ChatGPT

다중 CSP LLM 모델 테스트 서비스 아키텍처 제안

서비스 개요와 요구사항

사용자가 AWS, GCP, Azure, OpenAI 등 **주요 클라우드 서비스 제공자(CSP)**의 **대형 언어 모델(LLM) API**를 하나의 웹 서비스에서 손쉽게 시험해볼 수 있는 오픈소스 플랫폼을 설계합니다. 이 플랫폼은 **"BYOK(Bring Your Own Key)"** 방식으로 동작하여 사용자가 자신의 API 키를 입력하고 본인 계정으로 발생한 비용을 부담합니다. 주요 요구사항은 다음과 같습니다:

- **다양한 LLM API 통합:** AWS, GCP Vertex AI, Azure OpenAI, OpenAI API, Anthropic Claude 등 모든 주요 CSP의 LLM API를 하나의 인터페이스에서 지원해야 합니다 [1] [2].
- **사용자 키 사용:** 사용자가 각 서비스의 API 키를 직접 입력하여 자신의 키로만 요청을 보내고 비용을 부담하게 합니다. 키를 안전하게 관리하고 저장하거나 필요한 경우 저장하지 않는 방식 등 **보안 고려**가 필요합니다.
- **웹 기반 프론트엔드:** 브라우저에서 동작하는 웹 UI를 제공하며, 전체 시스템을 오픈소스로 공개합니다.
- **라이브러리 + 웹 서비스 구성:** 백엔드에서 여러 LLM API 호출 로직을 **라이브러리화**하고, 이를 호출하는 **웹 서비스**(API 서버)를 둡니다. 필요하면 라이브러리만 별도로 활용할 수 있고, 웹 UI는 이 라이브러리를 활용해 서비스를 구성합니다.
- **편리한 UI/UX:** 모델 종류 선택, 파라미터 조절(예: 온도값, 최대 토큰 등), 프롬프트 입력과 **응답 확인**을 직관적으로 할 수 있는 UI를 제공합니다.
- **확장성과 구조 설명:** 프론트엔드/백엔드 분리 설계, 인증 처리 방식(API 키 관리), 새로운 모델 API를 쉽게 추가할 수 있는 **플러그인 구조** 등을 고려합니다. 코드의 주요 컴포넌트 및 전체 아키텍처를 도식화하여 설명합니다.

이하에서는 위 요구를 충족하는 시스템의 아키텍처와 구성 요소를 상세히 설계합니다.

전체 시스템 아키텍처

전체 구조는 클라이언트-서버 모델을 따르며, **웹 브라우저 UI(프론트엔드)**와 **백엔드 서버**, 그리고 각 **LLM 제공자 API**로 구성됩니다. 사용자는 브라우저에서 서비스를 열고, 원하는 LLM 모델과 파라미터를 선택한 뒤 프롬프트를 입력합니다. 이 입력과 함께 사용자의 API 키가 백엔드로 안전하게 전달되면, **백엔드**는 해당 키와 모델 정보에 맞춰 **각 CSP의 LLM API**를 호출하고 결과를 받아서 사용자에게 반환합니다. 아래 그림은 제안하는 시스템의 주요 구성과 데이터 흐름을 보여줍니다.

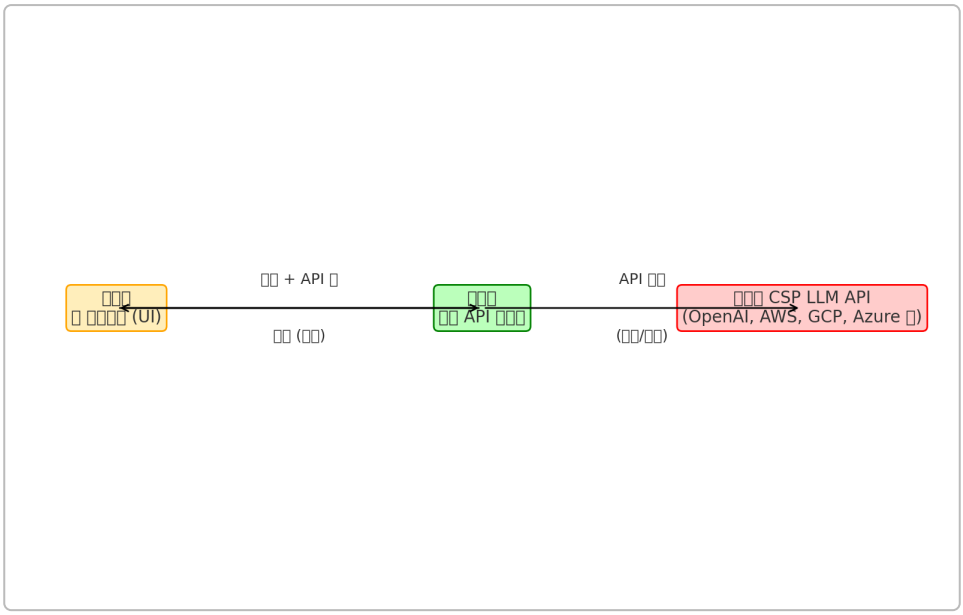

그림 1: 다양한 클라우드 LLM 제공자를 아우르는 웹 서비스 아키텍처 개념도. 사용자가 웹 UI에서 모델과 프롬프트를 입력하면, 백엔드가 해당 CSP의 LLM API를 호출하여 응답을 반환한다. 백엔드는 통합 라이브러리를 통해 여러 제공자의 API 차이를 추상화하고, 사용자 키 관리 및 보안을 담당한다.

그림 1의 흐름을 단계별로 설명하면 다음과 같습니다:

1. **사용자 입력:** 웹 UI에서 사용자가 테스트할 **모델 종류**(예: OpenAI GPT-4, Vertex PaLM, Azure GPT etc)를 선택하고 프롬프트와 필요한 파라미터를 입력합니다. 또한 처음 이용 시 각 제공자의 **API 키**를 입력하거나, 미리 설정된 키를 불러옵니다.
2. **요청 전송:** 프론트엔드는 사용자의 요청 내용을 JSON 등의 형식으로 **백엔드의 REST API** (예: `POST /api/chat`)에 전송합니다. 이 때 사용자가 입력한 API 키도 안전하게 포함됩니다. (만약 보안을 위해 키를 백엔드에 저장하지 않고 브라우저에서만 유지한다면, 요청마다 키를 보내거나 토큰화된 방식으로 전달합니다.)
3. **백엔드 처리:** 백엔드 서버에서는 요청을 받아 **모델 통합 라이브러리**를 통해 해당 **모델 제공자용 클라이언트**를 호출합니다. 예를 들어 선택된 모델이 `OpenAI의 GPT-3.5`라면 OpenAI SDK에 프롬프트를 전달하고, `Vertex AI PaLM`이면 Google 클라우드 SDK를 통해 호출하는 식입니다. **통합 라이브러리**는 각 API의 차이를 숨기고 일관된 인터페이스로 호출할 수 있게 합니다 [1]. 백엔드 서버는 사용자의 API 키를 호출에 사용하되, 자체적으로는 그 키를 보존하지 않도록 주의합니다 (자세한 보안은 아래 참조).
4. **응답 수신:** 해당 LLM 제공자의 API가 모델 응답(result)을 반환하면, 백엔드가 그 결과를 받아서 필요에 따라 공통 포맷으로 가공합니다. 예를 들어 OpenAI API와 Anthropic API의 응답 구조가 다르더라도, 이를 통합 라이브러리가 **OpenAI ChatCompletion 포맷** 등 일정한 형태로 통일할 수 있습니다 [3].
5. **결과 전송:** 백엔드는 최종 생성된 **응답 데이터**(예: 모델의 답변 텍스트 등)를 프론트엔드에 전달합니다. 프론트엔드는 이를 사용자 화면에 보여주며, 추가로 토큰 사용량이나 latency 등을 표시할 수 있습니다. 사용자는 이어서 대화를 이어가거나, 다른 모델을 선택하여 비교 테스트할 수도 있습니다.

이러한 구조를 통해, 사용자는 하나의 웹 인터페이스에서 다양한 LLM 서비스를 빠르게 교차 비교하고 활용할 수 있습니다. 또한 백엔드가 **OpenAI 호환 통일 API** 게이트웨이 역할을 하므로(예: 모든 요청을 OpenAI ChatCompletion과 유사한 형식으로 처리), 애플리케이션 레벨에서도 일관성을 유지할 수 있습니다 [4] [1].

프론트엔드 설계 (Web UI)

프론트엔드는 **React** 기반의 SPA 또는 **Next.js** 등의 프레임워크로 구현하여 사용자 경험을 향상시킵니다. 주요 UI 요소와 기능은 다음과 같습니다:

- **모델 및 제공자 선택 UI:** 화면 상단에 지원되는 모델들의 드롭다운 또는 아이콘 리스트를 제공합니다. 예를 들어 OpenAI GPT-3.5/4, Azure OpenAI GPT-4, Google PaLM 2, Anthropic Claude 등 모델을 **제공자별로 구분**하여 나열하고, 사용자가 손쉽게 전환할 수 있게 합니다. 선택하면 현재 활성화된 모델의 정보(맥스 토큰 한도 등)를 표시합니다.
- **API 키 입력 및 관리:** 사용자가 처음 특정 제공자를 선택하면 해당 API 키를 입력하도록 모달 창이나 설정 패널을 띄웁니다. 입력된 키는 **브라우저 로컬스토리지**나 세션에 암호화하여 저장해 재사용하거나, 백엔드 세션 토큰과 연계해 **서버에 저장하지 않고** 호출 시마다 전달되도록 처리합니다. 예를 들어 ChatGPT 오픈소스 UI인 TypingMind는 "본인 API 키를 사용하고, 사용량만큼만 비용 지불"하는 로컬 UI를 제공하는데 [5], 이 서비스도 유사한 BYOK 방식을 채택합니다. 키 관리 UI에서는 키를 편집/삭제하거나, 보안을 위해 화면에 노출하지 않는 등의 UX를 설계합니다.
- **프롬프트 입력 영역:** 채팅형 인터페이스로 구현하여 사용자 메시지를 입력하고 모델 응답을 대화 형태로 볼 수 있게 합니다. 또는 단일 질문-답변 형태로 구현할 수도 있지만, **대부분 LLM이 대화 지향**이므로 채팅 UI가 적합합니다. 입력 폼에는 멀티라인 텍스트박스와 "보내기" 버튼이 있으며, `Shift+Enter`로 줄바꿈, Enter로 전송이 가능하게 하는 등 채팅 사용자 경험을 고려합니다.
- **파라미터 조절 인터페이스:** 모델 온도(temperature), Top-p, 최대 토큰 수, N-best 응답 개수 등의 **생성 파라미터**를 사용자가 조절할 수 있도록 슬라이더나 숫자 입력 필드를 제공합니다. 각 파라미터에 툴팁을 달아 의미를 설명하고, 모델 기본값을 표시하여 사용자가 조정 여부를 인지하게 합니다. 특정 제공자의 API가 지원하지 않는 파라미터는 UI에서 비활성화하거나 숨겨, 혼란을 줄입니다.
- **응답 표시 영역:** 모델의 답변을 보기 좋게 표시합니다. 마크다운 포맷을 지원하는 모델 응답일 경우 렌더링하여 보여주고, 코드 블록이 있으면 Syntax Highlight를 적용합니다. 멀티턴 대화 시 사용자 메시지와 모델 응답을 교차로 말풍선 스타일로 표시하고 각 턴의 모델 종류도 표시하면, **다양한 모델의 응답 비교**에도 활용할 수 있습니다.
- **추가 UX 요소:** 응답 생성 중에는 로딩 애니메이션을 표시하고, 스트리밍 지원 시 토큰이 들어오는 대로 실시간 출력하여 인터랙티브함을 높입니다. 또 이전에 실행한 요청과 응답을 좌측 사이드바에 **기록(History)** 형태로 남겨 두어, 사용자가 나중에 동일 프롬프트를 다른 모델로 다시 실행하거나 결과를 재확인할 수 있게 합니다. (History 데이터도 브라우저에 저장하거나 백엔드 DB에 저장하되 사용자별로 구분해야 합니다.)

프론트엔드는 이러한 UI 요소를 구성하며 **REST API 호출**을 통해 백엔드와 통신합니다. 보안상 **API 키나 민감 정보가 프론트엔드 코드에 하드코딩되지 않도록** 주의하고, 입력 받은 키도 가능하면 메모리에만 두거나 암호화하여 저장합니다. 전체 프론트엔드는 오픈소스로 공개하여, 다른 개발자가 자신의 API 키로 이 UI를 활용하거나, 필요하면 커스터마이징할 수 있도록 합니다.

백엔드 설계 (API 서버 및 통합 라이브러리)

백엔드는 **경량화된 API 서버**와, 다수의 LLM API를 아우르는 **통합 라이브러리(core)**로 구성됩니다. API 서버는 HTTP 요청을 받아서 적절한 라이브러리 호출을 수행하고 결과를 반환하는 역할을 합니다. 주요 설계 포인트는 다음과 같습니다:

- **기술 스택:** 백엔드 API 서버는 선택된 언어나 프레임워크로 구현할 수 있습니다. 예를 들어 **Node.js(Express)**나 **FastAPI(Python)**, 또는 **Go** 등을 선택할 수 있습니다. 중요한 것은 동시에 여러 API 호출을 효율적으로 처리할 **비동기 처리 능력**과, 다양한 외부 SDK를 다룰 수 있는 **호환성**입니다. Python의 경우 OpenAI, Google Cloud, AWS SDK 등이 잘 제공되고 있어 라이브러리 구현에 용이하며, Node 역시 대응 패키지가 있습니다.
- **통합 API 인터페이스:** 백엔드의 핵심은 각기 다른 LLM 제공자들의 API를 일관성 있게 호출하는 **통합 인터페이스**입니다. 이는 자체 라이브러리 형태로 구현할 수 있으며, 함수 호출 한 번으로 `provider`와 `model`, `prompt`, `parameters`, `api_key` 등을 넘기면 알아서 해당 제공자의 API를 호출하고 결과를 반환하도

록 합니다. 예를 들어 Mozilla의 any-llm 오픈소스 프로젝트는 “인기 있는 여러 LLM 제공자를 하나의 Python SDK로 호출”할 수 있도록 설계되었는데, 구성 파라미터 하나만 바꾸면 **다른 제공자의 모델로 전환**이 가능할 정도의 통일된 인터페이스를 제공합니다 [6]. 우리 시스템도 이러한 원리를 적용하여, **OpenAI의 ChatCompletion 형식**을 내부 표준으로 삼고 다른 API 호출을 래핑합니다 [3]. 통합 라이브러리는 각 제공자의 **공식 SDK**를 우선 활용하여 구현함으로써 유지보수성을 높입니다 [7]. (공식 SDK가 없는 경우 REST API를 직접 호출)

- **LLM 제공자 플러그인 구조:** 새로운 모델이나 제공자를 추가하기 쉽도록, 백엔드 라이브러리를 **플러그인 아키텍처**로 설계합니다. 예를 들어 `BaseLLMProvider` 라는 추상 클래스(또는 인터페이스)에 `send_request(prompt, params, api_key)` 와 같은 메서드를 정의하고, 각 공급자별로 이를 상속 구현합니다. `OpenAIProvider` 는 OpenAI SDK 호출 로직을, `AzureProvider` 는 Azure OpenAI용 호출 로직을, `GoogleVertexProvider` 는 Vertex AI 호출을 각각 구현하는 식입니다. 이렇게 하면 새로운 제공자의 API 키 형태나 호출 URL, 파라미터 차이만 해당 클래스에 캡슐화되고, **공통 인터페이스로 제어**할 수 있습니다.
- 예를 들어 AWS의 경우 Amazon Bedrock이나 SageMaker 엔드포인트 호출 로직을 `AWSProvider` 로 구현하고, 필요한 인증 방식 (액세스 키/시크릿키, 리전 등 설정)도 이 안에서 처리합니다. GCP Vertex AI의 경우 OAuth2 또는 API 키 인증과 프로젝트 정보 등을 세팅하는 로직을 `GoogleProvider` 에 구현합니다. 이때 **공식 SDK**(예: `boto3` , `google-cloud-aiplatform` , `openai` 패키지 등)를 사용하면 내부 구현이 간결해지고 최신 API 변화에도 대응하기 쉽습니다 [7].
- 이러한 플러그인 구조를 통해 **확장성**이 확보됩니다. 새로운 AI 스타트업의 모델 API가 나와도, 해당 SDK를 불러와 플러그인 클래스를 하나 추가하면 이 서비스에 통합시킬 수 있습니다. (오픈소스 기여자들이 플러그인 클래스를 추가해 Pull Request를 보내는 형태로 발전 가능)
- **REST API 디자인:** 백엔드는 프론트엔드와 통신하기 위한 RESTful API 엔드포인트를 설계합니다. 예를 들어 `POST /api/v1/generate` 또는 `/api/chat` 로 프롬프트와 설정을 받아 응답을 돌려주는 단일 endpoint를 둘 수 있습니다. 또는 채팅 대화 유지를 위해 `POST /api/chat` (새 질문), `GET /api/history` (대화 내용 조회) 등의 엔드포인트를 추가할 수도 있습니다. 단순한 구조에서는 한 엔드포인트로 요청/응답만 처리하고 상태는 프론트엔드가 관리하도록 하면 구현이 쉬우나, 멀티유저나 세션을 고려하면 백엔드에서 세션ID별로 대화 이력을 관리하게 할 수도 있습니다. 이 경우 인메모리 캐시나 DB를 활용합니다.
- **스트리밍 및 양방향 통신:** 사용자 경험 향상을 위해 백엔드가 OpenAI의 SSE(Stream)응답이나 WebSocket을 통한 **스트리밍** 생성을 지원할 수 있습니다. 구현 시 `Content-Type: text/event-stream` 형태로 토큰을 실시간 전송하거나, WebSocket 채널을 열어 모델이 토큰 생성할 때마다 프론트엔드에 push하는 방식을 고려합니다. 이를 위해 백엔드 프레임워크가 비동기 처리를 지원해야 하며, 각 플러그인에서도 스트리밍 모드를 제공하는 SDK 활용이 필요합니다 (예: OpenAI SDK의 stream 옵션).
- **오픈소스와 모듈화:** 백엔드 코드는 오픈소스로 공개되므로 **모듈 구조**와 **코딩 스타일**을 명확히 합니다. 앞서 언급한 통합 라이브러리는 별도 패키지(`providers/` 디렉토리 등)에 넣고, 웹 API 관련 코드는 `server/` 디렉토리에 구성하는 식입니다. 실제 유사한 오픈소스 예인 LLM Gateway 프로젝트의 코드 구조를 보면, Next.js 기반 프론트엔드(`apps/ui`)와 Hono (Fastify 유사) 기반 백엔드 API(`apps/api`), 그리고 모델/제공자 정의 모듈(`packages/models`) 등으로 레포지토리를 구성하고 있습니다 [8]. 이러한 분리는 코드 이해와 유지보수에 용이하고, 라이브러리 부분을 별도로 배포하거나 테스트하기에도 좋습니다.

다음 표는 제안 시스템의 주요 구성 요소를 기술 스택과 함께 정리한 것입니다:

구성 요소	예시 기술 스택	주요 역할 및 기능
프론트엔드 (Web UI)	React + Next.js, Vue.js 등	사용자 입력 화면. 모델 선택, 프롬프트 입력, 파라미터 조절 UI 제공. API 키 입력 및 로컬 저장. 모델 응답 표시 (채팅/텍스트 렌더링).
백엔드 API 서버	Node/Express 또는 Python/FastAPI 등	REST API 엔드포인트 제공. 프론트엔드 요청 수신 및 처리. 통합 라이브러리를 호출해 LLM API 요청 대행. 다중 사용자 시 세션 관리 및 로그 기록. (필요 시 WebSocket/SSE로 스트림 지원)

구성 요소	예시 기술 스택	주요 역할 및 기능
LLM 통합 라이브러리	Python 패키지 또는 Node 모듈	여러 LLM 제공자의 SDK/API를 추상화한 코어 모듈. 일관된 함수 호출로 다양한 모델에 질의 가능 [1]. OpenAI 호환 형식으로 입력/출력 표준화.
모델 제공자 플러그인	각 제공자별 SDK (boto3, openai, etc) 사용	통합 라이브러리 내의 확장 모듈. 제공자별 API 호출 방법 구현 (엔드포인트 URL, 인증 방식, 파라미터 변환 등). 새로운 제공자 추가 시 해당 플러그인 모듈 추가로 지원.
데이터베이스(선택사항)	PostgreSQL, MongoDB 등	사용자 계정 및 API 키를 저장(암호화)하거나, 대화 기록 및 사용 로그 저장용. (필수는 아니나, 멀티테넌시나 추후 분석을 위해 도입 가능)

백엔드는 API 키 등의 민감정보를 다루므로 **보안**에 특별히 주의해야 합니다. 다음 섹션에서 키 관리와 전송에 대한 보안 고려사항을 다룹니다.

API 키 보안 및 관리

사용자의 API 키를 다루는 부분은 이 서비스의 **보안 핵심 요소**입니다. 잘못하면 키가 유출되어 악용되거나, 타인이 비용을 유발할 수 있으므로 다음과 같은 원칙을 적용합니다:

- **서버 미보관 (Stateless) 원칙:** 가장 안전한 방법은 **API 키를 서버에 아예 저장하지 않는 것**입니다. 사용자가 요청을 보낼 때 키를 함께 보내도록 하고, 백엔드는 그 키를 사용해 외부 API를 호출하지만 **그 이후로 저장하지 않는** 방식입니다. 이렇게 하면 서버측에 키 유출 위험이 거의 없으며, 각 요청은 사용자의 세션에 의존합니다. 다만 사용 편의를 위해 사용자가 키를 자주 입력하지 않도록, 프론트엔드에서 **브라우저 내 저장**(로컬스토리지에 암호화 저장 등)을 활용할 수 있습니다. 예를 들어 어떤 오픈소스 LLM UI들은 사용자의 OpenAI 키를 브라우저에 저장하고 호출 시마다 보내도록 하여, 서버는 키를 기억하지 않습니다 [5].
- **암호화 저장:** 만약 서버 측에 키를 저장해야 하는 요구사항이 있다면 (예: 여러 번 세션을 이어서 쓰거나, 여러 기기에서 동일 계정으로 접속), **키를 반드시 암호화하여 저장**합니다. 방법으로는 데이터베이스에 키를 넣을 때 대칭키로 암호화하고, 그 대칭키는 환경변수 혹은 키관리시스템(KMS)에 보관합니다. 데이터베이스 자체도 접근 권한을 최소화하고, 키 저장 전용 테이블에는 엄격한 ACL을 적용합니다.
- 일부 아키텍처는 **제로 knowledge** 방식을 취하기도 합니다. 예를 들어 Moz://a가 제안한 any-llm 플랫폼에서는 **클라이언트 측에서 공개/비밀키 쌍을 만들고, API 키를 브라우저에서 암호화한 후 서버는 암호문만 저장**하는 구조를 사용합니다 [9]. 서버가 암호화된 키를 받더라도 복호화할 수 없고, 오직 사용자의 브라우저(비밀키 소유)만 복호화하여 호출 순간에 키를 쓸 수 있게 하는 것입니다. 이러한 **엔드-투-엔드 암호화** 접근은 매우 높은 보안을 제공하지만 구현이 복잡하므로, 기본 구현에서는 서버 미보관 원칙 + 필요시 암호화 저장 정도로도 충분합니다.
- **환경변수 활용: 절대로 키를 소스코드에 하드코딩하지 않습니다** [10]**.** 이는 기본 보안 수칙으로, 개발 단계부터 운영까지 모든 비밀정보는 깃 저장소 등에 노출되지 않도록 합니다. 각 제공자별 API 키 이름(예: `OPENAI_API_KEY`)을 정하고, 백엔드 서버는 환경변수나 Vault에서 키를 불러쓰도록 할 수 있습니다. 다만 이 서비스는 사용자 각각의 키를 받아야 하므로, **단일 키로 모든 요청을 처리하지 않도록** 해야 합니다. (만약 운영자가 공용 키를 제공하면 비용 부담 및 남용 문제가 생김) 따라서 환경변수에는 운영상 필요한 값들만 넣고, 사용자 키는 위에서 언급한 세션이나 암호화 DB 방식을 사용합니다. 개발용으로 키를 넣어 테스트할 경우에도 `.env` 파일로 분리하고 커밋하지 않는 등 기본 수칙을 지킵니다 [11].
- **전송 구간 암호화:** 프론트엔드와 백엔드 통신은 반드시 **HTTPS**를 사용하여 TLS로 암호화합니다. 이는 기본이지만 중요하며, WebSocket 사용 시에도 WSS (TLS) 프로토콜로 통신해야 합니다. 또한 HTTP 헤더나 URL 파라미터로 키가 노출되지 않게 하고, **요청 본문에 포함**하도록 합니다 (HTTPS로 암호화되므로 안전 전달).
- **권한 분리와 제한:** 가능하다면 API 키 자체에 제한을 두는 것도 안전에 도움이 됩니다. 일부 제공자는 서브키 발급이나 IP 제한, 호출 제한 설정이 가능합니다. 예를 들어 OpenAI는 조직 단위 API 키나 제한된 권한 키 발급은 지원되지 않지만, **Google Cloud는 서비스 계정 키에 역할 제한**을 걸 수 있습니다. AWS 역시 IAM을 통해

Bedrock 호출 전용 사용자를 만들고 해당 키만 쓰도록 할 수 있습니다. 사용자가 이런 설정을 할 수 있다면 권장하여, **최소 권한의 원칙**을 지키도록 안내합니다 [11].

- **로깅 주의:** 백엔드에서 **로그를 남길 때 API 키나 프롬프트 등 민감정보를 기록하지 않습니다.** 오류 발생 시에도 키 값은 마스킹하거나 제외하고 로깅해야 합니다. 또 데이터베이스에 대화 내용이나 로그를 저장할 때도, 사용자의 프롬프트/응답에 민감한 개인정보가 없도록 권고하고, 로그 접근 권한을 제한합니다. (GDPR 등 개인정보 규제를 준수)
- **사용량 모니터링:** 각 키별로 API 호출 사용량이나 비용을 추적하여, **이상 패턴**이 감지되면 사용자에게 알리거나 차단하는 것도 고려합니다. 예를 들어 특정 사용자의 키로 단기간 과도한 요청이 발생하면 UI상 경고를 주고, 혹시 키가 노출된 것은 아닌지 안내합니다. 이는 보안이라기보다 비용통제 및 이상징후 대응이지만, 궁극적으로 사용자 자산(키)을 보호하는 차원입니다.

정리하면, **"사용자 키를 서버가 몰라도 서비스를 제공"**하는 것이 이상적 목표입니다 [12]. 현실적으로는 일시적으로 알고 사용할 수밖에 없지만, 저장하지 않고, 노출 없도록 하는 정책을 통해 신뢰성을 확보합니다. 이러한 보안 조치는 오픈소스 프로젝트로서 투명하게 문서화하여, 사용자들이 안심하고 자신의 키를 입력할 수 있도록 합니다.

플러그인 확장 구조

앞서 백엔드 설계에서 언급한 플러그인 구조를 좀더 상세히 설명하면 다음과 같습니다. 시스템은 **새로운 모델/API 제공자를 쉽게 추가**할 수 있도록 모듈화되어야 합니다. 이를 위해 고려하는 사항:

- **모델/제공자 정의 모듈:** 예를 들어 `providers/` 디렉토리 아래에 각 제공자별 모듈이 위치합니다. `providers/openai.py`, `providers/azure.py`, `providers/google.py`, `providers/aws.py` 등 파일을 두고, 각각 해당 API 호출 기능을 캡슐화합니다. 만약 Go나 Node로 구현한다면 파일 구조는 다르겠지만, 개념적으로 **한 제공자당 하나의 드라이버**가 있다고 보면 됩니다. 이들 모듈은 공통된 인터페이스를 구현하므로, 백엔드 로직은 `provider_name`으로 모듈을 선택하고 동일한 메서드를 호출하기만 하면 됩니다. 새로운 모듈을 추가하면 설정파일에 등록하거나, 특정 폴더 내 클래스들을 자동으로 탐색해 로드하는 기능을 넣어 **플러그인 자동인식**을 구현할 수 있습니다.
- **공통 데이터 모델:** 서로 다른 API라도 입력과 출력의 데이터 구조를 통일해야 사용 및 확장이 쉽습니다. 예를 들어 **프롬프트 메시지 구조**를 OpenAI ChatCompletion의 포맷(`{"role": "user", "content": "..."}`)으로 통일하고, 출력도 OpenAI의 응답 객체 구조(`choices`, `usage` 등)를 따르게 하면 프론트엔드 처리 로직도 단순해집니다 [3]. 플러그인 개발자는 자신의 API 결과를 이 공통 구조로 **매핑**해주기만 하면 됩니다. 이러한 표준화 작업은 통합 라이브러리 레이어에서 처리하되, 각 플러그인이 헬퍼 함수를 통해 쉽게 변환하도록 돕습니다.
- **예외 처리와 재시도:** 각 플러그인은 API 오류나 네트워크 예외를 표준화된 예외 형태로 raise하여 상위에서 일괄 처리할 수 있게 합니다. 또한 특정 모델 호출이 실패했을 때 fallback 메커니즘도 고려할 수 있습니다. (예: GPT-4 호출 실패 시 GPT-3.5로 자동 대체 등). 실제 LiteLLM 같은 프록시는 **모델 fallback과 라우팅** 기능을 넣어, 비용이나 속도에 따라 다른 모델로 보내기도 합니다 [13] [14]. 우리 서비스에서도 확장으로 그런 기능을 고려하여, 플러그인 레이어에서 **우선순위**나 **대체 목록**을 지원할 수 있습니다. 다만 초기 구현에서는 명시적으로 선택된 모델만 호출합니다.
- **버전 관리와 테스트:** 오픈소스 프로젝트이므로 여러 기여자가 플러그인을 추가/수정할 수 있습니다. 따라서 각 플러그인에 대한 **단위 테스트**를 구축해두고, 실제 API 호출을 모의(mock)하거나 작은 쿼리로 테스트하여 동작을 검증합니다. 특히 키 유효성, 권한 부족, 쿼ота 초과 등의 경우를 시뮬레이션해 잘 처리되는지 확인합니다. 또한 새로운 API 버전이 나왔을 때 업그레이드가 필요하므로, 예를 들어 OpenAI API v2가 나오면 `openai_v2.py` 플러그인을 추가하고 설정으로 버전을 선택할 수 있게 하는 등 **버전별 플러그인** 구조도 유연하게 설계합니다.
- **문서화:** 새로운 제공자 추가 방법에 대해 기여자용 가이드를 작성합니다. 예를 들어 "새로운 LLM API를 추가하려면 `providers/` 폴더에 클래스 추가 + `providers/__init__.py`에 등록 + 환경변수 키 이름 정의" 등의 절차를 문서화하여, 외부 개발자들도 쉽게 확장에 참여하도록 합니다.

요약하면, 플러그인 구조는 **전략 패턴**을 적용한 모듈 설계로 볼 수 있으며, 이를 통해 이 시스템은 향후 등장할 어떤 LLM 서비스든 연결만 하면 테스트해볼 수 있는 **확장형 테스트베드**가 될 것입니다.

구성 요소 및 코드 구조

마지막으로, 이 서비스의 전체 코드베이스 구조를 아키텍처 관점에서 정리합니다. 오픈소스로 공개될 것이므로 폴더 구성과 책임 분리가 명확해야 합니다. 예시로 다음과 같은 디렉토리 구조를 갖출 수 있습니다:

```
📦 LLMTester/   (프로젝트 루트)
├─ ☒frontend/          # 프론트엔드 React/Next.js 애플리케이션
│   ├─ pages/, components/, ...
│   └─ ... (UI 관련 코드 및 정적 자원)
├─ ☒backend/           # 백엔드 API 서버 애플리케이션
│   ├─ app.py / index.js     # 엔트리포인트 (Flask/Express 서버 초기화)
│   ├─ routes/              # 엔드포인트별 컨트롤러 (예: generate.py 또는 chat.js)
│   └── ...
├─ ☒llm_providers/    # LLM 통합 라이브러리 모듈들
│   ├─ __init__.py         # 통합 인터페이스 (함수 send_request 등)
│   ├─ base.py             # 추상 클래스 BaseProvider 정의
│   ├─ openai_provider.py  # OpenAI API 구현 [1]
│   ├─ azure_provider.py   # Azure OpenAI 구현
│   ├─ gcp_provider.py     # GCP Vertex AI 구현
│   ├─ aws_provider.py     # AWS Bedrock/SageMaker 구현
│   └── ... (기타 제공자 구현 모듈)
├─ ☒plugins/          # (선택) 플러그인 확장 모듈 위치 - providers와 동일 역할, 분리 가능
├─ ☒models/         # (선택) 프론트엔드+백엔드 공용으로 쓰는 데이터 모델 정의 (Pydantic
schemas 등)
├─ ☒db/             # (선택) 데이터베이스 관련 (ORM 모델 정의 등)
├─ ☒tests/          # 테스트 코드
└─ ☒README.md       # 프로젝트 설명과 실행방법
```

위는 한 예시이며, 실제 구현에 따라 달라질 수 있습니다. 예를 들어 LLM Gateway 오픈소스는 `apps/ui` (Next.js 대시보드), `apps/api` (백엔드 API), `apps/gateway` (LLM 프록시 게이트웨이), `packages/models` (모델 및 제공자 정의) 등의 구조를 갖고 있습니다 [8]. 중요한 점은 **프론트엔드**와 **백엔드/라이브러리**의 분리가 명확하고, **모델/제공자 정의**가 한 곳에 모여 있다는 것입니다.

코드 구조 설계시 **관심사의 분리(SoC)**를 철저히 합니다. 프론트엔드는 UI/UX에 집중하고, 백엔드는 라우팅과 보안/로그에 집중, LLM 통합 코드는 API 호출 로직에 집중하도록 계층화합니다. 이런 구조는 확장이나 유지보수뿐만 아니라, 오픈소스 협업 시 여러 개발자가 동시에 각각의 부분을 작업하기에도 유리합니다.

또한, **아키텍처 다이어그램** 등 문서를 리포지토리에 포함시켜 개발자들이 전체 그림을 쉽게 이해하도록 합니다. 시스템의 구성 요소 (UI, API Server, Provider Plugins, DB 등)과 이들 간의 관계를 도식화한 그림은 README 등에 포함할 예정입니다.

결론

이번에 제안한 구조는 다양한 클라우드 제공자의 LLM 서비스를 **단일 웹 플랫폼**에서 시험하고 활용할 수 있도록 고안되었습니다. 사용자는 하나의 UI에서 여러 모델을 손쉽게 바꿔가며 프롬프트를 테스트하고 응답을 비교할 수 있고, 각자 **자신의 API 키로 비용을 투명하게 관리**할 수 있습니다. 이러한 접근은 한 업체에 종속되지 않고 최적의 모델을 선택하게 해 주며, 실제 많은 기업/개발자들이 겪는 **LLM 통합의 복잡성**을 줄여줍니다 [15] [16].

설계의 핵심 포인트는 **유연성과 보안**입니다. 모든 주요 API를 아우르는 **유니버설 API 게이트웨이**로서 동작하면서도 [4], 사용자 키를 안전하게 다루는 것에 중점을 두었습니다. 특히 **오픈소스**로 공개됨에 따라, 투명한 코드와 모듈화된 구조로 커뮤니티의 협업과 검증을 받을 수 있습니다. 이는 신뢰성 향상과 기능 확장의 선순환을 가져올 것입니다.

마지막으로, 이 아키텍처는 필요에 따라 **기업 환경으로 확장**할 수도 있습니다. 예를 들어 팀별 키 관리, 요청량 모니터링, 과금 대시보드 등의 엔터프라이즈 기능을 추가하면 조직 내 **LLM Ops 플랫폼**으로 발전시킬 수도 있습니다 [17] [18]. 반대로 개인 개발자는 불필요한 부분을 비활성화하고 경량으로 사용 가능하도록, 설계 단계부터 **모듈별 on/off**가 가능하게 할 것입니다.

以上과 같은 구조와 설계를 통해 요구된 기능을 충족하는 **다중 CSP LLM 테스트 서비스**를 구현할 수 있을 것으로 판단됩니다. 이 보고서를 바탕으로 프로토타입을 개발하고 실제 모델 API들을 연동함으로써, 사용자들이 하나의 웹 UI에서 다양한 AI 모델을 자유롭게 실험해볼 수 있는 환경을 실현할 수 있을 것입니다.

참고 문헌 및 소스: 본 제안은 최신 LLM 통합 오픈소스 동향 [1] [8]과 클라우드 벤더 권고사항 [19] [18], 그리고 보안 모범사례 [10]를 바탕으로 작성되었습니다. 주요 참고 자료로 LiteLLM, LLM Gateway, any-llm 등 관련 프로젝트의 구조와 원칙을 참고하였으며, 적절한 인용을 통해 신뢰성을 확보했습니다.

---

[1] [2] [10] [11] [13] [14] [17] [18] LiteLLM: Access 100+ AI Models Through a Single API (Free & Self-Hosted) | by Manu Francis | Medium
https://medium.com/@manuedavakandam/litellm-access-100-ai-models-through-a-single-api-free-self-hosted-b6f7be7a51dc

[3] [6] [7] Introducing any-llm: A unified API to access any LLM provider
https://blog.mozilla.ai/introducing-any-llm-a-unified-api-to-access-any-llm-provider/

[4] [8] GitHub - theopenco/llmgateway: Route, manage, and analyze your LLM requests across multiple providers with a unified API interface.
https://github.com/theopenco/llmgateway

[5] TypingMind — LLM Frontend Chat UI for AI models
https://www.typingmind.com/

[9] [12] We designed a zero-knowledge architecture for multi-LLM API key management (looking for feedback) : r/LocalLLM
https://www.reddit.com/r/LocalLLM/comments/1pecn7e/we_designed_a_zeroknowledge_architecture_for/

[15] [16] OpenRouter AI: The Ultimate 2025 Guide for Developers & Power Users
https://skywork.ai/skypage/en/OpenRouter-AI-The-Ultimate-2025-Guide-for-Developers-Power-Users/1974364239776378880

[19] Guidance for Multi-Provider Generative AI Gateway on AWS
https://aws.amazon.com/solutions/guidance/multi-provider-generative-ai-gateway-on-aws/