출 주체가 둘이어도 각자 키를 관리하는 데 문제가 없다면 LiteLLM SDK를 두 곳에 쓰면 됩니다. 반면 키가 여러 설정 파일에 복사되어 교체가 어려워졌거나 누가 얼마를 썼는지 답해야 하는 상황이 되면 LiteLLM Proxy가 필요합니다.

아래 표는 다섯 가지 상황을 예로 들어 두 형태에 대응시킨 것입니다.

상황	맞는 형태	판단 근거
파이썬 애플리케이션 하나가 프로바이더 두세 곳을 호출합니다	LiteLLM SDK	호출 주체가 하나이고 키를 담당자가 직접 관리합니다
같은 애플리케이션에 재시도와 폴백이 필요합니다	LiteLLM SDK	Router가 LiteLLM SDK에 들어 있습니다
에이전트 여러 개와 보조 스크립트가 같은 모델을 함께 씁니다	LiteLLM Proxy	호출 주체가 여럿이고 키와 모델 설정을 한 곳에 모아야 합니다
파이썬이 아닌 서비스나 기성 에이전트 프레임워크를 연결합니다	LiteLLM Proxy	OpenAI 호환 클라이언트만으로 연결합니다
팀이나 소비자별 사용량과 예산을 나누어 관리합니다	LiteLLM Proxy와 Postgres	가상 키가 필요하고 가상 키에는 데이터베이스가 필요합니다

표의 셋째 줄 이후가 AI 에이전트를 운영하는 조직이 주로 해당하는 상황입니다. OpenAI 호환 방식으로 모델 주소와 키를 설정하는 에이전트 프레임워크라면, 에이전트 코드에 LiteLLM SDK를 넣는 것보다 LiteLLM Proxy의 주소를 설정에 적는 편이 변경 범위가 작습니다. 에이전트 밖에서 실행하는 보조 스크립트와 독립 서비스도 같은 주소로 모을 수 있으므로 호출 경로를 하나로 정리할 수 있습니다.

3.2.2 키 보관 위치와 장애 영향 범위로 본 두 형태의 운영 차이

키 보관 위치와 장애 영향 범위는 두 형태에서 서로 반대 방향으로 나타납니다. LiteLLM SDK를 쓰면 프로바이더 키가 애플리케이션마다 들어가므로 키의 수와 위치가 애플리케이션 수만큼 늘어나지만, 한 애플리케이션의 장애는 그 애플리케이션에서 끝납니다. LiteLLM Proxy를 쓰면 프로바이더 키가 서버 한 곳에 모이고, 가상 키를 쓰는 구성에서는 소비자가 LiteLLM Proxy에서 발급한 키만 받습니다[S03]. 그 대신 LiteLLM Proxy가 멈추면 그 뒤에 있는 모든 소비자의 모델 호출이 함께 멈춥니다. 이렇게 한 곳이 멈추면 전체가 멈추는 구성 요소를 단일 장애점(single point of failure)이라고 합니다.

아래 대조표는 두 형태를 같은 항목으로 비교하고, 형태마다 생기는 부담을 마지막 줄에 함께 적은 것입니다.

항목	LiteLLM SDK	LiteLLM Proxy
설치 위치	파이썬 애플리케이션의 프로세스 안	애플리케이션 밖의 공용 서버
사용 주체	LLM 프로젝트를 만드는 개발자	조직의 LLM 접근을 관리하는 플랫폼 팀
선택 시점	호출 주체가 하나일 때	호출 주체가 둘 이상이고 키를 한 곳에서 관리해야 할 때
키 관리	프로바이더 키를 애플리케이션마다 보관합니다	프로바이더 키는 서버에만 두고 소비자에게는 가상 키를 발급합니다
장애 범위	해당 애플리케이션 하나	LiteLLM Proxy 뒤의 모든 소비자

항목	LiteLLM SDK	LiteLLM Proxy
따라오는 부담	애플리케이션이 늘면 키와 설정이 함께 늘어납니다	서버를 운영해야 하고 단일 장애점이 됩니다

LiteLLM Proxy가 단일 장애점이라는 사실은 도입 전에 받아들여야 하는 조건입니다. 오픈마루의 운영 구성을 예로 드는 이유는 이 조건이 실제 운영에서 어떤 모습인지 보여 주기 때문입니다. 오픈마루는 LiteLLM Proxy를 도커 컨테이너 한 개로 운영하므로 이 컨테이너가 멈추면 모든 에이전트가 함께 멈추며, 다음 단계 후보로 복제와 헬스체크를 두고 있습니다. 같은 구성에서 모델을 바꿀 때는 에이전트 프로파일 16개의 설정을 그대로 두고 LiteLLM Proxy 설정 파일의 별칭 정의만 고칩니다. 프로파일은 설정 파일과 메모리를 따로 갖는 에이전트 단위이고, 별칭은 요청에 적는 모델 이름을 실제 백엔드에 연결하는 설정 항목입니다.

두 형태의 운영 차이는 어느 쪽이 더 안전한가로 판정할 수 없고, 어떤 부담을 선택하느냐의 문제입니다.

LiteLLM SDK는 장애를 좁게 가두는 대신 키와 설정을 여러 곳에 흩어 놓고, LiteLLM Proxy는 키와 설정을 한 곳에 모으는 대신 그 한 곳의 가용성을 책임져야 합니다. LiteLLM Proxy를 도입하는 조직은 키 통제의 이점과 함께 서버의 복제와 상태 점검 계획을 같은 문서에 적어 두어야 합니다.

3.3 공통 Router 기능과 LiteLLM Proxy 전용 기능의 경계

LiteLLM의 기능은 Router가 제공하는 공통 기능과 LiteLLM Proxy에만 있는 전용 기능 두 묶음입니다. 재시도, 폴백, 부하 분산은 Router의 기능이어서 LiteLLM SDK와 LiteLLM Proxy 양쪽에서 동작하고, 가상 키, 예산, 관리 화면은 LiteLLM Proxy에만 있습니다[S02][S06]. 이 경계를 알면 LiteLLM SDK로 충분하지 않으나 는 질문에 기능 이름으로 답할 수 있습니다. 폴백이 필요해서 LiteLLM Proxy를 도입한다는 식의 틀린 근거도 걸러 낼 수 있습니다.

3.3.1 양쪽에서 함께 쓰는 Router 기능: 재시도, 폴백, 부하 분산

Router는 LiteLLM SDK와 LiteLLM Proxy가 함께 쓰는 구성 요소이며 세 가지 기능을 제공합니다[S02]. 재시도는 실패한 요청을 같은 모델에 다시 보내는 기능이고, 폴백은 재시도 뒤에도 실패한 요청을 미리 정한 다른 모델로 넘기는 기능입니다. 부하 분산은 같은 모델 이름으로 묶은 배포 여러 개에 요청을 나누어 보내는 기능이며, 여기서 배포는 모델 이름에 연결한 실제 백엔드 하나를 가리킵니다[S06]. 세 기능은 LiteLLM SDK에서는 애플리케이션 코드로 Router를 구성해 쓰고, LiteLLM Proxy에서는 설정 파일에 적어 씁니다.

기능 소속을 표로 정리하면 다음과 같습니다.

기능	소속	LiteLLM SDK	LiteLLM Proxy
재시도	공통(Router)	사용 가능	사용 가능
폴백	공통(Router)	사용 가능	사용 가능
부하 분산	공통(Router)	사용 가능	사용 가능
가상 키	LiteLLM Proxy 전용	없음	사용 가능(Postgres 필요)
키·팀·사용자 예산	LiteLLM Proxy 전용	없음	사용 가능(Postgres 필요)
관리 화면	LiteLLM Proxy 전용	없음	사용 가능

기능	소속	LiteLLM SDK	LiteLLM Proxy
중앙 로깅	LiteLLM Proxy에서 제공	애플리케이션별 콜백만 제공	사용 가능

표의 위쪽 세 줄은 어느 형태를 선택해도 동작 방식이 같습니다. 차이는 설정이 놓이는 위치와 적용 범위에 있습니다. LiteLLM SDK에서는 폴백 순서를 애플리케이션마다 따로 정하므로 애플리케이션이 다섯 개면 폴백 정책도 다섯 벌을 관리해야 합니다. 반면 LiteLLM Proxy에서는 설정 파일 한 곳에 적은 폴백 순서를 모든 소비자가 똑같이 따릅니다.

LLM Gateway 요건을 판정할 때 폴백과 재시도는 LiteLLM Proxy의 고유 장점으로 계산하지 않는 것이 정확합니다. 이 기능들은 LiteLLM을 쓰면 형태와 상관없이 얻는 기능이기 때문입니다. LiteLLM Proxy를 선택해서 추가로 얻는 것은 폴백 정책을 한 곳에서 관리한다는 운영상의 이점이며, 기능 자체가 새로 생기는 것은 아닙니다.

3.3.2 LiteLLM SDK 만으로 남는 공백: 가상 키, 예산, 중앙 로깅

LiteLLM SDK만으로 구성된 환경에는 세 가지 기능이 없으며, 이 세 가지가 LiteLLM Proxy를 두는 이유입니다[S02][S03].

- 소비자별 키 발급: 프로바이더 키를 소비자에게 직접 나누어 주지 않고, 소비자마다 따로 발급한 가상 키로 접근하게 하는 기능
- 키·팀·사용자 단위 예산: 가상 키에 지출 한도를 연결하고 키별 지출을 추적하는 기능
- 한 곳에 모이는 로그: 모든 소비자의 요청과 응답을 한 곳에 기록하는 기능

세 기능은 모두 호출 주체를 구분하는 일과 관련이 있습니다. LiteLLM SDK는 애플리케이션 안에서 실행되므로 다른 애플리케이션이 무엇을 호출하는지 알 방법이 없고, 구조상 소비자 사이의 경계를 관리할 수 없습니다. 예를 들어 애플리케이션 세 개가 같은 프로바이더 키를 쓰면 프로바이더 쪽에서는 세 애플리케이션의 사용량을 키로 가릴 수 없고, 어느 애플리케이션이 비용을 늘렸는지는 각 애플리케이션의 로그를 따로 모아야 확인할 수 있습니다. LiteLLM Proxy에서는 가상 키가 지출 추적, 모델 접근 통제, 예산을 키 단위로 제공합니다[S03].

이 기능들은 LiteLLM Proxy의 공개판에 들어 있습니다. 공개판의 범위는 OpenAI 호환 게이트웨이, 가상 키, 지출 추적, 예산, 폴백, 요청·응답 로깅입니다[S04]. 소비자별 통제 때문에 LiteLLM Proxy를 도입하는 조직은 유료판 계약 없이 시작할 수 있습니다. 유료판은 관리 화면 SSO(Single Sign-On, 통합 로그인)나 감사 로그처럼 유료판에만 있는 기능이 필요할 때 검토하면 됩니다[S04].

LiteLLM SDK로 충분한지는 이 세 가지 공백이 독자의 환경에서 문제가 되는지로 판정합니다. 호출 주체가 하나이고 비용을 나누어 볼 필요가 없다면 세 공백은 문제가 되지 않으며, 이 경우 LiteLLM SDK는 부족한 선택이 아니고 문제에 맞는 선택입니다. 반면 세 가지 가운데 하나라도 필요하다면 LiteLLM SDK를 아무리 정교하게 구성해도 채워지지 않으므로 LiteLLM Proxy가 필요합니다. 두 형태는 우열 관계가 아니고 대상 문제가 다릅니다.

3.4 LiteLLM Proxy 의 두 운영 형태: YAML 단독 구성과 Postgres 연결 구성의 기능 경계

LiteLLM Proxy는 데이터베이스 연결 여부에 따라 쓸 수 있는 기능이 다릅니다. 폴백, 재시도, 쿨다운, 타임아웃, 부하 분산은 YAML 설정 파일만으로 동작하고 데이터베이스가 필요하지 않습니다[S06]. 반면 가상 키를 쓰

려면 Postgres 데이터베이스를 연결해야 합니다[S03]. LiteLLM Proxy의 운영 형태는 설정 파일 하나로 운영하는 YAML 단독 구성과 Postgres를 연결한 구성 두 가지로 나뉘며, 3.3절에서 LiteLLM Proxy 전용으로 분류한 기능 가운데 가상 키와 예산은 Postgres 연결 구성에서만 쓸 수 있습니다.

3.4.1 설정 파일만으로 동작하는 폴백·재시도·쿨다운과 마스터 키 인증

YAML 단독 구성은 상태를 저장하지 않는 컨테이너 한 개로 운영합니다. 모델 목록, 폴백 순서, 재시도 횟수, 쿨다운 시간, 타임아웃이 모두 설정 파일 하나에 들어 있고, LiteLLM Proxy는 시작할 때 이 파일을 읽어 그대로 동작합니다[S06]. LiteLLM Proxy 쪽에는 백업할 데이터가 없고, 컨테이너를 다시 띄우면 같은 설정 파일을 읽어 같은 방식으로 동작합니다.

이 구성에서는 마스터 키 하나로 인증합니다. 마스터 키는 다른 키를 만드는 관리자 키이며 `sk-` 로 시작해야 합니다[S03]. 가상 키에는 Postgres 데이터베이스가 필요하므로[S03], 데이터베이스가 없는 구성에서는 모든 소비자가 마스터 키를 그대로 사용합니다. 그 결과 LiteLLM Proxy는 요청이 인증됐는지만 확인할 수 있고, 그 요청을 어느 소비자가 보냈는지는 구분하지 못합니다.

YAML 단독 구성에서 가능한 기능과 불가능한 기능은 다음과 같습니다.

구분	기능
가능	OpenAI 호환 엔드포인트, 모델 별칭, 폴백, 재시도, 쿨다운, 타임아웃, 부하 분산, 마스터 키 인증
불가	가상 키, 팀, 예산, 속도 제한, 사용 기록

오픈마루의 LiteLLM Proxy가 이 YAML 단독 구성입니다. 이 사례를 드는 이유는 데이터베이스 없는 구성이 실제 운영에서 무엇을 주고 무엇을 남기는지 확인할 수 있기 때문입니다. 오픈마루는 읽기 전용으로 마운트한 YAML 파일 한 개와 마스터 키 하나로 LiteLLM Proxy를 운영하고, 모든 소비자가 같은 마스터 키를 사용합니다. 이 구성으로 모델 별칭과 폴백은 한 곳에서 관리하지만 가상 키, 팀, 예산, 속도 제한은 쓰지 않습니다. 관리 화면도 열리기는 하지만 데이터베이스가 없어 관리할 대상이 없습니다.

YAML 단독 구성은 운영 대상이 적다는 이점과 소비자를 구분할 수 없다는 한계를 함께 갖습니다. 호출 경로와 모델 설정을 한 곳으로 모으는 것이 우선 목표라면 이 구성으로 시작할 수 있고, 실제로 모델 교체와 폴백은 이 단계에서 이미 설정 파일 한 곳에서 관리합니다. 다만 3.3.2항에서 LiteLLM Proxy를 두는 이유로 꼽은 세 기능 가운데 가상 키와 예산은 이 구성에서 얻지 못합니다.

3.4.2 Postgres 연결로 열리는 가상 키·예산·지출 추적과 추가되는 운영 대상

Postgres를 연결하면 가상 키를 발급할 수 있고, 가상 키를 통해 세 가지 통제를 쓸 수 있습니다[S03]. 첫째는 키별 지출 추적이고, 둘째는 키마다 호출할 수 있는 모델을 제한하는 모델 접근 통제이며, 셋째는 키, 사용자, 팀 단위 예산입니다. 연결에는 Postgres 데이터베이스와 `DATABASE_URL` 설정이 필요하고, 가상 키는 마스터 키로 발급합니다. 이 기능들은 공개판에 포함되어 있어 데이터베이스만 준비하면 추가 계약 없이 사용할 수 있습니다[S04].

두 구성의 기능을 나란히 놓으면 다음과 같습니다.

항목	YAML 단독 구성	Postgres 연결 구성
폴백, 재시도, 쿨다운, 타임아웃, 부하 분산	사용 가능	사용 가능
인증	마스터 키 하나	마스터 키와 소비자별 가상 키
지출 추적	없음	키 단위로 추적
모델 접근 통제	없음	키마다 허용 모델 지정
예산	없음	키, 사용자, 팀 단위
운영 대상	컨테이너 한 개와 설정 파일	컨테이너와 Postgres 데이터베이스
복구 방법	컨테이너 재시작	컨테이너 재시작과 데이터베이스 복구

표의 마지막 두 줄은 Postgres 연결에 따르는 비용입니다. 데이터베이스가 생기면 운영 업무에 백업, 버전 관리, 접속 정보 보관, 장애 대응을 더해야 합니다. YAML 단독 구성에서는 컨테이너만 다시 띄우면 끝났던 복구 절차도 데이터베이스 복구를 포함하게 됩니다. Postgres 연결은 기능이 늘어나는 변경이면서 동시에 운영 대상이 하나 늘어나는 변경입니다.

두 구성은 조건에 따라 선택하는 대상이며 한쪽이 다른 쪽의 미완성 단계인 것은 아닙니다. 소비자가 적고 서로 신뢰하는 범위 안에 있으며 비용을 나누어 볼 필요가 없다면 YAML 단독 구성이 운영 부담에 비해 합리적입니다. 반면 소비자별 사용량을 나누어 보아야 하거나, 키 하나가 유출됐을 때 그 소비자의 키만 교체하고 싶다면 Postgres 연결이 선행 조건입니다. 키별 지출 추적도 가상 키가 있어야 가능하므로[S03], 소비자별 모니터링을 계획하는 조직은 데이터베이스 연결을 그보다 앞 단계에 두어야 합니다.

4장: LLM Gateway 가 갖춰야 할 조건과 LiteLLM Proxy 의 충족 범위

LLM Gateway는 여러 에이전트와 서비스가 모델을 호출할 때 공통으로 거치는 서버이며, 이 서버가 맡아야 할 일은 기본 요건 네 가지와 상세 요건 네 가지로 나눌 수 있습니다. 기본 요건은 OpenAI 호환 엔드포인트와 모델 별칭, 키 관리, 폴백, 요청·응답 로깅입니다. 상세 요건은 소비자별 가상 키와 예산, 지연과 첫 토큰 시간(TTFT, Time To First Token) 관측, 공급망 통제, LLM Gateway 자체의 가용성입니다. 기본 요건이 없는 제품으로는 호출 경로를 한 곳으로 모으는 목적을 이루지 못하고, 상세 요건을 갖추지 않은 구성에서는 소비자가 늘어날수록 비용 귀속과 장애 대응에 드는 시간이 늘어납니다.

요건을 먼저 정해 두어야 하는 이유는 제품 기능 목록이 평가 기준을 대신하기 쉽기 때문입니다. 기능 목록에서 출발하면 제품이 제공하는 항목만 점검하고, 조직에 필요한데 제품에 없는 항목은 검토 대상에서 놓칩니다. 반면 요건에서 출발하면 같은 표를 LiteLLM Proxy에도, 5장에서 비교하는 다른 오픈소스 LLM Gateway에도 그대로 적용할 수 있으므로 4장의 요건 평가표는 특정 제품의 소개 자료보다 점검표에 가깝습니다.

LiteLLM Proxy는 공개판만으로 기본 요건 네 가지를 모두 다룹니다. 공식 문서가 공개판에 포함된 기능으로 든 항목은 OpenAI 호환 게이트웨이, 가상 키, 지출 추적, 예산, 폴백, 요청·응답 로깅입니다[S04]. Prometheus 지표도 공개판 기능이어서 지연과 첫 토큰 시간 관측을 추가 비용 없이 시작할 수 있습니다[S05]. 관리 화면 SSO(Single Sign-On, 통합 로그인), 감사 로그, 외부 비밀 저장소 연동은 유료판에만 있으므로 조직의 보안 규정이 이 기능을 요구하는지를 따로 확인해야 합니다[S04].

요건마다 데이터베이스가 필요한지도 함께 구분해야 하는데, 모델 별칭, 폴백, 재시도, 쿨다운은 설정 파일만으로 동작하지만 가상 키는 Postgres 데이터베이스가 있어야 쓸 수 있습니다[S03][S06]. 도입 순서는 이 의존 관계에 따라 정합니다. 이 장에서는 여덟 요건을 하나씩 정의한 뒤 요건 항목, 판정 기준, 공개판 포함 여부, 데이터베이스 필요 여부를 한 표로 모았으며, 같은 표를 6장과 9장에서 오픈마루 운영 구성을 판정하는 기준으로 다시 씁니다.

4.1 기본 요건: OpenAI 호환 엔드포인트, 키 관리, 폴백, 요청·응답 로깅

기본 요건은 호출 주체가 둘 이상인 조직이라면 규모와 관계없이 필요한 최소 항목입니다. 클라이언트가 코드를 고치지 않고 연결할 수 있는 엔드포인트, 누가 호출할 수 있는지를 정하는 키, 백엔드 하나가 실패했을 때 다른 백엔드로 전환하는 폴백, 무엇이 오갔는지를 남기는 로깅이 여기에 해당합니다. 네 항목 가운데 하나라도 없으면 소비자는 프로바이더를 직접 호출하는 경로를 다시 만들기 때문에, LLM Gateway 제품을 검토할 때는 상세 기능을 보기 전에 이 네 가지가 유료 옵션 없이 제공되는지를 먼저 확인합니다.

LiteLLM Proxy는 네 항목을 모두 공개판에 포함합니다[S04]. 엔드포인트와 폴백은 설정 파일만으로 동작하고, 키 관리는 마스터 키 하나로 시작해 가상 키로 넓히는 구조입니다[S03][S06]. 기본 요건의 판정은 기능이 있는지보다 그 기능을 실제 구성에서 켜 두었는지를 기준으로 삼으며, 예를 들어 폴백 기능이 있어도 운영자가 폴백 대상을 한 건도 정의하지 않았다면 폴백 요건은 공백으로 판정합니다.

기본 요건 네 가지는 서로 의존하는데, 모델 별칭이 있어야 폴백 대상을 이름으로 지정할 수 있고 키가 있어야 로 그에 남은 요청을 호출 주체와 연결할 수 있습니다. 즉 엔드포인트와 별칭이 가장 먼저 갖춰야 할 항목이고 나머지 세 항목은 그 위에서 동작합니다.

4.1.1 OpenAI 호환 단일 엔드포인트와 모델 별칭으로 이루는 호출 경로 통일

LLM Gateway의 첫째 요건은 기존 클라이언트가 접속 주소와 키만 바꿔 연결할 수 있는 OpenAI 호환 엔드포인트입니다. 에이전트 프레임워크, 임베딩을 호출하는 검색 서비스, 보조 스크립트는 대부분 OpenAI 형식의 클라이언트를 이미 갖고 있습니다. 엔드포인트가 이 형식을 받으면 소비자 쪽 코드 수정 없이 호출 경로를 LLM Gateway로 옮길 수 있습니다. LiteLLM Proxy는 OpenAI 호환 클라이언트를 그대로 연결할 수 있는 엔드포인트를 제공하며, 그 뒤에서 100개 이상의 프로바이더를 같은 형식으로 호출합니다[S01][S02].

둘째 요건은 실제 백엔드를 소비자에게 드러내지 않는 모델 별칭입니다. 별칭은 클라이언트가 요청의 모델 이름으로 보내는 문자열이고 배포는 그 이름에 연결된 실제 추론 서버나 프로바이더 모델 하나이며, 두 용어는 뒤의 장에서도 같은 뜻으로 씁니다. LiteLLM Proxy 설정 파일에서는 `model_name` 항목이 별칭에 해당하며, 같은 `model_name` 으로 배포를 여러 개 두면 LiteLLM Proxy가 요청을 그 배포들에 나누어 보내 부하를 분산합니다[S06].

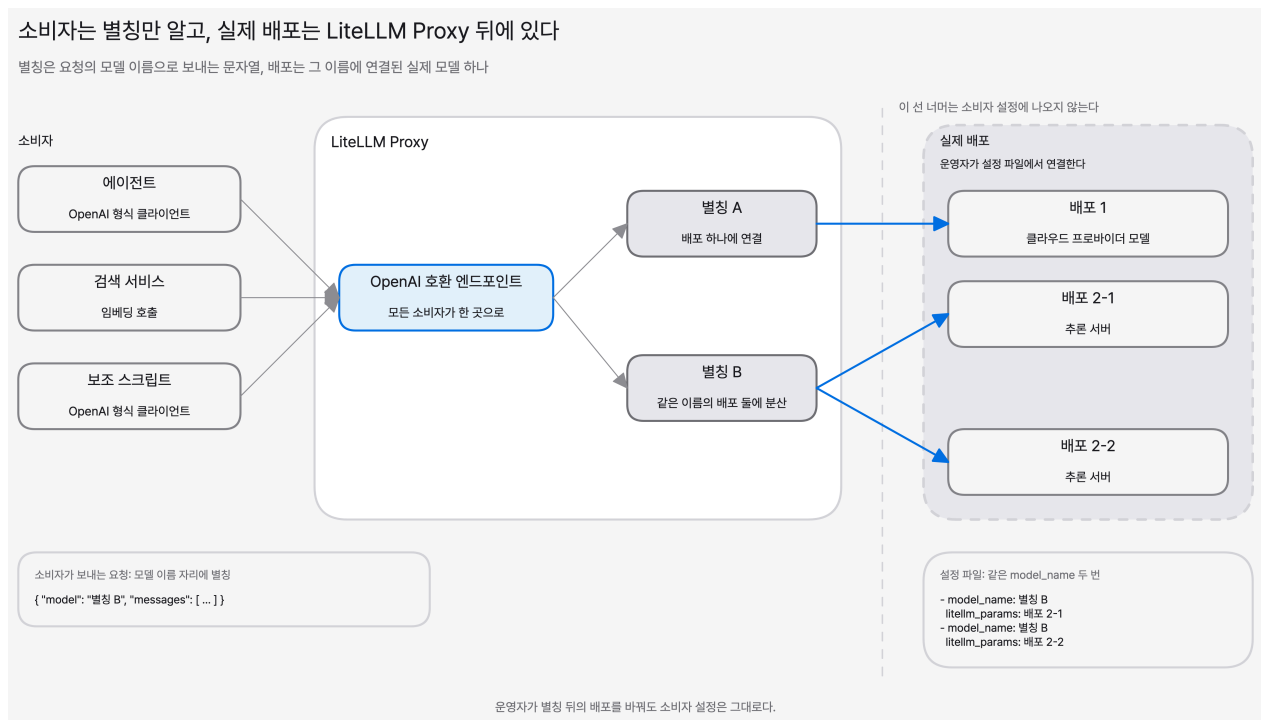


그림. 모델 별칭 하나와 그 뒤에 연결된 배포의 관계

별칭이 있으면 모델 교체가 소비자 설정 변경 없이 끝나는데, 소비자는 별칭만 알고 있어서 운영자가 별칭에 연결된 배포를 다른 모델로 바꿔도 소비자의 설정 파일을 수정할 필요가 없기 때문입니다. 반면 별칭 없이 프로바이더의 모델 이름을 그대로 노출하면 엔드포인트가 하나여도 모델을 교체할 때마다 모든 소비자의 설정을 수정해야 합니다. 이 요건의 판정 기준은 모든 소비자가 OpenAI 호환 엔드포인트 한 곳으로 호출하는지, 그리고 소비자 설정에 실제 백엔드 이름 대신 별칭이 들어 있는지 두 가지를 함께 확인합니다.

4.1.2 마스터 키와 가상 키로 나뉘는 키 관리와 요청·응답 로깅

키 관리 요건은 LLM Gateway가 프로바이더 키를 대신 보관하고, 소비자에게는 LLM Gateway 전용 키를 따로 발급하는 구조를 뜻합니다. 프로바이더 키가 소비자 쪽에 남아 있으면 소비자가 LLM Gateway를 건너뛰고 프로바이더를 직접 호출할 수 있으므로 호출 경로를 한 곳으로 모은 효과가 사라집니다. LiteLLM Proxy에서 소비자가 쓰는 키는 마스터 키와 가상 키 두 종류이며, 두 키의 역할은 아래 표와 같습니다[S03].

키 종류	역할	발급 단위	데이터베이스
마스터 키	다른 키를 만드는 관리자 키. sk- 로 시작해야 한다	관리자용으로 설정에 둔다	불필요
가상 키	소비자에게 나눠 주는 호출용 키. 지출 추적, 모델 접근 통제, 예산의 단위가 된다	소비자마다 발급하고 예산은 키, 팀, 사용자 단위로 건다	Postgres 필요

마스터 키만으로도 인증은 동작하지만 이 구성은 키 관리 요건을 부분적으로만 충족합니다. 모든 소비자가 같은 키로 호출하면 LiteLLM Proxy는 요청을 보낸 주체를 구분할 수 없고, 키가 유출되었을 때 교체 범위도 전체 소비자로 넓어집니다. 키 관리 요건은 가상 키 사용 여부로 판정하며, 소비자마다 가상 키를 발급했으면 충족, 마스터 키 하나를 함께 쓰면 부분 충족, 인증 없이 엔드포인트가 열려 있으면 공백입니다.

요청·응답 로깅은 어떤 소비자가 어떤 별칭으로 무엇을 요청했고 어떤 응답을 받았는지를 한 곳에 남기는 요건입니다. 호출 경로가 흩어져 있을 때는 로그도 소비자마다 따로 남기 때문에 장애 원인을 추적하려면 여러 시스템의 기록을 맞춰 봐야 합니다. LiteLLM Proxy는 요청·응답 로깅을 공개판에 포함하고, 중앙 로깅을 Proxy의 기능으로 설명합니다[S02][S04]. 다만 로깅 요건은 기록이 남는지만으로 판정하지 않으며, 요청별로 모델 별칭과 호출 주체가 함께 남아야 충족이고 요청 건수와 오류 건수만 확인할 수 있으면 부분 충족으로 판정합니다.

4.1.3 폴백 설정 4종과 안정성 설정 4종의 뜻

폴백 요건은 배포 하나가 실패했을 때 소비자가 오류를 받기 전에 LLM Gateway가 다른 배포로 요청을 다시 보내는 기능입니다. 온프레미스 추론 서버는 GPU 장비 한 대의 점검이나 재시작만으로 응답이 멈추므로, 에이전트를 상시 운영하는 조직에서는 폴백이 선택 사항이 될 수 없습니다. LiteLLM Proxy는 실패 유형에 따라 폴백 대상을 다르게 지정하도록 설정 키 네 가지를 두고, 재시도와 쿨다운을 조절하는 안정성 설정 키 네 가지를 함께 제공합니다[S06]. 운영자는 여덟 설정을 모두 설정 파일에 적으며, 이 설정들은 데이터베이스 없이 동작합니다. 재시도와 폴백은 LiteLLM SDK와 LiteLLM Proxy가 함께 쓰는 Router의 기능이므로 설정의 뜻은 두 형태에서 같습니다[S02].

묶음	설정 키	뜻
폴백	fallbacks	일반 오류로 실패했을 때 전환할 별칭 목록. 별칭 A가 실패하면 별칭 B로 보내는 형식으로 적는다
폴백	context_window_fallbacks	입력이 모델의 컨텍스트 한도를 넘었을 때 전환할 별칭
폴백	content_policy_fallbacks	프로바이더가 콘텐츠 정책 위반으로 거부했을 때 전환할 별칭
폴백	default_fallbacks	기본 폴백 대상을 적는 설정 키. 적용 조건은 도입 시점의 공식 문서로 확인한다
안정성	num_retries	모델 이름마다 다시 시도하는 횟수

묶음	설정 키	뜻
안정성	<code>allowed_fails</code>	1분 안에 이 횟수를 넘겨 실패하면 해당 배포를 쿨다운 상태로 전환하는 기준
안정성	<code>cooldown_time</code>	쿨다운 상태를 유지하는 시간(초)
안정성	<code>request_timeout</code>	요청 한 건의 응답을 기다리는 제한 시간

폴백 설정이 네 가지로 나뉜 이유는 실패 유형마다 적합한 대체 모델이 다르기 때문입니다. 컨텍스트 한도를 넘은 요청은 같은 크기의 다른 모델로 보내도 다시 실패하므로 컨텍스트가 더 큰 모델을 대상으로 지정해야 합니다. 추론 서버가 응답하지 않는 일반 오류는 비슷한 성능의 다른 배포로 전환하는 것만으로 해결할 수 있습니다. 콘텐츠 정책 위반은 프로바이더의 필터 기준이 원인이므로 다른 프로바이더의 모델을 대상으로 지정합니다.

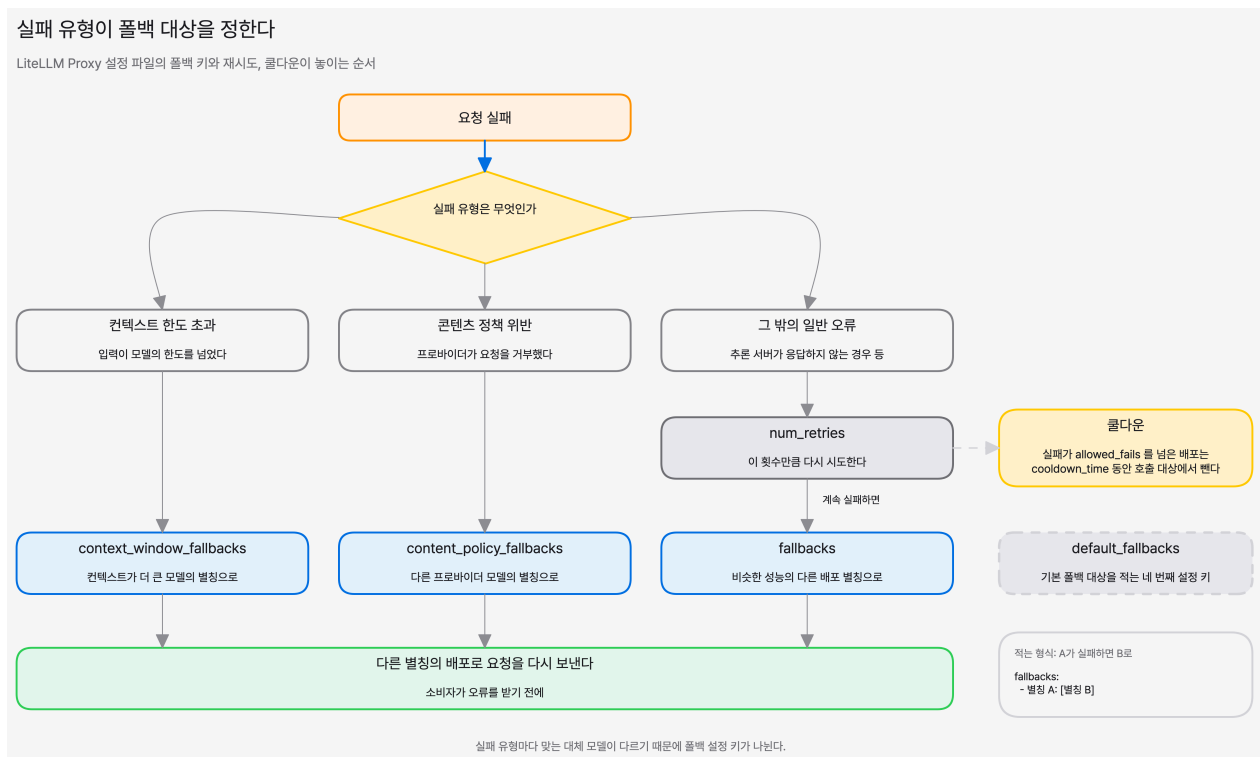


그림. 실패 유형에 따라 달라지는 폴백 대상과 재시도·쿨다운 순서

안정성 설정 네 가지는 폴백이 시작되기 전과 후의 동작을 정합니다. 요청이 실패하면 LiteLLM Proxy는 먼저 `num_retries` 만큼 다시 시도하고, 실패가 `allowed_fails` 를 넘으면 그 배포를 `cooldown_time` 동안 호출 대상에서 제외합니다[S06]. 쿨다운은 응답하지 않는 배포에 요청이 계속 쌓이는 것을 막는 장치이며, 서킷 브레이커와 비슷한 역할을 합니다. 재시도 횟수를 크게 설정하면 소비자가 오류를 받기까지의 대기 시간이 길어지고, 쿨다운 시간을 길게 설정하면 복구를 마친 배포가 다시 요청을 받기까지의 시간이 길어집니다. 이 관계를 알고 있어야 6장에 나오는 재시도 1회, 쿨다운 300초, 폴백 3건이 실제로 어떤 동작을 뜻하는지 읽을 수 있습니다.

4.2 상세 요건: 소비자별 가상 키와 예산, 첫 토큰 시간 관측, 공급망 통제, 가용성

상세 요건은 소비자 수와 호출량이 늘어난 뒤에 필요해지는 항목입니다. 에이전트 한두 개가 모델 하나를 호출하는 단계에서는 기본 요건만으로도 운영할 수 있습니다. 하지만 에이전트가 열 개를 넘고 예약 작업이 상시 실행되는 단계에서는 누가 얼마나 썼는지, 응답이 왜 느려졌는지, LLM Gateway가 멈추면 무엇이 함께 멈추는지를 설명할 수 있어야 합니다. 상세 요건 네 가지는 이 질문에 각각 대응합니다.

네 요건은 성격이 서로 다릅니다. 먼저 소비자별 가상 키와 예산, 그리고 관측은 LLM Gateway 제품이 제공하는 기능이므로 제품을 비교할 때 직접 확인할 수 있습니다. 반면 공급망 통제와 가용성은 제품 기능보다 배포 방식이 충족 여부를 정하므로, 같은 LiteLLM Proxy를 쓰더라도 이미지 버전을 고정했는지와 인스턴스를 몇 개 실행하는지에 따라 판정이 달라집니다.

상세 요건을 따로 세운 또 하나의 이유는 호출 경로를 한 곳으로 모으는 결정이 새로운 위험을 함께 만들기 때문입니다. 프로바이더 키와 모든 요청이 한 서버에 모이면 그 서버의 보안과 가용성이 조직 전체 AI 에이전트의 보안과 가용성이 됩니다. 상세 요건은 단일 경로를 선택한 조직이 그 선택에 뒤따라 반드시 점검해야 하는 항목입니다.

4.2.1 소비자별 가상 키와 키·팀·사용자 단위 예산, 지출 추적

소비자별 가상 키 요건은 에이전트, 서비스, 스크립트마다 서로 다른 키를 발급해 호출 주체를 구분하는 것입니다. LiteLLM Proxy의 가상 키는 지출 추적, 모델 접근 통제, 예산 세 가지 기능의 단위가 됩니다[S03]. 지출 추적은 키별로 사용한 비용을 집계하는 기능이고, 모델 접근 통제는 키마다 호출할 수 있는 별칭을 제한하는 기능입니다. 예산은 키, 사용자, 팀 단위로 한도를 정하는 기능이며 LiteLLM은 세 단위를 모두 공개판에서 제공합니다[S03][S04].

가상 키를 쓰려면 Postgres 데이터베이스를 연결하고 `DATABASE_URL` 을 설정해야 합니다[S03]. 키 목록, 키별 지출, 예산 잔액은 요청이 처리될 때마다 바뀌는 상태 정보이므로 읽기 전용 설정 파일에는 저장할 수 없기 때문입니다. 이 요건을 충족하려면 운영 대상에 데이터베이스가 하나 더 늘어나므로, 데이터베이스의 백업과 장애 대응을 LLM Gateway 운영 범위에 넣을 것인지를 도입 전에 함께 검토해야 합니다.

이 요건을 충족한 구성에서는 두 가지가 달라집니다. 첫째로 비용을 소비자에게 귀속할 수 있으므로 어느 에이전트의 예약 작업이 토큰을 가장 많이 쓰는지 근거를 갖고 조정할 수 있습니다. 둘째로 키 유출의 영향 범위가 그 키를 쓰던 소비자 하나로 줄어들고, 해당 키만 폐기하면 다른 소비자는 영향을 받지 않습니다. 판정 기준은 소비자마다 가상 키를 발급했고 키별 지출을 조회할 수 있으면 충족, 일부 소비자만 가상 키를 쓰면 부분 충족, 가상 키를 쓰지 않으면 공백입니다.

4.2.2 Prometheus 지표로 보는 지연, 첫 토큰 시간, 폴백 발생

관측 요건은 LLM Gateway가 처리한 요청의 지연, 첫 토큰 시간, 폴백 발생을 수치로 수집하는 것입니다. 첫 토큰 시간은 요청을 보낸 뒤 스트리밍 응답의 첫 토큰이 도착하기까지 걸린 시간이며, 사용자가 메신저에서 에이전트의 응답을 기다리는 체감 시간과 가장 가깝습니다. 전체 지연만 수집하면 긴 답변을 생성하느라 오래 걸린 요청과 모델 서버의 대기열에서 지체된 요청을 구분할 수 없습니다. 에이전트를 운영하는 LLM Gateway에서 전체 지연과 첫 토큰 시간을 따로 수집해야 하는 이유가 여기에 있습니다.

LiteLLM Proxy의 Prometheus 지표는 공개판 기능이며, 설정 파일에 `callbacks: ["prometheus"]` 한 줄을 추가해 켵니다[S05]. 이 설정을 추가하면 LiteLLM Proxy가 `/metrics` 경로로 지표를 노출하고, 지연과 첫 토큰 시간, 폴백 성공·실패 횟수가 그 안에 들어 있습니다. v1.85.0부터는 이 경로가 기본으로 Bearer 토큰 인증을 요

구하므로 수집기 설정에 토큰을 함께 넣어야 합니다[S05]. 관측 요건과 지표 묶음의 대응은 아래 표와 같고, 개별 지표 이름과 해석은 8장에서 설명합니다.

관측 요건	대응 지표 묶음	데이터베이스
응답 지연	지연 지표(전체 지연, 프로바이더 호출 지연, 첫 토큰 시간)	불필요
백엔드 장애와 우회	배포 상태 지표, 폴백·쿨다운 지표	불필요
요청량과 오류	요청 수 지표	불필요
소비자별 비용과 예산 잔액	비용·토큰 지표, 예산 지표	카·팀·사용자별 구분에는 가상 키와 Postgres 필요

관측 요건은 데이터베이스 없이도 충족을 시작할 수 있다는 점이 중요합니다. 지연, 첫 토큰 시간, 폴백 발생은 LiteLLM Proxy가 요청을 처리하면서 바로 계산하는 값이므로 가상 키가 없어도 수집할 수 있습니다. 다만 카·팀·사용자별 비용과 예산 지표는 가상 키가 있어야 값을 구분할 수 있습니다[S03][S05]. 판정 기준은 지연, 첫 토큰 시간, 폴백 발생을 모두 수집하면 충족, 요청 건수와 오류 건수만 확인할 수 있으면 부분 충족, 지표 수집이 꺼져 있으면 공백입니다.

4.2.3 공급망 통제와 가용성: 이미지 버전 고정, 복제, 헬스체크

공급망 통제는 LLM Gateway를 배포할 때 어떤 버전의 이미지와 패키지가 설치되는지를 운영자가 정하고 기록하는 요건입니다. LLM Gateway에는 조직이 쓰는 모든 프로바이더 키가 모이므로, 배포 과정에 악성 코드가 섞이면 피해 범위가 키 전체로 넓어집니다. LiteLLM은 2026년 3월에 악성 패키지 두 버전이 PyPI에 올라간 공급망 사고를 겪었으며, 공식 사고 보고는 안전한 버전으로 고정하는 조치를 권고했습니다[S07]. 공식 도커 이미지로 LiteLLM Proxy를 운영한 사용자는 영향을 받지 않았는데, 이미지가 의존성 버전을 고정해 두었기 때문입니다[S07]. 다만 이 설명은 LiteLLM 프로젝트가 배포하는 공식 이미지에 한정되므로, 제3자가 다시 빌드한 파생 이미지에는 그대로 적용할 수 없습니다. 사고의 경위와 수집 대상은 9장에서 설명하고, 여기서는 이 사례를 버전 고정이 요건이 되는 근거로만 듭니다.

가용성은 LLM Gateway가 멈췄을 때 모든 소비자가 함께 멈추는 문제에 대응하는 요건입니다. 호출 경로를 한 곳으로 모으면 LLM Gateway가 단일 장애점이 되므로 가용성은 백엔드 층과 LLM Gateway 층으로 나누어 점검합니다. 백엔드 층에서는 같은 별칭에 배포를 여러 개 두어 부하를 나누고, 실패한 배포를 쿨다운으로 제외 하며, 폴백으로 다른 별칭에 전환합니다[S06]. LLM Gateway 층에서는 인스턴스를 둘 이상 실행하는 복제와, 응답하지 않는 인스턴스를 감지해 교체하는 헬스체크가 필요합니다.

두 요건의 점검 항목은 아래 표와 같습니다. 백엔드 층의 항목은 LiteLLM Proxy 설정 파일로 충족하고, 나머지 항목은 컨테이너 실행 환경과 배포 절차에서 충족합니다.

요건	점검 항목	충족 수단
공급망 통제	공식 이미지인가, 제3자가 다시 빌드한 파생 이미지인가	배포 정의 파일
공급망 통제	이미지 태그를 특정 버전이나 다이제스트로 고정했는가	배포 정의 파일

요건	점검 항목	충족 수단
공급망 통제	새 릴리스를 확인한 뒤 올리는 절차가 있는가	배포 절차
가용성	별칭마다 배포가 둘 이상이거나 폴백 대상이 지정되어 있는가	LiteLLM Proxy 설정 파일
가용성	LLM Gateway 인스턴스를 둘 이상 실행하는가	컨테이너 실행 환경
가용성	응답하지 않는 인스턴스를 감지하는 헬스체크가 있는가	컨테이너 실행 환경

공급망 통제의 판정 기준은 버전이나 다이제스트를 고정하고 릴리스 확인 절차까지 갖추면 충족, 공식 이미지를 쓰지만 `latest` 같은 이동 태그를 쓰면 부분 충족입니다. 반면 공식 이미지가 아닌 파생 이미지이거나 버전·다이제스트를 확인하지 않으면 공백이며, 출처가 확인되지 않은 패키지를 직접 설치한 경우도 공백으로 판정합니다. 가용성은 백엔드 층과 LLM Gateway 층을 모두 갖추면 충족, 폴백이나 배포 묶음은 있지만 인스턴스가 하나이면 부분 충족, 둘 다 없으면 공백입니다. 두 요건은 운영자가 배포를 설계하는 단계에서 결정하므로, 제품 비교표에 항목이 없더라도 평가표에는 반드시 넣어야 합니다.

이 요건이 다루는 범위는 LLM Gateway 자체의 이미지와 패키지까지입니다. 온프레미스 추론 서버에 올리는 모델 가중치 파일에도 출처와 무결성을 확인해야 하는 별도의 공급망이 있지만, 그 확인은 모델을 추론 서버에 반입하는 절차에서 이루어지며 LLM Gateway가 대신해 주지 않습니다. LLM Gateway는 별칭 뒤에 연결된 배포가 응답하는지는 알 수 있어도 그 배포가 어떤 파일을 불러와 실행 중인지는 알지 못합니다. 따라서 오픈 모델을 직접 서빙하는 조직은 평가표의 공급망 통제 항목과 별도로 모델 파일의 반입 기준을 따로 두어야 합니다. 이 문서의 평가표에서 공급망 통제 판정은 LiteLLM Proxy 이미지에 대한 판정이며, 별칭 뒤의 모델 파일은 판정 대상에 넣지 않았습니다.

4.3 LiteLLM Proxy 공개판과 유료판의 충족 범위: SSO·감사 로그·외부 비밀 저장소의 경계

LiteLLM Proxy는 MIT 라이선스의 공개판과 별도 라이선스가 적용되는 유료판으로 나뉩니다. 도입을 검토하는 조직이 가장 먼저 확인하는 내용은 공개판만으로 운영할 수 있는 범위입니다. 공식 문서가 밝힌 공개판 기능과 앞에서 정의한 여덟 요건을 맞춰 보면, 제품 기능으로 충족하는 여섯 요건은 모두 공개판 범위에 들어 있습니다[S04][S05]. 나머지 두 요건인 공급망 통제와 가용성은 배포 방식으로 충족하므로 판의 구분과 관계가 없습니다.

유료판 전용 기능은 조직의 계정 체계나 감사 체계와 LiteLLM Proxy를 연결하는 영역에 모여 있습니다. 관리 화면 SSO, 감사 로그, JWT(JSON Web Token) 인증, IP 접근 제어 목록, 외부 비밀 저장소 연동이 대표 항목입니다[S04]. 모델 호출을 처리하는 기능은 공개판에 있고 관리자와 조직 정책을 다루는 기능은 유료판에 있으므로, 이 구분을 알고 있으면 검토 초기에 유료판이 필요한 조직인지를 빠르게 판단할 수 있습니다.

판의 경계는 제품 릴리스에 따라 달라질 수 있으므로 도입 시점의 공식 문서로 다시 확인해야 합니다. 아래 두 항의 내용은 LiteLLM 공식 문서의 Enterprise 페이지와 Prometheus 페이지에 적힌 범위를 넘지 않습니다[S04][S05].

4.3.1 공개판에 포함된 가상 키·예산·폴백·로깅·Prometheus 지표

LiteLLM 저장소의 라이선스는 MIT이며 상업적 이용이 가능하지만, 저장소의 `enterprise/` 디렉터리에는 별도 라이선스가 적용됩니다[S01]. 공개판에 포함된 기능으로 공식 문서가 든 항목은 여섯 가지이며 Prometheus 지표를 더하면 아래 목록이 됩니다[S04][S05].

- OpenAI 호환 게이트웨이
- 가상 키
- 지출 추적
- 예산
- 폴백
- 요청·응답 로깅
- Prometheus 지표

Prometheus 지표 가운데 유료판 전용으로 표시된 묶음은 Managed Batch 하나입니다[S05]. 지연, 첫 토큰 시간, 배포 상태, 폴백·쿨다운, 비용·토큰, 예산 지표는 공개판에서 수집할 수 있으므로, 운영 규모가 작은 단계에서도 추가 계약 없이 관측을 시작할 수 있습니다.

이 범위는 문서 뒤쪽의 논의와 직접 연결됩니다. 오픈마루의 운영 구성(6장)과 다음 단계로 제시하는 버전 고정, 지표 수집, 가상 키 발급(9장)은 모두 공개판 기능만으로 수행할 수 있습니다. 다음 단계를 판단할 때 검토할 항목은 라이선스 비용보다 운영 부담이며, Postgres를 추가로 운영할 것인지와 지표 수집기를 어디에 둘 것인지가 여기에 해당합니다.

4.3.2 유료판 전용 기능: 관리 화면 SSO, 감사 로그, JWT 인증, IP 접근 제어, 외부 비밀 저장소 연동

유료판에만 있는 기능은 공식 문서의 Enterprise 페이지에 나열되어 있으며, 아래 표는 그 항목을 공개판 기능과 나란히 놓은 것입니다[S04].

영역	공개판	유료판 전용
인증	마스터 키, 가상 키	관리 화면 SSO(OIDC, SAML), JWT 인증, IP 접근 제어 목록
감사	요청·응답 로깅	감사 로그와 보관 정책, 팀별 로깅과 로그 내보내기, 팀별 로깅 끄기
비밀 값 보관	—	외부 비밀 저장소 연동(AWS KMS, Azure Key Vault, HashiCorp Vault 등)
예산	키·사용자·팀 단위 예산	태그 기반 예산과 프로젝트 관리
가드레일	공식 문서의 공개판 기능 목록에는 적혀 있지 않음	키·팀 단위 가드레일
지표	Prometheus 지표	Managed Batch 지표 묶음

유료판도 공개판과 마찬가지로 조직의 자체 인프라에 설치하는 방식이며, 가격은 배포 규모에 따라 정합니다 [S04]. 유료판을 선택하더라도 LLM Gateway를 조직의 인프라 안에서 운영한다는 점은 공개판과 같으므로, 데이터 위치 때문에 자체 구축을 선택한 조직은 판을 바꾸더라도 구성 원칙을 유지할 수 있습니다.

유료판 검토가 필요한 조건은 조직의 보안 규정에서 나옵니다. 예를 들어 관리 화면에 접속하는 계정을 사내 계정 체계와 통합해야 하거나, 누가 어떤 키를 만들고 폐기했는지를 감사 로그로 보관해야 하는 조직은 유료판을 검토해야 합니다. 프로바이더 키를 환경변수 대신 외부 비밀 저장소에 두도록 규정한 조직도 여기에 해당합니다. 이런 규정이 없는 조직이라면 공개판으로 여덟 요건을 모두 다룰 수 있으므로, 공개판으로 운영을 시작하고 규정이 생기는 시점에 유료판을 검토하는 순서가 합리적입니다.

4.4 운영 구성 판정에 다시 쓰는 LLM Gateway 요건 평가표

LLM Gateway 요건 평가표는 4.1절부터 4.3절까지 정의한 여덟 요건을 한 장으로 모은 점검표입니다. 표의 열은 구분, 요건 항목, 판정 기준, 공개판 포함 여부, 데이터베이스 필요 여부 다섯 개입니다. 판정 값은 충족, 부분 충족, 공백 세 가지로 통일했으며, 부분 충족을 따로 둔 이유는 기능을 켜지만 일부 소비자나 일부 별칭에만 적용한 구성을 충족과 구분해야 하기 때문입니다.

위의 두 장이 이 표를 다시 사용합니다. 6장 끝에서는 오픈마루가 도커로 운영하는 LiteLLM Proxy 구성을 같은 여덟 항목으로 판정하고, 9장 첫 절에서는 한계 7건을 같은 항목 이름에 대응시켜 재판정합니다. 오픈마루의 구성을 판정 대상으로 삼는 이유는 데이터베이스 없이 설정 파일만으로 운영하는 구성이 어느 요건까지 충족하는지를 실제 사례로 보여 주기 때문입니다. 항목 이름과 순서는 위의 장에서도 바꾸지 않습니다.

독자는 같은 표를 자기 환경에도 적용할 수 있습니다. 공개판 포함 여부 열을 검토 중인 다른 제품의 무료 범위로 바꾸면 제품 비교표가 되고, 판정 열을 하나 추가하면 현재 구성의 점검 결과가 됩니다. 이때 판정 기준 열을 그대로 두어야 여러 제품을 같은 조건에서 비교할 수 있습니다.

4.4.1 요건 항목별 판정 기준과 공개판 포함 여부, 데이터베이스 필요 여부

여덟 요건의 판정 기준은 기능이 제품에 있는지보다 현재 구성에서 그 기능을 쓰고 있는지를 묻도록 작성했는데, 기능이 제품에 있어도 설정에서 꺼 두었다면 소비자와 운영자가 얻는 것이 없기 때문입니다. 공개판 포함 여부와 데이터베이스 필요 여부는 LiteLLM 공식 문서에 근거해 적었습니다[S03][S04][S05][S06].

구분	요건 항목	판정 기준	공개판 포함 여부	데이터베이스 필요 여부
기본	OpenAI 호환 엔드포인트와 모델 별칭	모든 소비자가 엔드포인트 한 곳으로 호출하고 소비자 설정에 별칭만 있으면 충족. 일부 소비자가 프로바이더를 직접 호출하면 부분 충족	포함	불필요
기본	키 관리	소비자마다 가상 키를 쓰면 충족. 마스터 키 하나를 함께 쓰면	포함	마스터 키는 불필요, 가상 키는 필요

구분	요건 항목	판정 기준	공개판 포함 여부	데이터베이스 필요 여부
		부분 충족. 인증이 없으면 공백		
기본	폴백과 안정성 설정	주요 별칭에 폴백 대상이 있고 재시도·쿨다운 값이 정해져 있으면 충족. 일부 별칭에만 있으면 부분 충족	포함	불필요
기본	요청·응답 로깅	요청별로 별칭과 호출 주체가 남으면 충족. 요청 건수와 오류 건수만 확인되면 부분 충족	포함	사용 기록을 LiteLLM Proxy에 보관하려면 필요
상세	소비자별 가상 키와 예산	소비자별 키가 있고 키별 지출을 조회할 수 있으면 충족. 일부 소비자만 쓰면 부분 충족. 쓰지 않으면 공백	포함	필요
상세	지연·첫 토큰 시간 관측	지연, 첫 토큰 시간, 폴백 발생을 모두 수집하면 충족. 요청 건수만 확인되면 부분 충족. 지표 수집이 꺼져 있으면 공백	포함	불필요. 키·팀·사용자별 구분에는 필요
상세	공급망 통제	버전이나 다이제스트를 고정하고 릴리스 확인 절차가 있으면 충족. 공식 이미지를 이동 태그로 쓰면 부분 충족. 파생 이미지이거나 버전·다이제스트를 확인하지 않으면 공백	해당 없음(배포 방식)	불필요
상세	LLM Gateway 가용성	배포 묶음이나 폴백이 있고 인스턴스 복제와 헬스체크가 있으면 충족. 인스턴스	배포 묶음·쿨다운·폴백은 포함. 복제와 헬스체크는 실행 환경에서 구성	불필요

구분	요건 항목	판정 기준	공개판 포함 여부	데이터베이스 필요 여부
		가 하나이면 부분 충족		

표를 읽을 때 주의할 항목은 키 관리와 소비자별 가상 키입니다. 두 항목은 모두 가상 키를 다루지만 묻는 내용이 다른데, 키 관리는 호출 주체를 인증하고 구분하는지를 묻고, 소비자별 가상 키와 예산은 그 구분 위에서 지출과 한도를 다루는지를 묻습니다. 예를 들어 가상 키를 발급했지만 예산과 지출 조회를 쓰지 않는 구성은 키 관리가 충족이고 소비자별 가상 키와 예산은 부분 충족입니다.

공개판 포함 여부 열에서 여섯 항목이 포함이고 두 항목이 배포 방식에 해당한다는 점도 확인해야 합니다. LiteLLM Proxy의 경우 공급망 통제와 가용성을 제외한 모든 요건을 공개판 기능으로 충족할 수 있습니다 [S04][S05]. 평가표에서 공백이나 부분 충족으로 판정한 항목이 있다면 그 원인은 유료 기능의 부재보다 구성 선택에 있으며, 9장이 다음 단계를 구성 변경으로 제시하는 근거도 이 사실입니다.

4.4.2 데이터베이스 유무에 따른 요건 충족 가능 범위

평가표의 마지막 열을 기준으로 여덟 요건을 다시 묶으면 데이터베이스 없이 충족할 수 있는 요건과 데이터베이스가 있어야 충족할 수 있는 요건으로 나뉩니다. LiteLLM Proxy의 운영 형태는 설정 파일만 쓰는 YAML 단독 구성과 Postgres 연결 구성 두 가지이며(3.4절), 아래 표는 구성별로 도달할 수 있는 최고 판정을 적은 것입니다[S03][S05][S06].

요건 항목	YAML 단독 구성	Postgres 연결 구성
OpenAI 호환 엔드포인트와 모델 별칭	충족 가능	충족 가능
키 관리	부분 충족까지(마스터 키)	충족 가능(가상 키)
폴백과 안정성 설정	충족 가능	충족 가능
요청·응답 로깅	부분 충족까지(호출 주체 구분 없음)	충족 가능
소비자별 가상 키와 예산	공백	충족 가능
지연·첫 토큰 시간 관측	충족 가능(키별 구분 제외)	충족 가능(키별 구분 포함)
공급망 통제	충족 가능	충족 가능
LLM Gateway 가용성	충족 가능	충족 가능

YAML 단독 구성으로도 여덟 요건 가운데 다섯은 충족까지 도달할 수 있습니다. 모델 별칭, 폴백, 재시도, 쿨다운은 설정 파일만으로 동작하고[S06], 지연과 첫 토큰 시간 지표는 폴백 한 줄로 수집되며[S05], 버전 고정과 복제는 배포 방식에 속합니다. 따라서 데이터베이스가 없는 구성도 호출 경로 통일, 장애 우회, 응답 지연 관측이라는 세 가지 목적은 달성할 수 있습니다. 운영 대상이 컨테이너 하나와 설정 파일 하나에 그친다는 점이 이 구성의 이점입니다.

반면 소비자를 구분해야 하는 요건은 모두 데이터베이스에 의존합니다. 가상 키는 Postgres가 있어야 발급할 수 있고[S03], 키·팀·사용자별 비용과 예산 지표는 가상 키가 있어야 값을 구분할 수 있습니다[S05]. 요청·응답

로깅도 호출 주체를 함께 남기려면 주체를 식별하는 가상 키가 먼저 필요합니다. 그 결과 YAML 단독 구성에서 키 관리와 로깅은 부분 충족이 상한이고, 소비자별 가상 키와 예산은 공백으로 남습니다.

도입 순서는 이 의존 관계를 따르며, 선행 조건이 없는 조치인 버전 고정과 지표 수집을 먼저 수행하고, 그다음에 Postgres를 연결해 가상 키를 발급하며, 마지막으로 키별 지표와 예산을 사용하는 순서입니다. 순서를 바꿔 키별 비용 대시보드부터 만들면 모든 값이 마스터 키 하나의 합계로만 나오므로 가상 키를 발급한 뒤 대시보드를 다시 구성해야 합니다. 9장의 다음 단계는 이 순서를 오픈마루의 운영 구성에 적용한 결과이며, 독자의 환경에서도 평가표의 마지막 열을 먼저 확인하면 같은 방식으로 순서를 정할 수 있습니다.

5장: 오픈소스 LLM Gateway 지형과 LiteLLM 의 위치: 상위 5개와 국내 공급 유형

LLM Gateway는 애플리케이션과 에이전트의 모델 호출을 한 곳에서 받아 프로바이더로 중계하는 서버이며, 오픈소스로 공개된 구현 가운데에도 출발 배경이 서로 다른 프로젝트가 여럿 있습니다. 이 장에서 비교하는 대상은 LiteLLM, Kong, Bifrost, Higress, Agent Router 다섯 개입니다. 다섯 후보의 스타 수, 라이선스, 저장소 생성 시기는 모두 2026년 10월 6일 GitHub API 조회값입니다[S11].

대안 비교가 지금 필요한 이유는 LLM Gateway를 한 번 정하면 모든 호출 주체의 엔드포인트와 키가 그 제품에 묶이기 때문입니다. 따라서 도입을 결정하는 쪽에서는 후보마다 무엇을 목적으로 만든 프로젝트인지, 어떤 라이선스로 공개돼 있는지, 누가 프로젝트를 관리하는지를 같은 항목으로 놓고 확인해야 합니다. 비교 대상을 뽑은 기준이 먼저 공개돼 있지 않으면 비교표는 결론에 맞춰 고른 목록으로 읽히기 쉽습니다.

국내에서 제안받는 솔루션은 오픈소스 프로젝트와 다른 방식으로 구분합니다. 국내 공급 솔루션은 회사명과 제품명을 쓰지 않고, 제품이 무엇을 중심 기능으로 삼는지에 따라 다섯 가지 유형으로만 설명합니다. 유형으로 구분해 두면 제안서에 적힌 기능 목록을 4장의 LLM Gateway 요건 평가표에 대응시키는 작업이 쉬워집니다.

이 장에서 얻는 판단 근거는 세 가지이며, 첫째는 다섯 후보를 어떤 기준으로 골랐고 조회 수치를 어디까지 믿을 수 있는가입니다. 둘째는 국내에서 제안받은 제품이 다섯 유형 가운데 어디에 속하는가입니다. 셋째는 LiteLLM 보다 다른 후보를 먼저 검토하는 편이 맞는 조직 조건이 무엇인가이며, 이 조건을 함께 적어 두어야 LiteLLM의 위치에 대한 설명도 과장 없이 읽힙니다.

5.1 상위 5개 선정 기준: 공개 저장소, 라이선스, GitHub 스타·포크, 저장소 생성 시기

비교 대상 다섯 개는 공개 저장소, 오픈소스 라이선스, LLM 호출 중계 기능이라는 세 조건을 먼저 적용한 뒤 GitHub 조회 수치를 확인해 골랐습니다. 수치의 출처는 GitHub API가 돌려주는 저장소 메타데이터 한 곳이며, 조회일은 2026년 10월 6일입니다[S11]. 선정 기준과 수치의 한계를 비교표보다 앞에 두는 까닭은 표의 숫자가 무엇을 센 값인지 알아야 비교가 성립하기 때문입니다.

5.1.1 GitHub API 조회 수치에 근거한 다섯 후보 선정 절차

다섯 후보는 누구나 열어 볼 수 있는 공개 저장소를 갖고 있고, 오픈소스 라이선스로 배포되며, LLM 호출을 중계하는 게이트웨이 기능을 제공합니다. 공개 저장소를 첫 조건으로 둔 것은 독자가 같은 수치를 직접 다시 조회할 수 있어야 하기 때문입니다. 라이선스 조건은 도입 뒤에 코드를 고치거나 자체 배포본을 유지할 수 있는지를 결정하므로 기능 비교보다 먼저 확인해야 합니다. 세 조건을 만족하는 프로젝트 가운데 GitHub 조회 수치로 채택 규모가 확인되는 LiteLLM, Kong, Bifrost, Higress, Agent Router를 비교 대상으로 삼았습니다[S11].

조회에는 GitHub API의 저장소 메타데이터를 사용했으며, 저장소마다 스타 수, 라이선스, 저장소 생성일, 저장소 설명을 읽었습니다. 선정에 쓴 기준과 기준마다 확인한 항목, 확인 방법은 아래 표에 정리돼 있습니다. 표의 값은 2026년 10월 6일 조회값이므로 독자가 다시 조회하면 스타 수는 달라져 있을 수 있으며, 수치는 후보의 채택 규모를 가늠하는 조회값으로만 쓰입니다.

선정 기준	확인한 항목	확인 방법
공개 저장소	저장소가 공개돼 있고 누구나 조회할 수 있는가	GitHub API 저장소 메타데이터
오픈소스 라이선스	저장소에 표시된 라이선스 종류	GitHub API 저장소 메타데이터, 저장소의 LICENSE
LLM 호출 중계 기능	저장소 설명과 공식 소개가 LLM 또는 AI 게이트웨이 기능을 밝히는가	저장소 설명, 재단 공지
채택 규모	스타 수, 포크 수	GitHub API 저장소 메타데이터
프로젝트 이력	저장소 생성 시기	GitHub API 저장소 메타데이터

경쟁 프로젝트가 스스로 발표한 성능 비교 수치는 선정 기준에도 비교표에도 넣지 않았습니다. 제작사가 낸 속도 비교는 측정 조건을 제3자가 확인하기 어렵고, 조건이 다르면 결과가 달라지기 때문입니다. Bifrost의 저장소 설명에는 속도 비교 문구가 실려 있지만 같은 이유로 비교 항목에 넣지 않았습니다[S11]. 비교에 쓴 정보는 공개 메타데이터와 재단 공지처럼 독자가 원문을 그대로 확인할 수 있는 자료로 한정됩니다.

5.1.2 스타 수 해석의 한계: API 게이트웨이 전체 수치와 LLM 전용 프로젝트 수치의 구분

Kong의 스타 44,242개는 2014년에 시작한 API 게이트웨이 프로젝트 전체에 쌓인 수치이며, LLM 기능만을 따로 센 값은 없습니다[S11]. Kong의 저장소 설명은 스스로를 API와 AI 게이트웨이로 소개하고 있으며, 스타를 누른 사용자 가운데 LLM 중계 기능 때문에 누른 비중은 메타데이터로 알 수 없습니다. 반면 LiteLLM은 2023년 7월에 저장소가 만들어진 뒤로 줄곧 LLM 호출을 다뤄 온 프로젝트이므로 스타 수가 LLM Gateway 용도의 관심과 가깝게 대응합니다. 따라서 두 수치를 같은 기준으로 놓고 크기만 비교하면 LLM Gateway로서의 채택 규모를 잘못 읽게 됩니다.

포크 수는 다섯 후보 가운데 LiteLLM만 조사돼 있으며, 나머지 네 후보의 포크 수는 비교표에서 비워 두었습니다. Kong은 시작 연도만 확인돼 있어 저장소 생성 월을 적지 않았습니다. 비어 있는 칸을 추정값으로 채우면 표 전체의 신뢰가 떨어지므로, 확인된 값만 싣고 나머지는 빈칸으로 남기는 방식을 택했습니다.

비교표의 수치는 조회값이 무엇을 센 값인지에 관한 아래 네 가지를 전제로 읽어야 합니다.

- 스타 수는 관심의 누적값이며 운영 중인 설치 수나 사용량을 뜻하지 않습니다.
- Kong의 스타 수에는 LLM과 무관한 API 게이트웨이 용도의 관심이 함께 들어 있으며, 저장소 설명이 API 게이트웨이를 함께 밝히는 Higress의 수치도 같은 주의를 두고 읽어야 합니다.
- 저장소 생성 시기가 이룰수록 스타가 쌓인 기간이 길므로, 생성 시기가 다른 프로젝트의 스타 수는 같은 기간의 값이 아닙니다.
- 프로젝트가 스스로 발표한 성능 수치는 표에 없으며, 성능은 도입 조직이 자기 환경에서 따로 확인해야 하는 항목입니다.

5.2 LiteLLM·Kong·Bifrost·Higress·Agent Router 비교표와 프로젝트 성격

다섯 후보는 같은 LLM Gateway 범주에 있지만, 저장소 설명과 재단 공지가 스스로 밝힌 범위를 기준으로 보면 프로젝트의 성격은 세 갈래로 나뉩니다. Kong과 Higress는 저장소 설명에서 API 게이트웨이를 함께 내세우는

프로젝트이고, Agent Router는 재단 공지에 따르면 모델 호출과 도구 호출을 함께 중계하는 에이전트 라우터입니다[S11][S12]. LiteLLM과 Bifrost는 LLM 호출 중계를 중심으로 하는 게이트웨이 묶음에 두었는데, 이 구분은 저장소 설명이 스스로 밝힌 범위를 따른 것이며 기능이나 성능을 따로 평가한 결과가 아닙니다. 성격이 다르다면 도입 조직이 이미 갖고 있어야 하는 운영 기반도 달라지므로, 스타 수와 함께 성격 구분을 후보 선택의 판단 항목으로 두어야 합니다.

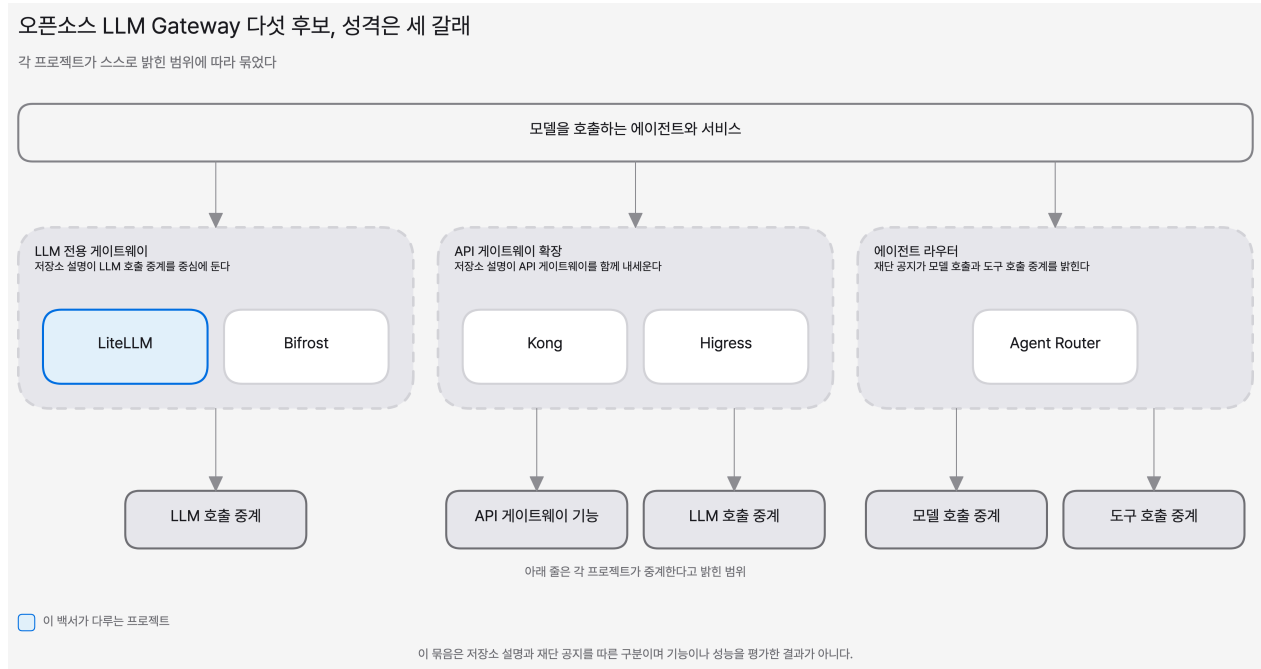


그림. 프로젝트 성격 세 갈래로 묶은 오픈소스 LLM Gateway 다섯 후보

5.2.1 다섯 후보의 스타·라이선스·저장소 생성 시기 비교

아래 표는 다섯 후보의 스타 수, 포크 수, 라이선스, 저장소 생성 시기, 프로젝트 성격을 한 줄씩 정리한 것이며, 프로젝트 성격 열은 저장소 설명이 스스로 밝힌 범위에 따라 적었습니다. LiteLLM은 스타 60,173개, 포크 12,046개이며 2023년 7월에 저장소가 만들어졌습니다. Kong은 44,242개, Higress는 9,492개, Bifrost는 8,565개, Agent Router는 2,188개의 스타를 갖고 있습니다. 모든 값은 2026년 10월 6일 GitHub API 조회 값입니다[S11].

프로젝트	스타	포크	라이선스	저장소 생성 시기	프로젝트 성격
LiteLLM	60,173	12,046	MIT (enterprise/ 디렉터리는 별도 라이선스)	2023년 7월	LLM 전용 게이트웨이
Kong	44,242		Apache-2.0	2014년 시작	API 게이트웨이 확장
Higress	9,492		Apache-2.0	2022년 10월	API 게이트웨이 확장
Bifrost	8,565		Apache-2.0	2025년 3월	LLM 전용 게이트웨이(저장소)

프로젝트	스타	포크	라이선스	저장소 생성 시기	프로젝트 성격
					설명이 밝힌 범위)
Agent Router	2,188		Apache-2.0	2024년 10월	에이전트 라우터

LiteLLM은 MIT 라이선스이고 나머지 네 후보는 모두 Apache-2.0 라이선스입니다[S11]. 두 라이선스 모두 상업적 이용과 수정, 재배포를 허용하므로 라이선스 종류만으로 후보가 탈락하지는 않습니다. 다만 LiteLLM 저장소의 `enterprise/` 디렉터리는 MIT가 아닌 별도 라이선스로 관리됩니다[S01]. LiteLLM을 검토하는 조직은 사용하려는 기능이 공개판 범위에 들어 있는지를 4장의 공개판·유료판 구분으로 함께 확인해야 합니다.

저장소 생성 시기는 프로젝트가 스타를 쌓아 온 기간을 알려 주며, 저장소 설명과 함께 보면 프로젝트의 범위를 짐작하는 단서가 됩니다. Kong은 2014년에 API 게이트웨이로 시작했고 Higress의 저장소는 2022년 10월에 만들어졌으며, 두 프로젝트의 저장소 설명은 각각 API와 AI 게이트웨이, AI 네이티브 API 게이트웨이라고 밝히고 있습니다[S11]. 이 장에서 두 프로젝트를 API 게이트웨이 확장형으로 묶은 근거는 저장소 설명이 API 게이트웨이를 함께 내세운다는 이 사실 하나입니다. 반면 LiteLLM은 2023년 7월, Agent Router는 2024년 10월, Bifrost는 2025년 3월에 저장소가 만들어졌으므로 세 저장소는 모두 2023년 이후에 생긴 프로젝트입니다.

5.2.2 Agent Router(옛 Envoy AI Gateway)의 1.0 출시와 Agentic AI Foundation 이전

Agent Router는 Envoy AI Gateway라는 이름으로 개발되던 프로젝트이며, 옛 저장소 주소로 접속하면 지금의 저장소로 넘어갑니다[S11]. 이 프로젝트는 2026년 6월에 1.0을 출시했고, 2026년 9월 9일 공지로 Agentic AI Foundation에 들어갔습니다[S12]. 조회 수치와 재단 공지에서 확인되는 연혁을 시간순으로 놓으면 아래 표와 같습니다.

시기	내용	근거
2024년 10월	저장소 생성	[S11]
2026년 6월	1.0 출시	[S12]
2026년 9월 9일	Agentic AI Foundation 합류 공지	[S12]

Agent Router는 모델 쪽에는 OpenAI 호환 인터페이스를 제공하고, 도구 쪽에는 MCP(Model Context Protocol, 에이전트가 외부 도구를 호출할 때 쓰는 규약) 라우터 하나를 제공합니다[S12]. 에이전트가 모델을 부르는 경로와 도구를 부르는 경로를 같은 프로젝트에서 중계하는 구성입니다. 재단 공지가 밝힌 이 범위에 따라 Agent Router는 비교표에서 LLM 전용 게이트웨이와 구분해 에이전트 라우터로 분류했습니다.

재단 공지에 따르면 Agent Router의 관리자 9석은 Bloomberg, Nutanix, AMD, Tetrade, Netflix 다섯 회사가 나눠 맡고 있으며 과반을 가진 회사가 없습니다. 같은 공지는 기여자가 132명이고 이들이 21개 조직에 속해 있다고 밝혔습니다[S12]. 특정 회사의 사업 방향에 따라 프로젝트 방향이 바뀌는 위험을 낮게 보려는 조직에는 이 관리 구조가 판단 근거가 됩니다. 반면 스타 수는 다섯 후보 가운데 가장 적으므로, 재단 거버넌스와 채택 규모를 각각 어느 정도 비중으로 볼지는 도입 조직이 정해야 합니다.

5.3 국내 공급 솔루션을 회사명 없이 구분한 제품 특징 유형 5가지

국내에서 공급되는 LLM Gateway 계열 솔루션은 제품이 중심으로 삼는 기능에 따라 다섯 유형으로 구분할 수 있습니다. 다섯 유형은 SaaS(Software as a Service, 공급자가 운영하는 서비스를 구독하는 방식)형 통합 과금 게이트웨이, GPU 플랫폼에 결합된 모델 라우터, 보안 필터링 프록시, 글로벌 API 게이트웨이의 국내 공급, 네트워크 어플라이언스형입니다. 유형별 설명은 이름이 뜻하는 범위까지만 적었으며, 유형마다 수치나 세부 기능을 단정하지 않았습니다. 실제 제품을 평가할 때는 유형을 먼저 구분한 다음 4장의 요건 평가표로 항목별 충족 여부를 따로 확인해야 합니다.

5.3.1 과금 통합형·GPU 플랫폼 결합형·보안 필터링형의 특징과 적합 조건

SaaS형 통합 과금 게이트웨이는 여러 프로바이더의 모델 사용료를 한 계정으로 모아 정산하는 것을 중심으로 삼습니다. 도입 조직은 공급자가 운영하는 엔드포인트를 사용하므로 게이트웨이 서버를 직접 둘 필요가 없습니다. 프로바이더별 계약과 결제를 줄이려는 조직, 게이트웨이 운영 인력을 따로 두기 어려운 조직에 맞습니다. 다만 호출 내용이 공급자의 서비스를 거치게 되므로 데이터 반출 기준이 엄격한 조직은 경유 경로와 로그 보관 방식을 먼저 확인해야 합니다.

GPU 플랫폼 결합형은 GPU 자원 관리 플랫폼이나 모델 서빙 플랫폼의 구성 요소로 제공되는 모델 라우터입니다. 이 유형은 플랫폼 위에 올린 모델들로 요청을 나눠 보내는 일을 중심으로 삼으며, 게이트웨이를 단독 제품으로 도입하기보다 플랫폼과 함께 도입하게 됩니다. 온프레미스 GPU 장비에 모델을 직접 올려 운영하고 그 플랫폼을 이미 쓰고 있거나 함께 도입할 조직에 맞습니다. 반면 외부 클라우드 프로바이더 호출까지 같은 경로로 모으려는 조직은 플랫폼 밖 모델의 지원 범위를 확인해야 합니다.

보안 필터링 프록시는 사용자와 모델 사이에서 요청과 응답을 검사하는 것을 중심으로 삼습니다. 이 유형은 민감 정보가 외부 모델로 나가는 것을 통제하는 데 초점을 두므로, 비용 귀속이나 폴백 같은 운영 기능은 제품마다 따로 확인해야 합니다. 외부 생성형 AI 서비스 사용을 통제해야 하는 보안 조직에 맞습니다. 세 유형의 중심 기능과 적합 조건은 아래 표와 같습니다.

유형	중심 기능	적합 조건
SaaS형 통합 과금 게이트웨이	여러 프로바이더 사용료의 통합 정산	게이트웨이를 직접 운영하지 않으려는 조직, 프로바이더별 계약을 줄이려는 조직
GPU 플랫폼에 결합된 모델 라우터	플랫폼에 올린 모델 사이의 요청 분배	온프레미스 GPU 플랫폼을 쓰고 있거나 함께 도입할 조직
보안 필터링 프록시	요청·응답 검사와 민감 정보 통제	외부 생성형 AI 사용을 통제해야 하는 보안 조직

유형을 구분한 뒤에는 제안서의 기능 목록을 LLM Gateway 기본 요건에 하나씩 대응시켜야 합니다. LiteLLM 공개판에 포함된 기능은 OpenAI 호환 게이트웨이, 가상 키, 지출 추적, 예산, 폴백, 요청·응답 로깅 여섯 가지이며, 이 목록은 다른 제품을 점검할 때도 같은 항목으로 쓸 수 있습니다[S04]. 예를 들어 보안 필터링 프록시를 제안받았다면 소비자별 키와 폴백이 기능 목록에 있는지를 별도로 확인하는 식입니다. 이렇게 대응시키면 중심 기능이 다른 제품들도 같은 표에서 비교할 수 있습니다.

5.3.2 글로벌 API 게이트웨이 공급형과 네트워크 어플라이언스형의 특징과 적합 조건

글로벌 API 게이트웨이 공급형은 해외에서 개발된 API 게이트웨이 제품을 국내 공급사가 구축과 기술 지원을 붙여 제공하는 유형입니다. 이 유형은 기존 API 게이트웨이의 인증, 라우팅, 정책 체계에 LLM 중계 기능을 추가하는 방식으로 접근합니다. 같은 API 게이트웨이를 이미 운영하고 있고 운영 절차와 담당 조직이 갖춰진 곳에 맞습니다. 다만 API 게이트웨이를 쓰지 않던 조직이 LLM 중계만을 위해 도입하면 API 게이트웨이 전체를 운영하는 부담이 함께 생깁니다.

네트워크 어플라이언스형은 네트워크 장비 형태로 공급되며 기존 네트워크 장비 운영 체계에 LLM 중계를 추가합니다. 이 유형은 네트워크와 보안 장비를 관리하는 조직이 LLM 트래픽도 같은 체계로 다루도록 하는 접근입니다. 장비 도입과 유지보수 절차가 정해져 있고 네트워크 조직이 LLM 호출 경로를 관리하게 될 곳에 맞습니다. 반면 모델 별칭이나 소비자별 예산처럼 애플리케이션 쪽 요구가 자주 바뀌는 환경에서는 설정 변경 절차가 그 속도에 맞는지를 확인해야 합니다.

두 유형은 이미 운영 중인 기반에 LLM 중계를 추가한다는 점이 같습니다. 도입 판단은 LLM Gateway 기능의 폭보다 기존 운영 체계와의 결합에서 얻는 이점을 중심으로 이뤄집니다. 가상 키, 예산, 풀백, 요청·응답 로깅 같은 기본 요건은 제품별로 따로 확인해야 합니다[S04]. 두 유형의 중심 기능과 적합 조건은 아래 표와 같습니다.

유형	중심 기능	적합 조건
글로벌 API 게이트웨이의 국내 공급	기존 API 게이트웨이 정책 체계에 LLM 중계 추가	같은 API 게이트웨이를 이미 운영하는 조직
네트워크 어플라이언스형	네트워크 장비 운영 체계에 LLM 중계 추가	네트워크 조직이 LLM 호출 경로를 관리하는 조직

5.4 다섯 후보 가운데 LiteLLM의 위치와 다른 후보가 더 맞는 조건

LiteLLM은 다섯 후보 가운데 스타 수가 가장 많고 포크는 12,046개이며, MIT 라이선스로 공개돼 있고, 100개 이상의 프로바이더를 OpenAI 형식으로 호출합니다[S11][S01]. 이 세 가지는 모두 공개 수치와 공식 문서에서 확인되는 사실이며, LiteLLM의 위치를 설명하는 근거는 이 세 가지로 한정됩니다. 반면 다른 후보가 더 맞는 조건은 프로젝트 성격의 차이에서 나오므로, 조건을 함께 확인해야 도입 판단이 한쪽으로 치우치지 않습니다.

다섯 후보를 비교할 때 모델의 토큰 생성 속도는 비교 항목에서 제외합니다. LLM Gateway는 추론을 수행하지 않으므로 어느 후보를 고르더라도 토큰 생성 속도는 뒤에 연결한 추론 서버와 프로바이더가 정합니다. 게이트웨이가 응답 시간에 더하는 것은 요청을 받아 넘기는 데 드는 자체 처리 시간이며, LiteLLM Proxy에서는 8장에서 설명하는 프록시 자체 지연 지표로 이 값을 따로 볼 수 있습니다. 따라서 후보를 고를 때는 처리 속도에 관한 주장보다 4장의 요건 평가표 항목을 기준으로 삼는 편이 판단에 유용합니다.

5.4.1 스타·포크 규모와 MIT 라이선스, 100개 이상 프로바이더로 본 LiteLLM의 위치

LiteLLM의 스타 60,173개와 포크 12,046개는 2026년 10월 6일 GitHub API 조회값이며, 스타 수는 다섯 후보 가운데 가장 큰 값입니다[S11]. 포크 수는 나머지 네 후보의 값이 비교표에 없으므로 크기를 견주지 않고 LiteLLM의 값으로만 적습니다. 최신 릴리스는 2026년 10월 3일에 나온 v1.104.0입니다[S11]. 채택 규모가 크면 운영 중에 생긴 문제의 해결 사례와 다른 도구와의 연동 정보를 구하기 쉽습니다. 5.1.2에서 밝힌 대로 스타 수는 관심의 누적값이므로, 이 수치는 운영 정보를 구하기 쉬운 정도를 가늠하는 용도로만 써야 합니다.

LiteLLM은 MIT 라이선스로 상업적 이용이 가능하고, 100개 이상의 프로바이더를 OpenAI 형식 하나로 호출합니다[S01][S02]. 프로바이더 지원 범위가 넓으면 온프레미스 추론 서버와 클라우드 모델을 같은 엔드포인트에 묶을 때 프로바이더별 변환 코드를 따로 만들 필요가 줄어듭니다. 가상 키, 지출 추적, 예산, 폴백, 요청·응답 로깅이 공개판에 들어 있으므로 기본 요건은 유료판 없이 갖출 수 있습니다[S04]. 아래 표는 LiteLLM의 위치를 근거와 함께 요약한 것입니다.

항목	LiteLLM의 값	근거
스타, 포크	60,173, 12,046 (스타 수는 다섯 후보 가운데 가장 많음)	[S11]
최신 릴리스	v1.104.0, 2026년 10월 3일	[S11]
라이선스	MIT, enterprise/ 디렉터리는 별도 라이선스	[S01]
프로바이더 지원	100개 이상을 OpenAI 형식으로 호출	[S01][S02]
공개판 기능	OpenAI 호환 게이트웨이, 가상 키, 지출 추적, 예산, 폴백, 요청·응답 로깅	[S04]

위 표는 LiteLLM이 다른 후보보다 빠르거나 안정적이라는 뜻을 담고 있지 않습니다. 성능과 안정성은 공개 메타 데이터로 비교할 수 없는 항목이고, 프로젝트가 스스로 낸 성능 주장은 비교 근거에서 제외했기 때문입니다. LiteLLM의 위치는 다섯 후보 가운데 스타 수가 가장 많고, 프로바이더마다 다른 LLM 호출 형식을 하나로 맞추려고 시작한 MIT 라이선스 프로젝트라는 범위로 한정됩니다[S08]. 자체 환경에서의 지연과 가용성은 8장의 모니터링 지표로 도입 조직이 직접 확인해야 합니다.

5.4.2 API 게이트웨이 확장형과 재단 거버넌스 프로젝트가 맞는 조직 조건

Kong이나 Higress 같은 API 게이트웨이 확장형은 같은 API 게이트웨이를 이미 운영하는 조직이 먼저 검토할 수 있는 후보입니다[S11]. 이런 조직은 인증 정책, 배포 절차, 담당 인력이 API 게이트웨이를 중심으로 갖춰져 있으므로 LLM 호출을 같은 체계에 올리면 새로 늘어나는 운영 대상을 줄일 수 있습니다. 반면 LLM 전용 게이트웨이를 따로 두면 관리할 서버와 설정 체계가 하나 더 생깁니다. 기존 API 게이트웨이의 운영 성숙도가 높을수록 확장형을 검토할 이유가 커집니다.

특정 회사에 치우치지 않은 프로젝트 관리 구조를 도입 조건으로 두는 조직은 Agent Router를 검토할 수 있습니다. Agent Router는 Agentic AI Foundation 소속이며 관리자 9석을 다섯 회사가 나눠 맡고 과반을 가진 회사가 없습니다[S12]. 반면 LiteLLM은 BerriAI라는 한 회사가 저장소를 관리하고 유료판을 함께 제공하는 프로젝트입니다[S01][S04]. 오픈소스 도입 기준에 재단 소속이나 중립 거버넌스 항목이 있는 조직은 이 차이를 판단 항목으로 올려야 합니다.

아래 표는 조직 조건과 먼저 검토할 후보 유형을 대응시킨 것입니다. 표의 대응은 프로젝트 성격에서 나오는 범위까지만 적은 것이며, 후보별 기능의 우열을 뜻하지 않습니다.

조직 조건	먼저 검토할 후보 유형	해당 프로젝트
여러 에이전트와 서비스의 LLM 호출을 한 엔드포인트로 모으려 하고, 별도 API 게이트웨이 기반이 없음	LLM 전용 게이트웨이(저장소 설명이 밝힌 범위)	LiteLLM, Bifrost
같은 API 게이트웨이를 이미 운영하고 있음	API 게이트웨이 확장형	Kong, Higress
재단 소속과 중립 거버넌스가 도입 조건임	재단 거버넌스 프로젝트	Agent Router
모델 호출과 MCP 도구 호출을 같은 프로젝트에서 중계하려 함	에이전트 라우터	Agent Router

위 조건에 해당하지 않는 조직, 즉 API 게이트웨이 기반 없이 여러 에이전트와 서비스의 모델 호출을 한 곳으로 모으려는 조직에는 LLM 전용 게이트웨이가 맞습니다. 그 가운데 LiteLLM은 다섯 후보 가운데 스타 수가 가장 많고 100개 이상의 프로바이더를 지원하며 기본 요건이 공개판에 들어 있습니다[S11][S01][S04]. 그러므로 API 게이트웨이 기반이 없고 재단 거버넌스가 도입 조건이 아니며 온프레미스 추론 서버와 클라우드 모델을 함께 쓰는 조직이라면, LiteLLM Proxy를 먼저 검토할 후보로 둘 수 있습니다. 다만 이 판단은 채택 규모와 공개판 범위에 근거한 것이므로, 지연과 안정성은 도입 조직이 자기 환경에서 따로 확인해야 합니다. 이 선택이 실제 구성에서 어떤 모습인지는 6장에서 오픈마루가 컨테이너 한 개와 설정 파일 한 개로 운영하는 구성의 모델 별칭과 폴백 설정으로 확인할 수 있습니다.

6장: 도커에서 운영하는 LiteLLM Proxy 구성: 모델 별칭 9개, 풀백 3건, 소비자 6종

오픈마루는 AI 에이전트가 쓰는 대화 모델과 임베딩 모델을 도커 컴포즈로 기동한 LiteLLM Proxy 컨테이너 한 개 뒤에 두고 운영합니다. LiteLLM Proxy는 플랫폼 팀이 조직 전체의 LLM 접근을 관리하려고 자체 인프라에 구축하는 OpenAI 호환 게이트웨이 서버입니다[S02]. 이 구성을 사례로 드는 이유는 에이전트 프로파일 16개와 주변 서비스가 실제로 한 길목을 거쳐 모델을 호출하고 있어, 4장의 요건이 운영 환경에서 어디까지 충족되고 어디가 비어 있는지를 구성 사실로 보일 수 있기 때문입니다.

에이전트 워크로드에서는 사용자 메시지 한 건이 모델 호출 한 건으로 끝나지 않습니다. 모델이 도구 호출을 요청할 때마다 에이전트는 도구 결과를 붙여 프록시를 다시 호출하므로, 대화 한 번이 프록시 요청 여러 건으로 늘어납니다. 추론 서버도 온프레미스 GPU 장비, 쿠버네티스 클러스터, 지사, 클라우드에 나뉘어 있습니다. 따라서 에이전트마다 백엔드 주소와 모델 이름을 직접 적어 두면 백엔드 하나가 바뀔 때 고쳐야 할 설정 파일이 프로파일 수만큼 생깁니다.

이 구성의 크기는 숫자 세 개로 요약되는데, 설정 파일 한 개에 모델 별칭 9개가 정의돼 있고 별칭 사이의 풀백은 3건이며 프록시를 호출하는 소비자는 6종입니다. 인증은 마스터 키 하나로 처리하고 데이터베이스와 캐시는 연결하지 않았습니다. 운영 대상은 컨테이너 한 개와 YAML 파일 한 개로 줄어들지만, 가상 키와 지표 수집처럼 저장소나 추가 설정이 필요한 기능은 쓰지 않는 상태입니다.

이 장은 배포 형태, 모델 별칭과 백엔드 묶음, 라우팅과 풀백, 소비자와 노출 방식 순서로 구성을 설명합니다. 한 계는 뒤로 모으지 않고 해당 구성이 처음 나오는 절에서 함께 적었습니다. 독자는 각 절의 표를 자기 환경의 구성과 나란히 놓고 별칭 수, 풀백 대상, 키 구조, 진입점 수를 건져 볼 수 있습니다. 마지막 항에서는 4장의 LLM Gateway 요건 평가표를 같은 항목으로 채워 충족 항목과 공백 항목을 가릅니다.

6.1 도커 컴포즈 단일 컨테이너 구성: 읽기 전용 YAML 한 개와 마스터 키 하나

오픈마루의 LiteLLM Proxy는 도커 컴포즈 프로젝트 하나에 컨테이너 한 개만 정의한 구성입니다. 3장에서 구분한 두 운영 형태 가운데 YAML 단독 구성에 해당하며, 이 형태가 운영 환경에서 어떤 모습인지 보이기 위해 구성 값을 그대로 옮겼습니다. LiteLLM SDK를 애플리케이션마다 넣는 방식에서는 프로바이더 키와 재시도 설정이 호출 코드마다 놓입니다. 반면 이 구성에서는 프록시를 거치는 경로의 백엔드 주소와 라우팅 설정이 모두 프록시의 설정 파일 한 곳에 있고, 소비자는 프록시 주소와 키 하나만 압니다.

6.1.1 컨테이너 한 개의 자원 제한과 데이터베이스·캐시 없는 구성

컨테이너는 LiteLLM 1.81 계열을 담아 커뮤니티가 다시 빌드한 파생 이미지를 `latest` 태그로 씁니다. LiteLLM 프로젝트가 배포하는 공식 이미지를 쓰는 구성이 아니며, 버전과 다이제스트도 고정하지 않았습니다. 컨테이너 자원은 메모리 2GB와 CPU 1.5 코어로 제한했고, 프로세스가 종료되면 도커가 컨테이너를 자동으로 다시 시작합니다. 프록시는 모델 추론을 직접 수행하지 않고 요청을 백엔드로 전달하는 역할만 하므로 GPU가 필요 없고, 이 정도 자원으로 에이전트 프로파일 16개와 호스트 서비스의 호출을 받고 있습니다. 설정은 읽기 전용으로 마운트한 YAML 파일 한 개가 전부이며, 컨테이너 안에서 설정이 바뀌는 경로는 없습니다.

데이터베이스인 Postgres와 캐시인 Redis는 연결하지 않았습니다. 컨테이너는 상태를 갖지 않으므로, 설정 파일을 고친 뒤 컨테이너를 다시 만드는 것으로 재배포가 끝나고 백업하거나 보관할 데이터가 없습니다. 다만 LiteLLM Proxy의 가상 키는 Postgres 데이터베이스를 요구하므로[S03], 이 구성에서는 가상 키와 그 위에서 동작하는 팀, 예산, 속도 제한, 사용 기록을 쓸 수 없습니다. 관리 화면은 열리지만 데이터베이스가 없어 화면에서 관리할 키나 팀이 없습니다.

인증 수단은 마스터 키 하나이며, 마스터 키는 원래 다른 키를 발급하는 관리자 키 역할을 하는데 이 구성에서는 모든 소비자가 호출할 때 쓰는 유일한 키로 쓰입니다. 프록시는 키가 맞지만 판정하고 호출한 소비자가 누구인지는 구분하지 않습니다. 아래 표는 배포 구성을 항목별로 정리한 것입니다.

항목	현재 구성	따라오는 결과
배포 형태	도커 컴포즈, 컨테이너 한 개	운영 대상이 적고 재배포가 단순함
이미지	LiteLLM 1.81 계열을 담은 커뮤니티 파생 이미지, <code>latest</code> 태그	버전과 다이제스트를 고정하지 않아 공급망 통제가 비어 있음
자원 제한	메모리 2GB, CPU 1.5 코어	프록시는 추론을 하지 않아 GPU 불필요
재시작 정책	자동 재시작	프로세스 종료 때 도커가 복구
데이터베이스·캐시	Postgres, Redis 모두 없음	가상 키, 예산, 속도 제한, 사용 기록 사용 불가
설정	읽기 전용 YAML 파일 한 개	변경 이력이 파일 하나로 모임
인증	마스터 키 하나	소비자를 구분하지 못함

6.1.2 30분 타임아웃과 미지원 파라미터 제거 설정

요청 타임아웃과 스트림 타임아웃은 모두 30분으로 설정했는데, 일반 대화 서비스라면 지나치게 긴 이 값이 에이전트 워크로드에서는 필요합니다. 기본 대화 모델은 약 26만 토큰의 컨텍스트를 받으며, 에이전트는 긴 문서와 도구 결과를 프롬프트에 누적한 채 추론을 요청합니다. 이때 타임아웃이 짧으면 모델이 정상적으로 응답을 만 들고 있는데도 프록시가 연결을 끊고, 그 실패가 재시도와 쿨다운으로 이어져 멀쩡한 백엔드가 라우팅 대상에서 빠지게 됩니다.

LiteLLM의 안정성 설정에는 요청 타임아웃이 재시도 횟수, 허용 실패 횟수, 쿨다운 시간과 함께 묶여 있습니다[S06]. 네 값은 서로 영향을 주므로 타임아웃만 따로 정할 수 없고, 이 구성은 재시도와 허용 실패 횟수가 각각 1 회여서, 타임아웃으로 생긴 실패가 재시도 한 번을 거친 뒤 곧 쿨다운으로 이어집니다. 타임아웃은 에이전트의 긴 추론과 도구 호출 반복을 고려해 30분으로 두었으며, 이렇게 하면 오래 걸리는 정상 요청이 타임아웃 실패로 집계되는 경우가 줄어듭니다.

호환 처리로는 프로바이더가 받지 않는 파라미터를 버리는 설정(`drop_params`)을 켜했습니다. 에이전트 프레임워크는 OpenAI 형식의 요청에 여러 선택 파라미터를 실어 보내는데, 온프레미스 추론 서버와 클라우드 모델이 받는 파라미터 집합은 서로 다릅니다. 이 설정이 없으면 별칭의 백엔드를 다른 추론 서버로 바꿨을 때 지원하지 않는 파라미터 때문에 요청이 거부될 수 있습니다. 이 설정은 소비자가 백엔드별 차이를 모르는 채로 같은 요청을 보낼 수 있게 하는 조건입니다.

설정	값	정한 이유
요청 타임아웃	30분	에이전트의 긴 추론과 도구 호출 반복
스트림 타임아웃	30분	스트리밍 응답도 같은 길이의 추론을 거침
미지원 파라미터 처리	버림	백엔드마다 다른 파라미터 집합 흡수

6.2 모델 별칭 9개와 백엔드 묶음: 온프레미스 vLLM, 2차 게이트웨이, 지사 추론 서버, 클라우드 경로

오픈마루의 설정 파일에는 모델 별칭 9개가 있고, 별칭마다 실제 백엔드 하나가 연결돼 있습니다. 이 묶음을 그 대로 소개하는 이유는 별칭이 소비자와 백엔드를 분리한다는 4장의 요건이 실제 파일에서 어떻게 쓰이는지를 보이기 위해서입니다. 소비자는 별칭 이름만 요청에 적고, 그 별칭이 어느 장비의 어느 모델로 연결되는지는 프록시의 설정 파일만 알고 있습니다. 백엔드를 유형으로 묶으면 온프레미스 GPU 장비의 vLLM, 쿠버네티스의 2차 게이트웨이, 다른 망의 llama.cpp 서버, 지사의 vLLM, 클라우드의 Azure OpenAI 다섯 가지입니다.

6.2.1 온프레미스 vLLM의 Qwen 계열 27B급 모델과 추론 컴·곰 별칭

기본 대화 모델 별칭은 온프레미스 GPU 장비에서 vLLM으로 서빙하는 Qwen 계열 27B급 오픈 모델로 연결됩니다. 두 에이전트 스택의 프로파일 16개가 모두 이 별칭을 기본값으로 쓰므로, 이 별칭의 백엔드가 응답하지 않으면 에이전트 전체의 대화가 영향을 받습니다. 프록시는 별칭을 백엔드 주소로 바꾸면서 설정 파일에 정의한 샘플링 파라미터와 추론 과정 출력을 끄는 설정을 요청에 함께 넣습니다. 에이전트는 샘플링 값이나 추론 관련 옵션을 요청마다 지정할 필요가 없고, 별칭 이름과 프록시 주소만 설정에 적어 두면 됩니다.

같은 모델을 추론(thinking)을 켜 채 호출하는 별칭이 하나 더 있으며, 두 별칭은 같은 vLLM 서버의 같은 모델을 가리키고 프록시가 요청에 넣는 파라미터만 다릅니다. LiteLLM의 설정에서 별칭은 소비자가 부르는 모델 이름이고, 그 아래에 실제 호출 대상과 파라미터를 적은 항목이 배포됩니다[S06]. 그래서 한 모델에 별칭을 여러 개 두면 용도별 파라미터 묶음에 이름을 붙이는 효과가 생깁니다.

이 방식에서는 소비자가 모델 서버의 파라미터 규격을 몰라도 됩니다. 예를 들어 빠른 응답이 필요한 에이전트 대화는 추론을 끈 별칭을, 단계가 긴 분석 작업은 추론을 켜 별칭을 이름으로 고릅니다. 다음은 두 별칭의 정의 형태를 가상의 이름과 주소로 줄여 적은 예입니다.

```
model_list:
- model_name: chat-default      # 추론 끄, 에이전트 기본값
  litellm_params:
    model: openai/qwen-27b
    api_base: http://vllm-backend/v1
    api_key: sk-...
- model_name: chat-default-thinking # 같은 모델, 추론 켜
  litellm_params:
    model: openai/qwen-27b
    api_base: http://vllm-backend/v1
```

```
api_key: sk-...  
# 추론을 켜는 파라미터를 여기에 덧붙인다(생략)
```

6.2.2 쿠버네티스 2차 게이트웨이를 거치는 별칭 3개와 35B급 모델

별칭 3개는 추론 서버로 직접 연결되지 않고 쿠버네티스에서 운영하는 또 하나의 LiteLLM 게이트웨이를 거칩니다. 이 글에서는 도커의 LiteLLM Proxy와 구분하기 위해 이를 2차 게이트웨이라고 부릅니다. 2차 게이트웨이 뒤에는 Qwen 계열 35B급 모델이 있고, 별칭 2개가 이 모델을 각각 추론을 끈 설정과 켜 설정으로 호출합니다. 나머지 별칭 하나는 2차 게이트웨이를 거쳐 온프레미스의 27B 모델로 다시 연결되는 우회 경로입니다.

게이트웨이 뒤에 게이트웨이를 두면 폴백 대상을 다른 운영 환경에 둘 수 있습니다. 도커 호스트 쪽에서 보면 2차 게이트웨이는 OpenAI 호환 엔드포인트 하나일 뿐이므로, 쿠버네티스 쪽에서 모델 서버를 늘리거나 교체해도 도커의 설정 파일은 그대로입니다. 온프레미스 GPU 장비의 vLLM이 응답하지 않으면 쿠버네티스 쪽 게이트웨이 뒤의 다른 모델이 요청을 받습니다.

다만 이 구조는 2차 게이트웨이에 대한 의존을 만듭니다. 폴백 3건이 모두 2차 게이트웨이의 별칭을 향하므로, 2차 게이트웨이가 응답하지 않는 동안에는 폴백 경로 전체가 함께 사라집니다. 요청이 게이트웨이를 두 번 거치면 타임아웃, 재시도, 키 관리도 두 곳에서 따로 설정해야 합니다. 결국 2차 게이트웨이는 폴백 대상이면서 동시에 별도로 관리해야 하는 운영 대상입니다.

모델 별칭 9개가 백엔드로 풀리는 경로와 폴백 3건

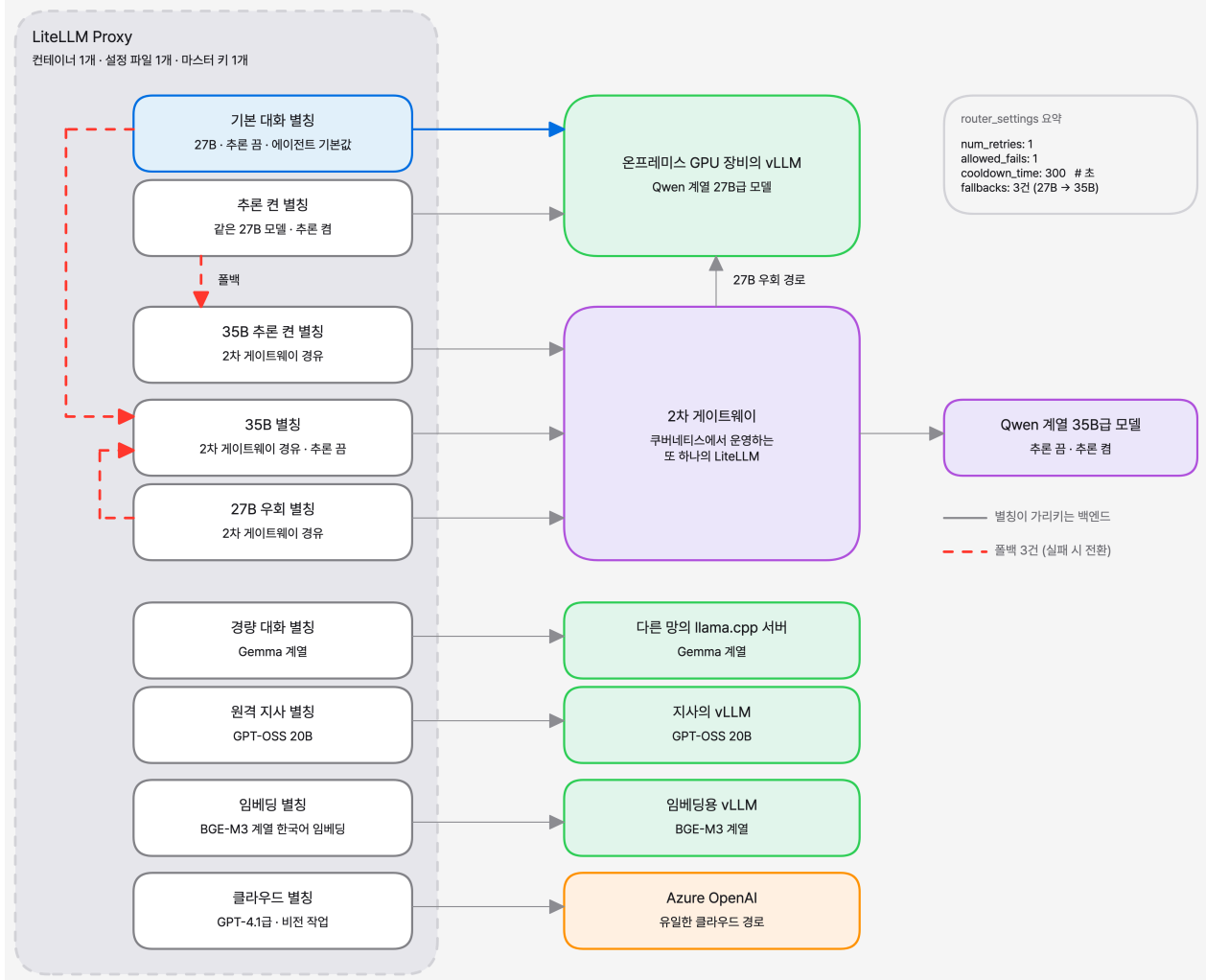


그림. LiteLLM Proxy의 모델 별칭 9개와 백엔드의 연결, 폴백 경로 3건

6.2.3 경량 모델, 지사 추론 서버, 임베딩, 클라우드 경로의 용도별 별칭

나머지 별칭 4개는 용도가 정해진 백엔드로 연결됩니다. 경량 대화 모델 별칭은 사내 다른 망의 llama.cpp 서버에서 실행하는 Gemma 계열 모델을 가리키며, 망 조건에 따라 프록시에서 이 서버로 연결되지 않을 때가 있습니다. 원격 지사 모델 별칭은 지사에 있는 vLLM의 GPT-OSS 20B를 별칭 하나로 묶은 것입니다. 두 별칭은 폴백 대상으로 지정하지 않았으므로, 백엔드가 응답하지 않으면 해당 별칭을 부른 요청은 실패로 끝납니다.

임베딩 별칭은 vLLM에서 서빙하는 BGE-M3 계열 한국어 임베딩 모델로 연결되고 벡터는 1024차원이며, 호스트에서 실행하는 슬라이드 검색 서비스가 이 별칭을 사용합니다. 대화 모델과 임베딩 모델이 같은 프록시의 별칭으로 정의돼 있어, 검색 서비스도 에이전트와 같은 주소와 같은 키로 모델을 호출합니다.

클라우드 모델 별칭은 Azure OpenAI의 GPT-4.1급 모델로 연결되며, 이 구성에서 프록시를 거치는 유일한 클라우드 경로입니다. 이 별칭은 이미지가 들어간 PDF를 읽는 작업처럼 비전 입력이 필요한 작업에 지정합니다. 에이전트의 기본 대화는 온프레미스 모델이 처리하고, 비전 작업처럼 이 별칭을 지정한 작업만 클라우드로 나갑니다. 아래 표는 별칭 9개를 유형 7가지로 묶어 백엔드와 용도를 정리한 것입니다.

별칭 유형	별칭 수	실제 백엔드	용도와 조건
기본 대화 모델	1	온프레미스 vLLM, Qwen 계열 27B급, 추론 곱	모든 에이전트의 기본값, 컨텍스트 약 26만 토큰
추론 컨 별칭	1	같은 vLLM의 같은 모델	파라미터로만 구분
2차 게이트웨이 경유	3	쿠버네티스의 LiteLLM 게이트웨이	35B급 추론 곱·곱, 27B 우회 경로
경량 대화 모델	1	다른 망의 llama.cpp, Gemma 계열	망 조건에 따라 연결되지 않을 때가 있음
원격 지사 모델	1	지사의 vLLM, GPT-OSS 20B	지사 추론 서버를 별칭 하나로 묶음
임베딩	1	vLLM, BGE-M3 계열 한국어 임베딩	1024차원, 슬라이드 검색 서비스가 사용
클라우드 모델	1	Azure OpenAI, GPT-4.1 급	유일한 클라우드 경로, 비전 작업에 지정

별칭 9개 가운데 여덟 개가 조직이 직접 운영하는 추론 서버나 게이트웨이로 향한다는 점은 운영 부담이 어디에 있는지도 알려 줍니다. 클라우드 경로에서는 프로바이더가 모델 서빙을 맡지만, 온프레미스 경로에서는 GPU 장비와 추론 서버, 모델 파일을 모두 조직이 관리해야 합니다. 별칭은 이 백엔드들을 소비자에게 드러내지 않을 뿐 운영 대상을 줄여 주지는 않습니다. 위 표는 소비자에게는 호출할 이름의 목록이고, 운영자에게는 직접 관리해야 할 백엔드의 목록입니다.

6.3 재시도 1회·쿨다운 300초·폴백 3건의 동작 순서와 2차 게이트웨이 의존

오픈마루 구성의 라우팅 설정은 재시도 1회, 허용 실패 1회, 쿨다운 300초, 폴백 3건으로 이루어집니다. 이 값을 그대로 적는 이유는 폴백이 어떤 장애를 받아 주고 어떤 장애는 받지 못하는지를 구체적인 설정으로 설명할 수 있기 때문입니다. 별칭마다 배포는 하나씩이고 부하 분산은 하지 않으므로, 이 구성에서 요청이 다른 백엔드로 전환되는 경로는 폴백 3건이 전부이며, 폴백이 받아 주는 것은 모델 서버 쪽 장애입니다. 프록시 컨테이너 자체가 멈추면 폴백을 판단할 주체가 없으므로 어떤 경로도 동작하지 않습니다.

6.3.1 27B 별칭에서 2차 게이트웨이의 35B 별칭으로 향하는 폴백 3건

폴백 3건은 모두 27B 모델을 가리키는 별칭에서 출발해 2차 게이트웨이의 35B 별칭에서 끝납니다. 기본 27B 별칭은 35B 별칭으로, 27B 추론 별칭은 35B 추론 별칭으로, 2차 게이트웨이를 거치는 27B 우회 별칭은 35B 별칭으로 전환됩니다. LiteLLM에서 폴백은 한 별칭이 실패하면 다른 별칭으로 넘긴다는 대응 목록으로 적습니다[S06]. 다음은 그 형태를 가상의 별칭 이름으로 옮긴 예이며, 재시도와 쿨다운 값은 이 구성의 값입니다.

```
router_settings:
  num_retries: 1
  allowed_fails: 1
  cooldown_time: 300
  fallbacks:
```

```
- chat-default: ["chat-large"]
- chat-default-thinking: ["chat-large-thinking"]
- chat-default-bypass: ["chat-large"]
```

이 폴백의 대상은 같은 모델의 복제본이 아니고 규모가 다른 모델입니다. 따라서 폴백이 일어난 뒤에는 에이전트가 받는 응답의 품질과 속도가 평소와 달라질 수 있습니다. 에이전트는 별칭 이름으로만 요청하므로, 프록시 쪽 기록이나 지표가 없으면 폴백이 일어났는지를 운영자가 가려내기 어렵습니다. 폴백은 설정하는 것으로 끝나지 않고 발생 여부를 관측할 수 있어야 하며, LiteLLM은 폴백 성공과 실패를 Prometheus 지표로 제공합니다 [S05].

최근 사흘의 컨테이너 로그에는 폴백이나 쿨다운이 발생한 기록이 없습니다. 이는 그 기간에 온프레미스 vLLM이 요청을 계속 처리했다는 사실을 말해 줄 뿐이며, 장애 상황에서 폴백 경로가 기대한 대로 동작한다는 근거가 되지 않습니다. 현재 로그에는 요청별 모델 별칭이 남지 않으므로, 폴백이 일어나더라도 어느 별칭의 요청이 전환됐는지를 로그만으로 가려내기 어렵습니다. LiteLLM은 배포별 쿨다운 발생도 Prometheus 지표로 제공하지만 [S05], 이 구성에서는 지표 엔드포인트가 꺼져 있어 그 값을 볼 수 없습니다.

6.3.2 실패 뒤 재시도 1회, 300초 쿨다운, 폴백으로 이어지는 처리 순서

에이전트의 요청이 실패했을 때 프록시가 처리하는 순서는 세 단계이며, 첫 단계에서 프록시는 같은 별칭으로 요청을 한 번 다시 보냅니다. 재시도도 실패하면 해당 배포를 300초 동안 쿨다운 상태로 두어 라우팅 대상에서 제외합니다. 그다음 폴백 목록에 적힌 2차 게이트웨이의 별칭으로 요청을 보내고, 그 응답을 원래 요청의 응답으로 에이전트에 돌려줍니다. 재시도 횟수, 허용 실패 횟수, 쿨다운 시간 각각의 정의는 4장의 안정성 설정 설명과 같습니다 [S06].

이 처리 순서는 에이전트의 요청 흐름 안에서 일어나는데, 사용자가 메신저로 메시지를 보내면 해당 프로파일의 게이트웨이 프로세스가 역할 문서, 스킴, 메모리로 프롬프트를 만들어 기본 별칭으로 대화 완성 요청을 보냅니다. 모델이 도구 호출을 요청하면 에이전트는 컨테이너 안에서 MCP(Model Context Protocol, 에이전트가 외부 도구를 호출할 때 쓰는 규약) 서버, 검색, 웹 추출 도구를 실행하고, 도구 결과마다 프록시를 한 번 더 호출합니다. 따라서 대화 한 번에 속한 여러 요청 가운데 일부는 27B 모델이, 나머지는 폴백된 35B 모델이 처리하는 경우가 생길 수 있습니다.

쿨다운 300초 동안에는 기본 별칭으로 들어오는 요청이 폴백 모델로 갑니다. 기본 별칭은 모든 에이전트의 기본 값이므로, 이 5분 동안 에이전트 전체의 트래픽이 2차 게이트웨이의 35B 모델 하나로 모입니다. 쿨다운이 짧으면 아직 복구되지 않은 백엔드로 요청이 되돌아가 실패가 반복되고, 길면 복구된 백엔드를 두고도 폴백 모델을 계속 쓰게 됩니다. 그래서 쿨다운 시간은 추론 서버가 재시작해 모델을 다시 적재하는 데 걸리는 시간에 맞춰 정해야 합니다.

6.3.3 별칭당 배포 하나와 단일 컨테이너에서 오는 가용성 한계

이 구성에는 가용성 측면의 한계가 두 가지 있으며, 첫째는 별칭마다 배포가 하나뿐이라는 점입니다. LiteLLM은 같은 모델 이름으로 배포를 여러 개 정의하면 그 사이에서 부하를 분산합니다 [S06]. 하지만 이 구성은 별칭 하나에 배포 하나만 두었으므로, 기본 별칭으로 들어온 요청은 모두 온프레미스 vLLM 한 곳으로 갑니다. 평상시에는 요청을 나눠 받을 두 번째 배포가 없고, 장애 때에는 규모가 다른 모델로 넘어가는 폴백만 있습니다.

둘째는 프록시가 단일 컨테이너라는 점으로, 모든 에이전트 프로파일이 프록시 주소 하나를 모델 엔드포인트로 쓰므로, 프록시 컨테이너가 멈추면 온프레미스 추론 서버가 정상이어도 에이전트 전체가 모델을 호출하지 못합

니다. 자동 재시작 정책은 프로세스가 종료된 경우를 복구하는 장치이며, 이 구성에는 응답하지 않는 프록시를 감지해 교체하는 헬스체크와 복제가 없습니다. 폴백도 프록시 안에서 판단하는 기능이므로 프록시 자체의 장애에는 대응하지 못합니다.

콘텐츠 공장 스택은 이 한계의 일부를 에이전트 프레임워크 수준의 폴백으로 보완합니다. 기본 모델 호출이 실패하면 같은 프록시의 35B 별칭으로 다시 요청하도록 설정돼 있습니다. 다만 이 폴백도 같은 프록시를 거치므로 프록시가 멈춘 경우에는 도움이 되지 않습니다. 길목을 하나로 모은 구성에서는 길목 자체의 복제와 헬스체크가 가용성의 전제 조건이 되며, 이 구성에서는 그 부분이 비어 있습니다.

구성 요소	현재 상태	장애 때 결과
별칭당 배포	하나	부하 분산 없음, 다른 모델로 폴백
폴백 대상	2차 게이트웨이의 35B 별칭	2차 게이트웨이가 멈추면 폴백 경로 없음
프록시 컨테이너	한 개, 자동 재시작	프록시가 멈추면 모든 에이전트의 모델 호출 중단
폴백 미지정 별칭	경량, 지사, 임베딩, 클라우드, 35B 별칭	백엔드 장애가 요청 실패로 이어짐

6.4 소비자 6종의 마스터 키 공유와 호스트 경유 단일 진입점

오픈마루의 LiteLLM Proxy를 호출하는 소비자는 6종이며, 모두 호스트를 거치는 진입점 하나로 들어와 같은 마스터 키를 씁니다. 소비자 목록과 진입 방식을 사례로 적는 이유는 이 두 가지가 가상 키를 설계할 때 그대로 입력이 되기 때문입니다. 누가 프록시를 쓰는지 알아야 키를 몇 개로 나눌지 정할 수 있고, 어떻게 들어오는지 알아야 접근 통제를 어디에 걸지 정할 수 있습니다. 에이전트 프로파일별 역할은 7장에서 다루므로 여기서는 소비자 유형만 구분합니다.

6.4.1 업무 에이전트 스택에서 호스트 서비스까지 소비자 6종의 호출 형태

소비자 6종은 호출 형태에 따라 에이전트 스택, 에이전트 주변 도구, 호스트 서비스로 나뉩니다. 에이전트 스택은 오픈소스 에이전트 프레임워크 Hermes Agent로 구성된 컨테이너 두 개입니다. 업무 에이전트 스택은 이름 있는 프로파일 11개와 기본 프로파일을, 콘텐츠 공장 스택은 프로파일 3개와 기본 프로파일을 실행합니다. LiteLLM Proxy가 OpenAI 호환 엔드포인트를 제공하므로[S02], 두 스택의 프로파일 16개는 모두 OpenAI 호환 방식으로 프록시 주소와 기본 별칭을 모델 설정에 적어 두었고 주소와 키는 컨테이너 환경변수로 한 번만 주입합니다.

소비자	호출 형태	에이전트 루프 경유
업무 에이전트 스택	프로파일별 게이트웨이 프로세스의 대화 처리, 예약 작업	경유
콘텐츠 공장 스택	같은 프레임워크 이미지를 다이제스트로 고정한 두 번째 컨테이너	경유

소비자	호출 형태	에이전트 루프 경유
에이전트 콘솔	모델 목록 조회, 사용자가 실행하는 1 토큰 점검 호출	경유하지 않음
에이전트 컨테이너 안의 보조 스크립 트	PDF 읽기, 명함 인식 같은 작업에서 프록시를 직접 호출	경유하지 않음
슬라이드 검색 서비스	호스트 프로세스의 임베딩 호출과 대 화 호출	경유하지 않음
영상 제작 도구 일부	호스트 프로세스의 직접 호출	경유하지 않음

소비자 6종 가운데 4종은 에이전트 루프를 거치지 않고 프록시를 호출합니다. 보조 스크립트는 PDF 읽기, 명함 인식, 이력서 평가, 콘텐츠 요약, 영상 첫 댓글 작성을 맡으며, 에이전트 컨테이너 안에서 실행되지만 모델 호출은 에이전트 루프 밖에서 프록시로 직접 보냅니다. 에이전트 콘솔은 사용자의 대화를 에이전트의 대시보드 API로 전달하므로 그 대화의 LLM 호출은 에이전트 컨테이너에서 나가고, 콘솔 자신이 프록시에 보내는 것은 모델 목록 조회와 점검 호출뿐입니다. 예약 작업 가운데 LLM을 쓰는 작업은 프로파일의 스케줄러가 기동해 대화와 같은 경로로 프록시를 호출하며, 스크립트만 실행하는 작업은 프록시를 호출하지 않습니다.

최근 사흘 동안 프록시가 받은 요청은 대화 완성 약 2,500건, 임베딩 약 40건, 모델 목록 조회 약 280건입니다. 같은 기간에 잘못된 모델 이름으로 인한 400 오류가 7건, 키 없이 들어온 요청이 11건 있었습니다. 다만 이 건수를 소비자별로 나눌 수는 없는데, 6종이 모두 같은 마스터 키를 쓰기 때문에 프록시에서는 요청이 어느 소비자에게서 왔는지 구분할 정보가 없습니다. 따라서 이 소비자 목록은 가상 키를 도입할 때 발급할 키의 목록과 같고, 지출 추적, 모델 접근 통제, 예산은 키가 나뉜 뒤에야 소비자 단위로 적용할 수 있습니다[S03].

소비자 6종은 진입점 하나로, 프록시 밖 경로 3건은 직접

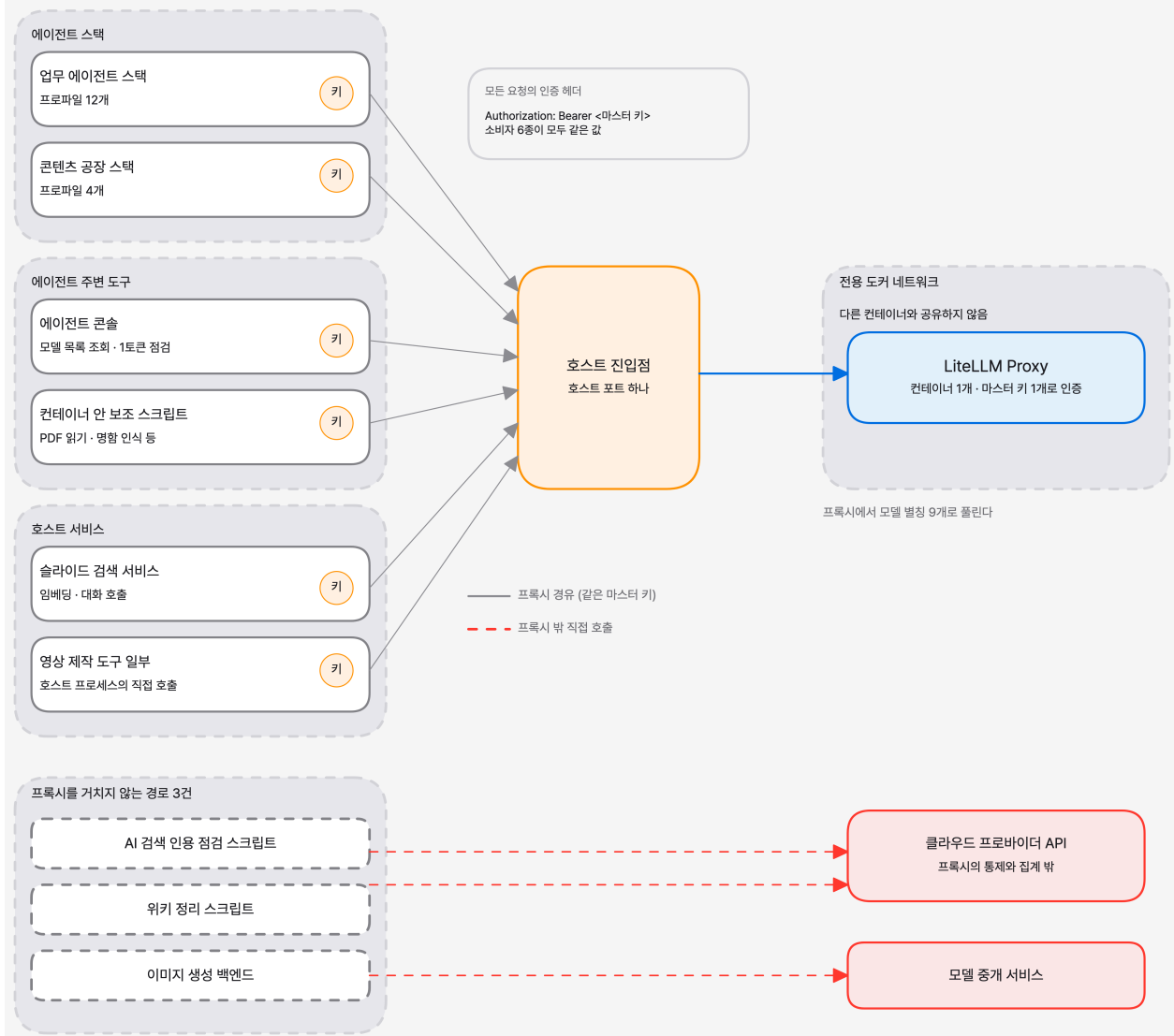


그림. 소비자 6종이 호스트 경유 진입점 하나로 LiteLLM Proxy에 모이는 구조와 프록시 밖 경로 3건

6.4.2 전용 도커 네트워크와 호스트 경유 진입점 하나로 제한한 노출 방식

프록시 컨테이너는 전용 도커 네트워크에 혼자 있고 다른 컨테이너와 네트워크를 공유하지 않으며, 호스트 포트 하나로만 열려 있습니다. 에이전트 컨테이너도 도커 내부 이름으로 프록시에 직접 연결하지 않고 호스트 게이트웨이 주소를 거쳐 이 포트에 들어옵니다. 컨테이너 안의 에이전트와 호스트에서 실행하는 검색 서비스가 프록시에 도달하는 경로는 같습니다.

진입점이 하나이면 접근 통제와 관측을 적용할 위치도 하나로 정해집니다. 에이전트 스택은 검색 네트워크와 웹 추출 네트워크처럼 여러 도커 네트워크에 연결되는데, 프록시가 그 네트워크들에 함께 있으면 같은 네트워크의 다른 컨테이너도 프록시를 호출할 수 있게 됩니다. 반면 프록시를 전용 네트워크에 격리하면 프록시에 도달하는 방법이 호스트 포트 하나로 한정됩니다. 새 소비자가 생겨도 방화벽 규칙이나 인증 정책을 추가할 위치가 달라지지 않습니다.

다만 진입점을 좁힌 것만으로 접근 통제가 끝나지는 않습니다. 호스트 포트에 도달할 수 있고 마스터 키를 가진 프로세스는 별칭 9개를 모두 호출할 수 있으며, 클라우드 모델 별칭도 예외가 아닙니다. 최근 사흘 동안 키 없이 들어온 요청이 11건 있었는데, 소비자별 키가 없는 현재 구성에서는 그 요청을 보낸 주체를 프록시에서 가려낼 수 없습니다. 네트워크 격리는 도달 범위를 줄이고, 소비자별 권한 구분은 모델 접근 통제 기능을 가진 가상 키가 맡아야 하는 일로 남아 있습니다[S03].

6.4.3 LLM Gateway 요건 평가표로 본 현재 구성의 충족 항목과 공백 항목

4장의 LLM Gateway 요건 평가표를 같은 항목 순서와 같은 판정 기준으로 현재 구성에 적용하면, 여덟 항목 가운데 충족이 하나, 부분 충족이 넷, 공백이 셋입니다. 판정 값은 충족, 부분 충족, 공백 세 가지입니다. 설정 파일만으로 동작하는 기능은 충족이나 부분 충족으로 판정됐고, 데이터베이스나 추가 설정이 필요한 기능과 이미지 배포 방식에 달린 공급망 통제는 공백으로 남았습니다.

구분	요건 항목	판정	현재 구성의 사실
기본	OpenAI 호환 엔드포인트와 모델 별칭	부분 충족	별칭 9개가 한 엔드포인트로 묶였으나 프록시를 거치지 않는 경로 3건이 남아 있음
기본	키 관리	부분 충족	프로바이더 키는 프록시가 갖지만 소비자 전부가 마스터 키 하나를 공유
기본	폴백과 안정성 설정	충족	재시도 1회, 쿨다운 300초, 폴백 3건
기본	요청·응답 로깅	부분 충족	컨테이너 표준 출력 로그로 요청·오류 건수만 확인됨. 모델 별칭은 남지 않음
상세	소비자별 가상 키와 예산	공백	데이터베이스가 없어 쓸 수 없음
상세	지연·첫 토큰 시간 관측	공백	지표 엔드포인트와 콜백이 꺼져 있음
상세	공급망 통제	공백	커뮤니티 파생 이미지를 <code>latest</code> 태그로 사용
상세	LLM Gateway 가용성	부분 충족	자동 재시작은 있으나 인스턴스가 하나

충족과 부분 충족으로 판정한 기본 요건은 길목을 하나로 모은 것만으로 얻은 결과입니다. 모델을 바꿀 때 에이전트 프로파일 16개의 설정은 그대로 두고 프록시 설정 파일의 별칭 정의만 고치면 되며, 온프레미스 추론 서버의 장애는 폴백이 2차 게이트웨이로 넘겨 받습니다. 다만 첫 항목이 충족에 이르지 못한 이유는 프록시를 거치지 않는 호출 경로 3건이 남아 있기 때문입니다. AI 검색 인용을 주마다 점검하는 스크립트와 위키 정리 스크립트는 클라우드 프로바이더 API를 직접 호출하고, 이미지 생성 백엔드 하나는 모델 중개 서비스를 직접 호출합니다. 키

관리도 프로바이더 키를 프록시 한 곳에 모은 데까지는 이르렀지만, 소비자 6종이 마스터 키 하나를 함께 쓰므로 부분 충족에 머뭙니다.

공백 세 항목은 필요한 조치가 서로 다릅니다. 가상 키, 예산, 지출 추적은 Postgres를 연결해야 쓸 수 있습니다[S03]. 반면 지연, 첫 토큰 시간(TTFT, Time To First Token), 폴백 발생 지표는 데이터베이스 없이 Prometheus 콜백 설정만으로 얻을 수 있습니다[S05]. 두 기능 모두 공개판에 포함돼 있으므로[S04][S05], 이 두 공백은 라이선스와 무관하게 구성 선택에서 비롯한 것입니다. 셋째 공백인 공급망 통제는 제품 기능이 아닌 배포 방식의 문제이므로 이미지를 바꾸고 버전을 고정하는 조치로 채웁니다.

공급망 통제 항목은 이미지의 출처와 이미지 태그를 구분해서 보아야 합니다. LiteLLM의 공식 도커 이미지는 의존성 버전을 고정해 두고 있어 2026년 3월의 PyPI 패키지 사고에서 영향을 받지 않았습니다[S07]. 이 구성이 쓰는 것은 공식 이미지가 아닌 커뮤니티 파생 이미지이므로 공식 이미지에 관한 그 설명을 그대로 가져올 수 없고, 태그도 `latest`여서 컨테이너를 다시 만들 때 어떤 버전이 내려올지를 운영자가 통제하지 못합니다. 4장의 판정 기준은 파생 이미지이거나 버전·다이제스트를 확인하지 않는 경우를 공백으로 두므로 이 항목의 판정은 공백입니다. 가용성 항목은 부분 충족인데, 자동 재시작이 프로세스 종료를 복구하고 모델 서버 장애는 폴백이 받지만 프록시 인스턴스가 하나뿐이고 헬스체크가 없기 때문입니다. 관측 공백의 세부은 8장에서, 부분 충족과 공백을 채우는 순서는 9장에서 이어집니다.

7장: AI 에이전트 토폴로지와 각 에이전트의 역할: 스택 2개와 프로파일 16개의 요청 흐름

오픈마루는 오픈소스 에이전트 프레임워크 Hermes Agent로 구성된 AI 에이전트 프로파일 16개를 도커 컨테이너 두 개에 나눠 운영하고, 이 프로파일들의 모델 호출을 모두 LiteLLM Proxy 한 곳으로 보냅니다. Hermes Agent는 Nous Research가 MIT 라이선스로 공개한 에이전트이며, 사용자가 지정한 엔드포인트의 모델을 코드 변경 없이 바꿔 쓸 수 있습니다[S10]. 이 구성을 사례로 드는 이유는 에이전트 수가 늘어날 때 모델 운영도 함께 복잡해지는지를 실제 배치에서 확인할 수 있기 때문입니다.

에이전트가 늘어날 때 관리 대상은 두 종류로 나뉩니다. 역할 문서, 메신저 채널, 예약 작업, MCP(Model Context Protocol) 서버 연결은 프로파일마다 따로 두어야 하므로 에이전트 수에 비례해 늘어납니다. 반면 모델 설정은 16개 프로파일이 모두 같아서 기본 대화 모델 별칭 하나, OpenAI 호환 방식, LiteLLM Proxy 주소 세 항목으로 끝납니다. 따라서 모델을 교체하거나 추론 서버를 옮길 때 고치는 곳은 프록시 설정 파일의 별칭 정의의 한 곳이고, 프로파일 16개의 설정은 그대로 둡니다.

이 장은 다섯 가지를 순서대로 설명합니다. 컨테이너와 네트워크의 배치를 먼저 보이고, 그다음 Hermes Agent의 프로파일이 따로 갖는 자원과 함께 쓰는 자원을 구분합니다. 이어서 업무 에이전트 스택의 역할표와 예약 작업, 콘텐츠 공장 스택의 생성·발행 분리, 에이전트 콘솔과 스킬 마켓플레이스의 연결 방식을 다룹니다. 마지막으로 사용자의 메시지가 모델에 도달하고 답이 돌아오기까지의 요청 흐름 7단계를 정리합니다.

같은 구성에는 한계도 함께 들어 있습니다. 프로파일들이 함께 쓰는 컨테이너 환경변수에 프록시 주소와 마스터 키가 있어서 모든 에이전트가 같은 키로 호출하고, 프록시를 거치지 않는 호출 경로도 3건이 남아 있습니다. 대화 한 번이 프록시 요청 여러 건으로 나뉘는 점도 함께 알아 두어야 하며, 요청 단위 기록만으로는 대화 단위의 비용과 지연을 알 수 없습니다. 마스터 키 공유, 프록시 밖 경로 3건, 요청 단위로만 남는 기록이라는 세 사실은 8장에서 모니터링 방식을 정할 때 그대로 전제가 됩니다.

7.1 컨테이너 22개와 컴포즈 프로젝트 9개의 배치: 에이전트 스택 2개와 검색·웹 추출 네트워크

오픈마루의 에이전트 구성은 한 호스트(Apple Silicon 맥)에서 실행되는 컨테이너 22개와 도커 컴포즈 프로젝트 9개로 이루어집니다. 에이전트가 실행되는 컨테이너는 그중 두 개이고, 나머지는 LiteLLM Proxy, 검색과 웹 추출 도구, 관리 화면, 스킬 보관소 같은 주변 계층입니다. 스택 사이의 네트워크와 볼륨 경계를 먼저 확인해 두면 장애와 권한이 어디까지 영향을 주는지 판단할 수 있습니다.

7.1.1 에이전트 스택 2개의 네트워크 경계와 분리된 데이터 볼륨

에이전트는 업무 에이전트 스택과 콘텐츠 공장 스택 두 개로 나뉘어 실행됩니다. 업무 에이전트 스택은 Hermes Agent 컨테이너 한 개에 이름 있는 프로파일 11개와 기본 프로파일을 두고, 콘텐츠 공장 스택은 두 번째 컨테이너에 프로파일 3개와 기본 프로파일을 둡니다. 두 스택을 더하면 프로파일은 16개입니다. 콘텐츠 공장 스택은 업무 에이전트 스택과 같은 Hermes Agent 이미지를 쓰되 이미지를 다이제스트(이미지 내용에서 계산한 고유 식별값)로 고정해 두었습니다. 다이제스트로 고정한 컨테이너는 같은 태그에 새 이미지가 올라와도 지정한 이미지를 그대로 씁니다.

두 스택은 각자 기본 네트워크를 따로 갖고 데이터 볼륨을 공유하지 않습니다. 그래서 한쪽 스택의 세션 DB와 메모리는 다른 쪽 스택에서 보이지 않고, 컨테이너도 따로이므로 한쪽을 재시작하는 작업이 다른 쪽 컨테이너를 건드리지 않습니다. 두 스택이 함께 연결되는 네트워크는 검색 네트워크와 웹 추출 네트워크 두 개뿐이며, 이 두 네트워크는 컴포즈 프로젝트 밖에서 만든 외부 네트워크입니다.

LiteLLM Proxy는 이 네트워크들 어디에도 속하지 않습니다. 프록시 컨테이너는 전용 도커 네트워크에 혼자 있고 호스트 포트 하나로만 열려 있으므로, 두 에이전트 스택을 포함한 모든 소비자는 호스트 게이트웨이 주소를 거쳐 프록시에 들어옵니다. 에이전트 컨테이너와 프록시 컨테이너는 도커 네트워크를 공유하지 않으며, 모델 호출이 지나는 진입점은 호스트의 포트 하나로 한정됩니다.

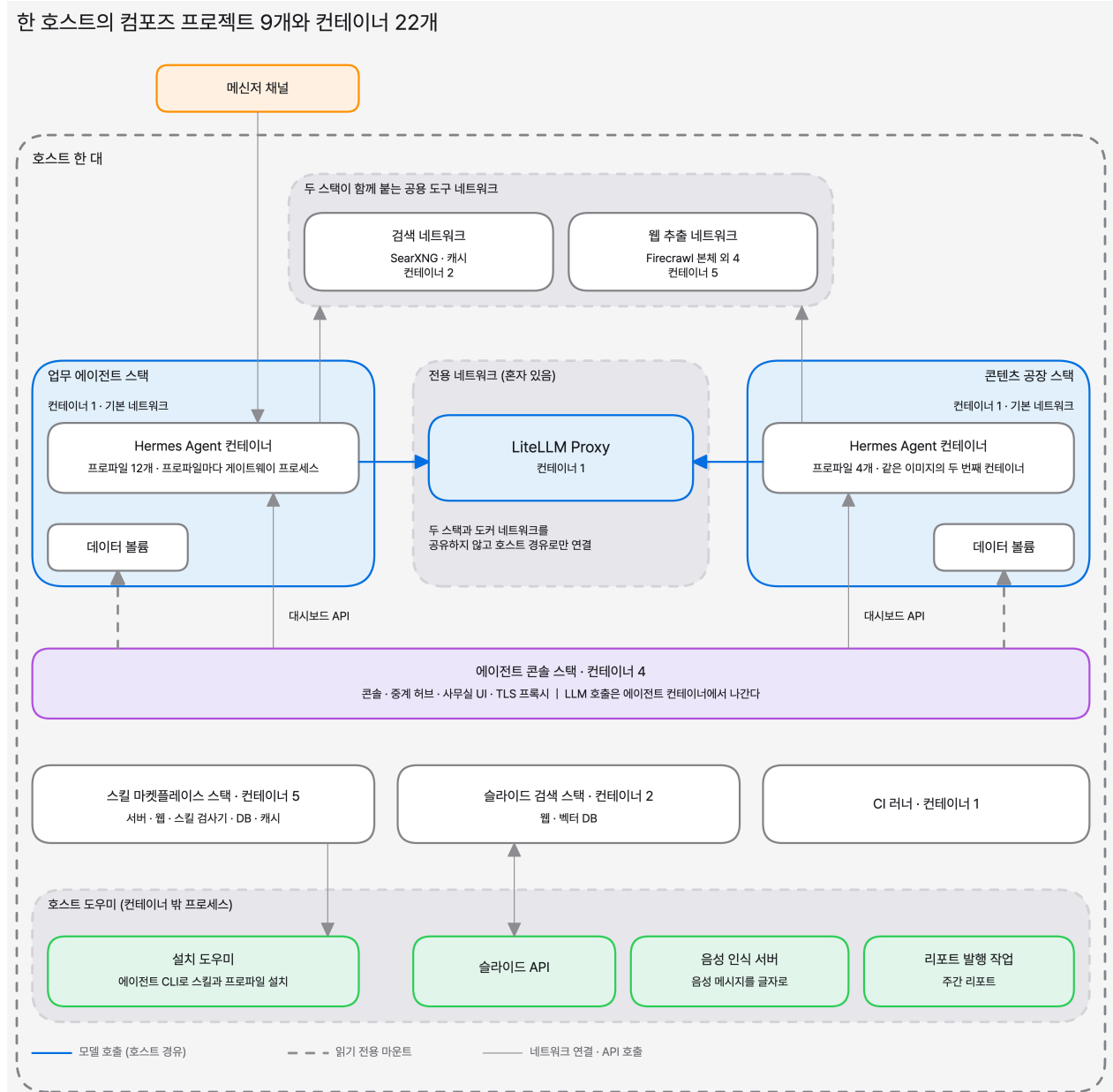


그림. 한 호스트에 배치된 컴포즈 프로젝트 9개와 컨테이너 22개: 에이전트 스택 2개, 공용 도구 네트워크 2개, LiteLLM Proxy 전용 네트워크의 경계

7.1.2 SearXNG 검색 네트워크와 Firecrawl 웹 추출 네트워크의 공용 도구 계층

웹 검색과 웹 추출은 두 에이전트 스택이 함께 쓰는 공용 도구 계층입니다. 검색 네트워크에는 SearXNG 컨테이너와 캐시가 있고, 웹 추출 네트워크에는 Firecrawl 컨테이너 5개(본체, 메시지 큐, 캐시, DB, 브라우저)가 있습니다. 프로파일 16개는 모두 웹 검색에 SearXNG를, 웹 추출에 Firecrawl을 쓰도록 설정돼 있으므로, 에이전트를 더 만들어도 검색 엔진이나 웹 추출 서버를 새로 실행할 필요가 없습니다. 모델 호출을 LiteLLM Proxy 한 곳으로 모은 것과 같은 원리가 도구 계층에도 적용된 셈입니다.

컴포즈 프로젝트 9개의 구성 요소와 컨테이너 수, 역할은 다음 표와 같습니다. 프로젝트별 컨테이너 수를 더하면 22개입니다. 에이전트 스택 2개를 뺀 나머지 7개 프로젝트는 LiteLLM Proxy, 에이전트가 호출하는 도구 계층, 에이전트를 관리하는 계층, 그 밖의 서비스로 나뉩니다.

컴포즈 프로젝트	구성 요소	컨테이너 수	역할
LiteLLM Proxy	단일 컨테이너	1	모든 소비자의 모델 호출을 받는 LLM Gateway
업무 에이전트 스택	Hermes Agent 컨테이너	1	이름 있는 프로파일 11개와 기본 프로파일 실행
콘텐츠 공장 스택	다이제스트로 고정한 Hermes Agent 컨테이너	1	프로파일 3개와 기본 프로파일 실행
검색	SearXNG, 캐시	2	두 에이전트 스택의 웹 검색 도구
웹 추출	Firecrawl 본체, 메시지 큐, 캐시, DB, 브라우저	5	두 에이전트 스택의 웹 추출 도구
에이전트 콘솔	콘솔, 중계 허브, 사무실 UI, TLS 프록시	4	에이전트 데이터 조회와 대화·변경 요청 전달
스킬 마켓플레이스	서버, 웹, 스킬 검사기, DB, 캐시	5	에이전트에 설치할 스킬과 프로파일 보관
슬라이드 검색	웹, 벡터 DB	2	슬라이드 검색 서비스의 웹과 벡터 DB
CI 러너	러너	1	CI 러너(용도는 이 문서 범위 밖)
합계	컴포즈 프로젝트 9개	22	—

컨테이너 밖 호스트에서 직접 실행하는 도우미도 네 가지가 있습니다. 음성 인식 서버, 슬라이드 API, 마켓플레이스 설치 도우미, 주간 리포트 발행 작업입니다. 음성 인식 서버는 메신저로 들어온 음성 메시지를 글자로 바꾸고, 마켓플레이스 설치 도우미는 스킬과 프로파일 설치 요청을 확인해 에이전트 CLI로 설치합니다. 호스트에서 실행되는 서비스 가운데 LiteLLM Proxy를 호출하는 것은 임베딩과 대화 호출을 쓰는 슬라이드 검색 서비스와 영상 제작 도구 일부입니다. 반면 음성 인식은 호스트의 로컬 음성 인식 서버가 처리하므로 프록시와 무관합니다.

7.2 Hermes Agent 프로파일 구조: 프로파일별 자원, 공유 자원, 16개 프로파일의 동일한 모델 설정

Hermes Agent에서 에이전트 하나는 프로파일 하나에 대응합니다. 오픈마루의 구성에서는 컨테이너 하나 안에서 프로파일마다 게이트웨이 프로세스가 하나씩 실행되고, 모든 프로파일의 모델 설정은 기본 27B 별칭, OpenAI 호환 방식, LiteLLM Proxy 주소로 같습니다. 이 사례를 드는 이유는 에이전트 16개의 역할 차이가 모델 설정이 아닌 다른 자원에서 만들어진다는 점을 프로파일 구조로 확인할 수 있기 때문입니다.

7.2.1 프로파일마다 따로 갖는 설정 파일과 세션 DB, 공유하는 스킬 디렉터리

Hermes Agent의 프로파일은 별도의 홈 디렉터리입니다. 프로파일 하나는 설정 파일, 환경 파일, 역할 문서, 메모리, 세션, 예약 작업, 상태 DB를 자기 디렉터리에 따로 두며, 두 에이전트 프로세스가 같은 프로파일을 함께 쓰면 안 됩니다[S10]. 그래서 영업 에이전트의 대화 기록과 재무 에이전트의 대화 기록은 서로 다른 세션 DB(SQLite)에 쌓이고, 한 프로파일의 메모리가 다른 프로파일의 답변에 섞이지 않습니다.

오픈마루의 구성에서 프로파일이 따로 갖는 자원과 함께 쓰는 자원은 다음 표와 같이 나뉩니다. 따로 갖는 자원은 에이전트의 성격과 기억을 정하고, 함께 쓰는 자원은 에이전트가 쓸 수 있는 기능과 모델 접속 정보를 정합니다.

구분	자원	정해지는 것
프로파일별	역할 문서	에이전트의 성격, 임무, 답변 방식
프로파일별	설정 파일	연결할 MCP 서버와 도구 구성
프로파일별	메신저 봇 토큰	사용자가 대화하는 채널
프로파일별	예약 작업 목록	정해진 시각에 실행할 업무
프로파일별	세션 DB(SQLite), 메모리	대화 기록과 장기 기억
공유	스킬 디렉터리, 보조 스크립트	문서 읽기, 요약 같은 공용 기능
공유	컨테이너 환경변수	LiteLLM Proxy 주소와 키

공유 자원 가운데 컨테이너 환경변수는 마스터 키 공유의 구조적 원인입니다. 프록시 주소와 키가 프로파일 단위 파일이 아닌 컨테이너 환경변수에 있으므로, 같은 컨테이너의 프로파일은 모두 같은 키로 LiteLLM Proxy를 호출합니다. 프록시에서 보면 영업 에이전트의 요청과 비서 에이전트의 요청은 같은 마스터 키를 가진 같은 종류의 요청입니다. 소비자별로 지출 추적, 모델 접근 통제, 예산을 적용하려면 가상 키가 있어야 하고, 가상 키는 Postgres 데이터베이스를 연결해야 쓸 수 있습니다[S03].

7.2.2 프로파일별 게이트웨이 프로세스와 프로파일마다 다른 MCP 서버 연결

Hermes Agent에서 게이트웨이는 메신저 계정을 에이전트에 연결하는 프로세스를 가리킵니다. 이 문서에서 LLM Gateway라고 부르는 LiteLLM Proxy와는 다른 대상이므로, 여기서는 게이트웨이 프로세스라고 구분해 씁니다. 게이트웨이 프로세스는 프로파일마다 서비스 항목이 하나씩 만들어지고 한 장비에서 여러 프로파일의 프로세스를 함께 실행할 수 있으며, 도커에서는 프로세스 감독기인 s6가 이 서비스들을 관리합니다[S10]. 오픈

마루의 에이전트 컨테이너에서도 프로파일마다 게이트웨이 프로세스가 하나씩 실행되고, 프로세스 감독기가 이 프로세스들을 관리합니다.

도구는 모든 프로파일에 공통인 것과 프로파일마다 다른 것으로 나뉩니다. 웹 검색은 SearXNG, 웹 추출은 Firecrawl, 터미널은 컨테이너 로컬 환경을 쓰며 이 세 가지는 16개 프로파일에 똑같이 설정돼 있습니다. 반면 MCP 서버는 프로파일마다 다르게 연결합니다. Hermes Agent는 설정 파일에 적힌 MCP 서버를 시작할 때 찾아 도구로 등록하고 서버별로 노출할 도구를 고를 수 있으므로[S10], 인프라 점검 에이전트에는 APM과 위키를, 영업 에이전트에는 CRM과 위키를, 공공 데모 에이전트에는 법령과 공문서 MCP를 연결하는 식의 구성이 가능합니다.

따라서 에이전트의 역할을 구분하는 요소는 역할 문서, 연결된 MCP 서버, 예약 작업 세 가지입니다. 재무 에이전트와 마케팅 에이전트는 같은 27B급 모델을 같은 별칭으로 호출하지만, 읽을 수 있는 데이터와 수행하는 업무가 다릅니다. 모델 계층을 모든 에이전트의 공용 자원으로 둘 수 있는 근거가 여기에 있으며, 에이전트를 하나 더 만들 때 LiteLLM Proxy 설정은 수정하지 않아도 됩니다.

7.3 업무 에이전트 스택 12개 프로파일의 역할과 예약 작업 94건

업무 에이전트 스택에는 이름 있는 프로파일 11개와 기본 프로파일이 있습니다. 프로파일마다 맡은 업무, 대화 채널, 연결된 MCP 서버, 예약 작업 수가 다르고, 최근 14일의 세션 수도 에이전트마다 크게 차이가 납니다. 소비자별 사용량을 나눠 봐야 하는 이유가 이 차이에서 나옵니다.

7.3.1 인프라 점검에서 영업까지 업무 에이전트의 역할·채널·연결 MCP

업무 에이전트 12개 프로파일의 역할은 다음 표와 같습니다. 마지막 열의 예약 작업 수는 LLM을 호출하는 작업만 센 값이며, 아홉 에이전트의 값을 더하면 48건입니다.

에이전트	역할	채널	연결된 MCP	LLM 사용 예약 작업
인프라 점검 에이전트	APM 데이터를 읽어 장애를 1차 조사, 읽기 전용	메신저	APM, 위키	3건(매일·매주·매월 점검)
엔지니어링 에이전트	백서와 블로그용 기술 조사, 엔지니어링 칸반 지표	메신저	위키 2종	6건
재무 에이전트	손익 조회와 보고	메신저	위키	2건
인사 에이전트	이력서 선별, 평가 보조	메신저	위키	1건
비서 에이전트	일정, 리마인더, 회의록, 브리핑	메신저	위키	5건
주식 리포트 에이전트 A	시장 리포트 작성, 본문은 위키에 요약은 메신저로 전달	메신저	위키	4건

에이전트	역할	채널	연결된 MCP	LLM 사용 예약 작업
주식 리포트 에이전트 B	종목 리포트 전문을 메신저로 전달	메신저	없음	5건
마케팅 에이전트	웹 분석, 검색 콘솔, 영상 지표, 브랜드 리포트	메신저	위키	13건
영업 에이전트	CRM 영업 기회, 입찰 공고, 영업 보고	메신저	CRM, 위키	9건
공공 데모 에이전트 2종	중앙부처와 지자체 공무원 업무 시연(법령 조회, 문서 초안)	콘솔 경유	법령, 공문서	0건
기본 프로파일	범용 운영 보조	없음	CRM, 위키	0건

채널이 메신저인 아홉 에이전트는 각자 자기 메신저 봇을 갖습니다. 사용자가 어느 에이전트의 봇에 메시지를 보냈는지에 따라 그 프로파일의 게이트웨이 프로세스가 요청을 받습니다. 공공 데모 에이전트 2종은 메신저 봇 없이 에이전트 콘솔을 거쳐 대화하며, 기본 프로파일은 채널 없이 운영 보조 용도로만 씁니다.

이 역할표는 가상 키의 발급 단위와 모니터링의 구분 단위를 정할 때 후보가 됩니다. 인프라 점검 에이전트는 읽기 전용으로 APM 데이터를 조회하고 영업 에이전트는 CRM 데이터를 다루므로, 두 에이전트에 허용할 모델과 예산은 다를 수 있습니다. 다만 현재 구성에서는 이 표의 에이전트가 모두 같은 마스터 키를 쓰므로, 프록시에서는 이 표의 행 단위로 사용량을 나눌 수 없습니다.

7.3.2 LLM 사용 48건, 스크립트 전용 30건으로 나뉘는 예약 작업과 세션 분포

업무 에이전트 스택에 등록된 예약 작업은 모두 94건입니다. Hermes Agent의 예약 작업은 자연어나 크론 식으로 등록하고 결과를 메신저로 보낼 수 있으며, LLM 없이 스크립트만 실행하는 방식도 지원합니다[S10]. 오픈마루의 94건은 이 두 방식과 꺼 둔 작업으로 나뉘며, 구성은 다음 표와 같습니다.

예약 작업 유형	건수	LiteLLM Proxy 호출
LLM을 쓰는 작업	48건	실행할 때마다 호출
스크립트만 실행하는 작업	30건	호출 없음
꺼 둔 작업	16건	호출 없음

스크립트 전용 작업 30건은 LLM을 호출하지 않고 스크립트만 실행하므로 LiteLLM Proxy에 요청을 보내지 않습니다. 그래서 예약 작업 수가 많다고 해서 프록시 부하가 그만큼 큰 것은 아니며, 부하를 가늠할 때는 LLM을 쓰는 48건만 계산에 넣어야 합니다. 예약 작업이 쓸 모델은 작업별 지정, 크론 기본 모델, 에이전트 기본 모델 순으로 정해지는데[S10], 오픈마루의 프로파일은 기본 모델이 모두 같은 27B 별칭이므로, 작업에 모델을 따로 지정하지 않은 예약 작업의 호출은 같은 별칭으로 들어갑니다.

최근 14일의 세션 수를 보면 에이전트별 차이가 뚜렷합니다. 영업 에이전트가 약 340건으로 가장 많고 그 대부분이 사람의 대화가 아닌 예약 작업입니다. 그다음은 비서 에이전트 약 70건, 주식 리포트 에이전트 2종 각 약

45건, 마케팅 에이전트 약 30건, 엔지니어링 에이전트 약 25건 순입니다.

에이전트	최근 14일 세션 수	비고
영업 에이전트	약 340	대부분 예약 작업
비서 에이전트	약 70	
주식 리포트 에이전트 2종	각 약 45	
마케팅 에이전트	약 30	
엔지니어링 에이전트	약 25	

이 분포에서 두 가지를 읽을 수 있습니다. 첫째, 영업 에이전트 하나의 세션이 표에 있는 나머지 에이전트의 합 (약 215)보다 많으므로 에이전트별 사용량은 고르지 않습니다. 둘째, 예약 작업은 스케줄러가 정해진 시각에 실행하므로 실행 시각이 겹치면 호출이 특정 시각에 몰릴 수 있습니다. 따라서 비용과 부하를 볼 때는 사람의 대화와 예약 작업을 구분하고 에이전트별로 나눠 봐야 하는데, 이 구분은 소비자별 가상 키가 있어야 프록시에서 가능합니다[S03].

예약 작업은 사람이 지켜보지 않는 시간에 모델을 호출한다는 점에서도 대화와 다릅니다. 대화에서는 사용자가 답을 보고 멈출 수 있지만, 예약 작업에서 에이전트가 도구 호출을 예상보다 오래 반복하면 그 사실을 바로 알아챌 사람이 없습니다. 이런 호출을 한도로 묶는 수단이 가상 키에 연결하는 예산과 속도 제한이며, 두 기능은 데이터베이스가 있어야 쓸 수 있습니다. 지금 구성에서는 예산과 속도 제한을 쓰지 않으므로, LLM을 쓰는 예약 작업 48건의 호출량은 프록시에서 제한되지 않습니다.

7.4 콘텐츠 공장 스택의 생성·발행 분리와 에이전트 콘솔·스킬 마켓플레이스의 위치

콘텐츠 공장 스택은 글을 생성하는 에이전트와 바깥으로 내보내는 에이전트를 나눈 구성이고, 에이전트 콘솔과 스킬 마켓플레이스는 에이전트를 조회하고 늘리는 관리 계층입니다. 세 구성 요소를 사려로 드는 이유는 에이전트의 권한과 수가 달라져도 모델 호출 경로는 새로 생기지 않는다는 점을 각각 다른 방식으로 보여 주기 때문입니다.

7.4.1 발행 자격증명을 발행 에이전트 하나에만 둔 콘텐츠 공장의 역할 분리

콘텐츠 공장 스택에는 문서 에이전트, 블로그 에이전트, 발행 에이전트 세 프로파일과 기본 프로파일이 있습니다. 세 에이전트의 역할과 권한은 다음 표와 같습니다.

에이전트	역할	발행 자격증명
문서 에이전트	백서 생성. 칸반 작업으로 일을 받는다	없음
블로그 에이전트	블로그 원고 생성	없음
발행 에이전트	워드프레스 3곳과 위키에 발행	있음(유일)

발행 자격증명은 발행 에이전트 프로파일 하나에만 있습니다. 문서 에이전트와 블로그 에이전트는 원고를 만들 수 있지만 워드프레스나 위키에 발행할 자격증명이 없으므로, 생성 단계에서 잘못된 내용이 만들어지더라도 그

결과가 바깥으로 직접 나가지 않습니다. 권한을 프로파일 단위로 줬던 이 방식은 LiteLLM Proxy의 키에도 같은 원리로 적용할 수 있습니다. 에이전트마다 가상 키를 따로 발급하면 키 하나가 유출됐을 때 교체할 범위가 그 에이전트 하나로 줄어듭니다.

콘텐츠 공장에 일을 전달하는 수단은 칸반입니다. Hermes Agent의 칸반은 모든 프로파일이 함께 쓰는 SQLite 기반 작업판이며, 작업마다 독립된 OS 프로세스가 작업자로 실행됩니다[S10]. 오픈마루의 구성에서도 칸반에 작업을 만들고 담당 프로파일을 지정하면 작업마다 새 작업자 프로세스가 실행돼 그 작업을 처리합니다. 백서 한 편을 만드는 작업이 길어져도 다른 작업은 별도 프로세스에서 진행됩니다.

이 스택은 프레임워크 수준의 풀백도 따로 갖습니다. 기본 모델이 응답하지 않으면 Hermes Agent가 같은 프록시의 35B 별칭으로 요청을 다시 보내도록 설정돼 있습니다. LiteLLM Proxy에도 27B 별칭에서 35B 별칭으로 가는 풀백이 있으므로, 콘텐츠 공장의 요청은 프록시와 프레임워크 두 계층에서 보호를 받습니다. 다만 두 풀백 모두 대상이 쿠버네티스에서 운영하는 2차 게이트웨이 뒤의 35B급 모델이므로, 2차 게이트웨이가 함께 멈춘 경우에는 두 풀백 모두 넘어갈 대상이 없어집니다.

7.4.2 읽기 전용 볼륨과 대시보드 API 로 연결되는 에이전트 콘솔

에이전트 콘솔 스택은 콘솔, 중계 허브, 사무실 UI, TLS 프록시 네 컨테이너로 이루어집니다. 이 스택은 두 에이전트 스택의 데이터 볼륨을 읽기 전용으로 마운트해 에이전트의 데이터를 읽고, 대화나 변경이 필요할 때는 에이전트의 대시보드 API를 호스트 경유로 호출합니다. 따라서 콘솔은 에이전트의 데이터를 볼 수 있지만 볼륨에 직접 쓸 수는 없고, 모든 변경은 에이전트가 제공하는 API를 거쳐서만 반영됩니다.

콘솔 자신은 LLM을 호출하지 않습니다. 콘솔이 LiteLLM Proxy에 보내는 요청은 모델 목록 조회와, 사용자가 버튼을 눌렀을 때 실행되는 1토큰 점검 호출 두 가지뿐입니다. 콘솔에서 사용자가 에이전트와 대화하면 그 메시지는 대시보드 API를 거쳐 에이전트에 전달되고, 모델 호출은 에이전트 컨테이너에서 나갑니다. 관리 화면을 추가하면서도 모델 호출 경로는 늘리지 않은 구성입니다.

이 구성은 호출 주체를 조사할 때 판단을 단순하게 만듭니다. 콘솔이 직접 모델을 호출했다면 콘솔용 키, 콘솔용 모델 설정, 콘솔용 사용량 집계가 따로 필요했을 것입니다. 반면 지금은 공공 데모 에이전트처럼 콘솔로만 대화하는 프로파일의 호출도 메신저로 대화하는 프로파일과 같은 경로, 같은 별칭으로 프록시에 들어옵니다.

7.4.3 스킬과 프로파일을 보관하는 마켓플레이스와 호스트 설치 도우미

스킬 마켓플레이스 스택은 서버, 웹, 스킬 검사기, DB, 캐시 다섯 컨테이너로 이루어지며, 에이전트에 설치할 스킬과 프로파일을 보관합니다. 마켓플레이스 스택이 맡는 일은 보관이고, 설치는 마켓플레이스 컨테이너가 아닌 호스트의 설치 도우미가 맡습니다. 이 도우미가 30초마다 설치 요청을 확인해 에이전트 CLI(명령줄 도구)로 스킬이나 프로파일을 설치합니다.

그 결과 보관하는 쪽과 설치하는 쪽이 나뉩니다. 스킬과 프로파일을 보관하는 일은 마켓플레이스 스택의 다섯 컨테이너가 하고, 에이전트 CLI를 실행해 설치하는 일은 호스트 도우미 하나가 합니다. 콘텐츠 공장에서 생성과 발행을 나눈 것과 비슷하게, 에이전트를 바꾸는 동작이 한 곳을 거치도록 한 구성입니다. 마켓플레이스 스택에는 스킬 검사기 컨테이너도 들어 있습니다.

마켓플레이스로 새 프로파일을 설치해 에이전트를 하나 늘려도 모델 쪽에서 바뀌는 것은 없습니다. 새 프로파일은 컨테이너 환경변수의 프록시 주소와 키를 그대로 쓰고 기본 27B 별칭을 호출하므로, LiteLLM Proxy 설정

파일과 추론 서버 구성은 수정하지 않습니다. 에이전트를 늘릴 때 프록시와 추론 서버 쪽에서 고칠 것이 없다는 점이 이 구성의 특징입니다.

7.5 메신저에서 모델까지 요청 흐름 7단계와 프록시 밖 경로 3건

사용자의 메시지 하나는 메신저 봇, 게이트웨이 프로세스, LiteLLM Proxy, 추론 서버를 차례로 거치고, 도구 호출이 있으면 프록시를 여러 번 왕복합니다. 오픈마루의 요청 흐름을 사례로 드는 이유는 대화 한 번이 프록시에서 요청 몇 건으로 기록되는지, 그리고 어떤 호출이 프록시를 아예 거치지 않는지를 단계별로 확인할 수 있기 때문입니다.

7.5.1 메시지 수신에서 스트리밍 응답까지의 단계와 LiteLLM Proxy 의 별칭 해석

요청 흐름은 다음 7단계로 정리됩니다. 1단계부터 5단계까지는 사용자의 대화 한 번이 처리되는 순서이고, 6단계와 7단계는 대화가 메신저가 아닌 다른 경로에서 시작되는 경우입니다.

단계	처리 주체	처리 내용
1	메신저 봇, 호스트 음성 인식 서버	사용자가 에이전트의 메신저 봇에 메시지를 보낸다. 음성이면 음성 인식 서버가 먼저 글자로 바꾼다
2	프로파일의 게이트웨이 프로세스	역할 문서, 스킴, 메모리로 프롬프트를 만들고 LiteLLM Proxy에 대화 완성 요청을 보낸다(기본 별칭, 공유 키)
3	LiteLLM Proxy	별칭을 온프레미스 vLLM으로 해석하고 샘플링 파라미터와 추론 끄 설정을 넣는다. 실패하면 재시도, 쿨다운, 폴백 순으로 처리한다
4	에이전트 컨테이너	모델이 요청한 도구(MCP 브리지, 검색, 웹 추출)를 실행하고 결과마다 프록시를 다시 호출한다
5	게이트웨이 프로세스	최종 답을 스트리밍으로 메신저에 돌려보낸다
6	프로파일의 스케줄러	예약 작업을 기동한다. LLM을 쓰는 작업은 2단계부터 같은 경로를 따르고, 스크립트 전용 작업은 프록시를 호출하지 않는다
7	에이전트 콘솔, 대시보드 API	콘솔에서 보낸 대화와 변경은 대시보드 API를 거치며, LLM 호출은 에이전트 컨테이너에서 나간다

2단계에서 에이전트는 모델의 실제 이름이나 추론 서버의 주소를 알지 못합니다. 게이트웨이 프로세스가 아는 것은 기본 대화 모델의 별칭과 LiteLLM Proxy 주소이며, 요청 형식은 OpenAI 호환 방식입니다. LiteLLM

Proxy가 플랫폼 팀이 자체 구축해 조직 전체의 LLM 접근을 관리하는 게이트웨이로 설계됐으므로[S02], 에이전트 쪽에서는 OpenAI 호환 클라이언트 설정만으로 연결이 끝납니다.

3단계에서 LiteLLM Proxy는 별칭을 온프레미스 GPU 장비의 vLLM으로 해석하고, 설정 파일에 정의된 샘플링 파라미터와 추론 과정 출력을 끄는 설정을 요청에 넣습니다. 요청이 실패하면 프록시는 한 번 재시도하고, 그 배포를 300초 동안 쿨다운 상태로 둔 뒤 2차 게이트웨이의 다른 모델로 요청을 전환합니다. 재시도 횟수, 쿨다운으로 전환하는 실패 허용 횟수, 쿨다운 시간은 LiteLLM의 안정성 설정 항목이고 폴백은 별칭에서 별칭으로 가는 목록으로 정의합니다[S06]. 이 처리는 모두 프록시 안에서 일어나므로 에이전트의 설정과 호출 방식은 달라지지 않습니다.

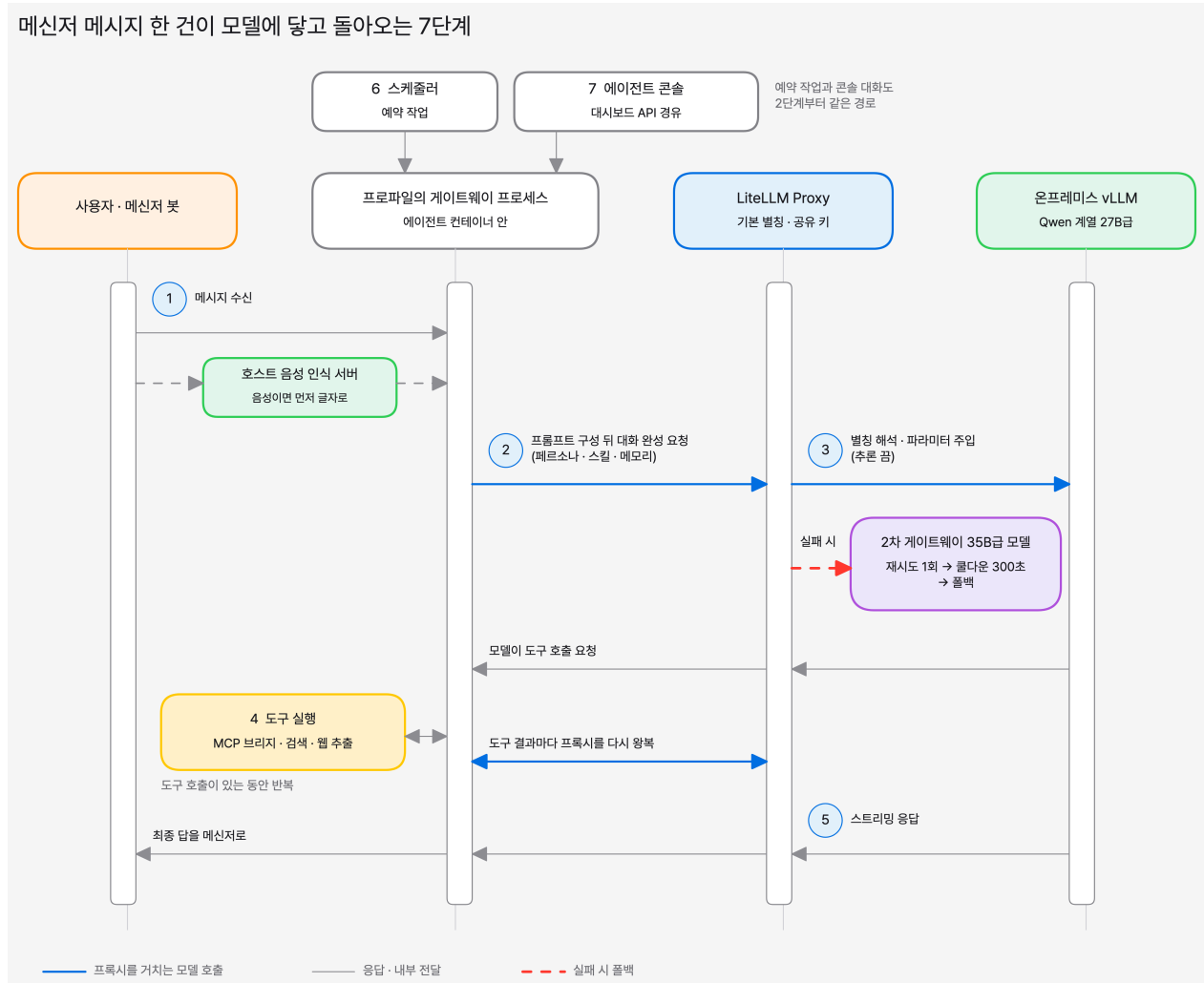


그림. 메신저 메시지가 게이트웨이 프로세스와 LiteLLM Proxy를 거쳐 추론 서버에 도달하고 스트리밍으로 돌아오는 7단계

5단계의 응답은 스트리밍으로 전달되므로 전체 응답 시간과 함께 첫 토큰 시간(TTFT, 요청을 보낸 뒤 첫 응답 토큰이 도착하기까지의 시간)을 봐야 합니다. OpenTelemetry GenAI 규격도 모델 서버의 첫 토큰 시간 지표가 스트리밍 요청에서 특히 중요하다고 설명합니다[S09]. 요청은 게이트웨이 프로세스의 프롬프트 구성, 프록시의 별칭 해석, 추론 서버의 대기과 생성 단계를 차례로 지나므로 지연도 이 단계별로 나눠 봐야 원인을 찾을 수 있습니다. 다만 현재 구성에서는 프록시의 지표 수집이 켜져 있지 않아 단계별 지연과 첫 토큰 시간을 볼 수 없습니다.

7.5.2 도구 호출마다 추가되는 프록시 왕복과 예약 작업·콘솔 경로

4단계에서 대화 한 번은 프록시 요청 여러 건으로 나뉩니다. 모델이 응답으로 도구 호출을 요청하면 에이전트는 컨테이너 안에서 그 도구를 실행하고, 실행 결과를 붙여 LiteLLM Proxy에 다시 요청을 보냅니다. 도구는 위키, CRM, 법령, 공문서, APM에 연결되는 MCP 브리지와 SearXNG 검색, Firecrawl 웹 추출이며, 도구 결과 하나마다 프록시 왕복이 한 번씩 추가됩니다. 예를 들어 어떤 대화에서 모델이 도구를 두 번 호출한 뒤 답을 만든다면, 사용자에게는 대화 한 번이지만 프록시에는 첫 요청과 도구 결과를 붙인 요청 두 건을 더해 세 건이 도착합니다.

그래서 대화 한 번의 비용과 지연은 프록시 요청 여러 건의 합입니다. 프록시가 기록하는 단위는 요청이므로, 요청별 지연이 짧더라도 도구 호출이 여러 번 이어진 대화는 사용자에게 길게 느껴질 수 있습니다. LiteLLM Proxy의 타임아웃을 요청과 스트림 모두 30분으로 길게 둔 것도 에이전트의 긴 추론과 반복되는 도구 호출을 고려한 설정입니다. 요청 단위 기록을 대화 단위로 묶어 보려면 에이전트, 프록시, 모델을 잇는 트레이스가 필요하며, 이 내용은 모니터링의 관측 대상에 포함됩니다.

도구 호출 자체는 LiteLLM Proxy를 지나지 않는다는 점도 구분해야 합니다. 이 구성에서 MCP 브리지, 검색, 웹 추출은 에이전트 컨테이너가 직접 실행하며, 프록시에 도착하는 것은 도구 결과를 붙인 다음 요청뿐입니다. 모델 호출은 LiteLLM Proxy를 거치고 도구 호출은 MCP와 도구 네트워크를 거치므로, 두 호출은 서로 다른 길목을 지납니다. 프록시 쪽에서 요청 수와 토큰은 셀 수 있어도 어떤 도구가 어느 시스템에 무엇을 요청했는지는 알 수 없습니다. 검색과 웹 추출의 사용 기록은 각 도구 쪽에서 따로 확인해야 합니다.

이 구분에 따라 LLM Gateway의 통제 범위는 모델 접근까지로 보아야 합니다. 에이전트가 위키나 CRM에 어떤 권한으로 접근하는지는 프로파일마다 연결한 MCP 서버의 구성에서 정해지며, 이 내용은 7.2.2에서 설명한 프로파일별 MCP 연결에 해당합니다. 인프라 점검 에이전트를 읽기 전용으로 둔 것처럼 도구 쪽 권한은 에이전트 구성에서 따로 정해야 합니다. 모델로 가는 길목을 하나로 모으는 일과 도구로 가는 권한을 나누는 일은 별개의 작업이며, 한쪽을 마쳤다고 다른 쪽이 해결되지는 않습니다.

예약 작업과 콘솔 경로도 같은 길목을 지납니다. 프로파일의 스케줄러가 예약 작업을 기동하면 LLM을 쓰는 작업은 메신저 대화와 똑같이 프롬프트를 만들어 프록시에 요청을 보내고, 도구 호출이 있으면 같은 방식으로 왕복합니다. 스크립트 전용 작업은 모델을 호출하지 않으므로 프록시에 아무 기록도 남기지 않습니다. 콘솔에서 보낸 대화는 대시보드 API를 거쳐 에이전트에 전달된 뒤 2단계부터 같은 순서로 처리됩니다. 결과적으로 메신저, 스케줄러, 콘솔 세 가지 시작 경로가 모두 에이전트 컨테이너에서 LiteLLM Proxy로 가는 한 경로로 모입니다.

7.5.3 LiteLLM Proxy 를 거치지 않는 프로바이더 직접 호출 경로 3건

오픈마루의 구성에는 LiteLLM Proxy를 거치지 않고 외부 API를 직접 호출하는 경로가 3건 있습니다. 세 경로는 스크립트 두 개와 이미지 생성 백엔드 하나이며, 모두 클라우드 프로바이더 API나 모델 중개 서비스를 직접 호출합니다. 이 사례를 드는 이유는 에이전트를 프록시에 연결한 뒤에도 프록시 밖 호출이 남을 수 있다는 점을 보여 주기 때문입니다.

경로	하는 일	호출 대상
AI 검색 인용 주간 점검 스크립트	AI 검색 서비스의 인용 현황을 매주 점검	클라우드 프로바이더 세 곳의 API
위키 정리 스크립트	위키 문서 정리	클라우드 프로바이더 API

경로	하는 일	호출 대상
이미지 생성 백엔드 하나	이미지 생성	모델 중개 서비스

이 세 경로의 호출은 프록시의 통제와 집계에서 빠집니다. 프록시의 모델 별칭, 폴백, 타임아웃 설정이 적용되지 않고, 이 호출들의 호출량은 프록시의 기록에 나타나지 않습니다. 이 단계의 다음 단계는 세 경로를 프록시 경유로 옮기고 클라우드 키를 프록시로 모으는 것이며, 그 순서는 9장에서 다룹니다. 에이전트 프레임워크를 프록시에 연결하는 것만으로는 모델 길목이 하나가 되지 않으며, 에이전트 밖의 스크립트와 서비스를 따로 조사해야 하는 이유가 여기에 있습니다.

음성 인식은 이 세 경로와 성격이 다릅니다. 음성 메시지는 호스트의 로컬 음성 인식 서버가 글자로 바꾸며, 이 처리는 LLM 호출이 아니므로 LiteLLM Proxy의 대상에 들지 않습니다. 반면 에이전트 컨테이너 안의 보조 스크립트(PDF 읽기, 명함 인식, 이력서 평가, 콘텐츠 요약, 영상 첫 댓글)는 에이전트 루프 밖에서 실행되지만 프록시를 직접 호출하므로 길목 안에 있습니다. 따라서 프록시 밖 경로를 프록시 경유로 옮기고 클라우드 키를 프록시로 모으는 작업의 대상은 위 표의 3건으로 한정됩니다.

8장: LLM Gateway 모니터링: 관측 대상 다섯 묶음과 현재 구성의 공백

LLM Gateway는 조직의 모델 호출이 모두 지나가는 한 곳이므로, 호출량과 지연, 오류, 백엔드 상태를 한 번에 볼 수 있는 유일한 위치입니다. LiteLLM Proxy는 이 정보를 Prometheus(시계열 지표를 주기적으로 수집해 저장하는 오픈소스 모니터링 시스템) 형식의 지표로 내보냅니다. 지표는 비용·토큰, 지연, 요청 수, 배포 상태, 폴백·쿨다운을 비롯한 여러 묶음으로 나뉘며, 이 장은 그 가운데 운영 판단에 바로 쓰이는 다섯 묶음을 다룹니다 [S05].

게이트웨이 모니터링이 일반 API 서버 모니터링과 달라지는 이유는 AI 에이전트의 호출 방식에 있습니다. 에이전트는 도구를 한 번 쓸 때마다 모델을 다시 부르기 때문에 사용자의 대화 한 번이 프록시 요청 여러 건으로 늘어납니다. 폴백이 일어나면 호출한 쪽은 오류 없이 다른 모델의 답을 받으므로, 기본 모델이 응답하지 않았다는 사실이 호출한 쪽에는 드러나지 않습니다. 따라서 지표가 없으면 응답이 느려진 원인이 어디에 있는지, 지금 어느 모델이 답하고 있는지, 누가 얼마나 쓰는지 게이트웨이를 두고도 알 수 없습니다.

지표 이름을 맞추는 규격으로는 OpenTelemetry GenAI 시맨틱 컨벤션(semantic conventions, 지표와 속성의 이름을 정한 규약)이 있습니다. 이 규격의 지표 문서와 토큰 지표 문서는 모두 Development 단계이며, 이는 지표 이름이 앞으로 더 바뀔 수 있다는 뜻입니다[S09]. LiteLLM Proxy의 지표 이름과 규격의 지표 이름을 대응시킬 때는 두 체계가 같은 값을 재는지, 이름이 고정된 것인지를 함께 확인해야 합니다.

오픈마루의 운영 구성을 사례로 드는 이유는, 호출 경로를 게이트웨이로 모은 뒤에도 관측 설정을 켜지 않으면 무엇을 볼 수 없는지를 구체적으로 보여 주기 때문입니다. 이 구성에서는 Prometheus 지표 엔드포인트가 꺼져 있고 성공 콜백도 없어, 남는 기록은 컨테이너 표준 출력 로그뿐입니다. 이 공백을 메우는 작업은 선행 조건에 따라 둘로 나뉘는데, 지연과 폴백 지표는 데이터베이스 없이 바로 얻고, 카·팀·사용자별 지표는 가상 키가 먼저 있어야 값이 나뉘므로, 이 부분이 지표 수집과 키 분리의 순서를 정합니다[S03][S05].

8.1 게이트웨이 관측 대상 다섯 묶음과 판단 용도: 비용·토큰, 지연, 요청 수, 배포 상태, 폴백·쿨다운

LiteLLM Proxy의 지표를 운영 판단에 따라 나누면 다섯 묶음이 되고, 묶음마다 답하는 질문이 다릅니다. 비용·토큰 지표는 누가 얼마나 쓰는지를 알려 주고, 요청 수 지표는 요청이 얼마나 도착하고 그중 얼마나 실패하는지를 알려 줍니다. 지연 지표는 응답이 느려진 원인이 프록시에 있는지 모델 서버에 있는지를 구분하는 데 쓰입니다. 배포 상태 지표와 폴백·쿨다운 지표는 백엔드가 정상인지, 다른 모델이 대신 답하고 있는지를 알려 줍니다 [S05].

에이전트 워크로드에서는 요청 단위의 값과 대화 단위의 값을 구분해서 읽어야 합니다. 오픈마루 구성에서 모델이 도구 호출을 요청하면 에이전트는 도구를 실행한 뒤 그 결과를 들고 프록시를 한 번 더 왕복합니다. 그 결과 프록시 지표의 요청 수는 대화 수보다 항상 크거나 같고, 요청 한 건의 지연은 사용자가 기다린 전체 시간의 일부에 해당합니다. 프록시 지표만으로 용량과 오류는 판단할 수 있지만, 대화 한 번이 끝나기까지 걸린 시간은 에이전트 쪽에서 따로 재야 합니다.

LiteLLM Proxy의 지표 목록에는 다섯 묶음 말고도 예산, 가드레일, MCP 도구 호출, 연결 서비스 상태 묶음이 있습니다[S05]. 이 가운데 예산 지표는 가상 키가 전제이므로 8.4절에서 선행 조건과 함께 다룹니다. 나머지 묶

음은 해당 기능을 프록시에서 쓰는 조직이 추가로 수집하면 되는 지표이며, 게이트웨이를 처음 관측하는 단계에서는 다섯 묶음으로 충분합니다.

8.1.1 비용·토큰 지표와 요청 수 지표로 보는 소비자별 사용량과 오류

비용·토큰 묶음과 요청 수 묶음은 사용량을 세는 지표이며, 비용·토큰 묶음에는 지출 지표 한 개와 토큰 지표 다섯 개가 있고, 요청 수 묶음에는 전체 요청, 실패 요청, 처리 중인 요청을 세는 지표가 있습니다[S05]. 다음 표는 두 묶음의 지표 이름과 각 지표로 내리는 운영 판단을 정리한 것입니다.

구분	지표 이름	운영 판단
지출	litellm_spend_metric	과금되는 경로의 지출 추세와 예산 편성
토큰 합계와 입력·출력	litellm_total_tokens_metric , litellm_input_tokens_metric , litellm_output_tokens_metric	추론 서버의 용량 산정, 입력과 출력의 비중
캐시 입력, 추론 출력	litellm_input_cached_tokens_metric , litellm_output_reasoning_tokens_metric	캐시로 처리한 입력의 비중, 추론 과정에 쓴 출력의 비중
전체 요청과 실패 요청	litellm_proxy_total_requests_metric , litellm_proxy_failed_requests_metric	실패 비율과 요청 증가 추세
처리 중인 요청	litellm_in_flight_requests	같은 시각에 겹치는 요청의 수

온프레미스 모델을 주로 쓰는 구성에서는 지출보다 토큰이 용량을 판단하는 단위가 됩니다. 오픈마루 구성에서 클라우드 경로는 Azure OpenAI 별칭 하나뿐이고, 나머지 별칭은 온프레미스 GPU 장비의 vLLM, 쿠버네티스의 2차 게이트웨이, 사내 다른 망의 llama.cpp 서버, 지사의 추론 서버로 향합니다. 따라서 지출 지표는 클라우드 별칭의 사용 추세를 보는 데 쓰고, 토큰 지표는 추론 서버가 감당하는 처리량을 보는 데 씁니다. 이 구성은 같은 모델에 추론을 컨 별칭과 끈 별칭을 따로 두고 있으므로, 추론 출력 토큰 지표는 추론을 컨 별칭이 추론 과정에 출력 토큰을 얼마나 쓰는지 살펴볼 때 쓸 수 있습니다.

온프레미스 경로에서 사용량 증가는 청구 금액 대신 처리 시간으로 나타납니다. 과금되는 경로에서는 출력 토큰이 늘면 지출 지표가 오르지만, 조직이 장비를 보유한 경로에서는 같은 증가가 대기열 시간과 처리 중인 요청 수의 증가로 나타납니다. 온프레미스 별칭의 토큰 지표는 지출 지표보다 지연 묶음의 지표와 함께 읽어야 합니다. 예를 들어 토큰 합계와 대기열 시간이 함께 늘었다면 추론 서버의 처리 용량이 한계에 가까워졌다고 볼 수 있습니다.

요청 수 지표는 오류와 부하를 가장 먼저 알려 주는 지표로, 실패 요청 수를 전체 요청 수로 나눈 비율이 오르면 백엔드 장애나 설정 오류를 의심할 수 있습니다. 에이전트의 예약 작업은 사람의 대화와 무관하게 정해진 시각에 모델을 호출하기 때문에, 요청 수의 시간대별 추세를 봐야 예약 작업이 물리는 시간과 대화 요청이 물리는 시간을 구분할 수 있습니다. 처리 중인 요청 수는 별칭마다 배포가 하나뿐인 구성에서 특히 유용한데, 동시에 겹치는 요청이 늘어나는 시점이 배포를 추가할지 판단하는 근거가 되기 때문입니다.

이 지표들을 소비자별로 나눠 보려면 가상 키가 있어야 하는데, 키·팀·사용자별 지출과 예산 지표는 가상 키가 있어야 값이 나뉘고, 가상 키를 쓰려면 Postgres 데이터베이스가 필요합니다[S03][S05]. 모든 소비자가 마스터 키 하나를 함께 쓰는 구성에서는 지표를 켜도 전체 합계만 나오므로, 누가 얼마나 쓰는지에 대한 답은 키 분리 뒤로 미뤄집니다.

8.1.2 지연 지표와 첫 토큰 시간으로 보는 에이전트 응답 품질

지연 묶음에는 지표가 다섯 개 있고, 각 지표는 요청이 처리되는 과정의 서로 다른 부분을 쬡니다[S05]. 지표를 하나만 보면 응답이 느려졌다는 사실은 알 수 있어도 조치할 대상을 정할 수 없습니다. 다음 표처럼 다섯 지표를 함께 놓고, 어느 값이 커졌는지에 따라 살펴볼 대상을 정합니다.

지표 이름	재는 값	값이 커졌을 때 살펴볼 대상
<code>litellm_request_total_latency_metric</code>	요청 전체 지연	아래 네 지표로 원인을 구분
<code>litellm_llm_api_latency_metric</code>	모델 API 호출 지연	추론 서버와 그 사이의 네트워크 경로
<code>litellm_llm_api_time_to_first_token_metric</code>	첫 토큰 시간	추론 서버의 대기 상태와 입력 처리
<code>litellm_overhead_latency_metric</code>	프록시 자체 지연	프록시 컨테이너의 자원
<code>litellm_request_queue_time_seconds</code>	대기열에서 기다린 시간	동시에 겹친 요청의 수

첫 토큰 시간(TTFT, Time To First Token)은 요청을 보낸 뒤 응답의 첫 토큰이 도착할 때까지 걸린 시간입니다. 스트리밍으로 답하는 에이전트에서는 이 값이 사용자가 답이 시작되기까지 기다린 시간에 가깝습니다. 오픈마루 구성에서도 에이전트의 최종 답은 스트리밍으로 메신저에 전달되므로, 답 전체가 완성되는 시각과 별도로 첫 글자가 나타나는 시각을 따로 봐야 합니다. 반면 전체 지연은 답의 길이에 따라 달라지기 때문에, 긴 보고서를 쓰는 요청과 짧은 질의응답을 같은 값으로 비교하기 어렵습니다.

지연 지표를 나눠 보는 목적은 조치 대상을 정하는 데 있으며, 모델 API 호출 지연이 전체 지연의 대부분을 차지하면 추론 서버나 그 경로를 살펴야 하고, 프록시 자체 지연이나 대기열 시간이 커지면 프록시 컨테이너를 살펴야 합니다. 예를 들어 오픈마루의 LiteLLM Proxy는 메모리 2GB와 CPU 1.5 코어로 제한한 컨테이너 한 개이므로, 프록시 자체 지연이 늘어나면 이 제한값을 먼저 검토하게 됩니다. 배포 상태 묶음의 `litellm_deployment_latency_per_output_token` 은 출력 토큰 한 개당 걸린 시간을 재므로, 답의 길이와 무관하게 추론 서버의 생성 속도를 비교하는 데 쓸 수 있습니다[S05].

에이전트용 게이트웨이에서는 타임아웃이 길다는 점도 지연 지표가 필요한 이유입니다. 오픈마루 구성은 에이전트의 긴 추론과 도구 호출 반복을 고려해 요청과 스트림의 타임아웃을 모두 30분으로 두었습니다. 타임아웃이 이만큼 길면 응답이 수 분씩 늦어져도 타임아웃 오류로는 드러나지 않을 수 있고, 그때는 실패 요청 수만 보서는 느려진 상태를 알기 어렵습니다. 따라서 타임아웃을 길게 둔 게이트웨이일수록 오류 수와 별도로 지연과 첫 토큰 시간의 추세를 봐야 합니다.

8.1.3 배포 상태와 풀백·쿨다운 지표로 보는 백엔드 장애와 우회 발생

배포(deployment)는 모델 별칭 뒤에 연결된 실제 백엔드 하나를 가리키며, 배포 상태 묶음은 이 백엔드가 정상적으로 응답하는지를 알려 줍니다. 이 묶음에는 배포 상태를 나타내는 `litellm_deployment_state`, 배포별 성공 응답과 실패 응답을 세는 `litellm_deployment_success_responses` 와 `litellm_deployment_failure_responses` 가 있습니다. 폴백·쿨다운 묶음에는 쿨다운 발생을 세는 `litellm_deployment_cooled_down`, 폴백의 성공과 실패를 세는 `litellm_deployment_successful_fallbacks` 와 `litellm_deployment_failed_fallbacks` 가 있습니다[S05].

쿨다운은 실패가 이어진 배포를 일정 시간 동안 요청 대상에서 제외하는 동작입니다. LiteLLM에서는 1분 안에 `allowed_fails` 횟수를 넘겨 실패한 배포를 `cooldown_time` 에 적은 시간(초) 동안 제외합니다[S06]. 오픈마루 구성은 재시도 1회, 실패 허용 1회, 쿨다운 300초로 설정돼 있고, 폴백 3건은 모두 27B급 모델로 가는 별칭에서 쿠버네티스 2차 게이트웨이 뒤의 35B급 모델 별칭으로 향합니다. 온프레미스 GPU 장비의 vLLM이 응답하지 않으면 이후 300초 동안 기본 별칭으로 들어온 요청은 2차 게이트웨이의 다른 모델이 처리합니다.

폴백은 호출한 쪽에서 보이지 않게 일어나는데, 에이전트는 기본 별칭으로 요청했고 정상 응답을 받았으므로 오류를 기록하지 않습니다. 하지만 실제로 답한 모델은 다른 모델이고 경로도 2차 게이트웨이를 한 번 더 거치므로, 답의 품질과 속도는 달라질 수 있습니다. 따라서 폴백 성공 지표가 늘어나는 것은 서비스가 유지되고 있다는 뜻인 동시에 기본 백엔드를 점검해야 한다는 신호이며, 지표가 없으면 기본 모델이 쿨다운 상태라는 사실을 알기 어렵습니다.

이 묶음의 지표가 알려 주는 것은 백엔드가 응답했는지까지이며, 응답 내용이 맞는지 알려 주지 않습니다. 배포 상태가 정상이고 실패 응답이 없더라도 답의 내용이 기대와 다를 수 있고, 그런 상태는 가용성 지표와 지연 지표 어디에도 나타나지 않습니다. 폴백으로 다른 모델이 답한 시간대가 대표적인 경우여서, 요청은 모두 성공으로 집계되지만 답을 만든 모델은 기본 모델이 아닙니다. 따라서 게이트웨이 지표는 응답 품질을 점검하는 절차를 대신하지 못하며, 폴백이 일어난 시간대의 답을 따로 살펴볼지 판단하는 단서로 쓰는 것이 알맞습니다.

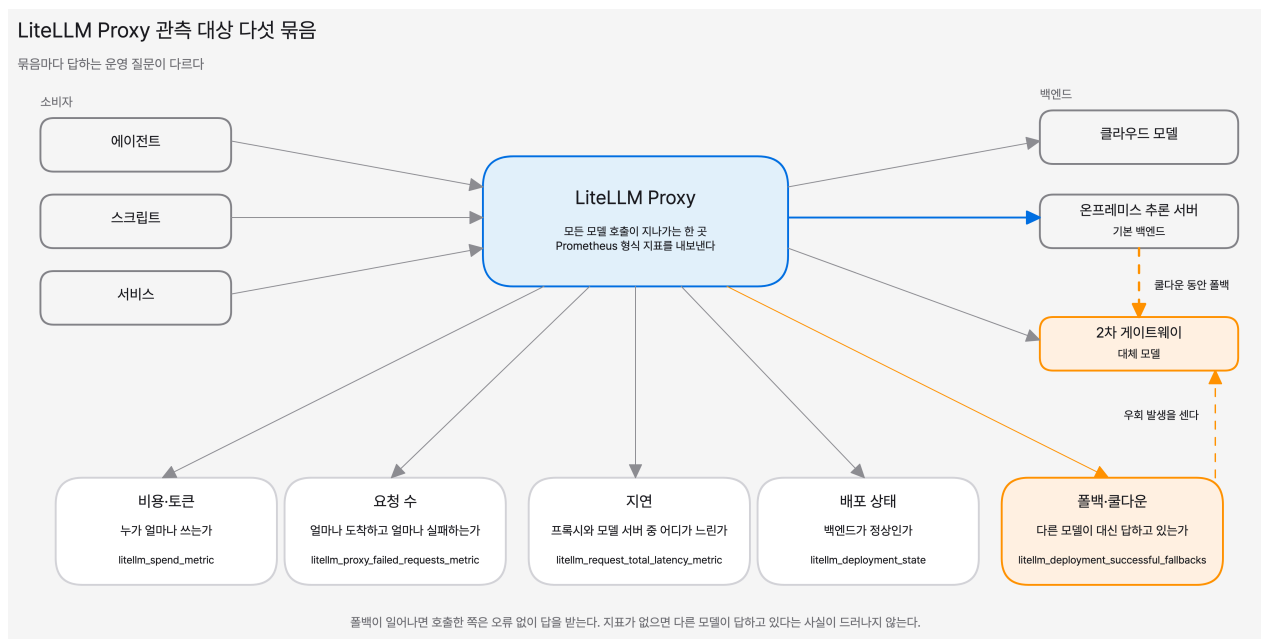


그림. LiteLLM Proxy 관측 대상 다섯 묶음과 묶음별 운영 판단

두 묶음의 지표 가운데 경보를 가장 먼저 걸 만한 대상은 폴백 실패인데, 폴백 실패 지표가 늘었다면 기본 백엔드에 이어 대체 백엔드도 답하지 못해 호출한 쪽에 오류가 전달됐을 수 있기 때문입니다. 폴백이 프록시 한 곳에서만 일어나지 않는다는 점도 함께 고려해야 하는데, 오픈마루의 콘텐츠 공장 스택은 에이전트 프레임워크 수준의

폴백도 갖고 있어, 기본 모델 호출이 실패하면 에이전트가 직접 35B 별칭으로 다시 요청합니다. 이 요청은 프록시에 35B 별칭을 지정한 요청으로 도착하므로 프록시의 폴백 지표에는 집계되지 않을 수 있습니다.

8.2 LiteLLM Prometheus 지표와 OpenTelemetry GenAI 규격의 대응, Development 단계 규격의 변경 가능성

LiteLLM Proxy의 Prometheus 지표 이름은 LiteLLM이 정한 이름이고, 다른 게이트웨이나 추론 서버의 지표 이름과 같지 않습니다. 이름을 맞추려는 규격이 OpenTelemetry(OTel, 지표·로그·트레이스를 수집하는 오픈소스 관측 프레임워크) GenAI 시맨틱 컨벤션입니다. 이 규격은 전용 저장소에서 관리되며, 클라이언트 지표, 모델 서버 지표, 토큰 지표, 에이전트·도구 지표의 이름을 정의합니다[S09].

OpenTelemetry GenAI 규격의 지표 문서와 토큰 지표 문서는 모두 Development 단계입니다[S09]. 전용 저장소가 만들어진 시기도 2026년 5월로 오래되지 않았습니다[S11]. 이 규격의 이름으로 대시보드와 경보 규칙을 설계할 수는 있지만, 이름이 바뀌면 고쳐야 할 위치를 미리 파악해 두어야 합니다. 예를 들어 계측 라이브러리의 버전을 올릴 때는 지표 이름이 달라졌는지를 먼저 확인하고, 대시보드의 질의문을 그 뒤에 고치는 순서를 운영 절차에 넣어 둡니다.

두 체계의 대응을 알아야 하는 실무상의 이유는 중복 집계입니다. 에이전트 프레임워크를 OpenTelemetry로 계측하고 프록시에서 Prometheus 지표를 수집하면, 같은 모델 호출의 토큰이 에이전트 쪽과 프록시 쪽에서 한 번씩 집계될 수 있습니다. 그런 구성에서 두 값을 더하면 사용량이 실제보다 크게 보일 수 있으므로, 어느 쪽 값을 비용 귀속의 근거로 쓸지 먼저 정해야 합니다.

8.2.1 LiteLLM 지표 이름과 OpenTelemetry GenAI 지표 이름의 묶음별 대응표

다음 표는 LiteLLM Proxy의 Prometheus 지표와 OpenTelemetry GenAI 규격의 지표를 묶음별로 나란히 놓은 것입니다[S05][S09]. 같은 줄에 있다고 해서 두 지표의 값이 같다는 뜻은 아니며, 같은 종류의 값을 서로 다른 위치에서 잴다는 뜻입니다. 표에서 줄표로 둔 칸은 이 장이 다루는 지표 목록 안에 대응시킬 이름이 없는 경우입니다.

묶음	LiteLLM Prometheus 지표	OpenTelemetry GenAI 지표
입력·출력 토큰	litellm_input_tokens_metric , litellm_output_tokens_metric	gen_ai.client.inference.usage.input_tokens , gen_ai.client.inference.usage.output_tokens
캐시·추론 토큰	litellm_input_cached_tokens_metric , litellm_output_reasoning_tokens_metric	캐시 읽기, 캐시 쓰기, 추론 토큰 지표
요청당 토큰 분포	—	gen_ai.client.inference.operation.input_tokens , gen_ai.client.inference.operation.output_tokens
지출	litellm_spend_metric	—
호출 시간	litellm_llm_api_latency_metric	gen_ai.client.operation.duration , gen_ai.server.request.duration

목록	LiteLLM Prometheus 지표	OpenTelemetry GenAI 지표
첫 토큰 시간	<code>litellm_llm_api_time_to_first_token_metric</code>	<code>gen_ai.client.operation.time_to_first_chunk</code> , <code>gen_ai.server.time_to_first_token</code>
출력 단위당 시간	<code>litellm_deployment_latency_per_output_token</code>	<code>gen_ai.client.operation.time_per_output_chunk</code> , <code>gen_ai.server.time_per_output_token</code>
요청 수	<code>litellm_proxy_total_requests_metric</code> , <code>litellm_proxy_failed_requests_metric</code>	—
에이전트·도구	—	<code>gen_ai.invoke_agent.duration</code> , <code>gen_ai.invoke_agent.inference_calls</code> , <code>gen_ai.invoke_agent.tool_calls</code> , <code>gen_ai.execute_tool.duration</code>

토큰 목록은 두 체계가 가장 가깝게 대응하는 부분입니다. 양쪽 모두 입력 토큰과 출력 토큰을 따로 세고, 캐시로 처리한 토큰과 추론에 쓴 토큰을 구분합니다. 반면 지출, 배포 상태, 폴백·쿨다운은 게이트웨이가 하는 일에서 나오는 값이므로 LiteLLM 쪽에만 이름이 있습니다. 규격의 이름으로 대시보드를 통일하더라도 폴백과 쿨다운을 보는 화면은 LiteLLM 지표 이름으로 따로 구성해야 합니다.

에이전트·도구 지표는 규격 쪽에만 있습니다. 에이전트 호출 한 번에 걸린 시간, 그 안에서 일어난 추론 호출 수, 도구 호출 수, 도구 실행 시간이 여기에 해당합니다[S09]. 이 지표들은 8.1절에서 구분한 대화 단위의 값을 채워 줍니다. 프록시 지표가 요청 한 건의 지연과 토큰을 알려 준다면, `gen_ai.invoke_agent.inference_calls` 는 대화 한 번이 프록시 요청 몇 건으로 늘어났는지를 에이전트 쪽에서 알려 줍니다. LiteLLM 지표 목록에도 `litellm_mcp_tool_calls_total` 이 있지만[S05], 오픈마루 구성에서는 에이전트가 MCP 도구를 에이전트 컨테이너 안에서 실행하므로 도구 호출 자체는 프록시를 지나지 않습니다.

규격은 지표에 붙이는 속성으로 `gen_ai.request.model` 과 `gen_ai.response.model` 을 따로 둥니다[S09]. 별칭으로 요청하는 게이트웨이 구성에서는 요청할 때 쓴 이름과 실제로 답한 모델의 이름이 다를 수 있으므로, 두 속성을 구분해 기록하면 폴백이 일어난 요청을 호출한 쪽에서도 가려낼 수 있습니다. 다만 이 속성 이름도 Development 단계 문서에 속하므로 지표 이름과 같은 방식으로 변경에 대비해야 합니다.

8.2.2 첫 토큰 시간 지표 세 개가 재는 구간의 차이

첫 토큰 시간에 해당하는 지표는 두 체계를 합쳐 세 개로, LiteLLM의 `litellm_llm_api_time_to_first_token_metric`, 규격의 클라이언트 지표인 `gen_ai.client.operation.time_to_first_chunk`, 규격의 모델 서버 지표인 `gen_ai.server.time_to_first_token` 입니다[S05][S09]. 세 지표는 이름이 비슷하지만 재는 위치가 서로 다르기 때문에, 같은 요청에 대해서도 값이 다르게 나옵니다.

세 지표의 차이는 시계를 누가 들고 있느냐에서 나옵니다. LiteLLM 지표는 이름대로 프록시가 모델 API를 호출한 쪽에서 재는 첫 토큰 시간이므로, 프록시와 추론 서버 사이의 네트워크와 추론 서버의 처리 시간이 값에 들어갈 수 있습니다. 규격의 클라이언트 지표는 호출하는 쪽이 첫 청크를 받을 때까지의 시간이며, 에이전트가 이 지표를 내면 프록시가 요청을 처리한 시간까지 포함됩니다. 규격의 모델 서버 지표는 추론 서버가 스스로 재는 값

으로, OpenTelemetry GenAI 규격은 이 지표를 대기열과 프리필(prefill, 입력 프롬프트를 처리하는 단계)에 쓴 시간을 재는 지표이며 스트리밍 요청에서 특히 중요하다고 설명합니다[S09].

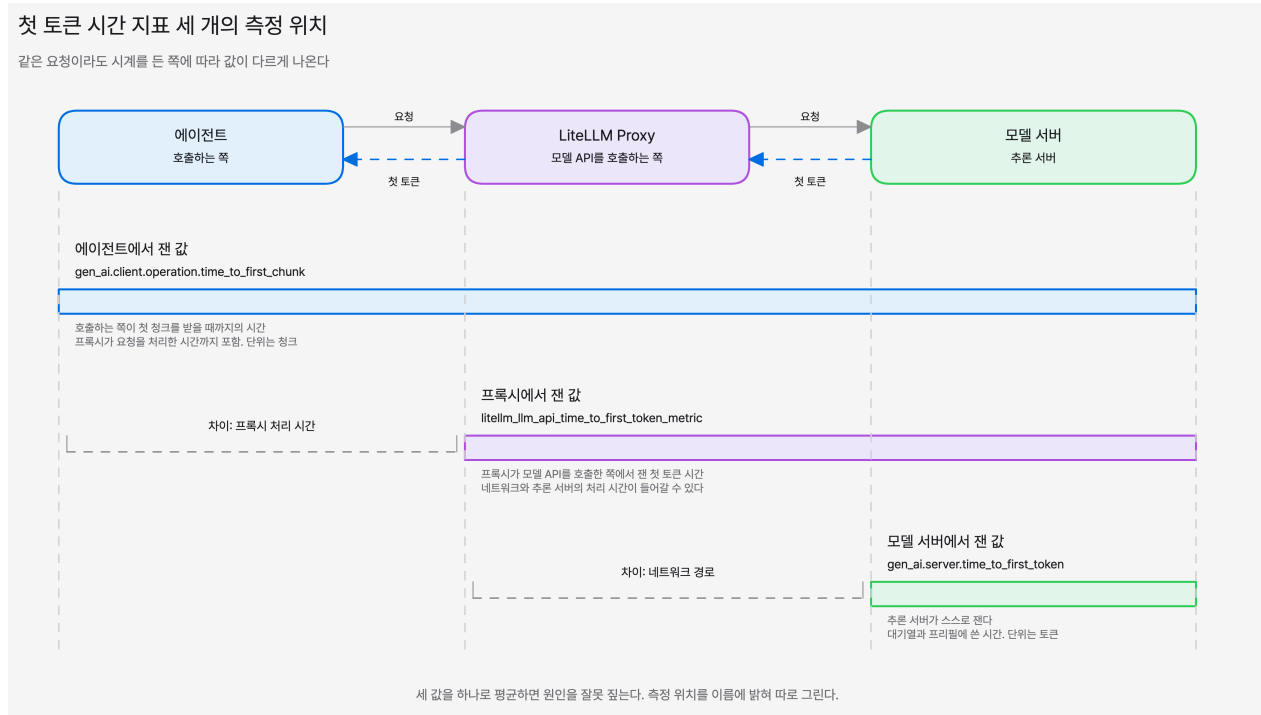


그림. 첫 토큰 시간 지표 세 개의 측정 위치와 포함 범위

세 값을 함께 보면 지연이 생긴 위치를 좁힐 수 있습니다. 모델 서버의 값은 그대로인데 프록시에서 잔 값이 늘었다면 프록시와 추론 서버 사이의 경로를 살펴야 합니다. 반면 프록시에서 잔 값은 그대로인데 에이전트에서 잔 값만 늘었다면 프록시 자체 지연이나 대기열 시간을 확인해야 합니다. 세 지표를 같은 값으로 묶어 하나의 그래프에 평균하면 원인을 잘못 짚을 수 있으며, 대시보드에서는 측정 위치를 이름에 밝혀 따로 그려야 합니다.

단위의 차이도 일대일 대응을 막습니다. 규격의 클라이언트 지표는 청크를 단위로 하고 모델 서버 지표는 토큰을 단위로 하는데, 스트리밍 응답의 청크 하나에는 토큰이 여러 개 실릴 수 있습니다. 폴백이 일어난 요청에서는 측정 범위가 더 벌어지는데, 오픈마루 구성에서 폴백 경로는 2차 게이트웨이를 거치므로 프록시에서 잔 첫 토큰 시간에 2차 게이트웨이의 처리 시간이 더해질 수 있고, 에이전트에서 잔 값에는 실패한 시도와 재시도에 쓴 시간까지 더해질 수 있습니다.

8.3 현재 구성의 관측 공백: 꺼진 지표 엔드포인트, 콜백 부재, 모델 별칭이 빠진 컨테이너 로그

오픈마루의 LiteLLM Proxy 구성은 호출 경로를 한 곳으로 모으는 일까지는 마쳤지만 그 한 곳을 관측하는 설정은 꺼지 않은 상태입니다. 이 구성을 사례로 드는 이유는 경로만 모으고 관측을 꺼지 않았을 때 무엇이 비는지를 항목별로 보여 주기 때문입니다. 이 구성은 LiteLLM 1.81 계열의 단일 컨테이너이고 데이터베이스와 캐시를 붙이지 않았습니다. 성공 콜백이 없고 Prometheus 지표 엔드포인트가 꺼져 있으며, OpenTelemetry와 외부 관측 도구 연동도 없습니다.

남는 기록은 컨테이너 표준 출력 로그뿐이고, 이 로그에는 요청별 모델 별칭이 남지 않습니다. 따라서 별칭 9개 가운데 어느 별칭으로 요청이 물리는지, 추론을 켜 별칭과 끈 별칭이 각각 얼마나 쓰이는지를 로그로는 나눌 수 없습니다.

폴백 3건이 설정돼 있지만 폴백이 일어나도 로그를 검색하는 것 말고는 확인할 방법이 없습니다. 관리 화면은 열리지만 데이터베이스가 없어 화면에서 관리하거나 조회할 대상이 없습니다. 8.1절의 다섯 묶음 가운데 현재 구성에서 볼 수 있는 것은 로그에서 센 요청 건수와 오류 건수이고, 다섯 묶음의 지표는 어느 것도 수집되지 않습니다.

8.3.1 컨테이너 표준 출력 로그로 볼 수 있는 것: 요청 건수와 오류 건수

현재 구성에서 볼 수 있는 값은 로그에서 센 요청 건수와 오류 건수입니다. 최근 사흘 동안의 로그에는 대화 완성 요청이 약 2,500건, 임베딩 요청이 약 40건, 모델 목록 조회가 약 280건 기록돼 있습니다. 이 수치로 호출의 대부분이 대화 완성이라는 규모감은 확인할 수 있습니다. 대화 완성 약 2,500건이 어느 별칭으로 향했는지는 로그에 남지 않습니다.

오류 건수도 로그에서 확인할 수 있으며, 같은 기간에 잘못된 모델 이름으로 인한 400 오류가 7건, 키 없이 도착한 요청이 11건 있었습니다. 이 두 수치는 로그만으로도 설정 문제가 드러난다는 점을 보여 줍니다. 별칭을 잘못된 호출 주체나 키 설정이 빠진 호출 주체가 있을 수 있다는 뜻이기 때문입니다. 다만 모든 소비자가 같은 마스터 키를 쓰고 같은 호스트 경유 주소로 들어오므로, 어느 소비자가 낸 오류인지를 프록시 쪽 기록만으로 가려내는 어렵습니다.

폴백과 쿨다운에 대해서는 최근 사흘 로그에 발생 기록이 없습니다. 이 사실 역시 로그를 검색해서 얻은 것이며, 폴백이 일어난 시각에 경보를 받는 구성은 아닙니다. 다음 표는 현재 구성에서 볼 수 있는 것과 볼 수 없는 것을 관측 항목별로 정리한 것입니다.

관측 항목	현재 구성	이유
요청 종류별 건수	로그에서 셀 수 있음	표준 출력 로그에 요청이 남음
상태 코드별 오류 건수	로그에서 셀 수 있음	표준 출력 로그에 오류가 남음
폴백·쿨다운 발생 여부	로그 검색으로만 확인	지표 엔드포인트 꺼짐
모델 별칭별 요청 수	볼 수 없음	로그에 요청별 별칭이 없음
지연과 첫 토큰 시간	볼 수 없음	지표 엔드포인트 꺼짐
토큰 사용량	볼 수 없음	지표 엔드포인트 꺼짐, 성공 콜백 없음
소비자별 사용량	볼 수 없음	마스터 키 하나를 공유

8.3.2 지연·첫 토큰 시간·폴백 발생·토큰 사용량·소비자별 사용량의 관측 부재

현재 구성은 지표와 트레이스를 수집하지 않으므로 지연, 첫 토큰 시간, 폴백 발생, 토큰 사용량을 볼 수 없습니다. 여기에 마스터 키 공유가 더해져 소비자별 사용량도 나눌 수 없습니다. 4장 요건 평가표의 '지연·첫 토큰 시간 관측' 항목이 현재 구성에서 공백으로 판정되는 근거가 이것입니다. 평가표는 지표 수집이 꺼져 있으면 이 항목을 공백으로 판정하는데, 현재 구성은 지표 엔드포인트와 콜백이 모두 꺼져 있습니다. 다음 표는 공백마다 그 결과로 내릴 수 없는 운영 판단을 대응시킨 것입니다.

관측 공백	원인이 되는 구성	내릴 수 없는 운영 판단
지연, 첫 토큰 시간	지표 엔드포인트 꺼짐	응답이 느릴 때 프록시와 추론 서버 중 조치 대상 선택
폴백·쿨다운 발생	지표 엔드포인트 꺼짐	기본 모델 대신 35B급 모델이 답한 시간대 파악
토큰 사용량	지표 엔드포인트 꺼짐, 성공 콜백 없음	추론 서버 증설 시기와 클라우드 경로 지출 예측
모델 별칭별 요청 수	로그에 별칭 없음	쓰이지 않는 별칭 정리와 배포 추가 대상 선정
소비자별 사용량	마스터 키 공유	에이전트별 비용 귀속과 예산 한도 설정
에이전트에서 모델까지의 연결	트레이스 수집 없음	대화 한 번이 만든 프록시 요청을 묶어 추적

지연을 볼 수 없다는 공백은 30분 타임아웃과 겹칠 때 영향이 커집니다. 타임아웃이 길면 응답이 늦어져도 타임아웃 오류가 나지 않을 수 있는데, 지연 지표도 없으므로 그때는 느려진 상태가 오류로도 지표로도 드러나지 않을 수 있습니다. 추론 서버의 처리 속도가 서서히 떨어지는 경우라면 프록시에서는 알아차리기 어렵고, 에이전트의 응답이 늦다는 사실만 사용자 쪽에서 확인될 수 있습니다.

폴백을 볼 수 없다는 공백은 2차 게이트웨이 의존과 연결됩니다. 폴백 3건이 모두 쿠버네티스 2차 게이트웨이 뒤의 35B급 모델로 향하므로, 온프레미스 vLLM이 응답하지 않으면 쿨다운 동안 기본 별칭의 요청이 2차 게이트웨이로 넘어갑니다. 이때 프록시의 배포 상태 지표와 폴백 지표가 있어야 두 백엔드의 상태를 한 화면에서 볼 수 있는데, 현재는 두 지표 모두 수집되지 않습니다. 프록시를 거치지 않는 경로 3건은 지표를 켜 뒤에도 프록시의 지표에 나타나지 않으므로, 관측 범위는 프록시를 경유하는 호출로 한정됩니다.

8.4 데이터베이스 없이 얻는 지표와 가상 키가 있어야 나뉘는 지표의 구분

LiteLLM Proxy의 지표 수집은 선행 조건에 따라 두 부분으로 나뉩니다. 폴백 지표와 지연 지표는 데이터베이스 없이도 나오고, 카·팀·사용자별 지표는 가상 키가 있어야 값이 나뉘며, 가상 키는 Postgres가 있어야 쓸 수 있습니다[S03][S05][S06]. 따라서 지표 수집을 한 번에 끝내는 작업으로 계획하면 데이터베이스 도입 일정에 전체가 묶이게 됩니다.

오픈마루 구성은 YAML 파일 한 개로 운영하는 구성이므로 이 구분이 그대로 적용됩니다. 첫째 부분은 설정 파일에 콜백을 추가해 프록시에 반영하면 얻을 수 있고, 둘째 부분은 Postgres를 연결하고 소비자별 가상 키를 발급한 뒤에야 의미 있는 값이 나오므로, 두 부분은 서로 다른 단계의 작업으로 나뉘야 합니다.

비용 조건도 확인해 둘 필요가 있습니다. Prometheus 지표는 LiteLLM 공개판에 포함된 기능이며, 유료판 전용으로 표시된 지표 묶음은 Managed Batch뿐입니다[S05]. 이 장에서 다룬 다섯 묶음과 예산 지표는 라이선스를 추가로 구매하지 않고 수집할 수 있습니다.

8.4.1 콜백 설정 한 줄로 얻는 지연·폴백·배포 상태 지표

Prometheus 지표는 설정 파일에 `callbacks: ["prometheus"]` 한 줄을 넣어 켜고, 프록시는 지표를 `/metrics` 경로로 노출합니다[S05]. 이 콜백을 켜면 데이터베이스 없이 다음 지표를 얻습니다.

- 지연: 요청 전체 지연, 모델 API 호출 지연, 프록시 자체 지연, 대기열 시간
- 첫 토큰 시간: `litellm_llm_api_time_to_first_token_metric`
- 폴백: 폴백 성공 횟수와 실패 횟수
- 쿨다운: 배포가 쿨다운에 들어간 횟수
- 배포 상태: 배포 상태와 배포별 성공·실패 응답 수

이 지표들은 8.3절에서 정리한 공백 가운데 지연, 첫 토큰 시간, 폴백 발생에 해당합니다. 오픈마루 구성에서는 데이터베이스를 붙이기 전에 설정 변경만으로 이 공백들부터 채울 수 있습니다. 다만 지표를 수집해 저장할 Prometheus 서버가 운영 대상에 하나 더해진다는 점은 계획에 넣어야 합니다. 지표를 내보내는 설정과 지표를 모아 보는 수집기는 별개의 작업이기 때문입니다.

버전을 올릴 때는 인증 조건이 달라질 수 있습니다. `/metrics` 경로는 v1.85.0부터 기본으로 Bearer 토큰 인증을 요구하므로, 그 버전 이상으로 올리면 수집기 설정에 토큰을 넣어야 합니다[S05]. 오픈마루 구성은 1.81 계열을 쓰고 있지만 이미지 태그가 latest여서, 이미지를 다시 받으면 버전이 함께 올라갈 수 있습니다. 지표 수집을 켜는 작업은 이미지 버전을 고정하는 작업과 함께 계획해야 하며, 버전을 올리는 시점에 수집기의 인증 설정을 같이 바꿔야 수집이 끊기지 않습니다.

지표를 켜 뒤 먼저 경보를 걸 대상은 세 가지로 좁힐 수 있는데, 폴백 실패 횟수가 늘면 호출한 쪽에 오류가 전달됐을 수 있으므로 즉시 알림 대상으로 둘 만합니다. 쿨다운 발생은 배포가 요청 대상에서 제외됐다는 뜻이므로 점검 대상입니다. 첫 토큰 시간의 추세는 추론 서버의 처리 속도가 떨어지는 경우를 오류가 나기 전에 알려 줄 수 있습니다.

8.4.2 가상 키와 Postgres 가 먼저 필요한 키·팀·사용자별 비용과 예산 지표

키·팀·사용자별 지출과 남은 예산 지표는 선행 조건이 있는 지표입니다. 남은 예산 지표로는 `litellm_remaining_team_budget_metric`, `litellm_remaining_api_key_budget_metric`, `litellm_remaining_user_budget_metric` 이 있으며, 이 값들은 가상 키가 있어야 키·팀·사용자별로 나뉘고, 가상 키를 쓰려면 Postgres 데이터베이스가 필요합니다[S03][S05]. 의존 관계를 한 줄로 요약하면 다음과 같습니다.

Postgres 연결 → 소비자별 가상 키 발급 → 키·팀·사용자별 지출 지표와 남은 예산 지표

다음 표는 지표 묶음과 선행 조건을 대응시킨 것입니다.

지표 묶음	선행 조건	오픈마루 구성에서 필요한 작업
지연, 첫 토큰 시간	Prometheus 콜백	설정 파일 변경
폴백·쿨다운, 배포 상태	Prometheus 콜백	설정 파일 변경
키·팀·사용자별 지출	Postgres, 가상 키	데이터베이스 연결과 소비자별 키 발급
키·팀·사용자별 남은 예산	Postgres, 가상 키, 예산 설정	위 작업 뒤 키별 예산 지정

마스터 키 하나를 공유하는 상태에서 Prometheus 콜백만 켜면 전체 합계는 보이지만 소비자 6종의 사용량은 구분되지 않으므로, 누가 얼마나 쓰는지를 알려면 키 분리가 지표 수집보다 먼저입니다. 반면 느려진 원인을 찾거나 폴백 발생을 아는 데에는 키 분리가 필요하지 않으므로, 이 지표들까지 데이터베이스 도입 뒤로 미룰 이유는 없습니다. 이 의존 관계가 9장에서 조치 순서를 정하는 근거가 됩니다.

키를 어느 단위로 나눌지에 따라 지표의 해상도가 정해집니다. 소비자 6종에 키를 하나씩 발급하면 에이전트 스택, 보조 스크립트, 검색 서비스 단위로 사용량이 나뉘지만, 에이전트별 사용량까지 보려면 한 단계가 더 필요합니다. 오픈마루 구성에서 Hermes Agent 프로파일들은 컨테이너 환경변수로 받은 프록시 주소와 키를 공유하므로, 프로파일마다 다른 키를 쓰도록 키 전달 방식을 먼저 바꿔야 합니다. 그래서 가상 키를 발급하기 전에 사용량을 나눠 볼 단위를 정하고, 그 단위에 맞춰 키를 전달하는 방법을 함께 설계해야 합니다.

9장: 한계와 다음 단계 그리고 기술적 우위 요약: 의존 관계에 따른 조치 순서

오픈마루가 도커로 운영하는 LiteLLM Proxy 구성에는 한계가 일곱 건 있으며, 그 내용은 마스터 키 공유, 데이터베이스 없음, 지표와 트레이스 수집 없음, 프록시를 거치지 않는 스크립트, 단일 컨테이너, 별칭당 배포 하나, latest 이미지 태그입니다. 이 구성을 사례로 드는 이유는 설정 파일 한 개와 컨테이너 한 개로 LLM Gateway를 시작하는 조직이라면 비슷한 공백을 만날 수 있기 때문입니다.

호출 경로를 한 곳으로 모으면 그 한 곳의 통제 수준이 전체의 통제 수준이 됩니다. 오픈마루의 구성에서는 에이전트 프로파일 16개와 LLM을 쓰는 예약 작업 48건이 프록시 하나를 거쳐 모델을 호출합니다. 프록시의 키 관리가 느슨하면 모든 에이전트의 키 관리가 느슨해지고, 프록시가 멈추면 모든 에이전트가 함께 멈춥니다.

공급망도 같은 이유로 중요한데, 2026년 3월 24일에는 LiteLLM의 PyPI 패키지 두 버전이 악성 코드가 들어간 채 약 40분 동안 배포된 일이 있었습니다[S07]. LLM Gateway에는 프로바이더 키가 모두 모이므로, 게이트웨이를 실행하는 이미지가 어느 버전인지 운영자가 직접 정하고 있어야 합니다.

이 장에서는 한계 일곱 건을 LLM Gateway 요건 평가표로 다시 판정하고, 공급망 사고의 경위를 공식 사고 보고에 따라 정리했습니다. 이어서 다음 단계를 낱자 없이 의존 관계에 따른 순서로 제시하고, LiteLLM Proxy의 기술적 우위와 도입 판단 기준으로 문서를 담습니다. 독자는 자기 환경에서 어떤 조치를 먼저 할 수 있고 어떤 조치에 선행 조건이 있는지를 여기서 가져갈 수 있습니다.

9.1 현재 구성의 한계 7건과 LLM Gateway 요건 평가표 재판정

오픈마루의 LiteLLM Proxy는 데이터베이스와 캐시를 붙이지 않은 단일 컨테이너이고, 인증 수단은 마스터 키 하나입니다. 이 구성은 모델 별칭 9개와 폴백 3건으로 호출 경로를 통일하는 데까지는 이르렀지만, 소비자를 구분하는 키, 지연을 보는 지표, 프록시 자체의 가용성은 아직 갖추지 않았습니다. 같은 구성으로 출발하는 조직이 자기 위치를 가늠할 수 있도록, 4장의 요건 평가표 항목을 그대로 가져와 현재 구성에 적용했습니다.

판정 값은 충족, 부분 충족, 공백 세 가지이고, 기본 요건 네 항목과 상세 요건 네 항목을 차례로 놓아 판정마다 근거가 되는 구성 사실을 한 줄로 적었습니다.

구분	요건 항목	판정	근거가 되는 구성 사실
기본	OpenAI 호환 엔드포인트와 모델 별칭	부분 충족	별칭 9개가 한 엔드포인트로 묶였으나 프록시를 거치지 않는 경로 3건이 남아 있다
기본	키 관리	부분 충족	프로바이더 키는 프록시가 갖지만 소비자 전부가 마스터 키 하나를 공유한다
기본	폴백과 안정성 설정	충족	재시도 1회, 쿨다운 300초, 폴백 3건
기본	요청·응답 로깅	부분 충족	컨테이너 표준 출력 로그로 요청·오류 건수만 확인된

구분	요건 항목	판정	근거가 되는 구성 사실
			다. 모델 별칭은 남지 않는다
상세	소비자별 가상 키와 예산	공백	데이터베이스가 없어 쓸 수 없다
상세	지연·첫 토큰 시간 관측	공백	지표 엔드포인트와 콜백이 꺼져 있다
상세	공급망 통제	공백	커뮤니티 파생 이미지를 latest 태그로 쓴다
상세	LLM Gateway 가용성	부분 충족	자동 재시작은 있으나 인스턴스가 하나다

충족으로 판정한 항목은 풀백과 안정성 설정 하나이고 부분 충족이 넷, 공백이 셋이며, 한계 일곱 건은 이 표의 공백과 부분 충족 항목에 빠짐없이 대응합니다. 마스터 키 공유는 키 관리에, 데이터베이스 부재는 소비자별 가상 키와 예산에, 지표와 트레이스 수집 부재는 지연·첫 토큰 시간 관측과 요청·응답 로깅에 해당합니다. 프록시를 거치지 않는 스크립트는 OpenAI 호환 엔드포인트와 모델 별칭에, 단일 컨테이너와 별칭당 배포 하나는 LLM Gateway 가용성에 해당합니다. latest 태그는 공급망 통제에 해당하며, 오픈마루가 쓰는 이미지는 LiteLLM 프로젝트의 공식 이미지가 아니라 LiteLLM을 담아 커뮤니티가 다시 빌드한 파생 이미지입니다.

공백 셋과 부분 충족 넷이라는 결과는 LiteLLM Proxy의 기능 범위에서 온 것이 아닙니다. 가상 키, 지출 추적, 예산, Prometheus 지표는 모두 공개판에 들어 있는 기능이기 때문입니다[S04][S05]. 이미지 버전 고정도 운영자가 정하는 설정입니다. 충족에 이르지 못한 일곱 항목은 현재 구성이 해당 기능과 설정을 쓰지 않거나 선행 조건인 데이터베이스를 두지 않은 데서 비롯했고, 그래서 다음 단계는 제품 교체 없이 구성 변경만으로 이어집니다.

9.1.1 마스터 키 공유와 데이터베이스 부재에서 오는 통제 공백

LiteLLM Proxy를 호출하는 소비자 6종, 곧 업무 에이전트 스택, 콘텐츠 공장 스택, 에이전트 콘솔, 에이전트 컨테이너 안의 보조 스크립트, 슬라이드 검색 서비스, 영상 제작 도구는 모두 같은 마스터 키를 씁니다. 그래서 프록시에서는 최근 사흘 동안 들어온 대화 완성 요청 약 2,500건을 소비자별로 나눌 수 없습니다. 어느 에이전트가 모델 자원을 많이 쓰는지 프록시 밖에서 에이전트의 세션 수로 짐작할 수밖에 없습니다.

키 유출에 대응하는 범위도 같은 이유로 넓어져서, 하나뿐인 키가 새면 모든 소비자의 키를 한꺼번에 바꿔야 하고, 특정 소비자만 차단하는 조치는 할 수 없습니다. 마스터 키는 다른 키를 만드는 관리자 키 역할을 하므로[S03], 일반 호출에 쓰는 키와 관리 권한을 가진 키가 구분되지 않은 상태입니다.

가상 키를 쓰려면 Postgres 데이터베이스가 있어야 하는데[S03], 오픈마루의 구성은 읽기 전용 YAML 파일 한 개가 설정의 전부이므로 가상 키, 팀, 예산, 속도 제한, 사용 기록을 모두 쓰지 않습니다. 가상 키가 제공하는 지출 추적, 모델 접근 통제, 예산 세 가지가 함께 비어 있는 셈이어서, 예를 들어 비전 작업에만 쓰려고 둔 클라우드 모델 별칭도 키를 가진 소비자라면 누구나 호출할 수 있습니다.

두 한계와 그 결과, 대응하는 다음 단계는 아래 표와 같고, 두 한계는 원인이 이어져 있어서 대응 단계도 데이터베이스 연결과 가상 키 발급이 한 묶음으로 움직입니다.

한계	결과	다음 단계
마스터 키 하나를 모든 소비자가 공유	소비자별 사용량을 나눌 수 없고, 키가 새면 전부 교체해야 한다	에이전트와 서비스별 가상 키 발급
데이터베이스 없음	가상 키, 예산, 속도 제한, 사용 기록을 쓸 수 없다	Postgres 연결

9.1.2 관측 부재, 프록시 밖 경로, 단일 컨테이너, 별칭당 배포 하나의 영향

현재 구성에는 성공 콜백이 없고 Prometheus 지표 엔드포인트가 꺼져 있으며 OpenTelemetry 연동도 없습니다. 남은 기록은 컨테이너 표준 출력 로그뿐이고 이 로그에는 요청별 모델 별칭이 남지 않습니다. 이 상태에서는 요청 지연, 첫 토큰 시간(TTFT, 요청을 보낸 뒤 첫 응답 토큰이 도착하기까지의 시간), 토큰 사용량은 프록시에서 볼 수 없습니다. 폴백도 마찬가지로, 최근 사흘 로그에 폴백과 쿨다운 기록이 없다는 사실 외에는 폴백 경로가 얼마나 자주 쓰이는지를 수치로 추적할 수단이 없습니다.

프록시를 거치지 않는 경로는 세 건으로, AI 검색 인용을 주마다 점검하는 스크립트와 위키 정리 스크립트는 클라우드 프로바이더 API를 직접 호출하고, 이미지 생성 백엔드 하나는 모델 중개 서비스를 직접 호출합니다. 이 호출들은 프록시의 키 통제와 집계에서 빠져 있으므로, 프록시에 가상 키와 지표를 갖추더라도 이 세 경로의 비용과 사용량은 계속 따로 조사해야 합니다.

프록시는 컨테이너 한 개이므로 프록시가 멈추면 소비자 6종이 모두 모델을 호출하지 못합니다. 자동 재시작은 켜져 있지만 재시작하는 동안의 요청을 대신 받을 복제본은 없습니다. 별칭마다 배포가 하나씩이어서 부하 분산을 하지 않습니다. LiteLLM은 같은 모델 이름으로 배포를 여러 개 두면 요청을 나눠 보내지만[S06], 현재 구성에서는 백엔드 한 곳이 느려졌을 때 폴백으로 다른 모델에 넘기는 방법만 남아 있습니다.

네 한계의 결과와 다음 단계를 한 표에 모으면, 관측은 설정 변경으로 바로 시작할 수 있고 나머지 셋은 구조를 바꾸는 작업이라는 점이 드러납니다.

한계	결과	다음 단계
지표·트레이스 수집 없음	지연, 첫 토큰 시간, 폴백 발생, 토큰 사용량을 볼 수 없다	Prometheus 지표 수집, OpenTelemetry 트레이스 연결
프록시를 거치지 않는 스크립트	해당 호출이 통제와 집계 밖에 있다	프록시 경유로 이전, 클라우드 키를 프록시로 집중
프록시가 단일 컨테이너	프록시가 멈추면 모든 에이전트가 멈춘다	복제와 헬스체크, 쿠버네티스 이전 검토
별칭당 배포 하나	부하 분산이 없다	같은 모델을 여러 배포로 구성

9.2 2026년 3월 공급망 사고의 경위와 latest 태그 구성에 주는 의미

LiteLLM은 2026년 3월 24일 PyPI 패키지 두 버전이 악성 코드를 담은 채 배포되는 공급망 사고를 겪었습니다[S07]. LLM Gateway 도입을 검토하는 조직이 이 사고를 알아야 하는 이유는 게이트웨이에 프로바이더 키가 모두 모여 있기 때문입니다. 호출 경로를 한 곳으로 모을수록 그 한 곳을 실행하는 소프트웨어의 출처와 버전이 더 중요해집니다.

이 사고에서 확인할 사실은 두 가지인데, 하나는 공식 도커 이미지로 LiteLLM Proxy를 운영한 쪽은 영향을 받지 않았다는 점입니다[S07]. 다른 하나는 오픈마루의 구성처럼 커뮤니티 파생 이미지를 latest 태그로 쓰면 어떤 버전이 실행되는지를 운영자가 직접 정하지 않는다는 점입니다. 오픈마루 사례를 함께 놓은 이유는 도커로 운영한다는 사실만으로 버전 통제가 갖춰지지 않는다는 점을 보이기 위해서입니다.

사고의 경위와 현재 구성의 태그 정책은 나눠서 봐야 합니다. 경위는 패키지 배포 경로에서 일어난 일이고 태그 정책은 운영자가 선택하는 설정이므로, 태그 정책은 선행 조건 없이 바로 바꿀 수 있습니다.

9.2.1 PyPI 악성 패키지 1.82.7·1.82.8 의 약 40분 노출 경위와 수집 대상

침해는 LiteLLM의 CI/CD(지속적 통합과 배포) 파이프라인에서 시작됐습니다. 보안 점검 단계에서 쓰던 Trivy 의존성이 먼저 침해됐고, 공격자는 정식 배포 절차를 거치지 않고 악성 패키지를 PyPI에 직접 올렸습니다. 문제가 된 버전은 1.82.7과 1.82.8이고, 두 버전은 10시 39분(UTC)부터 약 40분 동안 내려받을 수 있었고, 이후 PyPI가 격리했습니다[S07].



그림. 2026년 3월 24일 LiteLLM PyPI 패키지 공급망 사고의 진행 순서와 영향 범위

악성 코드는 설치된 환경에서 자격증명을 모아 암호화한 뒤 외부로 보냈으며, 수집 대상은 다음 다섯 가지입니다[S07].

- 환경변수
- SSH 키
- AWS, GCP, Azure의 클라우드 자격증명
- 쿠버네티스 토큰
- 데이터베이스 비밀번호

감염된 환경에는 `litellm_init.pth` 파일이 남으므로 이 파일의 존재 여부가 점검 기준이 됩니다[S07].

수집 대상 목록은 LLM Gateway가 보관하는 값과 겹칩니다. 게이트웨이에는 조직이 쓰는 프로바이더 키가 모이고, 데이터베이스를 연결했다면 접속 비밀번호도 함께 놓입니다. 따라서 게이트웨이가 침해되면 게이트웨이에 등록된 프로바이더 키를 모두 교체해야 할 수 있습니다. 호출 경로를 모으는 결정은 키를 모으는 결정이기도 하므로, 공급망 통제가 요건 평가표의 한 항목으로 들어가 있습니다.

9.2.2 공식 도커 이미지의 의존성 고정과 이미지 버전·다이제스트 고정 조치

공식 LiteLLM Proxy 도커 이미지를 쓴 운영 환경은 이 사고의 영향을 받지 않았습니다. 이미지가 의존성 버전을 고정해 두고 있어서 침해된 PyPI 패키지를 가져오지 않았기 때문입니다[S07]. 반면 악성 패키지 두 버전은 약 40분 동안 PyPI에서 내려받을 수 있는 상태였습니다[S07]. 같은 소프트웨어라도 의존성 버전을 고정한 배포 경로는 그 패키지를 가져오지 않았습니다.

공식 사고 보고가 권고한 조치는 세 가지입니다[S07].

- 비밀 값 전부 교체
- `litellm_init.pth` 파일 점검
- 안전한 버전으로 고정

세 번째 권고는 사고를 겪지 않은 조직에도 그대로 적용되는데, 버전을 고정해 두면 새 릴리스에 문제가 생겨도 운영 환경에 들어오기 전에 확인할 시간이 생기기 때문입니다.

오픈마루의 LiteLLM Proxy 컨테이너는 커뮤니티 파생 이미지를 latest 태그로 씁니다. latest 태그는 이미지를 내려받은 시점의 최신 이미지를 가리키므로, 이미지를 다시 받으면 버전이 바뀔 수 있고 그 변경을 운영자가 사전에 검토하지 않습니다. 또한 파생 이미지는 공식 이미지와 빌드 주체가 다르므로, 공식 사고 보고가 밝힌 공식 이미지의 의존성 고정이 이 이미지에도 같다고 전제할 수 없습니다. 같은 호스트의 콘텐츠 공장 스택은 에이전트 이미지를 다이제스트로 고정해 두었으므로, 프록시에도 같은 방식을 적용하면 됩니다. 버전 고정 전과 후의 배포 절차는 아래와 같이 달라집니다.

항목	latest 태그	버전·다이제스트 고정
실행 버전을 정하는 주체	이미지를 내려받은 시점	설정 파일에 적은 버전과 다이제스트
버전이 바뀌는 계기	컨테이너 재생성	운영자의 설정 변경
새 릴리스 반영	검토 없이 반영	릴리스 확인 뒤 반영
문제 발생 때 되돌리기	이전 버전을 따로 찾아야 한다	직전 설정으로 복구

9.3 다음 단계의 의존 순서: 호출 경로 조사, 버전 고정과 지표 수집, Postgres 와 가상 키, 프록시 밖 경로 이전, 복제

다음 단계는 일곱 가지이고 순서는 날짜가 아닌 의존 관계로 정했으며, 호출 경로 조사, 버전·다이제스트 고정, Prometheus 지표 수집, Postgres 연결, 가상 키 발급과 키별 지표 분리, 프록시 밖 경로 이전, 복제와 헬스체크 순입니다. 오픈마루 구성의 한계에서 나온 순서이지만, 단계마다 선행 조건을 적어 두었으므로 다른 환경에서는 이미 갖춘 단계를 건너뛰고 적용할 수 있습니다.

순서를 정하는 의존 관계는 두 가지입니다. 첫째로 키와 팀별로 나뉜 지표는 가상 키가 있어야 나오고 가상 키는 데이터베이스가 있어야 합니다[S03][S05]. 둘째로 폴백과 지연 지표는 데이터베이스 없이도 나옵니다[S05][S06]. 그래서 선행 조건이 없는 조치를 먼저 실행해 관측을 확보하고, 데이터베이스가 필요한 조치를 그 뒤에 둡니다.

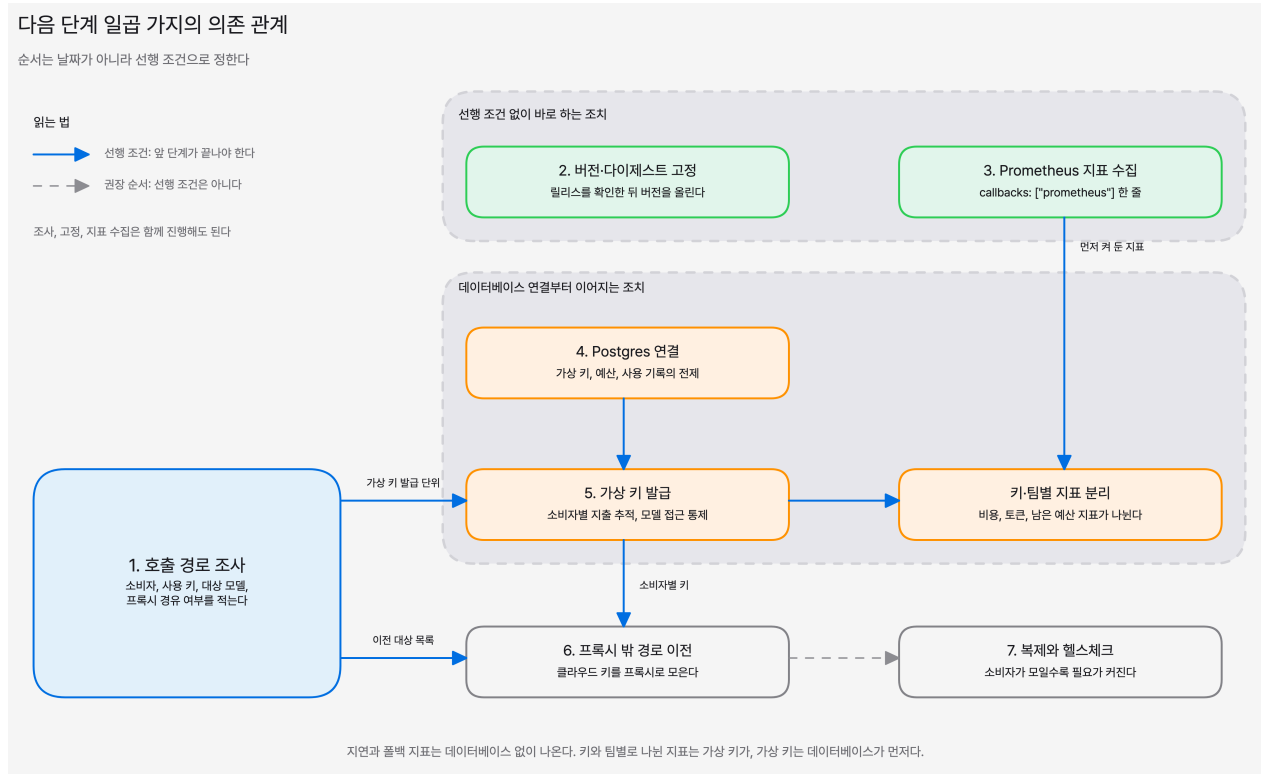


그림. 다음 단계 일곱 가지의 의존 관계: 선행 조건 없이 바로 하는 조치와 데이터베이스 연결부터 이어지는 조치

조사를 맨 앞에 둔 이유는 나머지 단계의 대상이 조사 결과에서 정해지기 때문입니다. 가상 키를 몇 개 발급할지, 어떤 스크립트를 프록시로 옮길지는 소비자 목록이 있어야 정할 수 있습니다. 반면 버전 고정과 지표 수집은 조사 결과를 기다리지 않아도 되므로 조사와 함께 진행해도 됩니다.

9.3.1 호출 경로 조사와 선행 조건 없는 조치: 버전·다이제스트 고정, Prometheus 지표 수집

첫 단계는 1장에서 제시한 조사표를 채우는 일로, LLM을 호출하는 주체마다 소비자, 사용 키, 대상 모델, 프록시 경우 여부를 적습니다. 오픈마루의 경우 이 조사에서 프록시를 호출하는 소비자 6종과 프록시를 거치지 않는 경로 3건이 나왔습니다. 조사에는 선행 조건이 없고, 결과물은 가상 키 발급 단위와 이전 대상 목록으로 그대로 쓰입니다.

두 번째 조치는 프록시 이미지의 버전과 다이제스트를 설정 파일에 적는 것입니다. 이 조치는 공식 사고 보고가 직접 권고한 내용이고[S07], 데이터베이스나 다른 구성 요소를 요구하지 않습니다. 고정한 뒤에는 새 릴리스의 변경 내용을 확인하고 나서 버전을 올리는 절차가 따라오지만, 운영 부담은 릴리스를 확인하는 일 하나가 늘어나는 정도입니다.

세 번째 조치는 설정 파일에 `callbacks: ["prometheus"]` 한 줄을 넣어 지표 엔드포인트를 켜는 것입니다.

Prometheus 지표는 공개판 기능이고 데이터베이스 없이 동작합니다[S05]. 이 단계에서 요청 지연, 첫 토큰 시간, 폴백 성공과 실패, 쿨다운, 배포 상태 지표가 나오므로 요건 평가표의 지연·첫 토큰 시간 관측 항목이 공백

에서 벗어납니다. 다만 지표 엔드포인트는 v1.85.0부터 기본으로 Bearer 토큰 인증을 요구하므로, 고정하는 버전이 v1.85.0 이상이라면 수집기 설정에 토큰을 함께 넣어야 합니다[S05].

이 단계까지 마쳐도 소비자별 사용량은 아직 나뉘지 않는데, 모든 소비자가 같은 키를 쓰는 동안에는 비용과 토큰 지표가 프록시 전체 합계로만 나오기 때문입니다. 풀백이 실제로 일어나는지, 어느 별칭의 응답이 느린지는 이 단계에서 수치로 볼 수 있습니다.

9.3.2 Postgres 연결 뒤 에이전트·서비스별 가상 키 발급과 키별 지표 분리

네 번째 조치는 LiteLLM Proxy에 Postgres를 연결하는 것입니다. 연결하면 가상 키로 지출 추적, 모델 접근 통제, 키·사용자·팀 단위 예산을 쓸 수 있고, 이 기능들은 공개판에 들어 있습니다[S03][S04]. 하지만 운영 대상이 하나 늘어납니다. 상태를 갖지 않던 프록시가 데이터베이스에 의존하게 되므로, 데이터베이스의 백업과 가용성도 프록시 운영의 일부가 됩니다. 이 부담을 받아들일지가 Postgres 연결 단계의 판단 기준입니다.

다섯 번째 조치는 조사에서 나온 소비자 목록에 따라 가상 키를 발급하는 것입니다. 발급 단위는 6장의 소비자 6종과 7장의 에이전트 역할표를 따릅니다. 지금은 LiteLLM Proxy의 주소와 키가 컨테이너 환경변수에 들어 있어 한 스택의 모든 프로파일이 같은 값을 공유합니다. 따라서 에이전트별로 키를 나누려면 키를 두는 위치도 컨테이너 단위에서 프로파일 단위로 옮겨야 합니다. 가상 키를 발급한 뒤에는 마스터 키를 관리 용도로만 남기고 일반 호출에서 빼는 작업이 이어집니다.

키를 한꺼번에 나누기 어렵다면 호출이 많은 소비자와 외부로 나가는 소비자부터 나눕니다. 오픈마루 구성에서는 세션 수가 가장 많은 영업 에이전트와 예약 작업이 가장 많은 마케팅 에이전트가 앞쪽에 오고, 클라우드 별칭을 쓰는 보조 스크립트도 과금되는 경로를 호출하므로 먼저 분리할 대상입니다. 반대로 모델 목록 조회와 점검 호출만 하는 에이전트 콘솔은 사용량이 작아 뒤로 미뤄도 됩니다. 이 순서를 적는 이유는 소비자 수만큼 키를 만드는 일보다 사용량과 외부 전송이 큰 곳부터 가르는 일이 먼저라는 판단을 보여 주기 때문입니다.

가상 키가 발급되면 앞 단계에서 켜 둔 Prometheus 지표 가운데 비용 지표, 토큰 지표, 남은 예산 지표가 키와 팀별로 나뉩니다[S05]. 지표 수집을 가상 키보다 먼저 켜 두면 키를 나누기 전과 후의 사용량을 같은 지표로 비교할 수 있다는 이점도 있습니다. 네 번째와 다섯 번째 조치의 선행 조건과 결과는 아래 표와 같습니다.

단계	선행 조건	얻는 것
Postgres 연결	없음. 데이터베이스 운영 준비	가상 키, 예산, 속도 제한, 사용 기록을 쓸 수 있는 상태
가상 키 발급	Postgres 연결, 호출 경로 조사 결과	소비자별 지출 추적, 모델 접근 통제, 키 단위 교체
키·팀별 지표 분리	가상 키 발급, Prometheus 지표 수집	소비자별 비용, 토큰, 남은 예산 지표

9.3.3 프록시 밖 경로의 프록시 경유 이전과 복제·헬스체크·쿠버네티스 이전 검토

여섯 번째 조치는 프록시를 거치지 않는 스크립트를 프록시 경유로 옮기고 클라우드 키를 프록시로 모으는 것입니다. 오픈마루의 경우 클라우드 프로바이더를 직접 호출하는 점검 스크립트와 정리 스크립트, 모델 중개 서비스를 직접 호출하는 이미지 생성 백엔드가 대상입니다. 이 조치를 가상 키 뒤에 둔 이유는 옮겨 온 스크립트에도 소비자별 키를 발급해야 사용량이 나뉘기 때문입니다. 이전을 마치면 LLM 호출 경로가 실제로 하나가 되고,

OpenAI 호환 엔드포인트와 모델 별칭 항목은 모든 소비자가 엔드포인트 한 곳으로 호출한다는 충족 기준을 채우게 됩니다.

이전의 효과는 사용량 집계에 그치지 않습니다. 세 경로는 모두 조직 밖의 API로 향하므로, 프록시 경유로 옮기면 외부로 나가는 호출이 프록시 설정 파일의 별칭 목록 안에 모두 들어옵니다. 그 뒤에는 어떤 소비자가 어느 외부 프로바이더를 호출할 수 있는지를 가상 키의 모델 접근 범위로 정할 수 있고, 외부 전송 경로를 묻는 질의에도 설정 파일과 키 목록으로 답할 수 있습니다. 반대로 이전을 미루면 그 스크립트들의 프로바이더 키는 계속 각자의 실행 환경에 남습니다.

일곱 번째 조치는 프록시의 가용성을 높이는 것이고, 검토 대상은 세 가지입니다. 프록시 컨테이너의 복제와 헬스체크, 같은 모델 이름에 배포를 여러 개 두는 부하 분산[S06], 쿠버네티스 이전입니다. 세 가지 가운데 쿠버네티스 이전은 운영 환경을 바꾸는 결정이라서 검토 항목으로 남겨 둡니다. 이 단계를 뒤에 둔 이유는 소비자가 프록시 한 곳으로 모일수록 프록시 정지의 영향이 커지고, 그만큼 복제의 필요가 분명해지기 때문입니다.

OpenTelemetry 트레이스로 에이전트, 프록시, 모델 서버의 호출을 하나로 잇는 작업도 이 단계에 둡니다. Prometheus 지표는 프록시가 본 수치만 알려 주므로, 에이전트의 도구 호출과 모델 호출이 한 대화 안에서 어떻게 이어지는지는 트레이스가 있어야 볼 수 있습니다. 다만 OpenTelemetry GenAI 시맨틱 컨벤션은 지표 문서와 토큰 지표 문서가 모두 Development 단계여서 이름이 바뀔 수 있습니다[S09]. 트레이스 연결은 앞의 조치보다 늦게 시작해도 되고, 속성 이름의 변경을 따라갈 준비를 함께 해야 합니다.

단계	선행 조건	얻는 것
프록시 밖 경로 이전	호출 경로 조사 결과, 가상 키 발급	모든 LLM 호출이 통제와 집계 안에 들어온다
복제와 헬스체크	없음. 소비자가 모일수록 필요가 커진다	프록시 정지 때 요청을 받을 복제본
같은 모델의 여러 배포 구성	같은 모델을 제공하는 백엔드가 둘 이상	별칭 단위 부하 분산
OpenTelemetry 트레이스 연결	에이전트와 프록시 양쪽의 트레이스 설정	에이전트, 프록시, 모델 호출이 이어진 추적

9.4 LiteLLM Proxy 기술적 우위 요약과 도입 판단 기준

LiteLLM Proxy의 기술적 우위는 네 가지로 요약되며, OpenAI 형식 하나로 100개 이상의 프로바이더를 호출한다는 점, 가상 키부터 Prometheus 지표까지 공개판에 들어 있다는 점, MIT 라이선스라는 점, 모델 별칭 정의만 바꿔 모델을 교체한다는 점입니다. 9.1절에서 한계 일곱 건을 먼저 정리했으므로, 이 우위들은 모두 조건과 함께 읽어야 합니다.

채택 규모도 판단 자료가 되는데, LiteLLM 저장소의 스타는 60,173개, 포크는 12,046개이며, 스타 수는 5장에서 비교한 다섯 후보 가운데 가장 많습니다[S11]. 이 수치는 2026년 10월 6일 GitHub API 조회값입니다. 다만 스타 수는 관심의 크기를 보여 줄 뿐 운영 적합성을 보증하지 않으므로, 아래 우위 요약표에는 기능과 구성으로 확인되는 항목만 넣었습니다.

도입 여부는 리스크 수용 가능성, 운영 부담, 대안 대비 우위 세 기준으로 판단합니다. 오픈마루의 운영 구성을 근거로 드는 이유는 이 세 기준에 답할 구체적인 사실을 운영 중인 구성에서 가져올 수 있기 때문입니다. 어떤 기능이 설정 파일만으로 동작하고 어떤 기능에 데이터베이스가 필요한지가 그 사실에 해당합니다.

9.4.1 OpenAI 형식 단일 호출, 공개판 기능 범위, MIT 라이선스, 별칭 기반 모델 교체

LiteLLM Proxy의 우위는 근거와 조건을 나란히 놓아야 과장 없이 읽히므로, 아래 표는 우위마다 근거 출처와 그 우위가 성립하는 조건을 함께 적은 것입니다.

우위	근거	조건
OpenAI 형식 하나로 100개 이상 프로바이더 호출	공식 저장소와 문서[S01][S02]	프로바이더마다 받는 파라미터가 다르다. 오픈마루 구성은 받지 않는 파라미터를 버리도록 설정했다
가상 키, 지출 추적, 예산, 폴백, 로깅, Prometheus 지표가 공개판에 포함	공식 문서[S04][S05]	가상 키와 예산은 Postgres가 있어야 한다. SSO와 감사 로그는 유료판이다
MIT 라이선스로 상업적 이용 가능	공식 저장소[S01]	저장소의 enterprise/ 디렉터리는 별도 라이선스다
별칭 정의 변경만으로 모델 교체	오픈마루 운영 구성	모든 소비자가 프록시를 경유해야 한다. 프록시 밖 경로는 따로 고쳐야 한다

네 번째 우위는 오픈마루의 구성에서 이미 얻고 있는 이점으로, 에이전트 프로파일 16개의 모델 설정은 모두 같은 기본 별칭과 같은 프록시 주소를 가리킵니다. 그래서 기본 대화 모델을 다른 모델로 바꿀 때 프로파일 16개의 설정은 그대로 두고 프록시 설정 파일의 별칭 정의 한 곳만 고칩니다. 반면 프록시를 거치지 않는 경로 세 건은 이 이점을 얻지 못하므로, 모델을 바꿀 때 경로마다 따로 수정해야 합니다.

LiteLLM Proxy가 모든 조직에 맞는 선택은 아니어서, 파이썬 애플리케이션 하나가 여러 프로바이더를 호출하는 정도라면 LiteLLM SDK만으로 충분할 수 있고, 이때 프록시는 운영 대상만 늘립니다[S02]. 조직의 보안 규정이 관리 화면 SSO나 감사 로그를 요구하면 공개판 범위를 넘어서므로 유료판을 함께 검토해야 합니다[S04]. 이미 API 게이트웨이를 운영하는 조직은 5장에서 다룬 다른 후보가 운영 부담 면에서 더 맞을 수 있습니다.

9.4.2 호출 경로를 LiteLLM Proxy 한 곳으로 모으는 결정과 키 분리·지표 수집의 후속 단계 지정

도입을 판단할 때 확인할 항목을 세 기준에 따라 점검표로 정리했고, 각 항목의 답은 오른쪽 열에 적은 결과 장에서 가져올 수 있습니다.

판단 기준	점검 항목	답을 얻는 근거
리스크 수용 가능성	프록시 정지가 전체 정지로 이어지는 구성을 복제 전까지 받아들일 수 있는가	9.1.2의 가용성 한계
리스크 수용 가능성	이미지 버전과 다이제스트를 고정하고 릴리스를 확인할 담당자가 있는가	9.2.2의 버전 고정 조치[S07]

판단 기준	점검 항목	답을 얻는 근거
운영 부담	가상 키를 위해 Postgres를 운영할 수 있는가	9.3.2의 선행 조건[S03]
운영 부담	지표를 받을 Prometheus 수집 환경이 있는가	9.3.1의 지표 수집[S05]
대안 대비 우위	필요한 기능이 공개판 범위 안에 있는가	9.4.1의 우위 요약표
대안 대비 우위	호출 주체가 둘 이상이고 키를 한 곳에서 관리해야 하는가	1장의 호출 경로 조사표

세 기준에 답한 뒤 내릴 결정은 하나이고 함께 지정할 후속 단계는 둘인데, 결정은 LLM 호출 경로를 LiteLLM Proxy 한 곳으로 모으는 것입니다. 후속 단계는 소비자별 가상 키로 마스터 키 공유를 끝내는 키 분리와, 지연·첫 토큰 시간·폴백 발생을 보는 모니터링 지표 수집입니다. 폴백과 별칭만으로도 프록시는 동작하지만, 키 분리와 지표 수집이 빠지면 호출 경로가 모였을 뿐 통제와 관측은 이전과 같은 수준에 머물렀습니다.

AI 에이전트 운영에서 먼저 정리할 대상은 모델보다 모델로 가는 호출 경로입니다. 경로가 하나로 모여 있으면 에이전트가 늘어도 모델 운영에서 손댈 곳은 프록시 한 곳으로 유지됩니다. 이 문서를 읽은 조직이 할 일은 두 가지입니다. 먼저 에이전트, 스크립트, 서비스에 흩어진 LLM 호출 경로를 조사해 LiteLLM Proxy 한 곳으로 모읍니다. 이어서 키 분리와 모니터링 지표 수집을 다음 단계로 정해 담당자와 선행 조건을 함께 기록합니다.

Appendix A. References

- [S01] BerriAI. "BerriAI/litellm 저장소 README·LICENSE".
<https://github.com/BerriAI/litellm>
- [S02] BerriAI. "LiteLLM 문서 첫 페이지 (SDK와 Proxy 구분)". <https://docs.litellm.ai/docs/>
- [S03] BerriAI. "LiteLLM 문서, Virtual Keys".
https://docs.litellm.ai/docs/proxy/virtual_keys
- [S04] BerriAI. "LiteLLM 문서, Enterprise". <https://docs.litellm.ai/docs/enterprise>
- [S05] BerriAI. "LiteLLM 문서, Prometheus metrics".
<https://docs.litellm.ai/docs/proxy/prometheus>
- [S06] BerriAI. "LiteLLM 문서, Fallbacks (reliability)".
<https://docs.litellm.ai/docs/proxy/reliability>
- [S07] BerriAI. "Security Update: Suspected Supply Chain Incident".
<https://docs.litellm.ai/blog/security-update-march-2026>
- [S08] Y Combinator. "Y Combinator 회사 소개, LiteLLM".
<https://www.ycombinator.com/companies/litellm>
- [S09] OpenTelemetry. "OpenTelemetry GenAI Semantic Conventions (gen-ai-metrics.md, gen-ai-token-metrics.md)". <https://github.com/open-telemetry/semantic-conventions-genai>
- [S10] Nous Research. "Hermes Agent 저장소와 공식 문서 (Profiles, Kanban, Cron, MCP, Running Many Gateways)". <https://github.com/NousResearch/hermes-agent>
- [S11] GitHub. "GitHub API 저장소 메타데이터 (BerriAI/litellm, Kong/kong, maximhq/bifrost, higress-group/higress, theagentrouter/agent-router)".
<https://api.github.com/repos/>
- [S12] Agentic AI Foundation. "Agent Router joins AAIF". <https://aaif.io/blog/agent-router-joins-aaif>

함께 읽을 자료

아래 두 건은 MSAP.ai 블로그에 공개된 글이며, 본문의 사실을 받치는 확인 근거 [S01] 부터 [S12] 까지와 구분해 [R1], [R2] 로 적습니다. 두 글은 같은 주제를 더 읽으려는 독자를 위한 안내이며, 이 번호가 붙은 문장은 해당 글을 근거로 삼지 않습니다.

- [R1] MSAP.ai. "AI 에이전트를 위한 LLM 게이트웨이 — LiteLLM 도입 가이드".
<https://www.msap.ai/resource/openai-hohwan-api/>
- [R2] MSAP.ai. "AI 에이전트란 무엇인가 — 정의, 구조, 기업 도입 기준".
<https://www.msap.ai/blog-home/ai-blog/what-is-ai-agent/>

Appendix B. Glossary

용어	정의
가상 키	마스터 키로 소비자마다 따로 발급하는 키입니다. 지출 추적, 모델 접근 통제, 예산을 키 단위로 적용하며 Postgres 데이터베이스가 있어야 쓸 수 있습니다.
공개판과 유료판	공개판은 누구나 쓸 수 있는 LiteLLM의 기능 범위이고, 유료판은 별도 계약이 필요한 범위입니다. 관리 화면 SSO와 감사 로그는 유료판에 속합니다.
공급망 통제	게이트웨이를 실행하는 이미지의 출처를 확인하고 버전이나 다이제스트를 고정해, 어느 버전이 실행되는지를 운영자가 정하는 일입니다.
다이제스트	이미지 내용에서 계산한 고유 식별값입니다. 다이제스트로 고정한 컨테이너는 같은 태그에 새 이미지가 올라와도 지정한 이미지를 그대로 씁니다.
단일 장애점	한 곳이 멈추면 전체가 멈추는 구성 요소입니다(single point of failure).
마스터 키	프록시 관리자가 쓰는 키이며, 다른 키를 만드는 관리자 키 역할을 합니다.
모델 별칭	호출 주체가 요청에 적는 모델 이름을 실제 백엔드에 연결하는 프록시 설정 항목입니다.
배포	모델 별칭 뒤에 연결된 실제 백엔드 하나를 가리킵니다(deployment). 추론 서버나 프로바이더 모델이 여기에 해당합니다.
부하 분산	같은 모델 이름으로 묶은 배포 여러 개에 요청을 나누어 보내는 기능입니다.
소비자	LiteLLM Proxy를 호출하는 쪽을 가리킵니다. 에이전트, 스크립트, 서비스가 여기에 해당합니다.
시맨틱 컨벤션	지표와 속성의 이름을 정한 규약입니다(semantic conventions).
재시도	실패한 요청을 같은 모델에 다시 보내는 기능입니다.
첫 토큰 시간(TTFT)	요청을 보낸 뒤 응답의 첫 토큰이 도착하기까지 걸린 시간입니다(Time To First Token).
쿨다운	실패가 이어진 백엔드를 정해진 시간 동안 호출 대상에서 빼는 동작입니다.

용어	정의
폴백	한 모델이 실패했을 때 요청을 다른 모델로 넘기는 동작입니다(fallback).
프로바이더	모델 API를 제공하는 사업자입니다.
프로파일	Hermes Agent에서 설정 파일, 메모리, 예약 작업을 따로 갖는 에이전트 단위입니다.
2차 게이트웨이	오픈마루가 쿠버네티스에서 따로 운영하는 또 하나의 LiteLLM 게이트웨이입니다. 폴백 대상 별칭이 이 게이트웨이를 거칩니다.
CI/CD	지속적 통합과 배포를 가리킵니다.
Hermes Agent	Nous Research가 MIT 라이선스로 공개한 오픈소스 에이전트 프레임워크입니다. 사용자가 지정한 엔드포인트의 모델을 코드 변경 없이 바꿔 쓸 수 있습니다.
LiteLLM Proxy	조직이 함께 쓰는 서버 형태의 LiteLLM입니다. OpenAI 호환 엔드포인트로 여러 호출 주체의 모델 호출을 받는 LLM Gateway입니다.
LiteLLM SDK	파이썬 애플리케이션 안에 넣어 쓰는 호출 라이브러리 형태의 LiteLLM입니다.
LLM	대규모 언어 모델입니다(Large Language Model).
LLM Gateway	호출 주체와 모델 사이에서 인증, 라우팅, 기록을 한 곳에서 처리하는 중계 서버입니다.
MCP	에이전트가 외부 도구를 호출할 때 쓰는 규약입니다 (Model Context Protocol).
OpenTelemetry	지표, 로그, 트레이스를 수집하는 오픈소스 관측 프레임워크입니다.
Prometheus	시계열 지표를 주기적으로 수집해 저장하는 오픈소스 모니터링 시스템입니다.
Router	LiteLLM SDK와 LiteLLM Proxy가 함께 쓰는 구성 요소이며 재시도, 폴백, 부하 분산을 제공합니다.
SaaS	공급자가 운영하는 서비스를 구독하는 방식입니다 (Software as a Service).
SSO	통합 로그인을 가리킵니다(Single Sign-On).

LiteLLM Proxy로 세우는 LLM Gateway: AI 에이전트 모델 호출 길목을 하나로 모으는 방법

CONTACT

WEB

msap.ai

www.msap.ai/

EMAIL

hello@msap.ai

TEL

02-6953-5427

0269535427

YOUTUBE

@msaptv

www.youtube.com/@msaptv

LINKEDIN

linkedin.com/showcas...

www.linkedin.com/showcase/msap-ai/

FACEBOOK

facebook.com/opennaru

www.facebook.com/opennaru



SCAN