AI APIs: Calling LLMs in Production

Running examples: **TerpGrader** and **TerpTutor** · Code: `claude-api-demos/` (demos `00` – `11`)

Learning Objectives

By the end of this lecture you should be able to:

1. Store an API key safely (**env var** or git-ignored `.env`), and explain why it never reaches a browser, app, or repo
2. Trace the **five-step request lifecycle** and dissect an API call — endpoint, headers, body, response, `usage`
3. Build a **multi-turn conversation** over a stateless API, and shape behavior with a **system prompt**
4. Explain what **temperature** does to sampling — and why current Claude models no longer expose it
5. **Stream** responses over SSE, and explain why it improves *perceived* latency
6. Get machine-readable output: **stop sequences**, **prefill** (older models), **structured outputs**
7. Classify errors as **retryable or not**, implement **backoff with jitter**, and **compute the cost** of a request
8. Use **LiteLLM** to switch between OpenAI, Gemini, Anthropic, the Claude SDK, and Ollama with one flag

From "It Works on My Laptop" to Production

In the Ollama lecture we called `client.messages.create(...)` and moved on. That one line hides a discipline. Once a class project becomes a service, five questions arrive at once:

- **Auth** — where does the API key live, and who can steal it?
- **Behavior** — how do I make it a tutor, a grader, or a JSON generator?
- **Latency** — why is the user staring at a spinner for 20 seconds?
- **Reliability** — what happens on `429 Too Many Requests` at the worst moment?
- **Money and lock-in** — who pays for the tokens, and can we switch providers next month?

> Most of these are **distributed-systems and operations problems** wearing an AI costume — you already have the tools from REST, FastAPI, and testing.

The Mental Model

> An LLM API is a **metered, throttled, remote function call** — a function in someone else's datacenter.

- every call presents a **credential** → auth
- the function has **no memory** between calls → you resend the conversation
- every call is **metered** in tokens, billed in dollars → cost
- the provider enforces a **speed limit** per customer → rate limits
- the answer crosses a network that can be slow or flaky → streaming, retries

None of this shows up in a happy-path demo. All of it shows up in production.

Connection to Earlier Material

Lecture	What carries over
REST	An LLM API *is* REST-ish: `POST` JSON, get JSON, act on the status code
LLMs	Tokens, embeddings, next-token sampling happen inside the provider; temperature *is* that sampling knob
Ollama	A local model server speaks HTTP too — the same client code can talk to it (LiteLLM exploits this)

Running Examples: TerpGrader and TerpTutor

TerpGrader — the back end

Grades **500 short-answer submissions** per assignment: ~900-token prompt (rubric + answer) → ~150-token verdict.

- runs **unattended** → auth from the environment
- needs **parseable JSON**
- 500 back-to-back requests **will** hit rate limits
- real money → **predict and cap** the spend

TerpTutor — the front end

A math-help chatbot students talk to.

- must **remember** the conversation → multi-turn
- must act like a **tutor**: hints, not answers → system prompt
- must feel **responsive** → streaming
- try **several providers**, incl. a free local model → LiteLLM

Core Concept 1

Getting and Storing API Keys

Authentication

Authentication binds each request to an account — for billing, rate limiting, and access control.

- **Static API keys** — a secret string sent in a header
- **OAuth bearer tokens** — short-lived, from a login flow
- **Cloud IAM signing** — when the model is served via AWS Bedrock or Google Vertex AI

An API key is **a credit card number that can also talk to an AI**. Anyone holding it can spend your money, up to your rate limit, until you notice.

Getting a Key

Anthropic

1. Log in at `console.anthropic.com`
2. **Get API Keys → Create Key**
3. Pick a workspace; name it (e.g. `CMSC389A`) — the name is for you
4. **Copy it now.** It's shown exactly once — lose it, delete it, make a new one

Everyone else works the same way

- **OpenAI** — `platform.openai.com/api-keys`
- **Gemini** — `aistudio.google.com/apikey`
- **Ollama** (local) — no key at all, only a URL

The Lifecycle of a Key, Done Right

```
create (provider console, least privilege / per-workspace)
   |
   ▼
store  ──► dev:  environment variable, or a .env file (git-ignored)
       ──► prod: secret manager (AWS Secrets Manager, GCP Secret
                 Manager, Vault) injected into the process env
   |
   ▼
use    ──► read from the environment at startup; never hardcode, never log
   |
   ▼
rotate ──► periodically, and IMMEDIATELY on any suspected leak
```

Option A: An Environment Variable

A named string the OS hands to every process started from a shell. Your code asks for it **by name**, so the secret never appears in source.

```
# macOS / Linux, this terminal only
export ANTHROPIC_API_KEY="sk-ant-api03-..."
export OPENAI_API_KEY="sk-proj-..."
export GEMINI_API_KEY="AIza..."
echo ${ANTHROPIC_API_KEY:0:10}  # prefix only
```

```
# Windows PowerShell
$env:ANTHROPIC_API_KEY = "sk-ant-..."   # this window
setx ANTHROPIC_API_KEY "sk-ant-..."     # permanent
```

Permanent on macOS/Linux: append the `export` to `~/.zshrc` / `~/.bashrc`, then `source` it.

```
import os
api_key = os.environ["ANTHROPIC_API_KEY"]
# KeyError (a loud crash) if missing
```

You rarely need even that: `Anthropic()`, `OpenAI()`, and LiteLLM look up their standard variable names on their own.

Option B: A `.env` File

A plain-text file of `NAME=value` lines. **`python-dotenv`** copies each line into `os.environ` — the rest of your code can't tell the difference.

```
# .env  -- NEVER commit this file
ANTHROPIC_API_KEY="sk-ant-api03-...your key..."
```

```
from dotenv import load_dotenv
load_dotenv()                  # finds .env and fills os.environ

from anthropic import Anthropic
client = Anthropic()           # picks up ANTHROPIC_API_KEY automatically
```

Three Files, Three Jobs

File	Committed?	Contains
`.env`	**never**	the real secret values
`.env.example`	yes	same names, placeholder values
`.gitignore`	yes	a line reading `.env`

```
# .gitignore
.env
.env.*
!.env.example
```

Commit the **shape** of the configuration, never the **values**. Add the `.gitignore` line **before** you create `.env` — once git tracks a file, ignoring it later doesn't untrack it.

Several Keys in One `.env`

```
# .env  (copied from .env.example, then filled in)
LLM_PROVIDER=claude-sdk          # anthropic | claude-sdk | openai | gemini | ollama

ANTHROPIC_API_KEY=sk-ant-api03-...
ANTHROPIC_MODEL=claude-opus-5-5

OPENAI_API_KEY=sk-proj-...
OPENAI_MODEL=gpt-4.1-mini

GEMINI_API_KEY=AIza...
GEMINI_MODEL=gemini-3.8-flash

OLLAMA_API_BASE=http://localhost:11434   # local server: a URL, not a secret
OLLAMA_MODEL=llama3.2
```

Model names live in configuration too. Opus → Haiku, or Flash → Pro, is a config change, not a code change.

Who Wins: The Shell or `.env`?

```python
# apidemo/config.py
def load_env() -> str | None:
    path = find_dotenv(usecwd=True)
    if path:
        load_dotenv(path, override=False)  # never clobber real env vars
    return path or None

def require(name: str) -> str:
    value = os.environ.get(name, "").strip()
    if not value or "REPLACE-ME" in value:
        raise MissingKeyError(
            f"{name} is not set. Add it to your .env file "
            f"(see .env.example) or run:  export {name}=...")
    return value

def mask(secret: str | None) -> str:
    """Just enough of a key to identify it -- safe to log."""
    if not secret:
        return "<unset>"
    if len(secret) <= 12:
        return "*" * len(secret)
    return f"{secret[:7]}...{secret[-4:]}"
```

By default `load_dotenv()` does **not** override a variable that's already set.

That's the right rule: in production a secret manager injects the real key, and a stale `.env` accidentally copied onto the server must never replace it.

`require()` fails **loudly and early**, with a message that says how to fix it.

Demo 0: Check Your Environment DEMO

Shell exported `OPENAI_API_KEY`; `.env` has `ANTHROPIC_API_KEY` and a stale `OPENAI_API_KEY`:

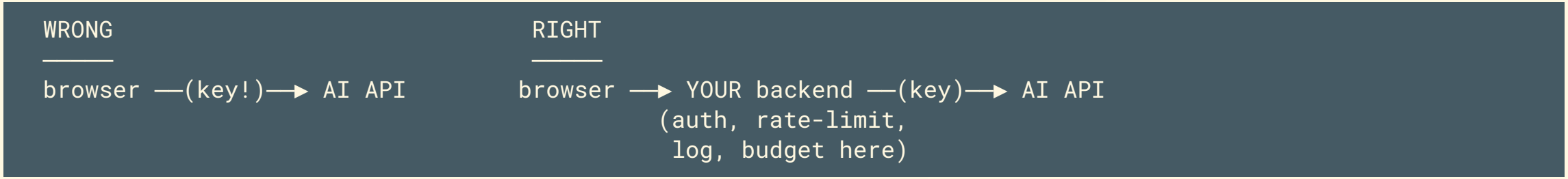
```
$ python demos/00_check_env.py
.env file: /Users/you/claude-api-demos/.env

ANTHROPIC_API_KEY    sk-ant-...x9Qw          from .env
OPENAI_API_KEY       sk-proj...1234          from shell     <- shell wins
GEMINI_API_KEY       <unset>                 from -
OLLAMA_API_BASE      http://localhost:11434
LLM_PROVIDER         claude-sdk
```

- `ANTHROPIC_API_KEY` was missing → `.env` filled it in
- `OPENAI_API_KEY` was already set → the file's value was ignored
- `GEMINI_API_KEY` is nowhere → any Gemini call fails with `MissingKeyError` **before** touching the network

Run this first whenever "my key doesn't work."

The Cardinal Rule: Keys Stay Server-Side



Never embed a key in anything you ship — browser JS, a mobile binary, a desktop app. All of it can be decompiled or read in the network tab. "Hidden" is not a thing on the client.

The proxy is also **where your product's controls live**: per-user auth, per-user rate limits, logging, abuse filtering, budget caps.

TerpGrader gets this for free (a server batch job). TerpTutor must route **every** chat message through its FastAPI backend.

Keys: Common Misconceptions

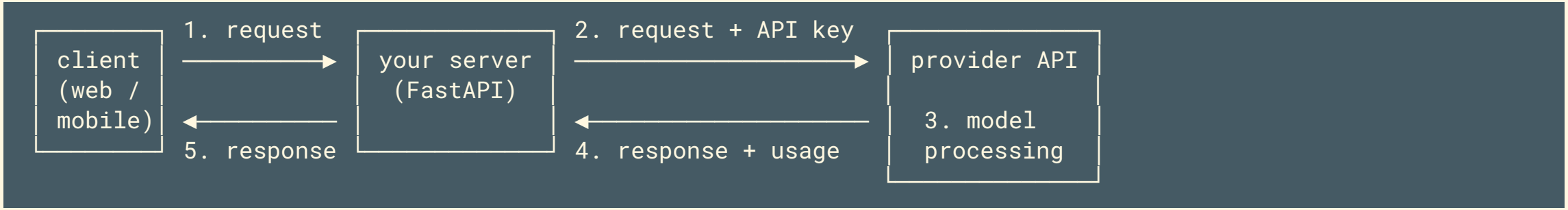
- **"Nobody will find my key."** GitHub is scraped continuously; a committed key is exploited **within minutes**. Providers auto-revoke leaked keys — the *best* case is that your job breaks.
- **"I'll delete the commit."** Git history is forever (and already scraped). **Leak = rotate.** No other response.
- **"The repo is private, so `config.py` is fine."** Private repos get forked, shared with TAs, made public, baked into Docker images — and rotating needs a code change.
- **"An env var is secure."** It's *appropriate*, not magic — anyone with shell access can read it. Production adds secret managers, scoping, rotation.
- **"`print(api_key)` is fine for debugging."** Logs get shipped to dashboards and pasted into Slack. Print `mask(key)`.

> **TerpGrader in production:** the key lives in Vault; the launcher injects `ANTHROPIC_API_KEY`. When a TA pasted it into Slack: rotate in the console (30 s), update Vault (one command), done — **nothing was hardcoded, so nothing else changed.**

Core Concept 2

The Request Lifecycle and Anatomy of a Call

The Five-Step Request Flow



Steps 1 and 5 are **your** API; steps 2 and 4 are the **provider's**. The key only ever travels on steps 2 and 4.

Inside Step 3: What the Model Does

1. **Tokenization** — text is split into tokens (words, pieces, spaces, punctuation). Billing and limits count these.
2. **Embedding** — each token becomes a vector of all its possible meanings. "Quantum" starts ambiguous.
3. **Contextualization** — transformer layers adjust each vector using its neighbors; next to "qubits," "quantum" settles on computing.
4. **Generation** — a probability for every possible next token; the model **samples** one, appends it, repeats.

After every token it checks stopping conditions — `stop_reason` says which fired:

Condition	`stop_reason`	For you
produced its end-of-turn token	`end_turn`	normal success
hit your `max_tokens` ceiling	`max_tokens`	**truncated** — maybe mid-JSON
generated one of your stop sequences	`stop_sequence`	intentional cut-off
safety systems declined	`refusal`	no usable answer; handle it

The Request, With Nothing Hidden

```
curl https://api.anthropic.com/v1/messages \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2026-10-21" \
  -H "content-type: application/json" \
  -d '{
    "model": "claude-haiku-5-5",
    "max_tokens": 1000,
    "system": "You are TerpGrader. Reply with JSON only: {\"score\": 0-10, \"feedback\": \"...\"}",
    "messages": [
      {"role": "user", "content": "RUBRIC: name the 3 steps of the TCP handshake.\nANSWER: SYN, then SYN-ACK, then ACK."}
    ]
  }'
```

Part	Purpose	Notes
`x-api-key`	authentication	who to bill and rate-limit — `$VAR`, never the literal
`anthropic-version`	API versioning	pins the wire format so changes don't silently break you
`model` / `max_tokens`	which model / output cap	a typo is a 404; the cap is a ceiling, not a target
`system` / `messages`	instructions / transcript	stateless: the full history every time

The Response

```
{
  "id": "msg_01XFDUDYJgAACzvnptvVoYEL",
  "type": "message",
  "role": "assistant",
  "model": "claude-opus-5-5",
  "content": [
    {"type": "thinking", "thinking": "",
     "signature": "EqQBCkgIBRAB..."},
    {"type": "text",
     "text": "{\"score\": 10, \"feedback\":
              \"All three steps, in order.\"}"}
  ],
  "stop_reason": "end_turn",
  "usage": {"input_tokens": 78,
            "output_tokens": 61}
}
```

`content` is a **list of blocks**, not a string. Current models may think first — a `thinking` block arrives **before** the text (text omitted by default, still billed). So: **never** `content[0].text`.

`stop_reason` — why it ended. For TerpGrader, `max_tokens` means truncated, unparseable JSON.

`usage` is the **billing ground truth**. Not your estimate, not a tokenizer library — this is what you pay for.

The First SDK Request DEMO 1

```python
# demos/01_first_request.py
from dotenv import load_dotenv
from anthropic import Anthropic

load_dotenv()           # copies .env into os.environ
client = Anthropic()    # finds ANTHROPIC_API_KEY
model = "claude-haiku-5-5"

message = client.messages.create(
    model=model,
    max_tokens=4000,    # a ceiling, not a target
    messages=[{"role": "user",
      "content": "What is quantum computing? "
                 "Answer in one sentence"}],
)

# content[0] may be a thinking block
text = "".join(b.text for b in message.content
               if b.type == "text")
print("ANSWER:", text)
print("stop_reason:", message.stop_reason)
print("usage:", message.usage.input_tokens, "in /",
      message.usage.output_tokens, "out")
```

```
ANSWER: Quantum computing is a type of
computation that uses superposition and
entanglement to process information in qubits...
stop_reason: end_turn
usage: 18 in / 64 out
```

1. `load_dotenv()` runs **before** `Anthropic()` — swap them and you get an auth error
2. Each message: a `role` (`user` / `assistant`) and `content`
3. `max_tokens` must cover **thinking + answer**, or the reply is truncated
4. `content[0].text` — works on older models (`claude-sonnet-4-5-20250929`, retiring Nov 30, 2026), not current ones

A Helper, and Three Levels of Abstraction

```python
# apidemo/chat.py
def text_of(message) -> str:
    """Join the text blocks of a response."""
    if message.stop_reason == "refusal":
        return "[the model declined this request]"
    return "".join(b.text for b in message.content if b.type == "text")
```

Level	Example	You get	You give up
raw HTTP	`curl`	total transparency; great for debugging	everything is manual
HTTP library	`httpx` (Demo 2)	testability, connection reuse, timeouts	you write retries, types, stream parsing
official SDK	`anthropic`	typed responses, auto-retry on 429/5xx, streaming helpers, typed exceptions	some magic you should understand anyway
multi-provider	LiteLLM	one call shape for many providers	provider-specific features, one more layer

Rule for real projects: use the official SDK (or LiteLLM when switching is a requirement). Demo 2 (`02_raw_http.py --break-key`, `--break-model`) shows raw 401 and 404 bodies so the SDK isn't magic.

Lifecycle: Common Misconceptions

Misconception	Reality
"`max_tokens` is a target length."	A hard cap. The model stops early when done, and is cut off (`stop_reason: max_tokens`) when not.
"The answer is `content[0].text`."	`content` is a list of blocks; the first may be thinking. Join the text blocks.
"I can count tokens with `len(text)/4`."	Fine for rough budgeting; wrong for billing. Use `usage` (after) or `count_tokens` (before).
"JSON in = JSON out."	The *envelope* is JSON; the generated text is whatever the model wrote. See Core Concept 7.

Core Concept 3

Multi-Turn Conversations

The API Is Stateless

> The provider stores **none** of your conversation. Each request is complete and independent; the model sees exactly the `messages` list in that request. A **multi-turn conversation** is something **your code** builds — keep every message, send the whole list every time.

Like talking to someone with no short-term memory who reads a transcript before every reply.

```
# demos/03_multi_turn.py, first half
chat([{"role": "user", "content":
  "Define quantum computing in one sentence"}])
chat([{"role": "user", "content":
  "Write another sentence"}])
```

```
Q1: Quantum computing is a paradigm
    of computation that uses qubits ...
Q2: The morning fog lifted slowly
    over the quiet harbor, ...
```

The second request had no context, so "another sentence" was about anything at all.

The Fix: Three Helpers

```python
def add_user_message(messages, text):
    messages.append({"role": "user",
                     "content": text})

def add_assistant_message(messages, text):
    messages.append({"role": "assistant",
                     "content": text})

def chat(messages):
    message = client.messages.create(
        model=model,
        max_tokens=4000,
        messages=messages,
    )
    return text_of(message)
```

```python
messages = []
add_user_message(messages,
    "Define quantum computing in one sentence")
answer = chat(messages)
add_assistant_message(messages, answer)

add_user_message(messages, "Write another sentence")
final_answer = chat(messages)   # full history sent
add_assistant_message(messages, final_answer)
```

```
call 1: [ user: "Define quantum computing..." ]

call 2: [ user:      "Define quantum computing...",
          assistant: "Quantum computing is ...",
          user:      "Write another sentence" ]
                     <- context included
```

Rules: the first message is `user`; roles normally alternate `user`, `assistant`, `user`, …

A Terminal Chatbot DEMO 4

```python
# demos/04_chat_loop.py (core loop)
messages = []
while True:
    user_input = input("\nyou> ").strip()
    if user_input == "/quit":
        break
    if user_input == "/reset":
        messages.clear()   # "forgotten" everything
        continue

    add_user_message(messages, user_input)
    response = client.messages.create(
        model=MODEL, max_tokens=4000,
        messages=messages)
    reply = text_of(response)
    add_assistant_message(messages, reply)

    print(f"claude> {reply}")
    print(f"  (turn {len(messages) // 2}: "
          f"{response.usage.input_tokens} input tokens)")
```

```
you> My name is Testudo and I am taking CMSC389A.
claude> Nice to meet you, Testudo! ...
  (turn 1: 21 input tokens)
you> What is my name and which class am I in?
claude> Your name is Testudo, and you're
        taking CMSC389A.
  (turn 2: 96 input tokens)
you> /reset
you> What is my name?
claude> I don't know your name -- you haven't
        told me yet.
  (turn 1: 12 input tokens)
```

Watch the input tokens grow. The whole history is resent — and **billed** — every turn. A 50-turn chat pays for turn 1 fifty times.

Multi-Turn: Common Misconceptions

- **"The API remembers my session."** There is no session. Clear your list and the model has forgotten everything — exactly what `/reset` shows.
- **"Storing history is the provider's job."** It's **yours** — usually a database keyed by conversation ID. Your server reloads it on each request.
- **"I can edit the history freely."** You can, and it's powerful (summarize old turns, remove a bad answer) — but the model then believes whatever the edited transcript says. **Never let users write `assistant` messages directly.**

Core Concept 4

System Prompts

What a System Prompt Is

> A **system prompt** is a string in the separate `system` field, outside `messages`. It tells the model what role to play and how to respond for the whole conversation.

The `messages` are the conversation; the system prompt is the **job description** handed to the employee before it starts.

```
def chat(messages, system=None):
    params = {"model": model, "max_tokens": 4000, "messages": messages}
    if system:                          # the API rejects system=None,
        params["system"] = system       # so only include it when it is set
    message = client.messages.create(**params)
    return text_of(message)
```

The `params` + `**params` pattern matters: optional fields must be **left out**, not set to `None`. The same pattern handles `stop_sequences` later.

Turning Claude into TerpTutor DEMO 5

```
system_prompt = """
You are a patient math tutor. Do not directly answer the user's question.
Instead, ask follow-up questions to help the user solve the problem.
"""
messages = []
add_user_message(messages, "I need to solve the equation 2x + 3 = 11. Can you help me?")
print(chat(messages))                          # no system prompt
print(chat(messages, system=system_prompt))  # TerpTutor
```

```
--- no system prompt ---
Sure! Subtract 3 from both sides: 2x = 8. Then divide both sides by 2: x = 4.

--- with the tutor system prompt ---
Of course! Let's work through it together. Right now x has two things
attached to it: it's multiplied by 2, and 3 is added. Which of those
do you think we should undo first, and what operation would undo it?
```

Same model, same question, completely different behavior.

A Code-Only Assistant — and Its Limits

```
system_prompt = """
You are a code assistant. Return the code
content only. Don't include any other text.
"""
messages = []
add_user_message(messages,
  "Write a Python function that checks a "
  "string for duplicate characters.")
print(chat(messages, system=system_prompt))
```

```
def has_duplicate_characters(s):
    """Check if a string contains duplicates."""
    return len(s) != len(set(s))
```

Often the model **still** wraps it in a markdown fence.

> A system prompt **strongly encourages** a format; it does not **guarantee** one. (Core Concept 7 shows how to guarantee it.)

- **Not a security boundary** — users can try to talk the model out of it (prompt injection). Enforce real rules in code.
- **Use the `system` field**, not the first user message — it stays fixed as the conversation grows.
- **It's resent and billed every request** — a long, stable system prompt is the classic **prompt-caching** candidate.

Core Concept 5

Temperature

Temperature Reshapes the Distribution

For each position the model computes a probability for every next token, then **samples**. **Temperature** (typically 0–1) reshapes that distribution first.

- **Low (near 0)** — sharpens it: the most likely token almost always wins. Predictable, repeatable.
- **High (near 1)** — flattens it: less likely tokens get picked more. Varied, creative.

After *"What do you think"*:

```
                 about   would    of    (others)
temperature 0.0  100%      0%     0%       0%      always "about"
temperature 0.5   ~55%   ~25%    ~7%     ~13%      usually "about"
temperature 1.0    30%    20%    10%      40%      the model's raw odds
```

Repeat for every token and the differences compound: at 0.0, nearly the same sentence each run; at 1.0, a different story every time. It changes **probabilities**, not guarantees.

Choosing a Temperature

Range	Good for	Running example
0.0 – 0.3 (low)	facts, coding, data extraction, moderation	TerpGrader: same answer → same grade
0.4 – 0.7 (medium)	summarization, educational content, problem-solving	TerpTutor explanations
0.8 – 1.0 (high)	brainstorming, creative writing, marketing, jokes	practice-problem story contexts

```
def chat(messages, system=None, temperature=1.0):
    params = {"model": model, "max_tokens": 1000,
              "messages": messages, "temperature": temperature}
    if system:
        params["system"] = system
    message = client.messages.create(**params)
    return message.content[0].text

add_user_message(messages, "Generate a one sentence story about a cat.")
for t in (0.0, 0.5, 1.0):
    print(f"temperature={t}:", chat(messages, temperature=t))
```

Current Claude Models No Longer Take a Temperature

> The API has changed — a good lesson in **why the `anthropic-version` header exists**.

- Current Claude models (Opus 5.5, Sonnet 5.5, Haiku 5.5, ...) reject non-default sampling settings, and API version `2026-10-21` removes `temperature`, `top_p`, and `top_k` from the Messages API on **every** model → a request that sends them gets a `400`.
- The current Python SDK sends that version by default, and `messages.create()` no longer has a `temperature` parameter at all.

Controlling variety on Claude now:

- through the **prompt** — "give five *different* ideas," "answer in exactly this format"
- through **structured outputs** when you need consistency (Core Concept 7)
- by **generating several candidates** and choosing when you need variety

The Experiment on a Provider That Still Has the Knob DEMO 6

OpenAI, Gemini, and every Ollama model still expose temperature.

```
python demos/06_temperature.py --provider ollama
python demos/06_temperature.py --provider openai
```

```
prompt = "Generate a one sentence story about a cat."
for temperature in (0.0, 0.5, 1.0):
    for i in range(args.samples):
        reply = ask(prompt, provider=args.provider,
                    temperature=temperature,
                    max_tokens=100)
        print(f"  {i+1}. {reply.text.strip()}")
```

```
=== temperature=0.0 ===
  1. The curious cat chased a sunbeam across
     the floor until it fell asleep in its warmth.
  2. The curious cat chased a sunbeam across
     the floor until it fell asleep in its warmth.
  3. The curious cat chased a sunbeam across
     the floor until it fell asleep in its warmth.

=== temperature=1.0 ===
  1. Whiskers stole the last sardine, then blamed
     the dog with a perfectly innocent blink.
  2. Every night at 3 a.m., Mochi sang opera to
     the refrigerator, hoping it would sing back.
  3. The old tabby guarded the library's rare-book
     room better than any alarm ever had.
```

Pass `--temperature` to a Claude provider and the switch prints a note and drops it.

Temperature: Common Misconceptions

- **"Temperature 0 means deterministic."** Nearly, not perfectly: ties and floating-point differences can still change output. Never rely on bit-for-bit repeatability.
- **"Higher temperature = smarter, more creative ideas."** It means **less likely** tokens. Past a point that's just incoherence.
- **"Temperature is a quality knob."** It's a **variety** knob. Use the prompt, the model choice, and evaluation for quality.

Core Concept 6

Streaming Responses

Why Stream?

Streaming returns the response incrementally over one long-lived HTTP response using **server-sent events (SSE)**: a `text/event-stream` body of `event: <type>` + `data: <json>` lines, delivering the answer as it's generated.

A TerpTutor reply can take 10–30 s. Non-streaming is a **letter**: nothing, nothing, nothing, then everything. Streaming is a **phone call**: words as they're produced.

- **Time to first token (TTFT)** — until *something* appears (~1 s when streamed)
- **Total time** — until the answer is complete (identical either way)

Streaming doesn't make the model faster; it makes the **experience** faster. And for long outputs it's **not optional** — a multi-minute response blows through HTTP timeouts unless streamed.

The Events — All in One Request

Event	Meaning
`message_start`	message begun; **input-token usage**
`content_block_start`	a block (thinking, text, tool use) begins
`content_block_delta`	a chunk — the text you display
`content_block_stop`	the block is complete
`message_delta`	`stop_reason` and **output-token usage**
`message_stop`	nothing more

```
event: message_start
data: {"type":"message_start","message":
       {"usage":{"input_tokens":21,...}}}

event: content_block_delta
data: {"type":"content_block_delta","index":1,
       "delta":{"type":"text_delta",
                "text":"Rex spotted"}}

event: content_block_delta
data: {... "text":" the cat"}}

event: message_delta
data: {"type":"message_delta",
       "delta":{"stop_reason":"end_turn"},
       "usage":{"output_tokens":212}}

event: message_stop
data: {"type":"message_stop"}
```

Bookkeeping lives at **both ends**: input tokens in `message_start`, output tokens and `stop_reason` in the final `message_delta`. A cost meter must read both.

Raw Events vs. the Streaming Helper DEMO 7

Raw — add `stream=True`:

```
stream = client.messages.create(
    model=model,
    max_tokens=4000,
    messages=messages,
    stream=True,
)
for event in stream:
    print(event)
```

`python demos/07_streaming.py events` — one line per event; only the text deltas are for the student.

Helper — just the text, plus the final message:

```
with client.messages.stream(
    model=model,
    max_tokens=4000,
    messages=messages,
) as stream:
    for text in stream.text_stream:
        print(text, end="", flush=True)

    final_message = stream.get_final_message()
```

- `flush=True` — or the terminal buffers and it all appears at once
- `get_final_message()` — a normal message with `content`, `stop_reason`, `usage`, for the database

Seeing the Difference

```
$ python demos/07_streaming.py race
non-streaming: first text after 8.4s (all 1093 chars at once)
streaming:     first text after 1.1s, finished after 8.2s
```

Same total time, about eight times faster to first output.

- **"Streaming is cheaper."** Same tokens, same price — it changes *when* you see them.
- **"Streaming is just for looks."** Also for correctness: long generations need it to avoid timeouts, which is why SDKs insist on streaming for very large `max_tokens`.
- **"I can reassemble the message from the deltas I printed."** Use `get_final_message()` — dropped events and non-text blocks make naive reassembly buggy.
- **"Errors arrive as events."** A `429` or `401` arrives **before** any events, as a normal error response; the SDK raises it when you open the stream.

Core Concept 7

Controlling Output Format

When a Program Reads the Output

Ask for JSON and you typically get:

````
```json
{
  "source": ["aws.ec2"],
  "detail-type": ["EC2 Instance State-change Notification"],
  "detail": { "state": ["running"] }
}
```
This rule captures EC2 instance state changes when instances start running.
````

The JSON is correct — but `json.loads` fails on the fence and the explanation, and a user of an "EventBridge rule generator" can't just copy the response. TerpGrader has the same problem with its verdicts.

Three tools, oldest to newest: stop sequences → prefill + stop → structured outputs.

Tools 1 & 2: Stop Sequences and Prefill

```python
def chat(messages, system=None, stop_sequences=None):
    params = {"model": model, "max_tokens": 4000,
              "messages": messages}
    if system:
        params["system"] = system
    if stop_sequences:
        params["stop_sequences"] = stop_sequences
    return text_of(client.messages.create(**params))

messages = []
add_user_message(messages,
  "Generate a very short event bridge rule as json")
add_assistant_message(messages, "```json")  # prefill

text = chat(messages, stop_sequences=["```"])
clean_json = json.loads(text.strip())
```

Stop sequences — generation ends the moment the model produces one; `stop_reason` becomes `stop_sequence`; the stop string isn't included.

Prefill — the *last* message is an `assistant` message you wrote; the model continues it.

1. The user message says what to generate
2. The prefill makes the model believe it already opened a code block → no preamble
3. It writes only the JSON
4. When it tries to close the block, the stop sequence ends generation → no fence, no explanation

The General Recipe — and the Catch

```
prompt = """
Generate three different sample AWS CLI
commands. Each should be very short.
"""
messages = []
add_user_message(messages, prompt)
add_assistant_message(messages,
  "Here are all three commands in a single "
  "block without any comments: ```bash")
text = chat(messages, stop_sequences=["```"])
text.strip()
```

```
aws s3 ls
aws ec2 describe-instances
aws iam list-users
```

Recipe: find what the model naturally wraps your content in (a fence, a list marker, a closing tag) — **prefill the opening, stop on the closing**.

> **Prefill is gone on current models.** A request whose last message is `assistant` returns a `400` on current Claude models (Opus 4.6+, Sonnet 4.6+, Haiku 5.5). Older models like `claude-haiku-4-5` still accept it — Demo 8 Part B uses one; Part B3 sends it to the current default so you see the error.

Stop sequences **on their own** still work everywhere.

Tool 3: Structured Outputs (Current Models)

Constrain the response to a **JSON schema**, validated for you. Describe the shape as a Pydantic model and call `messages.parse`:

```python
from pydantic import BaseModel

class EventPattern(BaseModel):
    source: list[str]
    detail_type: list[str]
    states: list[str]

response = client.messages.parse(
    model="claude-opus-5-5",
    max_tokens=4000,
    messages=[{"role": "user",
               "content": "Generate a very short EventBridge pattern that matches EC2 "
                          "instances entering the running state."}],
    output_format=EventPattern,
)
rule = response.parsed_output        # a validated EventPattern object
print(rule.source, rule.states)      # ['aws.ec2'] ['running']
```

No Pydantic? Pass `output_config={"format": {"type": "json_schema", "schema": ...}}` to `messages.create`.
TerpGrader: a `Verdict` model → **all 500 replies parse** (a truncated reply is still truncated).

Comparing the Tools

Technique	Works on	Guarantee	Use for
system prompt ("JSON only")	all models	none — usually works	quick experiments
stop sequences	all models	cuts output at a marker	ending lists, removing trailing text
prefill + stop sequence	**older models only**	strong, not validated	legacy code
structured outputs	**current models**	**valid against your schema**	anything a program parses

Core Concept 8

Errors and Rate Limits

The Error Taxonomy: Will Trying Again Help?

LLM APIs signal failures with HTTP status codes plus a JSON error body. A **rate limit** is a per-organization cap measured several ways at once — requests (**RPM**), input tokens (**ITPM**), output tokens (**OTPM**) per minute — signaled by `429` with a `retry-after` header.

Code	Meaning	Retryable?	Correct reaction
400	malformed request	no	fix your code (bad body, prefill on a current model, unsupported param)
401	bad/missing key	no	fix your configuration (run Demo 0)
403	key lacks permission	no	fix your account/scoping
404	bad model ID / endpoint	no	fix the model string
413	request too large	no	shrink the input (trim history, chunk docs)
429	rate limited	**YES**	wait (`retry-after`), then retry
500	provider bug	**YES**	retry with backoff
529	provider overloaded	**YES**	retry with backoff, consider shedding load

Retrying a `400` burns time and money forever; failing permanently on a `429` makes you fragile for no reason. **The taxonomy is the design.** And check `stop_reason` — `max_tokens` and `refusal` both arrive as a successful `200`.

Breaking Things on Purpose DEMO 9

```python
try:
    msg = client.messages.create(
        model=model, max_tokens=max_tokens,
        messages=msgs)
    print(f"stop_reason={msg.stop_reason!r}  "
          f"text={text_of(msg)!r}")
except anthropic.AuthenticationError as err:  # 401
    print("fix the key.  retryable:",
          is_retryable(err))
except anthropic.NotFoundError as err:        # 404
    print("fix the model id.  retryable:",
          is_retryable(err))
except anthropic.RateLimitError as err:       # 429
    print("wait and retry.  retryable:",
          is_retryable(err))
except anthropic.APIStatusError as err:       # other
    print(f"status {err.status_code}.  retryable:",
          is_retryable(err))
```

The SDK raises a **typed exception** per status code. Catch **most-specific first**.

```
=== bad key (401) ===
AuthenticationError -- fix the key.
  retryable: False

=== bad model (404) ===
NotFoundError -- fix the model id.
  retryable: False

=== max_tokens=5 ===
stop_reason='max_tokens'  text=''
                    <- HTTP 200, but useless
```

The third case is the subtle one: no exception, a successful response, no usable answer.

Rate Limits: Intuition and Arithmetic

Rate limits aren't punishment; they're **capacity allocation**. Say your tier allows **50 RPM** and **40,000 ITPM**.

> TerpGrader: 500 submissions × ~900 input tokens = **450,000 input tokens**.
> At 40,000 ITPM the job takes **at least ~12 minutes — no matter how parallel your code is.**

Beyond a point, parallelism doesn't buy throughput; your **token budget per minute** does. A well-behaved client:

- **paces itself** near the limit instead of slamming into it
- treats `429` as **normal weather**, not an emergency

Exponential Backoff with Jitter

- **Exponential** (1 s, 2 s, 4 s, 8 s…) — a stressed server needs *rapidly decreasing* pressure, not a fixed drumbeat
- **Jitter** (randomize each delay) — if 500 requests fail at once and all retry after exactly 2 s, they stampede and fail again (**thundering herd**)
- **Honor** `retry-after` — when the server says how long, obey it
- **Give up eventually** — infinite retries hide outages

```
attempt 1 ──▶ 429 (retry-after: 7)
              sleep 7.0s   (server said so)
attempt 2 ──▶ 429 (no header)
              sleep 2 × (0.5 + r) ≈ 1.6s
                           (jittered)
attempt 3 ──▶ 200 ✓   return response
```

A `401` on attempt 1 raises **immediately** — no amount of waiting fixes a bad key.

`with_retries` (`apidemo/retry.py`)

```python
RETRYABLE_STATUS = {408, 409, 429, 500, 502, 503, 504, 529}

def is_retryable(err: Exception) -> bool:
    if isinstance(err, anthropic.APIConnectionError):    # includes timeouts
        return True
    if isinstance(err, anthropic.APIStatusError):
        return err.status_code in RETRYABLE_STATUS
    return False

def with_retries(fn, *, max_attempts=4, base_delay=1.0, max_delay=30.0,
                 is_retryable=is_retryable, retry_after=retry_after_seconds,
                 sleep=time.sleep, rng=random.random, on_retry=None):
    for attempt in range(1, max_attempts + 1):
        try:
            return fn()
        except Exception as err:
            if not is_retryable(err) or attempt == max_attempts:
                raise                                    # fail fast / give up
            delay = retry_after(err)                     # server said so...
            if delay is None:                            # ...else backoff + jitter
                delay = min(max_delay, base_delay * 2 ** (attempt - 1)) * (0.5 + rng())
            if on_retry:
                on_retry(attempt, delay, err)
            sleep(delay)
```

Injected `sleep` / `rng` make tests instant. The SDK already retries 2× (`Anthropic(max_retries=...)`); Demo 9 uses `max_retries=0`.

Errors: Common Misconceptions

- **"429 means I'm being blocked."** It means *slow down for a minute* — routine at any real scale.
- **"Catch `Exception` and retry everything."** Retrying non-retryable errors is the classic infinite loop that costs money. Catch **typed** errors and consult the taxonomy.
- **"More threads = faster batch job."** Only until ITPM saturates. Then more threads just means more 429s.

Core Concept 9

Cost Management

How You're Billed

> Per token, with **separate input and output prices**, in dollars per million tokens (**MTok**). Cost = `input_tokens × in_price + output_tokens × out_price` — plus discounted **cache reads**, a premium for **cache writes** (~1.25× input for the 5-minute cache), and typically **50% off everything** via the batch API.

Model	Model ID	Input $/MTok	Output $/MTok	Character
Claude Opus 5.5	`claude-opus-5-5`	$4.00	$20.00	flagship quality
Claude Sonnet 5.5	`claude-sonnet-5-5`	$2.00	$10.00	balanced
Claude Haiku 5.5	`claude-haiku-5-5`	$0.10	$0.50	fast + cheap (prompts ≤ 100K tokens)

Reference prices, October 2026 — check the provider's pricing page; these change.

Tokens are the **taxi meter**. Input is cheaper and **you control it** (prompt, history, system prompt); output costs **5×** and you only cap it. **Thinking is billed as output.**

Pricing a TerpGrader Run

500 submissions × (900 in + 150 out) → **0.45 MTok in**, **0.075 MTok out**

Model	Input	Output	Per run	Semester (10 runs)
Opus 5.5	0.45 × $4 = $1.80	0.075 × $20 = $1.50	**$3.30**	~$33
Sonnet 5.5	0.45 × $2 = $0.90	0.075 × $10 = $0.75	**$1.65**	~$16.50
Haiku 5.5	0.45 × $0.10 = $0.045	0.075 × $0.50 = $0.04	**$0.08**	~$0.83

1. **Model choice is the biggest lever** — 40× between flagship and small. Ask: what's the *cheapest model that passes our quality bar*? Answer it with **evaluation** (grade 50 with each, compare to TAs), not vibes.
2. **Small because the prompt is small.** Paste a 20-page lab report instead and input jumps ~50× — **prompt design *is* cost design**.
3. **The rubric is identical in all 500 → prompt caching**: Opus re-reads it at $0.20 instead of $4.00 per MTok.

The Other Levers

- `max_tokens` — caps worst-case output spend
- **History trimming** — resent history is billed input, every turn (Demo 4 shows it growing)
- **Batch API** — TerpGrader is the perfect customer: submit all 500 asynchronously, results within hours, **50% off everything**
- **A local model** — for development and testing, Ollama costs nothing per token (Core Concept 10)

> The receipt (`usage`) arrives with every response. A system that doesn't read it doesn't know what it spends.

A Cost Meter with a Budget Guard (`apidemo/cost.py`)

```python
# model: (input $/MTok, output $/MTok, cache-read $/MTok)
PRICING_PER_MTOK = {
    "claude-opus-5-5":   (4.00, 20.00, 0.20),
    "claude-sonnet-5-5": (2.00, 10.00, 0.20),
    "claude-haiku-5-5":  (0.10, 0.50, 0.01),
}
CACHE_WRITE_MULTIPLIER = 1.25                  # 5-minute cache writes
def cost_of(model: str, usage) -> float:
    in_price, out_price, cache_read_price = PRICING_PER_MTOK[model]
    per = 1_000_000
    cost  = _get(usage, "input_tokens") / per * in_price
    cost += _get(usage, "output_tokens") / per * out_price
    cost += _get(usage, "cache_read_input_tokens") / per * cache_read_price
    cost += _get(usage, "cache_creation_input_tokens") / per * in_price * CACHE_WRITE_MULTIPLIER
    return cost

class CostMeter:                               # raises instead of overspending
    def __init__(self, budget_usd: float | None = None):
        self.budget_usd, self.total_usd, self.calls = budget_usd, 0.0, 0

    def record(self, model: str, usage) -> float:
        cost = cost_of(model, usage)
        if self.budget_usd is not None and self.total_usd + cost > self.budget_usd:
            raise BudgetExceeded(
                f"${self.total_usd + cost:.4f} would exceed budget ${self.budget_usd:.4f}")
        self.total_usd += cost
        self.calls += 1
        return cost
```

TerpGrader with Retries and the Meter DEMO 10

```
meter = CostMeter(budget_usd=args.budget)
try:
    for i, answer in enumerate(submissions, 1):
        msg = with_retries(lambda: client.messages.create(
            model=MODEL, max_tokens=2000, system=RUBRIC,
            messages=[{"role": "user", "content": f"ANSWER: {answer}"}]))
        cost = meter.record(MODEL, msg.usage)           # raises before overspending
        verdict = json.loads(text_of(msg))
        print(f"#{i}: score={verdict['score']:>2}  ${cost:.5f}  {verdict['feedback']}")
except BudgetExceeded as err:
    print(f"STOPPED: {err}")
```

```
$ python demos/10_cost_meter.py
#1: score=10  $0.00290  Correct steps in the correct order.
...
graded 5 answers for $0.01484 on claude-opus-5-5
projected 500-submission run: $3.30

$ python demos/10_cost_meter.py --budget 0.005
#1: score=10  $0.00290  Correct steps in the correct order.
STOPPED: $0.0063 would exceed budget $0.0050
```

Budget guards turn billing surprises into exceptions. (It records *after* a call, so it stops the *next* one.)

Cost: Common Misconceptions

- **"Input and output cost the same."** Output is 5× input. Verbose models and prompts are expensive on the wrong side.
- **"Costs are too small to matter."** Per request, yes. Per product they compound: 10k users × 20 turns/day × growing history is how startups get five-digit surprise invoices.
- **"The cheapest model is always right / the best model is always right."** Both wrong — the right model is **the cheapest one that passes your measured quality bar**, per task.

Core Concept 10

One Interface, Many Providers with LiteLLM

Why Switch Providers?

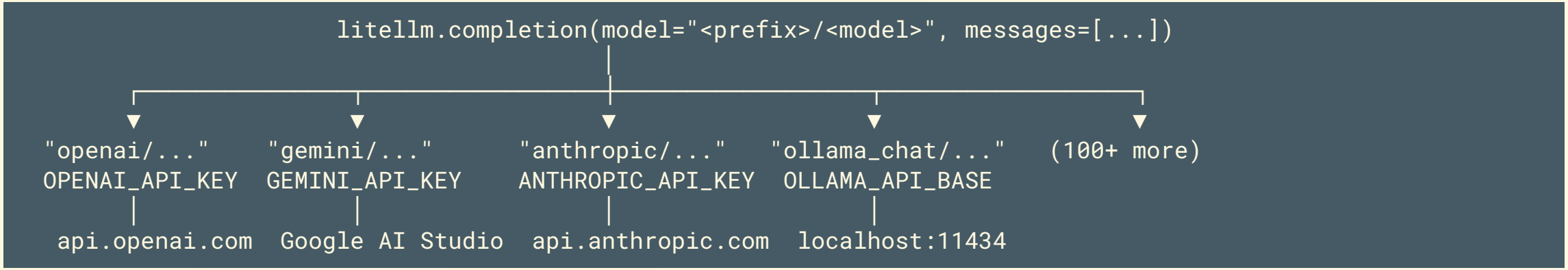
So far every line has been Claude-specific. Real teams rarely stay with one provider:

- **Cost and quality** — "cheapest model that passes the bar" should include other providers' models
- **Local development** — run TerpTutor against a free local Ollama model while debugging the UI; works offline
- **Privacy** — some data (student records under FERPA) may only be processed by a local model
- **Resilience** — when one provider has an outage, fall back to another

> Each provider has its own SDK, message format, and response shape. `if provider == ...` branches through your whole app don't scale.

LiteLLM: A Universal Power Adapter

LiteLLM exposes one function, `litellm.completion()`, in the OpenAI chat-completions format, and translates each call to the native API of **100+ providers**. Pick the provider with a **prefix on the model string**; it reads that provider's key from its standard env var; every response comes back in the **same OpenAI shape**.



Your appliance (your app) has one plug; the adapter has a socket for every country.

The Minimal Example

```python
# pip install litellm python-dotenv
from dotenv import load_dotenv
import litellm

load_dotenv()                                   # all keys from one .env

messages = [{"role": "system", "content": "Answer in one sentence."},
            {"role": "user",   "content": "What is the TCP handshake?"}]

for model in ["openai/gpt-4.1-mini",
              "gemini/gemini-3.8-flash",
              "anthropic/claude-opus-5-5",
              "ollama_chat/llama3.2"]:
    kwargs = {"api_base": "http://localhost:11434"} if model.startswith("ollama") else {}
    response = litellm.completion(model=model, messages=messages, max_tokens=2000, **kwargs)
    print(model, "->", response.choices[0].message.content)
    print("   tokens:", response.usage.prompt_tokens, "in /",
          response.usage.completion_tokens, "out")
```

Only the model string changes.

What LiteLLM Translated for You

Concept	Anthropic SDK	LiteLLM / OpenAI format
system prompt	`system="..."`	`{"role": "system", ...}` message
reply text	join text blocks of `message.content`	`response.choices[0].message.content`
token usage	`usage.input_tokens` / `output_tokens`	`usage.prompt_tokens` / `completion_tokens`
why it stopped	`stop_reason`	`choices[0].finish_reason`
streaming text	`stream.text_stream`	`chunk.choices[0].delta.content`
cost	compute from `usage` yourself	`litellm.completion_cost(completion_response=r)`

The Providers in the Demo Project (`apidemo/llm.py`)

`--provider`	Goes through	Model string sent	Needs
`openai`	LiteLLM	`openai/$OPENAI_MODEL`	`OPENAI_API_KEY`
`gemini`	LiteLLM	`gemini/$GEMINI_MODEL`	`GEMINI_API_KEY`
`anthropic`	LiteLLM	`anthropic/$ANTHROPIC_MODEL`	`ANTHROPIC_API_KEY`
`claude-sdk`	official `anthropic` SDK, no translation	`$ANTHROPIC_MODEL`	`ANTHROPIC_API_KEY`
`ollama`	LiteLLM to a local server	`ollama_chat/$OLLAMA_MODEL`	no key; `OLLAMA_API_BASE`
`claude-agent`	Claude Agent SDK (optional)	`$ANTHROPIC_MODEL`	`claude-agent-sdk` + Claude Code CLI

Why both `anthropic` and `claude-sdk`? Same model, different roads. `anthropic` shows Claude is just one more plug. `claude-sdk` gives Claude-specific features the moment they ship — `messages.parse`, caching controls, typed exceptions, streaming helpers. **A translation layer always lags the native API a little.**

`claude-agent` packages the Claude Code agent loop as a library — it can read files and run tools, so the switch disables tools and limits it to one turn.

The Switch, Step by Step (1/3)

First, **a table describes each provider**: which variable names its model, its LiteLLM prefix, which key it needs, and whether it accepts temperature.

```python
# apidemo/llm.py
@dataclass(frozen=True)
class Provider:
    name: str
    model_env: str              # env var naming the model
    default_model: str
    litellm_prefix: str | None  # None => not routed through LiteLLM
    key_env: str | None         # None => no key required
    sampling: bool = True       # does it accept temperature?

PROVIDERS = {
    "openai":     Provider("openai", "OPENAI_MODEL", "gpt-4.1-mini", "openai", "OPENAI_API_KEY"),
    "gemini":     Provider("gemini", "GEMINI_MODEL", "gemini-3.8-flash", "gemini", "GEMINI_API_KEY"),
    # Current Claude models reject temperature/top_p/top_k, so sampling=False.
    "anthropic":  Provider("anthropic", "ANTHROPIC_MODEL", "claude-opus-5-5", "anthropic",
                           "ANTHROPIC_API_KEY", sampling=False),
    "claude-sdk": Provider("claude-sdk", "ANTHROPIC_MODEL", "claude-opus-5-5", None,
                           "ANTHROPIC_API_KEY", sampling=False),
    "ollama":     Provider("ollama", "OLLAMA_MODEL", "llama3.2", "ollama_chat", None),
    # ... claude-agent
}
```

The Switch, Step by Step (2/3)

Second, **one function resolves the flag** — `--provider`, then `LLM_PROVIDER` from the environment or `.env`, then a default — and checks the key **before** any network call:

```
def resolve(provider: str | None = None) -> tuple[Provider, str]:
    name = (provider or os.environ.get("LLM_PROVIDER") or "claude-sdk").lower()
    if name not in PROVIDERS:
        raise ValueError(f"unknown provider {name!r}; choose from {', '.join(PROVIDERS)}")
    p = PROVIDERS[name]
    model = os.environ.get(p.model_env) or p.default_model
    if p.key_env:
        require(p.key_env)                    # fail fast, before any network call
    return p, model
```

`LLM_PROVIDER=gemini python -m apidemo.llm "..."` and `--provider gemini` do the same thing; the flag wins if both are set.

The Switch, Step by Step (3/3)

```python
def ask(messages, *, provider=None, max_tokens=4000, temperature=None, **extra) -> Reply:
    p, model = resolve(provider)
    temperature = _check_sampling(p, temperature)     # drop it for Claude, with a note
    start = time.perf_counter()

    if p.name == "claude-sdk":                        # native SDK path
        system, rest = _split_system(messages)        # system message -> system field
        params = {"model": model, "max_tokens": max_tokens, "messages": rest}
        if system:
            params["system"] = system
        msg = _anthropic_client().messages.create(**params)
        return Reply(p.name, model, text_of(msg), msg.usage.input_tokens,
                     msg.usage.output_tokens, cost_of(model, msg.usage),
                     time.perf_counter() - start)

    if p.name == "claude-agent":                      # optional Agent SDK path
        return _ask_agent(messages, model, start)
    import litellm                                    # everything else: LiteLLM
    resp = litellm.completion(**_litellm_kwargs(p, model, messages, max_tokens,
                                                temperature, extra))
    cost = _safe_cost(resp)                           # None for local/unknown models
    return Reply(p.name, model, resp.choices[0].message.content or "",
                 resp.usage.prompt_tokens, resp.usage.completion_tokens, cost,
                 time.perf_counter() - start)
```

Third, `ask()` returns one `Reply` (text, tokens, cost, seconds) **no matter who answered**.

Running It DEMO 11

```
python -m apidemo.llm --provider openai     "Explain TCP in one sentence"
python -m apidemo.llm --provider claude-sdk "Explain TCP in one sentence"
python -m apidemo.llm --provider ollama     "Explain TCP in one sentence" --stream
LLM_PROVIDER=gemini python -m apidemo.llm   "Explain TCP in one sentence"   # env instead of flag
python -m apidemo.llm --all                 "Explain TCP in one sentence"   # every provider
```

```
$ python demos/11_switch_providers.py --compare
provider      model                 secs    in   out       cost
openai        gpt-4.1-mini           1.2    41    27  $0.000060
   -> What could you do to both sides of the equation to get the 5x term by itself?
gemini        gemini-2.5-flash       1.9    38    24  $0.000072
   -> Good question! What operation would remove the "+ 2" from the left side?
anthropic     claude-opus-5-5        3.1    44    96  $0.002096
   -> Let's start by isolating the term with x. What could you do to both sides to ...
claude-sdk    claude-opus-5-5        2.9    44    91  $0.001996
   -> What's the first step you'd take to get the 5x term alone on one side?
ollama        llama3.2               0.8    52    31        n/a
   -> What do you think we should do first to get x by itself?
```

Read it like an engineer: every provider followed the system prompt; the local model was free and fastest; the flagship cost most and used the most output (thinking is billed as output). Whether it's *better enough* is an **evaluation** question. The transcript is plain dicts, so a conversation can even **switch providers midway**.

Local Models with Ollama

```
ollama serve                  # in one terminal (the desktop app does this for you)
ollama pull llama3.2          # download a small model once
python -m apidemo.llm --provider ollama "Explain TCP in one sentence"
```

> No key to leak, no per-token bill — but **the URL becomes the thing to protect**. An Ollama port exposed on a public network is an unauthenticated model anyone can use. Keep it on `localhost` or behind your backend, just like an API key.

LiteLLM: Common Misconceptions

- **"LiteLLM makes models interchangeable."** It makes the **call** interchangeable. The same prompt can behave differently on each model — re-evaluate whenever you switch.
- **"Every parameter works everywhere."** No parity guarantee. Temperature is the example: OpenAI, Gemini, Ollama take it; current Claude rejects it (hence `sampling=False`). Structured outputs, caching, and tool use differ too.
- **"One `.env` means one secret."** It now holds several — the `.gitignore` rule matters **more**, not less.
- **"The cost column is exact."** LiteLLM's price table can lag the providers'. `usage` token counts are ground truth; prices are an estimate.
- **Practical:** LiteLLM also ships a **proxy server** — one OpenAI-compatible endpoint with per-team keys and budgets in front of all providers. The proxy pattern from Core Concept 1, at organization scale.

Core Concept 11

Testing an API Integration Offline

Three Seams for Fakes

CI has no API key, no budget, and sometimes no network. **Every network boundary needs an injectable fake.**

- **A fake SDK client** — `chat()` accepts a `client` argument; the test passes an object whose `messages.create` records its arguments and returns a canned response
- **LiteLLM's `mock_response`** — `mock_response="..."` returns that text in a normal response, with no network call (streaming too)
- **Injected `sleep` and `rng`** — backoff tests run instantly and deterministically

The Tests

```python
# tests/test_chat.py -- the API rejects system=None; prove we never send it
def test_chat_omits_unset_optional_params():
    client = fake_client(text("ok"))
    assert chat([{"role": "user", "content": "hi"}], client=client, model="m") == "ok"
    sent = client.messages.calls[0]
    assert "system" not in sent and "stop_sequences" not in sent

# tests/test_llm.py -- every LiteLLM provider, for free
@pytest.mark.parametrize("provider", ["openai", "gemini", "anthropic", "ollama"])
def test_litellm_providers_return_a_reply(provider):
    reply = llm.ask("hi", provider=provider, mock_response="hello from mock")
    assert reply.provider == provider and reply.text == "hello from mock"

# tests/test_retry.py -- 429 with retry-after, then 429 without, then success
def test_honors_retry_after_then_backs_off_with_jitter():
    sleeps = []
    fn = Flaky(api_error(429, {"retry-after": "7"}), api_error(429))
    assert with_retries(fn, sleep=sleeps.append, rng=lambda: 0.3) == "ok"
    assert sleeps == [7.0, pytest.approx(2.0 * 0.8)]
```

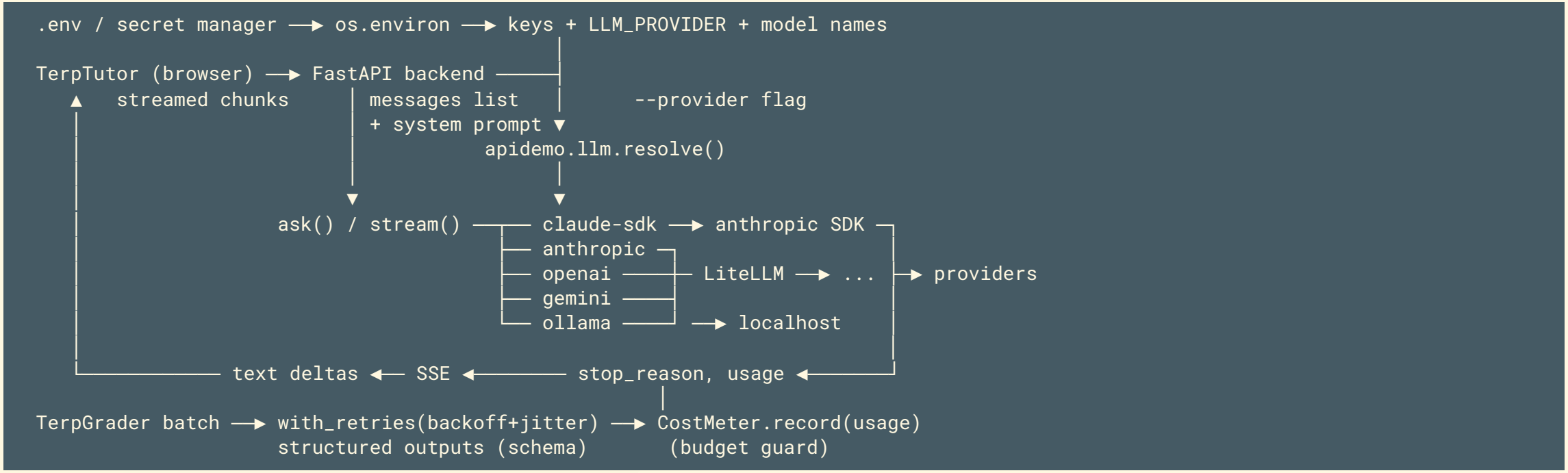
The Suite Runs With No Key

```
$ pytest -q
.....................................................  [100%]
45 passed in 2.03s
```

It also checks that:

- a missing key fails **before** any network call
- `.env` **never overrides** a real environment variable
- `mask()` never reveals the middle of a key
- `text_of` **skips thinking blocks**
- the **budget guard** fires

The Complete Picture



Keys enter once, from the environment. The conversation lives in your code and is resent every call. The system prompt sets behavior; structured outputs guarantee format. Every response feeds the meter; every call is wrapped in retries that respect the taxonomy. **The provider is a configuration value, not a code change.**

Live Demos

Command	Concept	Watch for
`python demos/00_check_env.py`	1	which key came from shell vs. `.env`; all masked
`python demos/01_first_request.py`	2	content blocks, `stop_reason`, `usage`
`python demos/02_raw_http.py --break-key`	2, 8	the same call as plain HTTP; a real 401 body
`python demos/03_multi_turn.py`	3	"another sentence" fails without history
`python demos/04_chat_loop.py`	3	input tokens grow; `/reset` erases memory
`python demos/05_system_prompt.py`	4	answered vs. tutored
`python demos/06_temperature.py -p ollama`	5	identical at 0.0, different at 1.0
`python demos/07_streaming.py race`	6	time to first token vs. total time
`python demos/08_structured_output.py`	7	fenced JSON, prefill, the 400, validated object
`python demos/09_errors_and_retries.py`	8	typed errors; `max_tokens` is a 200; backoff log
`python demos/10_cost_meter.py --budget 0.005`	9	the budget guard stops the batch
`python demos/11_switch_providers.py --compare`	10	one prompt, every provider
`pytest -q`	11	45 tests, no keys, no network

The Project: `claude-api-demos`

```
claude-api-demos/
├── README.md
├── pyproject.toml      # pytest config, `llm` script
├── requirements.txt    # anthropic, litellm, dotenv,
│                       # httpx, pydantic, pytest
├── .gitignore          # .env ignored; .example not
├── .env.example        # every provider, placeholders
├── apidemo/
│   ├── config.py       # load_env, require, mask
│   ├── chat.py         # helpers, chat, text_of
│   ├── retry.py        # taxonomy, with_retries
│   ├── cost.py         # pricing, CostMeter
│   └── llm.py          # the provider switch
├── demos/              # 00_check_env ... 11_switch
└── tests/              # all offline
```

```
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
pip install -e .        # makes `apidemo` importable
pytest -q               # offline -- no key needed

cp .env.example .env    # then paste your keys
python demos/00_check_env.py
python demos/01_first_request.py
python -m apidemo.llm --provider ollama \
  "Explain the TCP handshake in one sentence."
```

Reading order: `config.py` → `chat.py` → `retry.py` → `cost.py` → `llm.py`. Each file maps to a section of the notes.

Common Mistakes

1. Committing `.env`, or adding it to `.gitignore` only after git already tracks it
2. Calling `Anthropic()` before `load_dotenv()`
3. Reading `message.content[0].text` on a model that returns a thinking block first
4. Forgetting to append the assistant's reply to the history
5. Passing `system=None` or `stop_sequences=None` instead of omitting them
6. Copying old code that sends `temperature` or a prefill to a current Claude model → `400`
7. Retrying non-retryable errors; retrying without jitter
8. Ignoring `stop_reason` and parsing truncated output
9. Calling the provider directly from browser or mobile code
10. Assuming a prompt tuned for one provider behaves the same on another

Summary

1. **Keys are money.** Env var or git-ignored `.env`; commit `.env.example`, never `.env`; keys stay server-side; rotate on any leak.
2. **An API call is ordinary HTTP.** Auth + version headers, a JSON body, a JSON response whose **`usage` is the billing ground truth**. Join text blocks, not `content[0]`.
3. **The API is stateless** — your code keeps the transcript, resends it, and pays for it every turn.
4. **A system prompt** sets role and behavior; pass it only when set, never as `None`.
5. **Temperature** sharpens or flattens sampling. Current Claude models no longer accept it; others still do.
6. **Streaming** costs the same and feels far faster: `text_stream` to display, `get_final_message()` to store.
7. **Structured outputs** for machine-readable output on current models; prefill + stop is the older trick.
8. **Retryable or not?** Backoff with jitter, honor `retry-after`, give up eventually. Check `stop_reason` even on a `200`.
9. **Cost is an engineering variable** — model, prompt size, history, caching, batching; a budget guard turns invoices into exceptions.
10. **LiteLLM** makes the provider configuration. The *calls* are interchangeable; the models' behavior and features are not.

Questions?