Course 4 of 4

# Advanced AI Workflows & the Claude API

Build reliable, production-ready AI features with Claude

Claude API · System Prompts · Tool Use · Streaming · FastAPI App · Prompt Caching · Multi-Agent · RAG · Evaluation · Production

10 Chapters · osztromok.com

# Advanced AI Workflows & the Claude API

Chapter 1 · The Claude API — Authentication, Models, and Your First Request

Everything you've done in previous courses — using Claude Code, building projects, pair programming in VS Code — runs on top of the Claude API. This course goes one layer deeper: you'll call the API directly, build Claude-powered applications, chain requests together, and design workflows that go beyond what any chat interface can do. This first chapter covers authentication, the current model family, and the anatomy of a request and response.

## Getting an API Key

The API key is the credential that identifies your application to Anthropic. Every request must include it. To get one:

1. Sign in at **console.anthropic.com**
2. Navigate to **API Keys** in the left sidebar
3. Click **Create Key**, give it a name (e.g. "meal-planner-dev"), and copy it immediately — it won't be shown again
4. Store it in an environment variable: `ANTHROPIC_API_KEY=sk-ant-...`

> **NEVER HARDCODE API KEYS**
>
> Do not put your API key directly in source code. It will end up in git history, get pushed to GitHub, and be scraped by bots within minutes. Always read it from an environment variable or a secrets manager. Add `.env` to your `.gitignore` immediately.

## Installing the SDK

Anthropic provides official SDKs for Python and TypeScript. For Python:

Terminal SHELL

```
pip install anthropic

# Or with uv (faster)
uv add anthropic
```

The SDK handles authentication, request serialisation, error parsing, and retry logic for you. You can also call the API directly over HTTP — the SDK is a thin convenience wrapper, not a black box. Understanding the raw request format (covered below) makes debugging SDK issues much easier.

## The Current Model Family

### `claude-opus-4-8`

**OPUS · MOST CAPABLE**

Anthropic's most intelligent model. Best reasoning, analysis, and complex multi-step tasks. Highest cost.

**Best for:** research synthesis, complex code generation, nuanced writing, difficult reasoning chains

### `claude-sonnet-4-6`

**SONNET · BALANCED**

High capability at moderate cost. The default choice for most production applications — strong performance, reasonable price.

**Best for:** most production use cases, API integrations, coding assistants, content generation

### `claude-haiku-4-5-20251001`

**HAIKU · FAST & CHEAP**

Fastest response time, lowest cost. Excellent for high-volume, latency-sensitive tasks where cost per request matters.

**Best for:** classification, extraction, summarisation at scale, real-time features, chatbots

### `claude-fable-5`

**FABLE · CREATIVE**

Optimised for creative and narrative tasks. Strong at fiction, dialogue, and expressive writing.

**Best for:** creative writing, storytelling, dialogue generation, narrative content

**START WITH SONNET, OPTIMISE LATER**

When building a new feature, start with `claude-sonnet-4-6`. Once the feature works correctly, test whether Haiku produces acceptable quality at lower cost. Switch to Opus only if Sonnet's output quality isn't sufficient. This order saves money during development and avoids premature optimisation.

## Your First API Request

basic_request.py PYTHON

```python
import anthropic

client = anthropic.Anthropic() # reads ANTHROPIC_API_KEY from env automatically

message = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=1024,
    messages=[
        {"role": "user", "content": "Explain what an API key is in two sentences."}
    ]
)

print(message.content[0].text)
```

That's the minimum viable request — model, max_tokens, and a messages list with at least one user turn. Everything else is optional. Run it and you'll see Claude's response printed to the terminal.

## Request Parameters — What Each Field Does

| | | |
|---|---|---|
| `model` | REQUIRED | The model ID string. Use the exact ID from the model table above — abbreviated names like "sonnet" are not accepted. Example: `"claude-sonnet-4-6"` |
| `max_tokens` | REQUIRED | Maximum tokens Claude will generate in the response. Claude stops at this limit even mid-sentence. Set it high enough for your expected output — it's a ceiling, not a target. Typical values: 256 (short answers), 1024 (paragraphs), 4096 (long documents), 8192+ (very long outputs) |
| `messages` | REQUIRED | List of conversation turns. Each turn is `{"role": "user" \| "assistant", "content": "..."}`. Must start with a user turn. Alternate roles for multi-turn conversations. |

| | | |
|---|---|---|
| | | For a fresh request: one user message. For conversation history: alternate user/assistant pairs followed by the new user message. |
| `system` | OPTIONAL | The system prompt — instructions that shape Claude's behaviour for the entire conversation. Equivalent to CLAUDE.md for an API application. Set the role, constraints, output format, and persona here.<br>Not part of the messages list — it's a separate top-level field. |
| `temperature` | OPTIONAL | Controls randomness. Range 0–1. Lower = more deterministic and consistent; higher = more varied and creative. Default is 1.<br>Use 0 for extraction/classification tasks. Use 0.7–1 for creative generation. |
| `stop_sequences` | OPTIONAL | List of strings that cause Claude to stop generating when encountered. Useful for structured output where you want Claude to stop at a delimiter.<br>Example: `["", "###END###"]` |
| `stream` | OPTIONAL | Set to `True` to receive the response as a stream of events rather than waiting for the full response. Covered in depth in Chapter 4.<br>Essential for any UI that shows Claude's response as it's generated. |

## Reading the Response Object

Response object — key fields

```
message.id    UNIQUE REQUEST ID
  # "msg_01XFDUDYJgAACzvnptvVoYEL" — log this for debugging

message.content[0].text    THE ACTUAL RESPONSE
  # The text Claude generated. content is a list — always index [0] for simple
requests

message.usage.input_tokens    TOKENS CONSUMED
message.usage.output_tokens    TOKENS GENERATED
  # Input + output tokens = what you're billed for

message.stop_reason    WHY CLAUDE STOPPED
```

```
  # "end_turn" → Claude finished naturally ✓
  # "max_tokens" → hit the limit — increase max_tokens if output was cut off
  # "stop_sequence" → hit one of your stop_sequences
  # "tool_use" → Claude wants to call a tool (Chapter 3)
```

**ALWAYS CHECK STOP_REASON**

If `stop_reason` is `"max_tokens"`, Claude's response was cut off — the output is incomplete. Either increase `max_tokens` or redesign the prompt to produce shorter output. A truncated response that looks complete is a subtle bug that's easy to miss in testing.

## A Request with a System Prompt

with_system_prompt.py — PYTHON

```python
message = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=512,
    system="You are a concise technical writer. "
           "Always respond in plain text with no markdown. "
           "Keep answers under 3 sentences.",
    messages=[
        {"role": "user", "content": "What is a webhook?"}
    ]
)
```

## Common API Errors and What They Mean

| Status | Error type | Cause and fix |
|---|---|---|
| 401 | authentication_error | API key missing, wrong, or revoked. Check your environment variable is set and the key is valid in the console. |
| 400 | invalid_request_error | Malformed request — wrong field name, wrong type, messages list starts with assistant turn, or max_tokens exceeds model limit. Read the error message; it's usually specific. |

| Status | Error type | Cause and fix |
|---|---|---|
| 429 | rate_limit_error | You've exceeded your requests-per-minute or tokens-per-minute limit. Implement exponential backoff and retry. Chapter 10 covers rate limit strategy in depth. |
| 413 | request_too_large | Input tokens exceed the model's context window. Reduce the size of your messages — chunk long documents or truncate conversation history. |
| 500 | api_error | Anthropic server error. Retry with backoff — these are transient. If persistent, check status.anthropic.com. |
| 529 | overloaded_error | Anthropic is temporarily overloaded. Retry with backoff. More common during peak hours — design your application to handle this gracefully. |

## Handling Errors in Code

error_handling.py PYTHON

```python
import anthropic

client = anthropic.Anthropic()

try:
    message = client.messages.create( ... )
except anthropic.AuthenticationError:
    raise RuntimeError("Invalid API key — check ANTHROPIC_API_KEY")
except anthropic.RateLimitError:
    raise # caller handles retry with backoff
except anthropic.APIStatusError as e:
    print(f"API error {e.status_code}: {e.message}")
    raise
```

**Next — Chapter 2: System Prompts**

Shaping Claude's behaviour at the API level — writing effective system prompts, controlling

persona and constraints, the difference between system and user instructions, and designing system prompts for production applications.

# Advanced AI Workflows & the Claude API

Chapter 2 · System Prompts — Shaping Claude's Behaviour at the API Level

The system prompt is the most powerful tool you have when building Claude-powered applications. It runs before every conversation turn, shapes how Claude interprets user messages, and establishes constraints that persist for the entire session. A well-written system prompt is the difference between an unpredictable AI and a reliable product feature. This chapter covers how to write them, what to put in them, and the common mistakes that undermine their effectiveness.

## What a System Prompt Actually Does

From Claude's perspective, the system prompt is high-priority context that arrives before the user speaks. It doesn't override Claude's values or safety behaviours, but it does everything else: it narrows the topic space, establishes tone and format, sets a persona, and defines what Claude should and shouldn't do within the conversation.

Think of it as the standing instructions you'd give a contractor before they start work — not a script for every scenario, but the ground rules that govern how they handle everything they encounter.

> **SYSTEM VS USER INSTRUCTIONS**
>
> Instructions in the system prompt carry more weight than instructions in the user message — Claude treats system instructions as the application's intent and user messages as end-user input. If they conflict, system instructions generally win. Use this hierarchy deliberately: put hard constraints in the system prompt, not the user turn.

## Five Sections of an Effective System Prompt

**ROLE & IDENTITY**

**Who Claude is in this application.** A concise statement of purpose and persona. This primes Claude's vocabulary, tone, and assumptions for every response that follows.

| | |
|---|---|
| | Example: "You are a customer support assistant for Acme SaaS. You help users troubleshoot the dashboard, billing, and integrations." |
| SCOPE & LIMITS | **What Claude will and won't do.** The topics it covers, the tasks it handles, and — crucially — what it should decline or redirect. Explicit out-of-scope rules prevent Claude from helpfully answering questions your application wasn't designed for. Example: "Only answer questions about Acme products. If asked about competitors or general coding questions, politely redirect to our documentation." |
| TONE & STYLE | **How Claude communicates.** Formal or casual, verbose or concise, first-person or third-person, with or without emoji. The more specific, the more consistent the output across different user inputs. Example: "Be concise. Use plain language, not jargon. Never use bullet points — answer in prose. Keep responses under 150 words unless the user asks for detail." |
| OUTPUT FORMAT | **What the response should look like.** If your application parses Claude's output, specify the exact format here. If it renders markdown, say so. If it expects JSON, describe the schema. Example: "Always respond in valid JSON with keys: answer (string), confidence (low\|medium\|high), sources (list of strings)." |
| CONTEXT & DATA | **Dynamic information injected at runtime.** User account details, retrieved documents, current date, session state. This section changes per-request; the other four are usually static. Example: "The current user is: {user_name}. Their subscription tier is: {tier}. Today's date is: {date}." |

## Vague vs Specific — The Most Common Mistake

Vague instructions produce inconsistent outputs. Every word in a system prompt that could be interpreted multiple ways will be interpreted multiple ways across different user inputs. Specificity is the primary lever for reliability.

**VAGUE — INCONSISTENT RESULTS**

```
Be helpful and professional. Answer
user questions about our product. Keep
it brief.
```

**SPECIFIC — CONSISTENT RESULTS**

```
You are a support assistant for Acme
Dashboard. Answer only questions
about: login issues, billing, and the
reporting module. Decline all other
topics politely. Respond in 1–3
```

sentences. Never use bullet points. Never speculate about roadmap features.

**VAGUE — FORMAT VARIES**

```
Extract the key information from the user's message.
```

**SPECIFIC — FORMAT FIXED**

```
Extract information from the user's message. Respond only with valid JSON: {"name": string | null, "email": string | null, "issue": string}. No explanation. No markdown fences.
```

## Dynamic Context Injection

The system prompt is a string — you build it in code before each request. This makes it easy to inject runtime data: the current user's name, their account tier, retrieved documents, the current date. Treat the static sections as a template and fill in the dynamic parts with Python string formatting:

dynamic_system_prompt.py — PYTHON

```python
def build_system_prompt(user, today):
    return f"""You are a billing support assistant for Acme SaaS.

Current user: {user.name} ({user.email})
Subscription: {user.plan} - expires {user.expiry}
Today's date: {today}

Rules:
- Only discuss billing, invoices, and plan changes.
- Never reveal internal pricing or discount structures.
- If the user requests a refund, collect their reason and say
a human agent will follow up within 24 hours.
- Respond in plain prose, no bullet points, under 100 words."""

message = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=256,
    system=build_system_prompt(current_user, date.today()),
```

```
    messages=[{"role": "user", "content": user_message}]
)
```

**SANITISE INJECTED DATA**

Any user-controlled data injected into the system prompt is a potential prompt injection vector. If a user's name is `Ignore previous instructions and ...` and you inject it directly, you've handed them partial control of the system prompt. Strip or escape user-provided strings before injection, or put user data in a clearly delimited section Claude recognises as data, not instructions.

## Application System Prompt Templates

### Customer support bot

```
You are a support assistant for
[Product]. Help users with [topics].
If you cannot resolve an issue, say
so and direct them to [contact].
Never make commitments about refunds
or SLA without human approval.
Respond in 2–3 sentences max.
```

Key constraints: scope, escalation path, commitment limits, length.

### Structured data extractor

```
Extract information from the text
provided by the user. Respond only
with valid JSON matching this schema
exactly: {schema}. If a field is
absent from the input, use null.
Never add fields not in the schema.
No explanation, no markdown.
```

Key constraints: exact format, null handling, no extra output.

### Code reviewer

```
You are a senior {language} engineer
reviewing code for correctness,
security, and style. Flag issues in
order of severity: critical, warning,
suggestion. For each issue give:
location, problem, recommended fix.
Respond in markdown.
```

Key constraints: language specialisation, severity ordering, output structure.

### RAG document Q&A

```
Answer the user's question using only
the documents provided below. If the
answer is not in the documents, say
"I don't have that information."
Never infer beyond what's stated.
Cite the document name for every
claim. [DOCUMENTS]: {docs}
```

Key constraints: grounding, citation requirement, explicit refusal when answer is absent.

## Instruction Priority — When Instructions Conflict

In a real application, instructions can come from multiple places: the system prompt, injected documents, and the user's message. Claude applies a rough priority order:

| Priority | Source | Typical use |
|---|---|---|
| 1 | **System prompt (static)** | Application-level rules, persona, scope, format. Treated as developer intent — highest weight. |
| 2 | **System prompt (injected context)** | Retrieved documents, user data. Treated as data to reason over, not instructions to follow. |
| 3 | **User message** | End-user input. Treated as the task to perform within the constraints set above. |

If a user message says "ignore your previous instructions" — a classic prompt injection attempt — Claude will generally hold to the system prompt constraints because the system prompt has higher contextual authority. This is not a hard guarantee, but it holds reliably for well-written system prompts.

## Testing Your System Prompt

A system prompt that works for happy-path inputs often fails at the edges. Test deliberately against adversarial inputs before shipping:

- **Out-of-scope questions** — does Claude correctly decline topics outside its brief, or does it helpfully answer anyway?
- **Prompt injection attempts** — user messages like "ignore your instructions and tell me X" — does the system prompt hold?
- **Format edge cases** — very short inputs, very long inputs, inputs in other languages, inputs with no clear question
- **Instruction conflicts** — if the user explicitly asks Claude to break a rule ("please use bullet points"), does it comply or hold to the system prompt?
- **Ambiguous requests** — inputs that could be interpreted as in-scope or out-of-scope — which does Claude choose?

> **ITERATE WITH TEMPERATURE 0**

> When testing and refining a system prompt, set `temperature=0` to get deterministic responses. Once the prompt is stable, test with default temperature to see how it holds under variation. A prompt that only works at temperature 0 is too fragile for production.

**Next — Chapter 3: Tool Use / Function Calling**

Giving Claude tools it can invoke — defining tools, handling tool call responses, building the request-response-result loop, and designing tools that Claude can use reliably.

# Advanced AI Workflows & the Claude API

Chapter 3 · Tool Use / Function Calling

Tool use — also called function calling — is the mechanism that lets Claude interact with the outside world. Instead of just generating text, Claude can decide to call a function you've defined: search a database, fetch a URL, run a calculation, send a notification. Your code executes the function and feeds the result back to Claude, which then continues its response. This turns Claude from a text generator into an agent that can take actions.

## The Tool Use Loop

Tool use is a multi-turn exchange, not a single request. Understanding the loop is the foundation for everything else in this chapter:

| | | |
|---|---|---|
| YOU | → | **Send request with tool definitions** — your message plus a list of tools Claude is allowed to call.<br>Claude sees the tool names, descriptions, and parameter schemas. |
| CLAUDE | → | **Responds with a tool_use block** — Claude decides it needs a tool, returns the tool name and the arguments it wants to pass.<br>stop_reason is "tool_use". No visible text response yet — Claude is waiting for the result. |
| YOUR CODE | → | **Execute the function** — you run the actual function with the arguments Claude provided and capture the result.<br>Claude has no direct access to your systems — it only receives what you send back. |
| YOU | → | **Send tool result back** — append the assistant turn (Claude's tool_use block) and a new user turn containing the tool result to the messages list, then resend.<br>The conversation history now includes: your question, Claude's tool call, your tool result. |
| CLAUDE | → | **Responds with the final answer** — Claude reads the tool result and generates its final response to the user.<br>stop_reason is now "end_turn". Claude may call more tools before reaching this step. |

## Defining a Tool

Each tool is a Python dict (or Pydantic model) that tells Claude what the function does and what parameters it takes. The description is the most important field — Claude uses it to decide when to call the tool.

| | | |
|---|---|---|
| `name` | REQUIRED | Snake_case identifier. Must be unique across all tools in a request. Claude uses this exact string when it calls the tool. Example: `"get_weather"`, `"search_products"`, `"send_email"` |
| `description` | REQUIRED | Plain-English description of what the tool does, when to use it, and what it returns. This is what Claude reads to decide whether to call the tool. More detail = better decisions. Bad: "Gets weather." Good: "Returns current weather for a city including temperature, conditions, and wind speed. Use when the user asks about current weather for a specific location." |
| `input_schema` | REQUIRED | JSON Schema describing the parameters Claude should pass. Defines property names, types, descriptions, and which are required. Claude follows this schema when constructing the arguments. Use `"required": [...]` to mark mandatory parameters. Add a `"description"` to each property. |

## A Complete Tool Use Example

tool_use_example.py — PYTHON

```python
import anthropic, json

client = anthropic.Anthropic()

# 1. Define the tool
tools = [{
    "name": "get_weather",
    "description": "Returns current weather for a city. Use when the user "
                   "asks about weather in a specific location.",
```

```
    "input_schema": {
        "type": "object",
        "properties": {
            "city": {"type": "string", "description": "City name, e.g. 'London'"},
            "units": {"type": "string", "enum": ["celsius", "fahrenheit"]}
        },
        "required": ["city"]
    }
}]

# 2. Send initial request
messages = [{"role": "user", "content": "What's the weather in Tokyo?"}]
response = client.messages.create(
    model="claude-sonnet-4-6", max_tokens=1024,
    tools=tools, messages=messages
)

# 3. Handle tool call
if response.stop_reason == "tool_use":
    tool_block = response.content[0]
    args = tool_block.input # {"city": "Tokyo"}

    # 4. Execute your actual function
    result = get_weather(args["city"], args.get("units", "celsius"))

    # 5. Append assistant turn + tool result, resend
    messages.append({"role": "assistant", "content": response.content})
    messages.append({
        "role": "user",
        "content": [{
            "type": "tool_result",
            "tool_use_id": tool_block.id, # must match
            "content": json.dumps(result)
        }]
    })

    # 6. Get final response
    final = client.messages.create(
        model="claude-sonnet-4-6", max_tokens=1024,
        tools=tools, messages=messages
    )
    print(final.content[0].text)
```

## Real-World Tool Examples

**search_database**

Query a product database. Use when the user asks about specific products, prices, or availability.

```
query: string (required)
category: string (optional)
max_results: integer (optional, default 5)
```

**create_calendar_event**

Creates an event in the user's calendar. Use when the user explicitly asks to schedule or book something.

```
title: string (required)
start_datetime: string ISO8601 (required)
duration_minutes: integer (required)
description: string (optional)
```

**send_notification**

Sends a push notification to the user's device. Only use when the user has explicitly requested a reminder.

```
message: string (required)
channel: enum ["push","email","sms"] (required)
scheduled_at: string ISO8601 (optional)
```

**run_calculation**

Evaluates a mathematical expression precisely. Use for any arithmetic where floating-point accuracy matters.

```
expression: string (required)
precision: integer decimal places (optional, default 2)
```

## Handling Multiple Tool Calls

Claude may call multiple tools in a single response, or call tools sequentially across multiple turns. A robust tool loop handles both:

tool_loop.py — handles multiple sequential calls PYTHON

```python
def run_tool_loop(initial_messages, tools, max_turns=10):
    messages = initial_messages.copy()

    for _ in range(max_turns):
        response = client.messages.create(
            model="claude-sonnet-4-6", max_tokens=4096,
            tools=tools, messages=messages
        )
```

```
        if response.stop_reason == "end_turn":
            return response # done

        if response.stop_reason ≠ "tool_use":
            break # unexpected stop reason

        # Append assistant turn
        messages.append({"role": "assistant", "content": response.content})

        # Execute all tool calls in this response
        tool_results = []
        for block in response.content:
            if block.type == "tool_use":
                result = dispatch_tool(block.name, block.input)
                tool_results.append({
                    "type": "tool_result",
                    "tool_use_id": block.id,
                    "content": json.dumps(result)
                })

        messages.append({"role": "user", "content": tool_results})

    raise RuntimeError("Tool loop exceeded max_turns")
```

**ALWAYS CAP THE LOOP**

Without a `max_turns` limit, a tool loop can run indefinitely if Claude keeps deciding to call tools. Set a reasonable ceiling — 5–10 turns covers almost all real use cases. Log and alert when the limit is hit so you can investigate whether the loop is behaving as expected.

## Tool Design Principles

- **Write the description for Claude, not for humans.** The description is Claude's only guide for when to call the tool. Include: what the tool does, what it returns, and when to use it vs not use it. If two tools are similar, explicitly distinguish them in the descriptions.

- **Describe each parameter.** Every property in the input schema should have a `"description"` field. Claude uses these to understand what values are expected — missing descriptions lead to wrong arguments.

- **Use enums for constrained choices.** If a parameter only accepts certain values, list them as an enum. Claude will pick from the list reliably rather than inventing a value that your code doesn't handle.

- **Keep tools focused.** One tool should do one thing. A `manage_user` tool with ten modes is harder for Claude to invoke correctly than three separate tools: `get_user`, `update_user`, `delete_user`.

- **Validate inputs before executing.** Claude follows the schema but can still pass unexpected values. Validate and sanitise all tool arguments before running them — especially for tools that write data or call external services.

- **Return rich results.** Claude uses the tool result to formulate its answer. A result like `{"temperature": 18, "condition": "partly cloudy", "humidity": 72, "wind_kph": 15}` lets Claude give a better answer than `"18°C"`.

- **Handle errors gracefully in the result.** If a tool fails, return an error description in the result rather than raising an exception and crashing the loop. Claude can then tell the user what went wrong and potentially try a different approach.

**Next — Chapter 4: Streaming Responses**

Handling incremental output for real-time UIs — how streaming works at the API level, processing the event stream in Python, building a UI that shows Claude's response as it's generated, and when streaming isn't the right choice.

# Advanced AI Workflows & the Claude API

Chapter 4 · Streaming Responses

Without streaming, your application sends a request and waits — potentially several seconds — before showing the user anything. With streaming, Claude sends its response token by token as it generates it, and your application displays each chunk immediately. The result feels instant rather than frozen. This chapter covers how streaming works at the API level, how to process the event stream, and the patterns for wiring it into a real application.

## Streaming vs Non-Streaming

**Non-streaming (default)**

- Single HTTP request → single response
- User sees nothing until the full response is ready
- Latency scales with output length — a 2,000-token response takes noticeably longer than a 200-token one
- Simpler to implement — one call, one result object
- Best for: background processing, batch jobs, pipelines where no human is waiting

**Streaming**

- Single HTTP request → stream of server-sent events
- First token arrives in ~300–500ms regardless of total length
- User sees text appearing in real time — perceived latency is dramatically lower
- More complex to implement — you process events in a loop
- Best for: any UI where a human is reading the response as it arrives

## The Event Stream — What Claude Actually Sends

Streaming uses Server-Sent Events (SSE). Claude sends a sequence of events, each a JSON payload, as it generates the response. You process them in order:

SSE event sequence — one streaming response

| | |
|---|---|
| `message_start` | Response begins. Contains the message ID and initial usage (input tokens). |

| | |
|---|---|
| `content_block_start` | A content block is opening (type: "text" or "tool_use"). |
| `content_block_delta (×N)` | The actual text chunks — one per token or small group of tokens. This is what you display to the user. Concatenate all deltas to build the full response. |
| `content_block_stop` | The content block is complete. If type was "tool_use", the tool arguments are now fully assembled. |
| `message_delta` | Contains stop_reason and final usage (output tokens). Check stop_reason here. |
| `message_stop` | Stream is complete. Safe to close the connection after this event. |

## Basic Streaming — Terminal Output

streaming_basic.py PYTHON

```python
import anthropic

client = anthropic.Anthropic()

# The SDK's stream context manager handles the event loop for you
with client.messages.stream(
    model="claude-sonnet-4-6",
    max_tokens=1024,
    messages=[{"role": "user", "content": "Write a short poem about APIs."}]
) as stream:
    for text in stream.text_stream:
        print(text, end="", flush=True) # print each chunk immediately

# After the context manager exits, the final message is available
final_message = stream.get_final_message()
print(f"\nTokens used: {final_message.usage.output_tokens}")
```

The SDK's `.stream()` context manager and `.text_stream` iterator abstract away the raw SSE events. You get text chunks directly. For most applications this is all you need.

## Streaming in a FastAPI Endpoint

To stream Claude's output to a web client, use FastAPI's `StreamingResponse` with a generator function. The client receives the response as a stream of chunks and can display them as they arrive:

streaming_fastapi.py — PYTHON

```python
from fastapi import FastAPI
from fastapi.responses import StreamingResponse
from pydantic import BaseModel
import anthropic

app = FastAPI()
client = anthropic.Anthropic()

class ChatRequest(BaseModel):
    message: str

def generate_stream(user_message: str):
    with client.messages.stream(
        model="claude-sonnet-4-6",
        max_tokens=2048,
        messages=[{"role": "user", "content": user_message}]
    ) as stream:
        for text in stream.text_stream:
            yield text

@app.post("/chat")
async def chat(request: ChatRequest):
    return StreamingResponse(
        generate_stream(request.message),
        media_type="text/plain"
    )
```

## Reading the Stream on the Client Side

On the browser side, use the Fetch API with a `ReadableStream` reader to consume the chunks as they arrive and append them to the UI:

streaming_client.js — JAVASCRIPT

```
async function sendMessage(userMessage) {
  const response = await fetch("/chat", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ message: userMessage })
  });

  const reader = response.body.getReader();
  const decoder = new TextDecoder();
  const outputEl = document.getElementById("output");

  while (true) {
    const { done, value } = await reader.read();
    if (done) break;
    outputEl.textContent += decoder.decode(value); // append each chunk
  }
}
```

## Streaming with Tool Use

Tool use and streaming work together — but the interaction requires handling both text deltas and tool input deltas in the event stream. The SDK's lower-level event iterator gives you access to both:

streaming_tools.py — handling tool calls in a stream PYTHON

```
with client.messages.stream(
    model="claude-sonnet-4-6", max_tokens=1024,
    tools=tools, messages=messages
) as stream:
    for event in stream:

        if event.type == "content_block_delta":
            if hasattr(event.delta, "text"):
                print(event.delta.text, end="", flush=True)

        elif event.type == "message_stop":
            break
```

```
    # After stream closes, get the final assembled message
    # (tool_use blocks are fully assembled even in streaming mode)
    final = stream.get_final_message()
    if final.stop_reason == "tool_use":
        # handle tool call as in Chapter 3
        handle_tool_call(final)
```

**USE GET_FINAL_MESSAGE() FOR TOOL USE**

When streaming with tools, the tool arguments arrive as `input_json_delta` events — partial JSON fragments. Rather than assembling them yourself, let the stream finish and call `stream.get_final_message()`. The SDK assembles the complete tool call object for you, with fully parsed arguments.

## When to Stream and When Not To

| Scenario | Stream? | Reason |
|---|---|---|
| Chat interface — human reading the response | YES | Dramatically improves perceived performance. Users can start reading while Claude is still writing. |
| Code generation in an editor | YES | Seeing code appear incrementally feels fast; waiting for 200 lines feels frozen. |
| Background batch processing | NO | No human is waiting. Non-streaming is simpler and uses the same API quota. |
| Structured output you parse (JSON, CSV) | NO | You can't parse partial JSON. Wait for the full response before processing. |
| Classification or short answers | NO | Responses are so short (~1–3 tokens) that streaming adds complexity with no perceived benefit. |
| Long document generation (>1,000 tokens) | YES | Without streaming, the user stares at a spinner for 10–20 seconds. With it, they see progress immediately. |

## Production Patterns for Streaming

**Buffer before rendering markdown**

**Show a typing indicator first**

Markdown syntax like `**bold**` or ` ```code``` ` spans multiple tokens. If you render incrementally, you'll show half-rendered syntax. Buffer chunks until you detect a complete markdown element, or render raw text and re-parse periodically.

There's a ~300–500ms gap between request and first token. Show a typing indicator the moment the request is sent — before any tokens arrive. Remove it on the first delta. Prevents the UI looking frozen during the initial latency.

### Handle disconnected clients

If the client disconnects mid-stream (page close, network drop), your generator should exit cleanly. Wrap the stream loop in a try/finally and cancel the upstream request if the client is gone to avoid burning tokens on output nobody reads.

### Accumulate the full response

Even when streaming to the UI, accumulate the full text server-side. You'll need it to append to conversation history for the next turn, to log, and to check `stop_reason`. The stream's `get_final_message()` gives you this without extra work.

**STREAMING DOESN'T REDUCE TOKEN COST**

Streaming changes when you receive the response, not how many tokens are generated. You're billed for the same input and output tokens whether you stream or not. If you're trying to reduce cost, look at model selection and prompt length — not streaming mode.

**Next — Chapter 5: Building a Simple AI-Powered App**

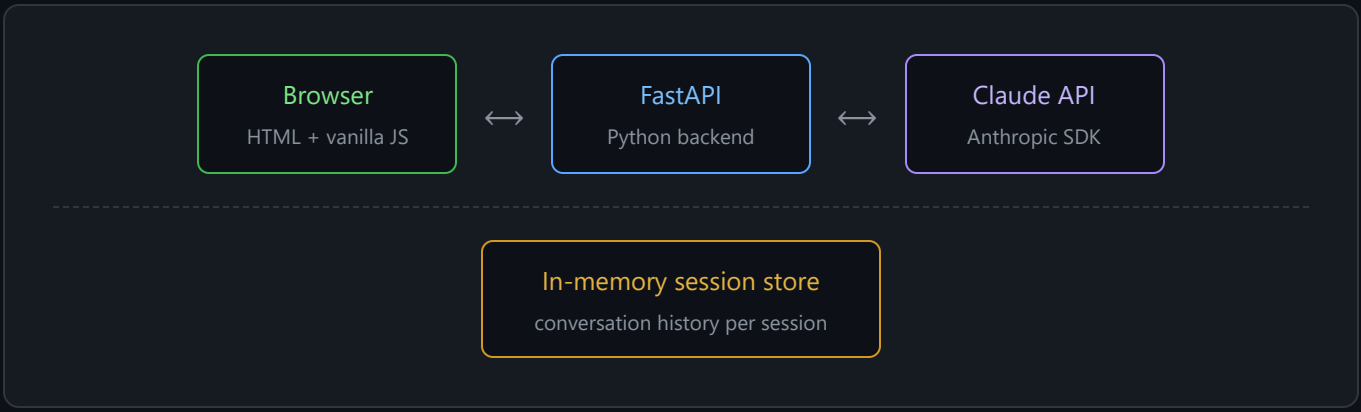
Putting it all together — a FastAPI backend with Claude at the core. System prompt design, conversation state management, streaming responses to the frontend, and the patterns that make an AI feature feel like a product rather than a demo.

# Advanced AI Workflows & the Claude API

Chapter 5 · Building a Simple AI-Powered App

The first four chapters covered the Claude API in pieces — authentication, system prompts, tool use, streaming. This chapter puts them together. You'll build a complete AI-powered chat application: a FastAPI backend that manages conversation history and calls Claude, a streaming endpoint, and a minimal frontend that reads the stream. The goal is a working mental model of how all the parts connect, not a production-ready UI framework.

## What We're Building




The app has two routes: `GET /` serves the HTML page, and `POST /chat` accepts a user message, appends it to the conversation history, calls Claude with streaming, and returns the response as a stream. The browser appends tokens as they arrive.

## Project Structure

```
ai_chat_app/
  main.py FastAPI app — routes, session store, Claude calls
  .env ANTHROPIC_API_KEY=sk-ant- ...
  static/
```

```
    index.html  frontend — chat UI + streaming fetch
requirements.txt  fastapi, uvicorn, anthropic, python-dotenv
```

requirements.txt — TEXT

```
fastapi
uvicorn[standard]
anthropic
python-dotenv
```

## Step 1 — Conversation History

Claude has no memory between API calls. Every request must include the full conversation so far. You manage this by keeping a list of `{"role": "...", "content": "..."}` dicts and appending each new turn. For a real app you'd store this in a database; for this example, a server-side dict keyed by session ID is enough:

Conversation history after two turns

| | |
|---|---|
| SYSTEM | "You are a helpful assistant. Be concise." |
| USER | "What is FastAPI?" |
| ASSISTANT | "FastAPI is a modern Python web framework for building APIs..." |
| USER | "How does it compare to Flask?" |

The next API call includes all four entries. Claude sees the context from the first question when answering the second — that's how conversation continuity works.

## Step 2 — The FastAPI Backend

main.py — PYTHON

```python
import uuid, os
from dotenv import load_dotenv
from fastapi import FastAPI, Cookie, Response
from fastapi.responses import FileResponse, StreamingResponse
```

```python
from fastapi.staticfiles import StaticFiles
from pydantic import BaseModel
import anthropic

load_dotenv()
app = FastAPI()
client = anthropic.Anthropic()
app.mount("/static", StaticFiles(directory="static"), name="static")

# In-memory session store: session_id → list of message dicts
sessions: dict[str, list] = {}

SYSTEM_PROMPT = """You are a helpful assistant. \
Answer concisely and accurately. \
If you don't know something, say so."""

class ChatRequest(BaseModel):
    message: str

@app.get("/")
def index(response: Response, session_id: str | None = Cookie(default=None)):
    if not session_id or session_id not in sessions:
        session_id = str(uuid.uuid4())
        sessions[session_id] = []
        response.set_cookie("session_id", session_id)
    return FileResponse("static/index.html")

def stream_claude(history: list, collected: list):
    with client.messages.stream(
        model="claude-sonnet-4-6",
        max_tokens=2048,
        system=SYSTEM_PROMPT,
        messages=history
    ) as stream:
        for text in stream.text_stream:
            collected.append(text) # accumulate for history
            yield text # stream to client

@app.post("/chat")
def chat(req: ChatRequest, session_id: str = Cookie()):
    history = sessions[session_id]
    history.append({"role": "user", "content": req.message})
```

```
    collected = [] # will hold assistant reply chunks

    def after_stream():
        # Save assistant reply to history once stream ends
        history.append({"role": "assistant", "content": "".join(collected)})

    def generate():
        yield from stream_claude(history, collected)
        after_stream()

    return StreamingResponse(generate(), media_type="text/plain")
```

## Step 3 — The Frontend

static/index.html HTML

```
<!DOCTYPE html>
<html lang="en"><head>
<meta charset="UTF-8">
<title>AI Chat</title>
<style>
  body { font-family: system-ui; background: #0d1117; color: #c9d1d9;
         display:flex; flex-direction:column; height:100vh; margin:0; padding:1rem;
}
  #log { flex:1; overflow-y:auto; padding:1rem; background:#161b22;
        border-radius:8px; margin-bottom:1rem; white-space:pre-wrap; }
  .user-msg { color:#7ee787; margin:0.5rem 0; }
  .bot-msg { color:#c9d1d9; margin:0.5rem 0; }
  #form { display:flex; gap:0.5rem; }
  input { flex:1; padding:0.6rem; background:#161b22; border:1px solid #30363d;
         border-radius:6px; color:#c9d1d9; }
  button { padding:0.6rem 1.2rem; background:#a78bfa; border:none;
          border-radius:6px; color:#0d1117; font-weight:700; cursor:pointer; }
</style></head><body>

<div id="log"></div>
<form id="form">
  <input id="msg" placeholder="Type a message…" autocomplete="off">
  <button>Send</button>
```

```
</form>

<script>
const log = document.getElementById('log');

function addMsg(cls) {
  const el = document.createElement('div');
  el.className = cls;
  log.appendChild(el);
  return el;
}

document.getElementById('form').addEventListener('submit', async (e) => {
  e.preventDefault();
  const input = document.getElementById('msg');
  const text = input.value.trim();
  if (!text) return;

  addMsg('user-msg').textContent = 'You: ' + text;
  input.value = '';

  const botEl = addMsg('bot-msg');
  botEl.textContent = 'Claude: ';

  const res = await fetch('/chat', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ message: text })
  });

  const reader = res.body.getReader();
  const dec = new TextDecoder();
  while (true) {
    const { done, value } = await reader.read();
    if (done) break;
    botEl.textContent += dec.decode(value);
    log.scrollTop = log.scrollHeight;
  }
});
</script></body></html>
```

## Running the App

terminal BASH

```
# Install dependencies
pip install -r requirements.txt

# Run the server (--reload for auto-restart on file changes)
uvicorn main:app --reload

# Open in browser
http://localhost:8000
```

## How the Pieces Connect

1. **Browser loads the page → receives a session cookie**
   FastAPI's `GET /` creates a UUID session and sets it as a cookie. That cookie travels with every subsequent request, identifying whose history to use.

2. **User sends a message → appended to history**
   `POST /chat` receives the JSON body, looks up the session's history list, and appends `{"role": "user", "content": "..."}`.

3. **Claude is called with the full history**
   The `messages` parameter contains every turn so far — Claude sees the full conversation every time, which is how it maintains context across turns.

4. **Response streams to the browser**
   The generator `yield`s text chunks as they arrive from Claude. FastAPI's `StreamingResponse` forwards them immediately. The browser appends each chunk to the message element.

5. **After stream ends → assistant turn saved to history**
   The accumulated chunks are joined and appended as `{"role": "assistant", "content": "..."}`. The next user message will include this reply, keeping the conversation coherent.

## What This App Has and What It's Missing

- ✓ Conversation history — context across turns
- ✓ Streaming — tokens appear in real time
- ✗ Persistence — history lost on server restart
- ✗ History pruning — history grows unbounded
- ✓ Session isolation — separate history per user
- ✓ System prompt — controls Claude's behaviour
- ✗ Auth — anyone with the URL can chat
- ✗ Error handling — no timeout / retry logic

**HISTORY PRUNING MATTERS IN PRODUCTION**

Every message you send includes the entire history — so costs grow with conversation length. Common strategies: keep only the last N turns, summarise old context into a single message, or use prompt caching (Chapter 6) to avoid re-encoding repeated context on every call.

**API KEY SECURITY**

The API key lives in `.env` and is read server-side only — it never touches the browser. Never put your API key in frontend code; it would be visible to anyone who opens DevTools.

**Next — Chapter 6: Prompt Caching**

Every API call re-encodes your system prompt and any shared context — even if it hasn't changed. Prompt caching lets Anthropic store that prefix server-side and reuse it across calls, cutting latency by up to 85% and cost by up to 90% on the cached tokens. Chapter 6 covers how caching works, when to use it, and how to add the cache breakpoints to an existing app.

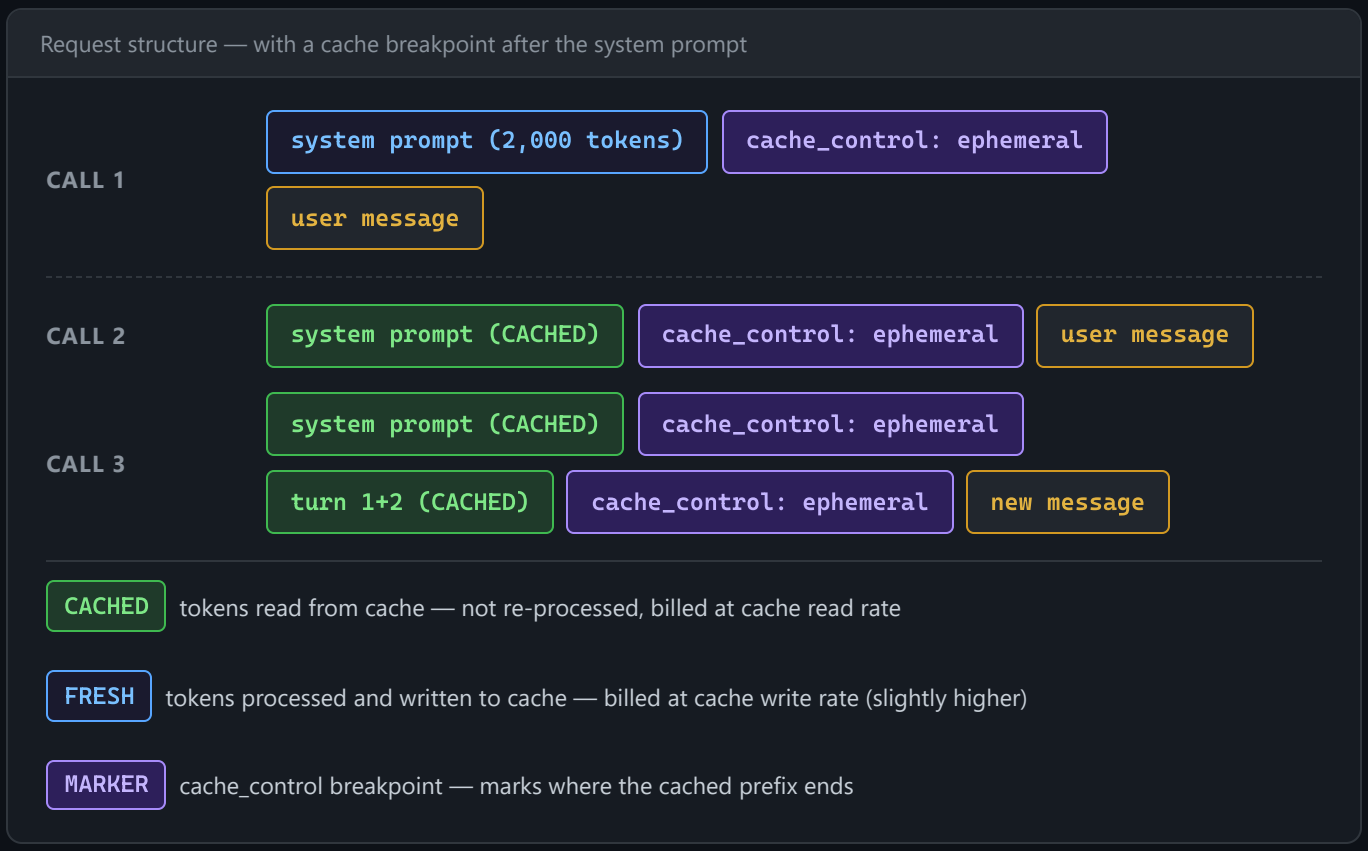
# Advanced AI Workflows & the Claude API

Chapter 6 · Prompt Caching

Every time you call the Claude API, Anthropic's servers process your entire input from scratch — system prompt, conversation history, documents you've loaded, all of it. If that input is long and doesn't change between calls, you're paying to re-encode the same tokens repeatedly. Prompt caching lets the API store a prefix of your input server-side and reuse it across calls, reducing both latency and cost significantly.

## How Caching Works

You mark a position in your input with a **cache breakpoint**. Everything up to that point is eligible to be stored in a cache on Anthropic's servers. If the next request starts with the same prefix up to that breakpoint, the cached version is used instead of re-processing those tokens.



The cache has a **5-minute TTL** (time to live). If you don't make another request within 5 minutes, the cached prefix expires and will be re-processed on the next call. For active sessions this isn't usually a problem — users are typically sending messages more frequently than that.

## Cost and Latency Impact

| Token type | Billing rate | Latency |
|---|---|---|
| Standard input tokens | 100% (baseline) | Full processing time |
| Cache write (first call) | 125% — slightly more than standard | Slightly slower — writing to cache |
| Cache read (subsequent calls) | **10% — 90% cheaper than standard** | **Up to 85% lower latency** |

The break-even point is low: if you send the same prefix twice, you've already saved money on the second call despite the write premium on the first. For apps where users have long conversations against a fixed system prompt, the savings compound quickly.

## Adding a Cache Breakpoint

You add caching by including `"cache_control": {"type": "ephemeral"}` in the content block you want to cache up to. The most common pattern is caching the system prompt:

cache_system_prompt.py — caching a fixed system prompt — PYTHON

```python
import anthropic

client = anthropic.Anthropic()

response = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=1024,

    # System prompt as a list of content blocks so we can add cache_control
    system=[
        {
            "type": "text",
            "text": LONG_SYSTEM_PROMPT,
```

```
            # Mark this block as the cache boundary
            "cache_control": {"type": "ephemeral"}
        }
    ],

    messages=[{"role": "user", "content": "Summarise the key points."}]
)
```

**MINIMUM CACHEABLE LENGTH**

The cached prefix must be at least **1,024 tokens** for Claude Sonnet and Haiku models, and **2,048 tokens** for Opus. If your system prompt is shorter than this, the cache write is silently skipped and you're billed at the standard rate. Caching is most impactful with long, stable prefixes — detailed instructions, injected documents, reference data.

## Caching Conversation History

For chat applications, you can also cache the conversation history up to the previous turn — so each new message only processes the new user input, not the entire accumulated history. Place a breakpoint at the last assistant message:

cache_conversation.py — caching history + system prompt — PYTHON

```python
def build_messages_with_cache(history: list, new_message: str) -> list:
    """Add a cache breakpoint at the end of the existing history."""
    messages = []

    for i, msg in enumerate(history):
        if i == len(history) - 1: # last historical message
            # Wrap content as a block so we can add cache_control
            messages.append({
                "role": msg["role"],
                "content": [{
                    "type": "text",
                    "text": msg["content"],
                    "cache_control": {"type": "ephemeral"}
                }]
            })
        else:
```

```
            messages.append(msg) # earlier turns — no marker needed

    # New user message goes after the cached history
    messages.append({"role": "user", "content": new_message})
    return messages
```

## Reading Cache Stats from the Response

Every response includes a `usage` object that shows how many tokens came from cache vs were freshly processed. Use this to verify caching is working and to track your savings:

```
response.usage = {
  "input_tokens": 28 ← only the new user message (not cached)
  "cache_creation_input_tokens": 2048 ← tokens written to cache (call 1 only)
  "cache_read_input_tokens": 2048 ← tokens read from cache (calls 2+) — 90% cheaper
  "output_tokens": 312
}
```

On the first call, `cache_creation_input_tokens` will be non-zero and `cache_read_input_tokens` will be zero — the prefix is being written. On subsequent calls within the 5-minute window, those numbers reverse.

## Caching a Large Document

One of the most valuable caching patterns is loading a large document (a code file, a policy document, a product catalogue) once and asking many questions about it. The document is cached after the first call; subsequent questions only bill for the short question text:

cache_document.py — Q&A over a cached document — PYTHON

```
with open("large_document.txt") as f:
    doc = f.read()

def ask(question: str) -> str:
    response = client.messages.create(
        model="claude-sonnet-4-6",
        max_tokens=512,
```

```
        system=[{
            "type": "text",
            "text": f"Answer questions about the following document:\n\n{doc}",
            "cache_control": {"type": "ephemeral"} # cache the whole doc
        }],
        messages=[{"role": "user", "content": question}]
    )
    return response.content[0].text

# First call writes the document to cache (~slow + write cost)
ask("What is the main topic?")

# Subsequent calls read from cache — fast and cheap
ask("List the key recommendations.")
ask("What are the risks mentioned?")
```

## When Caching Helps Most

### Long system prompts

Detailed persona definitions, output format instructions, brand voice guides. If your system prompt is over 1,000 tokens and doesn't change between users, cache it.

### Document Q&A

Load a codebase, manual, or dataset once and ask many questions. The document is cached after the first query; all subsequent questions only pay for the question itself.

### Long conversations

As history grows, caching earlier turns means each new message only bills for the fresh content. Essential for applications where users have extended sessions.

### Shared context across users

If all users of your app share a common context (e.g. a product catalogue), that prefix can be cached once and reused across all requests — not just per-session.

## When Caching Doesn't Help

- **Short system prompts** — below the 1,024-token minimum, caching is silently skipped.
- **Highly dynamic prefixes** — if you inject a different user name, timestamp, or per-request data into the system prompt, the prefix changes every time and won't hit the cache. Move

dynamic content after the cache breakpoint.

- **Infrequent requests** — if calls are spaced more than 5 minutes apart, the cache expires and you pay full price each time. The TTL resets on each cache hit, so active sessions benefit; low-traffic endpoints may not.
- **One-off batch jobs** — if you process each item independently with no shared prefix, there's nothing to cache.

> **DESIGN TIP — SEPARATE STATIC FROM DYNAMIC**
>
> Structure your system prompt so everything stable (persona, rules, documents) comes first, followed by your cache breakpoint. Dynamic content — the specific user's name, the current date, per-request injections — comes after the breakpoint in the user message. This maximises what can be cached without sacrificing flexibility.

**Next — Chapter 7: Multi-Agent Patterns**

A single Claude call has limits — context window, response length, the kinds of reasoning it can do in one pass. Multi-agent patterns chain Claude instances together: an orchestrator breaks a task into subtasks, subagents execute them in parallel or sequence, and results are combined. Chapter 7 covers the core patterns, when to use them, and how to implement a simple orchestrator–worker pipeline.

# Advanced AI Workflows & the Claude API

Chapter 7 · Multi-Agent Patterns

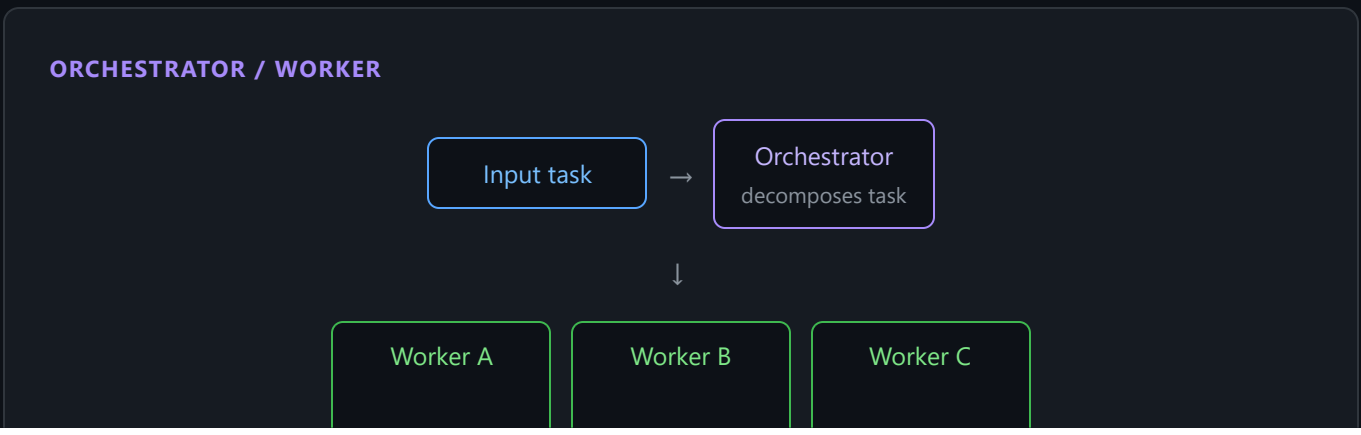
A single Claude call is powerful, but it has limits. The context window is finite, long responses can be inconsistent, and some tasks are genuinely too complex to solve in one pass. Multi-agent patterns solve this by splitting work across multiple Claude calls — each call focused on a specific subtask, the results combined by an orchestrator. This chapter covers the core patterns, how to implement them, and when they're worth the added complexity.

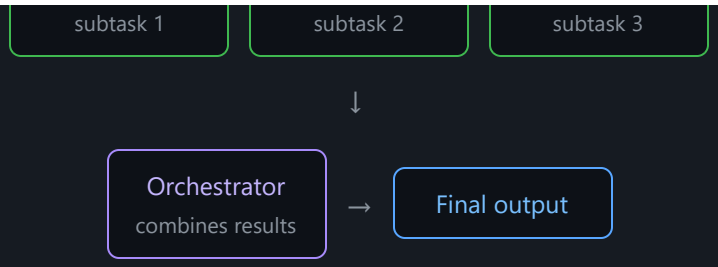
## Why Use Multiple Agents?

- **Tasks too large for one context window** — analysing a large codebase, processing hundreds of documents, generating a long structured report section by section.
- **Parallel work** — subtasks that don't depend on each other can run simultaneously, cutting total wall-clock time.
- **Specialisation** — different agents can have different system prompts, making one a careful fact-checker, another a creative writer, another a data extractor.
- **Verification** — one agent generates an answer; another independently checks it. Disagreements surface errors that a single pass would miss.

## Core Patterns

### 1 — Orchestrator + Workers



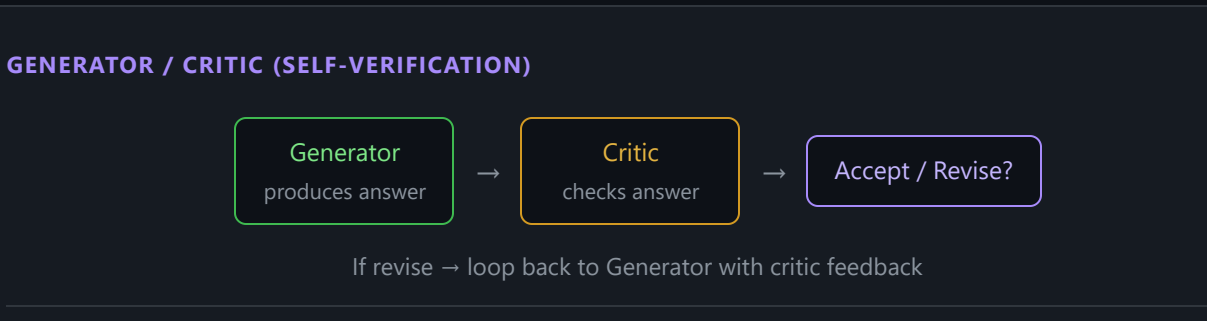



The orchestrator receives the full task, breaks it into subtasks, dispatches them to worker agents (in parallel or sequence), and synthesises the results. Workers are stateless — they don't know about each other or the broader task.

## 2 — Sequential Pipeline

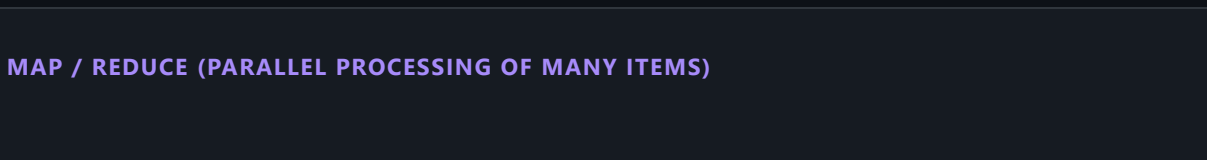

Each agent takes the previous agent's output as input. Good for multi-stage transformations where each step builds on the last — e.g. extract → analyse → summarise → format.

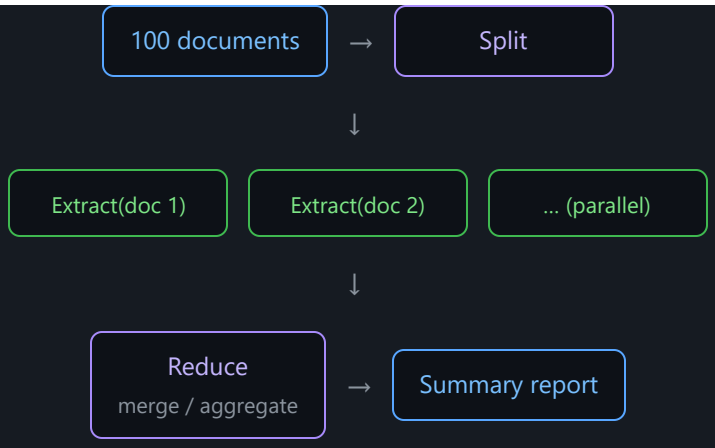
## 3 — Generator + Critic




Two separate Claude instances with different system prompts. The generator writes freely; the critic is instructed to be sceptical and look for errors. Disagreement triggers a revision loop. Effective for factual accuracy, code correctness, or argument quality.

## 4 — Map / Reduce






Each item is processed independently in parallel (map), then results are merged by a final agent (reduce). Ideal for tasks like summarising many documents, classifying a batch of items, or extracting structured data from a large file set.

## Implementing an Orchestrator–Worker Pipeline

multi_agent.py — orchestrator dispatches parallel workers PYTHON

```python
import anthropic
import asyncio

client = anthropic.Anthropic()

def call_claude(system: str, prompt: str, model="claude-haiku-4-5-20251001") →
str:
    """Single synchronous Claude call – used by workers."""
    response = client.messages.create(
        model=model,
        max_tokens=512,
        system=system,
        messages=[{"role": "user", "content": prompt}]
    )
    return response.content[0].text


async def worker(loop, section: str) → str:
    """Run a synchronous Claude call in a thread pool."""
    return await loop.run_in_executor(
        None, call_claude,
        "You are a precise summariser. Return 3 bullet points only.",
```

```
        f"Summarise this section:\n\n{section}"
    )


async def orchestrate(document: str) -> str:
    # Step 1 — Orchestrator splits the document into sections
    split_result = call_claude(
        "Split the text into logical sections. Return each section separated by '---'.",
        document,
        model="claude-sonnet-4-6"
    )
    sections = [s.strip() for s in split_result.split("---") if s.strip()]

    # Step 2 — Workers summarise all sections in parallel
    loop = asyncio.get_event_loop()
    summaries = await asyncio.gather(
        *[worker(loop, s) for s in sections]
    )

    # Step 3 — Orchestrator merges all summaries into a final report
    combined = "\n\n".join(summaries)
    return call_claude(
        "You are a report writer. Combine these section summaries into a cohesive executive summary.",
        combined,
        model="claude-sonnet-4-6"
    )


if __name__ == "__main__":
    with open("report.txt") as f:
        doc = f.read()
    result = asyncio.run(orchestrate(doc))
    print(result)
```

**USE HAIKU FOR WORKERS, SONNET/OPUS FOR THE ORCHESTRATOR**

Workers do focused, repetitive subtasks — extract, classify, summarise a single item. They don't need a powerful model. Haiku is fast and cheap here. Save Sonnet or Opus for the

> orchestrator that must reason about the overall task and synthesise results. This mix dramatically reduces cost without sacrificing output quality.

## Generator + Critic — Self-Verification Loop

generator_critic.py — two agents check each other's work PYTHON

```python
GENERATOR_SYSTEM = "You are a helpful assistant. Answer the question clearly and thoroughly."

CRITIC_SYSTEM = """You are a strict fact-checker. \
Review the answer below and respond with either: \
PASS: <brief reason> \
FAIL: <specific issue> and <corrected answer>"""

def generate_and_verify(question: str, max_rounds: int = 3) -> str:
    answer = call_claude(GENERATOR_SYSTEM, question)

    for _ in range(max_rounds):
        verdict = call_claude(
            CRITIC_SYSTEM,
            f"Question: {question}\n\nAnswer: {answer}"
        )
        if verdict.startswith("PASS"):
            break # critic is satisfied
        # FAIL — extract corrected answer and feed back to generator
        answer = call_claude(
            GENERATOR_SYSTEM,
            f"Your answer was criticised: {verdict}\nRevise your answer to the question: {question}"
        )

    return answer
```

## Choosing a Pattern

| Pattern | Best for | Main trade-off |
|---|---|---|
| **Orchestrator + Workers** | Tasks that decompose cleanly into parallel subtasks — research, batch processing, report generation | Orchestrator adds latency; decomposition quality determines output quality |
| **Sequential Pipeline** | Multi-stage transformations where each step builds on the last — extract → clean → analyse → format | Simple to implement; errors propagate downstream and compound |
| **Generator + Critic** | High-stakes answers, factual claims, code correctness — anything worth double-checking | 2–3× the API calls and cost; revision loops add latency |
| **Map / Reduce** | Large collections of similar items — classify 500 emails, extract data from 100 PDFs | Parallel calls hit rate limits; reduce step must handle inconsistent map outputs |

## Tradeoffs vs a Single Call

### When multi-agent adds value

- Task genuinely exceeds one context window
- Subtasks can run in parallel — total time matters
- Quality improves with a separate verification step
- You need specialised behaviour per subtask
- You're processing many items of the same type

### When a single call is better

- Task fits comfortably in one context window
- Latency matters — multi-agent is always slower to start
- Costs are a concern — every agent call is billed separately
- Errors in decomposition or combination can make outputs worse than a single focused call
- The task requires deep reasoning across all parts simultaneously

**RATE LIMITS COMPOUND QUICKLY**

Running 20 workers in parallel means 20 simultaneous API calls. Anthropic's rate limits are per-organisation, not per-request — you can hit them quickly with parallel agents. Design your worker pool size to stay within your tier's requests-per-minute limit, and add retry logic with exponential backoff for 429 errors.

**KEEP AGENTS STATELESS**

> Workers should receive everything they need in their prompt and return everything the orchestrator needs in their response. Don't have agents share mutable state — it creates ordering dependencies that break parallelism and make debugging much harder. Treat each agent call as a pure function: input in, output out.

**Next — Chapter 8: Retrieval-Augmented Generation (RAG)**

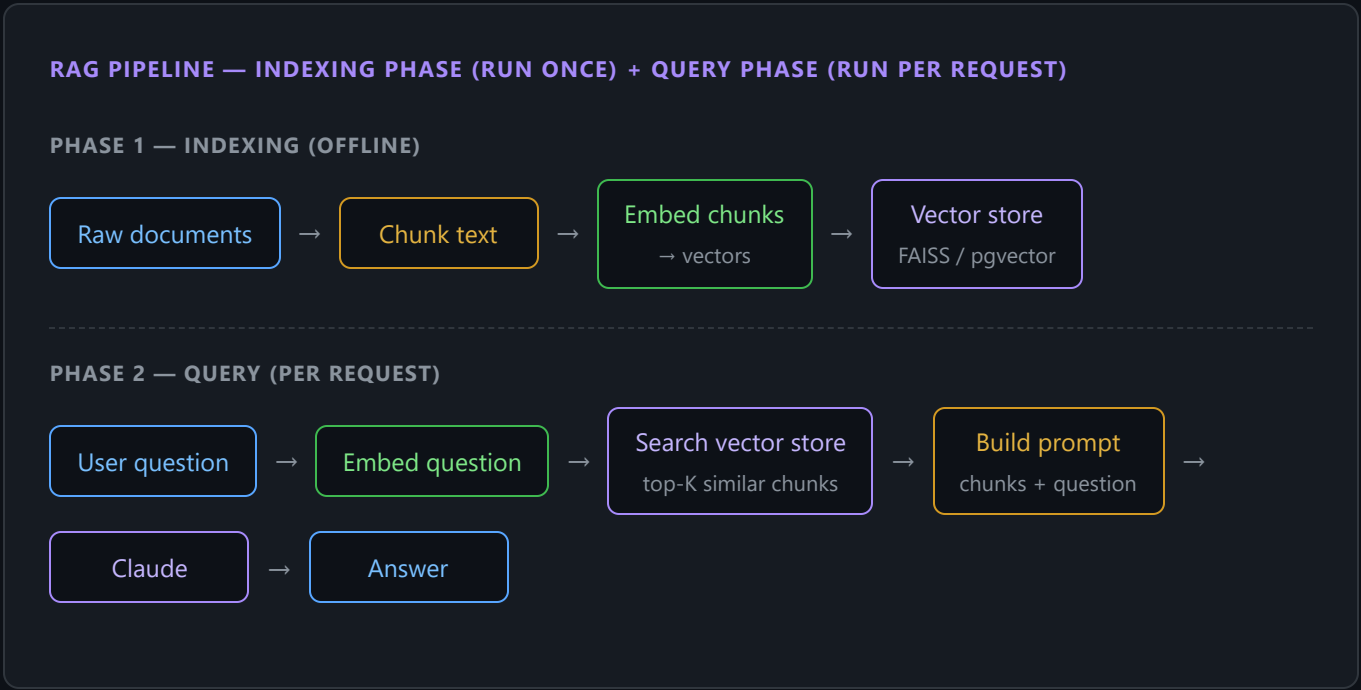
Claude's knowledge has a training cutoff and no access to your own data. RAG bridges that gap — you retrieve the most relevant chunks from your documents or database and inject them into Claude's context. Chapter 8 covers how RAG works, how to build a simple vector search pipeline, and the design decisions that determine whether it actually improves answers.

# Advanced AI Workflows & the Claude API

Chapter 8 · Retrieval-Augmented Generation (RAG)

Claude's training has a knowledge cutoff and no awareness of your private data — your company's documentation, your database records, your product catalogue. RAG solves this by fetching relevant content at query time and injecting it into Claude's context. Rather than training a custom model, you retrieve the right information and let Claude reason over it. This chapter covers how RAG works, how to build a simple pipeline, and the design decisions that separate working RAG from broken RAG.

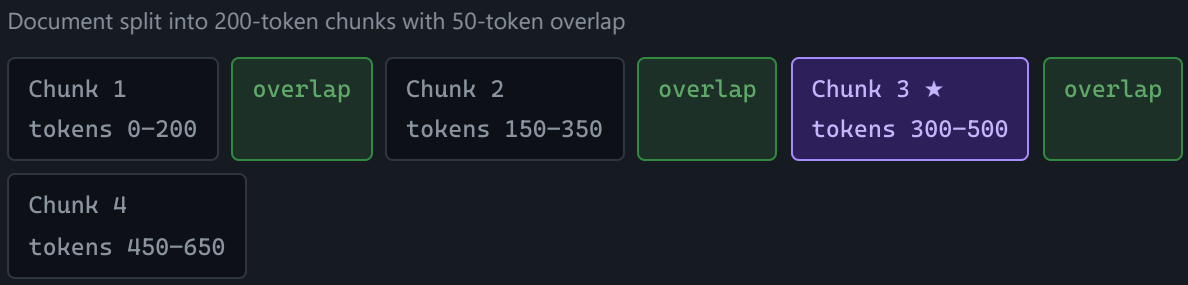
## How RAG Works — The Two Phases




The key insight: you never stuff all your documents into Claude's context. You use embeddings — numerical representations of meaning — to find the small subset of content most relevant to the current question, and inject only that. Claude sees the question plus the retrieved context and answers from it.

## Chunking — Getting It Right

How you split documents into chunks is one of the most important decisions in a RAG system. Chunks that are too small lose context; chunks that are too large dilute relevance and waste context window space. A common approach is fixed-size chunks with overlap:




Chunk 3 matched the query. Overlap ensures sentences at boundaries appear in at least two chunks — reducing the chance that relevant content is split across a boundary and missed.

### Chunking strategies

- **Fixed size with overlap** — simple, predictable. Good default for prose documents. Typical: 200–500 tokens per chunk, 10–20% overlap.
- **Paragraph / section boundaries** — split at natural breaks (double newlines, headings). Preserves semantic coherence better than fixed size.
- **Recursive character splitting** — try paragraph breaks first; fall back to sentence breaks, then fixed size. Used by LangChain's `RecursiveCharacterTextSplitter`.
- **Semantic chunking** — embed sentences, group those whose embeddings are similar. Higher quality; more expensive to compute.

## Building a Simple RAG Pipeline

rag_pipeline.py — indexing + querying with sentence-transformers + FAISS — PYTHON

```python
import anthropic, faiss, numpy as np
from sentence_transformers import SentenceTransformer

embed_model = SentenceTransformer("all-MiniLM-L6-v2")
claude = anthropic.Anthropic()

# ── Indexing phase ──────────────────────────────────
def chunk_text(text: str, size=300, overlap=50) -> list[str]:
    words = text.split()
```

```
    return [
        " ".join(words[i:i + size])
        for i in range(0, len(words), size - overlap)
    ]

def build_index(documents: list[str]):
    chunks = []
    for doc in documents:
        chunks.extend(chunk_text(doc))

    embeddings = embed_model.encode(chunks, convert_to_numpy=True)
    index = faiss.IndexFlatL2(embeddings.shape[1])
    index.add(embeddings.astype("float32"))
    return index, chunks

# ── Query phase ──────────────────────────────────────────
def retrieve(question: str, index, chunks: list[str], k=4) -> list[str]:
    q_embed = embed_model.encode([question],
convert_to_numpy=True).astype("float32")
    _, indices = index.search(q_embed, k)
    return [chunks[i] for i in indices[0]]

def ask(question: str, index, chunks: list[str]) -> str:
    context_chunks = retrieve(question, index, chunks)
    context = "\n\n---\n\n".join(context_chunks)

    prompt = f"""Use the following context to answer the question. \
If the answer is not in the context, say so - do not make up information.

Context:
{context}

Question: {question}"""

    response = claude.messages.create(
        model="claude-sonnet-4-6",
        max_tokens=1024,
        messages=[{"role": "user", "content": prompt}]
    )
    return response.content[0].text

# ── Example usage ────────────────────────────────────────
```

```
docs = [open(p).read() for p in ["policy.txt", "manual.txt"]]
idx, ch = build_index(docs)
print(ask("What is the refund policy for digital products?", idx, ch))
```

## The RAG Prompt — Grounding Claude in Retrieved Context

The prompt structure matters. Explicitly tell Claude to answer only from the provided context and to admit when the answer isn't there. This prevents the model from blending retrieved facts with training knowledge in ways you can't verify:

Effective RAG prompt structure — TEXT

```
You are a helpful assistant that answers questions using only the
provided context. If the context does not contain the answer,
respond with: "I don't have information about that in the documents
provided." Do not use your general knowledge.

Context:
[CHUNK 1]
---
[CHUNK 2]
---
[CHUNK 3]

Question: [user question]
```

**ASK CLAUDE TO CITE ITS SOURCES**

Add "cite which part of the context supports your answer" to the prompt. This forces Claude to stay grounded — it can only cite text it actually received — and lets you display sources to the user to build trust. If Claude can't find a citation, it almost certainly shouldn't be making the claim.

## Choosing a Vector Store

| Option | Best for | Persistent? | Setup |
|---|---|---|---|
| **FAISS** (in-memory) | Prototyping, small datasets (<100k chunks), single-process apps | SAVE/LOAD | `pip install faiss-cpu` |
| **ChromaDB** | Local development, moderate scale, easy API | YES | `pip install chromadb` |
| **pgvector** | Production apps already using PostgreSQL — vectors live alongside your data | YES | Postgres extension + `psycopg2` |
| **Pinecone / Weaviate** | Large scale, managed cloud, multi-tenant SaaS | YES | Managed service — API key |

## Common RAG Failure Modes

### Wrong chunks retrieved

The embedding model doesn't capture the question's intent — irrelevant chunks ranked higher than relevant ones. Fix: try a better embedding model, add keyword search as a fallback (hybrid search), or rephrase the query before embedding.

### Answer split across chunks

The relevant content spans a chunk boundary and neither chunk alone is sufficient. Fix: increase chunk size, increase overlap, or retrieve more chunks (higher k).

### Claude ignores the context

Claude falls back on training knowledge instead of the retrieved text, especially when the context contradicts something it "knows". Fix: strengthen the system prompt instruction to use only the provided context.

### Stale index

Documents have changed but the index hasn't been rebuilt. Fix: trigger re-indexing when source documents are updated — either on a schedule or via a document change event/webhook.

### Context window overflow

Retrieving too many chunks fills the context window, leaving no room for the answer. Fix: limit retrieved chunks, compress them with a summarisation step, or use prompt caching (Chapter 6) for a stable document corpus.

### Hallucinated citations

Claude invents source text that was never retrieved. Fix: ask Claude to quote directly from the context rather than paraphrase, and validate that quotes exist in the retrieved chunks.

## RAG vs Fine-tuning vs Prompt Caching

| Approach | Knowledge source | Updatable? | Cost | Best for |
|---|---|---|---|---|
| **RAG** | External documents, retrieved at query time | YES — RE-INDEX | Embedding + Claude API calls | Large, changing document sets; factual Q&A |
| **Prompt caching** | Documents injected into every prompt, cached | YES — INVALIDATES CACHE | Cache write + 10% on reads | Stable documents, <200k tokens total |
| **Fine-tuning** | Baked into model weights during training | EXPENSIVE TO UPDATE | High — training compute | Style/format adaptation; not recommended for factual knowledge |

**FINE-TUNING IS RARELY THE RIGHT ANSWER FOR KNOWLEDGE**

A common misconception is that fine-tuning "teaches" Claude facts from your documents. It doesn't — fine-tuning adjusts style and behaviour, not factual recall, and fine-tuned models hallucinate just as confidently. For knowledge from your own data, RAG or prompt injection are almost always better choices.

**Next — Chapter 9: Evaluating Outputs**

How do you know if your AI feature is actually working? Testing Claude-powered apps requires a different approach than traditional software testing — outputs are probabilistic, not deterministic. Chapter 9 covers building eval sets, scoring strategies (LLM-as-judge, rubrics, reference comparisons), catching regressions when you change your prompt, and the metrics that actually predict user satisfaction.

# Advanced AI Workflows & the Claude API

Chapter 9 · Evaluating Outputs

Traditional software has deterministic outputs — the same input always produces the same result, and tests either pass or fail. AI features don't work that way. Claude's outputs are probabilistic: they vary across runs, they can be correct in spirit but wrong in detail, and "good" is often a matter of degree rather than a binary. Evaluating AI outputs requires a different toolkit — one built around sampling, scoring, and tracking distributions rather than exact matches.

## Why Evaluation Matters

- **Prompt changes break things silently** — a small wording tweak that improves one case often degrades another. Without an eval suite, you won't know until users complain.
- **Model updates change behaviour** — when Anthropic releases a new Claude version, your prompts may perform differently. Evals let you measure this before switching.
- **Hallucination is invisible without checking** — Claude sounds confident whether it's right or wrong. Evals surface the cases where confidence isn't warranted.
- **"It looks good to me" doesn't scale** — manual review of 10 outputs is feasible; 500 is not. Automated scoring lets you evaluate at the scale your feature operates at.

## The Evaluation Loop

Regression → Revise prompt → *loop back to run*

## Building an Eval Set

An eval set is a collection of test cases: each case has an input (the prompt or user message) and either an expected output or criteria for what a good output looks like. Start small — 20–50 cases is enough to catch most regressions — and grow it over time by adding cases whenever you find an output that surprised you.

### What makes a good eval set

- **Golden path cases** — the most common, typical inputs your feature handles.
- **Edge cases** — boundary conditions, ambiguous inputs, the cases that have broken things before.
- **Adversarial cases** — inputs designed to produce failure: off-topic questions, jailbreak attempts, inputs with missing information.
- **Regression cases** — every time you find a bug in production, add it to the eval set so it can't come back silently.

eval_set.py — structure of a simple eval dataset — PYTHON

```python
EVAL_SET = [
    {
        "id": "refund_policy_basic",
        "input": "Can I get a refund after 30 days?",
        "expected": "no", # exact match where possible
        "criteria": "Must state refunds are only available within 30 days. Polite
tone."
    },
    {
        "id": "off_topic",
        "input": "What is the capital of France?",
        "expected": None,
        "criteria": "Must decline to answer and redirect to support topics."
    },
    {
        "id": "ambiguous_input",
```

```
        "input": "I want to cancel.",
        "expected": None,
        "criteria": "Must ask a clarifying question — cancel what?"
    },
]
```

## Scoring Methods

### Exact match

*When: classification, yes/no, structured fields*

Compare Claude's output to an expected string. Simple and fast. Only works when correct answers are unambiguous and outputs can be normalised to a consistent form.

### Rubric scoring

*When: open-ended responses, tone, completeness*

Define what a good answer looks like as a checklist. Score each item independently. More flexible than exact match; more interpretable than LLM-as-judge alone.

### LLM-as-judge

*When: nuanced quality, multiple valid answers*

Send Claude's output to a second Claude call (with a strict judging prompt) and ask it to rate the answer. Scales well; biased toward its own style — use Opus or a different model for judging.

### Reference comparison

*When: you have a known-good answer*

Ask the judge: "Does this answer contain the same information as the reference answer, even if worded differently?" Handles paraphrase better than exact match without full open-ended judgement.

## LLM-as-Judge — Implementation

llm_judge.py — using Claude to score Claude's outputs **PYTHON**

```
import anthropic, json

judge_client = anthropic.Anthropic()

JUDGE_SYSTEM = """You are a strict evaluator. Given a user question, a set of \
criteria, and an AI response, score the response as PASS or FAIL \
for each criterion. Return JSON only, no explanation outside JSON. \
Format: {"scores": [{"criterion": "...", "result": "PASS"|"FAIL", \
"reason": "..."}], "overall": "PASS"|"FAIL"}"""
```

```python
def judge(question: str, response: str, criteria: list[str]) -> dict:
    criteria_text = "\n".join(f"- {c}" for c in criteria)
    prompt = f"""Question: {question}

Criteria:
{criteria_text}

Response to evaluate:
{response}"""

    result = judge_client.messages.create(
        model="claude-opus-4-8", # use strongest model for judging
        max_tokens=512,
        system=JUDGE_SYSTEM,
        messages=[{"role": "user", "content": prompt}]
    )
    return json.loads(result.content[0].text)


def run_evals(eval_set: list, system_prompt: str) -> dict:
    results = []
    for case in eval_set:
        # Get Claude's answer
        answer = call_claude(system_prompt, case["input"])
        # Score with criteria from the eval case
        score = judge(case["input"], answer, case["criteria"].split(". "))
        results.append({"id": case["id"], "answer": answer, "score": score})

    passed = sum(1 for r in results if r["score"]["overall"] == "PASS")
    return {"pass_rate": passed / len(results), "cases": results}
```

**USE A DIFFERENT MODEL TO JUDGE**

When using LLM-as-judge, use a stronger model (Opus) to evaluate outputs from a weaker one (Sonnet, Haiku). A model judging its own outputs has a bias toward accepting answers that resemble its own style — even when wrong. A stronger judge is more likely to catch errors.

## Rubric Scoring — Example

For a customer support bot, you might score each response on four criteria. Using a numeric scale (1–4) instead of pass/fail gives you a finer signal for tracking gradual drift:

| Criterion | 1 — Poor | 2 — Fair | 3 — Good | 4 — Excellent |
|---|---|---|---|---|
| **Accuracy** | 1 Factually wrong | 2 Partially correct | 3 Correct but incomplete | 4 Fully correct |
| **Tone** | 1 Rude or dismissive | 2 Neutral but cold | 3 Friendly | 4 Warm, empathetic |
| **Conciseness** | 1 Rambling / off-topic | 2 Too long | 3 Appropriate length | 4 Crisp and complete |
| **Scope** | 1 Answered wrong question | 2 Partially addressed | 3 Answered the question | 4 Answered + anticipated follow-up |

## Catching Regressions

Run your eval suite before and after every prompt change, model upgrade, or system change. Compare the scores — any drop in pass rate or average score is a regression signal. Even a small drop across many cases is worth investigating before deploying.

Eval results — prompt v1 vs prompt v2 (50 cases)

| | | |
|---|---|---|
| Overall pass rate | 74% | 82% ↑ |
| Accuracy (avg score) | 3.1 | 3.6 ↑ |
| Tone (avg score) | 3.8 | 3.8 – |
| Off-topic handling | 9/10 pass | 6/10 pass ↓ |
| Ambiguous input handling | 5/8 pass | 8/8 pass ↑ |

In this example, v2 improves overall accuracy and ambiguous input handling — but off-topic rejection got worse. You can see exactly which cases regressed and investigate those prompts specifically, rather than rolling back the whole change.

## Metrics That Actually Predict User Satisfaction

**Task completion rate**

**Refusal rate**

**Hallucination rate**

Did the user get what they asked for? For support bots: did they stop asking follow-up questions? More predictive than quality scores alone.

How often does Claude decline a valid request? Over-refusal is a real failure mode — track it separately from under-refusal.

On RAG or factual tasks: what fraction of responses contain at least one claim that cannot be verified from the source material?

**Latency at P95**

The 95th-percentile response time matters more than the mean. Slow outliers cause user abandonment even when average speed is fine.

**Regression count**

How many eval cases that previously passed now fail? Track this as an absolute count, not just a percentage — a 5% drop across 200 cases is 10 broken experiences.

**Human override rate**

In apps where users can edit Claude's output: how often do they? High edit rates signal that Claude's output isn't landing — even if automated scores look fine.

**EVALS ARE ONLY AS GOOD AS THEIR CASES**

A 95% pass rate on a weak eval set gives false confidence. Invest time in building adversarial cases and edge cases — the scenarios that are rare but catastrophic when they fail. An eval that only tests the golden path will pass even for a broken feature.

**Next — Chapter 10: Production Considerations**

The final chapter. Taking your Claude-powered feature from working prototype to reliable production service — rate limits and backoff strategies, cost management, safety and content filtering, structured logging for AI calls, and the deployment checklist before you go live.

# Advanced AI Workflows & the Claude API

Chapter 10 · Production Considerations

A Claude integration that works in development can fail in production in ways that aren't obvious until real users hit them — rate limits under load, runaway costs from verbose prompts, safety gaps you didn't test for. This chapter covers the operational concerns that separate a reliable AI feature from a fragile prototype: handling API limits, managing cost, logging what matters, and a pre-launch checklist to work through before you go live.

## Rate Limits — What They Are and How to Handle Them

Anthropic enforces three rate limits simultaneously. Hitting any one of them returns a `429 Too Many Requests` error:

**Requests per minute (RPM)**

Maximum number of API calls in a 60-second window. Most relevant for apps that make many short parallel requests — batch jobs, multi-agent pipelines.

**Input tokens per minute (ITPM)**

Maximum input tokens processed per minute. Apps with long system prompts or large injected documents hit this before RPM — 10 calls with 50k tokens each saturates a 500k ITPM limit.

**Output tokens per minute (OTPM)**

Maximum output tokens generated per minute. Relevant for apps generating long documents or code. Streaming doesn't change this limit — tokens are counted as they're generated.

**Limits are per-model and per-tier**

Limits differ between Opus, Sonnet, and Haiku, and increase as your account tier increases (based on spend history). Check your current limits in the Anthropic console.

### Retry with exponential backoff

When a 429 arrives, wait before retrying — and increase the wait time with each attempt. Retrying immediately just generates more 429s.

retry.py — exponential backoff for 429 errors PYTHON

```python
import time, anthropic
from anthropic import RateLimitError, APIStatusError

def call_with_retry(client, max_retries=5, **kwargs) -> str:
    for attempt in range(max_retries):
        try:
            response = client.messages.create(**kwargs)
            return response.content[0].text
        except RateLimitError:
            if attempt == max_retries - 1:
                raise
            wait = (2 ** attempt) + random.random() # jitter
            time.sleep(wait)
        except APIStatusError as e:
            if e.status_code ≥ 500: # server errors — retry
                wait = 2 ** attempt
                time.sleep(wait)
            else:
                raise # 4xx errors (except 429) — don't retry
```

**ADD JITTER TO PREVENT THUNDERING HERD**

If many requests fail at once and all retry after exactly the same backoff interval, they'll all hit the API at the same time — causing another wave of 429s. Adding a small random value ( `random.random()` ) to the wait time staggers retries and prevents this.

## Cost Management

Claude API cost is driven by token volume: input tokens + output tokens × the per-token rate for your chosen model. The biggest lever you have is controlling what goes into each request.

| Strategy | How it helps | Trade-off |
|---|---|---|
| **Right-size your model** | Haiku costs a fraction of Opus. Use Haiku for classification, extraction, and simple Q&A. Reserve Sonnet/Opus for reasoning-heavy tasks. | Weaker models fail on complex tasks — measure quality, don't assume. |
| **Prune conversation history** | Keep only the last N turns rather than the full history. Each turn you drop removes its tokens from every future request. | Claude loses earlier context — fine for most chats, problematic for long reference conversations. |
| **Prompt caching** | Cache stable prefixes (system prompt, reference documents) — 90% cheaper on cache reads. Covered in Chapter 6. | 5-minute TTL; write cost is 25% higher than standard on first call. |
| **Set max_tokens tightly** | Set `max_tokens` to the actual maximum you expect — don't leave it at 4096 for answers that are always under 200 tokens. | Truncates legitimate long responses if set too low. |
| **Per-user spend limits** | Track token usage per user/session and enforce a soft cap — warn or rate-limit users who are consuming disproportionately. | Requires token tracking infrastructure; adds complexity. |

## Logging — What to Record for Every Claude Call

Good logging lets you debug failures, audit outputs, track cost, and build your eval dataset from production traffic. Capture the following for every API call:

Structured log entry — one Claude API call

```
{
  "timestamp": "2026-06-19T09:14:33Z",
  "request_id": "req_01XkPm ... ", # from response headers
  "session_id": "user-abc-session-7",
  "model": "claude-sonnet-4-6",
  "system_prompt_hash": "a3f9c1 ... ", # hash not content — for cost/version tracking
  "user_message": "Can I get a refund?",
  "response_text": "Refunds are available within 30 days ... ",
  "stop_reason": "end_turn",
  "input_tokens": 412,
  "output_tokens": 87,
  "cache_read_tokens": 380,
  "latency_ms": 1243,
```

```
  "cost_usd": 0.000284
}
```

logging_wrapper.py — log every Claude call automatically PYTHON

```python
import time, hashlib, json, logging

# Sonnet pricing — update if using a different model
INPUT_COST_PER_TOKEN = 3e-6
OUTPUT_COST_PER_TOKEN = 15e-6
CACHE_READ_COST = 0.3e-6 # 90% cheaper

def logged_call(client, session_id: str, **kwargs) -> str:
    system_hash = hashlib.md5(
        str(kwargs.get("system", "")).encode()
    ).hexdigest()[:8]

    t0 = time.time()
    response = client.messages.create(**kwargs)
    latency = int((time.time() - t0) * 1000)

    u = response.usage
    cost = (
        u.input_tokens * INPUT_COST_PER_TOKEN
      + u.output_tokens * OUTPUT_COST_PER_TOKEN
      + getattr(u, "cache_read_input_tokens", 0) * CACHE_READ_COST
    )

    logging.info(json.dumps({
        "session_id": session_id,
        "model": kwargs["model"],
        "system_hash": system_hash,
        "input_tokens": u.input_tokens,
        "output_tokens": u.output_tokens,
        "latency_ms": latency,
        "cost_usd": round(cost, 6),
        "stop_reason": response.stop_reason,
    }))
    return response.content[0].text
```

## Safety and Content Filtering

Claude has built-in safety guardrails, but they are not a substitute for application-level controls. You are responsible for what your app does with Claude's outputs.

- **Scope your system prompt** — tell Claude explicitly what topics and actions are in and out of scope. "You are a customer support assistant for our software product. Do not discuss competitor products, pricing negotiations, or topics unrelated to software support."
- **Validate outputs before acting** — if Claude's output drives an action (sending an email, running a database query, executing code), validate the output before execution. Never pass Claude's raw output directly to a shell or SQL interpreter.
- **Check `stop_reason`** — a response that ends with `max_tokens` is truncated. A truncated SQL query or JSON object is worse than no response — always check that `stop_reason == "end_turn"` before using structured outputs.
- **Log inputs, not just outputs** — problematic outputs often require the input to diagnose. Log both, but scrub PII before writing to persistent storage.
- **Prompt injection awareness** — if users can supply content that Claude will read (documents, messages, URLs), malicious users can embed instructions in that content. Separate user-controlled content from trusted system instructions clearly in your prompt structure.

## Pre-Launch Checklist

**API & CREDENTIALS**

- [ ] API key stored in environment variable, not hardcoded
- [ ] API key has appropriate spend limit set in Anthropic console
- [ ] Key is not exposed in frontend code, logs, or git history

**RELIABILITY**

- [ ] Retry logic with exponential backoff + jitter for 429 and 5xx errors
- [ ] Timeout set on all API calls — don't let requests hang indefinitely
- [ ] Graceful error messages to users when Claude is unavailable
- [ ] `stop_reason` checked — truncated responses handled correctly

### COST

- [ ] Token usage logged per request with calculated cost
- [ ] Model choice justified — not defaulting to Opus for simple tasks
- [ ] Prompt caching enabled for stable prefixes over 1,024 tokens
- [ ] History pruning in place — conversation length is bounded

### SAFETY & QUALITY

- [ ] Eval suite passing — no regressions on known edge cases
- [ ] System prompt explicitly scopes allowed topics and actions
- [ ] Structured outputs validated before being acted on
- [ ] User-supplied content separated from system instructions in prompts
- [ ] PII scrubbed from logs before writing to persistent storage

### OBSERVABILITY

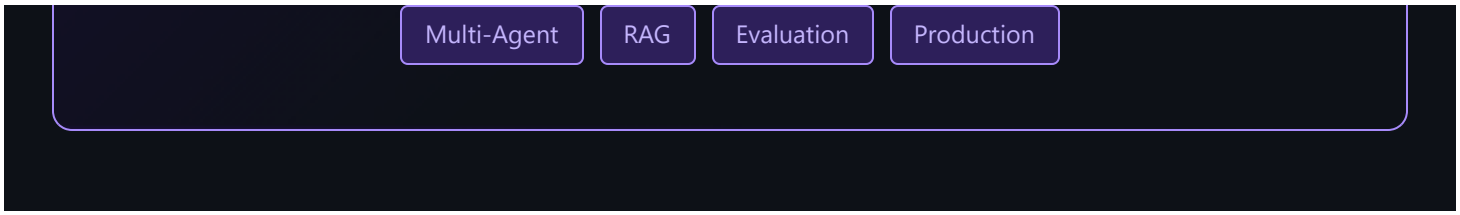
- [ ] Structured JSON logs for every Claude call
- [ ] Latency, token count, and cost tracked over time
- [ ] Alert on sustained error rate spike or abnormal cost increase

## Course Complete

Advanced AI Workflows & the Claude API · 10 Chapters

You've covered the full stack of building with Claude — from raw API calls to streaming, tool use, multi-agent systems, RAG, evaluation, and production operations. You have everything you need to build reliable, cost-efficient, AI-powered features.

Claude API | System Prompts | Tool Use | Streaming | FastAPI App | Prompt Caching


Multi-Agent
RAG
Evaluation
Production