API Comparison at a Glance

Every LLM provider has different SDK conventions, parameter names, and authentication patterns. This table saves you from digging through three different sets of docs when you need to switch or compare providers.

Feature	OpenAI	Anthropic	Google (Gemini)
SDK	`openai`	`anthropic`	`google-genai`
Auth	`OPENAI_API_KEY`	`ANTHROPIC_API_KEY`	`GOOGLE_API_KEY`
Models	gpt-4o, gpt-4.1, o3	claude-sonnet-4, claude-opus-4	gemini-2.5-pro, gemini-2.5-flash
Max output	16K (gpt-4o)	128K (claude-opus-4)	65K (gemini-2.5-pro)
Streaming	Yes	Yes	Yes
Function calling	Yes (tools)	Yes (tools)	Yes (tools)
Vision	Yes	Yes	Yes
System prompt	`system` role	`system` parameter	`system_instruction`

OpenAI API

OpenAI's API is the de facto standard that most other providers emulate. If you learn one API well, learn this one -- many proxy layers and gateways use its format as a universal interface.

Basic Completion

```
from openai import OpenAI
client = OpenAI()  # reads OPENAI_API_KEY

response = client.chat.completions.create(
    model="gpt-4o",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Explain RAG in one paragraph."}
    ],
    temperature=0.7,
    max_tokens=500,
    top_p=0.95,
    frequency_penalty=0.0,
    presence_penalty=0.0,
    stop=["\n\n"]  # optional stop sequences
)
print(response.choices[0].message.content)
```

OpenAI Streaming

```
stream = client.chat.completions.create(
    model="gpt-4o",
```

```
    messages=[{"role": "user", "content": "Hello"}],
    stream=True
)
for chunk in stream:
    delta = chunk.choices[0].delta.content
    if delta:
        print(delta, end="", flush=True)
```

OpenAI Function Calling

```
tools = [{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "Get current weather for a location",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {"type": "string", "description": "City name"},
                "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]}
            },
            "required": ["location"]
        }
    }
}]

response = client.chat.completions.create(
    model="gpt-4o",
    messages=[{"role": "user", "content": "Weather in Paris?"}],
    tools=tools,
    tool_choice="auto"  # or "required" or {"type":"function","function":{"name":"get_weather"}}
)

# Check if tool call was made
tool_calls = response.choices[0].message.tool_calls
if tool_calls:
    name = tool_calls[0].function.name
    args = json.loads(tool_calls[0].function.arguments)
```

Anthropic API

Anthropic's API differs from OpenAI in several key ways: the system prompt is a top-level parameter, content blocks are structured arrays, and tool use returns typed blocks instead of JSON strings.

Basic Completion

```
from anthropic import Anthropic
client = Anthropic()  # reads ANTHROPIC_API_KEY

message = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=1024,
    system="You are a helpful assistant.",
    messages=[
```

```
        {"role": "user", "content": "Explain RAG in one paragraph."}
    ],
    temperature=0.7,
    top_p=0.95,
    stop_sequences=["\n\nHuman:"]
)
print(message.content[0].text)
```

Anthropic Streaming

```
with client.messages.stream(
    model="claude-sonnet-4-20250514",
    max_tokens=1024,
    messages=[{"role": "user", "content": "Hello"}]
) as stream:
    for text in stream.text_stream:
        print(text, end="", flush=True)
```

Anthropic Tool Use

```
tools = [{
    "name": "get_weather",
    "description": "Get current weather for a location",
    "input_schema": {
        "type": "object",
        "properties": {
            "location": {"type": "string", "description": "City name"},
            "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]}
        },
        "required": ["location"]
    }
}]

message = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=1024,
    tools=tools,
    messages=[{"role": "user", "content": "Weather in Paris?"}]
)

# Check for tool use
for block in message.content:
    if block.type == "tool_use":
        tool_name = block.name
        tool_input = block.input  # already a dict
        tool_use_id = block.id
```

Google Gemini API

Gemini uses a distinct SDK structure from OpenAI and Anthropic, with config-based parameter passing and its own tool declaration format. It offers competitive pricing and tight integration with Google Cloud services.

Basic Completion

```
from google import genai

client = genai.Client()  # reads GOOGLE_API_KEY

response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="Explain RAG in one paragraph.",
    config={
        "system_instruction": "You are a helpful assistant.",
        "temperature": 0.7,
        "top_p": 0.95,
        "max_output_tokens": 1024,
    }
)
print(response.text)
```

Gemini Streaming

```
response = client.models.generate_content_stream(
    model="gemini-2.5-flash",
    contents="Hello"
)
for chunk in response:
    print(chunk.text, end="", flush=True)
```

Gemini Function Calling

```
from google.genai import types

weather_tool = types.Tool(
    function_declarations=[
        types.FunctionDeclaration(
            name="get_weather",
            description="Get current weather",
            parameters=types.Schema(
                type="OBJECT",
                properties={
                    "location": types.Schema(type="STRING"),
                    "unit": types.Schema(
                        type="STRING", enum=["celsius", "fahrenheit"]
                    ),
                },
                required=["location"],
            ),
        )
    ]
)

response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="Weather in Paris?",
```

```
    config={"tools": [weather_tool]}
)
```

Parameters Reference

The same concept often has different parameter names across providers. This cross-reference table prevents subtle bugs when porting code between OpenAI, Anthropic, and Gemini.

Parameter	OpenAI	Anthropic	Gemini	Range	Default
Temperature	`temperature`	`temperature`	`temperature`	0.0-2.0	1.0
Top P	`top_p`	`top_p`	`top_p`	0.0-1.0	1.0
Max output	`max_tokens`	`max_tokens`	`max_output_tokens`	1-model max	Varies
Stop sequences	`stop`	`stop_sequences`	`stop_sequences`	List[str]	None
Top K	N/A	`top_k`	`top_k`	1-N	N/A
Frequency penalty	`frequency_penalty`	N/A	`frequency_penalty`	-2.0-2.0	0
Presence penalty	`presence_penalty`	N/A	`presence_penalty`	-2.0-2.0	0
Seed	`seed`	N/A	`seed`	int	None

Structured Output (JSON Mode)

Getting reliable JSON from an LLM is critical for any production pipeline that parses model output programmatically. Each provider handles structured output differently -- some have native JSON mode, others require workarounds.

OpenAI

```
response = client.chat.completions.create(
    model="gpt-4o",
    messages=[{"role": "user", "content": "List 3 cities as JSON"}],
    response_format={"type": "json_object"}
)
```

Anthropic

```
# Use tool_use with a schema for structured output
# Or instruct in prompt: "Respond in valid JSON only"
```

Gemini

```
response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents="List 3 cities as JSON",
    config={"response_mime_type": "application/json"}
)
```

Error Handling

LLM APIs fail in predictable ways -- rate limits, timeouts, and context overflows account for 90% of production errors. Building proper retry logic from the start prevents cascading failures in your application.

Error	HTTP Code	Cause	Action
Rate limit	429	Too many requests	Exponential backoff
Auth error	401	Bad API key	Check key
Context overflow	400	Input too long	Truncate or chunk
Server error	500/503	Provider issue	Retry with backoff
Timeout	N/A	Slow response	Increase timeout, retry
Content filter	400	Safety trigger	Rephrase input

Retry Pattern

```
import time
from openai import RateLimitError, APIError

def call_with_retry(fn, max_retries=5):
    for attempt in range(max_retries):
        try:
            return fn()
        except RateLimitError:
            wait = 2 ** attempt
            time.sleep(wait)
        except APIError as e:
            if e.status_code >= 500:
                time.sleep(2 ** attempt)
            else:
                raise
    raise Exception("Max retries exceeded")
```

Cost Estimation

Token pricing varies by 100x across models, and the gap between input and output costs can be 4-5x. Knowing these numbers before you architect your system prevents budget surprises at scale.

Model	Input (per 1M tokens)	Output (per 1M tokens)
GPT-4o	$2.50	$10.00
GPT-4.1	$2.00	$8.00
GPT-4.1-mini	$0.40	$1.60
Claude Opus 4	$15.00	$75.00
Claude Sonnet 4	$3.00	$15.00
Claude Haiku 3.5	$0.80	$4.00
Gemini 2.5 Pro	$1.25-2.50	$10.00-15.00
Gemini 2.5 Flash	$0.15	$0.60

Token estimation: ~1 token per 4 characters in English; ~1 token per 0.75 words.

Common Pitfalls

These mistakes show up in nearly every first production deployment. Most are trivial to fix if you catch them early, but expensive to debug after launch.

Pitfall	Problem	Fix
No retry logic	Failures on transient errors	Implement exponential backoff
Ignoring rate limits	429 errors cascade	Use rate limiter, queue requests
Hardcoded model names	Breaks on deprecation	Use config/env vars for model names
No timeout	Hung requests	Set `timeout` parameter (30-120s)
Logging full responses	Cost, privacy, storage issues	Log metadata only, redact PII
Not counting tokens	Surprise bills	Pre-count with tiktoken/anthropic-tokenizer
Sync calls in async app	Blocked event loop	Use async client variants
Not handling empty responses	NoneType errors	Check response content before accessing