GEB AI

Geb API Developer Guide

Build your own application on Geb. Arabic-first, OpenAI-compatible.

Geb API v2. Endpoints use the `/v2` path prefix, so OpenAI-compatible SDKs work with `base_url` `https://api.gebai.ca/v2`. The `/v1` prefix remains supported as an alias for existing clients.

Geb API v2 | Guide revision 1.1 | 28 October 2026 | Base URL `https://api.gebai.ca/v2`

Contents

1 What the API is and is not
2 Quickstart: your first call in five minutes
3 Getting a key
4 Authentication
5 Endpoints
6 Streaming
7 Tool calls and function calling
8 Reasoning
9 Errors and retries
10 Limits and quotas
11 OpenAI SDK compatibility
12 Data handling
13 Version and support

1. What the API is and is not

The Geb API gives your application the same model that answers in Geb AI, through the chat completions interface that OpenAI client libraries already speak. Point an OpenAI SDK at `https://api.gebai.ca/v2`, use a Geb key, and it works.

Geb calls this API **API v2**, and its URL paths start with `/v2`. Existing clients that still call `/v1` keep working: `/v1/...` is an alias that reaches exactly the same handlers, with the same behaviour, headers, errors and quotas, and a call counts once whichever prefix it uses. New code should use `/v2`.

What you get

- **Chat completions** at `POST /v2/chat/completions`, buffered or streamed over server-sent events.
- **Strong Arabic and multilingual replies.** Geb answers in the language of your user's latest message, in 23 languages. Arabic is a first-class language, not a translation layer.
- **Your own functions** through the standard `tools` field (function calling).

- **Geb's built-in tools**, such as web search, weather, document and spreadsheet builders, when you opt in by name.
- **Structured output** with `response_format` (`json_object` and `json_schema`).
- **Image input** on user messages.
- **Reasoning control** with effort levels.
- **Stored conversations** you can continue by id, kept apart from any account memory.
- **Reply rules** (`geb_acceptance`): declare word ranges, forbidden phrases and grounding requirements, and Geb checks and refines the reply before returning it.
- **No per-token charges.** Your plan includes a number of calls per billing period (see section 10). Tokens are not metered.

What it is not

- **Not an OpenAI proxy.** Every reply comes from Geb's own model. Your request is never handed to another vendor's model.
- **Not memory-aware.** API calls never read your Geb account memory and never write to it. A call is built from your request alone (plus a stored conversation thread if you ask for one, see section 5.5).
- **Only chat completions.** There are no embeddings, audio, image generation, files, batch, fine-tuning, Assistants or Responses endpoints.
- **Single choice only.** `n`, `logprobs` and the legacy `functions` field are rejected.
- **Some sampling fields are accepted and ignored.** `stop`, `user`, `seed`, `temperature`, `top_p`, `presence_penalty`, `frequency_penalty`, `logit_bias`, `min_p`, `top_k` and `repetition_penalty` never cause an error, and they have no effect today.
- **Not callable from a browser on someone else's website.** The API answers cross-origin preflight requests only for Geb's own sites. Call it from your server, not from client-side JavaScript. Never put a key in a web page or mobile app.

---

2. Quickstart: your first call in five minutes

1. Get a key (section 3). It looks like `geb_live_` followed by 43 characters.
2. Put it in an environment variable. Never paste it into source code.
3. Make a call.

curl

CURL

```
export GEB_API_KEY="geb_live_..."   # paste your key here, in your shell only

curl https://api.gebai.ca/v2/chat/completions \
  -H "Authorization: Bearer $GEB_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "geb",
    "messages": [
      {"role": "user", "content": "Greet me in Arabic and in English, one line each."}
    ]
  }'
```

Python

PYTHON

```
# pip install openai
import os
from openai import OpenAI

client = OpenAI(
    base_url="https://api.gebai.ca/v2",
    api_key=os.environ["GEB_API_KEY"],
)

reply = client.chat.completions.create(
    model="geb",
    messages=[{"role": "user", "content": "Greet me in Arabic and in English, one line each."}],
)
print(reply.choices[0].message.content)
```

JavaScript (Node 18 or later)

JAVASCRIPT

```
// npm install openai
import OpenAI from "openai";

const client = new OpenAI({
  baseURL: "https://api.gebai.ca/v2",
  apiKey: process.env.GEB_API_KEY,
});

const reply = await client.chat.completions.create({
  model: "geb",
  messages: [{ role: "user", content: "Greet me in Arabic and in English, one line each." }],
});
console.log(reply.choices[0].message.content);
```

What you get back

JSON

```
{
  "id": "chatcmpl-...",
  "object": "chat.completion",
  "created": 1790000000,
  "model": "<model-id>",
  "choices": [
    {
      "index": 0,
      "message": { "role": "assistant", "content": "..." },
      "finish_reason": "stop"
    }
  ],
  "usage": { "prompt_tokens": 0, "completion_tokens": 0, "total_tokens": 0 }
}
```

`model` is the id of the model that answered, as listed by `GET /v2/models`. `usage.prompt_tokens` includes Geb's own base instructions, which lead every prompt.

If you got a `401`, check the key and the `Authorization` header (section 4). If you got a `429`, read section 10.

---

3. Getting a key

Keys are created by a signed-in Geb account, in the Geb web app.

1. Create or sign in to your account at `https://gebai.ca`.
2. Open **Profile**, then **API keys** (`https://gebai.ca/profile/api-keys`).
3. Choose **Create API key**. Give it a name, and optionally an expiry in days (1 to 365).
4. Copy the key immediately. It is shown once and cannot be retrieved later. Geb stores only a hash.

Things to know:

- An account can hold up to **10 active keys**. Revoke one to free a slot.
- Revoke a key in the same page at any time. A revoked or expired key answers `401` from the next request.
- Every key belongs to the account that created it. Calls, quota and rate limits are counted per account, across all of its keys.
- The only key scope is `chat:write`, which covers every endpoint in this guide.
- Every plan can use the API, Free included: a Free account can create keys and make 500 calls per billing period. How many calls each plan includes is in section 10.

Keep keys on your server. Use one key per application or environment, so you can revoke one without touching the others.

4. Authentication

Send the key as a bearer token on every request.

HTTP

```
Authorization: Bearer geb_live_...
```

CURL

```
curl https://api.gebai.ca/v2/models -H "Authorization: Bearer $GEB_API_KEY"
```

PYTHON

```
import os, requests

headers = {"Authorization": f"Bearer {os.environ['GEB_API_KEY']}"}
print(requests.get("https://api.gebai.ca/v2/models", headers=headers).json())
```

JAVASCRIPT

```
const res = await fetch("https://api.gebai.ca/v2/models", {
  headers: { Authorization: `Bearer ${process.env.GEB_API_KEY}` },
});
console.log(await res.json());
```

A request with no key, a malformed header, a different scheme (for example `Basic`), or an unknown, revoked or expired key all return the same answer:

HTTP

```
HTTP/1.1 401 Unauthorized
Content-Type: application/json; charset=utf-8
x-request-id: 0f658b6d9742d46fec9ce4b984f99667

{"error":{"message":"A valid developer API key is
required.","type":"invalid_api_key","param":null,"code":null,"request_id":"0f658b6d9742d46fec9ce4b9
84f99667"}}
```

A `403 permission_error` means the key is valid but the account may not use the API (for example the account is suspended).

5. Endpoints

All endpoints are under `https://api.gebai.ca`, use JSON, and need the bearer key. The canonical prefix is `/v2`. The same paths with a `/v1` prefix are an alias for existing clients and behave identically. Where a response or error names a path, it names the prefix you used.

Method and path	Purpose
`GET /v2/models`	List the model you can call, with live capabilities.

Method and path	Purpose
`GET /v2/models/{id}`	Retrieve that model. Unknown id gives `404`.
`POST /v2/chat/completions`	Create a chat completion, buffered or streamed.
`GET /v2/conversations`	List your stored conversations.
`GET /v2/conversations/{id}`	Read one stored conversation.
`DELETE /v2/conversations/{id}`	Delete one stored conversation.

Request bodies may be gzip or deflate compressed (`Content-Encoding: gzip`). The maximum request body is 32 MiB after decompression.

5.1 List models

CURL

```
curl https://api.gebai.ca/v2/models -H "Authorization: Bearer $GEB_API_KEY"
```

PYTHON

```
for m in client.models.list():
    print(m.id)
```

JAVASCRIPT

```
for await (const m of client.models.list()) console.log(m.id);
```

The response is an OpenAI-shaped list. Each model carries live facts about what the API can do right now:

JSON

```
{
  "object": "list",
  "data": [
    {
      "id": "<model-id>",
      "object": "model",
      "created": 1755475200,
      "owned_by": "geb-ai",
      "context_window": 0,
      "modalities": { "input": ["text", "image"], "output": ["text"] },
      "capabilities": { "streaming": true, "tools": true, "vision": true, "reasoning": true },
      "reasoning": {
        "always_on": true,
        "efforts": ["low", "medium", "high", "ultra"],
        "default_effort": "<level>",
        "parameters": ["reasoning", "reasoning_effort", "thinking", "enable_thinking", "..."]
      },
      "limits": {
        "max_tokens": "answer_ceiling",
        "max_tokens_default": "derived_from_context_window",
        "long_non_streamed": "keepalive"
      },
      "geb_tools": ["web_search", "weather_forecast", "..."],
      "status": { "healthy": true, "state": "<state>" }
    }
  ]
}
```

- `context_window` is the number of tokens the model can hold in one request. It is `null` when Geb cannot currently measure it.
- `capabilities.vision` and `"image"` in `modalities.input` are live: they are `true` only while an image request would be accepted right now.
- `geb_tools` is the exact list of built-in tool names a request may name (section 7.2).
- The values above are illustrative. Read the real ones from your own call.

Today the API serves one Geb model. The `model` field on a completion is accepted for OpenAI compatibility (any string is fine) and does not select between models. The response reports the id that actually answered.

5.2 Create a chat completion

`POST /v2/chat/completions`

Parameter	Type	Notes
`model`	string	Optional. Accepted for compatibility, see above.
`messages`	array	Required. Roles `system`, `user`, `assistant`, `tool`. `content` is a string, or on `user` messages an array of `text` and `image_url` parts.
`stream`	boolean	`true` streams server-sent events (section 6).
`max_tokens`	integer	Ceiling on the visible answer. Thinking is budgeted separately (section 8).

Parameter	Type	Notes
`tools`, `tool_choice`, `parallel_tool_calls`		Your own functions (section 7.1).
`response_format`	object	`{"type":"json_object"}` or `{"type":"json_schema", ...}`.
`stream_options`	object	`{"include_usage": true}` adds a final usage chunk to a stream.
`reasoning`, `reasoning_effort`, `thinking`, `enable_thinking`		Reasoning control (section 8).
`geb_tools`	boolean or array	Geb's built-in tools, off by default (section 7.2).
`geb_acceptance`	object	Reply rules (section 5.6).
`conversation_id`	string	`"new"` starts a stored conversation, an id continues one (section 5.5).
`locale`	`"en"`, `"ar"`, `"fr"`	Optional hint for the reply language when the message itself does not decide it.

Accepted but ignored (no error, no effect): `stop`, `user`, `seed`, `temperature`, `top_p`, `presence_penalty`, `frequency_penalty`, `logit_bias`, `min_p`, `top_k`, `repetition_penalty`.

Rejected with `400`: `n`, `logprobs`, `functions`. `geb_memory: true` is also rejected (`memory_unavailable`), because API calls have no memory.

Unknown fields are errors. A property Geb does not know, for example `max_completion_tokens` or `store`, returns `400` with a message such as `property store should not exist`. Use `max_tokens`. If your client library adds fields on its own, remove them.

The JSON body must be valid JSON. Trailing commas, comments, `NaN` and other syntax errors are rejected with `400 invalid_json_body`. A key that appears twice in one object is not an error: as in any standard JSON parser, the last value wins. Do not send duplicate keys on purpose.

Geb's own instructions lead every prompt. Geb answers as Geb, and its safety rules cannot be overridden. Your `system` message follows and frames the conversation. These instructions count toward `usage.prompt_tokens`.

Reply language. Geb replies in the language of your user's latest message and follows a switch mid-conversation. An explicit instruction such as "answer in French" wins. A message that decides nothing ("ok", code) keeps the language of the last clear message. Code, commands and tool arguments stay in their natural language. For Arabic, your `system` message decides the register: ask for "Modern Standard Arabic" or "Egyptian Arabic". Without one, written deliverables and translations are Modern Standard Arabic, and conversation is Egyptian Arabic. A request to translate returns only the translation.

Response. A normal completion looks like the quickstart response. `finish_reason` is `stop` (finished), `length` (cut at your `max_tokens`, or cut short, see section 6.4), `tool_calls` (your function is being called), or `content_filter`. Always check it before you parse JSON out of a reply.

PYTHON

```python
r = client.chat.completions.create(
    model="geb",
    messages=[
        {"role": "system", "content": "You are a concise assistant for a travel app."},
        {"role": "user", "content": "اقترح لي خطة ليوم واحد في القاهرة"},
    ],
    max_tokens=800,
)
print(r.choices[0].message.content, r.choices[0].finish_reason, r.usage.total_tokens)
```

JAVASCRIPT

```javascript
const r = await client.chat.completions.create({
  model: "geb",
  messages: [
    { role: "system", content: "You are a concise assistant for a travel app." },
    { role: "user", content: "Plan one day in Cairo." },
  ],
  max_tokens: 800,
});
console.log(r.choices[0].message.content, r.choices[0].finish_reason, r.usage.total_tokens);
```

5.3 Structured output

CURL

```bash
curl https://api.gebai.ca/v2/chat/completions \
  -H "Authorization: Bearer $GEB_API_KEY" -H "Content-Type: application/json" \
  -d '{
    "messages": [{"role":"user","content":"Extract: Layla, 34, lives in Giza."}],
    "response_format": {
      "type": "json_schema",
      "json_schema": {
        "name": "person",
        "strict": true,
        "schema": {
          "type": "object",
          "properties": {"name":{"type":"string"},"age":{"type":"integer"},"city":
{"type":"string"}},
          "required": ["name","age","city"],
          "additionalProperties": false
        }
      }
    }
  }'
```

- `json_schema` replies are schema-valid. Send `"strict": true` to have Geb check your schema first: every object then needs `additionalProperties: false` and every property must be listed in `required`, and a schema that cannot satisfy this is rejected with `400` naming the offending path. Without a `strict` field Geb uses your schema as sent and does not reject it, so a schema that breaks the strict rules is not an error, but its replies are only as constrained as the schema is.
- Check `finish_reason`. If it is `length`, the JSON is incomplete. A cut JSON body is never reported as `stop`.

- A streamed `response_format` call is held until Geb has checked the reply, then delivered (section 6.3). Heartbeats keep the connection open meanwhile.
- Geb's built-in tools do not run on a `response_format` call.

5.4 Image input

User messages may carry images as OpenAI content parts. Images go to Geb's own vision model.

PYTHON

```python
r = client.chat.completions.create(
    model="geb",
    messages=[{
        "role": "user",
        "content": [
            {"type": "text", "text": "What is in this picture?"},
            {"type": "image_url", "image_url": {"url": "https://example.com/cat.jpg"}},
        ],
    }],
)
```

JAVASCRIPT

```javascript
const r = await client.chat.completions.create({
  model: "geb",
  messages: [{
    role: "user",
    content: [
      { type: "text", text: "What is in this picture?" },
      { type: "image_url", image_url: { url: "https://example.com/cat.jpg" } },
    ],
  }],
});
```

- Sources: `https://` URLs (fetched by Geb) or `...` URIs. `http://` and other schemes are rejected.
- Formats: PNG, JPEG, WebP, GIF (first frame). The bytes must match the declared type.
- Limits: at most 16 images per request, 20 MB per image, 64 MB of decoded image data per request. Send large images as URLs: the whole request body is capped at 32 MiB.
- URLs that resolve to private or internal addresses are refused (`image_url_forbidden`).
- Images in a request are shown with the latest user message, in order. An image-only message is valid. Parts other than `text` and `image_url` (audio, files) are rejected.
- If the vision model is unavailable at that moment you get `503 vision_unavailable`, never a text-only guess. `GET /v2/models` reports `capabilities.vision: false` in the same state.

5.5 Conversations

You may send the whole history on every request, as with any chat API. Geb can also store the thread for you.

- Send `"conversation_id": "new"`. The id comes back in the response (`conversation_id`) and in the final stream chunk.
- Send that id on later calls to continue.

- If your request contains any `assistant` or `tool` message, you sent the history: your messages are used exactly as sent. If it contains only `system` and `user` messages, Geb adds the stored thread to the prompt, newest turns first, as many as fit in the context window. The new turn is stored either way.
- A stored thread is not account memory. Chat never reads it, and no other account can read it.

Route	Result
`GET /v2/conversations?limit=50&after=<cursor>`	`{"object":"list","data":` `[{"id","object":"conversation","created_at","la` newest first.
`GET /v2/conversations/{id}?limit=100&after=<cursor>`	The conversation with `messages: [{"role","content"` `has_more` and `next_cursor`.
`DELETE /v2/conversations/{id}`	`{"id","object":"conversation.deleted","deleted`

An id that is not yours answers `404`. These three routes count as API calls against your quota, like a completion; a call that fails does not use one.

CURL

```
curl https://api.gebai.ca/v2/chat/completions \
  -H "Authorization: Bearer $GEB_API_KEY" -H "Content-Type: application/json" \
  -d '{"conversation_id":"new","messages":[{"role":"user","content":"Remember: my shop sells
dates."}]}'
```

5.6 Reply rules (`geb_acceptance`)

For work where the reply must meet a checklist, declare the rules. Geb drafts, checks, refines what fails, and returns the best version with a report. Rule checks are not billed: the call is still one call.

JSON

```
{
  "messages": [{"role": "user", "content": "Write a 300 word product summary from the sources
below. ..."}],
  "geb_acceptance": {
    "version": 1,
    "rules": [
      {"type": "word_range", "path": "", "min": 250, "max": 350},
      {"type": "forbidden_phrases", "paths": ["/**"], "phrases": ["world-class", "cutting-edge"]},
      {"type": "claims_grounded", "paths": ["/**"]}
    ],
    "on_fail": "return_best_with_report"
  }
}
```

- Rule types: `word_range`, `forbidden_phrases`, `no_repeated_paragraphs`, `cited_ids_named`, `min_named_sources`, `claims_grounded`, `no_copy_from`, `phrases_require_source`. A `path` is a JSON Pointer into the reply; on a plain-text reply use `""` or `/**`.
- The reply carries `x_geb_acceptance: {"passed": true, "rounds": 1}`, or `passed: false` with a `failures` list.
- `on_fail: "error_422"` turns a final failure into `422 geb_acceptance_failed` instead.

- An invalid contract is `400 invalid_geb_acceptance`, with `param` naming the rule and `rule_index` its position.
- A stream with `geb_acceptance` is held until checks finish (section 6.3).

5.7 Headers on every response

Header	Meaning
`x-request-id`	The id of this call, on every response, successful or not. It equals `error.request_id` on an error. Quote it to support.
`x-geb-api-version`	`2` on both the `/v2` and the `/v1` prefix.
`X-RateLimit-Limit`	The per-minute limit, 120 on every plan (section 10.2).
`X-Quota-Limit-Month`, `X-Quota-Remaining-Month`, `X-Quota-Reset`	Your monthly call quota position (section 10.1).
`x-geb-reasoning-effort`, `x-geb-reasoning-adjusted`	The reasoning level used (section 8.4).

6. Streaming

Set `"stream": true`. The response is `text/event-stream`: server-sent events, each a line `data: <json>` followed by a blank line.

CURL

```
curl -N https://api.gebai.ca/v2/chat/completions \
  -H "Authorization: Bearer $GEB_API_KEY" -H "Content-Type: application/json" \
  -d '{"stream":true,"stream_options":{"include_usage":true},
       "messages":[{"role":"user","content":"Count to five in Arabic."}]}'
```

PYTHON

```
stream = client.chat.completions.create(
    model="geb",
    messages=[{"role": "user", "content": "Count to five in Arabic."}],
    stream=True,
    stream_options={"include_usage": True},
)
for chunk in stream:
    if chunk.choices and chunk.choices[0].delta.content:
        print(chunk.choices[0].delta.content, end="", flush=True)
    if chunk.choices and chunk.choices[0].finish_reason:
        print("\nfinish_reason:", chunk.choices[0].finish_reason)
```

JAVASCRIPT

```javascript
const stream = await client.chat.completions.create({
  model: "geb",
  messages: [{ role: "user", content: "Count to five in Arabic." }],
  stream: true,
  stream_options: { include_usage: true },
});
for await (const chunk of stream) {
  const choice = chunk.choices[0];
  if (choice?.delta?.content) process.stdout.write(choice.delta.content);
  if (choice?.finish_reason) console.log("\nfinish_reason:", choice.finish_reason);
}
```

6.1 Frame order

1. A comment line of padding at the start (comments begin with `:`), which flushes proxies. Ignore it.
2. A first `chat.completion.chunk` whose `delta` carries `role: "assistant"`.
3. Content chunks, each with `delta.content` text.
4. A final chunk with `delta: {}` and a `finish_reason`. It also carries `conversation_id` when you use conversations.
5. If you set `stream_options.include_usage`: one more chunk with `"choices": []` and a `usage` object.
6. `data: [DONE]`.

STREAM

```
data: {"id":"chatcmpl-...","object":"chat.completion.chunk","model":"<model-id>","choices":
[{"index":0,"delta":{"role":"assistant","content":"One"},"finish_reason":null}]}

data: {"id":"chatcmpl-...","object":"chat.completion.chunk","model":"<model-id>","choices":
[{"index":0,"delta":{"content":", two"},"finish_reason":null}]}

data: {"id":"chatcmpl-...","object":"chat.completion.chunk","model":"<model-id>","choices":
[{"index":0,"delta":{},"finish_reason":"stop"}]}

data: {"id":"chatcmpl-...","object":"chat.completion.chunk","model":"<model-id>","choices":
[],"usage":{"prompt_tokens":0,"completion_tokens":0,"total_tokens":0}}

data: [DONE]
```

Chunks with an empty `choices` array are metadata (usage, and Geb extensions such as `x_geb_acceptance` and `x_geb_partial`). A strict parser should skip them when reading text. Geb never streams its reasoning text.

6.2 Keepalives

While Geb is thinking or checking a reply, the connection stays open and Geb writes a comment line, `: heartbeat`, about every 15 seconds. Comments are ignored by every SSE parser. If you write your own parser, skip any line that starts with `:`.

Response headers on a stream include `x-request-id`, your `X-Quota-*` headers (section 10), and the reasoning headers (section 8). If you close the connection, Geb stops generating.

6.3 Held streams

With `response_format` or `geb_acceptance`, Geb checks the reply before sending it. The stream opens with the comment `: geb is checking the reply before sending it`, then only `: heartbeat` lines, then the checked content. If you need tokens as they are generated, send neither field.

6.4 How a stream ends

A stream always ends with `data: [DONE]`, in one of three ways.

1. **Complete.** A final chunk with `finish_reason` `stop` (or `tool_calls`), then `[DONE]`.
2. **Partial reply.** If the generation stalls or fails after text has already been sent, Geb ends the stream with what it sent: a metadata chunk `{"choices":[],"x_geb_partial":{"reason":"stalled","code":"geb_dispatch_stalled","complete":false,"chars":123}}`, then a final chunk with `finish_reason: "length"`, then `[DONE]`. Treat it as an incomplete answer. `reason` is one of `stalled`, `edge_deadline`, `deadline`, `upstream_error`. Partials are offered only on plain text calls; a call with `tools` or `response_format` gets the error frame below instead.
3. **In-band error.** If nothing usable was produced, a frame `data: {"error": {...}}` followed by `data: [DONE]`. The HTTP status was already `200`, so the error cannot be an HTTP status. The official OpenAI SDKs raise an error when they see this frame.

A stream that closes **without** `[DONE]` means the connection was lost. Treat it as a failure and retry (use an `Idempotency-Key`, section 9.3).

Problems Geb can see before it starts, such as a bad key, a validation error or an exhausted quota, come back as real HTTP error statuses, not as `200`.

6.5 Parsing a stream without an SDK

PYTHON

```python
import json, os, requests

r = requests.post(
    "https://api.gebai.ca/v2/chat/completions",
    headers={"Authorization": f"Bearer {os.environ['GEB_API_KEY']}"},
    json={"stream": True, "messages": [{"role": "user", "content": "Hello"}]},
    stream=True,
)
r.raise_for_status()
done = False
for raw in r.iter_lines(decode_unicode=True):
    if not raw or raw.startswith(":"):
        continue                        # blank line or keepalive comment
    payload = raw[len("data:"):].strip()
    if payload == "[DONE]":
        done = True
        break
    frame = json.loads(payload)
    if "error" in frame:
        raise RuntimeError(frame["error"])
    for choice in frame.get("choices", []):
        print(choice["delta"].get("content") or "", end="")
if not done:
    raise RuntimeError("stream ended without [DONE]; retry")
```

JAVASCRIPT

```javascript
const res = await fetch("https://api.gebai.ca/v2/chat/completions", {
  method: "POST",
  headers: {
    Authorization: `Bearer ${process.env.GEB_API_KEY}`,
    "Content-Type": "application/json",
  },
  body: JSON.stringify({ stream: true, messages: [{ role: "user", content: "Hello" }] }),
});
if (!res.ok) throw new Error(`HTTP ${res.status}: ${await res.text()}`);
const decoder = new TextDecoder();
let buffer = "";
let done = false;
for await (const part of res.body) {
  buffer += decoder.decode(part, { stream: true });
  let i;
  while ((i = buffer.indexOf("\n")) >= 0) {
    const line = buffer.slice(0, i).trim();
    buffer = buffer.slice(i + 1);
    if (!line || line.startsWith(":")) continue;
    const payload = line.slice(5).trim();
    if (payload === "[DONE]") { done = true; continue; }
    const frame = JSON.parse(payload);
    if (frame.error) throw new Error(JSON.stringify(frame.error));
    for (const c of frame.choices ?? []) process.stdout.write(c.delta?.content ?? "");
  }
}
if (!done) throw new Error("stream ended without [DONE]; retry");
```

6.6 Long buffered calls

A non-streamed call that is still running after about 70 seconds gets its `200` headers early, followed by a single space character about every 25 seconds until the JSON body arrives. A leading space is valid JSON, so `JSON.parse`, `fetch().json()`, Python `requests` and the OpenAI SDKs read the body unchanged. The response carries `x-geb-keepalive: on` when this happened.

If a call fails after those headers were sent, the status stays `200` and the body is the error object, so when `x-geb-keepalive: on` is present, check for a top-level `error` key. Send the request header `x-geb-keepalive: off` to opt out and get a `504 geb_edge_deadline` at about 95 seconds instead.

For long generations, stream. A streamed response has no total time limit while it is making progress.

---

7. Tool calls and function calling

There are two different kinds of tools. They are independent and can be sent together.

	`tools` (your functions)	`geb_tools` (Geb's built-in tools)
Whose functions	Yours	Geb's: web search, weather, documents, code
Who runs them	You	Geb
Side effects	None on Geb's side	Real (for example a generated file)
Default	Used when you send them	**Off**

If you send both, the model is shown your functions, any calls it returns are yours to execute, and Geb's own tool runner does not run on that turn.

7.1 Your functions (`tools`)

You describe your functions, Geb decides when to call one, and you run it and send the result back. This is the standard OpenAI protocol, so SDKs work unchanged.

CURL

```
curl https://api.gebai.ca/v2/chat/completions \
  -H "Authorization: Bearer $GEB_API_KEY" -H "Content-Type: application/json" \
  -d '{
    "parallel_tool_calls": false,
    "messages": [{"role":"user","content":"What is the weather in Alexandria?"}],
    "tools": [{
      "type": "function",
      "function": {
        "name": "get_weather",
        "description": "Current weather for a city.",
        "parameters": {
          "type": "object",
          "properties": {"city": {"type": "string"}},
          "required": ["city"]
        }
      }
    }]
  }'
```

The model answers with `finish_reason: "tool_calls"` and `message.content` set to `null`:

JSON

```
{
  "choices": [{
    "index": 0,
    "message": {
      "role": "assistant",
      "content": null,
      "tool_calls": [{
        "id": "chatcmpl-tool-9d22d13b24ef97ed",
        "type": "function",
        "function": { "name": "get_weather", "arguments": "{\"city\": \"Alexandria\"}" }
      }]
    },
    "finish_reason": "tool_calls"
  }]
}
```

Run the function, then send a second request containing your history, the assistant turn exactly as received, and one `tool` message per call:

PYTHON

```python
import json

tools = [{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "Current weather for a city.",
        "parameters": {
            "type": "object",
            "properties": {"city": {"type": "string"}},
            "required": ["city"],
        },
    },
}]
messages = [{"role": "user", "content": "What is the weather in Alexandria?"}]

first = client.chat.completions.create(
    model="geb", messages=messages, tools=tools, parallel_tool_calls=False
)
call = first.choices[0].message.tool_calls[0]
result = {"city": "Alexandria", "temp_c": 27, "sky": "clear"}   # your code runs here

messages.append(first.choices[0].message)
messages.append({
    "role": "tool",
    "tool_call_id": call.id,
    "name": call.function.name,
    "content": json.dumps(result),
})
final = client.chat.completions.create(model="geb", messages=messages, tools=tools)
print(final.choices[0].message.content)
```

JAVASCRIPT

```javascript
const tools = [{
  type: "function",
  function: {
    name: "get_weather",
    description: "Current weather for a city.",
    parameters: { type: "object", properties: { city: { type: "string" } }, required: ["city"] },
  },
}];
const messages = [{ role: "user", content: "What is the weather in Alexandria?" }];

const first = await client.chat.completions.create({
  model: "geb", messages, tools, parallel_tool_calls: false,
});
const call = first.choices[0].message.tool_calls[0];
const result = { city: "Alexandria", temp_c: 27, sky: "clear" };   // your code runs here

messages.push(first.choices[0].message);
messages.push({
  role: "tool",
  tool_call_id: call.id,
  name: call.function.name,
  content: JSON.stringify(result),
});
const final = await client.chat.completions.create({ model: "geb", messages, tools });
console.log(final.choices[0].message.content);
```

Rules:

- `tools[].type` must be `"function"`. `function.name` matches `^[a-zA-Z0-9_-]{1,64}$` and must be unique. `function.parameters` is a JSON Schema object. At most 128 tools, and at most 128 KiB serialized.
- `function.arguments` is a JSON-encoded **string**, in both directions. A `tool` message `content` must be a string: serialize your result yourself.
- **Answer every call.** If an assistant turn issued three calls, send three `tool` messages before the next completion. Otherwise you get `400 unanswered_tool_call` naming the missing ids. This is deliberate: left unanswered, a model may invent the missing results.
- `tool_choice`: omitted or `"auto"` lets the model decide, `"none"` forbids calls, `"required"` demands at least one, and `{"type":"function","function":{"name":"..."}}` allows only that function. With `"required"` or a named function, if the model answers in prose instead of calling, Geb asks it once more before replying. That costs one more model pass of latency, not a second call against your quota. If the model still answers in prose, you receive that prose with `finish_reason: "stop"`, so check `finish_reason` before you read `tool_calls`. With a named choice Geb also filters the response to that function. If you need the strongest guarantee, use `"required"` with a one-function catalogue. On a stream with `"required"` or a named function, text is held until the turn ends and then sent, so a prose answer arrives together at the end.
- Send `"parallel_tool_calls": false` unless you want several calls per turn. It gives exactly one clean call per turn.
- Geb removes byte-identical duplicate calls and drops a trailing call that was cut off by `length`. Arguments are otherwise passed through verbatim, including arguments that are not valid JSON, so validate them before use.

- Streaming works. Tool calls arrive as standard `delta.tool_calls` frames (an opening frame with `id` and `name`, then `arguments` fragments, accumulated per `index`). They arrive together at the end of the turn, not interleaved with generation. The accumulated result is identical to an OpenAI stream.
- A tool round trip is two requests, and each counts as a call.

7.2 Geb's built-in tools (`geb_tools`)

By default an API call is a plain model call: one generation, no tools, no side effects. Opt in with `geb_tools`.

Value	Behaviour
absent, `false`, `[]`	No Geb tools. This is the default.
`true`	Every Geb tool available right now. The model decides when to use them.
`["web_search", "weather_forecast"]`	Only the named tools. The model is shown exactly these, and a call to any other is refused.

CURL

```
curl https://api.gebai.ca/v2/chat/completions \
  -H "Authorization: Bearer $GEB_API_KEY" -H "Content-Type: application/json" \
  -d '{
    "messages": [{"role":"user","content":"What did the ECB decide at its last meeting?"}],
    "geb_tools": ["web_search"]
  }'
```

PYTHON

```
r = client.chat.completions.create(
    model="geb",
    messages=[{"role": "user", "content": "Weather in Cairo this weekend?"}],
    extra_body={"geb_tools": ["weather_forecast"]},   # the SDK merges extra_body into the request body
)
```

JAVASCRIPT

```
const r = await client.chat.completions.create({
  model: "geb",
  messages: [{ role: "user", content: "Weather in Cairo this weekend?" }],
  geb_tools: ["weather_forecast"],   // TypeScript: add // @ts-expect-error, the SDK types do not know this field
});
```

The public tool names (the live list for your call is `geb_tools` on `GET /v2/models`):

Name	What it does
`web_search`	Web search: titles, links and snippets.
`web_grounded_search`	Web search that also reads the top pages, for cited answers.
`url_fetch`	Fetches and reads one URL.

Name	What it does
`polinex_news`	Recent news items from the Polinex feed for the countries it covers.
`weather_forecast`	Current conditions and forecast for a place.
`currency_convert`	Live currency conversion.
`wikipedia_summary`	Wikipedia article summary.
`places_search`	Places and businesses near a location.
`geocode`	Address to coordinates and back.
`directions`	A route between places.
`distance_matrix`	Distance and travel time between several points.
`static_map`	A map image for a location.
`code_execute`	Runs Python in an isolated sandbox. Files it writes are returned as links.
`generate_text_file`	Produces a text file (md, csv, json, html and more) and returns a download link.
`build_document`	Builds a Word (.docx) document and returns a download link.
`build_workbook`	Builds an Excel (.xlsx) workbook and returns a download link.
`build_slides`	Builds a PowerPoint (.pptx) deck and returns a download link.

- A name Geb does not recognise, or one that is switched off, returns `400 invalid_geb_tools` with the list of valid names in `error.valid_geb_tools`. At most 32 names.
- With a list, the restriction is enforced twice: the model only sees the named tools, and any call to another tool is not executed (the model gets a tool error and answers without it).
- The three `build_*` tools are separate on purpose. Naming `build_document` allows a Word file only.
- Document building starts from your own instruction text, never from file-format words inside content you pasted. If the only mention of "xlsx" or "slides" is inside a code block or a long pasted body, Geb answers in text.
- Tools can take longer than a plain completion. Files produced by tools are stored under your account and returned as time-limited links.
- Your account memory, chat history, connectors (Google, Microsoft, Dropbox and so on) and other account tools are not available on the API.

8. Reasoning

Geb always thinks before it answers. You choose how hard. Geb does not return its reasoning text; you receive the answer, and the thinking tokens are included in `usage.completion_tokens`.

8.1 Levels

Level	Meaning
`low`	Shortest thinking.
`medium`	The default for harder questions.
`high`	Long thinking. Maps to the model's top setting.
`ultra`	The model's top setting.

`GET /v2/models` lists the levels (`reasoning.efforts`) and the server default (`reasoning.default_effort`).

8.2 What "reasoning off" means

Thinking is never switched off. Every "off" spelling (`"off"`, `"none"`, `false`, `"disabled"`, `0`) keeps thinking on with a **thinking budget of zero tokens**: the model closes its reasoning at once and starts the answer. You get a normal answer with no reasoning step. Use it for simple, latency-sensitive requests. For hard problems, prefer `medium` or `high`.

8.3 How to ask

Use whichever spelling your client already sends. None of them is ever rejected.

Parameter	Examples
`reasoning`	`"high"`, `"off"`, `"on"`, `{"effort": "medium"}`
`reasoning_effort`	`"low"`, `"medium"`, `"high"`, `"ultra"`, `"minimal"`, `"none"`, `"xhigh"`
`thinking`	`{"type": "enabled", "budget_tokens": 3000}`, `{"type": "disabled"}`, `true`
`enable_thinking`	`true`, `false`
`chat_template_kwargs`	`{"enable_thinking": false}`, `{"reasoning_effort": "high"}`
`extra_body`	`{"reasoning_effort": "medium"}`, for clients that send `extra_body` literally

Mappings: `minimal` becomes `low`. `xhigh`, `max` and `maximum` become `ultra`. A `budget_tokens` value maps to the nearest level (up to 1024 is `low`, up to 2048 `medium`, up to 4096 `high`, above that `ultra`). An unknown level (for example `"turbo"`) uses the server default.

If you send several spellings that disagree, the first one in this list that names a level wins: `reasoning`, `reasoning_effort`, `thinking`, `enable_thinking`, `chat_template_kwargs`, `extra_body`. A spelling that only switches thinking on never overrides a level named elsewhere.

CURL

```
curl https://api.gebai.ca/v2/chat/completions \
  -H "Authorization: Bearer $GEB_API_KEY" -H "Content-Type: application/json" \
  -d '{"reasoning_effort":"high","messages":[{"role":"user","content":"Plan a 3-city sales route to minimise travel."}]}' -i
```

PYTHON

```python
r = client.chat.completions.create(
    model="geb",
    messages=[{"role": "user", "content": "Plan a 3-city sales route to minimise travel."}],
    reasoning_effort="high",
)
# "ultra" is not in the SDK's typed list, send it through extra_body:
r = client.chat.completions.create(
    model="geb",
    messages=[{"role": "user", "content": "Prove that the square root of 2 is irrational."}],
    extra_body={"reasoning": "ultra"},
)
```

JAVASCRIPT

```javascript
const r = await client.chat.completions.create({
  model: "geb",
  messages: [{ role: "user", content: "Plan a 3-city sales route to minimise travel." }],
  reasoning_effort: "high",
});
```

8.4 What comes back

Response header	Value
`x-geb-reasoning-effort`	The level used, or `default` when you sent none.
`x-geb-reasoning-adjusted`	Present when a value was mapped or ignored, for example `enable_thinking=false->budget 0 (thinking stays on; think block closed)`.

8.5 max_tokens and thinking

`max_tokens` is the ceiling on the **visible answer**. Thinking is budgeted separately on top of it, so a small `max_tokens` does not starve the reasoning, and Geb never multiplies your number. If the answer reaches the ceiling it is cut there and `finish_reason` is `length`. When you do not set `max_tokens`, the answer budget is derived from the context window. `usage.completion_tokens` includes thinking tokens.

---

9. Errors and retries

9.1 The envelope

Every error response from a `/v2` route (and from its `/v1` alias) is JSON in one shape:

JSON

```
{
  "error": {
    "message": "A human-readable explanation.",
    "type": "invalid_request_error",
    "param": "messages",
    "code": "invalid_json_body",
    "request_id": "0f658b6d9742d46fec9ce4b984f99667"
  }
}
```

- `type` is the class of error. Branch on `type` and `code`, never on `message` wording.
- `param` names the offending field when one is to blame, otherwise `null`.
- `code` is a more specific machine-readable reason, or `null`.
- `request_id` is always present and is also sent as the `x-request-id` response header. Every response carries `x-request-id`, successful ones too. Quote it when you contact support. Send your own `x-request-id` if you like; Geb may still return its own trace id.
- Some errors add `retry_after_ms` at the top level next to `error`, and `Retry-After` as a header (seconds).
- Failures after a stream has started arrive in-band (section 6.4). Failures after a long buffered call has started arrive as a `200` body with an `error` key (section 6.6).
- A request to a path that does not exist returns a plain `{"message", "error", "statusCode"}` 404, not this envelope. If you see it, check the URL.

9.2 Catalogue

HTTP	type	code	Meaning and what to do
400	`invalid_request_error`	`null`	Validation failure: unknown property, wrong type, empty `messages`, unsupported field (`n`, `logprobs`, `functions`), message too large. Fix the request.
400	`invalid_request_error`	`invalid_json_body`	The body is not strict JSON.
400	`invalid_request_error`	`invalid_content_encoding`	The body is not valid gzip or deflate for its `Content-Encoding`.
400	`invalid_request_error`	`invalid_geb_tools`	Bad `geb_tools`. `error.valid_geb_tools` lists the accepted names.
400	`invalid_request_error`	`invalid_geb_acceptance`	Bad `geb_acceptance`. `param` and `rule_index` locate the rule.

HTTP	type	code	Meaning and what to do
400	`invalid_request_error`	`memory_unavailable`, `invalid_geb_memory`	`geb_memory: true`, or a non-boolean. API calls have no memory.
400	`invalid_request_error`	`unsupported_tool_choice_mode`, `unknown_tool`, `duplicate_tool_name`, `too_many_tools`, `tools_too_large`, `invalid_type`, `invalid_value`, `missing_required_parameter`	A `tools` or `tool_choice` problem. `param` points at the path, for example `tools[2].function.name`.
400	`invalid_request_error`	`unknown_tool_call_id`, `unanswered_tool_call`	A `tool` message does not match an earlier call, or a call has no result.
400	`invalid_request_error`	`invalid_content`, `unsupported_content_part`, `invalid_image_url`, `unsupported_image_type`, `invalid_image`, `too_many_images`, `image_too_large`, `images_too_large`, `image_url_forbidden`, `image_fetch_failed`, `image_fetch_timeout`	An image or content-part problem (section 5.4).
401	`invalid_api_key`	`null`	Missing, malformed, unknown, revoked or expired key. Do not retry.
403	`permission_error`	`null`	The key is valid but the account may not use the API. Do not retry.
404	`not_found_error`	`null`	Unknown model id on retrieve, or a conversation that is not yours.
409	`idempotency_key_reuse`, `idempotent_request_in_progress`	the same name as `type`	Same `Idempotency-Key` with a different body, or the first call is still running (section 9.3).
413	`invalid_request_error`	`request_too_large`	The decompressed body is over the 32 MiB limit.
415	`invalid_request_error`	`unsupported_content_encoding`	`Content-Encoding` other than gzip or deflate.

HTTP	type	code	Meaning and what to do
422	`invalid_request_error`	`geb_acceptance_failed`	`on_fail: "error_422"` and the reply still failed your rules. Not counted against your quota.
429	`rate_limit_exceeded`	`null`	Per-minute burst limit. Wait `Retry-After` seconds.
429	`rate_limit_error`	`geb_concurrency_limit`	Too many requests in flight at once for your account. Wait `Retry-After`, or for one to finish.
429	`quota_exceeded`	`null`	Monthly call quota used up. Waiting for the reset is the only fix, or a plan change.
502	`server_error`	`empty_answer`	The model finished without an answer. Retry **immediately**, the same request usually succeeds. `retry_after_ms` is 0.
503	`server_busy`	`null` or `maintenance`	Geb is busy or briefly unavailable. Wait `Retry-After` (about 5 seconds when busy, up to 120 for maintenance).
503	`server_error`	`vision_unavailable`	The vision model is unavailable right now. Retry later or send without images.
503	`server_error`	`degenerate_output`	The reply was cut for quality reasons and no clean retry was available. Retry.
503	`server_error`	`geb_instance_shutdown`	The server instance is restarting. Retry immediately.
504	`server_error`	`geb_edge_deadline`, `geb_dispatch_stalled`, `geb_dispatch_deadline`	The request took too long or made no progress. Retry with `"stream": true`.
5xx	`server_error`	`null`	Anything else. The message is deliberately generic. Retry with backoff and quote `request_id` if it persists.

On a stream the same errors arrive as the in-band error frame, plus `geb_incomplete_stream` (the generation ended before it finished; retry).

9.3 Retry guidance

- **Never retry** `400`, `401`, `403`, `404`, `409 idempotency_key_reuse`, `413`, `415`, `422`. The request must change.
- **Retry after the wait** for `429 rate_limit_exceeded` and `geb_concurrency_limit` (use `Retry-After`), and `503 server_busy`.
- **Do not hammer** `429 quota_exceeded`. Read `reset_at` in the body or `X-Quota-Reset`, and stop until then.
- **Retry immediately, once or twice** for `502 empty_answer` and `503 geb_instance_shutdown`.
- **Switch to streaming, then retry** for `504`.
- **Add jittered exponential backoff** for other `5xx`, with a cap of a few attempts.
- **Always send an `Idempotency-Key`** on calls you may retry. It is any string up to 255 characters, unique per logical request. The first call runs normally. A repeat with the same key and the same body returns the stored answer (`Idempotency-Replayed: true`), with no new generation and no quota used, for 24 hours. The replay has the same content, fields and values as the first answer; the order of keys inside the JSON object may differ, so compare parsed JSON, not bytes. The same key with a different body is `409 idempotency_key_reuse`. A repeat while the first is still running is `409 idempotent_request_in_progress`: wait, do not spin. A stream that failed part way is not stored, so a retry runs again. The `Idempotency-Status` header says `stored`, `replayed` or `disabled`.
- **Set your client timeout generously, or stream.** The OpenAI SDKs send their timeout as a header, and Geb returns the best partial answer (`finish_reason: "length"`, `x_geb_partial`) rather than nothing when your own timeout nears.

Every call ends with one of three things: the answer, a partial answer marked `finish_reason: "length"` with `x_geb_partial`, or a structured error. A non-streamed partial looks like this:

JSON

```
{
  "choices": [{"index": 0, "message": {"role": "assistant", "content": "..."}, "finish_reason":
"length"}],
  "x_geb_partial": {"reason": "stalled", "code": "geb_dispatch_stalled", "complete": false,
"chars": 812}
}
```

9.4 A retry helper

PYTHON

```python
import os, random, time, uuid, requests

URL = "https://api.gebai.ca/v2/chat/completions"
HEADERS = {"Authorization": f"Bearer {os.environ['GEB_API_KEY']}"}

def complete(body, attempts=5):
    key = str(uuid.uuid4())                       # same key on every attempt
    for n in range(attempts):
        r = requests.post(URL, json=body, headers={**HEADERS, "Idempotency-Key": key}, timeout=620)
        if r.ok:
            data = r.json()
            if "error" in data:                   # error after a keepalive commit
                raise RuntimeError(data["error"])
            return data
        err = r.json().get("error", {})
        code, kind = err.get("code"), err.get("type")
        if r.status_code in (400, 401, 403, 404, 413, 415, 422) or kind == "quota_exceeded":
            raise RuntimeError(f"{r.status_code} {kind}/{code}: {err.get('message')}
({err.get('request_id')})")
        if code in ("empty_answer", "geb_instance_shutdown"):
            wait = 0
        elif r.status_code == 504:
            raise RuntimeError("timed out: retry with stream=True")
        else:
            wait = float(r.headers.get("Retry-After", 2 ** n)) + random.random()
        time.sleep(wait)
    raise RuntimeError("gave up after retries")
```

JAVASCRIPT

```javascript
import { randomUUID } from "node:crypto";

export async function complete(body, attempts = 5) {
  const key = randomUUID();                       // same key on every attempt
  for (let n = 0; n < attempts; n++) {
    const res = await fetch("https://api.gebai.ca/v2/chat/completions", {
      method: "POST",
      headers: {
        Authorization: `Bearer ${process.env.GEB_API_KEY}`,
        "Content-Type": "application/json",
        "Idempotency-Key": key,
      },
      body: JSON.stringify(body),
    });
    const data = await res.json();
    if (res.ok && !data.error) return data;
    const err = data.error ?? {};
    if ([400, 401, 403, 404, 413, 415, 422].includes(res.status) || err.type === "quota_exceeded")
      throw new Error(`${res.status} ${err.type}/${err.code}: ${err.message} (${err.request_id})`);
    if (res.status === 504) throw new Error("timed out: retry with stream: true");
    const wait = ["empty_answer", "geb_instance_shutdown"].includes(err.code)
      ? 0
      : (Number(res.headers.get("Retry-After")) || 2 ** n) * 1000 + Math.random() * 1000;
    await new Promise((r) => setTimeout(r, wait));
  }
  throw new Error("gave up after retries");
}
```

10. Limits and quotas

10.1 Monthly call quota

Every plan includes the API, Free included, with a number of API calls per billing period. Tokens are not metered and there is no per-token charge.

Plan	Calls per period
Free	500
Basic	1,500
Pro	10,000
Master	100,000
Developer	Unlimited

- The period is your subscription billing period, or the calendar month (UTC) if you have no subscription.
- Only successful calls count. A call that fails does not use quota. A replay served from an `Idempotency-Key` does not use quota. The conversation routes count as calls. A tool round trip is two calls.
- Over quota, the API answers `429 quota_exceeded` with the details:

JSON

```
{
  "error": {
    "type": "quota_exceeded",
    "message": "Monthly API request quota reached: your Free plan includes 500 calls per billing
period and 500 have been used. The quota resets at 2026-11-01T00:00:00.000Z.",
    "tier": "free",
    "limit": 500,
    "used": 500,
    "remaining": 0,
    "reset_at": "2026-11-01T00:00:00.000Z",
    "request_id": "..."
  }
}
```

Every successful response carries your position in headers: `X-Quota-Limit-Month`, `X-Quota-Remaining-Month` and `X-Quota-Reset`. A quota `429` carries them too, with `Retry-After` set to the seconds until the reset.

CURL

```
curl -s -D - -o /dev/null https://api.gebai.ca/v2/models -H "Authorization: Bearer $GEB_API_KEY" |
grep -i "x-quota\|x-ratelimit"
```

PYTHON

```
r = requests.post(URL, json=body, headers=HEADERS)
print(r.headers.get("X-Quota-Remaining-Month"), "calls left until", r.headers.get("X-Quota-Reset"))
```

JAVASCRIPT

```
const res = await fetch(url, { method: "POST", headers, body });
console.log(res.headers.get("X-Quota-Remaining-Month"), "calls left until", res.headers.get("X-
Quota-Reset"));
```

10.2 Rate limits

Limit	Value
Requests per minute (burst), per account	120 on every plan: Free, Basic, Pro, Master and Developer. Counted across all of an account's keys.
Concurrent requests, per account	A fair share of the capacity available at that moment. Over it: `429` `geb_concurrency_limit`.

API usage is metered by the monthly call count (section 10.1). The per-minute limit exists only to stop bursts and abuse, so it is the same for every plan. It is sent as `X-RateLimit-Limit`. A burst `429` carries `Retry-After`. Spread bursts out, or stream fewer, longer requests.

10.3 Request limits

Limit	Value
Request body	32 MiB after decompression. gzip and deflate accepted.
Messages and message size	Bounded by the model's context window (`context_window` on `GET /v2/models`). Over the bound is a `400` that names it.
Images	16 per request, 20 MB each, 64 MB decoded in total.
Your tools (`tools`)	128 functions, 128 KiB serialized.
`geb_tools`	32 names.
`Idempotency-Key`	255 characters, kept 24 hours.
Active keys per account	10.
Stored conversations	No limit, kept until you delete them.

10.4 Time

Streaming has no total time limit while the model is making progress. A non-streamed call is kept alive past the edge timeout (section 6.6). A call that makes no progress for too long ends as a partial answer or a `504` rather than hanging.

---

11. OpenAI SDK compatibility

The OpenAI Python and JavaScript SDKs work with two changes: the base URL and the key.

PYTHON

```python
from openai import OpenAI
client = OpenAI(base_url="https://api.gebai.ca/v2", api_key="geb_live_...")
```

JAVASCRIPT

```javascript
import OpenAI from "openai";
const client = new OpenAI({ baseURL: "https://api.gebai.ca/v2", apiKey: "geb_live_..." });
```

SHELL

```shell
# Anything that reads the standard environment variables also works:
export OPENAI_BASE_URL="https://api.gebai.ca/v2"
export OPENAI_API_KEY="geb_live_..."
```

What works the same: `chat.completions.create` (buffered and streamed), `models.list` and `models.retrieve`, function calling, `response_format`, `stream_options.include_usage`, image content parts, and OpenAI-shaped errors with `Retry-After`.

Differences to plan for:

- **Only chat completions.** Calls to embeddings, audio, images, files, batches or the Responses API do not exist.
- **Unknown fields return 400.** Pass only fields listed in section 5.2. Notably use `max_tokens`, not `max_completion_tokens`, and do not send `store` or `metadata`.
- **Geb's own fields** (`geb_tools`, `geb_acceptance`, `conversation_id`, `locale`) go through `extra_body` in Python (the SDK merges it into the request body). In JavaScript, pass them as extra properties.
- **Sampling fields are ignored.** `temperature`, `top_p`, `stop`, `seed` and the penalties are accepted and have no effect.
- **One model.** `model` does not select anything. Use `client.models.list()` to see what is served.
- **Reasoning levels.** `reasoning_effort` accepts `low`, `medium`, `high`, and also `ultra` (send it via `extra_body` if your SDK version types the field).
- **Server-side use.** Do not call the API from a browser on your own domain.
- **Extra response fields.** Responses may carry Geb fields such as `conversation_id` and `x_geb_*`. They are additive.
- **Retries.** The SDKs retry on `429` and `5xx` by themselves. Add an `Idempotency-Key` via `extra_headers` so a retried call is not run, or counted, twice:

PYTHON

```python
client.chat.completions.create(
    model="geb",
    messages=[{"role": "user", "content": "Hello"}],
    extra_headers={"Idempotency-Key": "order-7421-reply-1"},
)
```

JAVASCRIPT

```javascript
await client.chat.completions.create(
  { model: "geb", messages: [{ role: "user", content: "Hello" }] },
  { headers: { "Idempotency-Key": "order-7421-reply-1" } },
);
```

12. Data handling

- **No account memory.** API calls never read from or write to your Geb account memory. This holds for every key and every plan.
- **Stored conversations** exist only if you use `conversation_id`. They are kept apart from memory, readable only by your account, and deleted when you delete the conversation.
- **API logs.** Each call is recorded for operations, support and abuse prevention: the request and reply content, model, parameters, token counts, latency, status, your user agent, the key id and the caller's country. Credentials and secrets found in the content (keys, tokens, passwords, signed URLs) are redacted before storage. Inline images are stored as size and hash only. A copy of each call is also visible to you, and deletable by you, in your own data in the Geb app. The operational record is kept for the life of the account and deleted when the account is deleted. API logs are never used as memory.
- **Tool and generated files** are stored under your account and returned as time-limited links.

Do not send data you are not allowed to send to a third-party service. If you need a data processing agreement, contact Geb before launch.

13. Version and support

Geb API v2. Served at `https://api.gebai.ca/v2`, with `/v1` kept as an alias for existing clients. Guide revision 1.1, 28 October 2026.

This guide describes the behaviour of the live API on that date. Fields marked ignored may be honoured later; new optional fields may appear in responses. Treat unknown response fields as additive.

When you need help, include the `request_id` from the error body (or the `x-request-id` header), the time of the call and the request shape without the key. Reach Geb through `https://gebai.ca/contact`.