ChatGPT

Unified LLM API for Node.js: Multi-Provider Integration Guide

Overview and Goals

This guide outlines how to implement a unified TypeScript/Node.js API client that abstracts over multiple Large Language Model (LLM) providers. The aim is to provide a single interface (`generateChat`, `callFunction`, `streamOutput`, etc.) that can talk to **OpenAI**, **Anthropic**, **Google (Gemini/PaLM)**, **Mistral**, **Cohere**, and **Meta's LLaMA** (via third-party APIs). We'll break down each provider's API specifics (endpoints, request format, auth), compare key differences (chat roles, function calling, streaming, multimodal support, model discovery), and design a normalized abstraction layer. Code examples in TypeScript are included to demonstrate usage, along with best practices for error handling, rate limiting, and monitoring. Finally, we discuss how to expose this abstraction as a serverless service for external consumption.

Provider API Breakdown

OpenAI (ChatGPT, GPT-4 Series)

API Endpoint & Authentication: OpenAI's primary endpoint for chat is `POST https://api.openai.com/v1/chat/completions`. Authentication is via API key in an `Authorization: Bearer <API_KEY>` header. Requests must include the model name (e.g. `"gpt-4"` or `"gpt-3.5-turbo"`).

Request Structure (Chat): The body uses a `messages` array where each message has a `role` and `content`. OpenAI supports three main roles: **system**, **user**, and **assistant** [1]. The system role sets overall behavior or instructions (e.g. `"You are a helpful assistant."`), user messages are prompts/queries, and assistant messages are model responses [2] [3]. For example, a chat payload might be:

```
{
  "model": "gpt-4",
  "messages": [
    {"role": "system", "content": "You are a polite assistant."},
    {"role": "user", "content": "Hello, how are you?"}
  ]
}
```

OpenAI's API **does not allow** arbitrary additional fields in messages (only role and content are accepted) [4].

Function Calling: OpenAI pioneered structured **function calling** in June 2023 to let the model request your application's functions. You can pass a list of function definitions in the request (each with `name`, `description`, and JSON Schema `parameters`). When enabled, the model can decide to return a special **function call** instead of a normal message [5] [6]. In a response, the assistant message may contain a `function_call` object (with the function `name` and a JSON string of `arguments`) instead of a text answer. For example, given a function `get_current_weather`, if the user asks "What's the weather in Boston?", GPT-4 might respond with:

```
{ "role": "assistant",
  "function_call": { "name": "get_current_weather", "arguments": "{\"location\":
\"Boston\",\"unit\":\"celsius\"}" } }
```

This signals your code to actually call `get_current_weather("Boston")` and then feed the result back to the model (usually as a message with role `"function"`). Function calling is a powerful way to **extend the model with tools**, letting it output JSON for you to execute [7] [8]. (Under the hood, the model is fine-tuned to know when to use these functions. It doesn't run code itself – it just outputs the function name/args, and you handle the rest [5] [9].)

Streaming Responses: OpenAI supports server-sent events (SSE) streaming. By including `stream: true` in the request, the API will stream partial message deltas as they are generated. The HTTP response is an `event-stream` where each event's data chunk contains JSON with either a fragment of the content or a `[DONE]` terminator. This allows token-by-token streaming in real time for better UX. Our unified API will need to consume and forward these chunks appropriately.

Multimodal (Images): OpenAI's GPT-4 is multimodal (text + image inputs) in the ChatGPT UI, but image input via API was historically unavailable. However, OpenAI's newer **GPT-4o** (introduced 2024) is a text-and-vision model accessible through the API [10]. GPT-4o can accept image data and text, returning text outputs, and is faster/cheaper than original GPT-4 [10]. The OpenAI API documentation indicates that images can be sent via a special content type or as part of a new "Responses API" with a model like `gpt-image-1` [11]. In practice, OpenAI's approach to images in the chat API is to allow a message with an image attached (for instance, using a special `{"role": "user", "content": <description>, "image": <binary or URL>}` format, as per their docs). For now, assume GPT-4o uses a similar message format as Google/Cohere (discussed below) where an image can be included as part of the user message content. If the unified interface is given an image and the provider is OpenAI GPT-4o, it should attach the image data in the manner OpenAI expects. (If a provider does *not* support images, the abstraction should throw or ignore the image content.)

Model Selection & Capabilities: OpenAI offers models like `gpt-3.5-turbo`, `gpt-4`, and specialized variants (e.g. 32k context versions, or GPT-4o). You can list available models via the OpenAI API or refer to their docs. Key differences include context length (e.g. GPT-4 supports 8K or 32K tokens), support for function calling (GPT-3.5 and GPT-4 both do, older models do not), and rate limits. Authentication and usage limits vary: for instance, OpenAI might impose RPM (requests per min) and TPM (tokens per min) quotas depending on your plan. Always handle HTTP 429 (Too Many Requests) errors by backing off or queuing requests.

Anthropic (Claude v1 – v2, Claude 4)

API Endpoint & Auth: Anthropic's primary interface is the **Claude API**. The latest version uses a **Messages API** similar to OpenAI's. Endpoint: `POST https://api.anthropic.com/v1/messages`. Auth is via an API key passed in a custom header (`x-api-key`) and an `anthropic-version` header specifying the API version date [12]. For example, one might call:

```
curl https://api.anthropic.com/v1/messages \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" \
  -H "Content-Type: application/json" \
  -d '{"model": "claude-2", "messages": [ ... ] }'
```

Request Structure (Chat): The request JSON includes a `model` (e.g. `claude-2` or newer Claude 4 variants) and a `messages` array of conversation turns [13]. **Roles:** Anthropic's models expect alternating user and assistant messages. They **do not use a system role inside** `messages`. Instead, to provide system-level instructions or context, Anthropic's API accepts a top-level `"system"` field in the JSON request [14]. You cannot include a message with `"role": "system"` in the array – it will be ignored or combined with a neighbor message [14]. So, for Anthropic, our abstraction should pull out any system message and put its content into a `system` parameter. (In practice, Anthropic often expects a "system prompt" or "preamble" string that plays a similar role to OpenAI's system message.)

For example:

```
{
  "model": "claude-2",
  "system": "You are a helpful coding assistant.",
  "messages": [
    {"role": "user", "content": "Explain recursion."},
    {"role": "assistant", "content": "Recursion is..."}
  ]
}
```

Claude will then produce the next assistant message in response to the last user query. The roles Anthropic supports are `user` and `assistant` only (no "function" role in this API). The API will combine consecutive messages of the same role if you accidentally supply them back-to-back [15].

Function/Tool Calling: Historically, Claude did not have an official function calling JSON schema like OpenAI's. Developers had to instruct Claude via prompt to output e.g. a JSON when it needs to call a tool. **However, by 2025 Anthropic introduced a formal tools interface.** Their documentation shows a top-level `tools` array in the API request, where each tool has a `name`, `description`, and `input_schema` (JSON Schema) [16] – essentially the same idea as OpenAI functions. Claude's newer models (Claude 2, Claude 4) have been trained to utilize these tool definitions. When a user prompt requires external info, Claude can return a **tool invocation**. Anthropic's response format for tool use is slightly different: the

model may produce an `assistant` message that contains a `tool_call` or `tool_calls` field. For instance, given a `get_weather` tool, Claude might output an assistant response indicating which function to call and with what args. In Anthropic's Agent API, the response could include something like:

```
{
  "role": "assistant",
  "tool_calls": [
    { "id": "get_weather_x123", "type": "function",
      "function": { "name": "get_weather", "arguments": "{\"location\":
\"Toronto\"}" } }
  ]
}
```

This is analogous to OpenAI's `function_call`, just structured differently [17]. The client (our code) is then expected to execute `get_weather("Toronto")` and send the result back as a new message for Claude to incorporate into its final answer. Anthropic supports **multi-step tool use** (Claude can plan to call multiple functions sequentially) and even has provisions for *parallel* calls [18] [19]. When integrating Anthropic, if `functions` (tools) are provided but the model does not support them (older Claude versions), our abstraction should handle that gracefully (e.g. by ignoring the function definitions or upgrading to a model that supports tools).

Streaming: Anthropic's API supports streaming via SSE as well. You can include a parameter `"stream": true` in the JSON, and the response will be incremental tokens on an SSE channel [20]. The unified API should handle this similar to OpenAI: open a stream and yield message content chunks as they arrive. (Anthropic also uses `stop_reason` fields – for example, `stop_reason: "end_turn"` in the final message – to indicate why generation stopped.)

Multimodal (Images): Claude's newest models (Claude 2, Claude 4 "Sonnet" and "Opus" editions) are **multimodal**, accepting image inputs alongside text [21]. Anthropic's docs note "text and image input, text output" for models like Claude Sonnet 4.5 [21]. This means you can send an image (likely as a URL or binary data) as part of the conversation. The exact API representation is via the message `content` field: Anthropic uses a content schema similar to others, where a message's content can be an array of segments, each possibly of type `"text"` or `"image"` (or a top-level separate field). For instance, you might provide:

```
{
  "model": "claude-sonnet-4-5-20250929",
  "messages": [{
    "role": "user",
    "content": [
      {"text": "What is depicted in this chart?"},
      {"image": "<base64_image_data_or_url>"}
    ]
  }]
}
```

Claude will then analyze the image and text together. Our unified API should detect if an image is included in the prompt and if the provider model can handle it (Claude 4, Google Gemini, Cohere Vision, etc.), then format accordingly. If the provider can't handle images, we must either reject that call or strip the image and warn the user.

Model Selection: Anthropic has multiple model families: e.g. **Claude 2** (100k context), **Claude Instant** (faster, lower cost), and by 2025, **Claude 4** (Sonnet, Opus variants) with up to 200k or even 1M token context windows [21]. Model names are often suffixed with version dates (e.g. `"claude-2.0-2023-13-01"` or `"claude-instant-1.2"`). The unified layer might maintain a mapping or allow configuring these model IDs. Also note that Anthropic models are offered through third-party platforms too (Amazon Bedrock, Google Vertex AI) with their own naming conventions [22]. But when calling Anthropic's API directly, use the Anthropic model ID.

Google (PaLM API / Gemini)

API Endpoint & Auth: Google's generative models are accessible via Google Cloud's Vertex AI **Generative Language API** (PaLM API) and the newer **Gemini API**. By late 2024, the PaLM 2 API (which included `text-bison` and `chat-bison` models) was **deprecated in favor of Gemini** [23]. We will focus on Google's **Gemini API** (available via Google Cloud's AI services). Authentication is through a Google Cloud API key (`x-goog-api-key` header) or OAuth (depending on context). The base endpoint is `https://generativelanguage.googleapis.com/v1beta/...`. Google provides both REST and SDK options.

Chat Request Structure: Google's API uses a more hierarchical JSON structure. Rather than a flat list of messages, you send a list of **Content** objects in a `contents` array [24]. Each Content has a `role` and a list of `parts`. Roles in Google's API are typically `"user"` for human prompts and `"model"` for the AI's responses [25] [26]. (They use `"model"` in place of "assistant".) A simple conversation with one prompt would be:

```
{
  "model": "gemini-2.5-bison",
  "contents": [
    {
      "role": "user",
      "parts": [ { "text": "Hello, world!" } ]
    }
  ]
}
```

If you include conversation history, you append more Content objects alternating roles (user/model) to simulate turns. Google's representation of each message as `parts` allows **multimodal data**: each Part can be either a text segment or an image. For example, you could have:

```
{
  "role": "user",
  "parts": [
```

```
    { "text": "Describe this image:" },
    { "image": { "inlineBytes": "<base64_image_data>", "mimeType": "image/
png" } }
  ]
}
```

or using a reference by URL if supported. The Google API's design is flexible in that regard [27] [28]. Google doesn't explicitly use a system role in the conversation, but they do allow an optional **"context"** or **"examples"** field in PaLM API (for static system instructions or few-shot examples). In the Gemini API reference, the notion of system message isn't prominent; you'd typically prepend system instructions into the first user message or use a separate field if provided.

Model Invocation & Functions: As of PaLM 2, Google's API did not have a built-in function calling schema like OpenAI's. The model would return only text. Developers could instruct the model to output JSON or a special format for tools, but it wasn't an official feature. With Gemini, Google's focus has been on multimodal and chat improvements, but there's no public indication of a JSON function call feature yet. Thus, our unified API should treat Google as **no function calling support** (capability fallback: if a function call is requested, either handle it at the application level or disable it). If needed, one could implement a *manual* approach: e.g. prompt the model with *"If the user's query requires using a tool, output a JSON like... instead of an answer."* and then parse it. But that is beyond native support. So for simplicity, when `callFunction` requests come in for Google, we'll either throw NotSupported or attempt a pure text answer.

Streaming: Google's Gemini API offers multiple ways to get output: a standard blocking call, an SSE streaming call, and even a bidirectional streaming via WebSocket for real-time chat [29]. The SSE endpoint is typically `...:streamGenerateContent`. If you call this endpoint with the same request JSON, the response will be a stream of events (each containing a partial `GenerateContentResponse`) [30]. The unified client should detect if `stream: true` and, for Google, hit the `streamGenerateContent` path. The SSE events contain `candidates` with partial text; as each chunk arrives, we extract the delta and forward it. Google also has a **"live" WebSocket API** for stateful chat [31], but implementing a WS client is more complex and usually not needed unless ultra-low latency bi-directional interaction is required. We can stick to SSE for streaming.

Multimodal: Multimodal is a **first-class feature** of Gemini. The model can handle image inputs and even generate non-text outputs (Google mentions separate Gen Media APIs for images and video, but also that Gemini has those capabilities built-in [32]). For our purposes, Gemini's chat will return text for chat models, but it can **understand images in the prompt**. Our abstraction should package images into the `parts` structure as described. If the user provides an image to a Google model, we'll convert it (likely to base64 bytes in JSON) with the appropriate MIME type.

Model Discovery: Google's models are identified by names like `models/chat-bison-001` (PaLM2) or `models/gemini-2.5-bison`. There might be different sizes or versions (Gemini 1, 2, 3 etc. as they evolve). Google's API might provide a list of available models via an endpoint or one can refer to documentation. In practice, you often know the model ID you want to use. The unified interface could keep a list of Google model IDs it supports, or accept a full model name from the user. Keep in mind that Google's

access may be restricted (some models might be in preview or require billing enabled). The API key's project must have the GenAI API enabled and appropriate quotas.

Cohere (Command / Chat Models)

API Endpoint & Auth: Cohere provides a cloud API for their proprietary models. The chat endpoint is typically accessible via their SDK or HTTP at `https://api.cohere.ai/v1/chat` (exact endpoint path may differ; their docs encourage using the SDK). Authentication is by API key in the `Authorization: Bearer` header, similar to OpenAI.

Request Structure (Chat): Cohere's **Chat API** uses a list of `messages`, each with `role` and `content`, much like OpenAI. Supported roles are `user`, `assistant`, `system`, **and** `tool` [33]. The system role here behaves the same as OpenAI's: an initial instruction to set behavior [34]. For example, you might send:

```
{
  "model": "command-nightly",
  "messages": [
    {"role": "system", "content": "You are an expert technical writer."},
    {"role": "user", "content": "Explain what an API is."}
  ]
}
```

Cohere will then produce an `assistant` message. The **"tool" role** is used in their function calling flow (described next). The content of messages in Cohere can be either plain text or more complex objects. For simple text, `content` is a string. For multimodal or tool interactions, `content` can be an **array of content blocks** with types (text, image, etc.) [35]. For example, to send an image with a prompt:

```
{
  "role": "user",
  "content": [
    { "type": "text", "text": "What is in this image?" },
    { "type": "image_url", "image_url": { "url": "<IMAGE_URL_OR_BASE64>" } }
  ]
}
```

Cohere's `command-vision` models can handle that and the response may also include references to the image. Our unified API should map a `Message` that contains, say, an `image` field to this structure when calling Cohere.

Function Calling (Tool Use): Cohere's v2 API introduced **tool use (function calling)** similar to OpenAI [36]. You can define tools with a schema and pass them in the `tools` parameter of the chat request. The model can decide to invoke those tools. The flow is a bit more involved: when you call `co.chat` with `tools` provided, the API may return an intermediate response that includes `response.message.tool_calls` – a list of tool calls the model wants to make [37] [17]. Each tool call has an `id`, `name`, and arguments. The

client (your code) should then execute those function(s) and add the results back into the conversation, *then* call the chat API again to get the final answer [38] [39]. Essentially, Cohere's API helps manage the agent loop: 1. **Step 1:** Send user message (and system if any) to `chat` with tools defined. 2. **Step 2:** Get response; if `tool_calls` is non-empty, extract the calls. The model also provides a `tool_plan` (a natural language reasoning about next steps) [40]. 3. **Step 3:** For each tool call, execute the actual function in your environment. Format the result as a **document content block** (with `type: "document"`) containing the data [41]. 4. **Step 4:** Append a message to the conversation with role `"assistant"` that includes the `tool_calls` and `tool_plan` it gave (so the history knows the model decided to use a tool), and then also append the function results (as role `"assistant"` with content of type document, tagged with the tool call ID) [41]. 5. Call the `chat` API again with the updated messages to let the model generate the final `"assistant"` answer that presumably uses the tool results.

While this is more complex than OpenAI's single-turn function call, it achieves the same goal. Our unified API can simplify this for the developer by internally handling this loop if desired (e.g. a `callFunction` helper that wraps the above steps). At minimum, if the unified interface is told to use functions with Cohere, it should prepare the `tools` schema and handle that the initial response may not be final text but a tool invocation that needs fulfillment. (Alternatively, the abstraction can expose a simpler model: e.g. always do one turn at a time and let the user of the library handle tool calls. But since we want a unified experience, automating it like OpenAI's style is ideal.)

Streaming: Cohere supports streaming responses as well. The Cohere SDK provides a `co.chat_stream()` which yields events in real-time [42]. Under the hood, Cohere's streaming uses event types like `message-start`, `content-delta`, etc., rather than raw text. For example, as the text is generated, you get `content-delta` events containing chunks of text [43]. In our abstraction, we don't need to expose these low-level events to the user. Instead, we can convert them into a continuous stream of text. Essentially, for streaming, our Cohere adapter will call the streaming endpoint and concatenate any `content-delta` event text into the outgoing stream. Once a `message-end` event is received, the stream is done [44]. This is a bit different from OpenAI/Anthropic SSE (which just send text directly), but the unified interface can hide that complexity.

Multimodal: Cohere has specific **Vision-capable models** (e.g. `command-x-vision` variants). If you use such a model and include image parts in the message content, the API will interpret them. We saw an example above with `{"type": "image_url", "image_url": {"url": ...}}` [35]. They also allow controlling the detail level of image analysis (low, high detail) [45]. The unified API should allow images in the message and for Cohere, format accordingly. If a user tries to send an image to a Cohere model that isn't vision-capable, the API may error or just not utilize it, so the abstraction could warn or throw in that case. (We could possibly auto-select a vision model if an image is present and the user didn't specify a model – e.g. switch `command-x` to `command-x-vision` – but that might be beyond scope; better to require the caller to pick an appropriate model or handle the error.)

Model Selection: Cohere offers a few model families: **Command** (their flagship large language model for generation, with versions like `command-nightly`, `command-xlarge-2023-11` etc.), smaller models like **Command-Light**, and others (like `embed` for embeddings, `rerank` for re-ranking tasks, etc.). For chat, you'll likely use the Command family. Model capabilities differ mainly in size and training; all support chat, and as of V2, tool use and vision if applicable. The unified layer should allow specifying the exact model name for Cohere (or choose a default). Also note Cohere's rate limits: for instance, trial API keys may allow

only 20 chat requests/min, while production keys allow 500/min [46]. Our error handling should check for 429s from Cohere and possibly advise the user or throttle.

Meta's LLaMA (via Third-Party APIs)

Meta's LLaMA family (LLaMA 2, etc.) is open-source but not provided as a hosted API by Meta. To integrate these models, we rely on **third-party services or self-hosting**. There are a few options:

- **Hugging Face Inference API:** HuggingFace hosts many open models (including LLaMA 2, Mistral 7B, etc.) that can be accessed via their unified inference HTTP API. You make a POST to a model endpoint (e.g. `https://api-inference.huggingface.co/models/meta-llama/Llama-2-70b-chat-hf`) with your prompt, and get the completion. The upside: one API for many models. Downside: you need an HF API token, and it might not support multi-turn chat out of the box (for chat models, the entire conversation must typically be sent as one prompt, since the API is stateless). We can still use this by concatenating conversation messages into a single prompt string (e.g., prefix user messages with "User:" and assistant with "Assistant:" as the model was trained). Our abstraction might support a provider called `"huggingface"` where the model name can be a HF repo, and we handle constructing a prompt from the messages. This is more manual (and function calling would be entirely manual if at all).

- **Other Hosted APIs (Replicate, Together AI, Bedrock):** Several platforms provide **OpenAI-compatible endpoints** for open models. For example, **Together.ai** has an API where you can choose a model (like Mistral or LLaMA) and call `POST /v1/chat/completions` with the same schema as OpenAI [47]. Amazon's **Bedrock** service similarly offers Meta's Llama-2 and others via a unified API (though Bedrock's request/response format is slightly different JSON). If using Together or a similar service, we could integrate it as another "provider" in our abstraction. Notably, Together's API uses the OpenAI schema (messages with role) even for open models, which simplifies integration [47]. We'd just need the endpoint URL and an API key.

- **Self-Hosted via an Adapter:** If the user runs a local LLM server (e.g. an API provided by a library like `llama.cpp` or `vLLM`), we can also plug that in. For example, vLLM has an OpenAI-compatible REST server mode [48]. Our unified client could be pointed at that local endpoint and treat it like OpenAI.

For this guide, we'll assume use of either HuggingFace or an OpenAI-compatible proxy for LLaMA/Mistral. **Capabilities:** LLaMA and Mistral models (depending on fine-tune) can engage in chat, but they typically **do not have built-in function calling** support or vision, unless specifically fine-tuned for it. (Some fine-tuned versions might follow the function call format if prompted, but it's not guaranteed). So the abstraction should mark these as limited: no function-call JSON scheme and text-only. Multimodal variants of open models exist (e.g. LLaVA for vision), but those require specialized endpoints or frameworks, likely not included here.

In summary, for Meta/Mistral via third-party: **endpoints and auth** will vary (e.g. HF uses Bearer token in `Authorization`, Together uses its own key). The request format might either be OpenAI-like (if using Together/Bedrock) or raw text (HF). Our unified API can provide an adapter that if `provider === 'huggingface'`, it will compose the prompt from messages, call the HF endpoint, and then parse the

completion into an assistant message. If `provider === 'together'` or `'bedrock'`, we use their OpenAI-compatible interface directly.

Model Selection: The model names here correspond to the specific model you want. E.g., `"llama2-70b-chat"` or `"mistral-7b-instruct-v0.2"`. In the config for our client, we might have to specify the exact model ID and possibly the API base URL for these open models. It's less plug-and-play than the others, but the unified interface can at least make it uniform to call them once configured.

Key Differences Across Providers

Now that each provider's basics are outlined, let's compare them on crucial dimensions and see how our abstraction will reconcile these differences:

- **Conversation Format (Roles & Messages):** OpenAI, Anthropic, Cohere, and Together API all use a **role-based message list** in JSON. Google uses a content-part structure. The unified interface will likely expose a simple `ChatMessage` type like:

  ```
  type Role = 'system'|'user'|'assistant'|'function';
  interface ChatMessage { role: Role; content: string; name?: string; }
  ```

  (We include `name` for function messages to hold the function name, like OpenAI's `{"role":"function","name":"funcName","content":"<result>"}` when returning function outputs.)

Under the hood, the adapter will transform this array for each provider: - **OpenAI/Cohere/Together:** Use as-is (possibly filtering out or converting the `function` role message, since OpenAI expects a function result message's role to be `"function"`, whereas Cohere might treat it as an assistant message with a document content). - **Anthropic:** Separate out the `system` message (if present) and put it in the top-level `system` field [14]; use only user/assistant in the list. - **Google:** Convert to its `contents` array. Each unified message becomes one `Content` with role mapping (`assistant` -> `model`, `function` -> could be treated as model output too, or we might decide not to support function messages for Google). If a message content is very large (e.g. system prompt), Google might prefer it as a `context` parameter rather than a turn; but since such context field is not clearly documented for Gemini, we will include it as a fake user turn like: role:user, content: systemText, with an instruction that it's context. Alternatively, we could inject it as an `context` field if their API allows (PaLM had a "prompt context" string outside of messages).

- **System & Initialization:** OpenAI and Cohere allow a system message in the messages list [49], whereas Anthropic uses a separate field, and Google lacks an explicit system role. Our abstraction can accept an optional system message; for Anthropic we handle it specially, for Google we might prepend it as a user prefix or leave it out if not needed. It's important developers know that not all providers honor system instructions equally: e.g. older Claude might not follow system prompt as strongly, etc.

- **Function Calling / Tools:** This is a major differentiator:

- **Built-in Support:** OpenAI and Cohere have first-class JSON-based function calling [5] [36]. Anthropic's latest API also supports tools with JSON schema [16]. Google and most open models do not natively support this.

- **Unified Interface:** We define something like:

```
interface FunctionDefinition { name: string; description?: string;
parameters: object /* JSON schema */; }
```

Our `generateChat` could accept a list of `functions: FunctionDefinition[]` and a flag or strategy for function calling (e.g. auto vs none). For providers that support it (OpenAI, Cohere, Anthropic), we include these in the request. For those that don't, we have two choices:

1. **Disable** – ignore the function definitions and just get a normal completion. (Possibly inject a note in the system prompt like "(Tool not available)".)
2. **Simulate** – for open models, one could implement a simple reflex: if user asks something that matches a function's purpose, the unified API itself could call the function and append the result to the prompt. But this ventures into building an agent, which is complex and outside our scope. It's safer to **fallback gracefully**: e.g. log or throw an error if `functions` were provided but unsupported by the chosen provider, so the developer knows that feature won't work.

- **Response Handling:** When a function call is invoked:

  - In OpenAI, the API response will include `message.function_call`. Our abstraction can detect that and either **automatically handle it** by calling the function and continuing, or expose it to the caller. A convenient design is to offer both:
    - A high-level method `generateChat` that by default **auto-resolves function calls** (making additional API calls internally until a final answer is reached), so the developer just gets the end answer.
    - Lower-level control where the developer can check if the response `type` is a function call and handle it manually (if they want to, say, stream intermediate steps or have custom logging).
  - For Cohere, as described, it requires multiple steps. We can encapsulate that loop inside our library.
  - For Anthropic, tool use might similarly require multiple turns (the Claude docs indicate the model can output a special "pause" where it expects the tool results before continuing [50]). We'll implement it akin to OpenAI's pattern.

- **Streaming:** All major providers except some open-model endpoints support streaming. Implementation differences:

- OpenAI/Anthropic: SSE text stream.
- Cohere: SSE with event types (content-delta etc.) [43].
- Google: SSE with JSON messages containing partial candidates [51].
- HuggingFace (open models): No streaming support in their basic API – you get the full completion after some time. Our abstraction should indicate that streaming is not available for those, or

simulate it by chunking the final output (not real streaming, but could emit chunks of the string every few milliseconds as a hack). Probably simpler to just not support streaming on those providers and throw if requested.

In our design, we can unify streaming by providing an **async iterator** or event emitter interface. For example:

```
const stream = client.streamOutput(request);
for await (const token of stream) {
  process.stdout.write(token);
}
```

Under the hood, the adapter will do provider-specific things: * For SSE (OpenAI, etc.), parse the `data:` lines and yield the `content` delta. * For Cohere, yield `event.delta.message.content.text` on each `content-delta` event [52] . * For Google, parse each SSE event's JSON for the new text segment. We must also handle stopping the stream: e.g. if the user cancels or on a `[DONE]` message (OpenAI sends a final event with `[DONE]`).

- **Multimodal (Images in prompts):** By 2025, OpenAI GPT-4o, Anthropic Claude 4, Cohere's Command Vision, and Google Gemini all accept image inputs. Key differences:
- **Input format:** Google and Cohere use a list of content parts with an image type [35] [27] . OpenAI might have a similar structure (not officially documented to the public, but likely the `messages` content can include an image if using the "Responses API" for GPT-4o). Anthropic likely follows a parts structure as well for images. So conceptually, we can treat them similarly: our `ChatMessage` type could allow `content` to be a **string or an object** like `{text?: string, image?: Buffer/URL}`. But that complicates the interface. Simpler is to allow a special message role or a convention (like a message with role `"user"` and content `"[image]"` plus an actual binary separately). However, to keep it clean, maybe allow `content` to be of type `any` in our interface with specific guidelines: if `typeof content === 'string'`, treat as text; if it's an `{image: ...}` object, handle accordingly.
- **Output format:** None of these models will *generate* images via the chat API (though Google has separate image generation models like Imagen). They only produce text descriptions or answers. So our unified API doesn't need to handle binary data in outputs, only inputs.

- If a model doesn't support image input (e.g. OpenAI GPT-3.5, Cohere Command (non-vision), a vanilla LLaMA), the abstraction should either filter out the image or warn. Possibly, we could convert the image to a caption via an OCR or just say "[image of X]" as fallback in the prompt, but that's beyond scope. It's safer to throw an error like "Provider X does not support image prompts" if the user tries it.

- **Model Capability Discovery:** Each provider has a way to list or describe models:

- OpenAI: `GET /v1/models` returns model IDs and basic info. We can call that if needed to populate options.
- Anthropic: No public endpoint to list models, but their docs enumerate them (Claude 1, 2, etc.). We might hardcode or configure available models.

- Google: Similarly, no public "list models" REST endpoint outside the Google Cloud console. The unified API might just let the user configure known model names.
- Cohere: Fewer models, listed in docs (we can code those in or allow dynamic specification).
- HuggingFace: Potentially thousands of models; obviously not listing them all. We'd take a model name string directly.

A practical approach: have a config where you register the models or choose one per provider. The unified client could optionally expose a method `listModels(provider)` that returns a static list of known ones (for user convenience) or calls the provider's API if available. In our code examples, we might not dive deep into this, but we should highlight the importance of **capability flags** – e.g. marking which model supports `functions` or `images` – for dynamic behavior. For instance, in our abstraction we could have a mapping:

```
const modelCapabilities = {
  openai: { "gpt-4": { functions: true, vision: true, maxContext: 8192 }, ... },
  anthropic: { "claude-2": { functions: false, vision: false, maxContext:
100k }, ... },
  cohere: { "command-x-2025": { functions: true, vision: false, maxContext:
4096 }, "command-vision-2025": { functions: true, vision: true } },
  // etc.
};
```

Our code can consult this to adapt requests (e.g., if `functions: false`, don't send function definitions).

Designing a Unified Abstraction Layer

With the differences in mind, we design an abstraction that normalizes usage. The core idea is to **provide a consistent set of methods** regardless of provider, handling under-the-hood format conversion and feature toggles.

Unified Interface and Core Types

Let's define some TypeScript types for clarity:

```
type Provider =
'openai'|'anthropic'|'google'|'cohere'|'huggingface'|'together'|'bedrock';

interface ChatMessage {
  role: 'system'|'user'|'assistant'|'function'|'tool';
  content: string;
  name?: string; // for 'function' role or for future use
}
interface FunctionDefinition {
  name: string;
  description?: string;
```

```
  parameters: any; // JSON Schema object
}
interface ChatRequest {
  provider: Provider;
  model: string;
  messages: ChatMessage[];
  functions?: FunctionDefinition[];
  stream?: boolean;
}

type ChatResponse = {
  message: ChatMessage;             // The assistant's answer (for non-stream
calls)
  functionCall?: { name: string, arguments: any }; // If the model requested a
function
  raw?: any; // raw provider response (optional)
}
```

We can then implement a class, say `UnifiedLLMClient`, which is configured with credentials for each provider:

```
class UnifiedLLMClient {
  constructor(config: { openai?: {apiKey: string}, anthropic?: {apiKey:
string}, /* ...other creds... */ }) { … }

  async generateChat(req: ChatRequest): Promise<ChatResponse> { … }

  streamOutput(req: ChatRequest): AsyncIterable<string> { … }

  // Optionally, helper for function call loop:
  async generateChatWithFunctions(req: ChatRequest): Promise<ChatResponse> { … }
}
```

The user of this client will always format their conversation as an array of `ChatMessage` and specify which provider and model to use for that request. (We could also allow configuring a "default provider/model" to avoid repetition if most calls go to one model.)

Request Adaptation per Provider

Inside `generateChat`, we will inspect `req.provider` and route to a provider-specific handler. Pseudocode:

```
async generateChat(req: ChatRequest): Promise<ChatResponse> {
  switch(req.provider) {
    case 'openai': return this.callOpenAI(req);
```

```
    case 'anthropic': return this.callAnthropic(req);
    case 'google': return this.callGoogle(req);
    case 'cohere': return this.callCohere(req);
    case 'huggingface': return this.callHF(req);
    // ...others
  }
}
```

Each `callX` method will take the unified `ChatRequest` and transform it to that provider's API format, perform the HTTP request(s), and parse the result back into a `ChatResponse`. Key adaptations:

- **Message Formatting:** E.g. `callAnthropic` will do:

```
const {model, messages, functions} = req;
let sysMsg = undefined;
const msgs = [];
for (const m of messages) {
  if (m.role === 'system') {
    sysMsg = m.content;
    continue;
  }
  if (m.role === 'function') {
    // anthropic doesn't support 'function' role; maybe include it as
assistant content.
    msgs.push({ role: 'assistant', content: m.content });
    continue;
  }
  // For user/assistant, content is straightforward:
  msgs.push({ role: m.role, content: m.content });
}
const body: any = { model, messages: msgs };
if (sysMsg) body.system = sysMsg;
if (functions && functions.length) {
  // Map our FunctionDefinition to Anthropic's "tools" format
  body.tools = functions.map(fn => ({
    name: fn.name,
    description: fn.description || "",
    input_schema: fn.parameters   // note: OpenAI uses "parameters",
Anthropic calls it "input_schema"
  }));
}
// If streaming requested, set stream: true or use streaming endpoint.
if (req.stream) body.stream = true;

// ... make HTTP POST to anthropic ...
```

Similarly, `callOpenAI` will map `functions` to the OpenAI JSON (they use `functions` key, and have an optional `"function_call": "auto"` parameter we might set) [53]. It will pass through messages except that OpenAI does support the `'function'` role (for function result messages) – we should include those if present in history. OpenAI also requires us to omit any messages with role `'function'` in the initial user prompt list if we are making the first call with function definitions – those are only used when sending the function's result back. So likely in most calls, the messages will be just system/user/assistant, and function calls are handled within a loop.

`callCohere` will need to handle the content blocks for images or tool calls. If our `ChatMessage.content` is just a string, we send it as `content: "..."`. If we have an `image` placeholder, we need to send `content: [{type:'text', text:...}, {type:'image_url', image_url:{url: ...}}]`. We could extend `ChatMessage` in our interface to allow content to be an array of `{type, text?, url?}` objects for multimodal, but that complicates usage for the developer. Instead, the developer could just attach images by a special convention (e.g., include a base64 string or an object in the content and our code detects it). For clarity: we might simply document that to send an image, the user should include a `ChatMessage` where `content` is a string with a placeholder or the image URL, and set `metadata: {image: true}` or something. However, to keep this guide high-level, assume we have some way to know when an image is included (perhaps the `ChatMessage` has a field `imageUrl`). Then:

```
if (m.imageUrl) {
  coMsgs.push({ role: m.role, content: [
     { type: 'text', text: m.content || "" },
     { type: 'image_url', image_url: { url: m.imageUrl } }
  ]});
} else {
  coMsgs.push({ role: m.role, content: m.content });
}
```

Also, if `functions` are provided to Cohere, we attach them as `tools` param, similar to Anthropic (Cohere calls them "tools" as well) [37].

- **Transport Differences:** Some providers (OpenAI, Cohere) have straightforward REST calls, while Google's requires hitting a specific path for the model (like `.../models/${model}:generateContent`). `callGoogle` implementation might be:

  ```
  const url = googleEndpointForModel(req.model, req.stream);
  // googleEndpointForModel would format the URL, e.g.
  // "https://generativelanguage.googleapis.com/v1beta2/models/
  <<modelId>>:generateContent"
  // or :streamGenerateContent if streaming.
  const payload = { contents: [] };
  for (const msg of messages) {
    // similar mapping as before:
    if (msg.role === 'system') {
  ```

```
    // we might prepend to content or ignore if we plan not to support
directly.
    continue;
  }
  const role = (msg.role === 'assistant' || msg.role === 'function') ?
'model' : 'user';
  const entry: any = { role, parts: [] };
  // If this message has an image (we detect maybe via msg.content
containing something), break content into parts:
  // For simplicity: assume no images here; if needed, handle like cohere
by splitting text vs image parts.
  entry.parts.push({ text: msg.content });
  payload.contents.push(entry);
}
// Then POST to the URL with payload.
```

Note: For Google, we need to set the API key in the URL or header and make sure to handle OAuth if needed (but likely we'd use API key for simplicity).

- **Multiple calls for function handling:** If we want `generateChat` to automatically handle function calls in one go, the logic will be:

- Make initial request.
- If provider returns a function call (OpenAI: `response.choices[0].message.function_call` present; Cohere: `tool_calls` present; Anthropic: `stop_reason: "pause_turn"` or tool call present), then:
  - Parse the function name and args.
  - Look up the actual function implementation (the user might register functions in a map when calling our client).
  - Execute it and get result.
  - Append a new message with role `"function"` (OpenAI style) or role `"assistant"` with the result content (Cohere/Anthropic style, possibly using special fields like in Cohere we attach the result as a document content with the call ID [54]).
  - Call the API again to get the final answer.
- Return the final answer.

The unified client can accept a parameter like `autoExecuteFunctions: boolean` or similar to toggle this behavior. For simplicity, we might always auto-execute if the actual function definitions & implementations are provided to the client.

We will likely require the user to provide actual JavaScript function implementations corresponding to the names (since our library can't magically know how to fulfill "get_weather"). Perhaps the config of `UnifiedLLMClient` could include a mapping `{ functionName: implementation }` for each provider's functions. If not provided, we can't execute and so either return the function call info to the user or throw an error. In our examples, we assume the developer handles it or only uses function calling when they've registered implementations.

Example: Unified Chat Call Implementation

Let's demonstrate with pseudo-code for OpenAI and Cohere, as they represent two patterns (single request vs multi-step tool use):

OpenAI Example Implementation (simplified):

```
async callOpenAI(req: ChatRequest): Promise<ChatResponse> {
  const body: any = {
    model: req.model,
    messages: req.messages.map(m => ({ role: m.role, content: m.content, name:
m.name })).filter(m => m.role !== 'function')
  };
  if (req.functions) {
    body.functions = req.functions;
    body.function_call = 'auto';  // let model decide if/when to call
  }
  if (req.stream) {
    // We'll not use this in callOpenAI (non-stream method); streamOutput
handles streaming.
  }
  const res = await fetch("https://api.openai.com/v1/chat/completions", {
    method: 'POST',
    headers: { 'Authorization': `Bearer ${this.openAIApiKey}`, 'Content-Type':
'application/json' },
    body: JSON.stringify(body)
  });
  const result = await res.json();
  // Check for errors etc.
  const choice = result.choices[0];
  if (choice.finish_reason === 'function_call' ||
choice.message?.function_call) {
    // The model wants to call a function
    return {
      message: { role: 'assistant', content: '', name: undefined },
      functionCall: {
        name: choice.message.function_call.name,
        arguments: JSON.parse(choice.message.function_call.arguments || '{}')
      }
    };
  } else {
    return {
      message: { role: 'assistant', content: choice.message.content }
    };
  }
}
```

If the returned `ChatResponse` has a `functionCall`, our `generateChat` could detect that and handle it (if auto mode): call the corresponding function, then append its result and call `callOpenAI` again to get the final answer. This final answer would be returned as `ChatResponse.message` with no functionCall.

Cohere Example Implementation (simplified):

```
async callCohere(req: ChatRequest): Promise<ChatResponse> {
  const body: any = { model: req.model, messages: [] };
  for (let m of req.messages) {
    if (m.role === 'function') {
      // The function result from a prior call: Cohere expects it as assistant
content of type document.
      // We assume m.name is the function name in this case.
      body.messages.push({
        role: 'assistant',
        content: [{
           type: 'document',
           document: { id: m.name ?? undefined, data: m.content }
        }]
      });
      continue;
    }
    // Normal message
    const msgObj: any = { role: m.role };
    // handle multimodal: if our ChatMessage had structure for that, we'd do
similar to earlier.
    msgObj.content = m.content;
    body.messages.push(msgObj);
  }
  if (req.functions) {
    // Map to Cohere's tools structure
    body.tools = req.functions.map(fn => ({
      name: fn.name,
      description: fn.description || "",
      parameters:
fn.parameters  // Cohere might also accept JSON Schema similar to Anthropic
    }));
  }
  // (Cohere's API might have a separate endpoint or SDK call; using fetch for
conceptual parity)
  const res = await fetch("https://api.cohere.ai/v1/chat", {
    method: 'POST',
    headers: { 'Authorization': `Bearer ${this.cohereApiKey}`, 'Content-Type':
'application/json' },
    body: JSON.stringify(body)
  });
```

```
  const result = await res.json();
  // Cohere response structure:
  // { id: "...", message: { role: "...", content: [ {type, text or
document}... ], tool_calls?: [...], tool_plan?: "..." }, finish_reason: "..." }
  const msg = result.message;
  if (msg.tool_calls && msg.tool_calls.length > 0) {
    // Model wants to call functions
    return {
      message: { role: 'assistant', content: '', name: undefined },
      functionCall: {
        name: msg.tool_calls[0].function.name,
        arguments: JSON.parse(msg.tool_calls[0].function.arguments || '{}')
      }
    };
    // (If multiple tool_calls are returned, we might need to handle them
sequentially or in parallel.
    // For simplicity, handle one at a time: callCohere would only return the
first requested function.)
  } else {
    // No function call, just final answer
    // Need to extract text content from possibly structured content array:
    let textOut = "";
    for (const part of msg.content) {
      if (part.type === 'text') textOut += part.text;
      // ignore other types in final answer for now (or handle citations, etc.)
    }
    return { message: { role: 'assistant', content: textOut } };
  }
}
```

We can see that both for OpenAI and Cohere, we return a `functionCall` in the response if the model is requesting a tool. The higher-level `generateChat` will catch that and do the iterative loop:

```
async generateChat(req: ChatRequest): Promise<ChatResponse> {
  // ... initial call
  let response = await this[`call${capitalize(req.provider)}`](req);
  if (response.functionCall) {
    const { name, arguments: args } = response.functionCall;
    // find the actual function implementation provided by user
    const funcImpl = this.functionRegistry?.[name];
    if (!funcImpl) throw new Error(`Function ${name} not implemented`);
    const result = await funcImpl(args);
    // append function result to messages and call again
    const newMessages = [...req.messages,
                         { role: 'function', name: name, content:
JSON.stringify(result) }];
```

```
    return this.generateChat({ ...req, messages: newMessages });
  }
  return response;
}
```

This simplistic loop assumes one function call; for multiple sequential calls, this would naturally loop multiple times. For parallel (Cohere can request multiple tool_calls at once [55]), we'd need to execute all, append results for each with their IDs, etc., then call again.

Unified Streaming Design

For streaming, we provide `streamOutput(req)` that returns an AsyncIterable (or could be a Node-style EventEmitter). Implementation example for OpenAI:

```
async * streamOpenAI(req: ChatRequest): AsyncIterable<string> {
  const body = { /* same as before, but with stream: true */ };
  const res = await fetch(openAIUrl, { headers, body: JSON.stringify({...body,
stream: true}) });
  if (!res.body) throw new Error("No stream supported");
  const reader = res.body.getReader();
  const decoder = new TextDecoder();
  let buffer = "";
  while (true) {
    const { done, value } = await reader.read();
    if (done) break;
    buffer += decoder.decode(value, { stream: true });
    // SSE events are separated by \n\n. Process any full events in buffer:
    let eventEnd;
    while ((eventEnd = buffer.indexOf("\n\n")) !== -1) {
      const event = buffer.slice(0, eventEnd);
      buffer = buffer.slice(eventEnd + 2);
      if (event.startsWith("data: ")) {
        const data = event.slice(6);
        if (data === "[DONE]") {
          return; // stream end
        }
        if (data.trim() === "") continue; // keep-alive ping
        const parsed = JSON.parse(data);
        const delta = parsed.choices[0].delta?.content || "";
        yield delta;
      }
    }
  }
}
```

For Cohere, streaming events come similarly via SSE but with a JSON object per event containing a field `event.type`. We'd do a similar loop, but parse JSON and if `event.type=="content-delta"`, yield `event.delta.message.content.text` [52]. If `event.type=="message-end"`, end the loop. Google's SSE would have to parse `GenerateContentResponse` chunks and output the new text portion (they often provide full candidate text each time or an incremental part – the docs aren't explicit in our snippet, but we can infer it's incremental).

The unified `streamOutput` will choose the correct `streamX` function. The consumer can iterate and accumulate tokens. If function calling is in use, streaming becomes tricky: e.g., what if the model triggers a function call half-way? In practice, OpenAI will not stream content and then suddenly a function call – if it's going to do a function call, it sends **no content** aside from the function call in a single message. So we would detect that as soon as the streaming yields a message with function_call (OpenAI's SSE would include something like `"role":"assistant","content":null,"function_call":{...}` event). Our abstraction could stop and handle it. However, implementing full function loop with streaming is complex. Simpler: we might **disable function calling when streaming** or require the user to handle it manually. For this guide, we can note that streaming combined with auto-function-calling is advanced – likely, one would either stream the final response after function resolution (e.g., first part non-stream to handle the function, then stream the answer). In interest of time, we assume streaming is mainly used for direct responses.

Error Handling, Retries, and Timeouts

In each `callX` method, we should handle HTTP errors. For example, if OpenAI returns 401 or 429, we throw an appropriate error (maybe a custom Error class like `RateLimitError` or `UnauthorizedError`). We should include the provider's error message for debugging. Wrapping all calls in try/catch and normalizing errors is helpful so that the developer using our unified client doesn't have to write separate error logic for each provider.

Rate limiting: We can implement a simple **leaky bucket** or queue per provider to throttle requests. For instance, using known limits (like Cohere 500/min, OpenAI 3,000 TPM, etc.), we could track usage. However, a robust implementation is complex. Alternatively, we rely on provider errors and suggest the developer handles it or configure our client with a maxRequestsPerSecond to avoid hitting limits.

Monitoring: We can emit logs or events for things like: - API call latency, - Number of tokens input/output (if available from response), - Usage quotas (OpenAI and Anthropic include token counts in the response [56] [57], we can sum them and perhaps expose a callback or telemetry hook).

Code Examples for Consuming the Unified API

Now, let's show how a developer would use this unified API in practice.

Initializing the Client:

```
const llm = new UnifiedLLMClient({
  openai: { apiKey: process.env.OPENAI_API_KEY },
  anthropic: { apiKey: process.env.ANTHROPIC_KEY },
  cohere: { apiKey: process.env.COHERE_API_KEY },
```

```
  google: { apiKey: process.env.GOOGLE_API_KEY },    // (assuming using API key
auth)
  huggingface: { apiKey: process.env.HF_API_KEY }    // if needed
  // together or bedrock could also have endpoints/keys
});
```

Basic Chat Completion (no functions, no streaming):

```
const messages: ChatMessage[] = [
  { role: 'system', content: 'You are a helpful assistant.' },
  { role: 'user', content: 'What does API abstraction mean?' }
];
const result = await llm.generateChat({ provider: 'openai', model: 'gpt-4',
messages });
console.log(result.message.content);
// -> should print the assistant's answer from GPT-4.
```

This would internally call `callOpenAI`, etc., and return the assistant message. We can do the same with any provider by just changing `provider` and `model`:

```
await llm.generateChat({ provider: 'cohere', model: 'command-nightly',
messages });
// Or Anthropic Claude:
await llm.generateChat({ provider: 'anthropic', model: 'claude-2', messages });
```

The interface remains the same.

Using Function Calling:

Suppose we want the model to be able to fetch weather via a function. We define the function and provide it:

```
const functions: FunctionDefinition[] = [{
  name: "getWeather",
  description: "Lookup weather for a city",
  parameters: {
    type: "object",
    properties: {
      location: { type: "string", description: "City name, e.g. London" }
    },
    required: ["location"]
  }
}];
// Register actual implementation in our client (so it knows what to do when
```

```
called)
llm.registerFunction("getWeather", async ({ location }) => {
  const resp = await fetch(`https://api.weatherapi.com?...&q=${location}`);
  const data = await resp.json();
  return { forecast: data.current.condition.text, temp_c: data.current.temp_c };
});

// Now use a model that supports function calling:
const messages: ChatMessage[] = [{ role: 'user', content:
"What's the weather in London?" }];

const response = await llm.generateChat({
  provider: 'openai', model: 'gpt-4',
  messages, functions
});
console.log(response.message.content);
// e.g. "The weather in London is Cloudy with 15°C."
```

What happens internally: we send the user query and function definition to GPT-4. GPT-4 decides to call `getWeather` (because the question triggers it). We detect the `function_call` in the response [7], auto-execute the `getWeather` implementation, then send the function's result back to GPT-4, which then responds with the final answer describing the weather. The unified client hides this multi-step process – from the caller's perspective it was a single `generateChat` call.

If we did the same with Cohere or Anthropic:

```
await llm.generateChat({ provider: 'cohere', model: 'command-nightly',
messages, functions });
```

The Cohere adapter would go through the tool use flow (the model would output a tool call for `getWeather`, our client executes it, etc.). The end result printed would be similar (assuming the models are smart about using the tool).

Providers that don't support functions (e.g. Google Gemini as of now) would simply ignore the `functions` parameter. Our client could log a warning or throw. Perhaps we add a flag `allowFunctionFallback` – if false, we throw an error when trying to use an unsupported feature; if true, we attempt to continue without it. For a robust system, explicitly handling this is important so you don't silently get a mediocre answer from a model that wasn't actually able to call the function.

Streaming Usage:

```
const req = { provider: 'openai', model: 'gpt-4', messages: [ {role:'user',
content:'Explain streaming in Node.js'} ] };
for await (const token of llm.streamOutput(req)) {
```

```
  process.stdout.write(token);
}
```

This would print the answer tokens from GPT-4 as they arrive. Similarly, `llm.streamOutput({ provider: 'anthropic', model: 'claude-2', messages: [...] })` would stream from Claude, etc. Under the hood, each provider's SSE is handled appropriately. The developer doesn't need to manage EventSource or manual SSE parsing for each platform.

If a developer wanted to handle the SSE events differently (say, to know when the message is done vs in progress), our unified API could emit a special terminator or an event object rather than pure strings. E.g., yield objects `{ done: false, content: "partial text" }` and then one final `{ done: true }`. But to keep it simple, we yield text chunks and the loop ends when done.

Error Handling and Resilience

Error Translation: Our client should translate common errors to a unified form. For example: - OpenAI might return a JSON: `{"error": {"message": "Rate limit exceeded", "type": "rate_limit"}}`. We catch 429 and throw, say, a `RateLimitError` with message `"OpenAI: Rate limit exceeded"`. - Anthropic might return 400 with a different error structure; we normalize it. - We include the provider name in errors to aid debugging in multi-provider contexts.

Retries: For transient errors (HTTP 502, timeouts, or rate limits if we choose to backoff), we can implement a retry mechanism. Ex: if OpenAI returns 429, wait a bit and retry up to N times. Or if network error, retry. But be careful not to double-execute function calls unintentionally (if a function itself had side effects, retrying could repeat them). Possibly, we should not automatically retry function call flows without idempotency. We can document that we perform, say, 3 retries on network errors by default.

Timeouts: Each provider call might take a while depending on model speed and prompt size. We should allow the developer to set a timeout. For instance, if the model does not respond in X seconds, abort the request (and maybe throw a TimeoutError). Node fetch supports AbortController for this. Our unified API can expose a `timeoutMs` option per call.

Logging/Monitoring: We recommend logging at **DEBUG** level the requests and responses (with sensitive info stripped). For example, log: "OpenAI request: model=gpt-4, messages length=2, funcCount=1" and "OpenAI response: 805 tokens, finish_reason=stop". Such logs help monitor usage and performance. We can also use provider-specific usage info: - OpenAI returns `usage` with token counts in the response – include that in the ChatResponse (maybe in the `raw` or an extended field). - Cohere returns `meta` with token counts and billed units [57] . - Anthropic returns `usage` with token counts [56] . We could parse these and aggregate if needed. A developer could plug in a custom logger or telemetry (like sending metrics to Datadog etc.) for each call.

Model Routing (Bonus): An advanced feature might be automatically routing to a provider based on some logic (cost, speed, etc.), but that's beyond our scope. The design here is that the caller explicitly picks the provider and model. However, once integrated, one could build a wrapper that decides "if prompt > 10k tokens, use Claude 2 (100k context); if question is about code, use GPT-4; if cost-sensitive, use a local

model", etc. Our unified API makes such routing easier to implement because the interface to invoke any model is the same.

Best Practices for Multi-Provider Integration

In building and using this unified API, keep these best practices in mind:

- **Abstract Transport & Retries:** Use a robust HTTP client that supports streaming and cancellation. For instance, in Node you might use `axios` or `node-fetch` for normal calls and a lower-level approach for SSE. Ensure to handle connectivity issues – e.g., implement retries with exponential backoff for transient failures, but avoid retrying on permanent errors (like 400 bad requests or a function call that will always fail due to invalid args). Respect provider's rate limit HTTP responses with appropriate backoff. Some providers include backoff hints in headers (e.g. `Retry-After`). Use them when available.

- **Rate Limiting:** As the integrator, you should consider enforcing rate limits **client-side** to avoid bursts that hit the provider limits. For example, using the data from provider docs, we saw Cohere allows 500/minute for chat on production keys [46]. If your application might spike above that, incorporate a token bucket or queue. Similarly, OpenAI's limits might be around 3,000 TPM for certain models – if you send a huge prompt, that could hit the limit. Monitoring the `usage` info from responses and counting requests can help you stay within bounds. The unified API can expose a config for max concurrency or global rate limit which, if set, it will queue calls and delay as needed.

- **Fallbacks and Graceful Degradation:** When a requested feature isn't supported by the chosen model, decide on a policy:

- If a function call is requested but unavailable, you might just call the model without functions and return its answer (possibly a less accurate answer). It might hallucinate the tool use or just ignore the missing capability. To illustrate, if you ask LLaMA "What's the weather in London?" and expected a tool call but it can't, it might just say "I don't have that info." You could intercept such a response and perhaps try a different provider or apologize.
- Alternatively, you can throw an error upfront ("Provider X does not support function calling") so the developer can handle it (maybe by switching provider or manually handling the query).

- For streaming not supported, you can either buffer then fake-stream the whole output (not realtime but at least same interface), or again, throw to tell the dev to not request streaming on that provider.

- **Testing Against Each API:** Because we're dealing with multiple external APIs, it's crucial to test our abstraction thoroughly for each. Write unit tests or integration tests that hit a dummy model or a lightweight prompt to ensure:

- Role mappings are correct (system prompts working on each, etc.).
- Function calls actually invoke and return final answer on each (for those that support).
- Streaming yields properly terminated sequences.

- Error paths return consistent exceptions.

- **Security:** Never log sensitive content (prompts or completions) in plaintext in production logs. If you must log, redact or hash them. Also, protect API keys carefully – our unified client should avoid accidentally sending the wrong key to the wrong endpoint (e.g., ensure that each provider's key is only used for that provider's calls).

- **Monitoring and Observability:** It's beneficial to build in hooks or callbacks for monitoring. For instance, allow the user to provide an `onLog` or `onUsage` callback in config. Then, after each call, we could call `onUsage({ provider, model, promptTokens, responseTokens, latencyMs })`. This helps track costs (especially since each provider has different pricing per token) and latencies. If one provider is consistently slow, an application might adjust usage or raise alerts.

- **Upgrading and Compatibility:** LLM APIs evolve rapidly. Design the abstraction to be flexible with new parameters. For example, OpenAI might introduce new roles or parameters (like the 2023-12 API introducing `user_profile` or others). We should allow passing extra fields through (maybe via an `options` bag that is provider-specific) for power users who need a feature that our abstraction doesn't explicitly model. Similarly, when providers upgrade (like PaLM -> Gemini, Claude versions, etc.), keep the abstraction up to date by mapping new features into the unified interface (e.g., if Google later supports function calling, we could route our `functions` there too).

Deploying the Unified API as a Service (Optional)

After implementing the unified Node.js client, you might want to expose it as a **microservice** or serverless endpoint so external applications (or front-end clients) can use multiple LLMs via a single API. Here are some considerations for that:

- **Serverless Function:** You could wrap the client in an AWS Lambda, Google Cloud Function, or Azure Function. The function would accept an HTTP request with a JSON payload similar to `ChatRequest` (provider, model, messages, etc.) and use the unified client to process it. This offloads the actual API calls to the cloud function environment (keeping your API keys safe on the server side) and returns the response to the caller. For example, an AWS Lambda in Node.js could instantiate `UnifiedLLMClient` once (reusing across invocations if warm) and handle incoming events.

- **REST Gateway:** Alternatively, build a small Express.js or Fastify server. Define endpoints like:

- `POST /chat/completion` – expects a JSON with the same structure as our `ChatRequest` (maybe slightly adjusted for HTTP, e.g. query params or headers for auth if needed) and returns the assistant response.
- `GET /chat/stream` – could upgrade to a SSE connection or WebSocket to pipe a streaming response. This is trickier due to server push needs. An easier way is to implement SSE: set the response header `Content-Type: text/event-stream` and write data events as you get them from the provider's stream, then flush.

This gateway could act as a unified **backend for your front-end**: your web app calls `/chat/completion` with `provider="openai"` or `"anthropic"`, and your service internally uses the correct API. The benefit is that your front-end doesn't need to store multiple API keys or handle different payload formats –

it only talks to your unified service. You can also enforce consistent authentication or rate limiting at this gateway level for your clients.

- **Multi-Tenancy and Keys:** If this service will serve multiple end-users or teams, you'll need a strategy for API keys. You might not want to use *your* keys for all user requests (since that could rack up cost). You could allow the client to supply an API key for a given provider (but that exposes keys to your service – possibly okay if you treat them carefully). Or you maintain a mapping of user -> keys server-side. For simplicity, if it's an internal tool, using a single set of keys might be fine. But for external consumption, you might implement an authentication layer and store each user's provider credentials encrypted, using them when that user's request comes in.

- **Scalability:** A serverless function is stateless and can scale out with concurrent executions – suitable for handling bursts of requests to different LLMs. Just be mindful of cold-start times (especially if initializing the `UnifiedLLMClient` with large libraries). Keep the client lightweight; avoid loading huge SDKs if not needed. Many API calls can be done with plain fetch and minimal overhead.

- **Cost and Policy Management:** By routing everything through one service, you can also implement centralized policies – e.g. limiting how many tokens a given user can use per day (to control cost), or logging all queries in a database for audit (ensure this is done securely and in compliance with privacy requirements). You can also swap out models under the hood: say you have a "default" provider choice or an ensemble – the gateway can decide to call multiple models and merge responses, etc., all hidden behind the unified interface.

Deploying this way turns your unified API into a **gateway** that abstracts model selection. This is similar to what some AI orchestration platforms do, but having your own gives you control. External apps then just integrate with *your* API – for them, it's one API, but behind the scenes it can reach any model as needed.

Conclusion: By implementing a unified TypeScript API client as outlined above, senior developers can seamlessly integrate multiple LLM providers into their applications. The abstraction handles the gritty details: different HTTP endpoints, message formats, streaming protocols, and feature support. This lets you focus on higher-level logic like orchestrating prompts, tools, and fallback strategies, instead of writing repetitive glue code for each LLM service. As the LLM landscape evolves (new models, new features), you can update the abstraction in one place and instantly benefit across all usage. With careful attention to error handling, rate limits, and monitoring, this unified approach can power robust, provider-agnostic AI features in production systems – whether it's for dynamic AI assistants, content generation tools, or any multi-model LLM product. By layering the abstraction into a centralized service, you further simplify consumption and maintain security of API keys and consistency of behavior.

Integrating a variety of AI models becomes much less daunting with this unified API – you get the strengths of each provider (be it OpenAI's latest GPT-4, Anthropic's large context Claude, Google's multimodal Gemini, Cohere's tool use, or the cost-savings of open-source models) all through a single, clean interface. Happy coding with your multi-LLM toolkit!

Sources:

- OpenAI Chat Completions (roles and format) [1] ; OpenAI function calling behavior [6] [7]
- Anthropic Claude API (message format and lack of system role) [14] ; Anthropic tool use definitions [16]
- Google Gemini API (content structure and streaming) [25] [51]
- Cohere Chat API (roles and response structure) [33] [57] ; Cohere function calling (tool use) flow [36] [17] ; Cohere multimodal example [35]
- Together AI (OpenAI-compatible API for open models) usage [47]
- Cohere rate limit example [46]
- Anthropic/Claude token usage in response [56] and model info [21] .

---

[1] [2] [3] Mastering Prompt Engineering: A Guide to System, User, and Assistant Roles in OpenAI API | by Mudassar Hakim | Medium
https://medium.com/@mudassar.hakim/mastering-prompt-engineering-a-guide-to-system-user-and-assistant-roles-in-openai-api-28fe5fbf1d81

[4] Can messages have metadata in chat completions? - API
https://community.openai.com/t/can-messages-have-metadata-in-chat-completions/1109098

[5] [6] [7] [8] [9] [53] Mastering function calling with OpenAI | genai-research – Weights & Biases
https://wandb.ai/onlineinference/genai-research/reports/Mastering-function-calling-with-OpenAI--VmlldzoxMzQ1MDk1NQ

[10] Hello GPT-4o | OpenAI
https://openai.com/index/hello-gpt-4o/

[11] Images and vision - OpenAI API
https://platform.openai.com/docs/guides/images-vision

[12] [13] [14] [15] [20] [56] Messages - Claude Docs
https://docs.claude.com/en/api/messages

[16] [50] How to implement tool use - Claude Docs
https://docs.claude.com/en/docs/agents-and-tools/tool-use/implement-tool-use

[17] [18] [19] [36] [37] [38] [39] [40] [41] [54] [55] Basic usage of tool use (function calling) | Cohere
https://docs.cohere.com/docs/tool-use-overview

[21] [22] Models overview - Claude Docs
https://docs.claude.com/en/docs/about-claude/models/overview

[23] PaLM API Overview - Google AI for Developers
https://ai.google.dev/palm_docs

[24] [25] [26] [27] [28] [29] [30] [31] [32] [51] Gemini API reference | Google AI for Developers
https://ai.google.dev/api

[33] [34] [49] [57] Using the Cohere Chat API for Text Generation | Cohere
https://docs.cohere.com/docs/chat-api

[35] [45] Using Cohere's Models to Work with Image Inputs | Cohere
https://docs.cohere.com/docs/image-inputs

42 43 44 52 A Guide to Streaming Responses | Cohere
https://docs.cohere.com/docs/streaming

46 Different Types of API Keys and Rate Limits | Cohere
https://docs.cohere.com/docs/rate-limits

47 Mistral API | Together AI
https://www.together.ai/models/mistral-beb7b

48 OpenAI-Compatible Server - vLLM
https://docs.vllm.ai/en/latest/serving/openai_compatible_server.html