How to Connect an App to Multiple AI Models with One API

Building an app that talks to more than one AI model is becoming the norm rather than the exception. A developer might want GPT-4 for long-form reasoning, a fast open-source model for chat, and a separate image model for generating visuals. The natural instinct is to wire each provider separately: one SDK for OpenAI, another for Anthropic, a third for a hosted open-source endpoint, and so on. That approach works, but it adds complexity fast. Every provider has its own authentication method, its own request schema, its own rate limits, and its own way of returning errors. Maintaining all those integrations can become a serious bottleneck.

A unified AI API solves this problem by providing a single interface that sits between your app and the underlying model providers. Instead of writing separate code paths for each model, you write one set of calls and let the unified layer handle the translation. The result is cleaner code, fewer points of failure, and the freedom to swap models without rewriting your application logic.

This article walks through the steps to connect an app to multiple AI models through a unified API. It covers what to look for when choosing a provider, how to set up authentication, how to structure requests, and how to handle responses. Along the way it highlights the practical trade-offs you will encounter.

Choose a unified API provider that matches your needs

The first step is selecting the service that will sit between your app and the model providers. Not all unified APIs are the same. Some support only text models. Others handle images, video, and music as well. Some charge a flat monthly fee; others charge per request. Some require a subscription; others do not.

A good starting point is to look at the compatibility layer. If you already have code written for OpenAI's API, you want a provider that accepts the same request format. That way you can switch with minimal code changes. The same applies if you rely on Anthropic's message format or a specific SDK. The less you have to change in your existing code, the faster you can move.

Pricing models vary, and it pays to understand how they work before you commit. Some unified APIs add a markup on top of the provider's list price. Others offer rates below list prices by aggregating demand or by using reseller agreements. A few operate without any subscription or per-request fee, which can be attractive for projects with unpredictable traffic.

Another factor is the breadth of models available. A unified API that covers only a handful of large language models may not serve you well if you later need image generation or audio transcription. Look for a service that includes the categories you expect to use now and those you might need in the future.

Obtain your base URL and API key

Once you have chosen a provider, you need two things: the base URL where your requests will be sent, and an API key that authenticates those requests. The base URL is the endpoint that replaces the standard OpenAI or Anthropic endpoint in your client library. The API key is a unique identifier that the provider uses to track usage and enforce access controls.

Most unified APIs let you generate an API key from a dashboard after you create an account. Treat this key like a password. Do not hard-code it into your source code or commit it to version control. Use environment variables or a secrets manager instead. If a key is compromised, revoke it from the dashboard and generate a new one.

The base URL is equally important. It determines where your requests land. A typical pattern is to replace https://api.openai.com/v1 with the unified provider's URL, such as https://api.unified-provider.com/v1. The exact format depends on the provider, so check their documentation.

Watch out for differences in the URL path. Some providers expect the same path structure as OpenAI, so your existing code works with a simple URL swap. Others use a different path convention, which may require a small adjustment in how you construct your requests.

Structure your REST requests using the unified schema

With the base URL and API key in hand, you can start sending requests. The beauty of a unified API is that the request schema stays consistent across models. You send the same fields - model, messages, max_tokens, temperature - regardless of whether the underlying provider is OpenAI, Anthropic, or a hosted open-source model.

The key difference is the model identifier. In a direct integration, you might use gpt-4o or claude-sonnet-4-20250515. In a unified API, the model identifier is often a normalized string that the provider maps to the actual model. For example, a unified API might accept claude-4-sonnet and internally translate it to the correct Anthropic identifier. Some providers let you use the original identifiers directly. Check the documentation to see which convention is supported.

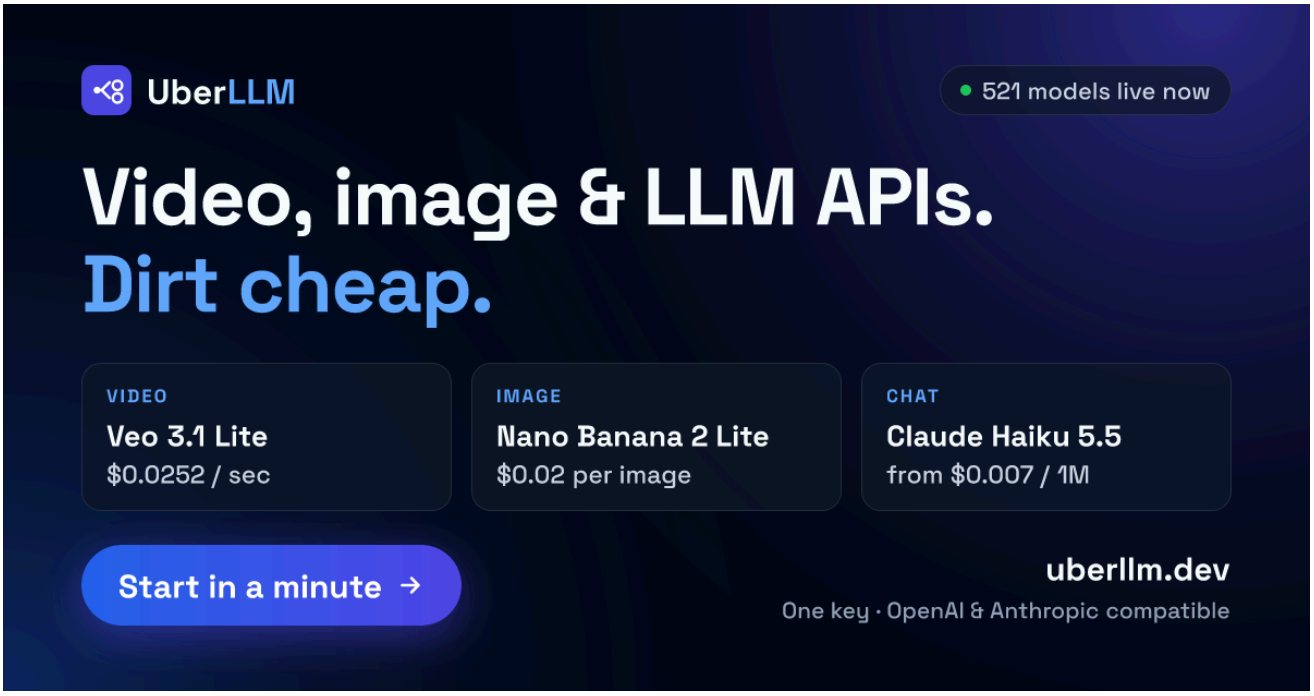
Authentication is handled through an HTTP header, typically Authorization: Bearer YOUR_API_KEY. If you are using an SDK, the library usually attaches this header automatically once you set the API key. If you are sending raw HTTP requests, you need to add the header yourself.

The request body for a chat completion might look like this:

{ "model": "claude-4-sonnet", "messages": [{"role": "system", "content": "You are a helpful assistant."}, {"role": "user", "content": "Explain quantum computing in simple terms."}], "max_tokens": 500 }
Notice that the structure is identical to what you would send to OpenAI's API. That consistency is the main advantage. You can switch from a text model to an image model by changing the model field and adjusting the message content. The surrounding code stays the same.

Handle responses and errors consistently

The response format from a unified API is designed to be predictable. A successful response returns the model's output in a standard structure, usually with fields for the generated text, token usage, and the model that was used. You can parse this response the same way for every model.



Error handling is where a good unified API earns its keep. When a model is overloaded, a direct integration might return a 429 status code with a rate-limit message. A different provider might return a 503. A unified API normalizes these into a consistent error format, so your error-handling code does not need to account for every provider's quirks.

That said, not all unified APIs handle errors equally well. Some pass through the raw error from the underlying provider, which defeats the purpose. Others provide a clean error object with a status code, a message, and a type. When evaluating a provider, test how it responds to

common failure scenarios: an invalid API key, an unknown model identifier, a request that exceeds the token limit, and a temporary server error.

Rate limiting is another area where a unified API can simplify your life. Instead of tracking separate rate limits for each provider, you work within a single rate-limit policy. The provider manages the underlying limits and queues or retries requests as needed. Some providers offer a queue mechanism that lets you send requests without worrying about hitting limits on the backend.

Choose your SDK or write raw HTTP calls

Most unified APIs offer an SDK for popular programming languages. If you are using Python, JavaScript, or Go, an SDK can save you time by handling authentication, request serialization, and response parsing. The SDK typically mirrors the official OpenAI or Anthropic client, so you can switch by changing the import and the base URL.

If an SDK is not available for your language, or if you prefer minimal dependencies, you can send raw HTTP requests. The REST interface is straightforward, and the response is plain JSON. A simple function that posts to the chat endpoint and returns the parsed response is all you need.

One thing to watch for is streaming. If your app shows responses as they are generated, you need streaming support. Check whether the unified API supports server-sent events (SSE) in the same way as the original provider. Some APIs stream the response token by token, while others buffer the whole response and send it at once. For a chat application, streaming makes a big difference in perceived latency.

Test with a single model, then expand

Once you have the basic integration working, test it with one model. Send a simple prompt, check that the response comes back correctly, and verify that error handling works. This is the time to catch issues like a wrong base URL, an incorrect model identifier, or a missing header.

After the single-model test passes, add a second model. Change only the model identifier in the request and see if the response is still parsed correctly. If the unified API normalizes the response format, you should see the same structure regardless of the underlying model. If you get different fields for different models, you may need to adjust your parsing logic.

Gradually add more models, including image generation and audio models if the API supports them. Each new category may introduce a different request schema, but within the same category, the schema remains consistent. For example, image generation requests might use a

prompt field and a size field, while chat requests use messages. The unified API keeps the schema stable for each endpoint.

Services like UberLLM provide exactly this kind of unified layer, accepting requests in the same format as OpenAI and Anthropic while giving access to a wide range of models through competing sellers. That means you can write your code once and use it across many models without worrying about the underlying provider's details.

Monitor usage and performance

After your app is running with multiple models, monitor how the unified API performs under load. Track response times for different models. Some models are inherently slower than others, but a well-designed unified API should not add noticeable overhead beyond the underlying provider's latency. If you see delays that are not explained by the model itself, investigate whether the provider is queueing requests or applying additional processing.

Token usage is another metric to watch. Most unified APIs include token counts in the response. Use those counts to understand your costs and to optimize your prompts. If one model consistently uses more tokens than another for the same task, consider whether the trade-off in quality is worth the extra cost.

Finally, keep an eye on error rates. A sudden spike in 500 errors might indicate an issue with a particular provider. A unified API can route around a failing provider if you configure it to fall back to an alternative model. Some providers let you set priority lists or fallback chains. That kind of resilience is one of the strongest reasons to use a unified API in the first place.

Connecting an app to multiple AI models through a single API is a practical way to reduce complexity, speed up development, and keep your options open. By choosing a provider that normalizes authentication, request schemas, and error handling, you can focus on building features rather than maintaining integrations. The steps outlined here give you a clear path to get started: pick a provider, set up your credentials, structure your requests consistently, handle errors gracefully, and expand your model selection as you go.