12

Calling LLMs in Python

SDKs, streaming, structured output, tools & retries

Chapter 12 · the core of the job — and the SDKs feel just like their JS siblings

12 The journey — five moves that make an LLM app

Everything you learned about dicts, JSON, generators and Pydantic converges here. A call is just a request out and an object back — these five ideas are all you need to read any agent codebase.

One golden thread ties them together: the API is stateless. The model remembers nothing — you resend the whole history every time.

12

The mental model

Messages in, a response object out — and nothing is remembered

What you send: a LIST of messages

Each message has a 'role' — one of system, user, or assistant — and its content (the text). That is the whole input.

What you get: a response OBJECT

A nested object you dig into for the text and, if you care, the token counts. It is just a dict/JSON to navigate — a skill you already have.

The single most important fact: the API is STATELESS. The model remembers nothing between calls. 'Conversation memory' is an illusion YOU maintain by resending the entire message history every single time.

Cost of statelessness: longer conversations resend more history, so cost grows with the length of the chat.

12

Making a client

The key comes from the environment — never from your code

```
from openai import OpenAI          # OpenAI SDK
from anthropic import Anthropic    # Anthropic SDK

openai_client = OpenAI()           # reads OPENAI_API_KEY from the environment
anthropic_client = Anthropic()     # reads ANTHROPIC_API_KEY from the environment
```

Never hard-code an API key in source

Neither constructor takes a key in real code — they read it from the environment. A key pasted into a file is the most common way secrets leak into git. This is a smell to flag on sight.

12 OpenAI — Chat Completions

Send messages, then dig the text out of the object

```
resp = openai_client.chat.completions.create(
    model='gpt-5.1',
    messages=[
        {'role': 'system', 'content': 'You are a terse
assistant.'},
        {'role': 'user', 'content': 'Capital of France?'},
    ],
    temperature=0.7,    # 0 = steadier, higher = more varied
    max_tokens=100,     # cap on the response length
)
```

Where the answer lives

`resp.choices[0].message.content`

A LIST of choices, each with a message, whose content is the string 'Paris.'

Token usage (your bill) sits at:
`resp.usage.total_tokens`

temperature controls variety; max_tokens caps length. The text is buried a few levels deep — read the path carefully, it changes per provider.

12

Two flatter surfaces

Same idea, a shorter path to the text

Chat Completions makes you dig. Two newer surfaces hand you the text with less digging — but note how the object graph differs each time.

```
# OpenAI Responses API — the newer, agent-oriented
surface
resp = openai_client.responses.create(
    model='gpt-5.1',
    input='Capital of France?',   # a string, or a
messages list
)
print(resp.output_text)   # => 'Paris.'  no digging
needed
```

```
# Anthropic Messages — system is a top-level param
resp = anthropic_client.messages.create(
    model='claude-sonnet-4-5',
    max_tokens=100,          # REQUIRED on Anthropic
    system='You are a terse assistant.',
    messages=[{'role': 'user', 'content': 'Capital of
France?'}],
)
print(resp.content[0].text)   # => 'Paris.'  a LIST
of blocks
```

Anthropic differences to notice: 'system' sits at the top level (not inside messages), max_tokens is required, and the reply is a content LIST of blocks — because a response can mix text, tool calls and thinking. Same concept, different object graph.

12

JS to Python — the SDKs are siblings

You already know these libraries, almost method-for-method

In Node / TypeScript

```
client.chat.completions.create({ ... })

resp.choices[0].message.content

maxTokens    (camelCase)
an { options } object
```

In Python

```
client.chat.completions.create( ... )

resp.choices[0].message.content

max_tokens  (snake_case)
keyword arguments, no wrapper object
```

The response path is IDENTICAL in both languages. The only Python-isms: keyword arguments instead of an options object, and snake_case names instead of camelCase. If you know the Node SDK, you can read the Python one.

12

Streaming — tokens as a generator

For live UIs, receive the answer piece by piece

```
stream = openai_client.chat.completions.create(
    model='gpt-5.1',
    messages=[{'role': 'user', 'content': 'Write a haiku.'}],
    stream=True,      # now an ITERATOR of chunks
)
for chunk in stream:
    delta = chunk.choices[0].delta.content    # NEW text or None
    if delta:
        print(delta, end='', flush=True)
```

The classic gotcha

The new text lives on delta (the INCREMENT), not message (the whole thing). Each loop pass gives you the next little piece to append.

Anthropic gives a higher-level helper (messages.stream) with a text_stream that hands you just the text deltas, and get_final_message() for the assembled whole. Both SDKs also ship async clients you consume with 'async for'.

12

Structured output — the reliability backbone

Constrain the model to a schema, then validate what comes back

```
from pydantic import BaseModel, ValidationError

class Contact(BaseModel):
    name: str
    email: str
    wants_demo: bool

completion = openai_client.chat.completions.parse(
    model='gpt-5.1',
    messages=[{'role': 'user', 'content': 'Jane Doe, jane@co.com, wants
a demo.'}],
    response_format=Contact,   # SDK sends the schema and parses the
reply
)
contact = completion.choices[0].message.parsed   # a validated Contact
```

The discipline

1. Define a schema (a Pydantic model)
2. Get JSON back
3. VALIDATE it
4. Catch the ValidationError

This replaces the fragile json.loads(text) with no schema.

Anthropic reaches the same goal via a single tool whose schema is your model, or Contact.model_validate_json(text). Whichever path — schema in, JSON out, validate, handle the error.

12

Tool calling — the heart of agents

An agent is just a while loop over a stateless API

```
messages = [{'role': 'user', 'content': 'Weather in Paris?'}]
while True:
    resp = openai_client.chat.completions.create(
        model='gpt-5.1', messages=messages, tools=tools)
    msg = resp.choices[0].message
    messages.append(msg)                # keep the turn in history
    if not msg.tool_calls:              # no tool asked -> done
        print(msg.content); break
    for call in msg.tool_calls:         # model asked to call a tool
        args = json.loads(call.function.arguments)  # a JSON STRING
        result = get_weather(**args)  # run the real function
        messages.append({'role': 'tool',
            'tool_call_id': call.id, 'content': result})
```

Two things MUST be right

1. Tool arguments come back as a JSON STRING you must parse — and validate. The model can emit malformed or hallucinated args.

2. Every tool result must carry the id that ties it to the request.

Anthropic is structurally the same: the model returns a tool_use block, you append a tool_result keyed by its id, and re-call until stop_reason is no longer tool_use. Nothing magical — just a loop.

12 Retries — because rate limits are normal

Wait longer after each failure (exponential backoff)

```
from tenacity import retry, wait_exponential, stop_after_attempt
import openai

@retry(
    retry=retry_if_exception_type(
        (openai.RateLimitError, openai.APIStatusError)),
    wait=wait_exponential(min=1, max=30),   # 1s, 2s, 4s ... up to
30s
    stop=stop_after_attempt(5),
)
def ask(prompt: str) -> str:
    resp = openai_client.chat.completions.create(...)
    return resp.choices[0].message.content
```

Why bother

Rate limits (429) and transient 5xx errors are normal, not rare. The tenacity @retry decorator says the policy declaratively; the hand-rolled version is a for loop with time.sleep(2 ** attempt).

Only retry the RETRYABLE errors — rate limits, 5xx, timeouts. A 400 'bad request' will fail identically every time, so retrying it just wastes time.

JS to Python: tenacity's @retry is the Python spelling of a wrapper like p-retry.

12

Two more habits — cost and injection

Watch what you spend; distrust text you interpolate

Token / cost awareness

Every response carries usage. You are billed per token, so log it. Remember: statelessness means you resend the whole history each turn, so cost grows with conversation length.

```
prompt = f'Summarise this ticket:

{user_ticket}'
```

Prompt injection risk

The moment user_ticket holds untrusted text, that text can carry instructions the model may follow ('ignore previous instructions and...'). Treat interpolated user content as DATA, not trusted instructions.

The defence: keep user text clearly delimited, put your real instructions in the SYSTEM role, and never let a template blindly concatenate a secret or another user's data into the prompt.

12

Reading & judging — the smells to hunt

Spotting these is most of what reviewing AI code means

Hard-coded API keys in source	should read from os.environ
No retry / backoff around the call	one rate-limit blip kills the job
No timeout on the HTTP client	a hung call wedges the worker
Unvalidated model JSON used directly	json.loads(out), no schema, no try
Swallowed API errors	a bare except: pass hides failures
Unbounded concurrency	thousands of calls, no semaphore
Fragile tool loop	assumes exactly one clean tool call
Injection-unsafe templating	user text glued into instructions

Each is a concrete, checkable red flag — none needs you to run the code to see it.

12 Putting it together

`stateless API`	→	`resend the full history every call`
`choices[0].message.content`	→	`dig for the text`
`stream=True`	→	`iterate chunk.delta.content`
`response_format=Model`	→	`validated Pydantic object`
`tool_calls loop`	→	`parse JSON args, return by id`
`@retry backoff`	→	`only on 429 / 5xx / timeout`

The five moves in one line each:

1. Messages in, object out — nothing is remembered.
2. Stream for live UIs (the delta gotcha).
3. Schema in, validated object out.
4. Tools = a while loop that parses and re-feeds.
5. Wrap it in retries; treat user text as data.

The teaching — a call is a request out and an object back, over a stateless API. Master these five moves and you can read, build, and JUDGE the code the whole agent world is written in.