

LLM APIs & SDKs

Working with Anthropic Claude and OpenAI APIs — from first call to production patterns.

Last reviewed 28 Sep 2026 ·

Prerequisites: [Python for DE](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/) · [Prompt Engineering](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/prompt-engineering/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/prompt-engineering/)

Related: [AI Agents](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/) · [AI Observability](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

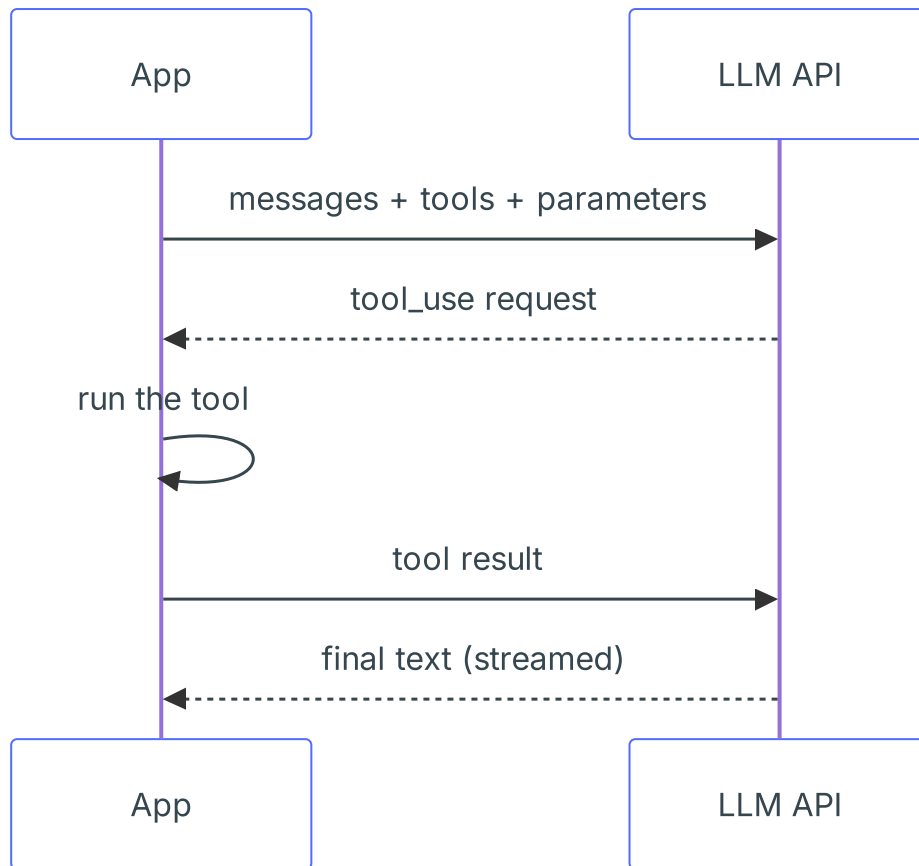
Overview

Challenge: Chat interfaces suit one-off questions, but automated workloads — classifying millions of support tickets, extracting fields from invoices every hour, or summarizing failed job logs — require calling the model programmatically.

Solution: an LLM API is an HTTP endpoint, usually accessed through an SDK, that accepts a model name, instructions, input, and settings, and returns generated content with token usage. It is an external service like any other in a pipeline, with the same concerns: authentication, rate limits, retries, cost, latency, and logging.

```
your code → client.messages.create(model, system,
messages, tools, ...) → LLM provider
                    ← content blocks (thinking / text / tool_use) +
usage (tokens) ←
```

Key design constraints: *tokens* (billing is per input and output token, and context windows are finite), *latency* (seconds rather than milliseconds — use streaming, batching, or concurrency), and *non-determinism* (validate outputs rather than trusting them).



On this page

Basic - [Provider Comparison](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#provider-comparison) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#provider-comparison) - [Anthropic SDK Setup](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#anthropic-sdk-setup) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#anthropic-sdk-setup) - [First API Call](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#first-api-call) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#first-api-call) - [OpenAI SDK Setup](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#openai-sdk-setup) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#openai-sdk-setup)

Intermediate - [Key Parameters](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#key-parameters) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#key-parameters) - [Streaming](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#streaming) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#streaming) - [Tool Use / Function Calling](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#tool-use--function-calling) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#tool-use--function-calling) - [Vision \(Image Input\)](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#vision-image-input) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#vision-image-input)

Advanced - [Structured Outputs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#structured-outputs) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#structured-outputs) - [Prompt Caching \(Anthropic\)](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#prompt-caching-anthropic) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#prompt-caching-anthropic) - [Batching](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#batching) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#batching) - [Production Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#production-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#production-patterns)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/#further-reading)

Provider Comparison

	Anthropic	OpenAI
Top model	Claude Fable 5.1 (demanding reasoning, long-horizon agents) · Claude Opus 5.5	GPT-6 family (gpt-6-astra , gpt-6-sol)
Fast model	Claude Haiku 4.5	gpt-6-luna
API style	Messages API	Responses API (recommended for new projects) · Chat Completions
Tool use	Yes	Yes (function calling)
Vision	Yes	Yes
Structured output	Native JSON schema (output_config.format , messages.parse)	response_format with a JSON schema
Prompt caching	Yes (cache_control)	Yes
Python SDK	anthropic	openai

Model names change often — check [Anthropic's models overview](https://platform.claude.com/docs/en/models/overview) (https://platform.claude.com/docs/en/models/overview) and [OpenAI's models page](https://developers.openai.com/api/docs/models) (https://developers.openai.com/api/docs/models) before choosing. OpenAI examples below use gpt-6-sol .

Anthropic SDK Setup

```
pip install anthropic
```

```
import anthropic
import os

# Client reads ANTHROPIC_API_KEY from environment by
# default
client = anthropic.Anthropic()

# Or explicitly
client =
anthropic.Anthropic(api_key=os.environ["ANTHROPIC_API_KEY"]
)
```

```
# .env
ANTHROPIC_API_KEY=sk-ant-...
```

Model IDs (Claude)

```
# Current models (as of September 2026) – list them live
with client.models.list()
CLAUDE_FABLE    = "claude-fable-5-1"        # demanding
reasoning and long-horizon agentic work
CLAUDE_OPUS     = "claude-opus-5-5"         # Anthropic's
suggested starting point for most workloads
CLAUDE_SONNET   = "claude-sonnet-5-5"       # the best
combination of speed and intelligence
CLAUDE_HAIKU    = "claude-haiku-4-5"        # fastest, lowest
cost
```

Older IDs such as `claude-sonnet-5` stay available for a while, but they are marked legacy. Every model has a published retirement date, and among the current models Haiku 4.5's is the nearest: check [model deprecations](https://platform.claude.com/docs/en/about-claude/model-deprecations) (<https://platform.claude.com/docs/en/about-claude/model-deprecations>) before you pin a model in a long-lived pipeline.

First API Call

```
import anthropic

client = anthropic.Anthropic()
```

```

message = client.messages.create(
    model="claude-sonnet-5-5",
    max_tokens=1024,
    messages=[
        {"role": "user", "content": "Explain what a data
lakehouse is in 3 bullet points."}
    ]
)

print(next(b.text for b in message.content if b.type ==
"text"))

```

With a system prompt

```

message = client.messages.create(
    model="claude-sonnet-5-5",
    max_tokens=1024,
    system="You are a concise technical writer. Always use
bullet points.",
    messages=[
        {"role": "user", "content": "What is Apache
Iceberg?"}
    ]
)

```

Response object

```

message.id          # unique message ID
message.model       # model used
message.stop_reason  # "end_turn" | "max_tokens" |
"stop_sequence" | "tool_use" | "pause_turn" | "refusal" |
"model_context_window_exceeded"
message.usage       # Usage(input_tokens=45,
output_tokens=210, ...)
message.content     # list of content blocks: "thinking",
"text", "tool_use", ...

# Text response – current models may return a thinking

```

```
block first, so find the text block
next(b.text for b in message.content if b.type == "text")
```

OpenAI SDK Setup

```
pip install openai
```

```
from openai import OpenAI

client = OpenAI() # reads OPENAI_API_KEY from env

# Responses API: the recommended interface for new projects
response = client.responses.create(
    model="gpt-6-sol",
    instructions="You are a helpful data engineer.",
    input="What is the difference between a fact and a
dimension table?",
    max_output_tokens=1024,
)

print(response.output_text)
response.usage.input_tokens, response.usage.output_tokens
```

The Chat Completions API is still supported and appears in a lot of existing code.

`max_tokens` is deprecated there in favour of `max_completion_tokens`, so use the new name:

```
response = client.chat.completions.create(
    model="gpt-6-sol",
    max_completion_tokens=1024,
    messages=[
        {"role": "system", "content": "You are a helpful
data engineer."},
        {"role": "user", "content": "What is the
difference between a fact and a dimension table?"}
    ]
)

print(response.choices[0].message.content)
```

```
# Token usage
response.usage.prompt_tokens
response.usage.completion_tokens
response.usage.total_tokens
```

Key Parameters

Parameter	Description	Typical values
<code>model</code>	Which model to use	see model IDs above
<code>max_tokens</code>	Max output tokens (OpenAI Chat Completions: <code>max_completion_tokens</code>)	256–4096 for most tasks
<code>temperature</code>	Randomness (0=deterministic, 1=creative)	Not a parameter of the Python SDK v1.0 and later (passing it raises <code>TypeError</code>). Opus 4.7 and later models, including Opus 5.5, and Sonnet 5.5 return a 400 for a non-default value. Earlier models, Haiku 4.5 among them, still accept it in the request body: <code>extra_body={"temperature": 0}</code>
<code>top_p</code>	Nucleus sampling (alternative to temperature)	Same restrictions as <code>temperature</code> ; <code>top_k</code> too
<code>output_config</code>	Effort (<code>{"effort": "low"... "max"}</code>) and structured output format	The main control on current Claude models. Haiku 4.5 does not support <code>effort</code>
<code>stop_sequences</code>	Stop generation at these strings	<code>["\n\n", "END"]</code>
<code>system</code>	System prompt (Anthropic)	Instructions, persona, format

```
# For data extraction on a model that still accepts
sampling parameters – want determinism.
# The Python SDK does not define `temperature`, so it goes
```



```

in the request body
message = client.messages.create(
    model="claude-haiku-4-5-20251001",
    max_tokens=512,
    extra_body={"temperature": 0},      # deterministic
    messages=[{"role": "user", "content": "Extract the
table name from: SELECT * FROM orders"}]
)

# For creative content generation on current models – no
sampling params (they return a 400);
# ask for variety in the prompt and tune effort instead
message = client.messages.create(
    model="claude-sonnet-5-5",
    max_tokens=1024,
    output_config={"effort": "low"},    # low | medium |
high | xhigh | max
    messages=[{"role": "user", "content": "Write 3 clearly
different error message suggestions for a failed
pipeline."}]
)

```

Streaming

Stream tokens as they're generated — essential for interactive UIs and long outputs.

```

# Anthropic streaming
with client.messages.stream(
    model="claude-sonnet-5-5",
    max_tokens=2048,
    messages=[{"role": "user", "content": "Explain PySpark
window functions in detail."}]
) as stream:
    for text in stream.text_stream:
        print(text, end="", flush=True)

# Get the full message after streaming
message = stream.get_final_message()
print(f"\nTotal tokens: {message.usage.input_tokens +
message.usage.output_tokens}")

```

```
# OpenAI streaming
stream = client.chat.completions.create(
    model="gpt-6-sol",
    max_completion_tokens=2048,
    stream=True,
    messages=[{"role": "user", "content": "Explain PySpark
window functions."}]
)

for chunk in stream:
    if chunk.choices[0].delta.content:
        print(chunk.choices[0].delta.content, end="",
flush=True)
```

```
# Async streaming (for FastAPI / async apps)
import asyncio
import anthropic

async_client = anthropic.AsyncAnthropic()

async def stream_response(prompt: str):
    async with async_client.messages.stream(
        model="claude-sonnet-5-5",
        max_tokens=1024,
        messages=[{"role": "user", "content": prompt}]
    ) as stream:
        async for text in stream.text_stream:
            yield text
```

Tool Use / Function Calling

Let the model call functions you define — the model decides when to call them and with what arguments.

```
import anthropic
import json

client = anthropic.Anthropic()
```

```

# Define tools
tools = [
    {
        "name": "run_sql_query",
        "description": "Execute a SQL query against the
data warehouse and return results",
        "input_schema": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "The SQL query to
execute"
                },
                "database": {
                    "type": "string",
                    "description": "Target SQL dialect",
                    "enum": ["postgres", "bigquery",
"snowflake", "redshift", "spark"]
                }
            },
            "required": ["query", "database"]
        }
    },
    {
        "name": "get_table_schema",
        "description": "Get the schema (column names and
types) for a given table",
        "input_schema": {
            "type": "object",
            "properties": {
                "table_name": {"type": "string"}
            },
            "required": ["table_name"]
        }
    }
]

# Simulated tool executor
def execute_tool(name: str, inputs: dict) -> str:
    if name == "run_sql_query":
        return json.dumps({"rows": [{"count": 1523}]},

```

```

"elapsed_ms": 340})
    if name == "get_table_schema":
        return json.dumps({"columns": [
            {"name": "order_id", "type": "VARCHAR"},
            {"name": "amount", "type": "DECIMAL(10,2)"},
            {"name": "status", "type": "VARCHAR"},
        ]})
    return "Tool not found"

# Agentic loop
messages = [{"role": "user", "content": "How many orders
are in the orders table?"}]

while True:
    response = client.messages.create(
        model="claude-sonnet-5-5",
        max_tokens=1024,
        tools=tools,
        messages=messages
    )

    # If model is done, print and exit
    if response.stop_reason == "end_turn":
        for block in response.content:
            if hasattr(block, "text"):
                print(block.text)
            break

    # If model wants to use a tool
    if response.stop_reason == "tool_use":
        # Add assistant's response to messages
        messages.append({"role": "assistant", "content":
response.content})

        # Execute each tool call
        tool_results = []
        for block in response.content:
            if block.type == "tool_use":
                result = execute_tool(block.name,
block.input)

                tool_results.append({
                    "type": "tool_result",

```

```

        "tool_use_id": block.id,
        "content": result
    })

    # Add tool results to messages
    messages.append({"role": "user", "content":
tool_results})

```

Vision (Image Input)

Send images alongside text for analysis, OCR, chart reading, etc.

```

import anthropic
import base64
from pathlib import Path

client = anthropic.Anthropic()

# Option 1: Base64 encode a local image
image_data =
base64.standard_b64encode(Path("pipeline_diagram.png").read
_bytes()).decode("utf-8")

message = client.messages.create(
    model="claude-sonnet-5-5",
    max_tokens=1024,
    messages=[
        {
            "role": "user",
            "content": [
                {
                    "type": "image",
                    "source": {
                        "type": "base64",
                        "media_type": "image/png",
                        "data": image_data,
                    },
                },
                {
                    "type": "text",

```

```

        "text": "Describe this data pipeline
        diagram. List each component and how they connect."
    },
],
}
],
)

# Option 2: Image from URL
message = client.messages.create(
    model="claude-sonnet-5-5",
    max_tokens=1024,
    messages=[
        {
            "role": "user",
            "content": [
                {
                    "type": "image",
                    "source": {
                        "type": "url",
                        "url":
"https://example.com/chart.png"
                    }
                },
                {"type": "text", "text": "What trend does
this chart show?"}
            ]
        }
    ]
)

```

Structured Outputs

Anthropic: native structured outputs (recommended)

```

from pydantic import BaseModel

class PipelineMetadata(BaseModel):
    pipeline_name: str
    schedule: str | None

```

```

source_system: str
destination: str
is_incremental: bool
estimated_rows: int | None

response = client.messages.parse(
    model="claude-sonnet-5-5",
    max_tokens=1024,
    messages=[{"role": "user", "content": "We have a
nightly job that pulls 50k new transactions from the
payments API and loads them into the warehouse at 2am."}],
    output_format=PipelineMetadata,
)
metadata = response.parsed_output      # validated
PipelineMetadata instance

```

Anthropic: tool-based structured output

Useful when the model should *choose* between several tools. Add `"strict": True` to guarantee the arguments match the schema. Forcing a specific tool (`tool_choice= {"type": "tool", ...}` or `{"type": "any"}`) is rejected with a 400 on Claude Opus 5.5, Sonnet 5.5 and Fable 5.1, so on those models use native structured outputs when you need one specific shape. Earlier models, such as Sonnet 5, still accept a forced tool.

```

tools = [{
    "name": "extract_pipeline_metadata",
    "description": "Extract structured metadata from a
pipeline description",
    "strict": True,
    "input_schema": {
        "type": "object",
        "properties": {
            "pipeline_name": {"type": "string"},
            "schedule": {"type": "string"},
            "description": "cron expression or plain English",
            "source_system": {"type": "string"},
            "destination": {"type": "string"},
            "is_incremental": {"type": "boolean"},
            "estimated_rows": {"type": "integer"}
        },
        "required": ["pipeline_name", "schedule",
"source_system", "destination",

```

```

        "is_incremental", "estimated_rows"],
        "additionalProperties": False
    }
}]

response = client.messages.create(
    model="claude-sonnet-5-5",
    max_tokens=512,
    tools=tools,
    tool_choice={"type": "auto"}, # the default; naming
the tool here is rejected on current models
    messages=[{"role": "user", "content": "Use the
extract_pipeline_metadata tool on this description. We have
a nightly job that pulls 50k new transactions from the
payments API and loads them into the warehouse at 2am."}]
)

# Extract the structured result (the model may also add
text or thinking blocks)
for block in response.content:
    if block.type == "tool_use":
        metadata = block.input
        print(metadata)
# Example output: {'pipeline_name':
'payments_transactions_load', 'schedule': '0 2 * * *',
# 'source_system': 'payments API', 'destination':
'warehouse',
# 'is_incremental': True, 'estimated_rows': 50000}

```

OpenAI: JSON mode

`json_object` only guarantees *valid JSON*; for schema-valid output use `response_format=`
`{"type": "json_schema", ...}` or the SDK's `client.chat.completions.parse(...,`
`response_format=PydanticModel)`.

```

from openai import OpenAI
import json

client = OpenAI()

response = client.chat.completions.create(
    model="gpt-6-sol",

```

```

    response_format={"type": "json_object"},
    messages=[
        {"role": "system", "content": "Always respond with valid JSON."},
        {"role": "user", "content": "Extract pipeline name, schedule, and source from: nightly payments-API-to-warehouse job at 2am"}
    ]
)

data = json.loads(response.choices[0].message.content)

```

Prompt Caching (Anthropic)

Cache long, repeated content (system prompts, documents) to reduce cost and latency. Cache reads cost a fraction of the normal input price: about a tenth on most models, and less on some (see the pricing page). Writing to the cache costs more than a normal input token.

```

# Mark content for caching with cache_control: {"type":
"ephemeral"}
# Ephemeral cache = 5 minutes TTL by default (resets on
each use); add "ttl": "1h" for a one-hour cache

long_document = Path("data_dictionary.md").read_text()

response = client.messages.create(
    model="claude-sonnet-5-5",
    max_tokens=1024,
    system=[
        {
            "type": "text",
            "text": "You are a data dictionary assistant.
Answer questions about the schema.",
        },
        {
            "type": "text",
            "text": long_document,
            "cache_control": {"type": "ephemeral"} # cache
this large document

```

```

        }
    ],
    messages=[{"role": "user", "content": "What columns
does the orders table have?"}]
)

# Check cache usage
print(response.usage.cache_creation_input_tokens) # tokens
written to cache (first call)
print(response.usage.cache_read_input_tokens)      #
tokens read from cache (subsequent calls)

```

When to cache: - Large system prompts. A prompt below the model's minimum cacheable length is processed normally with no error and nothing cached; the minimum depends on the model (512 to 4,096 tokens today, 4,096 for Haiku 4.5), so check that `cache_read_input_tokens` is above zero on the second call - Reference documents (data dictionaries, schemas, codebases) - Few-shot examples at the start of the system prompt - Multi-turn conversations where the context grows large

Batching

For offline workloads (document processing, bulk classification), use the Batch API — 50% cheaper. Most batches finish within an hour, and results are available when every request has finished or after 24 hours, whichever comes first.

```

# Anthropic Batch API
requests = []
for i, row in enumerate(data):
    requests.append({
        "custom_id": f"row-{i}",
        "params": {
            "model": "claude-haiku-4-5-20251001",
            "max_tokens": 256,
            "messages": [{"role": "user", "content":
f"Classify this log: {row['log']}"}]
        }
    })

batch = client.messages.batches.create(requests=requests)
print(f"Batch ID: {batch.id}, status:
{batch.processing_status}")

```

```

# Poll for completion
import time
while True:
    batch = client.messages.batches.retrieve(batch.id)
    if batch.processing_status == "ended":
        break
    time.sleep(60)

# Retrieve results
for result in client.messages.batches.results(batch.id):
    print(result.custom_id, next(b.text for b in
    result.result.message.content if b.type == "text"))

```

Production Patterns

Retry with exponential backoff

```

import time
import anthropic
from anthropic import RateLimitError, APIStatusError

def call_with_retry(client, max_retries=3, **kwargs):
    for attempt in range(max_retries):
        try:
            return client.messages.create(**kwargs)
        except RateLimitError:
            wait = 2 ** attempt
            print(f"Rate limited. Waiting {wait}s...")
            time.sleep(wait)
        except APIStatusError as e:
            if e.status_code >= 500: # server error –
                retry
                time.sleep(2 ** attempt)
            else:
                raise # client error – don't retry
    raise RuntimeError("Max retries exceeded")

```

Cost tracking

```

# Prices change often, so keep them in config, not code.
pricing.json holds USD per 1M tokens:
# {"claude-sonnet-5-5": {"input": 2.00, "output": 10.00},
...} (example values)
# Current prices:
https://platform.claude.com/docs/en/about-claude/pricing
import json
from pathlib import Path

PRICING = json.loads(Path("pricing.json").read_text())
# Ignores prompt-cache pricing and the 50% batch discount

def estimate_cost(response) -> float:
    model = response.model
    if model not in PRICING:
        return 0.0
    p = PRICING[model]
    input_cost = response.usage.input_tokens / 1_000_000
    * p["input"]
    output_cost = response.usage.output_tokens / 1_000_000
    * p["output"]
    return input_cost + output_cost

response = client.messages.create(...)
print(f"Cost: ${estimate_cost(response):.6f}")

```

Async for concurrent calls

```

import asyncio
import anthropic

async_client = anthropic.AsyncAnthropic()

async def classify(text: str, idx: int) -> dict:
    response = await async_client.messages.create(
        model="claude-haiku-4-5-20251001",
        max_tokens=64,
        extra_body={"temperature": 0},      # Haiku 4.5
    )
    accepts it; the SDK does not define it
    messages=[{"role": "user", "content": f"Classify as
PASS or FAIL: {text}"}]

```

```

    )
    return {"idx": idx, "result": next(b.text for b in
response.content if b.type == "text").strip()}

async def classify_all(texts: list[str]) -> list[dict]:
    tasks = [classify(text, i) for i, text in
enumerate(texts)]
    return await asyncio.gather(*tasks)

results = asyncio.run(classify_all(["Row count > 0", "NULL
in required field", "Schema matches expected"]))

```

Logging all LLM calls

```

import logging
import uuid
from functools import wraps

logger = logging.getLogger("llm_calls")

def log_llm_call(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        call_id = str(uuid.uuid4())[:8]
        logger.info(f"[{call_id}] LLM call | model=
{kwargs.get('model')} | "
                    f"prompt_preview=
{str(kwargs.get('messages', ''))[:100]}")
        response = func(*args, **kwargs)
        logger.info(f"[{call_id}] LLM done | tokens=
{response.usage.input_tokens}+
{response.usage.output_tokens} | "
                    f"stop={response.stop_reason}")
        return response
    return wrapper

@log_llm_call
def create_message(client, **kwargs):
    return client.messages.create(**kwargs)

```

Common Pitfalls

Pitfall	Symptom	Fix
Reading <code>response.content</code> <code>[0].text</code>	Crashes or returns empty text when the first block is a thinking or tool block	Iterate the blocks and pick <code>type == "text"</code>
Not checking <code>stop_reason</code>	Truncated JSON (<code>max_tokens</code>), unhandled tool calls, or silent refusals	Handle <code>max_tokens</code> , <code>tool_use</code> , <code>pause_turn</code> , and <code>refusal</code> explicitly
<code>max_tokens</code> set too low	Output cut off mid-sentence or mid-JSON	Generous limits (thousands, not hundreds) for generation; stream long outputs
Copying parameters between models	400 errors — e.g. <code>temperature</code> on Opus 4.7 and later or Sonnet 5.5, forced <code>tool_choice</code> on Opus 5.5 / Sonnet 5.5 / Fable 5.1, assistant prefill on the 4.6+ family; and a <code>TypeError</code> for <code>temperature</code> in the Python SDK v1.0+	Check the model's supported parameters; control behavior with <code>effort</code> and structured outputs
API keys in code or notebooks	Leaked keys and surprise bills	Environment variables or a secrets manager; separate keys per environment with spend limits
Unbounded <code>asyncio.gather</code> over thousands of calls	429 rate-limit storms	Cap concurrency with a semaphore; the Batch API for offline work
Rebuilding a big identical prefix on every call	Paying full input price for the same system prompt and documents	Prompt caching — stable content first, <code>cache_control</code> on it
Synchronous calls in a row-by-row pipeline	Jobs that take days	Batch API (50% cheaper, asynchronous) or bounded concurrency
No logging of prompts, outputs, and token usage	Can't debug bad outputs or explain the bill	Log model, prompt version, tokens, latency, stop reason, and request ID

Pitfall	Symptom	Fix
Trusting output as data	Invalid values flow into the warehouse	Validate with schemas; quarantine failures like any bad record

Cheat Sheet

Task	Anthropic (Python)
Basic call	<code>client.messages.create(model="claude-sonnet-5-5", max_tokens=1024, messages=[{"role": "user", "content": "..."}])</code>
Get the text	<code>next(b.text for b in r.content if b.type == "text")</code>
System prompt	<code>system="You are..."</code>
Reasoning depth	<code>output_config={"effort": "low" "medium" "high" "xhigh" "max"}</code>
Stream	<code>with client.messages.stream(...) as s: for t in s.text_stream: ... → s.get_final_message()</code>
Structured output	<code>client.messages.parse(..., output_format=PydanticModel).parsed_output</code>
Tools	<code>tools=[{"name", "description", "input_schema", "strict": True}]</code> → <code>handle tool_use</code> → <code>send back tool_result</code>
Cache a big prefix	<code>cache_control={"type": "ephemeral"}</code> (top-level automatic) or on a specific block
Count tokens before sending	<code>client.messages.count_tokens(model=..., messages=...)</code>
Bulk offline jobs	<code>client.messages.batches.create(requests=[...])</code> → <code>poll</code> → <code>batches.results(id)</code>
Available models	<code>client.models.list()</code>
Retries / timeouts	<code>anthropic.Anthropic(max_retries=5, timeout=60.0)</code>

Concept	Anthropic	OpenAI
Endpoint	Messages API	Responses (recommended) / Chat Completions
System prompt	<code>system=</code> parameter	<code>system / developer</code> message or <code>instructions</code>
Output location	<code>content blocks</code>	<code>choices[0].message.content / output items</code>
JSON schema output	<code>output_config.format / messages.parse</code>	<code>response_format / .parse()</code>
Usage	<code>usage.input_tokens , usage.output_tokens</code>	<code>usage.prompt_tokens , usage.completion_tokens</code>
Offline discount	Message Batches (50%)	Batch API (50%)

Choosing a model: Anthropic suggests starting with Opus for most workloads, or with Sonnet where speed and cost matter · measure on your eval set · move up (Fable) when quality still falls short at higher effort · move down (Sonnet, Haiku) for high-volume classification or extraction once evals prove it's good enough

Interview Questions

Q: What are tokens, and why do they matter when you use an LLM API? A: Tokens are the units models read and write — roughly 3–4 characters of English text on average. They matter three ways: cost (billed per input and output token, with output usually several times pricier), limits (context window and `max_tokens` cap how much fits in and comes out), and latency (output tokens are generated sequentially, so long outputs take longer). Count tokens before large calls, and design prompts and outputs to be as short as the task allows.

Q: How would you process a million records with an LLM in a data pipeline? A: Offline, with the provider's batch API — roughly half price, with results in hours — submitting records in chunks, keyed by a `custom_id` so results can be joined back regardless of order. Use a cheap model validated on a sample, cache the shared instructions, and request structured output. Treat it like any pipeline: make it idempotent (skip records already processed), validate and quarantine bad outputs, track cost per run, and write results to a table with the model and prompt version recorded.

Q: What is prompt caching and when does it help? A: The provider stores the processed prefix of a prompt — system instructions, tool definitions, large documents — so later requests with the same prefix are cheaper (on Claude, cache reads cost about a tenth of normal input) and faster. It helps when a large, stable prefix is reused across many calls: RAG with a fixed knowledge base, long system prompts, multi-turn conversations, agent loops. Caching is a prefix match, so put stable content first and anything varying (timestamps, user questions) after it.

Q: How do you handle rate limits and transient errors? A: Retry 429s and 5xx errors with exponential backoff and jitter, respecting the `retry-after` header — the official SDKs do this automatically with a configurable retry count. Don't retry 4xx client errors like invalid requests. At the system level, cap concurrency, spread load with queues or batch APIs, and alert when retries or error rates climb.

Q: How does tool use (function calling) work? A: You describe tools with a name, a description, and a JSON schema for the inputs. The model decides whether to call one and returns a `tool_use` block with arguments instead of (or before) a final answer. Your code executes the tool, sends the result back as a `tool_result`, and the model continues — possibly calling more tools — until it produces a final answer. The model never executes anything itself; your code stays in control of what actually runs.

Q: Why might the same prompt give different answers on different runs, and how do you deal with it? A: Generation involves sampling, and current reasoning models don't expose a temperature knob at all, so outputs vary. For pipelines, reduce variance where it matters: constrain outputs with structured schemas or enumerations, give clear rubrics and examples, validate results in code, and measure consistency on an eval set. When you need a stable answer for a given input, store it rather than regenerating it.

Further Reading

- [Claude API documentation](https://platform.claude.com/docs/en/api/overview) (https://platform.claude.com/docs/en/api/overview) and [Python SDK](https://github.com/anthropics/anthropic-sdk-python) (https://github.com/anthropics/anthropic-sdk-python)
- [Claude models overview](https://platform.claude.com/docs/en/models/overview) (https://platform.claude.com/docs/en/models/overview) and [pricing](https://platform.claude.com/docs/en/about-claude/pricing) (https://platform.claude.com/docs/en/about-claude/pricing)
- [Prompt caching](https://platform.claude.com/docs/en/build-with-claude/prompt-caching) (https://platform.claude.com/docs/en/build-with-claude/prompt-caching) and [Message Batches](https://platform.claude.com/docs/en/build-with-claude/batch-processing) (https://platform.claude.com/docs/en/build-with-claude/batch-processing)
- [OpenAI API reference](https://developers.openai.com/api/reference/overview) (https://developers.openai.com/api/reference/overview)
- [Claude Cookbooks](https://github.com/anthropics/claude-cookbooks) (https://github.com/anthropics/claude-cookbooks) — runnable notebooks for tool use, RAG, extraction, and more
- [Model deprecations](https://platform.claude.com/docs/en/about-claude/model-deprecations) (https://platform.claude.com/docs/en/about-claude/model-deprecations) — retirement dates for every Claude model

Previous: [Prompt Engineering](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/prompt-engineering/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/prompt-engineering/) · **Next:** [Embeddings](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/) · **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 28, 2026