Library User Guide — `llm_service/client.py`

0) Overview

This library provides a **Python client SDK** for interacting with **Groq's OpenAI-compatible LLM APIs**, including:

- Chat completions (like OpenAI Chat API)
- Responses API (flat prompt → completion)
- Streaming responses (SSE)
- Batch API for bulk jobs

It supports both **synchronous** (`GroqLLM`) and **asynchronous** (`AsyncGroqLLM`) usage with consistent interfaces.

---

1) Installation & requirements

1.1 Python version

- Supports Python **3.9–3.12**
- Tested in Linux & macOS; works in containers

1.2 Dependencies

- `httpx` (auto-installed if packaged correctly)
- No hard dependencies beyond stdlib

1.3 Install

```
pip install llm-service   # if published to PyPI
# or
pip install git+https://github.com/<your-org>/llm_service.git
```

---

2) Configuration

2.1 Environment variables

Set the following in your environment:

```
export GROQ_API_KEY=sk-xxxx...
# Optional override (default already points to Groq)
export GROQ_API_BASE=https://api.groq.com/openai
```

2.2 Idempotency keys

- Non-stream calls: auto-assigned for you (safe retries)
- Stream calls: optional; provide if you want deduplication

---

3) Quickstart

3.1 Chat completion (sync)

```
from llm_service.client import GroqLLM, llm_input_for_chat, Message

with GroqLLM() as cli:
    inp = llm_input_for_chat(
        model="llama-3.3-70b-versatile",
        messages=[Message(role="user", content="Write a haiku about
code")]
    )
    res = cli.chat(inp)
    print(res.text)
```

3.2 Streaming chat (sync)

```
with GroqLLM() as cli:
    inp = llm_input_for_chat(
        model="llama-3.3-70b-versatile",
        messages=[Message("user", "Tell me a story, stream it")]
    )
    for chunk in cli.chat_stream(inp):
        if chunk.delta:
            print(chunk.delta, end="", flush=True)
```

3.3 Responses API (async)

```
import asyncio
from llm_service.client import AsyncGroqLLM, llm_input_for_prompt

async def main():
    async with AsyncGroqLLM() as cli:
        inp = llm_input_for_prompt(
            model="llama-3.1-8b-instant",
            prompt="Explain recursion in simple terms"
        )
        res = await cli.response(inp)
        print(res.text)

asyncio.run(main())
```

3.4 Responses API streaming (async)

```
async with AsyncGroqLLM() as cli:
    inp = llm_input_for_prompt(
        model="llama-3.1-8b-instant",
        prompt="List 5 startup ideas",
        stream=True
    )
    async for chunk in cli.response_stream(inp):
        if chunk.delta:
            print(chunk.delta, end="", flush=True)
```

---

4) Features & capabilities

4.1 Supported endpoints

- `chat` / `chat_stream` → Chat completion API
- `response` / `response_stream` → Prompt-completion API
- `batch_create`, `batch_retrieve`, `batch_list`, `batch_cancel` → Bulk jobs

4.2 Message model

```
Message(role="user", content="Hello")
Message(role="system", content="You are a helpful assistant")
Message(role="assistant", content="Hello back!")
Message(role="tool", content="...")  # optional tool integration
```

4.3 Options

- `temperature`, `top_p` — sampling
- `max_completion_tokens` / `max_output_tokens` — capped to model limits automatically
- `stop` — list of stop sequences
- `tools` / `tool_choice` — for tool-calling
- `response_format` — for JSON/structured output
- `service_tier` — control vendor compute tier if available

4.4 Return type

Every API returns an `LLMResult`:

```
LLMResult(
```

```
    kind="message" | "stream_chunk" | "batch",
    text="final text" (non-stream),
    delta="streamed chunk text",
    tool_calls=[...] (if present),
    finish_reason="stop|length|tool_calls|...",
    id="response_id",
    raw=<full vendor response>
)
```

---

5) Error handling

5.1 Exceptions

- `GroqSDKError` → base error (timeouts, generic)
- `GroqHTTPError` → includes:
    - `status` (HTTP status code)
    - `code` (vendor error code)
    - `request_id` (for vendor support)
    - `body` (raw body string)

5.2 Example

```
try:
    with GroqLLM() as cli:
        ...
except GroqHTTPError as e:
    print(f"HTTP {e.status}, code={e.code}, req={e.request_id}")
except GroqSDKError as e:
    print("Generic SDK error", e)
```

---

6) Best practices

1. **Reuse client objects**:
    - Expensive to construct per request.
    - Use context managers (`with` / `async with`) in short-lived jobs.
    - Long-running apps: create once and reuse.
2. **Streaming**:
    - Callers must consume generator fully or close it early.
    - Use `.delta` for incremental text.
3. **Token caps**:
    - Library enforces vendor caps automatically.
    - If you request more tokens, you'll silently get the max allowed.
4. **Idempotency**:
    - Non-stream requests get an auto `Idempotency-Key`.
    - For retries, you can provide your own.
5. **Timeouts**:
    - Defaults tuned for LLMs.
    - Override if you expect longer completions.
6. **Tracing/metrics**:
    - Use `on_event` hook if you want to send request metrics to your observability system.

---

7) Batch API usage

7.1 Create batch

```
with GroqLLM() as cli:
    batch_payload = {
        "requests": [
            {"custom_id": "1", "method": "POST", "url":
"/v1/responses",
             "body": {"model": "llama-3.1-8b-instant", "input":
"First"}},
            {"custom_id": "2", "method": "POST", "url":
"/v1/responses",
             "body": {"model": "llama-3.1-8b-instant", "input":
"Second"}}
        ]
    }
    res = cli.batch_create(LLMInput(batch=batch_payload))
    print("Batch ID:", res.id)
```

7.2 Retrieve status

```
res = cli.batch_retrieve(batch_id)
print(res.raw)   # contains vendor batch object
```

---

8) Common pitfalls

- **Forgetting to set `GROQ_API_KEY`** → raises `GroqSDKError("GROQ_API_KEY is not set.")`
- **Mixing sync & async** → use the right client for your code.
- **Not consuming stream** → server keeps connection open; always break or exhaust generator.
- **Large stop sequences** → vendor may reject; keep ≤ 4.
- **Assuming retries hide errors** → retries are limited (default 3); always handle `GroqHTTPError`.

---

9) Example: complete workflow

```
from llm_service.client import GroqLLM, llm_input_for_chat, Message

with GroqLLM() as cli:
    # Step 1: Send chat
    inp = llm_input_for_chat(
        "llama-3.3-70b-versatile",
        [Message("system", "You are a summarizer"),
         Message("user", "Summarize this: ...")]
    )
    res = cli.chat(inp)
    print("Summary:", res.text)

    # Step 2: Stream a continuation
    for chunk in cli.chat_stream(
        llm_input_for_chat("llama-3.3-70b-versatile", [Message("user",
"Continue...")])
    ):
        if chunk.delta:
            print(chunk.delta, end="", flush=True)

    # Step 3: Batch jobs
    batch_input = {"requests": [...]}  # see section 7
    batch_res = cli.batch_create(LLMInput(batch=batch_input))
    print("Batch created:", batch_res.id)
```

---

10) Troubleshooting

- **Timeouts**: Increase `read` timeout if generating long outputs.
- **429 Too Many Requests**: Client retries automatically; if persistent, reduce concurrency.
- **Streaming cuts early**: Ensure HTTP/2 enabled and no proxy buffering SSE.

- **Transport errors**: Typically DNS/proxy/firewall issues; check connectivity.

---

11) Library limitations

- No built-in circuit breaker (caller should implement if required).
- Tool calls in Responses API are not synthesized; forwarded as-is only.
- Library does not log anything by default (safe to use in sensitive environments).

✅ With this guide, an application developer can safely **install, configure, and use** the library for synchronous/asynchronous calls, streaming, and batch jobs, with awareness of features, error handling, and best practices.