LLM API Integration Reference 2026

Reference Card · Updated April 2026

Provider Comparison

Use these numbers for rough budgeting. Actual rate limits depend on your tier and recent spend. Pricing per 1M tokens (input/output).[1][2]

PROVIDER	AUTH METHOD	BASE URL	FLAGSHIP MODELS	PRICING (INPUT/OUTPUT PER 1M)	NOTES ON RATE LIMITS
Anthropic	`x-api-key` header + `anthropic-version: 2023-06-01`	https://api.anthropic.com	claude-opus-4-6, claude-sonnet-4-6, claude-haiku-4-5	Opus ~$5/$25, Sonnet ~$3/$15, Haiku ~$1/$5	TPM/RPM by tier. Usage returned in every response.
OpenAI	`Authorization: Bearer $OPENAI_API_KEY`	https://api.openai.com/v1	gpt-5.4, gpt-5.4-mini, gpt-5.4-nano	gpt-5.4 ~$2.50/$15, mini ~$0.75/$4.50	RPM + TPM. Scales with spend tier.
xAI / Grok	Bearer token (OpenAI compatible)	https://api.x.ai/v1	grok-4.20, grok-4.1-fast	grok-4.20 ~$2/$6, fast ~$0.20/$0.50	Tiered by cumulative spend. TPM/RPM per model.

Setup Checklist

Anthropic

- `pip install anthropic==0.88.*` or `npm install @anthropic-ai/sdk`
- Set `ANTHROPIC_API_KEY`
- `client = anthropic.Anthropic()` (reads env var)
- First call: include `anthropic-version` header via SDK (automatic)

OpenAI

- `pip install openai` (1.x) or `npm install openai`
- Set `OPENAI_API_KEY`
- `client = OpenAI()`
- First call: `model="gpt-5.4-mini"`

xAI

- Use OpenAI SDK
- Set base URL to `https://api.x.ai/v1`
- Use same key pattern as OpenAI. Test with `grok-4.1-fast` first.

Basic Completion

Python

```
import anthropic
client = anthropic.Anthropic()

response = client.messages.create(
 model="claude-sonnet-4-6",
 max_tokens=1024,
 messages=[{"role": "user", "content": "Explain PID control loops."}]
)
print(response.content[0].text)
```

TypeScript

```
import OpenAI from "openai";
const client = new OpenAI({ baseURL: "https://api.x.ai/v1" }); // or OpenAI base

const response = await client.chat.completions.create({
 model: "gpt-5.4-mini",
 messages: [{ role: "user", content: "Explain PID control loops." }]
});
console.log(response.choices[0].message.content);
```

Streaming

Python (Anthropic)

```
with client.messages.stream(
 model="claude-sonnet-4-6",
 max_tokens=1024,
 messages=[{"role": "user", "content": "Tell a short story."}]
) as stream:
 for chunk in stream:
 if chunk.type == "content_block_delta":
 print(chunk.delta.text or "", end="", flush=True)
```

TypeScript (OpenAI compatible)

```
const stream = await client.chat.completions.create({
 model: "grok-4.1-fast",
 messages: [{ role: "user", content: "Tell a short story." }],
 stream: true
});
```

```
for await (const chunk of stream) {
 const delta = chunk.choices[0]?.delta?.content || "";
 process.stdout.write(delta);
}
```

Tool Calling

Python (Anthropic style)

```
tools = [{
 "name": "get_weather",
 "description": "Get current weather",
 "input_schema": {
 "type": "object",
 "properties": {"location": {"type": "string"}},
 "required": ["location"]
 }
}]

response = client.messages.create(
 model="claude-opus-4-6",
 max_tokens=1024,
 tools=tools,
 messages=[{"role": "user", "content": "Weather in Tokyo?"}]
)

if response.content[0].type == "tool_use":
 # execute tool, then continue conversation with tool_result
 pass
```

TypeScript (OpenAI style)

```
const tools = [{
 type: "function",
 function: {
 name: "get_weather",
 description: "Get current weather",
 parameters: {
 type: "object",
 properties: { location: { type: "string" } },
 required: ["location"]
 }
 }
}];

const response = await client.chat.completions.create({
 model: "gpt-5.4",
 messages: [{ role: "user", content: "Weather in Tokyo?" }],
 tools: tools
});
```

Structured Output (JSON)

Python (Anthropic - strict tool)

```
response = client.messages.create(
 model="claude-sonnet-4-6",
 max_tokens=1024,
 tools=[{
 "name": "extract_data",
 "input_schema": {
 "type": "object",
 "properties": {
 "name": {"type": "string"},
 "value": {"type": "number"}
 },
 "required": ["name", "value"]
 },
 "strict": True
 }],
 tool_choice={"type": "tool", "name": "extract_data"},
 messages=[{"role": "user", "content": "Data: temperature 72."}]
)
```

TypeScript (OpenAI - response_format)

```
const response = await client.chat.completions.create({
 model: "gpt-5.4-mini",
 messages: [{ role: "user", content: "Parse this: temperature 72." }],
 response_format: {
 type: "json_object"
 }
});
```

Error Handling with Retry and Backoff

```
import time
import random
from anthropic import RateLimitError, APIError

def call_with_retry(client, max_retries=5):
 for attempt in range(max_retries):
 try:
 return client.messages.create( ... ) # your call
 except RateLimitError:
 wait = (2 ** attempt) + random.uniform(0, 1)
 time.sleep(wait)
 continue
 except APIError as e:
 if e.status_code in (408, 500, 502, 503, 504): # timeout or server error
 time.sleep(1)
 continue
 raise
 raise Exception("Max retries exceeded")
```

Add timeout via `httpx` client configuration in SDK init for both providers.

Cost Tracking Middleware Pattern

```
class CostTracker:
 def __init__(self):
 self.total_input = 0
 self.total_output = 0

 def track(self, usage):
 self.total_input += usage.input_tokens or 0
 self.total_output += usage.output_tokens or 0
 cost = (self.total_input * 0.003 / 1_000_000) + (self.total_output * 0.015 / 1_000_000) # Sonnet example
 return cost

# Wrap every call. Log per-request. Reset daily or per-session.
```

Common Integration Gotchas

- Anthropic requires `max_tokens` on every call. OpenAI defaults exist but set it anyway.
- Tool calling loops can explode costs. Always cap max turns and total tokens.
- Streaming + usage tracking differs. OpenAI final chunk often carries usage. Anthropic reports on message_stop.
- xAI base URL change is easy to forget in multi-provider code. Use a client factory.
- Rate limits are per-model and per-organization. Monitor headers and the usage object every response.
- Structured output is stricter with Anthropic tools than simple JSON mode. Use `strict: True` when possible.
- Long context (>200k on Anthropic 4.6) is standard pricing but still expensive. Cache repeated prefixes.

These patterns work today with the listed SDK versions. Test token counting yourself. The usage object is the only reliable source.