TOKEN FACTORY · API

Token Factory API Developer Guide

OpenAI-compatible serverless inference on curated open models: authentication, models, chat completions with streaming, reasoning and tool calling, rate limits, errors and usage reporting.

OpenAI-compatible | Streaming (SSE) | Tool calling | Reasoning control | Usage & billing API

API version: public v1

November 2026

Contents

1	Overview and base URL	models, compatibility
2	How a request flows	diagram
3	Quickstart	first call in three steps
4	Authentication and API keys	create, use, rotate, revoke
5	Models	GET /models, GET /models/{model}
6	Chat completions	request, response, streaming
7	Reasoning (thinking) control	Qwen3 models
8	Tool calling	function calling round trip
9	Safe retries and response headers	Idempotency-Key, X-Request-Id
10	Rate limits and quotas	headers, 429, 402
11	Errors	envelope, codes, handling
12	Team API: keys, usage, billing and metrics	programmatic account access
13	Endpoint reference	one-page summary
14	Known issues	current behaviour and workarounds

Conventions used in this guide

Placeholder	Meaning
`https://<your-panel-domain>`	The address of your panel, as you use it in the browser.
`$TF_BASE_URL`	`https://<your-panel-domain>/api/tf/public/v1`
`$TF_API_KEY`	Your Token Factory API key, `hai_tf-` followed by 32 hexadecimal characters.
`<team_id>`	Your team's ID, used only by the Team API in section 12.

All timestamps are RFC 3339 (UTC). Response values shown in examples are illustrative; field names and shapes are exact.

```
# Shell setup used by the curl examples
export TF_BASE_URL="https://<your-panel-domain>/api/tf/public/v1"
export TF_API_KEY="hai_tf-<your-key>"
```

1. Overview and base URL

Token Factory gives your applications serverless access to open large language models hosted on dedicated GPUs. You do not manage servers, GPUs or model weights: you create an API key, pick a model and send requests. You pay for the tokens you use.

The inference API is **OpenAI-compatible**. Any client or framework that talks to the OpenAI Chat Completions API (the official OpenAI SDKs for Python and JavaScript, LangChain, LlamaIndex, curl and others) works by changing two settings: the **base URL** and the **API key**.

Base URL	`https://<your-panel-domain>/api/tf/public/v1`
Protocol	HTTPS, JSON request and response bodies (UTF-8)
Authentication	`Authorization: Bearer hai_tf-…`
Streaming	Server-Sent Events (`text/event-stream`) when `"stream": true`
Max request body	8 MB per inference request

Available models

The models your key may call are always listed by `GET /models` (section 5). At the time of writing the catalogue is:

Model ID (use in requests)	Name	Context window	Tool calling	Reasoning
`qwen-qwen3-1-7b`	Qwen3-1.7B	40,960 tokens	Yes	Yes, on by default; can be switched off
`qwen-qwen3-8-27b-fp8`	Qwen3.8-27B-FP8	262,144 tokens	Yes	Yes, on by default; can be switched off

> **CHOOSING A MODEL**
> `qwen-qwen3-1-7b` is small and very fast: a good fit for classification, extraction, routing and short answers. `qwen-qwen3-8-27b-fp8` is the more capable model for coding, multi-step reasoning, agents and long documents, with a 256K-token context window.

Endpoints at a glance

Method	Path (relative to base URL)	Purpose
GET	`/models`	List the models your API key may call
GET	`/models/{model}`	Retrieve one model
POST	`/chat/completions`	Generate a chat completion, buffered or streamed

Account management (API keys, usage, billing ledger, prices, limits and latency metrics) is available through the Team API described in section 12.

2. How a request flows

Every inference call follows the same path. The gateway authenticates your key, confirms your team may use the requested model, applies your rate limits and quota, forwards the request to a GPU serving the model, and records the token usage that you are billed for.

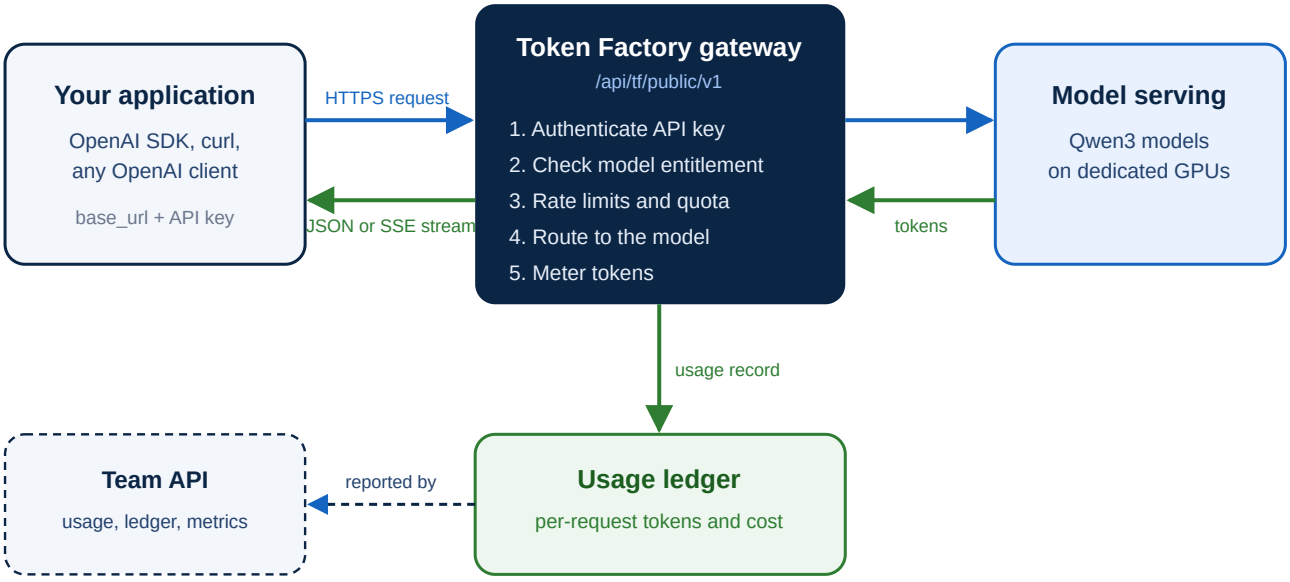


Figure 1. Request path for `/chat/completions`. Refusals (invalid key, unknown model, rate limit) are answered by the gateway before any GPU work is done and are not billed.

Key properties

- **Your key identifies your team.** There is no team parameter on the inference API; usage is always attributed to the team that owns the key.
- **One key, every model.** A key is not bound to a model; you choose the model per request with the `model` field.
- **Refused calls cost nothing.** Authentication, entitlement, rate-limit and quota checks happen before the model is called.
- **Exact metering.** Every billed call is recorded with its input, cached-input and output token counts, so the Team API's usage reports match the `usage` block you received.

3. Quickstart

1. **Create an API key.** In your panel, open the Token Factory Playground, go to **Keys** and choose **Create key**. Copy the secret when it is shown: it is displayed only once. (Section 4 also shows how to create keys through the API.)

2. **Install an OpenAI client.** `pip install openai` for Python, or `npm install openai` for JavaScript/TypeScript.

3. **Point the client at Token Factory and send a request.**

CURL

```
curl "$TF_BASE_URL/chat/completions" \
  -H "Authorization: Bearer $TF_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "qwen-qwen3-8-27b-fp8",
    "messages": [{"role": "user", "content": "What is the capital of Nepal? Answer in one
sentence."}],
    "max_tokens": 200,
    "chat_template_kwargs": {"enable_thinking": false}
  }'
```

PYTHON

```
from openai import OpenAI

client = OpenAI(
    base_url="https://<your-panel-domain>/api/tf/public/v1",
    api_key="hai_tf-<your-key>",      # better: read it from an environment variable
)

resp = client.chat.completions.create(
    model="qwen-qwen3-8-27b-fp8",
    messages=[{"role": "user", "content": "What is the capital of Nepal? Answer in one sentence."}],
    max_tokens=200,
    extra_body={"chat_template_kwargs": {"enable_thinking": False}},
)
print(resp.choices[0].message.content)
print(resp.usage.prompt_tokens, resp.usage.completion_tokens)
```

JAVASCRIPT / TYPESCRIPT

```
import OpenAI from "openai";

const client = new OpenAI({
  baseURL: "https://<your-panel-domain>/api/tf/public/v1",
  apiKey: process.env.TF_API_KEY,
});

const resp = await client.chat.completions.create({
  model: "qwen-qwen3-8-27b-fp8",
  messages: [{ role: "user", content: "What is the capital of Nepal? Answer in one sentence." }],
  max_tokens: 200,
  chat_template_kwargs: { enable_thinking: false },  // extra field, passed through (TypeScript: cast
the params)
});
console.log(resp.choices[0].message.content);
```

RESPONSE 200


```
{
  "id": "chatcmpl-c002eb483aec4842a6c8edf39bdf495d",
  "object": "chat.completion",
  "created": 1795159265,
  "model": "qwen-qwen3-8-27b-fp8",
  "choices": [
    {"index": 0, "finish_reason": "stop",
     "message": {"role": "assistant", "content": "The capital of Nepal is Kathmandu."}}
  ],
  "usage": {"prompt_tokens": 24, "completion_tokens": 10, "total_tokens": 34,
            "prompt_tokens_details": {"cached_tokens": 0}}
}
```

4. Authentication and API keys

Every inference request must carry a Token Factory API key in the standard bearer header:

```
Authorization: Bearer hai_tf-<32 hexadecimal characters>
```

A key belongs to a team. Every call made with it is attributed to, rate-limited for, and billed to that team. Keys are not bound to a model.

Create a key in the panel

1. Sign in to your panel and open the Token Factory Playground: **Token Factory** › **Playground** › **Open Playground**, or `https://<your-panel-domain>/pg/` in the same browser session.
2. Select **Keys** in the navigation, enter a **Key name** (for example `production-backend`) and choose **Create key**.
3. Copy the secret from **New secret, shown once** and store it in your secret manager. It cannot be displayed again; if it is lost, regenerate the key.

Create a key through the API

Keys can also be managed programmatically with the Team API (section 12), which authenticates with your panel credentials rather than a Token Factory key.

POST `/api/team/<team_id>/tf/api-keys` — PANEL CREDENTIALS

Body field	Type	Description
`name`	string	Display name for the key.
`label_model_id`	string	Optional display label only. It does not restrict which models the key can call.

CURL

```
curl -X POST "https://<your-panel-domain>/api/team/<team_id>/tf/api-keys" \
  -H "X-API-Key: $PANEL_API_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"name": "production-backend"}'
```

RESPONSE 201

```
{
  "id": "<key-id>",
  "team_id": "<team_id>",
  "name": "production-backend",
  "key_name": "production-backend",
  "secret": "hai_tf-<32 hex characters>",      // returned only on create and regenerate
  "key": "hai_tf-<32 hex characters>",         // same value, alternate field name
  "masked": "hai_tf-…a1b2",
  "base_url": "https://<your-panel-domain>/api/tf/public/v1",
  "status": "active",
  "created_by": "<user-id>",
  "created_at": "2026-11-20T09:41:01Z"
}
```

`last_used_at` and `regenerated_at` are added to the key object once the key has been used or regenerated.

> **USE THE RETURNED BASE_URL**
> Every key object carries `base_url`, the exact public base URL to configure in your client.

Rotate and revoke

Method	Path	Effect
GET	`/api/team/<team_id>/tf/api-keys`	List keys (masked preview only). Paged; see section 12.
GET	`/api/team/<team_id>/tf/api-keys/{key_id}`	One key, masked.
POST	`/api/team/<team_id>/tf/api-keys/{key_id}/regenerate`	Issues a new secret for the same key ID. The old secret stops working immediately; usage history stays attached to the same key ID.
DELETE	`/api/team/<team_id>/tf/api-keys/{key_id}`	Revokes one key.
DELETE	`/api/team/<team_id>/tf/api-keys`	Revokes every key in scope. Returns `{"revoked": <count>}`.

Revocation and regeneration take effect immediately for new requests. Past usage stays attributed to the key ID.

Security best practices

- Keep keys on your servers. Never embed them in browser or mobile code or commit them to source control.
- Use one key per application or environment so you can rotate or revoke one without affecting the others, and so usage can be broken down per key.
- Rotate keys periodically with **regenerate**, and immediately if you suspect exposure.
- Every authentication failure returns the same `401 invalid_api_key`, whether the key is unknown, malformed or revoked.

5. Models

GET `/models` — API KEY

Lists exactly the models your key's team may call, in the OpenAI list format, with Token Factory details under `hai`.

CURL

```
curl "$TF_BASE_URL/models" \
  -H "Authorization: Bearer $TF_API_KEY"
```

PYTHON

```
for m in client.models.list().data:
    print(m.id, m.hai["context_length"])
```

RESPONSE 200


```
{
  "object": "list",
  "data": [
    {
      "id": "qwen-qwen3-1-7b",
      "object": "model",
      "name": "Qwen3-1.7B",
      "created": 1792565320,
      "owned_by": "hosted-ai",
      "hai": {
        "type": "base",
        "status": "ready",
        "serving": true,
        "supports_tools": true,
        "supports_lora": false,
        "context_length": 40960,
        "gpu_type": "",
        "serving_gpu_type": "",
        "modalities": ["text"],
        "model_id": "<model-uuid>"
      }
    },
    {
      "id": "qwen-qwen3-8-27b-fp8",
      "object": "model",
      "name": "Qwen3.8-27B-FP8",
      "created": 1795094867,
      "owned_by": "hosted-ai",
      "hai": { "type": "base", "status": "ready", "serving": true, "supports_tools": true,
               "supports_lora": false, "context_length": 262144, "gpu_type": "",
               "serving_gpu_type": "", "modalities": ["text"], "model_id": "<model-uuid>" }
    }
  ]
}
```


GET `/models/{model}` — API KEY

Returns the same object for one model ID (OpenAI `models.retrieve`). An ID your team cannot call returns `404 model_not_found`.

```
curl "$TF_BASE_URL/models/qwen-qwen3-1-7b" -H "Authorization: Bearer $TF_API_KEY"
```

Model object fields

Field	Description
`id`	The value to send as `model` in a chat request.
`name`	Human-readable model name.
`hai.status`	`ready` or `unavailable`.
`hai.serving`	`true` when the model is live right now. A model that is not serving answers chat calls with `409 service_not_ready`.
`hai.context_length`	Maximum tokens (prompt plus completion) per request.
`hai.supports_tools`	Whether the model accepts `tools` (section 8).
`hai.modalities`	Accepted input types; `["text"]` for the current models.
`hai.type`	`base` for the catalogue models.
`hai.serving_gpu_type`	The GPU class the model is currently served on, when your provider records one; otherwise an empty string. `hai.gpu_type` carries the same value under an older name.
`hai.model_id`	The model's billing identifier, the same value that appears as `model_id` in pricing and usage reports.

6. Chat completions

`POST` `/chat/completions` API KEY

Generates a model response for a conversation. The request body is the standard OpenAI Chat Completions body.

Request body

Field	Type	Description
`model` *(required)*	string	A model `id` from `GET /models`.
`messages` *(required)*	array	The conversation. Roles: `system`, `user`, `assistant`, `tool`.
`max_tokens` / `max_completion_tokens`	integer	Upper bound on generated tokens. Either spelling is accepted. A per-model maximum may apply.
`stream`	boolean	`true` to receive Server-Sent Events. Must be a JSON boolean.
`stream_options`	object	Optional. `include_usage` is always enabled for you, so the final chunk carries usage.
`temperature`, `top_p`	number	Sampling controls.
`stop`	string / array	Sequences where generation stops.
`seed`	integer	Best-effort deterministic sampling.
`presence_penalty`, `frequency_penalty`	number	Repetition controls.
`tools`, `tool_choice`	array / string|object	Function calling (section 8). For `tool_choice`, see section 14.
`n`	integer	Number of choices. Omit it or send `1`.
`chat_template_kwargs`	object	Model template options, for example `{"enable_thinking": false}` (section 7).

Identity is always taken from your API key; fields such as `team_id` in the body are ignored.

Response body

Field	Description
`id`, `object`, `created`, `model`	Standard OpenAI fields. `object` is `chat.completion`.
`choices[].message.content`	The answer text (`null` when the model calls a tool).
`choices[].message.reasoning_content`	The model's reasoning, when thinking is enabled (section 7).
`choices[].message.tool_calls`	Tool calls requested by the model (section 8).
`choices[].finish_reason`	`stop`, `length` (hit `max_tokens`) or `tool_calls`.
`usage.prompt_tokens`	Input tokens, including cached tokens.

`usage.prompt_tokens_details.cached_tokens`	Input tokens served from the prompt cache, billed at the cached-input rate.
`usage.completion_tokens`	Generated tokens (reasoning tokens included), billed at the output rate.

Responses may include additional informational fields, for example `tf_metrics` with server-side timing and throughput. Clients should ignore fields they do not recognise.

> **HOW A CALL IS BILLED**
> Input is billed as `prompt_tokens - cached_tokens` at the input rate, `cached_tokens` at the cached-input rate, and `completion_tokens` at the output rate. Reusing a long, identical system prompt across calls increases `cached_tokens` and lowers cost.

Streaming

Set `"stream": true` to receive tokens as they are generated. The response is `text/event-stream`; each event is a `data:` line with a `chat.completion.chunk` object, and `data: [DONE]` marks the end of the response. The last chunk before `[DONE]` has an empty `choices` array and carries `usage`. One informational event may follow `[DONE]`: a chunk whose `delta` holds `hai_timing` (server-side timing, including `ttft_ms`). The OpenAI SDKs stop reading at `[DONE]`; if you parse the event stream yourself, do the same or ignore events after it.

CURL

```
curl -N "$TF_BASE_URL/chat/completions" \
  -H "Authorization: Bearer $TF_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{"model": "qwen-qwen3-8-27b-fp8", "stream": true, "max_tokens": 300,
       "messages": [{"role": "user", "content": "Write a haiku about GPUs."}],
       "chat_template_kwargs": {"enable_thinking": false}}'
```

PYTHON

```
stream = client.chat.completions.create(
    model="qwen-qwen3-8-27b-fp8",
    messages=[{"role": "user", "content": "Write a haiku about GPUs."}],
    max_tokens=300,
    stream=True,
    extra_body={"chat_template_kwargs": {"enable_thinking": False}},
)
for chunk in stream:
    if chunk.choices and chunk.choices[0].delta.content:
        print(chunk.choices[0].delta.content, end="", flush=True)
    if chunk.usage:                                  # final chunk
        print("\ntokens:", chunk.usage.total_tokens)
```

JAVASCRIPT / TYPESCRIPT

```
const stream = await client.chat.completions.create({
  model: "qwen-qwen3-8-27b-fp8",
  messages: [{ role: "user", content: "Write a haiku about GPUs." }],
  stream: true,
  chat_template_kwargs: { enable_thinking: false },
});
for await (const chunk of stream) {
  process.stdout.write(chunk.choices[0]?.delta?.content ?? "");
  if (chunk.usage) console.log("\ntokens:", chunk.usage.total_tokens);
}
```

STREAM (ABRIDGED)

```
data: {"id":"chatcmpl-d5f7...","object":"chat.completion.chunk","model":"qwen-qwen3-8-27b-fp8",
       "choices":[{"index":0,"delta":{"role":"assistant"},"finish_reason":null}]}
data: {"id":"chatcmpl-d5f7...","object":"chat.completion.chunk",
       "choices":[{"index":0,"delta":{"content":"Silicon"},"finish_reason":null}]}
...
data: {"id":"chatcmpl-d5f7...","object":"chat.completion.chunk","choices":[],
       "usage":{"prompt_tokens":58,"completion_tokens":60,"total_tokens":118,
                "prompt_tokens_details":{"cached_tokens":0}}}
data: [DONE]
data: {"object":"chat.completion.chunk",
       "choices":[{"index":0,"delta":{"hai_timing":{"total_ms":66,"ttft_ms":33,...}}}]}
```

7. Reasoning (thinking) control

Both Qwen3 models can **think before answering**. Thinking is **on by default**. The reasoning is returned separately from the answer, so your application can show, log or discard it:

- Non-streamed: `choices[0].message.reasoning_content` holds the reasoning and `choices[0].message.content` the final answer.
- Streamed: reasoning arrives in `delta.reasoning_content` chunks, followed by the answer in `delta.content` chunks.

Reasoning tokens are generated tokens: they count toward `completion_tokens` and `max_tokens`. Leave enough `max_tokens` headroom when thinking is on, or the answer may be cut off with `finish_reason: "length"`. With thinking on, `content` can start with blank lines; trim it before display.

Switch thinking off

For fast, low-cost answers (extraction, classification, chat UIs) send `chat_template_kwargs` with `enable_thinking: false`:

CURL

```
curl "$TF_BASE_URL/chat/completions" \
  -H "Authorization: Bearer $TF_API_KEY" -H "Content-Type: application/json" \
  -d '{"model": "qwen-qwen3-1-7b",
       "messages": [{"role": "user", "content": "What is 17 * 23? Reply with just the number."}],
       "max_tokens": 100,
       "chat_template_kwargs": {"enable_thinking": false}}'
```

PYTHON

```
resp = client.chat.completions.create(
    model="qwen-qwen3-1-7b",
    messages=[{"role": "user", "content": "What is 17 * 23? Reply with just the number."}],
    max_tokens=100,
    extra_body={"chat_template_kwargs": {"enable_thinking": False}},   # omit to keep thinking on
)
```

THINKING ON (DEFAULT)

```
"message": {
  "role": "assistant",
  "reasoning_content": "Okay, 17 times 20 is
    340, 17 times 3 is 51, so 340 + 51
    = 391 ...",
  "content": "391"
}
"usage": {"prompt_tokens": 24,
          "completion_tokens": 572}
```

THINKING OFF

```
"message": {
  "role": "assistant",
  "content": "391"
}


"usage": {"prompt_tokens": 28,
          "completion_tokens": 4}
```

Reading reasoning in code

PYTHON

```python
resp = client.chat.completions.create(model="qwen-qwen3-8-27b-fp8", messages=msgs, max_tokens=2048)
msg = resp.choices[0].message
reasoning = getattr(msg, "reasoning_content", None)   # extra field, present when thinking is on
answer = msg.content

# Streaming
for chunk in client.chat.completions.create(model="qwen-qwen3-8-27b-fp8", messages=msgs,
stream=True):
    if not chunk.choices:
        continue
    d = chunk.choices[0].delta
    if getattr(d, "reasoning_content", None):
        print(d.reasoning_content, end="")      # thinking
    elif d.content:
        print(d.content, end="")                # answer
```

MULTI-TURN CONVERSATIONS

When you send the conversation back for the next turn, include only the assistant's `content`; do not include `reasoning_content` in the `messages` history.

8. Tool calling

Models with `hai.supports_tools: true` accept OpenAI-format `tools`. The model decides when to call a function and returns structured arguments; your code runs the function and sends the result back in a `tool` message. Both current models support tool calling.

Step 1: offer tools

PYTHON

```python
import json

tools = [{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "Get the current weather for a city",
        "parameters": {
            "type": "object",
            "properties": {"city": {"type": "string"}},
            "required": ["city"],
        },
    },
}]
messages = [{"role": "user", "content": "What's the weather in Kathmandu right now?"}]

resp = client.chat.completions.create(
    model="qwen-qwen3-8-27b-fp8", messages=messages, tools=tools, tool_choice="auto",
    extra_body={"chat_template_kwargs": {"enable_thinking": False}},
)
msg = resp.choices[0].message
```

RESPONSE 200 (FINISH_REASON: TOOL_CALLS)

```
"choices": [{
  "index": 0,
  "finish_reason": "tool_calls",
  "message": {
    "role": "assistant",
    "content": null,
    "tool_calls": [{
      "id": "chatcmpl-tool-acaaae0d4cb8bdf0",
      "type": "function",
      "function": {"name": "get_weather", "arguments": "{\"city\": \"Kathmandu\"}"}
    }]
  }
}]
```

Step 2: run the tool and return the result

PYTHON

```python
messages.append(msg)                                  # the assistant turn with tool_calls
for call in msg.tool_calls:
    args = json.loads(call.function.arguments)
    result = {"temp_c": 18, "condition": "clear"}     # your real implementation here
    messages.append({"role": "tool", "tool_call_id": call.id, "content": json.dumps(result)})

final = client.chat.completions.create(
    model="qwen-qwen3-8-27b-fp8", messages=messages, tools=tools,
    extra_body={"chat_template_kwargs": {"enable_thinking": False}},
)
print(final.choices[0].message.content)
# "Kathmandu is currently clear with a temperature of 18°C (64°F). ..."
```

The same with curl

```
curl "$TF_BASE_URL/chat/completions" \
  -H "Authorization: Bearer $TF_API_KEY" -H "Content-Type: application/json" \
  -d '{
    "model": "qwen-qwen3-1-7b",
    "messages": [{"role": "user", "content": "What is the weather in Kathmandu right now?"}],
    "tools": [{"type": "function", "function": {
      "name": "get_weather", "description": "Get the current weather for a city",
      "parameters": {"type": "object", "properties": {"city": {"type": "string"}}, "required":
["city"]}}}],
    "tool_choice": "auto",
    "chat_template_kwargs": {"enable_thinking": false}
  }'
```

TIPS

Write clear function descriptions and parameter schemas; they are the model's only guide. Always validate `arguments` (it is a JSON string) before acting on it. Tool definitions count as prompt tokens on every call.

9. Safe retries and response headers

Idempotency-Key

Networks fail. To retry a chat request without risk of being billed twice, send an `Idempotency-Key` header with a unique value per logical request (for example a UUID you generate):

- The same key with the **same body** within 24 hours is billed **once**, however many times it is retried.
- The same key with a **different body** is refused with `422 idempotency_key_reused`. Use a new key for new work.
- A retry sent while the first attempt is still running gets `409 idempotency_key_in_use`. Wait briefly and retry with the same key.

Without the header every call is billed, because two identical prompts sent on purpose are two units of work.

CURL

```
curl "$TF_BASE_URL/chat/completions" \
  -H "Authorization: Bearer $TF_API_KEY" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: order-8812-summary" \
  -d @request.json
```

PYTHON

```
import uuid
resp = client.chat.completions.create(
    model="qwen-qwen3-1-7b", messages=msgs,
    extra_headers={"Idempotency-Key":
                   str(uuid.uuid4())},
)
```

Response headers

Header	Description
`X-Request-Id`	Unique ID of the request (for example `req-…`). Include it when you contact support. If you send your own `X-Request-Id`, see section 14.
`X-Elapsed-Ms`	Server-side time spent on the request in milliseconds, on chat completions. On a streamed call it is the time until the stream started.
`X-RateLimit-*`	Your current rate-limit state, when your team has rate limits configured (section 10).
`Retry-After`	Seconds to wait, on a `429` rate-limit or quota refusal.

10. Rate limits and quotas

Limits are set per team by your provider and apply to all of the team's keys together. They can include any of:

Limit	Applies to
Requests per minute	Chat completion calls.
Tokens per minute	Input plus output tokens of chat completion calls.
Concurrent requests	Calls in flight at the same time (team-wide, and optionally per user).
Token quota	Total chat tokens per quota window (for example per month), and optional hourly or daily per-user budgets.

Your team's limits in force, and the tokens remaining in the current quota window, are returned by `GET /api/team/<team_id>/tf/limits` (section 12). A limit that is not set is omitted from that response, meaning no limit applies.

Rate-limit headers

When your team has a requests-per-minute or tokens-per-minute limit, chat completion responses (including refusals) carry:

```
X-RateLimit-Limit-Requests: 600        X-RateLimit-Limit-Tokens: 400000
X-RateLimit-Remaining-Requests: 597    X-RateLimit-Remaining-Tokens: 399118
X-RateLimit-Reset-Requests: 60s        X-RateLimit-Reset-Tokens: 60s
```

Only the pair for a dimension that is actually limited is sent. Windows are one minute.

When a limit is reached

HTTP	Code	Meaning and what to do
429	`rate_limited`	A per-minute or concurrency limit was reached. Wait for `Retry-After` seconds, then retry.
429	`hourly_quota_exhausted`	An hourly token budget is used up. It refills within the hour; honour `Retry-After`.
402	`insufficient_balance`	The team's token quota for the window is exhausted, or the team's access is suspended. Retrying does not help; contact your provider or wait for the next quota window.

CLIENT BEHAVIOUR

The official OpenAI SDKs retry `429` and `5xx` responses automatically (2 retries by default, configurable with `max_retries`). For high-throughput workloads, cap your own concurrency to your team's limit and back off on `429`.

11. Errors

Inference errors use the OpenAI error envelope, so OpenAI SDKs raise their usual exception types:

ERROR BODY

```
{
  "error": {
    "code": "model_not_found",
    "message": "The model does not exist.",
    "type": "not_found_error",
    "request_id": "req-...",
    "param": "..."            // present when one request field is at fault
  }
}
```

Branch on `error.code`, which is stable. `message` is human-readable and may change.

HTTP	Code	Meaning	Retry?
400	`invalid_request`	The body is not valid (not a JSON object, `stream` not a boolean, or rejected by the model). `message` explains why and is safe to show to a user.	No, fix the request
400	`context_length_exceeded`	The prompt (messages plus tools) is longer than the model's context window. `param` is `messages`. For `max_tokens`, see section 14.	No, shorten input
401	`invalid_api_key`	Missing, malformed, unknown or revoked key.	No
402	`insufficient_balance`	Token quota exhausted, or team access suspended.	No
403	`model_not_entitled`	Your team is not permitted to use this model or Token Factory.	No
404	`model_not_found`	No such model for your key. Check `GET /models`.	No
409	`service_not_ready`	The model exists but is not serving at the moment.	Yes, later
409	`model_not_priced`	The model is not yet enabled for billing on your team. Contact your provider.	No
409	`idempotency_key_in_use`	A request with this `Idempotency-Key` is still running.	Yes, same key
413	`payload_too_large`	Request body over 8 MB.	No
422	`idempotency_key_reused`	This `Idempotency-Key` was used with a different body.	No, new key
429	`rate_limited`	Rate or concurrency limit reached.	Yes, after `Retry-After`
429	`hourly_quota_exhausted`	Hourly token budget used up.	Yes, after `Retry-After`
502 / 503 / 504	`upstream_unavailable`	The model is starting up or temporarily unavailable.	Yes, with backoff
500	`api_error`	Unexpected server error.	Yes, with backoff

Handling errors in Python

```python
import time, openai

def chat(**kwargs):
    for attempt in range(5):
        try:
            return client.chat.completions.create(**kwargs)
        except openai.RateLimitError as e:                  # 429
            time.sleep(int(e.response.headers.get("Retry-After", "5")))
        except openai.APIStatusError as e:
            code = e.code                                   # error.code from the body
            if e.status_code in (500, 502, 503, 504) or code == "service_not_ready":
                time.sleep(2 ** attempt)                    # transient: back off and retry
                continue
            raise                                           # 400/401/402/403/404: do not retry
    raise RuntimeError("request did not succeed after retries")
```

Handling errors in JavaScript

```javascript
try {
  const resp = await client.chat.completions.create({ model, messages });
} catch (err) {
  if (err instanceof OpenAI.APIError) {
    console.error(err.status, err.code, err.message, err.headers?.["x-request-id"]);
  }
  throw err;
}
```

12. Team API: keys, usage, billing and metrics

The Team API gives programmatic access to your team's Token Factory account: API keys, usage reports, the per-request billing ledger, prices, limits and latency metrics. It is part of the panel API, not the inference API.

Base URL	`https://<your-panel-domain>/api/team/<team_id>/tf`
Authentication	`X-API-Key: <panel API token>` (a panel API token for a user who manages the team, created in the panel's profile security settings), or `Authorization: Bearer <panel session token>`. A Token Factory `hai_tf-` key is **not** accepted here.
Errors	`{"code": <number>, "message": "...", "errors": [...], "trace_id": "..."}`; `403` when the caller does not manage the team.

```
# Python setup for the examples in this section
import requests
TEAM_URL = "https://<your-panel-domain>/api/team/<team_id>/tf"
H = {"X-API-Key": PANEL_API_TOKEN}
```

Usage summary

GET `/api/team/<team_id>/tf/usage` — PANEL CREDENTIALS

Aggregated usage over a window. Query: `from`, `to` (RFC 3339; default the last month), `group_by` (`key`, `model`, `day` or `hour`), `api_key_id`, `model_id`.

```
summary = requests.get(f"{TEAM_URL}/usage", headers=H, params={"group_by": "model"}).json()
```

RESPONSE 200


```
{
  "currency": "USD",
  "currency_precision": 2,
  "range": {"from": "2026-10-20T00:00:00Z", "to": "2026-11-20T10:00:00Z"},
  "rows": [
    {"group": {"model_id": "<model-uuid>"},
     "input_uncached_tokens": 412933, "cached_input_tokens": 1024, "output_tokens": 88214,
     "training_tokens": 0, "embedding_tokens": 0, "storage_gb_hours": 0,
     "cost_total": 2, "cost_total_exact": "2.02110"}
  ],
  "totals": {"group": {}, "input_uncached_tokens": 412933, "cached_input_tokens": 1024,
             "output_tokens": 88214, "training_tokens": 0, "embedding_tokens": 0,
             "storage_gb_hours": 0, "cost_total": 2, "cost_total_exact": "2.02110"}
}
```


BILL FROM COST_TOTAL_EXACT

Costs are in minor currency units (for USD, cents). `cost_total` is rounded to a whole minor unit for display and is often `0` for a single small call. `cost_total_exact` is the exact, unrounded value as a decimal string: parse it with a decimal type, never a float.

Billing ledger

GET `/api/team/<team_id>/tf/usage/ledger` — PANEL CREDENTIALS

Append-only, one row per billed unit, for exact reconciliation. Query: `cursor` (exclusive `ledger_id` watermark), `limit` (default 100, max 250), `from`, `to`, `api_key_id`, `model_id`, `include_errored` (add zero-cost failed-call rows, marked `"errored": true` with an `error_code`; they never count toward totals). Prices are frozen onto each row when it is written. `ledger_id` is unique per row; use `request_id` to join a row to your own logs.

```python
from decimal import Decimal
watermark = load_watermark()                    # 0 on first run
while True:
    page = requests.get(f"{TEAM_URL}/usage/ledger", headers=H,
                        params={"cursor": watermark, "limit": 250}).json()
    for row in page["data"]:
        record(row["ledger_id"], Decimal(row["cost_total_exact"]))   # de-duplicate on ledger_id
        watermark = max(watermark, row["ledger_id"])
    save_watermark(watermark)
    if not page["has_more"]:                     # loop on has_more, not on next_cursor
        break
```

LEDGER ROW


```
{
  "ledger_id": 100482,
  "team_id": "<team_id>",
  "api_key_id": "<key-id>",
  "model_id": "<model-uuid>",
  "adapter_id": null,
  "kind": "inference",
  "request_id": "req-...",
  "occurred_at": "2026-11-20T09:40:02.749Z",
  "quantities": {"input_uncached_tokens": 24, "cached_input_tokens": 0, "output_tokens": 10,
                 "training_tokens": 0, "embedding_tokens": 0, "storage_gb_hours": 0},
  "unit_prices": {"input_per_1m": 10, "cached_input_per_1m": 5, "output_per_1m": 40,
                  "input_per_1m_exact": "10", "cached_input_per_1m_exact": "5",
                  "output_per_1m_exact": "40"},
  "tariff_version": "2026-11-01",
  "currency": "USD",
  "cost_total": 0,
  "cost_total_exact": "0.00064"
}
```


Prices

GET `/api/team/<team_id>/tf/pricing` — PANEL CREDENTIALS

Current per-model prices for your team, in **minor currency units per 1,000,000 tokens** (for USD, `40` means 40 cents per million tokens), with the date they apply from. Each rate also has an unrounded `*_exact` string. The response also carries the fine-tuning rate per model (`fine_tuning_per_1m`) and team-wide `storage_per_gb_hour` and `embedding_per_1m` rates. Add `?reachable=true` to list only the models in your catalogue, each with `model_name`.

```
{
  "currency": "USD",
  "implement_from": "2026-11-20",
  "models": [
    {"model_id": "<model-uuid>", "model_name": "Qwen3-1.7B",
     "input_per_1m": 10, "cached_input_per_1m": 5, "output_per_1m": 40, "fine_tuning_per_1m": 50,
     "input_per_1m_exact": "10", "cached_input_per_1m_exact": "5", "output_per_1m_exact": "40",
     "fine_tuning_per_1m_exact": "50"}
  ],
  "storage_per_gb_hour": 1, "storage_per_gb_hour_exact": "1",
  "embedding_per_1m": 5, "embedding_per_1m_exact": "5"
}
```


Limits

`GET` `/api/team/<team_id>/tf/limits` **PANEL CREDENTIALS**

The caps in force for your team. `access_type` is `unlimited`, `restricted` or `no_access` (the team cannot call Token Factory). A cap that is not set is omitted. Per-user caps, when set, appear as `max_concurrent_per_user`, `max_tokens_per_user_hour` and `max_tokens_per_user_day`.


```
{
  "team_id": "<team_id>",
  "access_type": "restricted",
  "max_requests_per_minute": 600,
  "max_tokens_per_minute": 400000,
  "max_concurrent": 16,
  "max_total_tokens": 50000000,
  "quota_window": "month",
  "tokens_remaining": 48211004
}
```


Latency and cost metrics

`GET` `/api/team/<team_id>/tf/metrics` **PANEL CREDENTIALS**

p50/p90/p95/p99 time-to-first-token (`ttft_ms`), time-per-output-token (`tpot_ms`) and end-to-end latency (`e2e_ms`), bucketed by `hour` (default) or `day`, each bucket with the cost of the same requests. Query: `from`, `to` (default the last 7 days), `bucket`, `api_key_id`, `model_id`. `measured` is the number of requests that carried a timing.

```
m = requests.get(f"{TEAM_URL}/metrics", headers=H, params={"bucket": "day"}).json()
for b in m["buckets"]:
    print(b["bucket"], b["requests"], b["ttft_ms"]["p95"], b["e2e_ms"]["p95"], b["cost_total_exact"])
```

Team model catalogue

`GET` `/api/team/<team_id>/tf/models` **PANEL CREDENTIALS**

The same catalogue as `GET /models` on the inference API, read with panel credentials instead of a Token Factory key.

Paging

List endpoints (API keys, ledger) return `{"data": [...], "next_cursor": "...", "has_more": true|false}`. Pass `limit` (API keys: default 50; ledger: default 100; at most 250 rows per page) and `cursor` to page. Keep requesting while `has_more` is `true`; `next_cursor` is empty on the last page. For API keys, the cursor is opaque: a cursor that names no row returns `400 invalid_cursor`; restart without one. For the ledger, the cursor is a numeric `ledger_id` watermark: a value past the newest row returns an empty page, and a non-numeric value returns `400`.

13. Endpoint reference

Inference API

Base: `https://<your-panel-domain>/api/tf/public/v1` · Auth: `Authorization: Bearer hai_tf-…`

Method	Path	Description
GET	`/models`	List the models your key may call.
GET	`/models/{model}`	Retrieve one model.
POST	`/chat/completions`	Chat completion: buffered or streamed, reasoning control, tool calling, `Idempotency-Key`.

Team API

Base: `https://<your-panel-domain>/api/team/<team_id>/tf` · Auth: `X-API-Key` panel API token or panel session bearer token

Method	Path	Description
POST	`/api-keys`	Create an API key (secret returned once).
GET	`/api-keys`	List API keys (masked).
GET	`/api-keys/{key_id}`	Get one API key (masked).
POST	`/api-keys/{key_id}/regenerate`	New secret, same key ID.
DELETE	`/api-keys/{key_id}`	Revoke one key.
DELETE	`/api-keys`	Revoke all keys in scope.
GET	`/usage`	Aggregated usage and cost.
GET	`/usage/ledger`	Per-request billing ledger.
GET	`/pricing`	Current prices per model.
GET	`/limits`	Rate limits, concurrency and quota in force.
GET	`/metrics`	Latency percentiles with cost.
GET	`/models`	Team model catalogue.

Checklist for production

- Read the API key from an environment variable or secret manager; one key per service and environment.
- Set `enable_thinking: false` for latency-sensitive paths; leave thinking on (with a generous `max_tokens`) for hard reasoning.
- Use streaming for interactive UIs.
- Send an `Idempotency-Key` on requests you may retry.
- Honour `Retry-After` on `429`; back off on `5xx` and `409 service_not_ready`.
- Log `X-Request-Id` with every call so support can trace it.
- Reconcile billing from the ledger's `cost_total_exact` using decimal arithmetic.

14. Known issues

These behaviours differ from the rest of this guide today. Each one has a simple workaround, and this section is updated as they are resolved.

Area	Current behaviour	What to do
`tool_choice`	Treated as `"auto"` : the model decides whether to call a tool. The values `"none"` , `"required"` and a named function are accepted but not yet enforced.	To prevent tool calls, leave `tools` out of the request. To steer the model to a function, name it in the instructions, and check that `finish_reason` is `"tool_calls"` before reading `tool_calls` .
`X-Request-Id` sent by the client	Your value is used as the request's ID: it is returned in the `X-Request-Id` response header and stored as `request_id` on the ledger row.	Make the value unique per call (for example a UUID), or omit it and let the server assign one. When reconciling the ledger, de-duplicate on `ledger_id` , not on `request_id` .
`max_tokens` above the context window	A short prompt with a `max_tokens` larger than the model's context window is accepted (`200`). Only a prompt that is itself longer than the context window returns `400` `context_length_exceeded` .	Keep prompt tokens plus `max_tokens` within the model's `hai.context_length` from `GET /models` .
`logprobs` and `n`	`logprobs` and `top_logprobs` are accepted, but `choices[0].logprobs` is not returned. `n` must be `1` ; other values return `400` .	Do not depend on `logprobs` . To get several completions, send separate requests.