TECH BRIEF

MSAP.ai

AI 플랫폼을 위한 필수 게이트웨이 LiteLLM

온프레미스 사내 AI 플랫폼을 구축하는

온프레미스 사내 AI 플랫폼을 구축하는 조직이 늘면서, 여러 대규모 언어 모델(LLM)을 일관된 방식으로 다루는 문제가 중요해지고 있습니다. 이 백서는 LiteLLM을 그 해법으로 제시하되, 한 가지 오해를 먼저 바로잡습니다. LiteLLM은 추론(inference) 자체를 빠르게 만드는 도구가 아니라, 여러 모델 앞에서 요청을 표준화하고 분배하는 **게이트웨이**입니다.

2026-07-05 · 작성자: SMART CF Team · msap.ai

목차

AI 플랫폼을 위한 필수 게이트웨이 LiteLLM

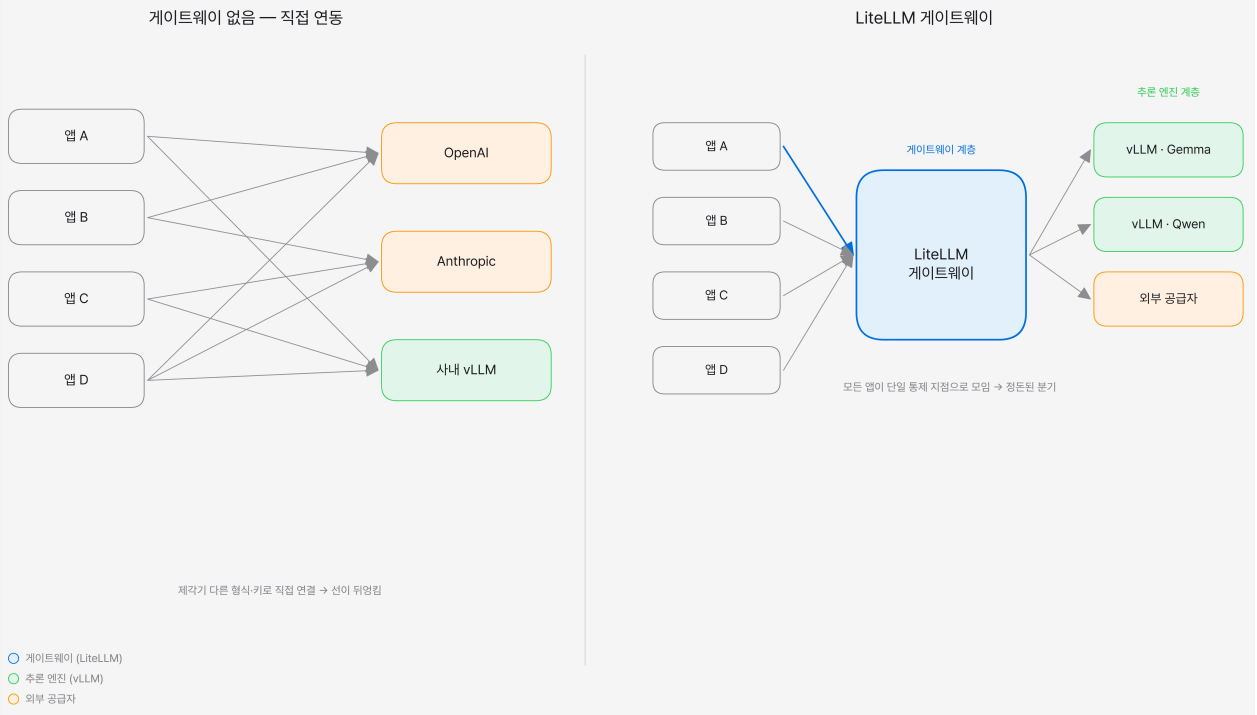
온프레미스 사내 AI 플랫폼을 구축하는 조직이 늘면서, 여러 대규모 언어 모델(LLM)을 일관된 방식으로 다루는 문제가 중요해지고 있습니다. 이 백서는 LiteLLM을 그 해법으로 제시하되, 한 가지 오해를 먼저 바로잡습니다. LiteLLM은 추론(inference) 자체를 빠르게 만드는 도구가 아니라, 여러 모델 앞에서 요청을 표준화하고 분배하는 **게이트웨이**입니다. 실제 토큰을 생성하는 추론 가속은 vLLM 같은 추론 엔진 계층의 역할이며, LiteLLM은 그 앞단에서 라우팅·폴백·캐싱·비용 추적으로 서비스 수준을 끌어올립니다. 표준 구성은 경쟁이 아니라 조합입니다. 곧 `클라이언트 → LiteLLM(게이트웨이) → vLLM(추론 엔진, Gemma·Qwen 서빙)` 입니다.

이어지는 아홉 개 장에서는 도입이 필요한 이유부터 개발 배경, 라이선스, 기초 개념, 설치, 온프레미스 Gemma·Qwen 튜닝, 도입 전후 운영 비교, 유사 오픈소스와의 계층별 비교, 그리고 엔터프라이즈 적용 방안과 단계별 도입 로드맵까지 차례로 살펴봅니다.

1장: LiteLLM 도입이 필요한 핵심 이유

온프레미스에서 AI 플랫폼을 운영하기로 결정했다면, 가장 먼저 마주하는 질문은 "어떤 추론 엔진을 쓸 것인가"가 아니라 "여러 모델과 여러 애플리케이션을 어떻게 하나의 통제 지점으로 묶을 것인가"입니다. 결론부터 말씀드리면, 이 통제 지점을 별도로 두지 않은 조직은 예외 없이 세 가지 비용을 반복해서 지불하게 됩니다. 애플리케이션마다 흩어진 연동 코드를 유지보수하는 비용, 어느 팀이 얼마나 썼는지 알 수 없는 상태에서 발생하는 예산 사고, 그리고 키와 접근 권한이 분산되어 감사(audit)가 불가능해지는 위험입니다. LiteLLM은 이 세 가지를 게이트웨이(gateway) 계층에서 한 번에 정리합니다.

여기서 한 가지를 분명히 구분하고 시작하겠습니다. LiteLLM은 게이트웨이, 즉 프록시(proxy)입니다. 토큰을 실제로 생성하는 추론(inference)을 더 빠르게 만들어 주지는 않습니다. 실제 토큰 생성과 추론 가속은 vLLM, TGI, SGLang 같은 추론 엔진 계층의 역할입니다. LiteLLM이 제공하는 '가속'은 이런 종류가 아니라, 라우팅(routing)·로드밸런싱(load balancing)·폴백(fallback)·캐싱(caching)을 통해 서비스 전체의 처리량과 가용성, 안정성을 끌어올리는 것입니다. 다시 말해 LiteLLM과 vLLM은 서로 경쟁하는 대안이 아니라, 아래 그림처럼 위아래로 쌓아 올리는 조합입니다.



게이트웨이가 없는 직접 연동 구조와 LiteLLM을 통제 지점으로 두는 구조의 대비.

표준 구성은 다음과 같이 읽으시면 됩니다. 클라이언트가 요청을 보내면 LiteLLM(게이트웨이)이 이를 받아 인증·정책·라우팅을 처리하고, 실제 토큰 생성은 그 뒤에 놓인 vLLM 같은 추론 엔진(Gemma나 Qwen 모델을 서빙)이 담당합니다. 이 장에서는 이 조합이 왜 필요한지를, 게이트웨이가 없을 때 조직이 실제로 겪는 문제를 통해 먼저 설명하겠습니다. 라우팅 알고리즘이나 설정 방법 같은 기술 상세는 뒤 장으로 미루겠습니다.

1.1 게이트웨이 부재가 만드는 운영 리스크

게이트웨이 없이 애플리케이션이 모델 공급자에 직접 붙는 구조는 처음에는 가장 단순해 보입니다. 문제는 모델이 하나에서 둘로, 팀이 하나에서 여럿으로 늘어나는 순간부터 시작됩니다. 파편화된 연동과 보이지 않는 사용량이라는 두 가지 리스크가 동시에 자라나며, 이 둘은 시간이 지날수록 되돌리기 어려워집니다.

1.1.1 모델별 API 파편화 문제

모델 공급자마다 요청과 응답의 형식, 인증 방식, 파라미터 이름이 서로 다릅니다. OpenAI 계열과 Anthropic 계열, 그리고 사내에 직접 띄운 vLLM 엔드포인트는 같은 "텍스트를 생성한다"는 작업을 하면서도 요청 본문의 구조와 응답에서 결과를 꺼내는 경로가 제각각입니다. 게이트웨이가 없으면 이 차이를 흡수하는 코드가 애플리케이션마다 흩어져 들어갑니다. [S1][S2]

공급자별 형식 차이를 간단히 정리하면 다음과 같습니다.

구분	OpenAI 호환 API	Anthropic Messages API	사내 vLLM (OpenAI 호환 모드)
인증 헤더	`Authorization: Bearer`	`x-api-key`	자체 발급 키 또는 무인증
시스템 프롬프트	messages 배열 내 role	별도 `system` 필드	OpenAI 형식 그대로
응답에서 본문 위치	`choices[].message.content`	`content[].text`	`choices[].message.content`
토큰 사용량 필드	`usage`	`usage` (키 이름 상이)	`usage`
스트리밍 형식	SSE 청크	SSE 청크(스키마 상이)	SSE 청크

표에서 보이듯 차이는 사소해 보이지만, 각 차이마다 분기 코드가 필요합니다. 이 분기가 조직 차원에서 어떤 비용으로 환산되는지를 정리하면 문제의 크기가 분명해집니다.

- **중복 구현 비용**: 새 모델을 추가할 때마다 그 모델을 쓰는 모든 애플리케이션이 각자 연동 코드를 고쳐야 합니다. 애플리케이션이 다섯 개면 다섯 곳을 수정합니다.
- **회귀(regression) 위험**: 공급자가 응답 스키마를 바꾸면, 그 변화가 애플리케이션마다 흩어진 코드 전반을 동시에 깨뜨립니다. 어디가 깨졌는지 추적하는 데만 상당한 시간이 듭니다.
- **모델 교체의 경직성**: 더 저렴하거나 더 빠른 모델로 바꾸고 싶어도, 연동이 코드에 고정되어 있으면 교체 자체가 개발 과제가 됩니다. 실험 비용이 높아 최적화를 미루게 됩니다.
- **지식 분산**: 어떤 애플리케이션이 어떤 공급자의 어떤 방언(dialect)을 쓰는지가 팀별로 흩어져, 담당자가 바뀌면 파악에만 오랜 시간이 걸립니다.

LiteLLM은 이 파편화를 표준화된 단일 인터페이스로 흡수합니다. 애플리케이션은 하나의 형식(OpenAI 호환 형식)으로만 요청을 보내고, 게이트웨이가 그 요청을 각 공급자의 실제 형식으로 번역합니다. 새 모델을 붙이는 일은 애플리케이션 코드가 아니라 게이트웨이 설정의 문제가 되며, 모델 교체는 설정 한 줄을 바꾸는 일로 줄어듭니다. [S1][S2]

1.1.2 인증·비용 가시성 부족에 따른 영향

두 번째 리스크는 눈에 잘 보이지 않기 때문에 더 위험합니다. 게이트웨이가 없으면 모델 공급자 키가 애플리케이션마다 흩어져 저장되고, 팀별 사용량은 집계되지 않습니다. 이 상태에서는 두 가지가 동시에 무너집니다. 하나는 비용 통제이고, 다른 하나는 감사 추적(audit trail)입니다. [S4][S7]

키가 분산되면 어떤 일이 벌어지는지 구체적인 시나리오로 살펴보겠습니다.

> *한 팀이 프로토타입을 만들면서 코드에 넣어 둔 API 키가 실수로 반복 호출 루프에 걸렸다고 가정하겠습니다. 게이트웨이가 없다면 이 호출은 어떤 상한선도 거치지 않고 공급자에게 그대로 전달됩니다. 사용량은 팀별로 나뉘어 집계되지 않으므로, 문제가 청구서로 드러나기 전까지 아무도 알아채지 못합니다. 월말에야 예상의 몇 배에 달하는 비용이 확인되고, 그제*

> *야 어느 키가 원인이었는지 역추적을 시작합니다. 키가 흩어져 있으므로 이 역추적조차 오래 걸립니다.*

이 시나리오의 핵심은 단순히 돈을 더 썼다는 데 있지 않습니다. **사고가 진행되는 동안 그것을 멈출 지점이 어디에도 없었다**는 데 있습니다. 게이트웨이가 있으면 팀별·키별 예산 한도(budget)와 호출 한도(rate limit)를 사전에 설정해 두므로, 같은 사고가 한도에 닿는 순간 차단되고 즉시 알림으로 드러납니다.

게이트웨이 유무에 따라 비용 가시성과 통제 능력이 어떻게 달라지는지 대비하면 다음과 같습니다.

항목	게이트웨이 없음 (직접 연동)	LiteLLM 게이트웨이 도입
API 키 관리	애플리케이션마다 분산 저장	게이트웨이에서 중앙 발급·회수
팀별 사용량 집계	불가능하거나 수작업 취합	가상 키(virtual key) 단위로 자동 집계
예산 상한	없음(청구서로 사후 확인)	팀·키·모델별 사전 한도 설정
비용 초과 대응	사후 역추적	한도 도달 시 실시간 차단·알림
감사 추적	로그가 앱마다 흩어짐	요청 단위 중앙 로깅
접근 권한 회수	키를 쓴 코드를 일일이 수정	게이트웨이에서 키 하나 비활성화

표의 오른쪽 열이 보여 주는 것은 결국 하나의 원리입니다. 모든 요청이 반드시 한 지점을 통과하게 만들면, 그 지점에서 사용량을 측정하고 권한을 통제하며 기록을 남길 수 있다는 것입니다. LiteLLM은 가상 키라는 장치로 이를 구현합니다. 실제 공급자 키는 게이트웨이 안에만 두고, 각 팀과 애플리케이션에는 게이트웨이가 발급한 가상 키를 나눠 줍니다. 이 가상 키마다 예산과 호출 한도, 사용 가능한 모델 범위를 지정할 수 있으며, 문제가 생기면 해당 가상 키만 비활성화하면 됩니다. 실제 공급자 키는 그대로 보호됩니다. [S4][S7]

여기서 한 가지 오해를 미리 짚어 두겠습니다. LiteLLM의 코어는 MIT 라이선스로 공개되어 있어 자유롭게 사용할 수 있지만, 세밀한 접근 제어나 감사 기능 일부는 별도의 상용 조건이 적용되는 enterprise 기능에 속합니다. '전부 무료'라고 단정하기보다는, 도입 범위에 따라 코어로 충분한지 상용 기능이 필요한지를 나누어 검토하시는 편이 정확합니다. 구체적인 기능 경계는 뒤 장에서 다루겠습니다. [S4]

1.2 문제 정리와 다음 장 예고

지금까지 살펴본 두 리스크, 즉 공급자마다 다른 형식이 코드에 흩어지는 파편화와 키·사용량이 보이지 않는 통제 공백은 모두 하나의 원인에서 나옵니다. 요청이 반드시 지나가야 하는 단일 통제 지점이 없다는 것입니다. 참고로 LiteLLM 프로젝트의 최초 릴리스 연월과 같은 세부 이력은 이 문서에서 단정하지 않겠습니다. 정확한 값은 확인이 필요합니다.

LiteLLM은 클라이언트와 추론 엔진 사이에 게이트웨이 계층을 세워 이 두 문제를 동시에 정리합니다. 표준화된 단일 인터페이스로 형식 파편화를 흡수하고, 가상 키와 중앙 로깅으로 비용 가시성과 감사 추적을 되살립니다. 다음 장에서는 이 게이트웨이가 라우팅·로드밸런싱·폴백·캐싱을 통해 서비스 수준을 어떻게 끌어올리는지, 그리고 그 아래에서 vLLM 같은 추론 엔진과 어떻게 조합되는지를 구체적으로 살펴보겠습니다.

2장: LiteLLM의 등장 배경과 개발 주체

도구를 도입하기에 앞서 그 도구가 어디에서 왔고 무엇을 위해 만들어졌는지를 확인하는 일은 신뢰의 첫 단추입니다. 온프레미스 AI 플랫폼을 설계하는 담당자에게 게이트웨이는 모든 추론 요청이 지나가는 길목입니다. 그 길목에 놓일 소프트웨어가 어떤 문제의식에서 출발했는지 이해하면, 도구가 해결해 주는 범위와 해결해 주지 않는 범위를 명확히 구분할 수 있습니다. 이 장에서는 LiteLLM의 개발 주체와 최초 목적, 그리고 두 가지 제공 형태가 어떤 설계 관점에서 나왔는지를 살펴봅니다.

먼저 한 가지를 분명히 해 두겠습니다. LiteLLM은 게이트웨이(프록시)입니다. 모델의 추론 자체를 빠르게 만드는 도구가 아닙니다. 토큰을 실제로 생성하는 추론 가속은 vLLM, TGI, SGLang 같은 추론 엔진의 역할이며, 이들은 GPU 상에서 배치 처리와 메모리 관리를 최적화해 처리량과 지연 시간을 개선합니다 [S8]. LiteLLM이 말하는 '가속'은 성격이 다릅니다. 여러 백엔드로의 라우팅, 장애 시 폴백, 응답 캐싱을 통해 서비스 수준을 끌어올리는 것입니다. 표준 구성은 두 계층의 조합으로 이해하면 됩니다.

- 클라이언트 → LiteLLM(게이트웨이) → vLLM(추론 엔진)

게이트웨이 계층과 추론 엔진 계층은 각자 다른 문제를 풉니다. 이 구분을 유지한 채로 이후 논의를 읽어 주시기 바랍니다.

2.1 BerriAI와 오픈소스 게이트웨이의 출발

2.1.1 개발 주체와 최초 목적

LiteLLM은 BerriAI라는 조직이 주도하는 오픈소스 프로젝트이며, 소스 코드는 GitHub의 `BerriAI/litellm` 저장소에서 공개적으로 관리됩니다 [S1]. 즉 특정 폐쇄형 제품에 종속된 내부 도구가 아니라, 공개된 저장소에서 이슈와 기여를 받아 발전해 온 커뮤니티 기반 소프트웨어입니다. 온프레미스 환경을 운영하는 조직에게 소스 공개라는 사실은 그 자체로 검증 가능성을 뜻합니다. 게이트웨이가 요청을 어떻게 변환하고 어디로 보내는지를 코드 수준에서 직접 확인할 수 있기 때문입니다.

최초 릴리스 연월과 같은 구체적 시점 정보는 이 문서에서 단정하지 않겠습니다. 정확한 최초 공개 시기는 확인이 필요합니다. 다만 프로젝트의 지향점은 저장소와 문서를 통해 분명하게 드러납니다. LiteLLM이 풀고자 한 문제는 모델 공급자별 인터페이스의 파편화입니다.

AI 모델 시장에는 상용 API와 오픈 웨이트 모델, 그리고 자체 호스팅한 추론 엔진까지 서로 다른 호출 규약을 가진 공급자가 다수 존재합니다. 각 공급자는 요청 형식, 인증 방식, 응답 구조, 오류 코드 체계가 조금씩 다릅니다. 애플리케이션이 여러 모델을 함께 쓰려 하면, 공급자마다 별도의 연동 코드를 작성하고 유지해야 하는 부담이 생깁니다. LiteLLM은 이 파편화를 하나의 형식으로 흡수하려는 목적에서 출발했습니다. 100개가 넘는 공급자를 OpenAI 호환 형식이라는 단일 규약으로 부를 수 있게 만드는 것이 핵심 지향점입니다 [S1][S2].

여기서 도구의 성격이 드러납니다. LiteLLM의 목표는 표준화입니다. 새로운 성능을 창출하는 것이 아니라, 이미 존재하는 다양한 모델을 일관된 방식으로 부를 수 있게 하는 데 초점이 있습니다. 지원 공급자의 범위를 정리하면 다음과 같습니다.

구분	예시 성격	LiteLLM이 흡수하는 차이
상용 API 모델	외부 SaaS 형태로 제공되는 대형 모델	인증 헤더, 요청 파라미터, 응답 스키마
오픈 웨이트 모델	자체 배포 가능한 공개 모델	엔드포인트 규약, 파라미터 명칭
자체 호스팅 추론 엔진	vLLM 등 사내 GPU 위에서 구동	접속 경로, 모델 식별자 매핑

표에서 보듯 LiteLLM이 하는 일은 서로 다른 백엔드의 차이를 게이트웨이 안쪽으로 감추는 것입니다. 애플리케이션은 언제나 같은 형식으로 요청을 보내고, 실제 어떤 공급자로 그 요청이 향하는지는 게이트웨이가 결정합니다. 자체 호스팅한 vLLM 추론 엔진 역시 이 100개 이상의 공급자 목록 안에 포함되므로, 온프레미스 조직은 상용 API와 사내 GPU 모델을 동일한 인터페이스로 나란히 다룰 수 있습니다 [S2][S8]. 이 점이 온프레미스 플랫폼 설계에서 특히 의미가 큽니다. 외부 모델로 시작해 점진적으로 사내 모델로 전환하거나 둘을 병행할 때, 애플리케이션 코드를 다시 쓰지 않아도 되기 때문입니다.

2.1.2 단일 표준 인터페이스라는 설계 관점

LiteLLM이 파편화를 흡수하는 방식은 하나의 인터페이스로 모든 모델을 감싸는 것입니다. 그런데 이 인터페이스를 어떤 형태로 도입하느냐에는 선택지가 있습니다. LiteLLM은 크게 두 가지 형태로 제공됩니다 [S2].

첫째는 SDK, 즉 라이브러리 형태입니다. 애플리케이션 코드 안에 라이브러리를 직접 포함해 호출하는 방식입니다. 함수 호출 한 번으로 여러 공급자를 부를 수 있고, 별도의 서버를 띄우지 않아도 됩니다. 프로토타이핑이나 단일 애플리케이션 안에서 모델 호출을 통일하려는 경우에 적합합니다.

둘째는 프록시 서버, 즉 게이트웨이 형태입니다. 독립된 서버 프로세스로 LiteLLM을 띄우고, 여러 애플리케이션이 이 서버를 공통의 관문으로 삼아 요청을 보내는 방식입니다. 이 형태에서는 인증 키 관리, 사용량 집계, 라우팅 정책, 폴백 규칙을 한곳에서 중앙 집중적으로 운영할 수 있습니다. 여러 팀과 여러 서비스가 같은 모델 자원을 공유하는 온프레미스 플랫폼이라면 이 게이트웨이 형태가 자연스러운 선택입니다.

두 형태의 차이를 한 줄로 비유하면 이렇게 정리할 수 있습니다. SDK는 각자의 주방에 들여놓는 조리 도구이고, 프록시 서버는 모두가 함께 쓰는 공용 급식소의 배식 창구입니다. 앞의 것은 애플리케이션마다 개별적으로 갖추는 도구이고, 뒤의 것은 조직 전체가 하나의 창구를 공유하는 구조입니다.

두 형태를 비교하면 다음과 같습니다.

항목	SDK (라이브러리)	프록시 서버 (게이트웨이)
배포 형태	애플리케이션 코드에 내장	독립 서버 프로세스
적용 범위	단일 애플리케이션	여러 애플리케이션 공통
키·인증 관리	애플리케이션이 각자 보유	게이트웨이에서 중앙 관리
사용량·정책 통제	개별적, 분산	중앙 집중
대표 적용 상황	프로토타입, 개별 서비스	사내 공용 AI 플랫폼
운영 부담	애플리케이션마다 반복	한곳에서 일괄 운영

두 형태가 채택한 공통의 요청 규약은 OpenAI 형식입니다 [S2]. LiteLLM이 다른 규약이 아닌 OpenAI 형식을 표준으로 삼은 데에는 실용적인 이유가 있습니다. 이미 많은 애플리케이션, 프레임워크, 도구가 OpenAI 형식의 요청과 응답을 전제로 개발되어 있어, 이 형식을 공통 인터페이스로 채택하면 기존 코드베이스와의 호환성이 넓게 확보됩니다. 개발자가 새로운 규약을 학습할 필요 없이 익숙한 형식 그대로 요청을 보내면, 게이트웨이가 뒷단의 실제 공급자 규약으로 변환해 줍니다. 표준을 새로 만들기보다 사실상의 관행으로 자리 잡은 형식을 채택함으로써 도입 장벽을 낮춘 셈입니다.

라이선스 구조도 이 시점에 짚어 둘 필요가 있습니다. LiteLLM의 코어는 MIT 라이선스로 공개되어 있어, 온프레미스 환경에서 자유롭게 도입하고 수정할 수 있습니다. 다만 저장소의 `enterprise/` 영역에 해당하는 기능은 별도의 상용 라이선스가 적용됩니다. 코어와 상용 기능의 경계를 도입 초기에 확인해 두면, 이후 라이선스 조건과 관련한 오해를 피할 수 있습니다.

정리하면, LiteLLM은 BerriAI가 모델별 인터페이스의 파편화를 풀기 위해 만든 오픈소스 게이트웨이이며, 100개 이상의 공급자를 OpenAI 호환 형식이라는 단일 규약으로 부를 수 있게 하는 것을 최초 목적으로 삼습니다. SDK와 프록시 서버라는 두 형태는 같은 표준 인터페이스를 서로 다른 규모의 도입 상황에 맞추어 제공하는 선택지입니다. 사내 여러 서비스가 모델 자원을 공유하는 온프레미스 플랫폼이라면 프록시 서버 형태가 중앙 통제와 운영 효율 측면에서 유리합니다. 그리고 이 모든 표준화 작업은 게이트웨이 계층에서 이루어지며, 실제 토큰을 생성하는 추론 가속은 그 뒤의 vLLM 같은 추론 엔진이 담당한다는 계층 구분은 이 백서 전반에 걸쳐 일관되게 유지됩니다 [S8].

3장: 라이선스 구조와 기업 도입 검토 사항

LiteLLM은 오픈소스 게이트웨이입니다. 그러나 오픈소스라는 표현이 곧 모든 기능을 무상으로 사용할 수 있다는 뜻은 아닙니다. LiteLLM의 코어 코드는 MIT 라이선스로 자유롭게 제공되지만, 기업 환경에서 필요로 하는 일부 운영 기능은 별도의 상용 라이선스 아래 제공됩니다. 이 두 계층을 구분하지 못하면 도입 계획 단계에서 비용과 규정 준수 측면의 오판이 생길 수 있습니다. 이 장은 그 경계를 명확히 정리하여, 무상으로 시작할 수 있는 범위와 유료 전환을 검토해야 하는 시점을 판단하는 데 도움을 드리는 것을 목표로 합니다.

앞선 장에서 살펴본 표준 구성, 즉 클라이언트에서 LiteLLM 게이트웨이를 거쳐 vLLM과 같은 추론 엔진으로 이어지는 조합은 대부분 MIT 코어만으로도 구축할 수 있습니다. 여기서 LiteLLM은 요청을 중계하고 표준화하는 게이트웨이 역할을 담당하며, 실제 추론 가속은 vLLM 등 추론 엔진 계층이 수행합니다. 두 계층의 역할이 다른 만큼, 라이선스 검토 역시 게이트웨이 계층에 한정해 진행하면 됩니다.

3.1 MIT 코어와 상용 엔터프라이즈 경계

LiteLLM은 크게 두 부분으로 나누어 이해하는 편이 정확합니다. 하나는 MIT 라이선스로 배포되는 코어이며, 다른 하나는 별도의 상용 라이선스가 적용되는 엔터프라이즈 기능입니다. 코어는 모델 호출을 표준화하고 여러 공급자를 하나의 인터페이스로 통합하는 게이트웨이의 본질적 기능을 담고 있습니다. 엔터프라이즈 기능은 조직 규모가 커지고 여러 팀이 동일한 게이트웨이를 공유하기 시작할 때 필요해지는 통제 및 감사 기능을 담고 있습니다.

이 경계를 사전에 이해해 두면, 초기에는 무상 코어로 검증을 진행하다가 조직의 필요가 명확해지는 시점에 상용 기능 도입을 검토하는 단계적 접근이 가능합니다.

3.1.1 MIT 라이선스 적용 범위

LiteLLM 코어 코드는 MIT 라이선스로 제공됩니다 [S3]. MIT 라이선스는 오픈소스 라이선스 가운데 가장 제약이 적은 축에 속합니다. 소스 코드를 자유롭게 사용, 복제, 수정, 병합, 배포할 수 있으며, 상업적 목적의 사용과 재배포도 허용됩니다. 사내 환경에서 코드를 수정하여 자체적으로 운영하는 것도 라이선스상 제한이 없습니다.

MIT 라이선스가 요구하는 핵심 조건은 저작권 고지와 라이선스 전문을 배포물에 포함하는 것입니다. 이는 소프트웨어를 재배포하거나 파생물을 외부에 제공할 때 적용되는 조건이며, 조직 내부에서만 운영하는 경우에는 부담이 크지 않습니다. 또한 MIT 라이선스는 소프트웨어를 있는 그대로(as-is) 제공하며 별도의 보증을 하지 않는다는 점을 명시하고 있습니다.

무상으로 시작할 수 있는 범위를 정리하면 다음과 같습니다.

구분	MIT 코어로 가능한 범위
게이트웨이 기본 운영	여러 모델 공급자를 단일 인터페이스로 통합하여 호출

구분	MIT 코어로 가능한 범위
표준 구성 연동	클라이언트에서 LiteLLM을 거쳐 vLLM 등 추론 엔진으로 연결
코드 수정	사내 요구에 맞춘 소스 코드 수정 및 자체 빌드
상업적 사용	사내 서비스 및 상업적 목적의 활용
배포	저작권 고지와 라이선스 전문 포함 시 재배포

MIT 라이선스는 이처럼 넓은 자유를 보장하지만, 라이선스 문구의 정확한 해석과 조직 정책과의 정합성은 법무 검토가 필요한 영역입니다. 특히 파생물을 외부에 배포하는 시나리오나, 사내 규정상 오픈소스 사용 승인 절차가 있는 조직이라면 라이선스 원문을 근거로 검토를 진행하시기를 권합니다. 라이선스 원문은 코어 저장소의 LICENSE 파일에서 확인할 수 있으며, 본 백서의 서술은 도입 판단을 돕기 위한 요약일 뿐 법률 자문을 대신하지 않습니다 [S3].

정리하면, LiteLLM 게이트웨이의 본질적 기능은 MIT 코어만으로 충분히 구축하고 운영할 수 있으며, 초기 검증과 소규모 운영 단계에서는 별도 비용 없이 시작할 수 있습니다.

3.1.2 엔터프라이즈 기능 도입 검토 포인트

조직이 게이트웨이를 여러 팀과 다수의 사용자에게 개방하기 시작하면, 코어만으로는 충족하기 어려운 운영 요구가 나타납니다. 통합 인증, 접근 이력 추적, 세분화된 권한 통제와 같은 요구입니다. LiteLLM은 이러한 기능을 엔터프라이즈 기능으로 별도 제공하며, 이 기능들은 MIT가 아닌 상용 라이선스 아래에 있습니다 [S4]. 관련 자료에서는 대략 100명 이상의 사용자 또는 10개 이상의 유스케이스 규모에서 이러한 기능의 필요성이 두드러진다고 안내하고 있습니다 [S4].

무상으로 제공되는 코어 기능과 상용 라이선스가 적용되는 엔터프라이즈 기능의 구분은 다음과 같습니다.

기능 영역	MIT 코어	엔터프라이즈(상용)
모델 통합 및 호출 중계	제공	—
표준 구성 연동(vLLM 등)	제공	—
SSO(통합 인증 연동)	—	상용 라이선스 [S4]
감사 로그(접근 이력 추적)	—	상용 라이선스 [S4]
세분화된 접근제어	—	상용 라이선스 [S4]
전문 기술 지원	—	상용 라이선스 [S4]

이 표에서 보듯 게이트웨이의 기본 동작은 코어에 속하고, 조직 차원의 통제와 규정 준수를 위한 기능은 엔터프라이즈 계층에 속합니다. 도입을 결정하기 전에 다음 항목을 확인 목록으로 활용하시면 유료 전환 시점을 판단하는 데 도움이 됩니다.

- 게이트웨이를 사용하는 사용자 수가 여러 팀에 걸쳐 늘고 있는지, 관련 자료가 제시한 규모(사용자 100명 이상, 유스케이스 10개 이상)에 근접하는지 [S4]
- 사내 계정 체계와의 통합 인증(SSO) 연동이 보안 정책상 필수인지
- 모델 호출에 대한 감사 로그가 규정 준수나 내부 통제 요건으로 요구되는지
- 팀·프로젝트·모델 단위의 세분화된 접근제어가 필요한지
- 운영 안정성 확보를 위해 공급사의 전문 기술 지원 계약이 필요한지
- 조직의 예산 주기와 라이선스 조달 절차가 도입 일정과 맞물리는지

LiteLLM의 MIT 코어와 상용 엔터프라이즈 기능의 경계.

엔터프라이즈 기능의 구체적인 가격, 계약 조건, 지원 범위는 시점과 조직 규모에 따라 달라질 수 있으므로, 최신 조건은 공급사를 통한 별도 확인이 필요합니다. 본 백서에서는 어떤 기능이 유료 경계에 위치하는지를 밝히는 데 초점을 두었으며, 정확한 상용 조건은 도입 검토 단계에서 별도로 확인하시기를 권합니다.

정리하면, 조직은 MIT 코어로 게이트웨이를 도입하여 검증과 초기 운영을 진행하고, 사용자와 유스케이스가 확대되어 통합 인증·감사 로그·세분화된 접근제어가 필요해지는 시점에 엔터프라이즈 기능 도입을 검토하는 방식이 합리적입니다. 이 경계를 미리 파악해 두는 것이 '전부 무료'라는 오해를 피하고 도입 계획의 정확도를 높이는 출발점입니다.

4장: LLM 게이트웨이 기초 개념 정리

이 장은 이후의 설치·운영·튜닝 장을 읽기 전에 반드시 갖춰야 할 최소한의 개념을 정리합니다. 특히 비전문 실무자도 어려움 없이 따라올 수 있도록, 각 용어를 일상의 익숙한 장면에 먼저 비유해 그린 뒤 정확한 정의를 덧붙이는 방식으로 설명합니다. 이 백서 전체에서 가장 자주 되풀이되는 구분이 하나 있습니다. 바로 **실제로 답변을 만들어 내는 계층**과 **그 요청을 받아 전달하고 관리하는 계층**의 차이입니다. 이 두 계층을 헷갈리면 이후 장의 설명이 뒤엉키기 쉬우므로, 이 장에서 그 경계를 분명하게 세워 두겠습니다. [S8][S12]

4.1 핵심 용어와 계층 구분

기술 문서를 처음 접하는 분들이 가장 먼저 부딪히는 어려움은 "LLM", "게이트웨이", "추론 엔진", "프록시" 같은 단어가 비슷비슷하게 들린다는 점입니다. 그러나 이 단어들은 서로 다른 일을 맡는 별개의 부품을 가리킵니다. 이 절에서는 그 부품들이 어떻게 나뉘어 있고, 각자 무슨 일을 하는지를 하나의 그림으로 묶어서 보여 드리겠습니다.

4.1.1 추론 엔진과 게이트웨이의 역할 구분

레스토랑을 떠올려 보시기 바랍니다. 손님이 식당에 들어오면 가장 먼저 마주치는 곳은 안내데스크입니다. 안내데스크 직원은 손님을 맞이하고, 예약을 확인하고, 주문을 받아 적어 주방으로 넘깁니다. 그러나 안내데스크 직원이 직접 요리를 하지는 않습니다. 실제로 재료를 다듬고 불을 켜서 음식을 만들어 내는 곳은 주방입니다.

LLM을 다루는 시스템도 이와 똑같은 구조를 가집니다. 여기서 **주방에 해당하는 것이 추론 엔진**이고, **안내데스크에 해당하는 것이 게이트웨이**입니다. 추론 엔진은 실제로 문장을 한 조각씩 만들어 내는 무거운 계산을 담당하고, 게이트웨이는 그 앞에서 요청을 받아 정리하고 알맞은 주방으로 넘겨 주는 역할을 담당합니다. 두 계층은 하는 일이 전혀 다르며, 서로를 대체하지 않습니다. 이 구분은 이 백서 전체에서 반복해서 등장하므로 지금 확실히 익혀 두시기 바랍니다. [S8]

조금 더 정확하게 정의하면 다음과 같습니다. **추론(inference)** 이란 이미 학습이 끝난 인공지능 모델이 입력을 받아 그에 대한 출력을 실제로 계산해 내는 과정을 말합니다. LLM의 경우 이 출력은 문장을 이루는 작은 조각들이 하나씩 이어져 만들어집니다. 이 계산을 효율적으로 수행하도록 만들어진 전용 소프트웨어를 **추론 엔진(inference engine)** 이라고 부르며, 대표적인 예가 vLLM입니다. vLLM은 Gemma나 Qwen 같은 실제 모델을 메모리에 올려 놓고, 요청이 들어오면 그 모델을 돌려서 답변 토큰을 생성합니다. [S8]

반면 **게이트웨이(gateway)** 는 여러 추론 엔진 앞에 서서, 들어오는 요청을 받아 인증·기록·전달을 처리하고 알맞은 엔진으로 넘겨 주는 중개 계층입니다. LiteLLM이 바로 이 게이트웨이 역할을 맡습니다. LiteLLM 자체는 문장을 만들어 내지 않습니다. 다시 강조하면, **토큰을 생성하는 무거운 계산은 vLLM 같은 추론 엔진의 몫이고, LiteLLM은 그 계산을 직접 하지 않는 중개자**입니다. 이 점을 혼동하면 "LiteLLM이 느리다"거나 "LiteLLM이 답을 만든다"는 식의 잘못된 진단으로 이어지므로 주의가 필요합니다. [S8][S12]

다음 표는 두 계층의 역할을 나란히 놓고 비교한 것입니다.

구분	추론 엔진 (예: vLLM)	게이트웨이 (예: LiteLLM)
비유	음식을 직접 만드는 주방	손님을 맞는 안내데스크
핵심 역할	모델을 돌려 답변 토큰을 생성	요청을 받아 알맞은 엔진으로 중개
주요 부담	GPU 등 무거운 계산 자원	요청 정리·전달의 가벼운 처리
모델 보유 여부	실제 모델(Gemma·Qwen 등)을 적재	모델을 직접 적재하지 않음
담당 업무 예	토큰 생성, 배치 처리, 메모리 관리	인증, 사용량 기록, 요청 라우팅
없으면 생기는 문제	답변 자체를 만들 수 없음	여러 엔진을 하나로 묶어 쓰기 어려움

표에서 보듯 두 계층은 서로의 빈자리를 채워 주는 관계입니다. 주방만 있고 안내데스크가 없으면 손님이 어느 창구로 주문해야 할지 알 수 없고, 안내데스크만 있고 주방이 없으면 아무리 주문을 받아도 음식이 나오지 않습니다. 마찬가지로 추론 엔진만으로는 여러 모델을 하나의 창구로 묶어 관리하기 어렵고, 게이트웨이만으로는 답변을 만들어 낼 수 없습니다. 그래서 실제 현장에서는 이 둘을 함께 조합해 사용합니다. 표준적인 구성은 다음과 같은 흐름을 가집니다.

클라이언트 → LiteLLM(게이트웨이) → vLLM(추론 엔진, Gemma·Qwen)

여기서 클라이언트란 답변을 요청하는 쪽, 즉 사용자의 애플리케이션이나 서비스를 말합니다. 요청은 먼저 게이트웨이인 LiteLLM에 도착하고, LiteLLM은 그 요청을 정리해 뒤편의 추론 엔진 vLLM으로 넘깁니다. vLLM은 Gemma나 Qwen 같은 모델을 돌려 답변을 만들어 다시 게이트웨이로 돌려주고, 게이트웨이는 그 결과를 클라이언트에게 전달합니다. 이 좌우 흐름을 한 장의 그림으로 익혀 두면 이후 장의 설명이 훨씬 수월해집니다. [S8][S12]

클라이언트에서 시작해 LiteLLM 게이트웨이를 거쳐 vLLM 추론 엔진에 이르는 좌우 흐름. 게이트웨이는 안내데스크, 추론 엔진은 주방에 해당합니다.

4.1.2 OpenAI 호환 인터페이스와 토큰 개념

앞 항에서 요청이 게이트웨이를 거쳐 추론 엔진으로 흘러간다고 설명했습니다. 그렇다면 그 "요청"은 대체 어떤 모습일까요. 여기서 등장하는 개념이 **표준 요청 형식**과 **토큰**입니다. 이 두 가지는 이후 설치 장과 튜닝 장을 이해하는 데 꼭 필요한 최소 지식이므로 차근차근 살펴보겠습니다.

먼저 표준 요청 형식을 주문서에 비유해 보겠습니다. 여러 식당을 하나의 배달 앱으로 주문한다고 상상해 보시기 바랍니다. 식당마다 메뉴판 양식이 제각각이라면 손님은 매번 새 양식을 익혀야 하고 배달 앱도 복잡해집니다. 그런데 만약 모든 식당이 똑같은 양식의 주문서를 쓴다면, 손님은 한 가지 양식만 익히면 어느 식당에든 주문할 수 있습니다. 소프트웨어의 세계에도 이런 공통 주문서가 있습니다. 그것이 바로 **OpenAI 호환 인터페이스(OpenAI-compatible interface)** 입니다. [S2]

OpenAI 호환 인터페이스란, OpenAI가 널리 쓰이도록 정한 요청·응답 양식을 그대로 따르는 방식을 말합니다. 이 양식은 사실상의 공통 표준처럼 자리 잡아서, 많은 도구와 서비스가 이 형식으로 요청을 주고받습니다. LiteLLM은 이 표준 주문서를 받아들이는 창구 역할을 합니다. 즉, 애플리케이션이 이 공통 양식 하나로 요청을 보내기만 하면, LiteLLM이 그 뒤에 있는 서로 다른 추론 엔진이나 모델로 알맞게 전달해 줍니다. 덕분에 사용하는 모델이 Gemma에서 Qwen으로 바뀌더라도 클라이언트가 주문서 양식을 새로 배울 필요가 없습니다. 이것이 표준 형식이 주는 가장 큰 실용적 이점입니다. [S2]

이제 토큰을 살펴보겠습니다. 흔히 "글자 수만큼 비용이 든다"고 짐작하기 쉽지만, LLM이 실제로 세는 단위는 글자가 아니라 **토큰(token)** 입니다. 토큰은 문장을 잘게 나눈 조각이라고 이해하시면 됩니다. 긴 단어는 여러 조각으로 나뉘기도 하고, 짧고 흔한 단어는 하나의 조각이 되기도 합니다. 레고 블록으로 모형을 조립하는 장면을 떠올려 보시기 바랍니다. 완성된 모형(문장)은 여러 개의 블록(토큰)이 이어져 만들어지며, 사용한 블록의 개수만큼 재료가 든 셈이 됩니다. LLM도 이와 같아서, 입력으로 받은 토큰과 출력으로 만들어 낸 토큰의 개수를 합해 사용량을 계산합니다. 많은 서비스가 이 토큰 개수를 기준으로 과금하므로, 토큰은 곧 사용량과 비용을 재는 단위이기도 합니다.

앞 항에서 다룬 추론 엔진의 역할을 여기에 겹쳐 보면 그림이 더 선명해집니다. 추론 엔진이 "답변 토큰을 하나씩 생성한다"고 했던 그 토큰이 바로 지금 설명한 토큰입니다. 즉 vLLM 같은 추론 엔진은 이 조각들을 하나하나 이어 붙여 문장을 완성하고, 게이트웨이인 LiteLLM은 그 과정에서 오간 토큰의 개수를 기록해 사용량 관리에 활용할 수 있습니다. 이렇게 표준 요청 형식과 토큰이라는 두 개념이 맞물려 게이트웨이와 추론 엔진의 협업을 이룹니다.

다음 표는 이 장에서 다룬 핵심 용어를 한자리에 모아 정의한 것입니다. 이후 장을 읽다가 용어가 헷갈릴 때 되돌아와 확인하시기 바랍니다.

용어	한 줄 정의	이 백서에서의 예
LLM	방대한 문장을 학습해 사람처럼 글을 이어 쓰는 대규모 언어 모델	Gemma, Qwen

용어	한 줄 정의	이 백서에서의 예
추론	학습이 끝난 모델이 입력을 받아 실제로 답을 계산해 내는 과정	질문을 받아 답변 토큰을 만들어 냄
추론 엔진	추론 계산을 효율적으로 수행하는 전용 소프트웨어(주방에 비유)	vLLM
게이트웨이	요청을 받아 알맞은 추론 엔진으로 중개하는 계층(안내데스크에 비유)	LiteLLM
프록시	클라이언트와 엔진 사이에서 요청을 대신 받아 전달하는 중간 서버	LiteLLM Proxy
토큰	문장을 잘게 나눈 조각으로, 사용량과 비용을 재는 단위	입력·출력 토큰의 합계
OpenAI 호환 인터페이스	널리 쓰이는 공통 요청·응답 양식(표준 주문서에 비유)	LiteLLM이 받아들이는 요청 형식

표에서 **프록시(proxy)** 라는 용어가 새로 등장했습니다. 프록시란 클라이언트와 실제 처리 주체 사이에 서서 요청을 대신 받아 전달하는 중간 서버를 뜻합니다. LiteLLM은 이 프록시 형태로 동작하며, 그래서 문서에서는 흔히 LiteLLM Proxy라고 부릅니다. 게이트웨이와 프록시는 맥락에 따라 거의 같은 대상을 가리키는 말로 쓰이는데, 게이트웨이가 "여러 엔진을 하나의 창구로 묶어 관리한다"는 역할에 무게를 둔 표현이라면, 프록시는 "요청을 대신 받아 전달하는 중간 서버"라는 형태에 무게를 둔 표현입니다. 두 용어의 이러한 미묘한 강조 차이가 문맥에 따라 어떻게 달리 쓰이는지는 이후 운영 장에서 더 구체적으로 다룰 예정입니다. 이 백서에서는 두 용어가 함께 등장하더라도 같은 LiteLLM 계층을 가리킨다고 이해하시면 됩니다. 이러한 세부 표기 관행이 배포 환경마다 조금씩 달라질 수 있다는 점은 확인이 필요합니다. [S2][S12]

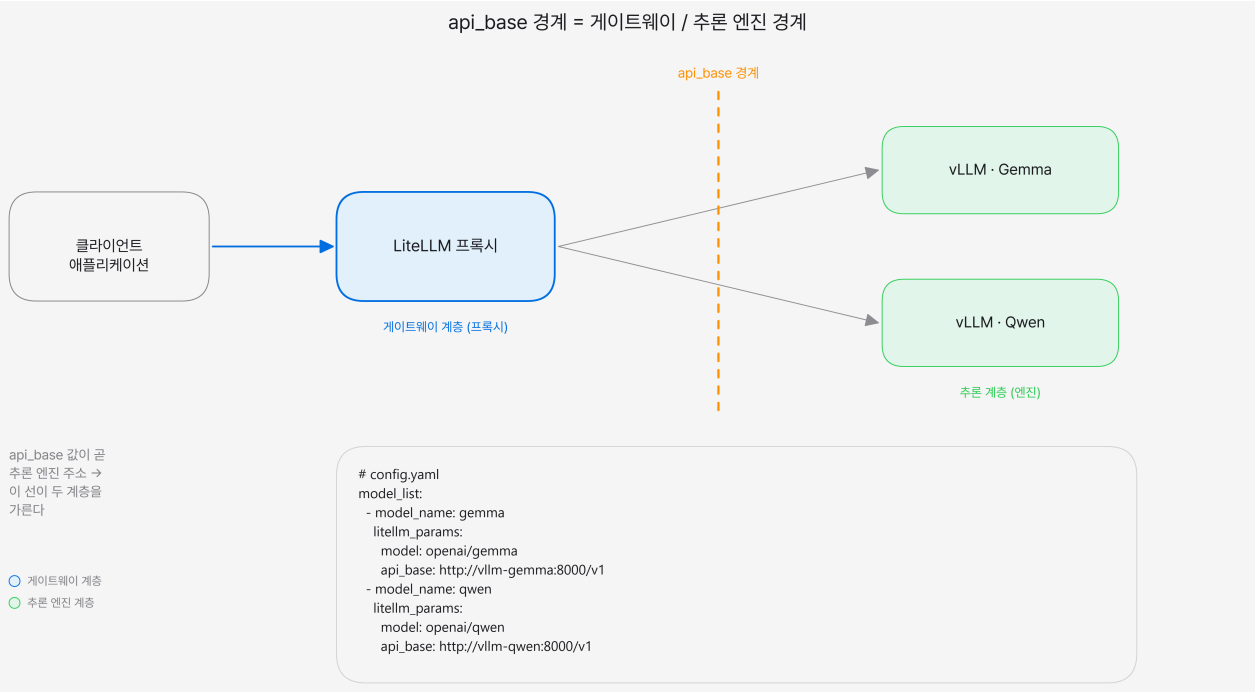
이로써 이후 장을 읽는 데 필요한 최소한의 지도가 그려졌습니다. 요청은 표준 주문서(OpenAI 호환 형식)에 담겨 안내데스크(게이트웨이 LiteLLM)에 도착하고, 안내데스크는 이를 주방(추론 엔진 vLLM)으로 넘기며, 주방은 레고 블록(토큰)을 이어 붙여 답을 만들어 다시 돌려줍니다. 이 그림 하나만 머릿속에 두어도 다음 장의 설치와 설정 과정이 한결 명료하게 다가올 것입니다.

5장: LiteLLM 설치와 초기 구성

앞 장에서 게이트웨이와 추론 엔진이 서로 다른 계층이라는 점을 정리했습니다. 이 장에서는 그 구분을 실제 구성 파일과 배포 명령으로 옮겨 봅니다. 목표는 분명합니다. 엔지니어가 가능한 한 적은 요소로 LiteLLM 프록시를 띄우고, 온프레미스에 이미 떠 있는 추론 엔진을 하나의 OpenAI 호환 엔드포인트로 묶는 지점까지 도달하는 것입니다.

한 가지를 먼저 짚어 둡니다. LiteLLM은 게이트웨이(프록시)이며, 그 자체가 토큰을 만들어 내지 않습니다. 실제 추론 연산은 뒤쪽 추론 엔진, 즉 vLLM 위에 올라간 Gemma나 Qwen 같은 모델이 담당합니다. 따라서 이 장에서 다루는 컨테이너는 CPU만으로도 가볍게 기동됩니다. GPU가 필요한 쪽은 게이트웨이가 아니라 그 뒤에 있는 추론 엔진입니다. 이 계층 구분은 설치 단계의 자원 산정에서부터 그대로 드러납니다.

전체 흐름은 다음 한 줄로 요약됩니다. 클라이언트 애플리케이션이 LiteLLM 프록시를 호출하고, 프록시는 `config.yaml`에 등록된 `api_base` 주소로 요청을 넘기며, 그 주소 뒤에서 vLLM이 실제 토큰을 생성합니다.



클라이언트에서 LiteLLM 프록시를 거쳐 vLLM 추론 엔진(Gemma·Qwen)에 도달하는 표준 구성. 프록시 계층과 추론 계층을 색으로 구분해 표시.

5.1 컨테이너 기반 최소 설치

이 절에서는 부가 요소를 모두 걷어 낸 최소 구성으로 프록시를 기동합니다. 먼저 단일 컨테이너와 설정 파일 하나로 시작하고, 비용 추적이나 캐싱이 필요해지는 시점에 저장소 계층을 선택적으로 덧붙이는 순서로 설명합니다. 이렇게 나누어 접근하면 초기 도입 단계에서 필요 이상으로 무거운 구성을 세우는 일을 피할 수 있습니다.

5.1.1 Docker Compose 배포 구성

가장 단순한 배포는 프록시 컨테이너 하나와 설정 파일 하나로 이루어집니다. LiteLLM은 공식 컨테이너 이미지를 제공하며, 프록시는 기동 시점에 마운트된 `config.yaml`을 읽어 어떤 모델을 어떤 주소로 중개할지 결정합니다 [S7].

아래는 부가 저장소 없이 프록시만 띄우는 Docker Compose 최소 구성입니다.

```
 # docker-compose.yaml — 최소 구성 (프록시 단독)
services:
  litellm:
    image: ghcr.io/berriai/litellm:main-latest
    ports:
      - "4000:4000"
    volumes:
      # 프록시가 기동 시 읽는 설정 파일을 컨테이너 안으로 전달합니다.
      - ./config.yaml:/app/config.yaml
    command: ["--config", "/app/config.yaml", "--port", "4000"]
    environment:
      # 관리용 마스터 키. 클라이언트 인증의 기준값입니다.
      LITELLM_MASTER_KEY: "sk-your-master-key"
    restart: unless-stopped
```

이 구성이 하는 일은 명확합니다. 4000번 포트로 OpenAI 호환 요청을 받아, `config.yaml`에 적힌 규칙대로 뒤쪽 추론 엔진에 전달하는 것입니다. 컨테이너 자체는 요청을 받아 넘기는 중개 역할만 하므로 연산 부하가 낮고, 별도의 GPU 없이 일반적인 서버 자원으로 기동됩니다. 무거운 GPU 자원은 이 컨테이너가 아니라 `config.yaml`의 `api_base`가 가리키는 추론 엔진 쪽에서 요구됩니다.

기동은 설정 파일을 같은 디렉터리에 둔 상태에서 다음 명령으로 수행합니다.

```
docker compose up -d
```

정리하면, 최소 설치의 요건은 컨테이너 런타임, 프록시 이미지, 그리고 모델 등록 규칙을 담은 `config.yaml` 세 가지입니다. 이 세 요소만으로 게이트웨이는 동작 상태에 진입합니다. 도입 초기의 부담이 낮다는 점은 이처럼 실제 구성에서 그대로 확인됩니다. `config.yaml`의 내용은 5.2에서 구체적으로 다룹니다.

5.1.2 PostgreSQL·Redis 연동 옵션

프록시가 동작을 시작한 뒤, 운영 규모가 커지면 두 가지 요구가 생깁니다. 하나는 팀별·키별 사용량과 비용을 지속적으로 기록해 두는 일이고, 다른 하나는 반복되는 요청의 응답을 재사용해 뒤쪽 추론 엔진의 부하를 줄이는 일입니다. 앞쪽은 PostgreSQL이, 뒤쪽은 Redis가 맡습니다. 중요한 점은 이 둘이 프록시 기동의 필수 조건이 아니라는 것입니다. 5.1.1의 최소 구성은 두 저장소 없이도 완결됩니다.

아래 표는 각 구성 요소가 담당하는 역할과 필요·선택 여부를 정리한 것입니다.

구성 요소	역할	필요 여부	붙이는 시점
LiteLLM 프록시 컨테이너	요청 수신과 라우팅, 폴백 중개	필수	처음부터
`config.yaml` 설정 파일	모델 등록과 라우팅·폴백 규칙 정의	필수	처음부터
PostgreSQL	사용량·비용 추적, 가상 키와 예산 정보 저장	선택	비용 가시성·다중 팀 관리가 필요할 때
Redis	응답 캐싱, 여러 프록시 인스턴스 간 레이트리밋 상태 공유	선택	캐싱 또는 프록시 다중화가 필요할 때 [S5]

표에서 보듯 PostgreSQL과 Redis는 운영 성숙도가 올라간 뒤에 붙이는 요소입니다. 특히 Redis는 여러 프록시 인스턴스를 두고 요청당 처리량(rpm) 제한을 인스턴스 사이에서 일관되게 적용하려 할 때 필요해집니다 [S5]. 단일 인스턴스로 시작하는 단계에서는 두 저장소 모두 뒤로 미룰 수 있습니다.

권장하는 순서는 다음과 같습니다. 먼저 프록시만으로 라우팅과 폴백이 의도대로 동작하는지 확인하고, 비용 가시성이 필요해지는 시점에 PostgreSQL을 연결하며, 캐싱이나 프록시 다중화가 논의되는 시점에 Redis를 추가합니다 [S7]. 이렇게 순차적으로 붙이면 초기 단계에서 운영하지 않을 저장소까지 미리 세워 두는 과설계를 피할 수 있습니다.

5.2 config.yaml 모델 등록

프록시가 실제로 어떤 모델을 부를지는 전적으로 `config.yaml`이 결정합니다. 이 절에서는 온프레미스 Gemma·Qwen을 OpenAI 호환 엔드포인트로 등록하는 기본 구조를 먼저 보고, 이어서 여러 모델에 요청을 분산하는 라우팅 전략과 장애 시 다음 모델로 넘기는 폴백 선언을 정리합니다.

5.2.1 model_list 기본 구조

`config.yaml`의 핵심은 `model_list`입니다. 이 목록의 각 항목은 하나의 배포(deployment)를 나타내며, 클라이언트가 호출할 이름인 `model_name`과 그 이름 뒤의 실제 연결 정보인 `litellm_params`로 이루어집니다 [S7]. `litellm_params` 안의 `api_base`가 바로 추론 엔진을 가리키는 주소입니다.

```
# config.yaml — model_list 기본 구조
model_list:
  - model_name: gemma            # 클라이언트가 호출하는 이름
    litellm_params:
      model: hosted_vllm/gemma-3-27b-it
      # 아래 주소 뒤에서 vLLM이 실제 토큰을 생성합니다(게이트웨이 아님).
      api_base: http://10.20.22.22:9100/v1
      api_key: "none"            # 온프레미스 엔진이 인증을 요구하지 않으면 자리표시 값
```

```
- model_name: qwen
  litellm_params:
    model: hosted_vllm/qwen3-32b
    # 이 주소 뒤도 마찬가지로 추론 엔진(vLLM) 계층입니다.
    api_base: http://10.20.22.23:9100/v1
    api_key: "none"
```

여기서 계층 구분을 다시 확인할 필요가 있습니다. `model_name`과 그 위쪽은 게이트웨이가 아는 이름표에 해당하고, `api_base` 뒤는 vLLM이 실제 연산을 수행하는 추론 엔진 계층입니다. 클라이언트는 `gemma`나 `qwen`이라는 이름만 알면 되고, 그 이름이 어느 서버·어느 포트의 엔진으로 연결되는지는 프록시가 이 설정을 보고 처리합니다. 모델을 교체하거나 서버를 옮길 때 클라이언트 코드를 고칠 필요 없이 `api_base` 값만 바꾸면 되는 것도 이 구조 덕분입니다.

클라이언트 쪽 호출은 일반적인 OpenAI 형식 그대로입니다. 엔드포인트를 프록시 주소로 지정하고 `model` 필드에 등록한 `model_name`을 넣으면, 요청은 프록시를 거쳐 해당 추론 엔진에 도달합니다 [S5].

```
curl http://localhost:4000/v1/chat/completions ₩
 -H "Authorization: Bearer sk-your-master-key" ₩
 -H "Content-Type: application/json" ₩
 -d '{
   "model": "gemma",
   "messages": [{"role": "user", "content": "안녕하세요"}]
 }'
```

이 요청에서 응답을 만들어 내는 주체는 프록시가 아니라 `api_base` 뒤의 vLLM입니다. 프록시는 요청을 받아 올바른 엔진으로 전달하고 응답을 되돌려 주는 역할을 수행합니다. 참고로 위 예시의 모델 식별자 접두어 표기 방식은 사용하는 LiteLLM 버전과 추론 엔진 연동 방식에 따라 달라질 수 있어 확인이 필요합니다.

5.2.2 라우팅 전략과 폴백 선언

같은 모델을 여러 서버에 띄워 두었다면, 요청을 그 서버들에 어떻게 나눌지 정해야 합니다. 이 판단을 라우팅 전략이 담당합니다. LiteLLM은 몇 가지 전략을 제공하며, 조직 상황에 맞는 하나를 `router_settings`에서 선택합니다 [S5].

라우팅 전략	분배 기준	적용이 맞는 상황
simple-shuffle	무작위(가중치 반영 가능) 분배	백엔드 성능이 비슷하고 단순한 균등 분배로 충분할 때
least-busy	진행 중 요청 수가 가장 적은 곳 우선	요청 처리 시간의 편차가 커서 대기 부하를 고르게 맞추고 싶을 때

라우팅 전략	분배 기준	적용이 맞는 상황
usage-based-routing	사용량(토큰·요청 수) 기준 분배	백엔드별 rpm 한도를 넘지 않도록 사용량을 관리하려 할 때
latency-based-routing	최근 응답 지연이 낮은 곳 우선	응답 속도를 우선해 지연이 낮은 백엔드로 요청을 모으고 싶을 때

전략 선택과 함께 각 배포에 요청당 처리량(rpm) 한도를 지정해 두면, 특정 백엔드에 요청이 몰려 한도를 넘기는 상황을 사전에 눌러 둘 수 있습니다 [S5]. 아래 예시는 같은 모델의 두 배포에 라우팅 전략과 폴백을 함께 선언한 구성입니다.

```
 # config.yaml — 라우팅 전략과 폴백 선언
model_list:
  - model_name: gemma
    litellm_params:
      model: hosted_vllm/gemma-3-27b-it
      api_base: http://10.20.22.22:9100/v1
      rpm: 120                  # 이 배포의 요청당 처리량 한도
  - model_name: gemma           # 같은 이름의 두 번째 배포(다른 서버)
    litellm_params:
      model: hosted_vllm/gemma-3-27b-it
      api_base: http://10.20.22.24:9100/v1
      rpm: 120

router_settings:
  routing_strategy: least-busy   # 대기 부하가 가장 적은 배포로 분배

litellm_settings:
  fallbacks:
    # gemma 그룹 전체가 실패하면 qwen 그룹으로 넘어갑니다.
    - gemma: ["qwen"]
```

폴백은 라우팅과 목적이 다릅니다. 라우팅이 정상 상태에서 요청을 나누는 규칙이라면, 폴백은 한 모델 그룹이 응답에 실패했을 때 다음 그룹으로 요청을 넘기는 규칙입니다 [S6]. 위 구성에서 `gemma`로 보낸 요청이 해당 그룹의 모든 배포에서 실패하면, 프록시는 같은 요청을 `qwen` 그룹으로 자동 전환합니다. 클라이언트는 이 전환을 인지하지 않고 정상 응답을 받습니다.

이 자동 폴백이 가용성을 지키는 핵심입니다. 특정 추론 엔진이 내려가더라도 프록시가 살아 있는 다른 그룹으로 요청을 넘기므로, 한 모델의 장애가 곧바로 서비스 중단으로 이어지지 않습니다. 여기서도 계층 구분은 유지됩니다. 라우팅과 폴백은 게이트웨이가 요청의 흐름을 조정하는 기능이며, 각 요청에 대한 실제 토큰 생성은 그 뒤의 추론 엔진이 담당합니다. 게이트웨이는 흐름과 안정성을, 추론 엔진은 연산을 맡는다는 역할 분담이 이 설정에서도 그대로 드러납니다.

이 장에서 세운 최소 프록시와 모델 등록, 그리고 라우팅·폴백 선언은 이후 장에서 다룰 추론 엔진 계층의 성능 튜닝과 짝을 이룹니다. 속도와 용량을 결정하는 파라미터는 다음 장의 추론 엔진 설정으로 이어집니다.

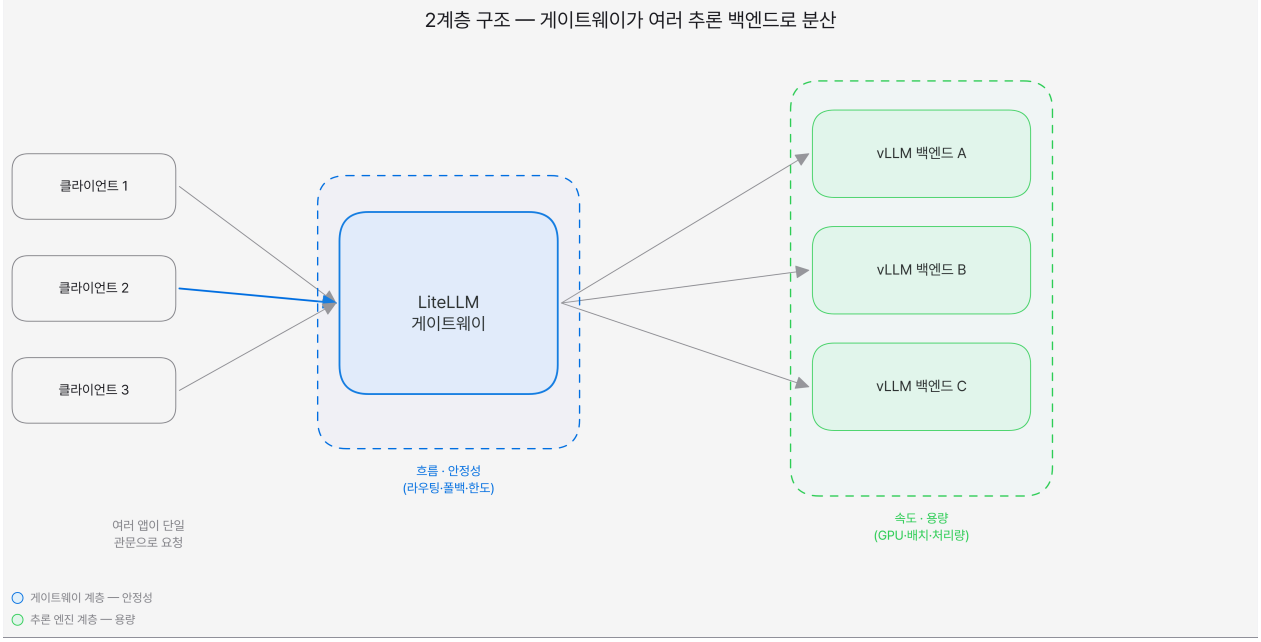
6장: 온프레미스 Gemma·Qwen 연동과 성능 튜닝

온프레미스에서 Gemma나 Qwen 같은 오픈 모델을 운영할 때 성능은 두 계층에서 갈립니다. 하나는 모델이 실제로 도는 추론 엔진 계층이고, 다른 하나는 그 앞에서 요청을 받아 나누어 주는 게이트웨이 계층입니다. 이 둘을 뒤섞어 이해하면 튜닝의 방향을 잃기 쉽습니다. 속도와 용량, 즉 한 요청을 얼마나 빨리 처리하고 한 GPU에 얼마나 많은 요청을 담을 수 있는지는 추론 엔진인 vLLM의 몫입니다. 반면 요청이 어느 백엔드로 흘러가고, 과부하와 장애 상황에서 얼마나 안정적으로 버티는지는 게이트웨이인 LiteLLM의 몫입니다.

이 구분을 처음에 분명히 해 두는 편이 좋습니다. Gemma나 Qwen 자체를 위해 만지는 튜닝 파라미터는 대부분 추론 엔진, 곧 vLLM 쪽 설정입니다. 모델 이름이 붙어 있다고 해서 그것이 게이트웨이에서 조절되는 값이라는 뜻은 아닙니다. LiteLLM에서 다루는 값은 그 모델을 "어떻게 부를지", "얼마나 기다릴지", "여러 대 중 어디로 보낼지"에 관한 것이며, 모델이 토큰을 생성하는 속도 자체를 빠르게 만들지는 않습니다.

아래 표는 이 장에서 다루는 두 계층의 설정을 한눈에 대비한 것입니다. 세부 설명에 들어가기 전에 이 지형도를 먼저 익혀 두시기 바랍니다.

구분	(a) vLLM 파라미터 — 추론 엔진 계층	(b) LiteLLM 설정 — 게이트웨이 계층
결정하는 것	속도·용량(처리량, 메모리, 컨텍스트 물리 한도)	흐름·안정성(라우팅, 타임아웃, 폴백)
대표 파라미터	`max-model-len`, `gpu-memory-utilization`, `tensor-parallel-size`, `quantization`	`max_input_tokens`(컨텍스트 선언), `timeout`, `routing_strategy`, `fallbacks`
조절 위치	각 GPU 노드에서 모델을 띄우는 엔진 실행 인자	여러 백엔드 앞단의 라우터 구성
Gemma·Qwen 튜닝과의 관계	모델별 튜닝 대부분이 여기에 속함	모델의 존재와 한도를 "선언"할 뿐, 생성 속도는 못 바꿈
잘못 만졌을 때	OOM(메모리 부족), 처리량 저하	요청 오라우팅, 불필요한 타임아웃·폴백



클라이언트 요청이 LiteLLM 게이트웨이를 거쳐 여러 vLLM 추론 백엔드로 분산되는 2계층 구조.

6.1 추론 엔진 계층 파라미터 (vLLM에서 실제 속도·용량 결정)

추론 엔진 계층에서는 GPU라는 물리 자원을 어떻게 나누어 쓸지가 관건입니다. 여기서 정하는 값은 실측 처리량과 직결되며, 잘못 잡으면 모델이 아예 뜨지 못하거나 메모리 부족으로 중간에 멈춥니다. Gemma와 Qwen 모두 vLLM 위에서 널리 검증되었으므로 [S11], 아래 파라미터는 두 모델에 공통으로 적용됩니다.

6.1.1 max-model-len과 컨텍스트 길이 설정

`max-model-len`은 모델이 한 요청에서 다룰 수 있는 최대 토큰 길이, 곧 입력과 출력을 합친 컨텍스트의 물리적 상한을 정합니다. 이 값을 책상 크기에 비유할 수 있습니다. 책상이 넓으면 한 번에 여러 자료를 펼쳐 놓고 볼 수 있지만, 그만큼 방에서 차지하는 공간도 커집니다. 컨텍스트 길이도 마찬가지여서, 길게 잡을수록 긴 문서를 한 번에 처리할 수 있으나 GPU 메모리를 그만큼 더 소모합니다.

vLLM의 기본값은 32768입니다 [S9]. 그런데 이 기본값이 항상 최선은 아닙니다. 실제 워크로드가 4천 토큰 안팎의 짧은 대화 위주라면, 책상을 방 전체만큼 넓게 펼쳐 둘 이유가 없습니다. `max-model-len`을 실제로 필요한 길이에 맞추어 낮추면 각 요청이 예약하는 메모리 공간이 줄고, 남는 메모리로 더 많은 동시 요청을 받을 수 있어 메모리 부족(OOM) 위험을 낮추는 데 도움이 됩니다 [S9].

```
# Qwen 모델을 vLLM으로 띄우는 예시 — 컨텍스트를 8192로 조정
vllm serve Qwen/Qwen2.5-7B-Instruct ₩
  --max-model-len 8192
```

위 예시는 컨텍스트 상한을 8192로 낮춘 경우입니다 [S11]. 워크로드가 짧은 요청 위주라면 이렇게 적정 길이로 줄이는 편이 안정적입니다. 반대로 장문 요약이나 긴 코드베이스 분석처럼 실

제로 긴 입력이 오간다면 이 값을 넉넉히 두어야 하며, 그때는 늘어난 메모리 소요를 다음 항에서 다룰 다른 파라미터로 함께 관리해야 합니다. 요컨대 `max-model-len`은 "얼마나 넓은 책상이 실제로 필요한가"를 워크로드에 맞추어 정하는 값이며, 무작정 크게 잡는 설정이 아닙니다.

6.1.2 gpu-memory-utilization과 양자화 선택

`gpu-memory-utilization`은 vLLM이 GPU 메모리 중 얼마만큼을 모델과 처리용으로 확보할지를 비율로 정합니다. 이 값을 방을 채우는 짐의 양에 비유할 수 있습니다. 방을 꽉 채울수록 물건을 많이 둘 수 있지만, 사람이 지나다닐 여유가 사라집니다. 메모리도 마찬가지여서, 높게 잡을수록 더 많은 요청을 담을 수 있으나 예상치 못한 순간에 여유가 없어 문제가 생길 수 있습니다.

기본값은 0.9이며, 단일 인스턴스에서 처리량을 최대로 끌어올릴 때는 0.95까지 올릴 수 있습니다 [S9]. 다만 0.95는 여유를 거의 남기지 않는 설정이므로, 다른 프로세스가 같은 GPU를 함께 쓰거나 요청 크기가 들쭉날쭉한 환경에서는 0.9 부근을 유지하는 편이 안전합니다.

모델이 한 GPU에 담기지 않을 만큼 크다면 `tensor-parallel-size`로 모델을 여러 GPU에 나누어 실을 수 있습니다. 큰 짐을 방 하나에 다 넣지 못할 때 여러 방에 나누어 두는 것과 같습니다. 모델을 N개의 GPU로 분할하면 각 GPU가 감당하는 메모리 압력이 분산됩니다 [S9].

여기에 더해 양자화(quantization)는 처리량과 메모리를 동시에 개선하는 강력한 선택지입니다. 양자화는 모델 가중치를 더 낮은 정밀도로 표현하여 저장 공간과 계산 부담을 줄이는 기법입니다. 짐을 압축 포장해 부피를 줄이는 것에 견줄 수 있습니다. FP8 양자화를 적용하면 H100 GPU에서 FP16 대비 처리량이 약 1.5~2배 높아지고 VRAM 사용량이 약 50% 줄어듭니다 [S10]. Qwen 2.5를 비롯한 다수 모델이 vLLM의 FP8 양자화로 검증되었습니다 [S10].

다만 양자화는 정밀도를 낮추는 만큼 출력 품질이 미세하게 달라질 수 있습니다. 처리량과 메모리 절감이라는 이점과 정밀도 손실 가능성을 함께 저울질하여, 서비스 품질 요구 수준에 맞추어 선택하시기 바랍니다. 품질 검증이 충분한 워크로드라면 FP8은 온프레미스 비용을 크게 낮추는 실용적인 선택입니다.

아래 표는 이 항에서 다룬 vLLM 파라미터를 비유와 함께 정리한 것입니다.

vLLM 파라미터	한 줄 비유	역할
`max-model-len`	한 번에 펼칠 책상 크기	요청당 컨텍스트 물리 상한 결정
`gpu-memory-utilization`	방을 채우는 짐의 양	GPU 메모리 확보 비율(기본 0.9, 최대 0.95)
`tensor-parallel-size`	짐을 나눠 두는 방의 개수	모델을 N개 GPU로 분할해 메모리 압력 분산
`quantization` (FP8 등)	짐의 압축 포장	정밀도를 낮춰 처리량·메모리 개선

다음 표는 양자화 전후의 대략적인 처리량·메모리 변화를 대비한 것입니다(H100 기준) [S10].

구분	FP16(양자화 전)	FP8(양자화 후)
상대 처리량	기준(1배)	약 1.5~2배
VRAM 사용량	기준(100%)	약 50%
정밀도	높음	미세한 손실 가능

FP8의 정밀도 손실이 특정 업무에서 허용 범위에 드는지는 실제 데이터로 확인이 필요합니다. 일반적인 이점은 위와 같으나, 모델과 과제에 따라 결과가 달라질 수 있기 때문입니다.

6.2 게이트웨이 계층 설정 (LiteLLM에서 흐름·안정성 결정)

게이트웨이 계층에서는 앞서 다룬 물리적 속도를 바꾸지 않습니다. 대신 요청이 어디로, 어떤 조건에서 흐를지를 정하여 전체 시스템의 안정성과 가용성을 끌어올립니다. LiteLLM은 여러 추론 백엔드 앞에 서서 라우팅과 부하 분산을 담당하는 게이트웨이입니다 [S5]. 이 계층의 설정은 개별 요청을 더 빨리 만들지는 못하지만, 많은 요청과 장애 상황을 매끄럽게 다스립니다.

6.2.1 컨텍스트 윈도우 선언과 타임아웃

게이트웨이는 각 모델이 감당할 수 있는 컨텍스트 길이를 미리 알고 있어야 합니다. 이 값은 6.1.1의 `max-model-len` 처럼 물리적 상한을 새로 정하는 것이 아니라, 이미 정해진 추론 엔진의 한도를 게이트웨이가 "선언"해 두는 것입니다. 식당 입구에서 "이 방은 최대 8인석"이라고 안내판을 붙여 두는 것과 같습니다. 안내판이 방 크기를 바꾸지는 않지만, 9명이 오면 입구에서 미리 돌려보낼 수 있습니다. 마찬가지로 컨텍스트 한도를 초과하는 요청은 백엔드까지 가지 않고 게이트웨이에서 사전에 차단할 수 있어, 뒤늦은 실패로 자원을 낭비하는 일을 줄입니다.

타임아웃은 게이트웨이가 하나의 응답을 얼마나 오래 기다릴지에 대한 한도입니다. 손님이 주방에 주문을 넣고 무한정 기다리지 않도록 정해 둔 대기 시간과 같습니다. 이 한도를 넘도록 응답이 지연되면 게이트웨이는 해당 백엔드가 지금 건강하지 않다고 판단하고, 다음 항에서 다룰 폴백으로 넘어갈 근거로 삼습니다. 타임아웃을 너무 짧게 잡으면 정상 응답도 실패로 처리되고, 너무 길게 잡으면 장애를 늦게 알아차립니다.

아래 표는 게이트웨이 계층의 주요 설정 항목을 정리한 것입니다 [S5].

게이트웨이 설정 항목	한 줄 비유	역할
컨텍스트 윈도우 선언	방 정원 안내판	초과 요청을 백엔드 도달 전에 사전 차단
타임아웃	주문 대기 한도 시간	지연 시 폴백 전환 판단 근거

이 항목들은 모델의 생성 속도와는 무관합니다. 그것은 6.1에서 다룬 추론 엔진의 몫이며, 여기서는 어디까지 받아 주고 얼마나 기다릴지를 정할 뿐입니다.

6.2.2 모델 라우팅과 부하 분산 설정

라우팅은 들어온 요청을 여러 추론 백엔드 중 어디로 보낼지 정하는 규칙입니다. LiteLLM의 라우터는 같은 모델을 여러 대의 vLLM 인스턴스에 배포해 두고, 요청을 그 사이에 나누어 흘려보냅니다 [S5]. 여러 개의 계산대를 열어 두고 손님을 가장 한산한 줄로 안내하는 안내원에 비유할 수 있습니다.

부하 분산이 처리량과 가용성을 함께 끌어올리는 원리는 이렇습니다. 요청이 한 인스턴스에 몰리지 않고 여러 인스턴스로 고르게 퍼지므로, 시스템 전체가 단위 시간에 처리하는 요청 수가 늘어납니다. 또한 한 백엔드가 장애를 일으켜도 나머지 인스턴스가 요청을 이어받으므로 서비스가 끊기지 않습니다. 여기에 우선순위를 두어 특정 백엔드를 먼저 쓰게 하거나, 응답이 느린 인스턴스를 잠시 제외하도록 구성할 수도 있습니다.

```
 # LiteLLM 라우터에 같은 Qwen 모델을 여러 vLLM 백엔드로 부하 분산하는 예시
model_list:
  - model_name: qwen-chat
    litellm_params:
      model: openai/Qwen2.5-7B-Instruct
      api_base: http://vllm-node-1:8000/v1
  - model_name: qwen-chat
    litellm_params:
      model: openai/Qwen2.5-7B-Instruct
      api_base: http://vllm-node-2:8000/v1

router_settings:
  routing_strategy: least-busy
```

위 구성에서 클라이언트는 `qwen-chat`이라는 하나의 이름만 부르면 되고, 게이트웨이가 두 백엔드 중 덜 바쁜 쪽으로 요청을 보냅니다 [S5]. 여기서 다시 강조해 둘 점이 있습니다. 이 부하 분산은 추론 자체를 가속하는 계층과 다릅니다. 각 vLLM 인스턴스가 토큰을 만들어 내는 속도는 6.1에서 다룬 파라미터가 정하며, 라우팅은 이미 정해진 속도의 인스턴스들을 여럿 세워 두고 요청을 고르게 배분할 뿐입니다. 즉 게이트웨이는 개별 응답을 빠르게 하는 것이 아니라, 전체 흐름을 안정적이고 끊김 없이 유지하는 역할을 맡습니다.

정리하면, 온프레미스에서 Gemma와 Qwen의 성능을 끌어올릴 때 속도와 용량은 vLLM 파라미터에서, 흐름과 안정성은 LiteLLM 설정에서 손봅니다. 모델 이름이 붙은 튜닝이라 하여 모두 게이트웨이의 일이 아니며, 실제 처리량을 좌우하는 대부분의 값은 추론 엔진 계층에 있다는 점을 기억하시기 바랍니다.

7장: 게이트웨이 도입 전후 서비스 운영 비교

앞 장들에서 LiteLLM이 게이트웨이(gateway) 계층에서 무엇을 하는지를 기능 단위로 살펴보았습니다. 이 장에서는 시선을 바꾸어, 그 기능들이 실제 운영 현장에서 어떤 차이를 만드는지를 도입 전과 후의 대비로 보여 드리겠습니다. 의사결정자와 IT 담당자에게 중요한 것은 개별 기능의 목록이 아니라, 그 기능이 부재할 때 조직이 매일 지불하는 비용과 도입 이후 그 비용이 어떻게 사라지는지입니다.

먼저 한 가지를 다시 분명히 해 두겠습니다. LiteLLM은 게이트웨이, 즉 프록시(proxy)이며 토큰을 생성하는 추론(inference) 자체를 빠르게 만들지는 않습니다. 실제 추론과 그 가속은 vLLM 같은 추론 엔진 계층의 몫입니다. 이 장에서 다루는 '개선'은 추론 속도가 아니라, 장애 상황에서의 서비스 연속성과 비용 통제라는 운영 차원의 개선입니다. 두 계층을 혼동하지 않으시도록 미리 구분해 둡니다.

이 장의 흐름은 단순합니다. 먼저 게이트웨이가 없을 때 조직이 실제로 겪는 두 가지 운영 부담을 짚고(7.1), 이어서 LiteLLM 도입 이후 그 부담이 어떻게 해소되는지를 대응 시간과 비용 가시성의 관점에서 대비하겠습니다(7.2). 사례는 특정 고객사의 수치를 인용하는 대신, 온프레미스 AI 플랫폼 운영에서 반복적으로 관찰되는 일반적 운영 패턴으로 서술하겠습니다.

7.1 도입 이전의 운영 부담

게이트웨이 없이 애플리케이션이 여러 모델 엔드포인트에 직접 연결되는 구조는 평상시에는 별 문제가 없어 보입니다. 부담이 드러나는 것은 두 가지 상황에서입니다. 하나는 특정 모델이 장애를 일으켜 다른 모델로 전환해야 하는 순간이고, 다른 하나는 여러 팀의 사용량과 비용이 누적되어 예산을 넘어서는 순간입니다. 이 두 부담은 모두 "요청이 반드시 지나가는 단일 통제 지점이 없다"는 하나의 구조적 원인에서 나옵니다.

7.1.1 장애 대응의 수작업 의존

모델 하나에 장애가 나면 무슨 일이 벌어지는지부터 짚겠습니다. 게이트웨이가 없는 구조에서는 각 애플리케이션이 특정 모델 엔드포인트를 직접 바라보고 있습니다. 그 엔드포인트가 응답하지 않거나 오류를 반환하기 시작하면, 애플리케이션은 스스로 대체 경로를 찾지 못합니다. 다른 모델로 전환하려면 사람이 개입해 설정을 바꾸거나 코드를 배포해야 합니다. [S6]

이 수작업 전환이 왜 위험한지는 야간 장애 시나리오로 살펴보면 분명해집니다.

> *새벽 두 시, 주력으로 쓰던 모델 공급자 측 엔드포인트가 응답을 멈췄다고 가정하겠습니다. 이 모델에 직접 연결된 여러 애플리케이션이 동시에 오류를 반환하기 시작합니다. 담당자가 알림을 받고 접속해 원인을 파악하고, 대체 모델의 엔드포인트 주소와 인증 방식을 확인해 설정을 바꾼 뒤, 각 애플리케이션에 반영하기까지 상당한 시간이 흐릅니다. 그동안 서비스는 계속 실패 응답을 내보내고, 그 실패는 고스란히 사용자에게 전달됩니다.*

이 시나리오의 핵심은 대응이 느리다는 것 자체가 아니라, **장애를 감지한 시점과 서비스가 회복되는 시점 사이의 간격이 전적으로 사람의 반응 속도에 달려 있다**는 데 있습니다. 담당자가 자리를 비웠거나, 대체 모델의 접속 정보가 정리되어 있지 않거나, 전환 절차가 문서화되어 있지 않으면 그 간격은 더 길어집니다. 게다가 장애는 근무 시간을 가려서 오지 않습니다.

도입 이전의 장애 대응 부담을 정리하면 다음과 같습니다.

항목	게이트웨이 없음 (직접 연동)
장애 감지	애플리케이션별로 개별 오류 확인, 통합 신호 없음
대체 모델 전환	사람이 설정 변경·재배포로 수동 처리
대응 가능 시간대	담당자 대기 시간에 종속(야간·휴일 취약)
전환 지식	대체 엔드포인트·인증 정보가 담당자에게 분산
서비스 영향	전환 완료까지 실패 응답이 사용자에게 노출
재현성	대응 절차가 문서화되지 않으면 매번 임기응변

표가 보여 주는 공통점은, 장애 대응의 모든 단계가 자동화되지 않고 사람의 판단과 조작에 걸려 있다는 점입니다. 이 구조에서는 대응이 아무리 숙련되어도 지연과 위험을 원천적으로 없앨 수 없습니다.

7.1.2 비용 추적 부재에 따른 위험

두 번째 부담은 장애처럼 요란하게 드러나지 않기 때문에 더 오래 방치됩니다. 게이트웨이가 없으면 어느 팀이 어떤 모델을 얼마나 호출했는지가 한곳에 모이지 않습니다. 사용량이 팀별·프로젝트별로 집계되지 않으므로, 비용은 월말 청구서로 합산된 총액이 되어서야 눈에 들어옵니다. [S4]

이 가시성 부재가 만들어 내는 위험은 세 갈래로 나뉩니다.

- **통제 지점의 부재**: 사용량에 사전 상한이 걸려 있지 않으므로, 잘못된 반복 호출이나 예상보다 무거운 사용이 발생해도 그것을 중간에서 멈출 장치가 없습니다.
- **책임 소재의 불명확**: 총액만 보이고 팀별 내역이 보이지 않으면, 비용이 초과되었을 때 어느 활동이 원인이었는지 사후에 역추적해야 합니다. 역추적에는 시간과 인력이 듭니다.
- **의사결정 근거의 부재**: 어떤 모델이 어떤 팀에서 얼마의 비용으로 어떤 가치를 내는지 데이터가 없으므로, 더 저렴한 모델로의 전환이나 사용 조정 같은 최적화 판단을 내릴 근거가 없습니다.

이 위험을 부서 단위의 손실로 환산해 보면 크기가 더 분명해집니다. 예산이 정해진 부서에서 월 비용이 예상의 몇 배로 나왔다고 가정하겠습니다. 그 초과분은 다른 계획에서 끌어와 메워야 하는 실제 손실이 됩니다. 더 큰 문제는 원인을 특정하지 못하면 같은 초과가 다음 달에도 반복될 수 있다는 점입니다. 손실이 일회성이 아니라 구조적으로 재발하는 것입니다.

비용 가시성이 없을 때 발생하는 영향을 정리하면 다음과 같습니다.

항목	게이트웨이 없음 (직접 연동)
팀별 사용량 파악	불가능하거나 수작업 취합
비용 인지 시점	월말 청구서 확인 후(사후)
예산 초과 대응	사후 역추적, 원인 특정에 시간 소요
재발 방지	원인 미상 시 다음 달 반복 가능
부서 영향	초과분을 타 계획에서 충당하는 실제 손실
최적화 근거	팀·모델별 비용 데이터 부재로 판단 불가

정리하면, 비용 추적의 부재는 단순히 "얼마 썼는지 모른다"에 그치지 않고, 통제·책임·의사결정이라는 세 가지 관리 기능을 동시에 마비시킵니다. 이 상태가 오래갈수록 조직이 감당하는 위험은 커집니다.

7.2 도입 이후의 운영 개선

앞 절에서 본 두 부담은 모두 단일 통제 지점의 부재에서 비롯되었습니다. LiteLLM은 클라이언트와 추론 엔진 사이에 게이트웨이 계층을 세워, 모든 요청이 반드시 한 지점을 지나가게 만듭니다. 이 한 지점에서 폴백(fallback)과 비용 집계가 작동하기 시작하면, 앞의 두 부담이 어떻게 달라지는지를 순서대로 보겠습니다.

7.2.1 자동 폴백과 가용성 확보

먼저 장애 대응입니다. 게이트웨이가 있으면 모델 전환이 더 이상 사람의 몫이 아닙니다. LiteLLM은 여러 모델을 하나의 논리적 그룹으로 묶어 두고, 우선 순위가 높은 모델이 오류를 반환하거나 응답하지 않으면 미리 지정해 둔 다음 모델로 요청을 자동 전환합니다. 이 전환은 요청 단위로 즉시 일어나며, 애플리케이션은 어떤 모델이 실제로 응답했는지 신경 쓸 필요 없이 동일한 형식의 응답을 받습니다. [S6]

이 구조에서 앞의 야간 장애 시나리오가 어떻게 달라지는지 그림으로 대비해 보겠습니다.

수작업 전환과 자동 폴백에서 장애 발생부터 서비스 회복까지의 흐름 대비.

그림이 강조하는 것은 하나입니다. 자동 폴백은 장애 감지와 서비스 회복 사이의 간격을 사람의 반응 속도에서 떼어 냅니다. 담당자가 자리에 있든 없든, 야간이든 휴일이든, 전환은 게이트웨이가 정해진 순서에 따라 즉시 수행합니다. 사람의 역할은 실시간 조작에서, 폴백 순서를 사전에 설계하고 사후에 로그를 검토하는 일로 옮겨 갑니다.

도입 전후의 대응을 나란히 놓으면 차이가 분명해집니다.

항목	게이트웨이 없음 (직접 연동)	LiteLLM 게이트웨이 도입
장애 시 모델 전환	사람이 설정 변경·재배포로 수동 처리	게이트웨이가 요청 단위로 자동 전환
회복까지의 간격	담당자 반응 속도에 종속	감지 즉시 다음 모델로 전환
야간·휴일 대응	담당자 대기 여부에 취약	시간대와 무관하게 동일 동작
사용자 영향	전환 완료까지 실패 응답 노출	폴백 성공 시 사용자에게 오류 미노출
사람의 역할	실시간 수동 조작	폴백 순서 설계·사후 로그 검토
대응 재현성	매번 임기응변	설정으로 고정되어 일관된 동작

여기서 한 가지는 과장하지 않고 짚어 두겠습니다. 자동 폴백이 모든 장애를 무결하게 흡수한다는 뜻은 아닙니다. 지정해 둔 대체 모델이 함께 장애를 겪거나, 폴백 순서가 실제 상황에 맞게 설계되지 않았다면 효과는 제한됩니다. 즉 자동 폴백의 가치는 "장애가 없어진다"가 아니라 "장애 시 회복이 사람의 반응 속도에서 벗어나 예측 가능해진다"는 데 있습니다. 이 예측 가능성이 곧 가용성(availability) 개선의 실질입니다.

7.2.2 중앙 집중식 비용 가시화 결과

두 번째 개선은 비용 쪽입니다. 모든 요청이 게이트웨이를 지나가게 되면, 그 지점에서 누가·어떤 모델을·얼마나 호출했는지가 자연히 한곳에 모입니다. LiteLLM은 팀과 애플리케이션에 가상 키(virtual key)를 나눠 주고, 이 가상 키 단위로 사용량과 비용을 자동으로 집계합니다. 실제 공급자 키는 게이트웨이 안에만 두고, 바깥에는 통제 가능한 가상 키만 내보내는 구조입니다. [S4] [S7]

이 중앙 집중이 앞 절의 세 가지 위험을 각각 어떻게 되돌리는지 결과 중심으로 보겠습니다.

- **통제의 회복**: 가상 키마다 예산 한도와 호출 한도를 사전에 지정할 수 있으므로, 잘못된 반복 호출이나 과도한 사용이 한도에 닿는 순간 차단되고 알림이 발생합니다. 사고가 진행되는 도중에 멈출 지점이 생기는 것입니다.
- **책임 소재의 명확화**: 비용이 팀·키·모델 단위로 나뉘어 집계되므로, 초과가 발생하면 어느 활동이 원인이었는지를 사후 역추적 없이 곧바로 확인할 수 있습니다.
- **의사결정 근거의 확보**: 어떤 팀이 어떤 모델에 얼마를 쓰는지가 데이터로 남으므로, 더 저렴한 모델로의 전환이나 사용 조정 같은 판단을 근거를 갖고 내릴 수 있습니다. 이 데이터가 뒷받침될 때 비용 절감은 감이 아니라 검증된 흐름이 됩니다.

도입 전후의 비용 가시성을 나란히 대비하면 다음과 같습니다.

항목	게이트웨이 없음 (직접 연동)	LiteLLM 게이트웨이 도입
팀별 사용량 파악	불가능하거나 수작업 취합	가상 키 단위로 자동 집계
비용 인지 시점	월말 청구서 확인 후(사후)	실시간 대시보드로 상시 확인
예산 상한	없음(사후 확인)	팀·키·모델별 사전 한도 설정
초과 대응	사후 역추적	한도 도달 시 실시간 차단·알림
원인 특정	총액만 보여 시간 소요	키·모델 단위로 즉시 식별
최적화 판단	근거 데이터 부재	팀·모델별 비용 데이터 기반

표의 오른쪽 열이 공통으로 가리키는 것은 하나의 원리입니다. 모든 요청이 한 지점을 반드시 통과하게 만들면, 그 지점에서 측정·통제·기록이 동시에 가능해진다는 것입니다. 비용 가시화의 성과는 단순히 숫자가 보인다는 데 있지 않고, 그 숫자를 근거로 예산을 통제하고 다음 결정을 내릴 수 있게 된다는 데 있습니다.

다만 도입 범위에 대해서는 한 가지를 정확히 구분해 두겠습니다. 사용량 집계와 가상 키 같은 기본적인 비용 통제는 LiteLLM 코어에서 활용할 수 있지만, 세밀한 접근 제어나 일부 거버넌스(governance)·감사 기능은 별도의 상용 조건이 적용되는 enterprise 기능에 속합니다. '모든 기능이 무료로 제공된다'고 단정하기보다는, 조직의 요구 범위에 따라 코어로 충분한지 상용 기능이 필요한지를 나누어 검토하시는 편이 정확합니다. 구체적인 기능 경계와 가격 조건은 확인이 필요합니다. [S4]

정리하면, 도입 전의 두 부담이었던 수작업 장애 대응과 비용 통제 공백은, 게이트웨이라는 단일 통제 지점을 세우는 것만으로 자동 폴백과 중앙 집중식 비용 가시화라는 두 개선으로 뒤집힙니다. 이 대비가 곧 LiteLLM을 AI 플랫폼의 필수 게이트웨이로 검토해야 하는 운영상의 근거입니다.

8장: 유사 오픈소스와의 계층별 비교

LiteLLM을 처음 검토하는 엔지니어가 가장 자주 부딪히는 오해는 "LiteLLM과 vLLM 중 무엇을 써야 하는가"라는 질문입니다. 이 질문은 성립하지 않습니다. 두 도구는 경쟁 관계가 아니라 서로 다른 계층에서 각자의 역할을 수행하며, 실제 운영 환경에서는 함께 놓이는 경우가 많습니다. vLLM 공식 문서 역시 vLLM을 추론 엔진으로, LiteLLM을 게이트웨이로 규정하고 두 도구를 결합해 사용하는 구성을 안내합니다[S8].

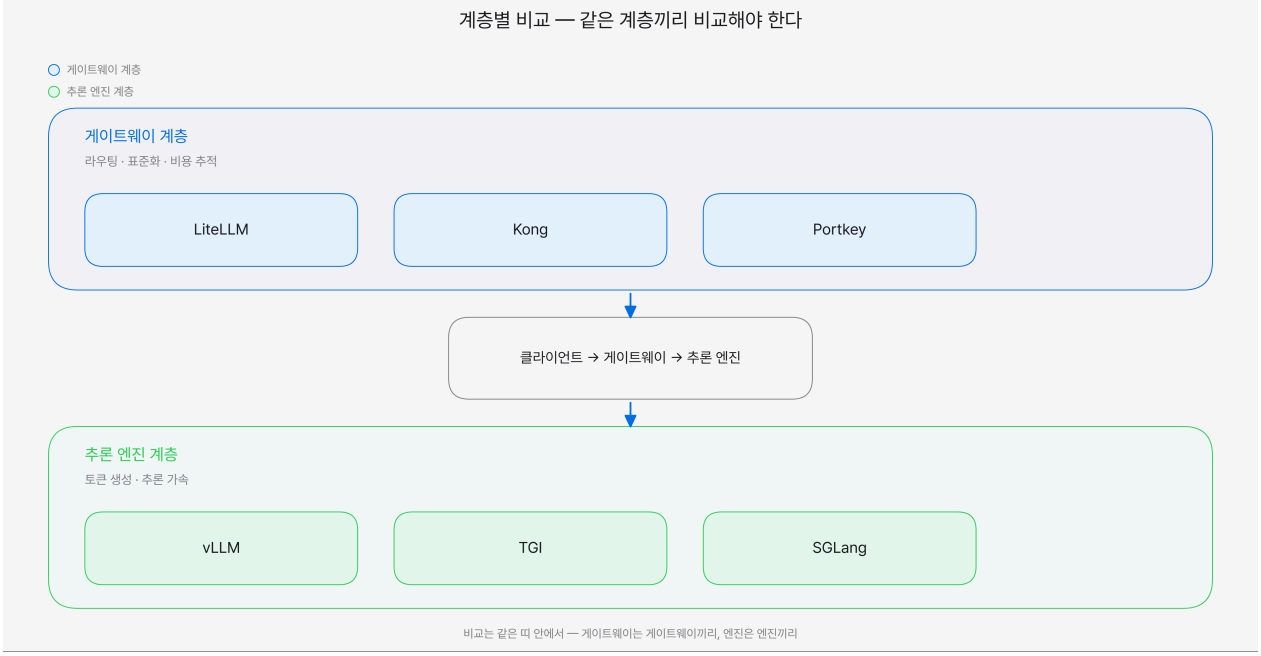
이 장은 검색 의도에 맞춰 유사 도구와의 계층 차이를 정리합니다. 먼저 추론 엔진 계층과 게이트웨이 계층을 명확히 구분한 뒤, 같은 계층 안에서만 대안을 비교하는 원칙을 세웁니다. 그다음 성능·확장성·운영 비용, 그리고 라이선스·보안·호환성이라는 세 가지 기준으로 도구를 비교해 조직 상황에 맞는 조합을 고르는 판단 근거를 제공합니다.

핵심 관점을 먼저 요약하면 다음과 같습니다. 추론 속도를 끌어올리고 싶다면 추론 엔진 계층을 보아야 하고, 여러 모델을 표준화하고 접근을 통제하며 비용을 관측하고 싶다면 게이트웨이 계층을 보아야 합니다. 두 목적은 서로 배타적이지 않으므로, 대부분의 실무 구성은 두 계층을 동시에 채택합니다.

8.1 추론 엔진과 게이트웨이의 계층 구분

계층을 나누는 기준은 "요청 경로에서 어떤 일을 담당하는가"입니다. 추론 엔진은 모델 가중치를 메모리에 적재하고 실제 토큰을 생성하는 계층입니다. 게이트웨이는 여러 모델과 제공자 앞단에 놓여 요청을 표준 형식으로 받아 적절한 백엔드로 라우팅하고, 인증·요금 한도·비용 집계·재시도 같은 공통 관심사를 처리하는 계층입니다.

주방에 빗대면 이해가 쉽습니다. 추론 엔진은 실제로 요리를 만드는 조리대이고, 게이트웨이는 여러 조리대로 들어오는 주문을 받아 분배하고 서빙 규칙을 관리하는 홀의 접수대입니다. 조리대의 화력을 높이고 싶다면 조리대를, 주문 접수와 정산 방식을 통일하고 싶다면 접수대를 손보아야 합니다. 두 역할을 한 도구에 뭉뚱그리면 비교 자체가 어긋납니다.



추론 엔진 계층과 게이트웨이 계층의 역할·대표 도구 구분.

8.1.1 vLLM·TGI·SGLang 추론 엔진군

추론 엔진군은 모델을 GPU 메모리에 올려 놓고 프롬프트를 받아 토큰을 생성하는 계층입니다. 이 계층의 관심사는 처리량(throughput), 지연 시간(latency), GPU 메모리 효율입니다. 추론 속도를 개선하려는 목적이라면 바로 이 계층을 살펴보아야 합니다.

vLLM은 PagedAttention 기법으로 KV 캐시 메모리를 페이지 단위로 관리해 높은 처리량을 제공하는 추론 엔진입니다. 연속 배칭(continuous batching), 텐서 병렬(tensor parallel), 정량화(quantization) 등 대규모 서빙에 필요한 파라미터를 폭넓게 제공합니다[S9]. TGI(Text Generation Inference)는 프로덕션 서빙에 초점을 둔 추론 서버로, 텍스트 생성 워크로드에 맞춘 최적화와 운영 편의 기능을 제공합니다. SGLang은 구조화된 생성과 복잡한 프롬프트 흐름 제어에 강점을 둔 엔진으로, 프로그래밍 가능한 생성 패턴과 프리픽스 캐시 재사용을 강조합니다.

세 엔진 모두 OpenAI 호환 API를 노출할 수 있으므로, 게이트웨이는 이들을 하나의 백엔드로 취급해 연결합니다. 즉 어떤 추론 엔진을 고르든 그 앞단에 게이트웨이를 둘 수 있으며, 이 지점이 두 계층이 결합되는 접점입니다.

추론 엔진	주요 특징	강점 영역	인터페이스
vLLM	PagedAttention 기반 KV 캐시 관리, 연속 배칭, 텐서 병렬[S9]	높은 처리량, 대규모 동시 요청	OpenAI 호환 서버
TGI	프로덕션 서빙 지향 최적화, 운영 기능 내장	안정적 프로덕션 서빙	OpenAI 호환 / 자체 API

추론 엔진	주요 특징	강점 영역	인터페이스
SGLang	구조화 생성, 프리픽스 캐시 재사용, 프롬프트 흐름 제어	복잡한 생성 패턴, 프리픽스 재사용	OpenAI 호환 서버

세 엔진의 상대적 성능은 모델·하드웨어·워크로드 특성에 따라 달라지므로, 구체적 수치 우열은 조직의 벤치마크로 확인이 필요합니다. 이 표는 각 엔진이 지향하는 방향을 정리한 것이며, 어느 하나가 항상 우월하다는 뜻은 아닙니다.

8.1.2 Kong·Portkey 게이트웨이군

게이트웨이군은 LiteLLM과 같은 계층에 있는 대안입니다. 이 계층의 관심사는 여러 모델·제공자의 표준화, 인증과 접근 통제, 요금 한도, 비용 관측, 재시도와 폴백입니다. 게이트웨이를 선택할 때는 반드시 같은 계층 안에서만 대안을 비교해야 하며, 추론 엔진과 나란히 놓고 우열을 가려서는 안 됩니다.

Portkey는 LLM 요청을 대상으로 하는 게이트웨이로, 라우팅·재시도·관측·비용 추적 같은 LLM 특화 기능을 제공합니다[S12]. Kong은 원래 범용 API 게이트웨이로 성장한 도구이며, AI 게이트웨이 기능을 더해 LLM 트래픽을 다룰 수 있게 확장되었습니다[S12]. Cloudflare 역시 엣지 인프라 위에서 AI 게이트웨이 기능을 제공하는 대안으로 언급됩니다[S12]. LiteLLM은 오픈소스 기반으로 100개 이상의 제공자를 OpenAI 호환 형식으로 통합하고, 프록시 서버 형태로 키 관리·예산·요금 한도를 제공하는 게이트웨이입니다.

게이트웨이	성격	특징	배포 형태
LiteLLM	LLM 전용 오픈소스 게이트웨이	다수 제공자 OpenAI 호환 통합, 키·예산·요금 한도, 프록시 서버	자체 호스팅
Portkey	LLM 전용 게이트웨이 [S12]	LLM 특화 라우팅·재시도·관측·비용 추적	SaaS / 자체 호스팅
Kong	범용 API 게이트웨이 기반 확장[S12]	폭넓은 플러그인 생태계에 AI 게이트웨이 기능 결합	자체 호스팅 / 관리형
Cloudflare AI Gateway	엣지 기반 게이트웨이 [S12]	엣지 인프라 위 관측·캐시·요금 제어	관리형

이 네 도구는 모두 게이트웨이 계층에 속하므로 직접 비교가 가능합니다. 어떤 도구가 적합한지는 조직이 자체 호스팅을 선호하는지, 관리형 서비스를 선호하는지, 기존에 어떤 인프라를 운영 중인지에 따라 달라집니다.

8.2 평가 기준별 비교 정리

계층을 구분했으니 이제 도구를 어떤 기준으로 판단할지 정리합니다. 여기서도 원칙은 동일합니다. 같은 계층 안에서만 비교하고, 계층이 다른 도구를 한 표에 뒤섞지 않습니다. 기준을 성능·확장성·운영 비용, 그리고 라이선스·보안·호환성으로 나누어 살펴봅니다.

8.2.1 성능·확장성·운영 비용 비교

성능, 확장성, 운영 비용이라는 3가지 기준은 계층에 따라 의미가 다릅니다. 추론 엔진 계층에서 성능은 토큰 생성 처리량과 지연 시간을 뜻하지만, 게이트웨이 계층에서 성능은 요청을 백엔드로 전달하는 과정에서 더해지는 부가 지연과 처리 용량을 뜻합니다. 두 계층의 성능을 같은 축에 올려 비교하면 잘못된 결론에 이르므로, 아래 표는 계층별로 나누어 정리했습니다.

계층	성능의 의미	확장성의 의미	운영 비용의 의미
추론 엔진 (vLLM·TGI·SGLang)	토큰 생성 처리량, 지연 시간, GPU 메모리 효율 [S9]	GPU 추가·텐서 병렬로 모델 서빙 용량 확장	GPU 자원 비용이 지배적
게이트웨이 (LiteLLM·Portkey·Kong 등)	요청당 부가 지연, 동시 요청 처리 용량[S12]	프록시 인스턴스 수평 확장, 다수 백엔드 라우팅	게이트웨이 인스턴스 운영 비용, 관리형은 사용량 과금

조직 상황에 맞는 조합을 고를 때는 목적을 먼저 확정해야 합니다. GPU를 최대한 활용해 추론 속도를 높이는 것이 과제라면 추론 엔진의 배칭·병렬 파라미터를 조정하는 것이 우선입니다[S9]. 반대로 여러 모델과 제공자를 하나의 창구로 묶고 비용과 접근을 통제하는 것이 과제라면 게이트웨이를 도입하고 그 확장성과 부가 지연을 검토하는 것이 우선입니다[S12]. 두 과제가 함께 있다면 추론 엔진 위에 게이트웨이를 얹는 구성이 자연스럽습니다.

관리형 게이트웨이와 자체 호스팅 게이트웨이의 운영 비용 구조가 다르다는 점도 판단에 반영해야 합니다. 관리형 서비스는 초기 운영 부담이 낮은 대신 사용량에 따라 과금되고, 자체 호스팅 오픈소스 게이트웨이는 인프라 운영 책임을 지는 대신 라이선스 비용 없이 규모를 키울 수 있습니다. 각 방식의 총소유비용은 트래픽 규모와 내부 운영 역량에 따라 달라지므로 조직별 산정이 필요합니다.

8.2.2 라이선스·보안·호환성 비교

라이선스, 보안, 호환성은 도입 결정에 필요한 비기능 요건입니다. 이 항목들은 성능처럼 벤치마크로 측정되지 않고, 조직의 정책과 규제 요건에 맞는지를 확인하는 방식으로 검토해야 합니다. 그래서 아래는 표와 함께 확인 목록 형태로 정리했습니다.

라이선스 측면에서 LiteLLM은 오픈소스로 배포되어 자체 호스팅과 소스 검토가 가능합니다. 라이선스의 정확한 조건과 상용 활용 범위는 배포 시점의 원문을 기준으로 확인이 필요합니다[S3]. Portkey와 Cloudflare AI Gateway는 관리형 서비스 성격이 강하고, Kong은 오픈소스 코어와 상용 확장이 구분되는 구조를 갖습니다. 각 도구의 정확한 라이선스 구분은 공급자 문서로 확인이 필요합니다[S12].

도구(게이트웨이 계층)	라이선스 성격	배포·운영 방식	확인이 필요한 지점
LiteLLM	오픈소스[S3]	자체 호스팅	상용 활용 범위, 배포 시점 라이선스 원문[S3]
Portkey	상용 서비스 중심[S12]	SaaS / 자체 호스팅 옵션	자체 호스팅 시 조건, 데이터 처리 위치
Kong	오픈소스 코어 + 상용 확장[S12]	자체 호스팅 / 관리형	AI 게이트웨이 기능의 상용 여부
Cloudflare AI Gateway	관리형 서비스[S12]	엣지 관리형	데이터 경유 정책, 요금 구조

보안과 호환성은 다음 확인 목록으로 점검하기를 권합니다.

- 인증·접근 통제: 개별 API 키 발급, 팀·프로젝트 단위 권한 분리, 요금 한도 설정이 조직 정책과 맞는지 확인이 필요합니다.
- 데이터 처리 위치: 요청·응답이 외부 관리형 인프라를 경유하는지, 자체 인프라 안에서만 처리되는지 확인이 필요합니다. 규제 대상 데이터를 다루는 경우 이 항목이 도입 가능 여부를 좌우합니다.
- 감사·관측: 요청 로그, 비용 집계, 사용량 추적이 규정된 보관·감사 요건을 충족하는지 확인이 필요합니다.
- 호환성: 사용 중인 추론 엔진과 외부 제공자가 OpenAI 호환 형식으로 연결되는지, 게이트웨이가 이들을 백엔드로 지원하는지 확인이 필요합니다. LiteLLM처럼 다수 제공자를 OpenAI 호환 형식으로 통합하는 게이트웨이는 이 항목에서 유리합니다.
- 라이선스 정합성: 자체 호스팅 요건, 상용 활용 범위, 재배포 조건이 조직의 소프트웨어 정책과 충돌하지 않는지 확인이 필요합니다[S3].

비기능 요건은 성능 수치보다 도입 성패에 더 크게 작용하는 경우가 많습니다. 특히 데이터 처리 위치와 라이선스 정합성은 기술 검토가 끝난 뒤에도 도입을 막을 수 있는 항목이므로, 검토 초기에 위 목록으로 점검하기를 권합니다. 계층을 정확히 구분하고 같은 계층 안에서 이 기준들을 적용하면, 추론 엔진과 게이트웨이를 각각 제자리에 배치한 견고한 조합을 설계할 수 있습니다.

8.3 OpenRouter와 LiteLLM 비교

게이트웨이 계층 안에서도 실무자들이 자주 함께 저울질하는 대상이 OpenRouter입니다. 두 도구는 "여러 모델을 하나의 방식으로 호출한다"는 목적이 비슷해 보이지만, 운영 모델이 근본적으로 다릅니다. 이 차이는 온프레미스 AI 플랫폼을 검토하는 조직에 특히 중요합니다[S13].

OpenRouter는 **관리형 클라우드 애그리게이터**입니다. 여러 공급자의 모델을 하나의 통합 API와 통합 청구로 묶어 제공하며, 사용자는 인터넷 너머의 서비스를 호출하기만 하면 됩니다. 직접 운영할 인프라가 없다는 점이 가장 큰 장점입니다. 반면 LiteLLM은 **자체 호스팅 오픈소스 게이트웨이**로, 조직의 인프라 안에서 실행됩니다[S13].

두 방식의 차이를 정리하면 다음과 같습니다.

기준	OpenRouter (관리형 SaaS)	LiteLLM (자체 호스팅)
실행 위치	공급자가 운영하는 클라우드	조직이 운영하는 자체 인프라
데이터 경로	요청이 중간 관리 계층을 거쳐 공급자로 전달	요청이 자체 인프라를 거쳐 공급자로 직접 전달
데이터 잔존	전면 보장 없음. Zero Data Retention은 일부 모델만 적용, 일부 모델은 로깅 가능성 고지[S13]	중간 애그리게이터가 없어 별도 잔존 정책을 따질 대상이 없음(공급자 약관은 별개)[S13]
인프라 운영	불필요	PostgreSQL·Redis·Docker 직접 운영
비용 구조	공급자 가격 통과 + 선불 크레딧 구매 시 5.5% 플랫폼 수수료(구매당 최소 금액 존재)[S13]	자체 호스팅은 플랫폼 수수료 없음. 인프라 비용 부담
적합 상황	인프라 운영 없이 빠른 다중 모델 접근·통합 청구가 필요할 때	데이터 레지던시·거버넌스·자체 통제가 필요할 때

온프레미스 AI 플랫폼의 핵심 요구가 "민감한 요청 데이터가 조직 네트워크를 벗어나지 않아야 한다"는 것이라면, 요청이 외부 관리 계층을 통과하는 OpenRouter 모델은 데이터 레지던시 검토가 까다로워집니다. 반면 LiteLLM 자체 호스팅은 애플리케이션에서 조직이 운영하는 게이트웨이를 거쳐 선택한 공급자로 곧장 이어지므로, 중간에 별도 잔존 정책을 따질 계층이 없습니다[S13].

두 도구가 배타적인 것은 아닙니다. **하이브리드 구성**도 가능합니다. LiteLLM을 사내 표준 게이트웨이(애플리케이션이 바라보는 단일 엔드포인트)로 두고, OpenRouter를 그 뒤에 놓인 공급자 중 하나로 등록하는 방식입니다. 이렇게 하면 애플리케이션은 조직이 소유한 안정적 인터페이스와 팀별 예산·권한 관리를 유지하면서, 폭넓은 모델 접근은 OpenRouter가 뒷단에서 담당하도록 나눌 수 있습니다[S13]. 온프레미스 데이터 통제를 유지하되 특정 실험 모델만 외부에서 빌려 쓰고 싶을 때 유용한 절충안입니다.

정리하면, OpenRouter와 LiteLLM은 같은 게이트웨이 계층에 속하지만 "누가 인프라를 운영하고 데이터가 어디를 지나는가"에서 갈립니다. 구축형 AI 플랫폼의 데이터 통제 요건이 강할수록 자체 호스팅 게이트웨이가 유리하며, 필요 시 하이브리드로 두 방식의 장점을 함께 취할 수 있습니다.

출처

- [S3] LiteLLM 라이선스 문서
- [S8] vLLM 공식 문서 — LiteLLM 연동 안내 (vLLM=추론 엔진, LiteLLM=게이트웨이, 함께 사용)

- [S9] vLLM 파라미터 문서
- [S12] LiteLLM vs Portkey·Kong·Cloudflare 게이트웨이 비교 자료
- [S13] OpenRouter vs LiteLLM — 관리형 SaaS와 자체 호스팅 게이트웨이 비교 자료

9장: 엔터프라이즈 AI 플랫폼 적용 방안과 도입 로드맵

앞선 장들에서 LiteLLM의 라우팅, 폴백, 비용 추적, 거버넌스 기능을 개별적으로 살펴보았습니다. 이 마지막 장은 그 기능들을 하나의 그림으로 모읍니다. 구축형 AI 플랫폼에서 LiteLLM이 왜 필요한지를 멀티모델 표준화와 거버넌스라는 두 축으로 정리하고, 소규모 파일럿에서 전사 확산까지 이어지는 단계별 도입 로드맵을 제시합니다. 여기서 한 가지 구분을 다시 분명히 하고 시작하겠습니다. LiteLLM은 여러 모델 앞에 서는 게이트웨이(프록시)입니다. 추론 속도 자체를 끌어올리는 일은 vLLM과 같은 추론 엔진 계층의 몫입니다. 두 계층은 서로 대체하는 관계가 아니라, 추론 엔진이 모델을 빠르게 서빙하고 게이트웨이가 그 앞에서 표준화와 통제를 담당하는 상호 보완 관계입니다. 이 장의 모든 제안은 이 전제 위에서 이루어집니다. [S2][S8]

9.1 플랫폼 통합 적용 방안

구축형 AI 플랫폼을 운영하는 조직은 시간이 지날수록 다루어야 할 모델의 수가 늘어납니다. 온프레미스에 직접 서빙하는 오픈소스 모델, 클라우드 사업자가 제공하는 상용 모델, 특정 업무에 특화된 소형 모델이 뒤섞이면서 각각의 호출 방식과 인증 체계가 달라집니다. 이 절에서는 LiteLLM이 이러한 이질성을 어떻게 하나의 접점으로 묶어내는지, 그리고 그 위에서 거버넌스를 어떻게 세우는지를 살펴봅니다.

9.1.1 멀티모델 표준화 해결 방안

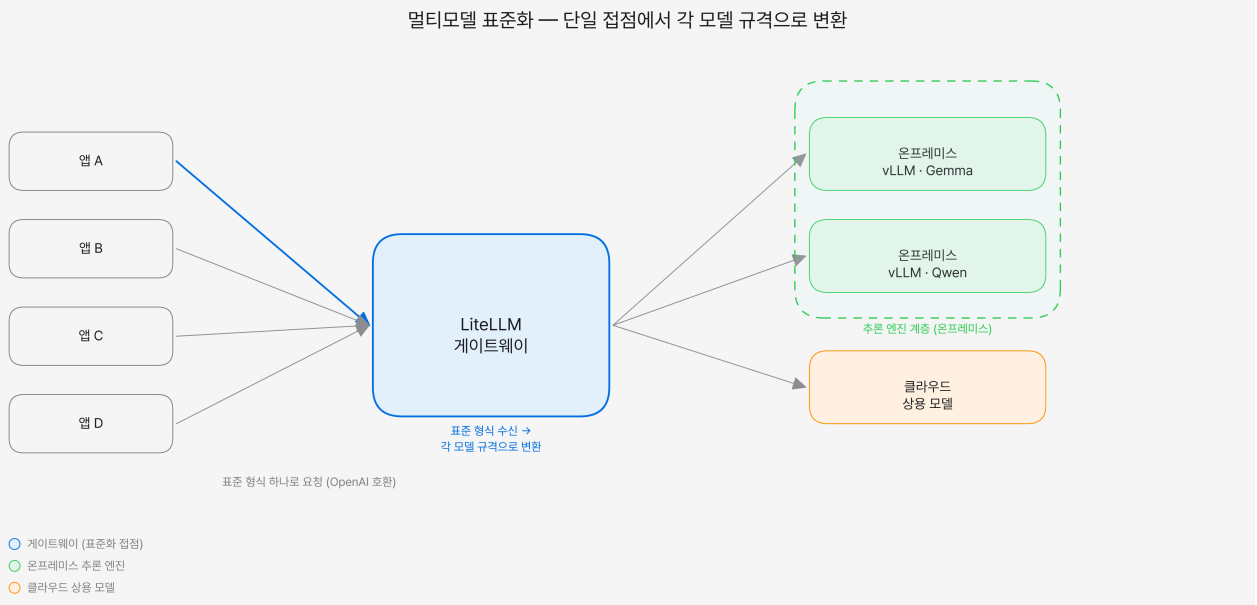
여러 모델을 도입한 조직이 가장 먼저 마주하는 문제는 애플리케이션마다 모델 연동 코드를 다르게 작성해야 한다는 점입니다. 상용 모델은 상용 모델대로, 온프레미스 모델은 온프레미스 모델대로 요청 형식과 응답 구조, 인증 헤더가 제각각입니다. 새로운 모델을 추가하거나 기존 모델을 교체할 때마다 애플리케이션 코드를 함께 손봐야 한다면, 모델 다양성이 오히려 확장을 가로막는 부담이 됩니다.

LiteLLM은 서로 다른 모델을 하나의 표준 형식으로 묶어 이 문제를 해결합니다. 애플리케이션은 게이트웨이가 노출하는 단일 인터페이스로 요청을 보내고, 게이트웨이가 각 모델의 고유한 호출 규격으로 변환해 전달합니다. 뒤에 어떤 모델이 있든 애플리케이션이 마주하는 접점은 동일하게 유지됩니다. 새 모델을 추가하는 일은 게이트웨이 설정에 항목 하나를 등록하는 작업으로 좁혀지며, 애플리케이션 코드는 그대로 둘 수 있습니다. [S2][S7]

이 구조는 통합 물류센터에 비유할 수 있습니다. 여러 공장에서 서로 다른 규격으로 물건이 들어오더라도, 물류센터를 거치면 동일한 규격의 상자로 재포장되어 나갑니다. 이후 공정은 물건이 어느 공장에서 왔는지 신경 쓸 필요 없이 표준 상자만 다루면 됩니다. 공장이 늘거나 바뀌어도 물류센터가 그 변화를 흡수하므로, 이후 공정은 영향을 받지 않습니다. LiteLLM은 이 물류센터의 역할을 맡아 모델 계층의 변화를 애플리케이션과 분리합니다.

표준화가 가져오는 이점은 코드 단순화에 그치지 않습니다. 온프레미스 모델과 클라우드 모델을 같은 형식으로 다룰 수 있으므로, 업무 성격에 따라 모델을 유연하게 배치할 수 있습니다. 민감한 데이터를 다루는 요청은 내부에 서빙한 모델로, 범용적인 요청은 비용 효율이 높은 모델로 보내는 배치를 설정 수준에서 조정할 수 있습니다. 여기서 온프레미스 모델의 응답 속도는

vLLM 같은 추론 엔진이 담당하며, 게이트웨이는 그 앞에서 어느 모델로 요청을 보낼지 결정하는 표준화된 접점을 제공합니다. [S2][S8]



멀티모델 표준화 구성도 — 다수의 애플리케이션이 LiteLLM 게이트웨이라는 단일 접점으로 요청을 보내고, 게이트웨이가 온프레미스 추론 엔진(vLLM 등)과 클라우드 상용 모델로 표준화된 형식을 각 모델의 고유 규격으로 변환해 전달하는 구조.

9.1.2 거버넌스와 팀별 예산 관리

표준화된 접점이 마련되면, 그 접점을 통과하는 모든 요청을 한곳에서 통제할 수 있는 기반이 함께 생깁니다. 여러 팀이 각자 모델을 직접 호출하던 방식에서는 누가 어떤 모델을 얼마나 사용하는지 파악하기 어렵고, 비용도 사후에야 집계됩니다. 게이트웨이를 단일 통로로 두면 인증과 사용량 집계, 정책 적용을 중앙에서 일관되게 수행할 수 있습니다.

LiteLLM은 중앙 인증을 통해 모든 요청이 발급된 접근 키를 거치도록 합니다. 이 접근 키는 팀이나 프로젝트 단위로 발급되며, 각 키마다 사용할 수 있는 모델의 범위와 예산 한도를 지정할 수 있습니다. 특정 팀의 키에는 그 팀이 승인받은 모델만 허용하고, 월 사용 한도를 설정해 한도에 도달하면 요청을 제한하는 방식으로 운영할 수 있습니다. 사용 내역은 감사 로그로 남아 언제 어떤 키가 어떤 모델을 호출했는지 추적할 수 있습니다. [S4][S5]

이러한 기능은 기술 용어를 넘어 조직의 통제 언어로 옮겨 이해할 수 있습니다. 접근 키는 팀에 부여된 사용 권한증이며, 예산 한도는 각 팀에 배정된 예산 상한입니다. 감사 로그는 지출 내역서에 해당합니다. 재무 담당자가 부서별 예산을 배정하고 집행 내역을 확인하듯, 플랫폼 운영자는 팀별 접근 키에 한도를 배정하고 사용 내역을 확인합니다. 모델 사용이 통제 밖의 영역이 아니라 예산 관리 체계 안으로 들어오는 것입니다.

다음 표는 거버넌스 기능을 조직 운영 관점에서 정리한 것입니다.

거버넌스 기능	게이트웨이 동작	조직 운영 의미
중앙 인증	모든 요청이 발급된 접근 키를 통과	승인된 사용자만 모델에 접근
팀별 접근 키	팀·프로젝트 단위로 키 발급, 모델 범위 지정	팀별 사용 권한과 책임 구분
예산 한도	키마다 사용 한도 설정, 초과 시 제한	부서별 예산 배정과 초과 방지
감사 로그	호출 시각·키·모델·사용량 기록	지출 내역 추적과 사후 감사
사용량 집계	키·모델별 사용량 실시간 집계	비용 배분과 예산 소진 현황 파악

거버넌스 체계가 갖추어지면 모델 사용이 예측 가능하고 통제 가능한 운영 영역으로 자리 잡습니다. 이는 구축형 AI 플랫폼을 실무에 안정적으로 얹기 위한 전제 조건이며, 다음 절에서 다룰 단계적 도입의 토대가 됩니다. [S4]

9.2 단계별 도입 로드맵

LiteLLM을 처음부터 전사 규모로 도입하기보다, 소규모 파일럿으로 핵심 동작을 검증한 뒤 확산하는 단계적 접근을 권장합니다. 파일럿에서 라우팅과 폴백, 비용 추적이 기대대로 작동하는지 확인하고, 성공 기준을 사전에 합의해두면 확산 단계의 의사결정이 훨씬 수월해집니다. 이 절은 검증 절차와 확산 로드맵을 순서대로 제시합니다.

9.2.1 파일럿 검토와 검증 절차

파일럿의 목적은 게이트웨이가 실제 업무 흐름에서 안정적으로 동작하는지를 제한된 범위에서 확인하는 것입니다. 대상은 한두 개 팀의 대표적인 사용 사례로 좁히고, 검증할 항목을 미리 정해 결과를 명확히 판단할 수 있게 합니다. 특히 하나의 모델에 장애가 생겼을 때 다른 모델로 요청이 자동으로 넘어가는 폴백 동작과, 사용량이 정확히 집계되는지를 중점적으로 확인합니다. [S6][S7]

다음은 파일럿 단계에서 점검할 항목을 정리한 체크리스트입니다.

- [ ] 단일 접점 연동: 애플리케이션이 게이트웨이의 표준 인터페이스만으로 여러 모델을 호출하는지 확인
- [ ] 라우팅 동작: 요청이 설정된 규칙에 따라 의도한 모델로 전달되는지 확인
- [ ] 폴백 동작: 특정 모델 장애 시 대체 모델로 자동 전환되는지, 사용자에게 오류가 노출되지 않는지 확인
- [ ] 비용 추적: 팀·모델별 사용량과 비용이 정확히 집계되는지 확인
- [ ] 예산 한도: 접근 키의 한도 설정이 적용되고, 한도 초과 시 요청이 제한되는지 확인
- [ ] 감사 로그: 호출 내역이 시각·키·모델 단위로 기록되고 조회 가능한지 확인

- [ ] 온프레미스 모델 연동: 내부 추론 엔진에 서빙한 모델이 게이트웨이를 통해 정상 호출되는지 확인
- [ ] 성능 영향: 게이트웨이 경유로 발생하는 지연이 업무 요구 수준 안에 있는지 확인

성공 기준은 파일럿을 시작하기 전에 합의해두어야 합니다. 예를 들어 폴백이 발생한 요청의 성공률, 비용 집계의 정확도, 게이트웨이 경유로 늘어나는 응답 지연의 허용 범위를 수치로 정해두면, 파일럿 종료 시점에 확산 여부를 객관적으로 판단할 수 있습니다. 기준을 사후에 정하면 결과 해석이 흐려지므로, 사전 합의가 검증 절차의 핵심입니다. [S7]

9.2.2 확산 도입 제안과 성과 측정

파일럿이 합의된 기준을 충족하면 적용 범위를 단계적으로 넓혀갑니다. 확산은 한 번에 전사로 전환하기보다, 팀과 사용 사례를 순차적으로 추가하며 운영 경험을 축적하는 방식이 안전합니다. 각 단계마다 거버넌스 정책을 함께 확장하고, 성과를 지표로 확인한 뒤 다음 단계로 넘어가는 흐름을 권장합니다.

다음 표는 파일럿 이후의 확산 도입 로드맵을 단계별로 정리한 것입니다.

단계	대상 범위	주요 활동	성과 측정 지표
1단계: 파일럿	1~2개 팀, 대표 사용 사례	라우팅·폴백·비용 추적 검증, 성공 기준 합의	폴백 성공률, 비용 집계 정확도, 응답 지연
2단계: 제한 확산	초기 도입 부서 확대	팀별 접근 키·예산 한도 운영, 정책 표준화	팀별 예산 준수율, 가용성(장애 시 서비스 연속성)
3단계: 전사 확산	전 조직 애플리케이션	단일 접점 전면 적용, 신규 모델 온보딩 절차화	모델 온보딩 소요 시간, 전사 비용 가시성
4단계: 정착·최적화	운영 안정화	사용 패턴 기반 모델·예산 재배분	비용 절감률, 전체 가용성, 감사 대응 시간

확산 도입을 제안할 때 근거로 삼을 성과 지표는 비용과 가용성에 집중하는 편이 설득력이 높습니다. 비용 측면에서는 팀별 예산 관리로 확보되는 지출 가시성과, 업무 성격에 맞는 모델 배치로 얻는 절감 효과를 제시할 수 있습니다. 가용성 측면에서는 폴백을 통해 특정 모델 장애가 서비스 중단으로 이어지지 않는다는 점, 즉 서비스 연속성이 확보된다는 점을 강조할 수 있습니다. 이 두 지표는 경영진이 도입 판단을 내릴 때 직접적으로 참고하는 항목이며, 파일럿에서 측정한 실제 수치로 뒷받침할 수 있습니다. [S4][S7]

지표를 측정하는 기준 시점과 방법도 함께 정해두어야 합니다. 도입 전 상태를 기준선으로 기록하고, 각 확산 단계 종료 시점에 동일한 방법으로 다시 측정해 변화를 비교하는 방식이 바람직합니다. 측정 방법이 단계마다 달라지면 성과를 일관되게 비교하기 어려우므로, 초기에 정한 측정 체계를 확산 전 과정에 동일하게 적용할 것을 권장합니다.

지금까지 이 백서는 LiteLLM이 구축형 AI 플랫폼에서 어떤 역할을 하는지를 여러 각도에서 살펴보았습니다. 다시 강조하면, LiteLLM은 여러 모델 앞에 서는 게이트웨이이며, 추론 속도를 담당

하는 추론 엔진 계층과는 역할이 다릅니다. 멀티모델을 하나의 표준 형식으로 묶고 중앙에서 거버넌스를 적용하는 이 게이트웨이 계층은, 모델의 수가 늘고 조직의 요구가 복잡해질수록 구축형 AI 플랫폼에 없어서는 안 될 필수 계층이 됩니다. MSAP.ai가 지향하는 통합 AI 플랫폼 관점에서 보면, 표준화와 거버넌스를 담당하는 이 계층은 여러 모델과 여러 팀을 하나의 운영 체계 안에 안정적으로 담아내는 토대이며, 소규모 파일럿에서 시작해 단계적으로 확산하는 접근이 그 토대를 견고하게 다지는 현실적인 경로입니다.

참고 문헌 (References)

- **[S1]** BerriAI/litellm (GitHub 저장소) — https://github.com/BerriAI/litellm
- **[S2]** LiteLLM Proxy (LLM Gateway) 공식 문서 — https://docs.litellm.ai/docs/providers/litellm_proxy
- **[S3]** litellm/LICENSE (MIT) — https://github.com/BerriAI/litellm/blob/main/LICENSE
- **[S4]** LiteLLM Enterprise 문서 — https://docs.litellm.ai/docs/enterprise
- **[S5]** LiteLLM Router / Load Balancing 문서 — https://docs.litellm.ai/docs/routing
- **[S6]** LiteLLM Fallbacks (Reliability) 문서 — https://docs.litellm.ai/docs/proxy/reliability
- **[S7]** LiteLLM Proxy Config 개요 — https://docs.litellm.ai/docs/proxy/configs
- **[S8]** LiteLLM on vLLM (vLLM 공식 문서) — https://docs.vllm.ai/en/stable/deployment/frameworks/litellm/
- **[S9]** vLLM 스루풋 최적화 파라미터 — https://medium.com/@kaige.yang0110/vllm-throughput-optimization-1-basic-of-vllm-parameters-c39ace00a519
- **[S10]** vLLM Production Deployment 2026 (FP8·멀티 GPU) — https://www.spheron.network/blog/vllm-production-deployment-2026/
- **[S11]** vLLM - Qwen 배포 가이드 — https://qwen.readthedocs.io/en/v2.5/deployment/vllm.html
- **[S12]** LiteLLM vs Portkey·Kong·Cloudflare 게이트웨이 비교 — https://contabo.com/blog/litellm-vs-ai-gateways/
- **[S13]** OpenRouter vs LiteLLM — 관리형 SaaS와 자체 호스팅 게이트웨이 비교 — https://openrouter.ai/blog/insights/openrouter-vs-litellm/

AI 플랫폼을 위한 필수 게이트웨이 LiteLLM

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