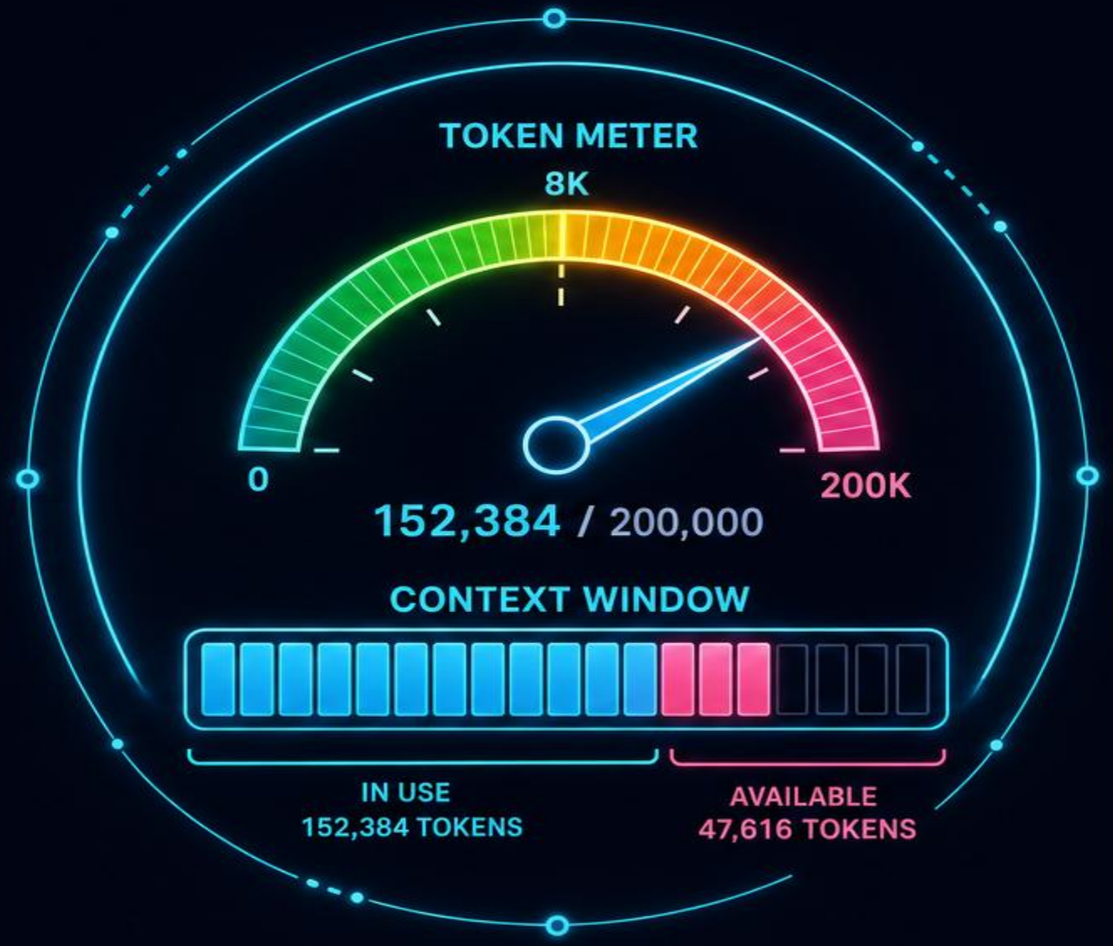
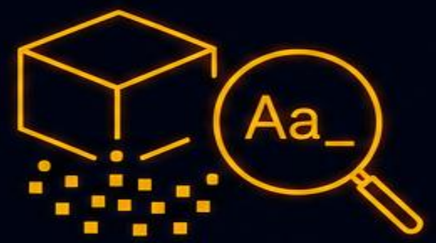
# Tokens and the Context Window

Almost everything you build with a model is measured in tokens.






# Three Core Definitions



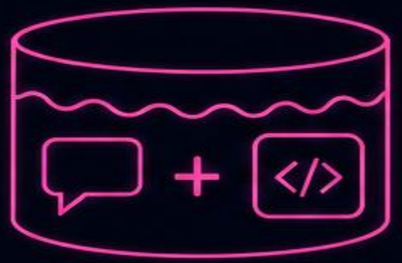

## Token

The atomic unit of text the model reads and generates, a sub-word chunk roughly a few characters long.



## Context Window

The maximum number of tokens the model can attend to in a single request, its total working space.




## Token Budget

Everything you send plus everything the model generates draws from one shared pool of tokens.

# Tokens Aren't Words

A token is a fragment of text, not a whole word.



**One token runs about four characters on average**

Token length is small and varies.
It's not the same as a character.



**A token averages about 0.55 words, a bit over half a word**

Most tokens are parts of words.
Some are full words, some are not.



**Counts shift with language, code, and formatting**

Different scripts, punctuation,
and whitespace change token counts.

**One sentence, many tokens**

| | | | | | |
|---|---|---|---|---|---|
| The | quick | brown | fox | jumps | over |
| the | lazy | dog | . | Amazing | ! |

**A model's structured response (parsed JSON)**

```
{
  "currency": "USD",
  "amount": 123.45,
  "items": [
    { "sku": "A1", "qty": 2, "price": 49.99 },
    { "sku": "B2", "qty": 1, "price": 23.47 }
  ],
  "tax": 9.87,
  "total": 133.32
}
```

**So a word count is only a loose proxy for a token count.**

# Counting Tokens Correctly



## Rough Estimate

About four characters, or roughly 0.55 words per token. Good for a quick ballpark estimate, nothing more.



## Authoritative Count

The count tokens endpoint returns the exact token number for the specific model you pass it.



## Avoid Guessing

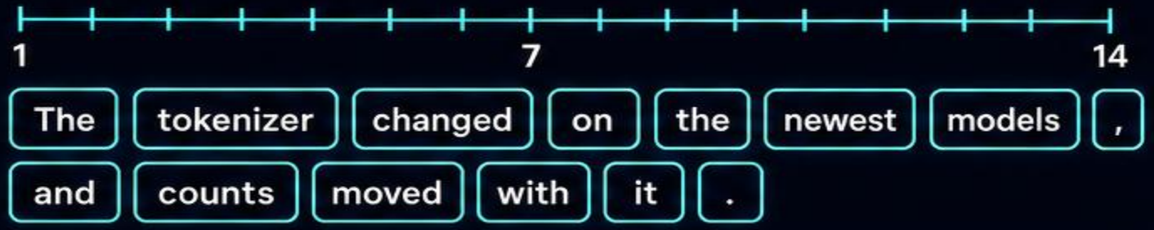
A fixed chars-per-token rule or another model's tokenizer drifts from Claude's real counts, worst on dense code.

# Tokenizers Are Model-Specific

The tokenizer changed on the newest models, and counts moved with it.

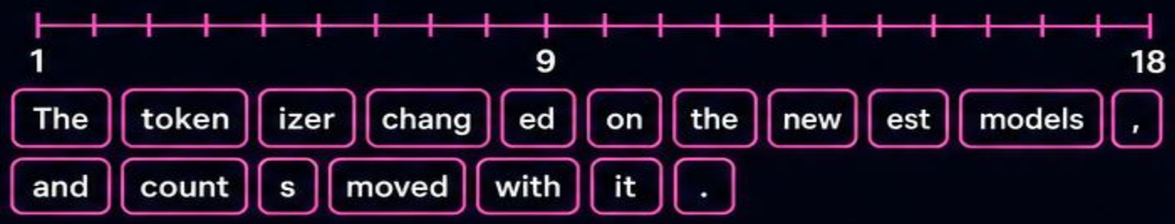
## Opus 4 (Old Tokenizer)

Token count: 14




## Opus 4.7 (New Tokenizer)

Token count: 18



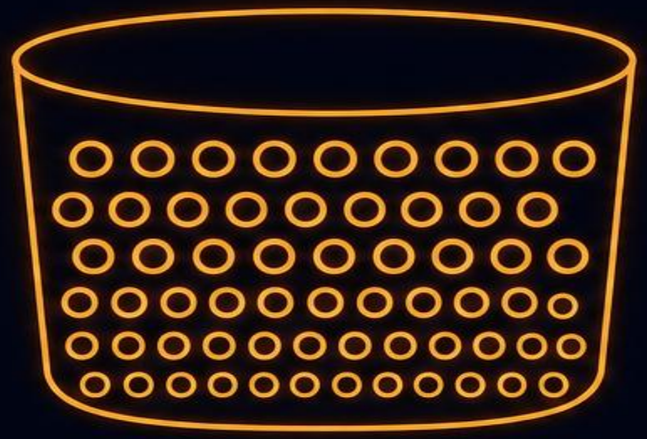

**Opus 4.7 and later,** plus **Fable 5** and **Sonnet 5,** split text more finely

**The same text** can produce about thirty percent more tokens

**Token counts** are model-specific, so re-check them per model

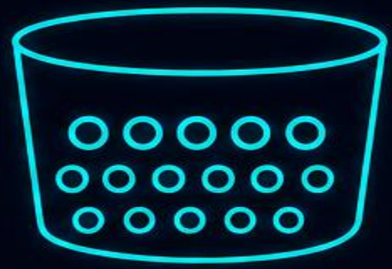
**Same text, different tokenizer, different count.**

# How Big Is the Window



## One Million Tokens

Fable 5, Opus 4.8, and Sonnet 5 attend to up to one million tokens, around 555,000 English words in a single request.



## Two Hundred Thousand

Haiku 4.5 attends to up to 200,000 tokens, around 150,000 words, so the same prompt has to fit a smaller budget.

# What Fills the Window

The window holds far more than your latest question.

**The system prompt and every tool definition**

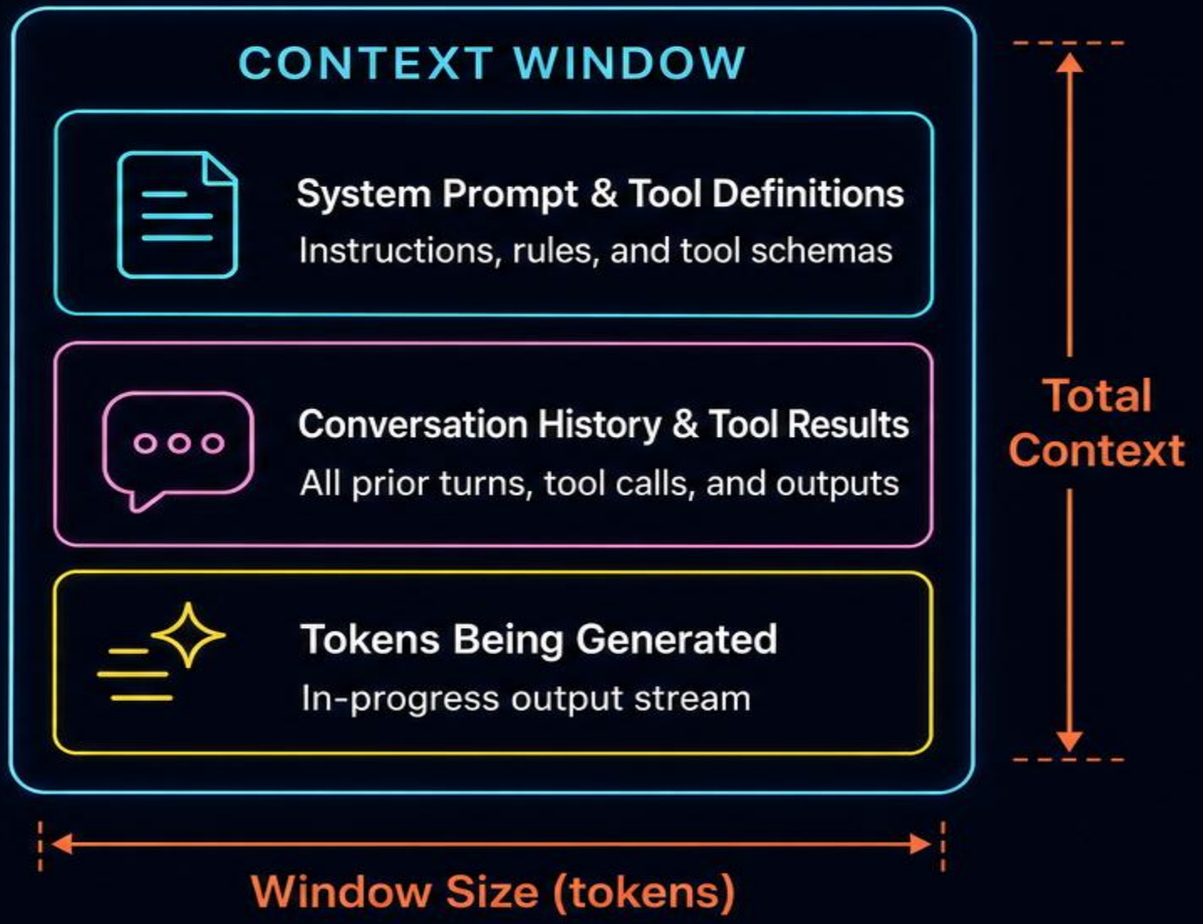
The instructions, rules, and tool schemas that shape the model.

**The full conversation history and tool results**

All prior turns, tool calls, and their outputs in context.

**The tokens the model is generating right now**

The in-progress output stream that's still being produced.




**Every one of these draws from the same shared budget.**

# Three Token Categories



## Input Tokens

Everything you send: the system prompt, history, tool definitions, and the current user turn on this call.



## Output Tokens

Everything the model generates, including any thinking, billed higher than input tokens.

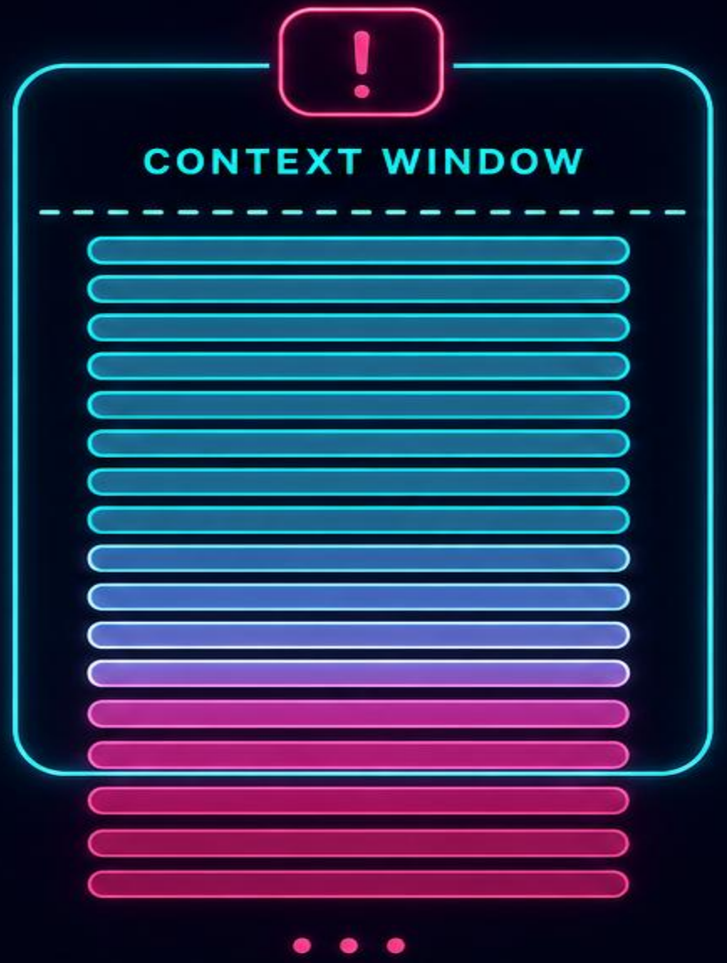


## Thinking Tokens

Reasoning tokens billed as output tokens, reported as a read-only breakdown inside the output total.

# When the Window Fills Up

Overflow is a problem you prevent, not one the model fixes for you.




**Exceeding the window** returns an error, not an auto-summary

**You prune or compact** history to stay under budget

**A full one-million-token** request bills at the standard rate

# Sizing the Token Budget



**A token** is a sub-word chunk, not a whole word



**The window** spans prompt, history, tools, and generation



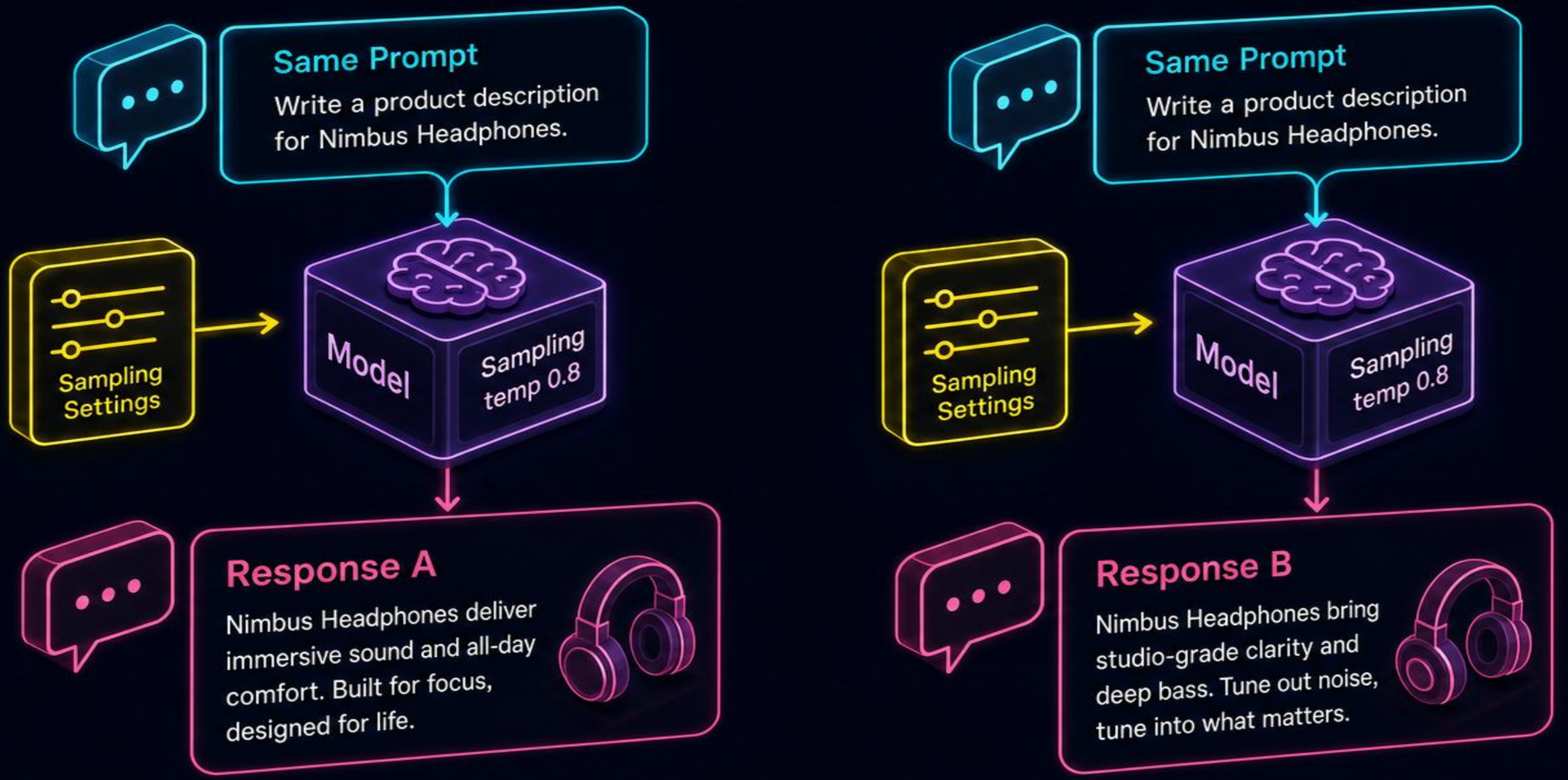

**Fable 5, Opus 4.8, and Sonnet 5** hold 1M tokens;
Haiku 4.5 holds 200k



**Count** with the endpoint, and re-baseline per model

# Sampling and Output Variation

Send the same prompt twice and you can get two different answers.



# Three Sampling Terms



## Sampling

The process of selecting each next token from the model's output probability distribution.




## Probability Distribution

The model's ranked guesses for the next token, each with a likelihood the sampler draws from.




## Non-Determinism

The property that the same prompt can yield different outputs across calls, so replies aren't byte-identical.

# How Sampling Once Worked

On older models, three controls shaped how the next token got picked.



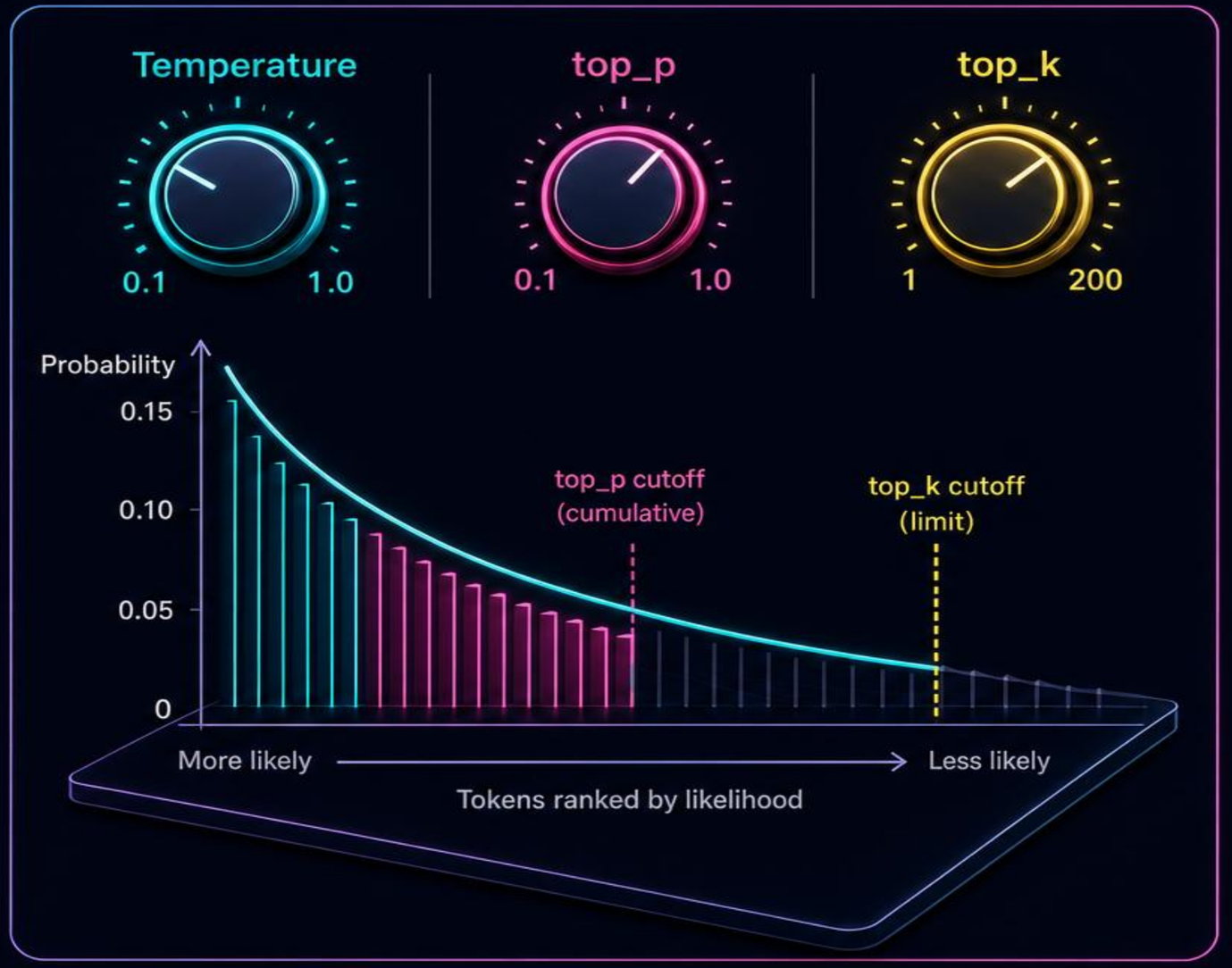
**Temperature**
flattened or sharpened
the odds



**top_p**
kept the most likely tokens
by cumulative probability



**top_k**
capped the pool to
a fixed number of candidates



# Removed on Current Models



## Rejected Parameters

Passing temperature, top_p, or top_k returns an HTTP 400 error and rejects the whole request outright.



## Which Models

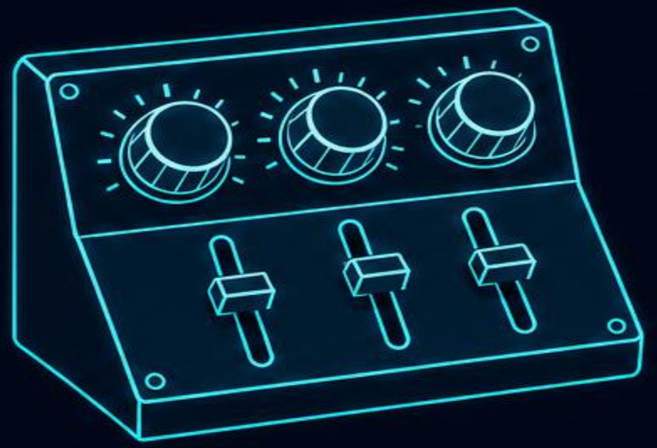
Fable 5, Opus 4.8, Opus 4.7, and Sonnet 5 no longer accept these sampling knobs.



## Steer Instead With

Shape behavior through clearer prompting and the effort setting now, not the old sampling parameters.

# Old Knobs vs New Levers



## Legacy Sampling Knobs

temperature, top_p, and top_k adjusted the randomness of sampling directly. They stay valid on older tiers like Sonnet 4.5, Haiku 4.5, and earlier models.



## Current Steering

Prompting and effort replace those knobs. On current models the old sampling parameters are rejected outright with an error, rather than quietly ignored as before.

# Same Prompt, Different Output

Even identical inputs can still produce different responses on different calls.

**IDENTICAL PROMPT**

Summarize this log line:
"2025-05-15 14:32:01 ERROR [payment] Charge failed for order 9f2a3b. Reason: card_declined"

**CALL #1**

The payment for order 9f2a3b failed because the customer's card was declined.

**What happened**

| | |
|---|---|
| Time | 14:37:21 |
| Region | us-east-1 |
| Route | shard-07 |
| Hardware | gpu-node-a |
| Config | serve-2411 |

**CALL #2**

Order 9f2a3b could not be charged because the card was declined.

**What happened**

| | |
|---|---|
| Time | 15:02:11 |
| Region | eu-west-1 |
| Route | shard-12 |
| Hardware | gpu-node-b |
| Config | serve-2416 |

Same prompt. Different context. Different output.
Both valid, both plausible.

### Next-token selection is probabilistic at its core

At each step, the model samples from a probability distribution. Small random differences compound into different text.

### Serving conditions like routing can shift behavior

Different times, regions, hardware, model shards, and configurations can all influence the output.

### Weights are fixed per model, yet the output still varies

The model parameters don't change within a version, but context and sampling decisions do.

So never assume that two separate calls return the exact same string.

# The Temperature-Zero Trap



## The Myth

Setting temperature to zero makes output fully deterministic and perfectly repeatable on every call.



## Never True

Even on older models that accepted it, zero temperature never guaranteed byte-identical outputs run to run.



## Rejected Now

On current models the temperature parameter is rejected with an error, so you cannot set it at all.

# Where Variation Comes From



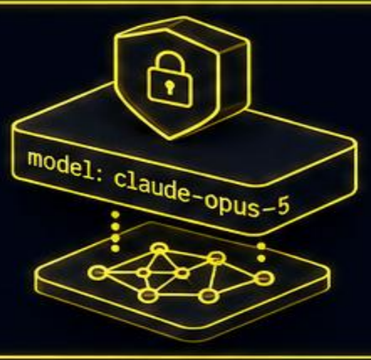
## Probabilistic Sampling

Each token is drawn from a probability distribution, so a lower-ranked token can still surface on any call.



## Serving Infrastructure

Routers, safety classifiers, and serving-side sampling logic can shift behavior without any change to the weights.




## Fixed Weights, Fluid Output

A pinned model ID keeps the weights constant, yet the observable output can still drift over time.

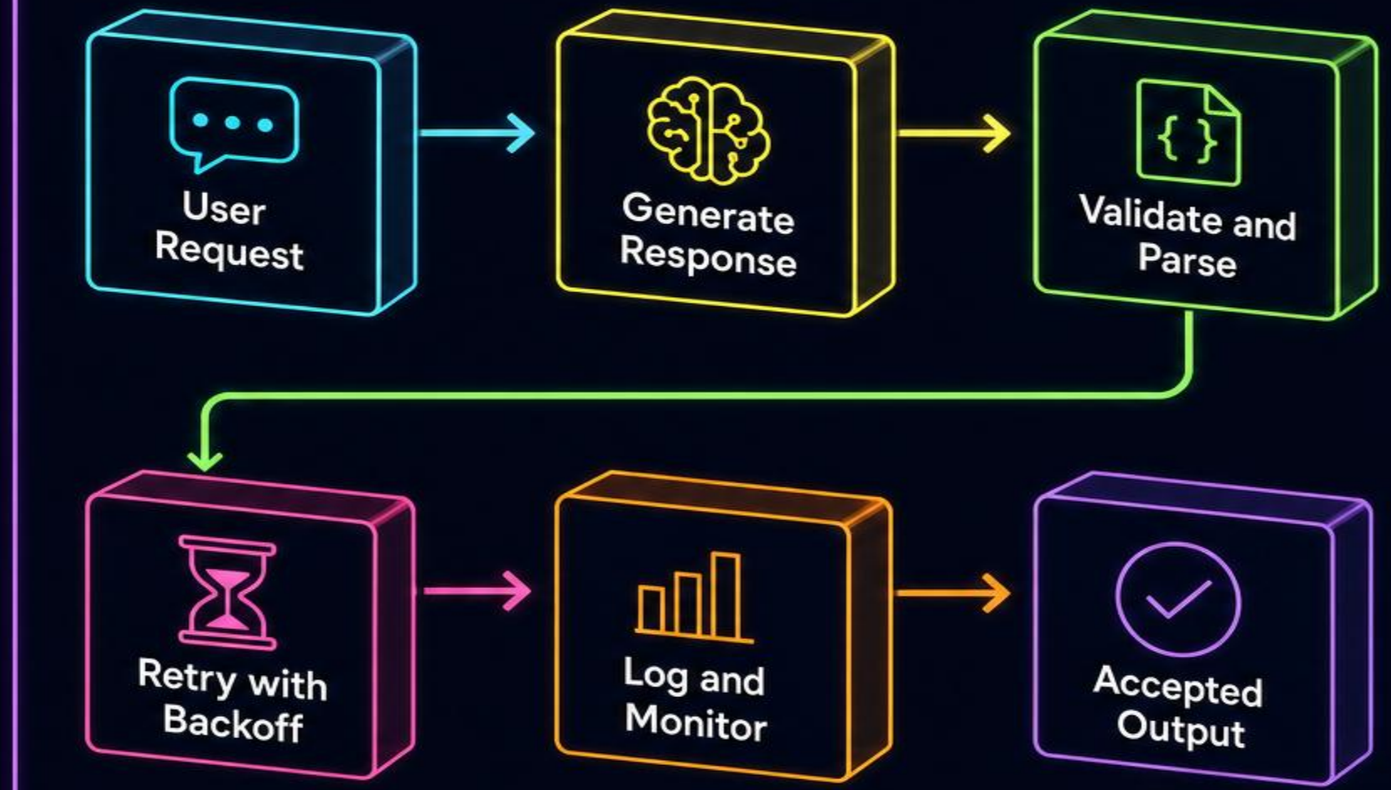
# Building for Non-Determinism

Design your pipelines to expect variation instead of fighting it.

**Validate and parse** every response defensively, every time

**Don't compare output** against one byte-identical expected string

**Add retries and checks** wherever an exact format matters



# Why Output Varies



**Legacy knobs** were temperature, top_p, and top_k



**Those parameters** return an error on current models



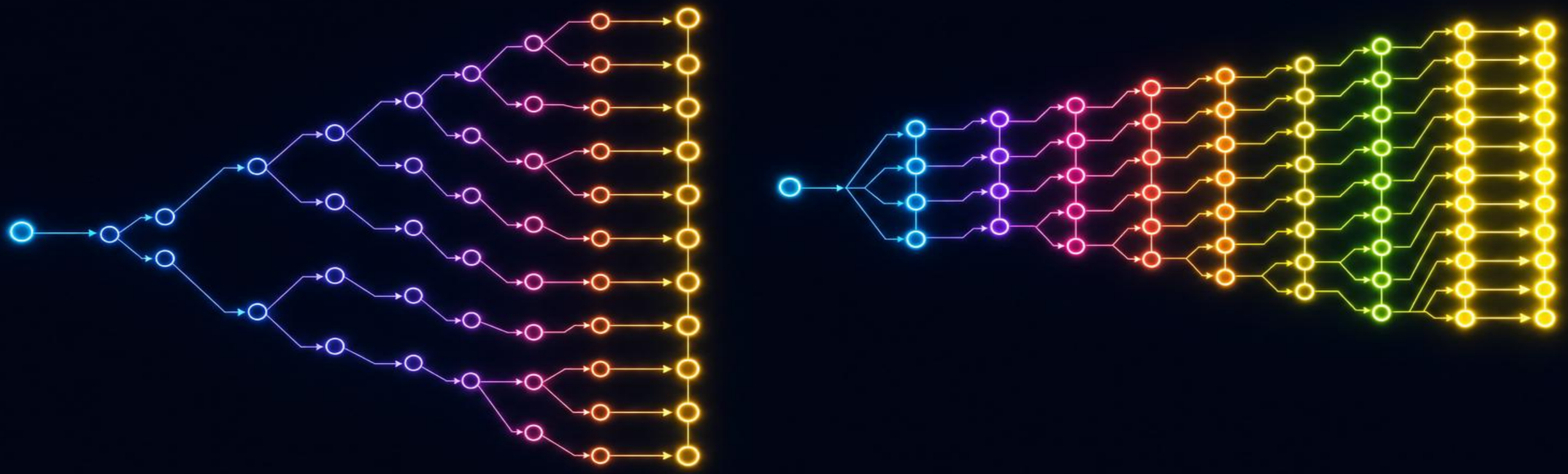
**Temperature zero** was never fully deterministic



**Validate and parse defensively** rather than expecting one string

# One Token at a Time

The model doesn't plan a full answer and then read it out.



# Three Generation Terms



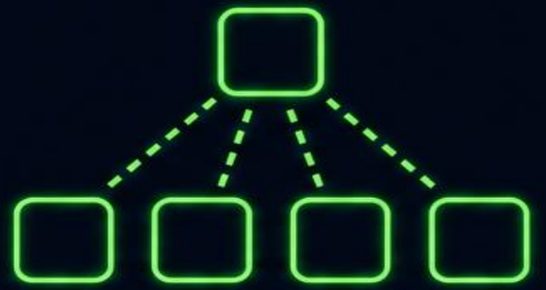
## Next-Token Generation

The mechanism by which the model produces text one token at a time until it stops.



## Autoregressive

Each new token is generated left to right, feeding back in as input for the next one.



## Conditioning

Every token is chosen based on all preceding tokens, including the model's own prior outputs.

# There Is No Lookahead

The model doesn't plan a full answer and then read it out.

### It emits one token, then reconsiders the sequence

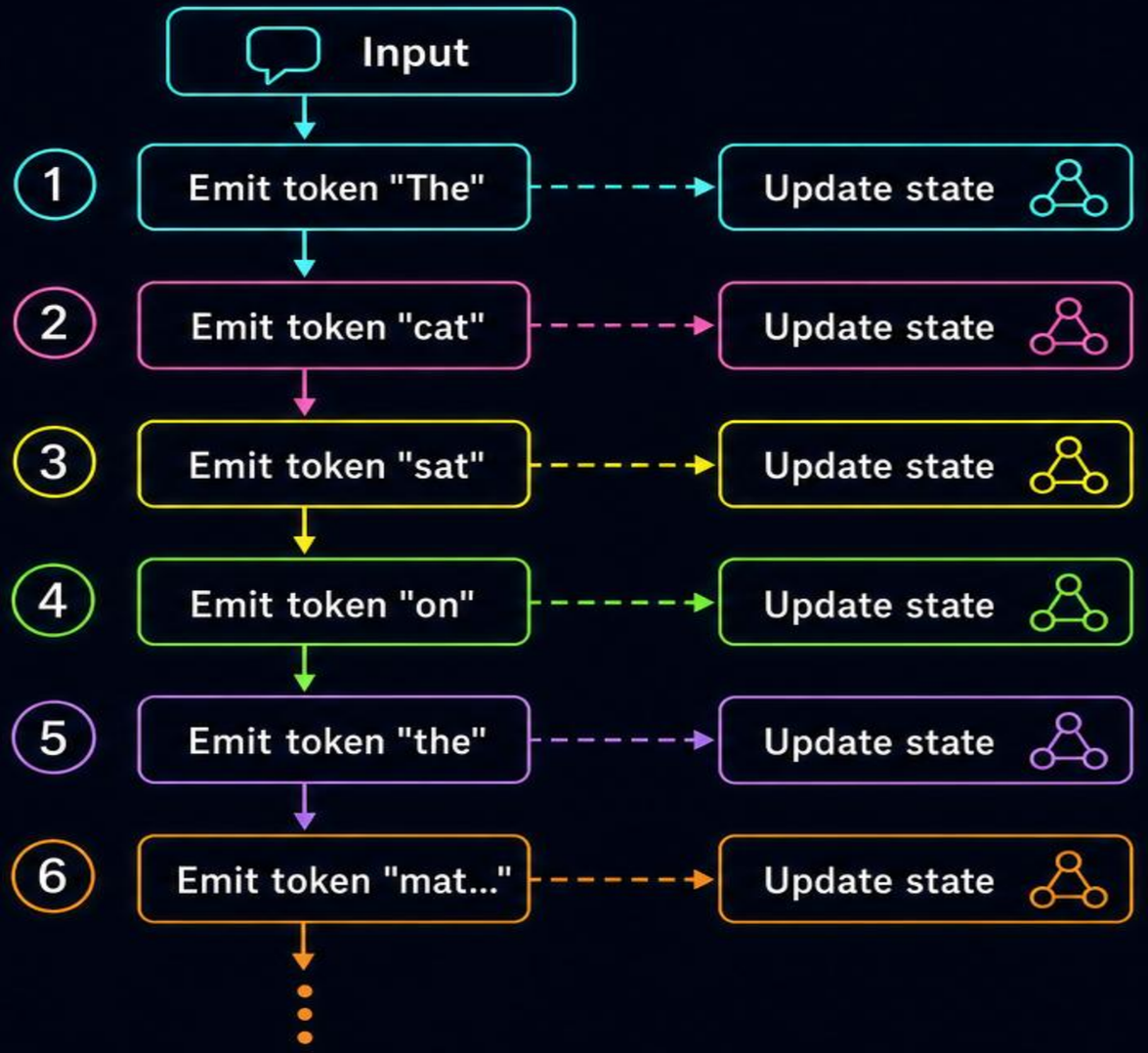
After each token, the model updates its state and decides what comes next based on the new context.

### Each token conditions on all prior tokens

Every token is generated using the entire sequence so far, not just the last token.

### There's no pre-written response waiting to be revealed

The model isn't reading from a hidden script. It builds the response token by token in real time.



# Two Mental Models



## Compose Then Emit

The wrong picture. The model writes a complete answer internally, reviews the whole thing, then hands you the finished text all at once.

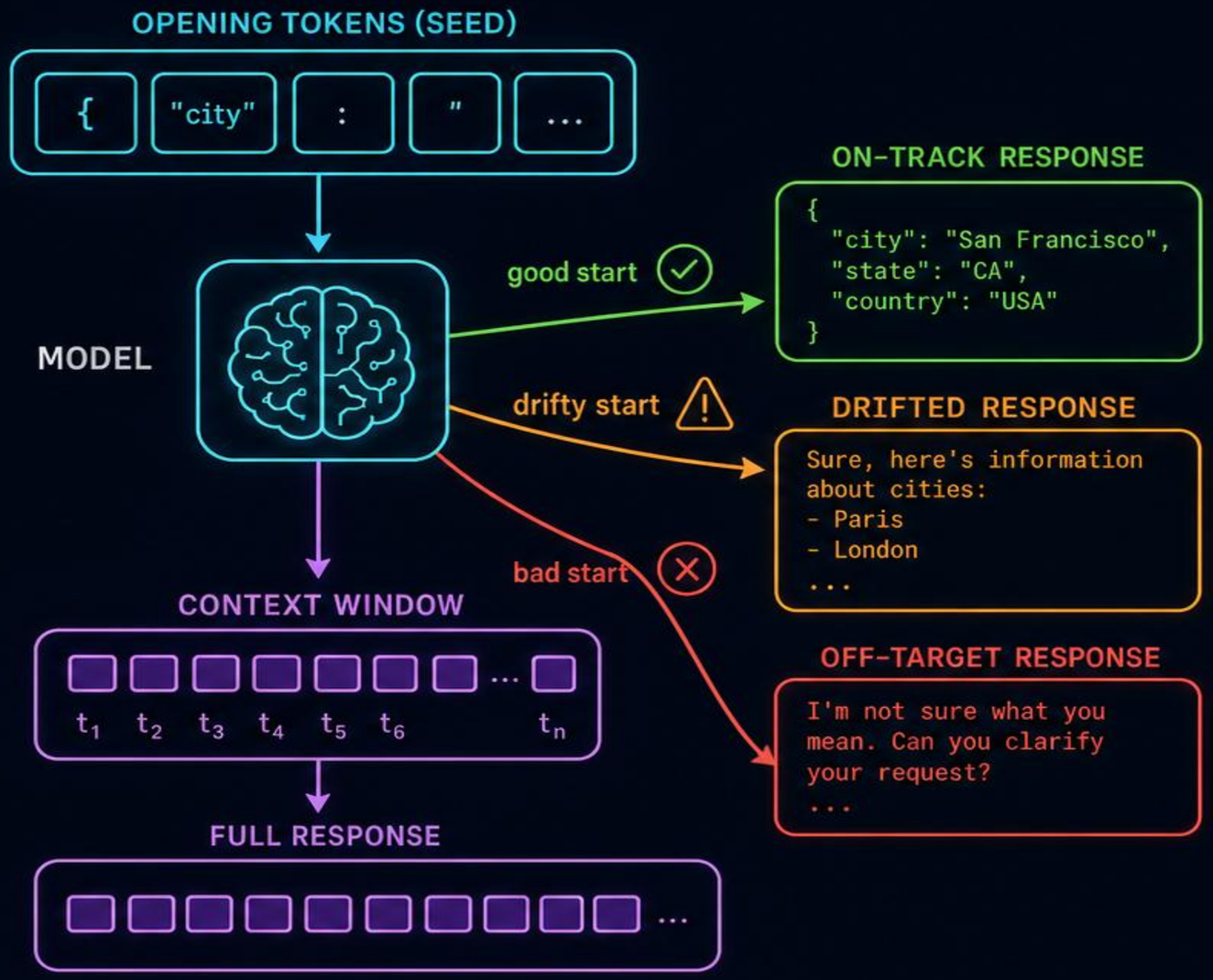


## Generate Left to Right

The right picture. It produces each token in sequence, left to right, with no lookahead to a finished draft it is building toward.

# Early Tokens Steer the Rest

Because each token conditions the next, the opening shapes everything.




### A strong format anchor early keeps output on track

Clear structure or schema at the start acts like rails for the model.

### Drift in the first tokens compounds down the response

Small deviations early create larger divergence as tokens accumulate.

### Structured openings make parsing far more reliable

Machines and systems can depend on predictable, parseable outputs.

# Streaming the Tokens



## Incremental Delivery

Server-sent events surface each token as it’s generated, well before the full reply has finished.



## Faster First Byte

Users see output sooner, a faster first byte, because you don’t wait for the whole response.




## Same Generation

Streaming changes only how output is delivered, not how the tokens are produced under the hood.

# Anchoring Output in Practice

You can exploit left-to-right generation when you prompt.

### Lead with the format you want the answer to follow

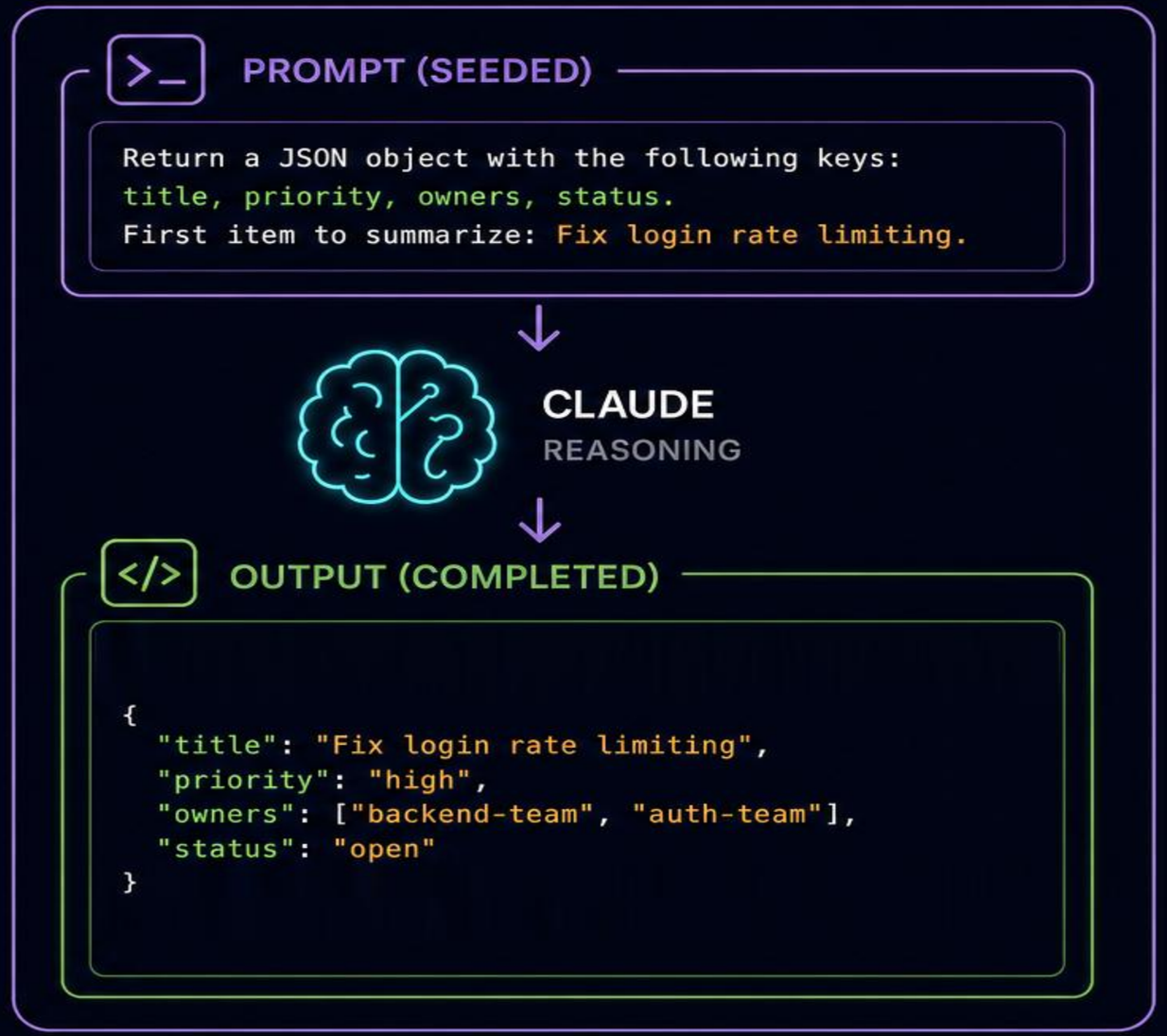
Show the structure first so the model knows what to produce.

### Give a clear first line for the model to continue

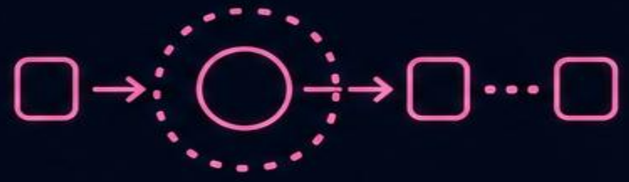
Start with a strong opening to set topic, tone, and intent.

### Prefer structured outputs when an exact shape matters

Use JSON, tables, or schemas to ensure consistent, machine-ready results.



# Tying the Three Together



## Tokens

The unit generated one at a time, and the same unit that fills the context window and the bill.



## Sampling

How each next token is picked from the distribution, probabilistic and non-deterministic.

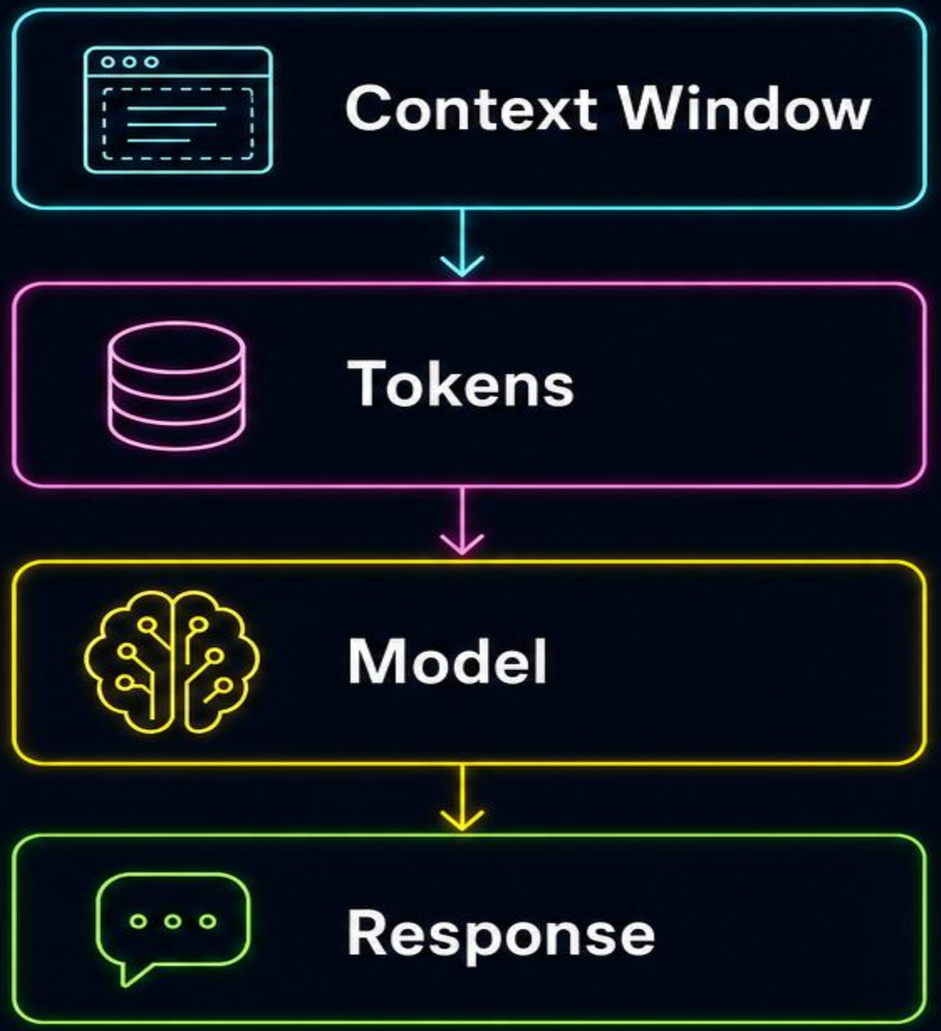


## Generation

The left-to-right loop that turns those picks into a full response.

# What This Means for Builders

These three fundamentals reshape how you design your systems around the model.




**Budget your tokens deliberately across the whole context window**

Plan how input, history, tools, and output consume tokens. Leave enough headroom for reasoning and the final answer.

**Expect variation, and validate every single response**

Outputs can differ. Check for correctness, completeness, and safety before you trust or act on them.

**Anchor your format early to steer the generation**

Define the structure, schema, and examples up front so the model stays consistent and usable for your system.

# Generation, Token by Token



**Generation** is autoregressive and left to right



**Each token** conditions on all prior tokens



**Early tokens** and format anchors steer the rest



**Streaming** delivers tokens before the reply completes

# Fast Mode and Adaptive Thinking

Speed and reasoning are things you tune on a Claude call, not just reasons to pick a different model.

## One Claude API Call

### SPEED

Slower — Faster

Lower latency
Less thinking time

### THINKING

Shallower — Deeper

More reasoning
Higher quality

```
{
  "model": "claude-opus-5",
  "max_tokens": 1024,
  "messages": [{ "role": "user", "content": "Summarize this log" }],
  "thinking": { "type": "adaptive" },
  "stream": true
}
```

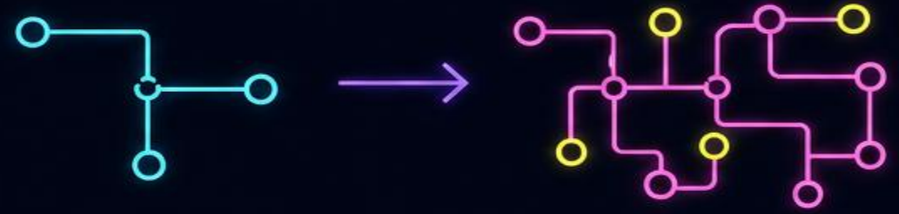
RESPONSE

## FAST MODE

Lower latency,
less thinking

## ADAPTIVE MODE

More thinking,
higher quality

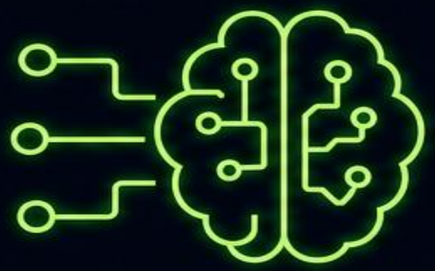


# Three Ways to Tune a Call



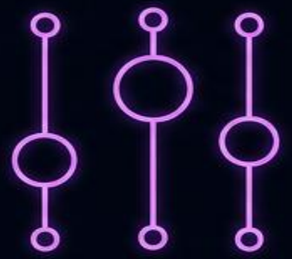
## Fast Mode

A latency lever that pushes the same model to stream output faster, at premium pricing. It changes throughput, not which model you call.



## Thinking

Letting the model reason privately before it answers. The current path is adaptive thinking, available on some models but not all.



## Effort

How much work the model spends on a response. That lever gets its own lesson coming up.

# What Fast Mode Does

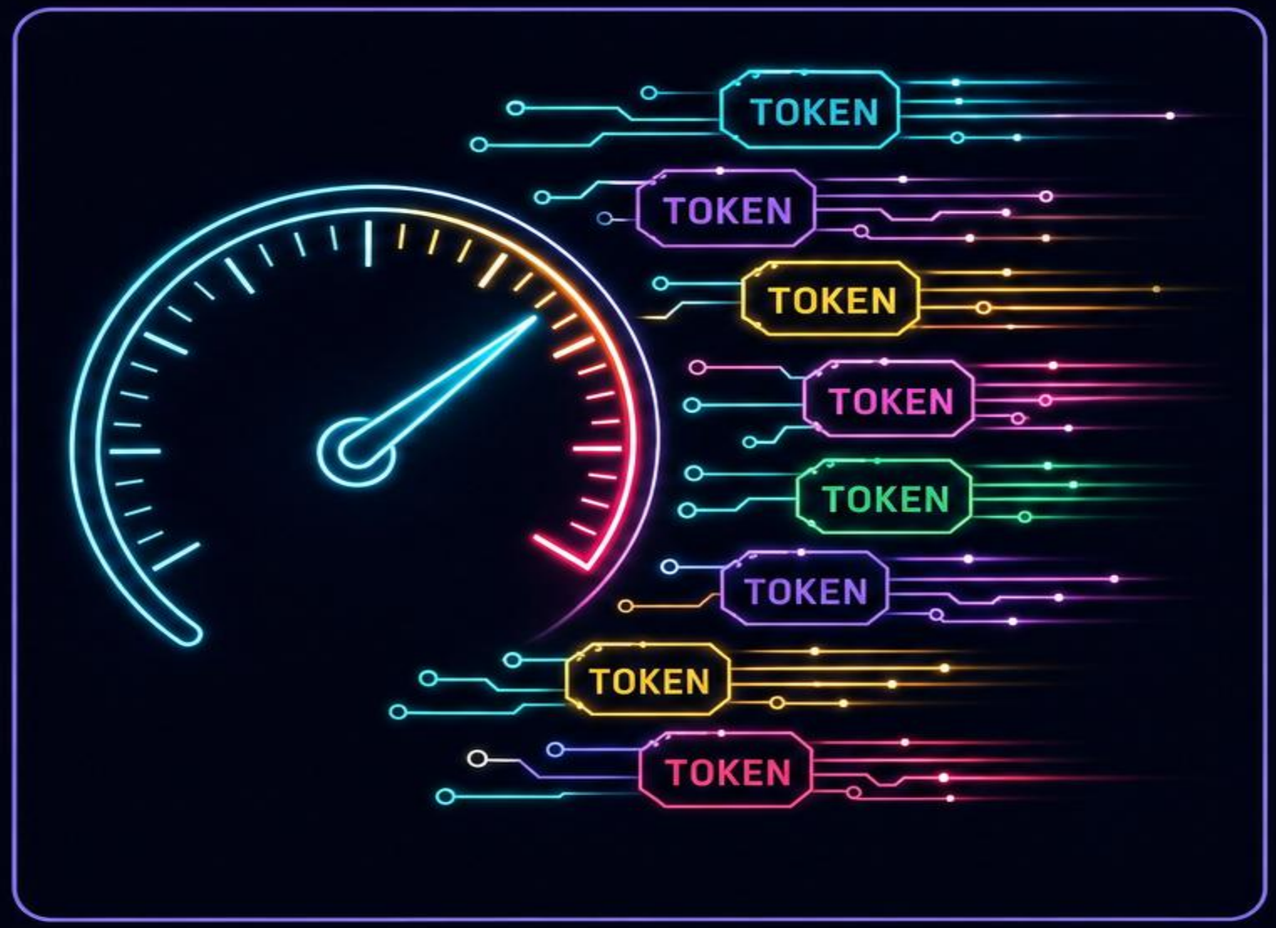
Fast mode trades money for speed on the same model.

**Pushes up to 2.5x** higher output tokens per second

**Runs at premium pricing** for that extra throughput

**Ships as a research preview** behind a beta flag

**Set with a top-level speed** option on the request



# Where Fast Mode Runs



## Opus 4.8

The durable fast-capable tier. Build new fast mode work here, because this is the long-term home.



## Opus 4.7

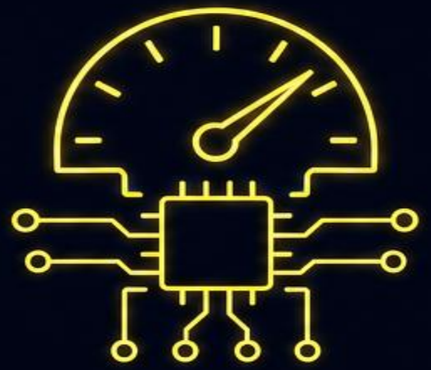
Also supports fast mode today, but this path is deprecated and is on its way out.



## Not Everywhere

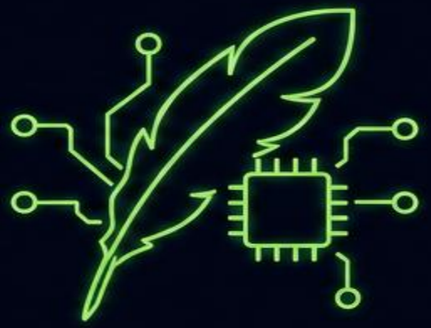
Fast mode is off the table on the Batch API and on any third-party model platform.

# Fast Mode vs a Smaller Model



## Fast Mode

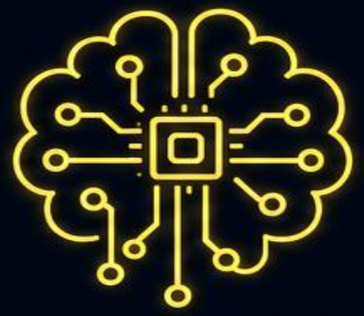
Keeps the same model and its quality, then pays a premium to push tokens out faster. You are buying speed on that model, not changing what it can do.



## Smaller Model

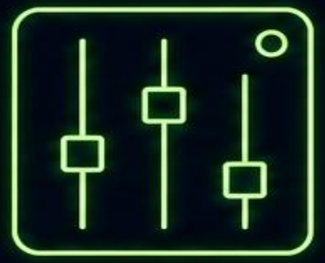
Swaps in a lighter, cheaper model like Haiku 4.5. You change capability and price at once, and usually gain some speed as a side effect.

# Thinking Before Answering



## Extended Thinking

The general idea of letting the model reason first, spending thinking tokens before it writes the visible answer.



## Adaptive Thinking

The current path. You turn it on and the model decides when and how much to think, with no budget to tune.



## Legacy Budget Mode

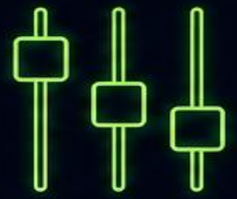
The older style where you set an explicit thinking budget. It survives only on older models now.

# Adaptive Thinking by Model



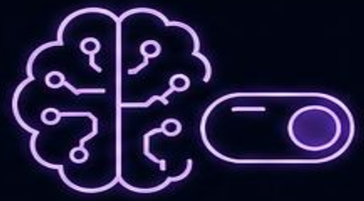
## Fable 5

Thinking is always on here. You cannot turn it off at all.



## Opus 4.8 and 4.7

Adaptive is supported, but it stays off unless you set it explicitly.



## Sonnet 5

Adaptive is on by default here. Omit the setting and it thinks. Pass thinking type disabled to turn it off.

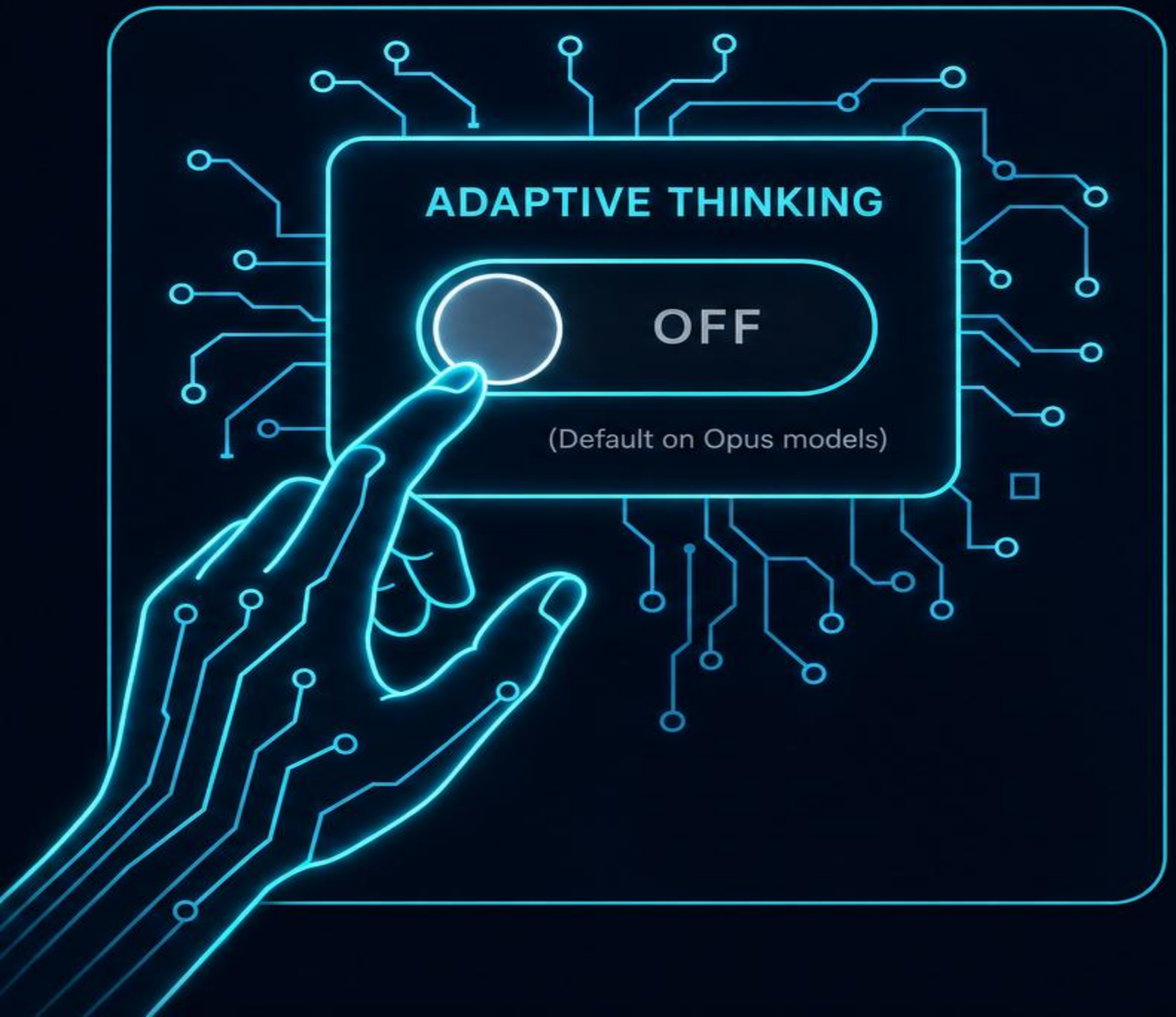



## Haiku 4.5

No adaptive thinking at all. It uses the older budget-based mode instead.

# Off by Default on Opus, So Set It

On the Opus models, thinking is off until you ask for it.




**Omitting the setting**
on Opus 4.8 or 4.7
means no thinking

**Turn on adaptive thinking**
explicitly there when
you want reasoning

**Fable 5 always thinks,**
and Sonnet 5 thinks
unless you disable it

**Reasoning stays hidden**
by default, shown
only if you ask

# Adaptive vs Legacy Budget Thinking



## Adaptive Thinking

The model decides how much to reason, with no budget to set. This is the current mode on frontier models, and the older budget style is rejected there.




## Legacy Budget Mode

You set an explicit token budget for thinking. It still works on older models like Sonnet 4.5 and Opus 4.5, but not on the newest ones.

# Speed and Thinking Levers



**Fast mode** buys speed on Opus 4.8 and 4.7, at premium pricing



**Adaptive thinking** defaults off on the Opus models and on for Sonnet 5



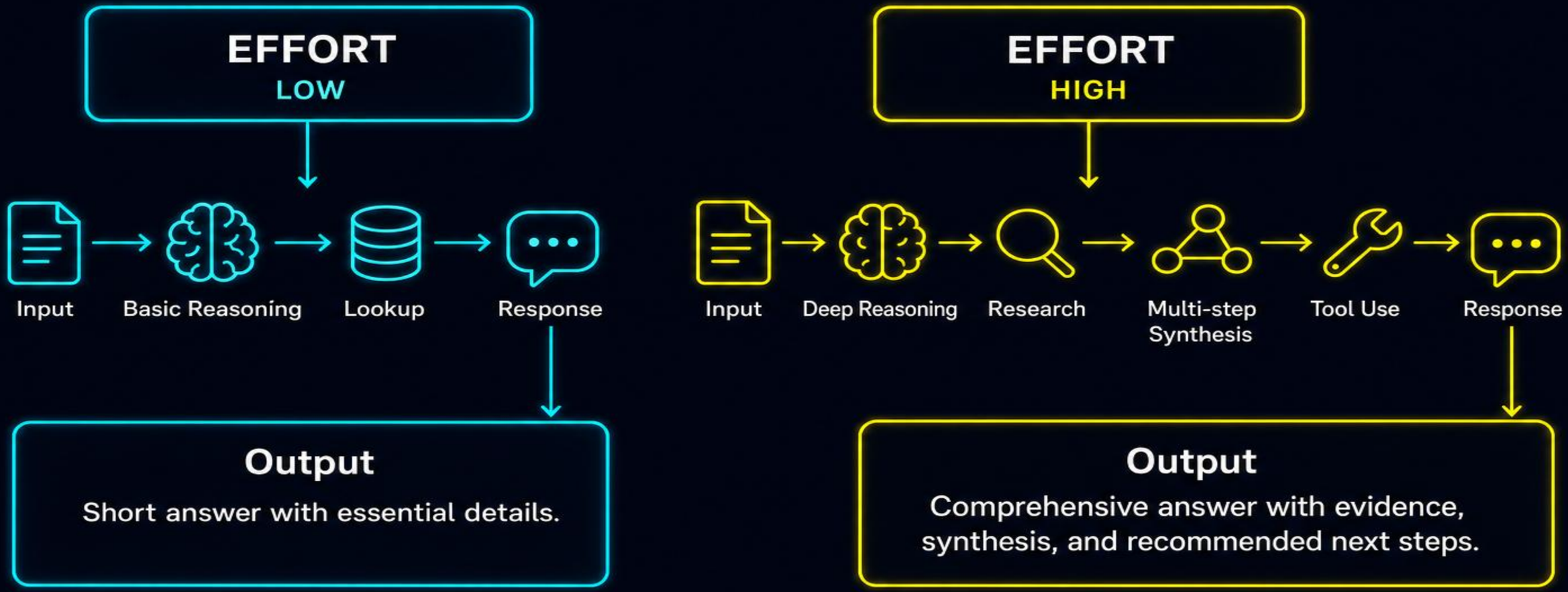
**Fable 5** always thinks, Haiku 4.5 uses the older budget mode



**Availability** is model-specific, so name the model first

# Effort Levels for Reasoning Depth

Effort is the dial for how much work Claude puts into a single response.

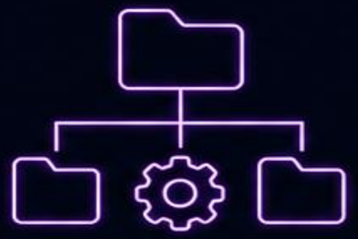


# Effort in Three Ideas



## Effort

The control for how much work the model spends on a response. Higher effort means more reasoning, more latency, and more cost.



## Where It Lives

A nested setting, not a top-level field. It sits under the output config as `output_config.effort`.



## The Default

Leave it out and you get `high`, which sits in the upper band, not the middle.

# The Five Effort Levels

Effort runs across five named levels, from least work to most.

**low,** the least work, fastest and cheapest

**medium,** a middle setting

**high,** the default, in the upper band

**xhigh,** above high, restricted to certain models

**max,** the most work the model will spend



# Higher Effort vs Lower Effort



## Higher Effort

The model reasons more and makes more thorough tool calls before it answers. You are buying answer quality, and you pay for it with added latency and cost.



## Lower Effort

The model does less. It consolidates its tool calls, trims the preamble, and answers faster and more cheaply, trading some depth for speed.

# High-End Tiers by Model



## xhigh

Restricted to Fable 5, Opus 5, Opus 4.8, Opus 4.7, and Sonnet 5. Errors on older tiers.



## max

The top level. Supported on Sonnet 5, Sonnet 4.6, Fable 5, and Opus 4.6 and up.

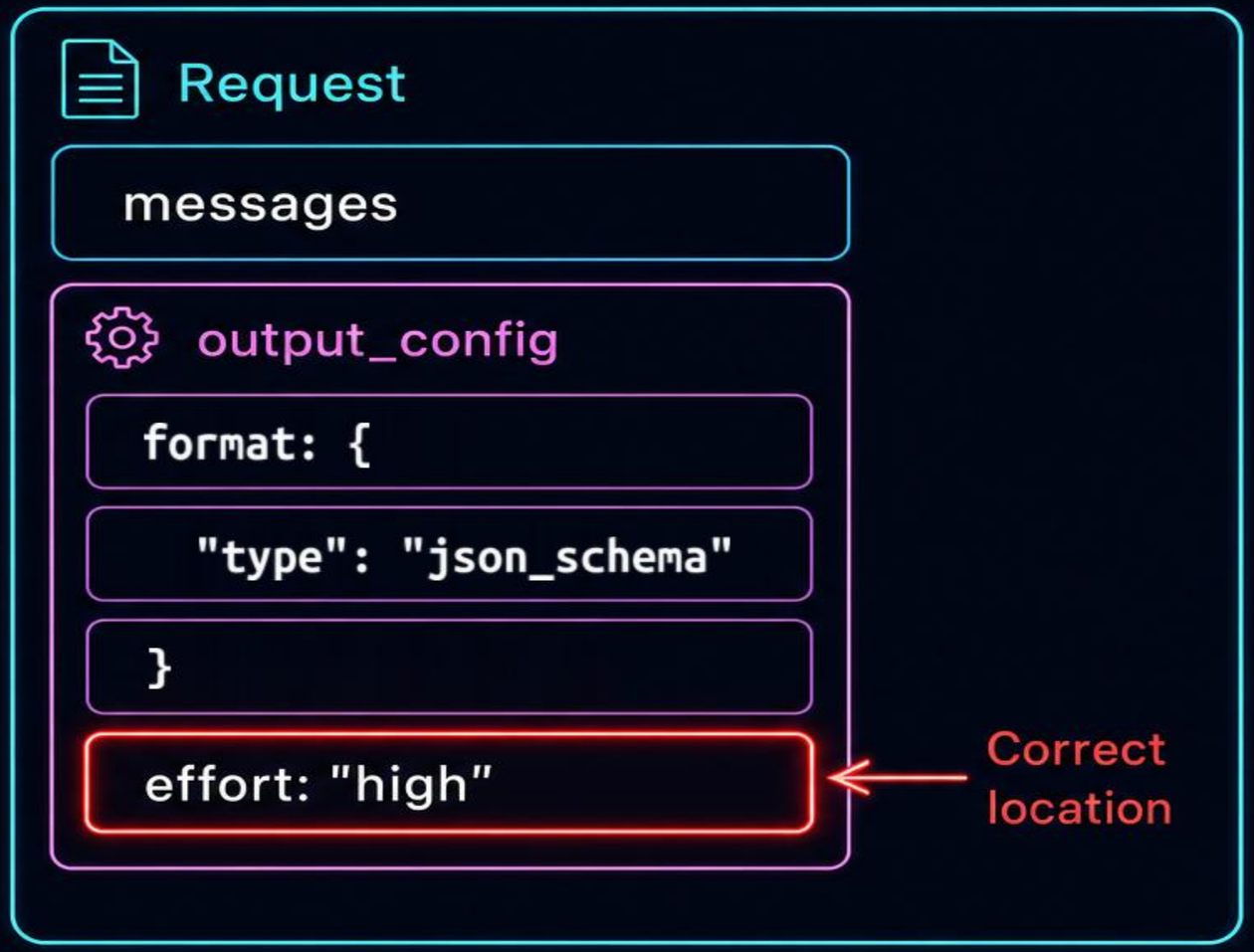


## Older Models

Reach past their limit and the request errors, so check before you set it.

# Effort Is Nested, Not Top-Level

The most common effort mistake is putting it in the wrong place.




**It lives under the output config,**
as output_config.effort

**A top-level effort field**
is the classic wrong guess

**Leave it out**
and the model defaults to high

**Not every model**
offers xhigh or max

# Three Levers, One Goal



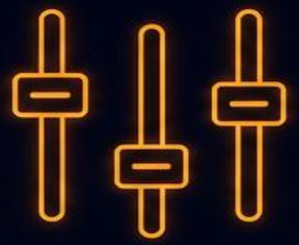
## Fast Mode

Optimizes latency. It pushes output faster on a supported model, at premium pricing, without changing capability.



## Thinking

Optimizes answer quality. It lets the model reason first, through adaptive thinking on current models.



## Effort

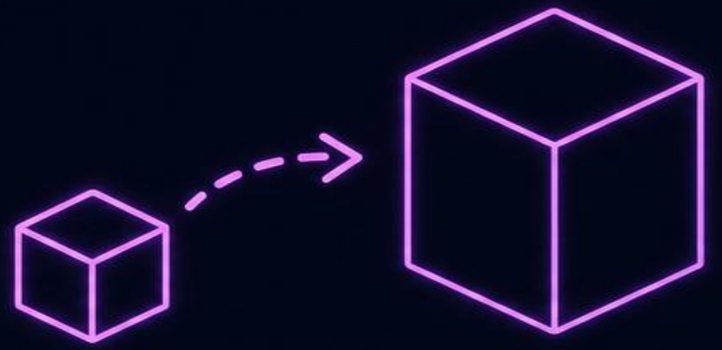
Optimizes within a single model. It dials how much work it spends, trading intelligence against latency and cost.

# Effort vs Switching Models



## Tune Effort

Stay on the same model and dial its work up or down. Cheaper and simpler, and the docs often prefer this as your first lever.



## Switch Models

Move to a different model entirely. A bigger jump that changes capability and price, worth it when effort alone can’t close the gap.

# Choosing for a Workload

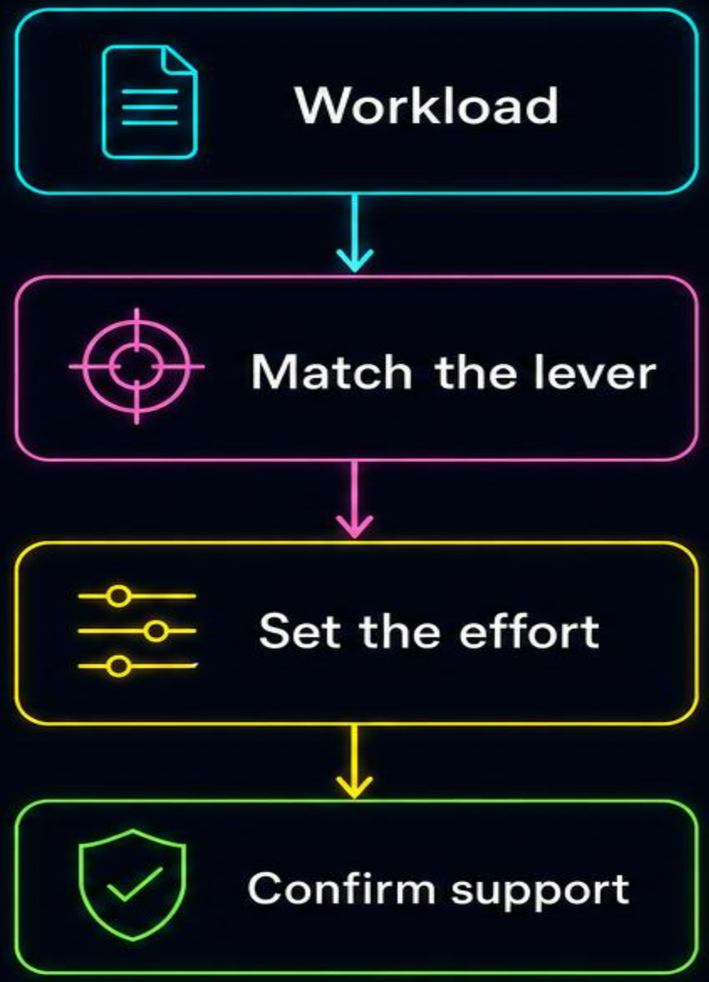
Pick your lever by what the workload actually needs.

**Real-time and cost-sensitive?**
Lean on low effort or a lighter model

**Latency-critical on a big model?**
Fast mode on Opus 4.8

**Hard reasoning?**
Adaptive thinking plus higher effort

**Always confirm**
the model supports the option first



# Effort and the Three Levers



**Levels** run low, medium, high, xhigh, max, with high the default



**Effort** is nested under output_config.effort, not top-level



**xhigh and max** are limited to certain models



**Weigh** fast mode, thinking, and effort by latency, cost, and quality

# Zero-Shot and One-Shot Prompting

Every prompt makes a choice about examples.



**DEVELOPER PROMPT**

"Classify the sentiment of the review as Positive, Neutral, or Negative."

**ZERO-SHOT**
No examples

Model infers the pattern

**REVIEW**

The product works well and the support was helpful.

**PREDICTION**
**Positive**

**DEVELOPER PROMPT**

"Classify the sentiment of the review."

**EXAMPLE:**

Review: "I love this!"
Sentiment: Positive

**ONE-SHOT**
One example

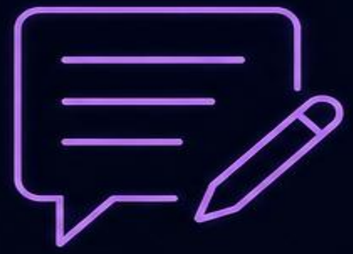
Model leverages the example

**REVIEW**

The product works well and the support was helpful.

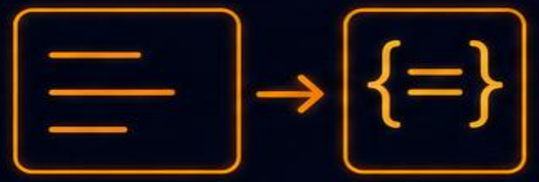
**PREDICTION**
**Positive**

# Three Ways to Prompt



## Zero-Shot

You state the task with no worked examples. The model leans on its pretrained knowledge and the clarity of your instruction alone.




## One-Shot (Single-Shot)

You include exactly one input and output example before the real request, so one canonical case defines the pattern you want.



## Multi-Shot

You supply two or more examples that demonstrate format, tone, or reasoning, covered in the next lesson.

# What Zero-Shot Really Means

Zero-shot is the plain instruction: describe the task and let the model do it.



**No examples,**
so the prompt stays lean and cheap on tokens



**Leans entirely**
on the model's pretrained skill and your wording



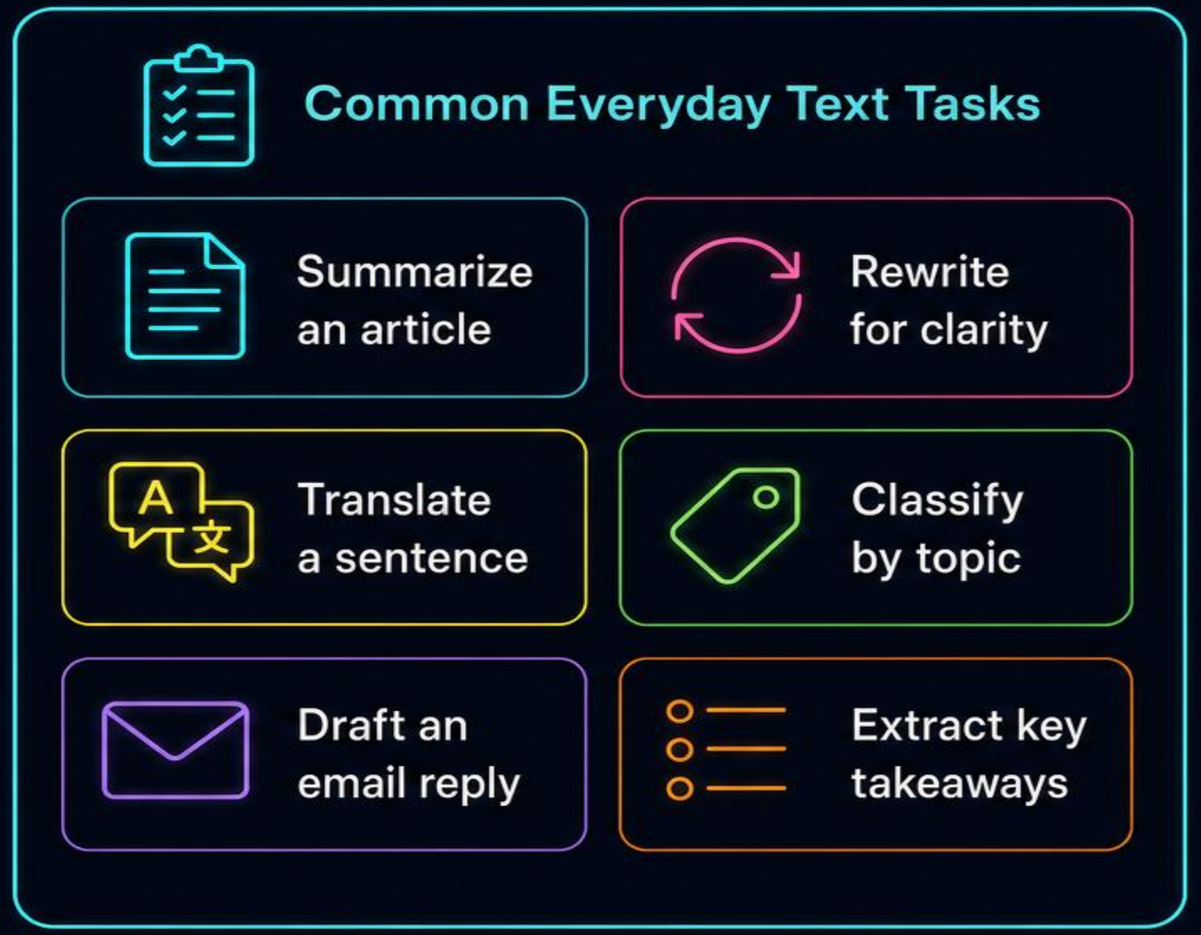
**Best when**
the task is common and well understood

**Zero-shot instruction**

````
1  Summarize the following customer
   support ticket in 3 bullet points.
2
3  Ticket:  ```text
4  My order #4821 has not arrived yet.
   It was supposed to be delivered
   on May 10. Please advise.
5  ```
````

# When Zero-Shot Is the Right Default

Start here for tasks the model has clearly seen many times before.




### Summaries, rewrites, translations, and simple classification

Tasks where the model has seen many similar examples in its training.

### Anything where the output format is obvious from the ask

When the desired structure and style are already clear.

### Cases where extra examples would only add token cost

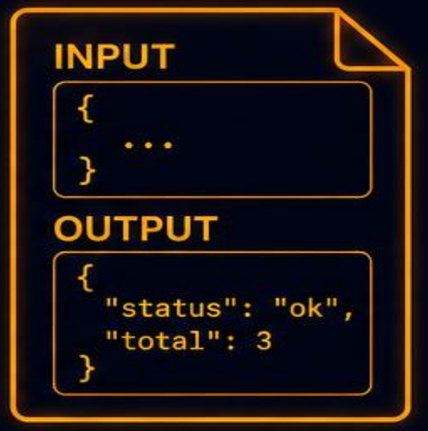
Zero-shot saves tokens and keeps prompts simple.

# A Shot Versus an Instruction



## An Instruction

Tells the model what to do in words. It describes the task, the format, or the rules, but it does not show a finished result to copy.




## A Shot

Shows a full worked example: a sample input paired with the exact output you want back. It is demonstration, not description.

# What Counts as a Shot



## Worked Example

A sample input paired with its finished output, shown to the model before the real request.




## Not an Instruction

A rule like keep it brief is guidance, not a shot. No sample output means no shot.



## Lives in the Turns

Shots sit in the message history as prior input and output pairs the model reads.

# What One Example Adds

One-shot hands the model a single template to copy.



**Pins down an output format**
words alone struggle to convey



**Anchors tone or structure**
with one concrete case



**Keeps the prompt small**
while still showing the target

**ONE EXAMPLE (INPUT)**

```
{
  "task": "Summarize this report",
  "report_id": "RPT-2025-05-18",
  "length": "3 bullets",
  "tone": "Executive",
  "output": [
      "Key finding 1",
      "Key finding 2",
      "Key finding 3"
  ]
}
```

↓ COPIED

**MODEL OUTPUT (BASED ON EXAMPLE)**

```
{
  "task": "Summarize this report",
  "report_id": "RPT-2025-06-01",
  "length": "3 bullets",
  "tone": "Executive",
  "output": [
      "Revenue grew 12% year over year.",
      "Operating margin improved to 18%.",
      "R&D investment increased to support growth."
  ]
}
```

# When One Example Is Enough

Reach for one-shot when a single case fully defines the pattern.

ONE-SHOT EXAMPLE RESPONSE

```
{
  "type": "order_summary",
  "order_id": "A7X-9283",
  "customer": {
    "id": "C-1049",
    "name": "Acme Labs",
    "tier": "Gold"
  },
  "items": [
    { "sku": "WID-128", "name": "Widget Pro", "qty": 3,
      "unit_price": 49.99, "currency": "USD" },
    { "sku": "ACC-201", "name": "Docking Station", "qty": 1,
      "unit_price": 89.00, "currency": "USD" }
  ],
  "totals": {
    "subtotal": 238.97, "tax": 19.12, "shipping": 0.00,
    "total": 258.09, "currency": "USD"
  },
  "status": "confirmed",
  "estimated_ship_date": "2025-06-10"
}
```



**A specific JSON shape**
you want every response to match



**A distinctive tone**
or phrasing that words describe poorly




**A niche format**
the model rarely produces unprompted

One clean example often does what a paragraph of rules cannot.

# Three Signals to Read



## Task Familiarity

If the model has clearly done this task type often, a plain zero-shot instruction usually suffices.



## Format Precision

If the exact shape of the output is hard to describe, one worked example pins it down fast.



## Token Budget

Every example you add costs input tokens, so add one only when it earns its keep.

# Choosing Zero or One Shot



**Zero-shot** states the task with no examples, best for common work



**One-shot** shows one worked example to lock a format or tone



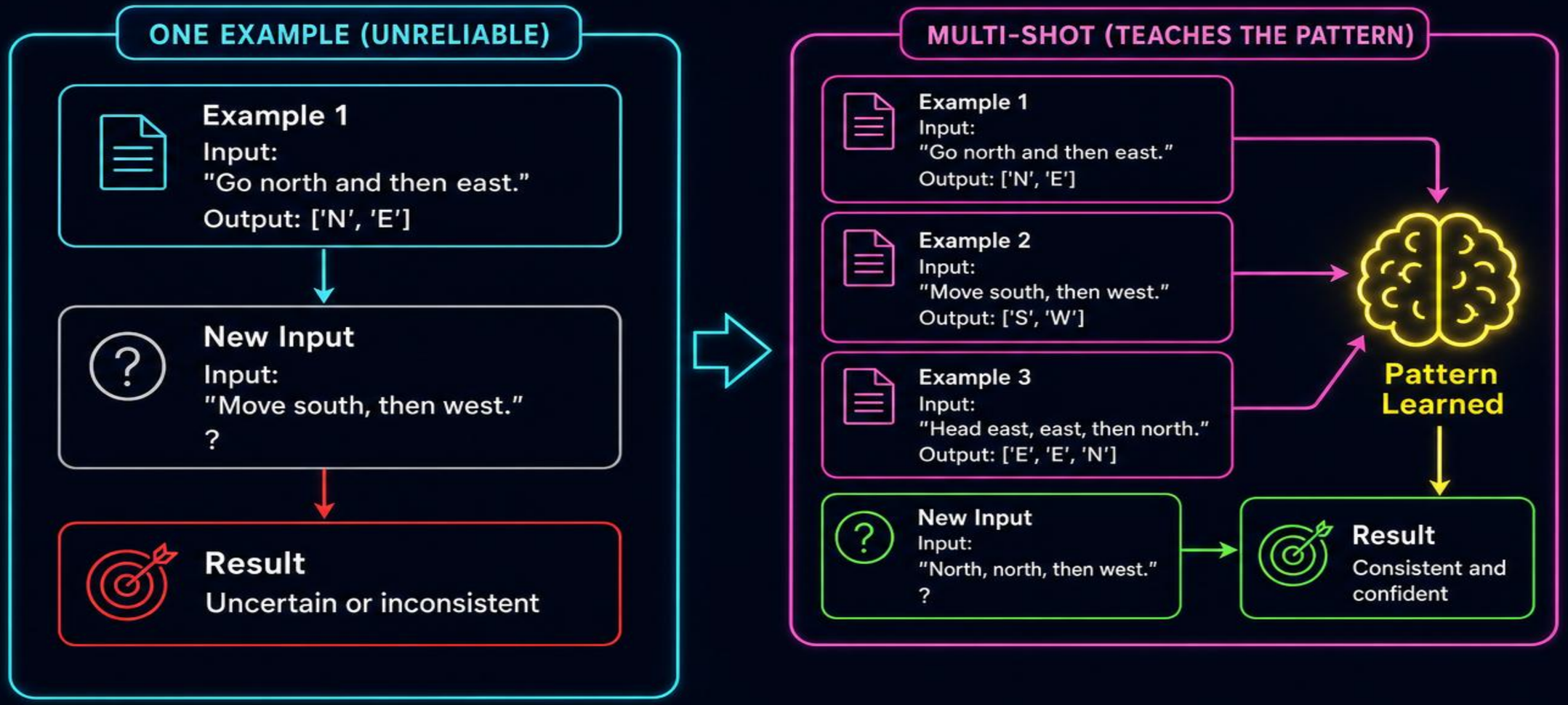
**A shot** is a demonstrated input and output, not just an instruction



**Add an example** only when it improves the result

# Multi-Shot Prompting and Example Choice

When one example isn't enough, you move to multi-shot:
two or more worked examples that teach the model a pattern.



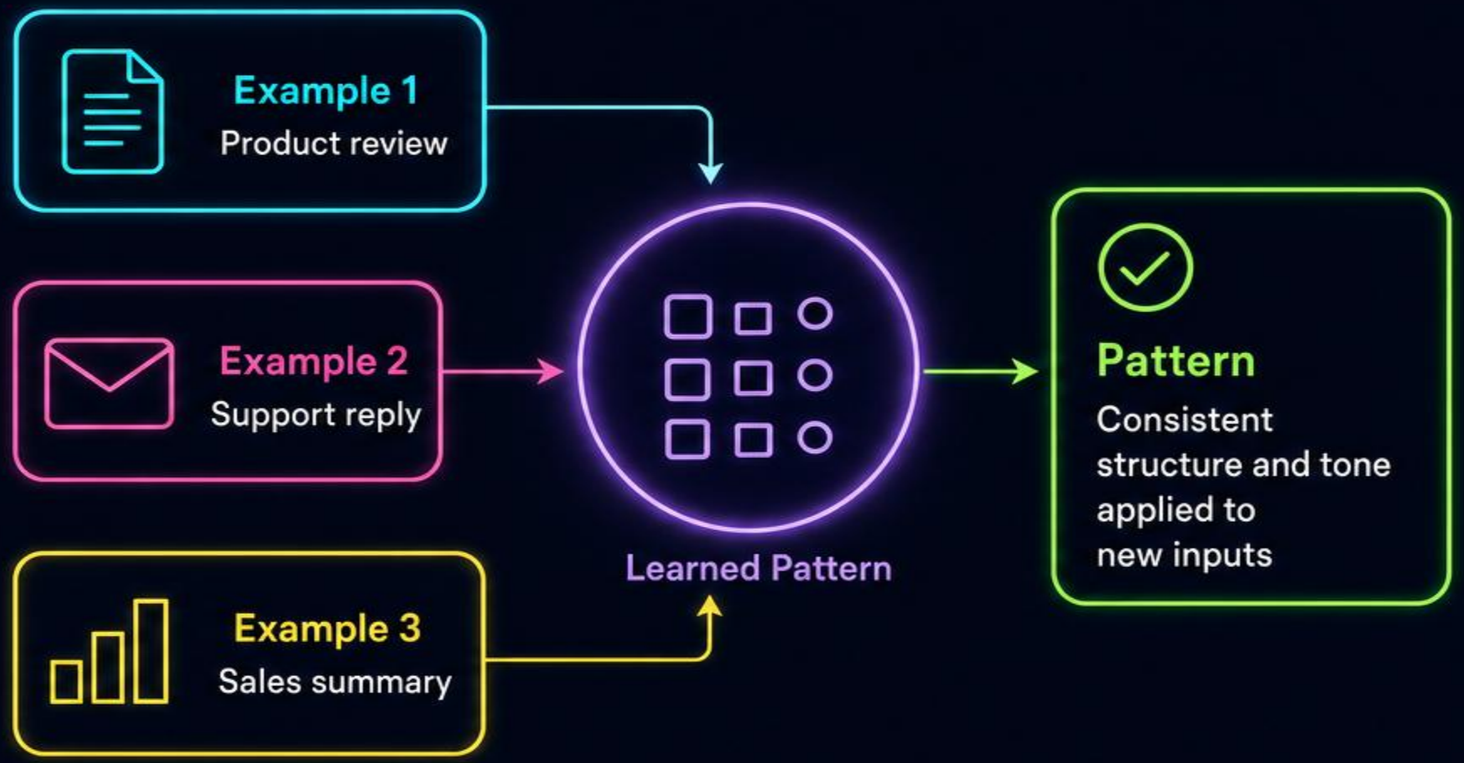
# What Multi-Shot Demonstrates

Multi-shot means two or more examples working together to show a pattern.

**Reinforces** an output format across several cases, not just one

**Demonstrates** tone, structure, or a reasoning style to imitate

**Shows how** to handle variation the model might otherwise miss




Several examples let the model generalize the rule, not one instance.

# Three Marks of a Good Example



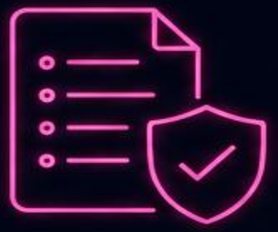
## Relevant

The example mirrors your actual use case closely, so the pattern it teaches transfers directly to real inputs.



## Diverse

Examples vary enough to cover edge cases, so the model doesn't overfit to one narrow shape.

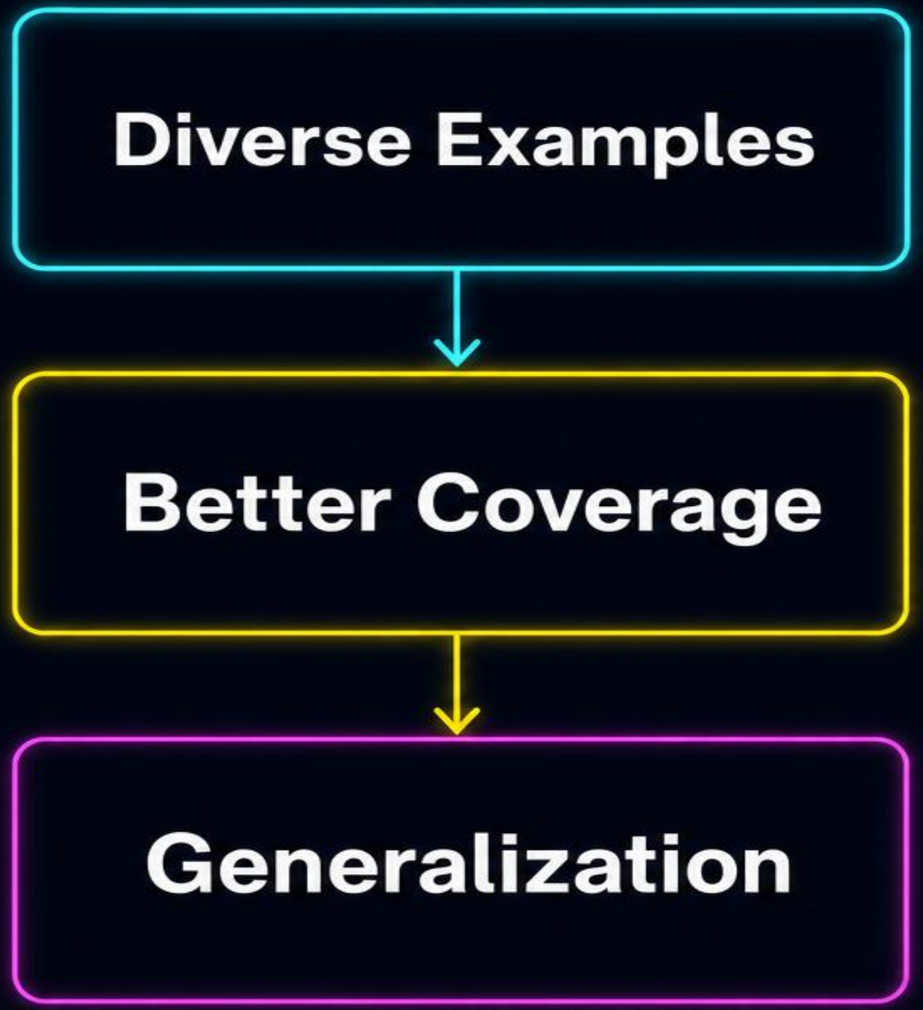


## Structured

Each example is clearly delimited from your instructions, so the model reads it as a sample, not a command.

# Quality Beats Quantity

The instinct to pile on examples usually backfires. Aim for a few strong ones.




**A handful of diverse examples beats a long list of near-duplicates**

Diversity gives models better coverage and generalization.

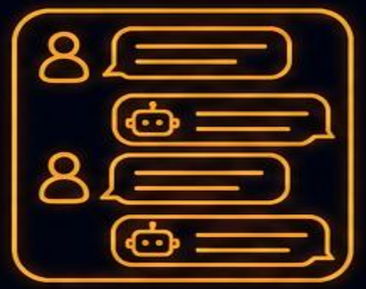
**Each example should teach something the others don't**

Every example should add new signal, not repeat the same point.

**Redundant examples add token cost without adding coverage**

More isn't better if it doesn't expand what the model learns.

# Where Examples Live



## In the Message History

Few-shot examples sit as prior input and output turns the model reads before your real request.



## Not a Prefill Trick

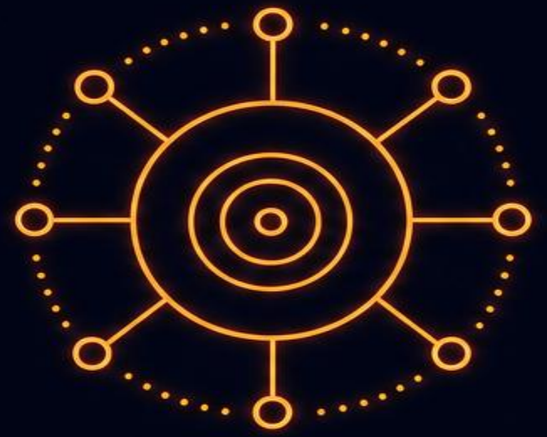
On current models, prefilling the assistant turn to force a format is rejected. Use examples or structured output instead.



## Delimit Each One

Mark clear boundaries around every example so the model reads it as a sample, not an instruction.

# Coverage Versus Clutter



## Coverage

Examples that span the real input range,
including the tricky edge cases,
teach the model to generalize well
beyond the obvious cases it will see.



## Clutter

Many similar examples of the easy case
bloat the prompt, waste tokens,
and can overfit the model
to the happy path.

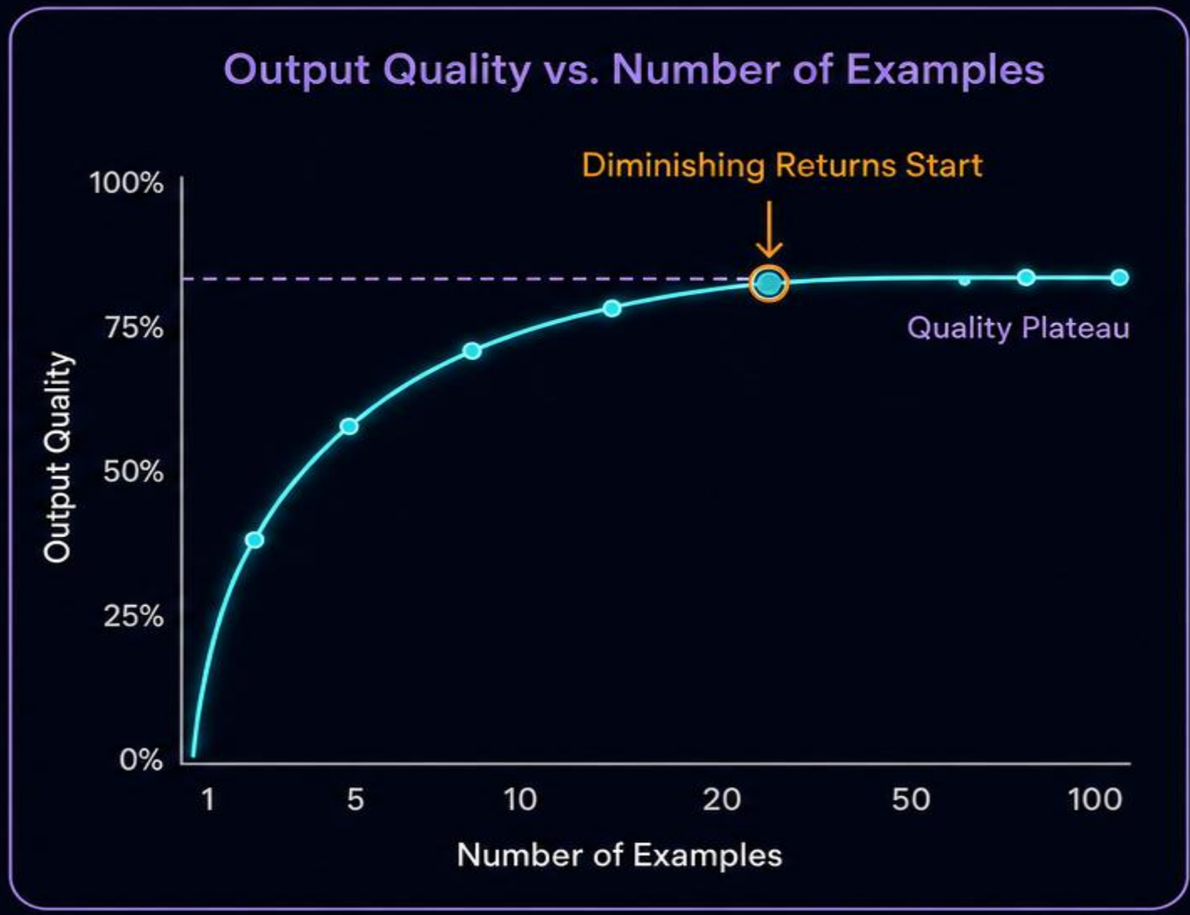
# When More Examples Stop Helping

Adding examples shows diminishing returns. At some point each new one earns nothing.

**Once coverage is complete,**
extra examples only repeat what's shown

**Every added example**
still costs input tokens on every call

**Watch for the point where**
output quality flattens out



# Placing Examples on Current Models

Where you put examples matters as much as choosing them.




### Put few-shot examples in the message history as prior turns

Few-shot examples work best when they appear as earlier turns in the conversation, not buried in the same message as your instructions.

### Delimit each one so the model separates it from instructions

Use clear start and end markers around each example so the model can separate examples from the task you want it to perform.

### Skip assistant prefills, which current models reject

Do not include assistant-prefilled responses as part of your prompt. Current models reject or ignore assistant-prefilled content.

# Three Habits That Pay Off



## Curate, Don’t Dump

Choose a small set of canonical examples that portray the behavior you want, instead of stuffing in every case.



## Test the Set

Check whether your examples are relevant and diverse, and drop any that don’t add new coverage.



## Match the Placement

Keep examples as prior turns and delimit them, so the model reads demonstrations, not stray instructions.

# Choosing and Placing Examples



**Several examples** teach a pattern the model generalizes



**Relevant, diverse examples** beat a pile of near-duplicates



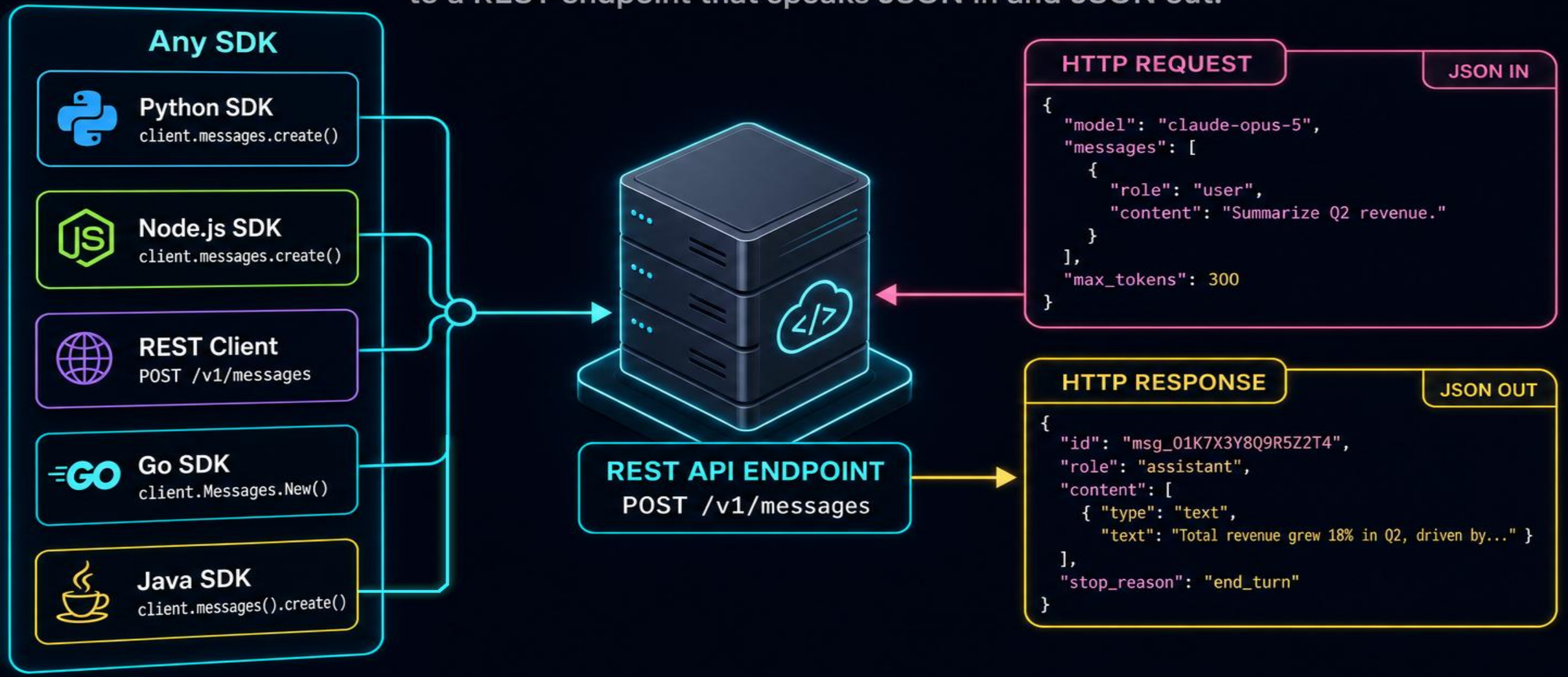
**Coverage of edge cases** matters more than sheer count



**Place examples as prior turns,** since prefills are now rejected

# One REST API Under Every SDK

Every call you make to Claude, in any language, is one HTTP request to a REST endpoint that speaks JSON in and JSON out.



# REST, JSON, and the Endpoint




## REST interface

The Claude API is an HTTP REST service you reach at a URL, getting a structured response back.




## JSON body

Requests carry a JSON body of model, messages, and params. Responses return JSON content blocks and metadata.




## Messages endpoint

Nearly everything routes through one endpoint, a POST to /v1/messages. Tools and caching are features of that call.

# What a Request Looks Like

A single request tells Claude everything it needs to produce one turn.



**A model id names**
which Claude you want



**A messages array**
holds the role-tagged conversation



**Generation params**
like max_tokens and stop_sequences shape the output

REQUEST BODY

```
POST https://api.anthropic.com/v1/messages
Content-Type: application/json
x-api-key: $ANTHROPIC_API_KEY
anthropic-version: 2023-06-01

{
  "model": "claude-opus-5",
  "max_tokens": 1024,
  "stop_sequences": ["</assistant>"],
  "messages": [
    { "role": "user", "content": "What's the weather in Tokyo?" }
  ],
  "system": "You are a helpful assistant.",
  "tools": [
    {
      "name": "get_weather",
      "description": "Get current weather for a location",
      "input_schema": { "type": "object", "properties": { "location": { "type": "string" } } }
    }
  ]
}
```

# The Same Call, Two Ways



## Raw REST call

You build the HTTP request yourself. Set the headers, serialize the JSON body, POST it to the endpoint, and parse the JSON response by hand.

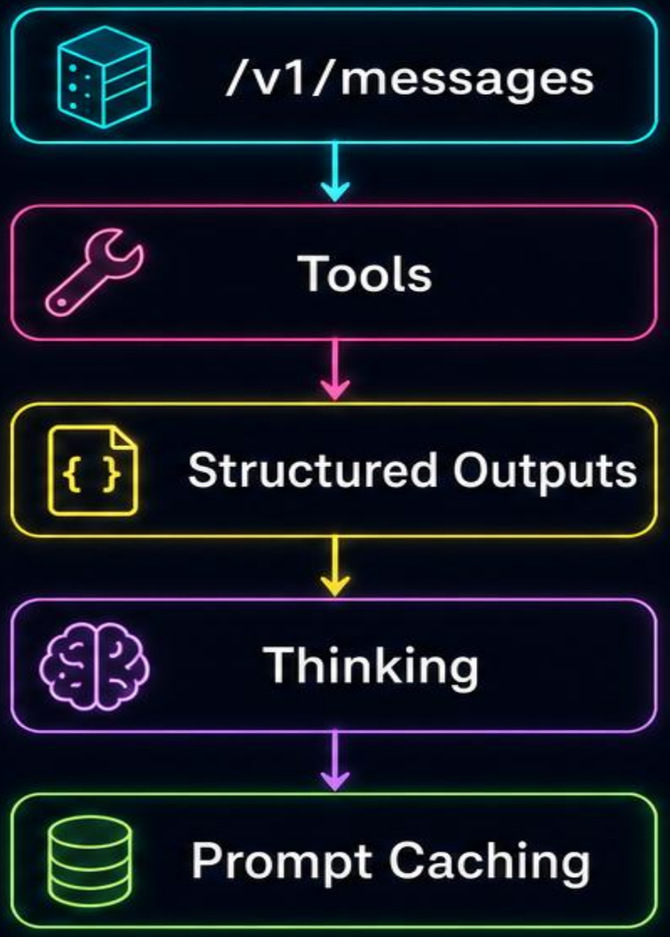


## SDK call

You call client.messages.create with typed arguments. The SDK builds the exact same request, sends it, and hands you a typed response object.

# One Endpoint, Many Features

You do not learn a new API for each capability. They all ride the same call.






### Tools and tool-use loops are fields on the request

Declare the tools once. The model can decide when and how to use them.



### Structured outputs and thinking are request options

Shape the response format and enable extended reasoning when needed.



### Prompt caching is a control on message blocks

Cache stable context to reduce latency and cost across repeated calls.

Master the one endpoint and every feature becomes familiar.

# Supporting Endpoints



## Batches

Submit many requests at once for asynchronous processing at half the standard price, through the /v1/messages/batches endpoint.



## Files and models

Upload reusable files at /v1/files, and list the models available to you at /v1/models.




## Token counting

Count tokens before you send with /v1/messages/count_tokens, so you can budget cost and context up front.

# The API Is Stateless

The endpoint remembers nothing between calls. Each request stands alone.



### You resend the full conversation history every turn

Each API call includes the entire message array so the model has full context.



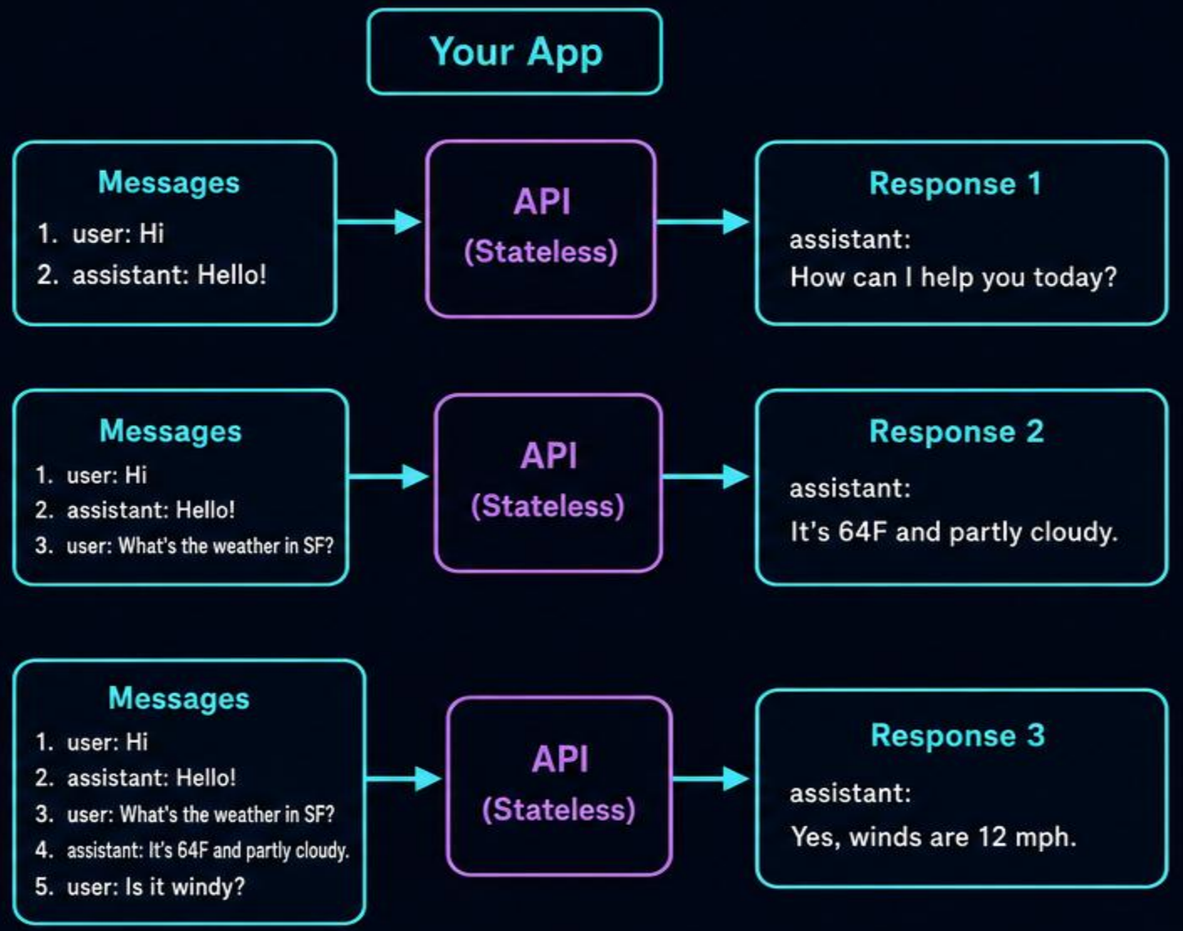
### There is no session or server-side memory to rely on

The API stores nothing for you. It does not keep state between requests.

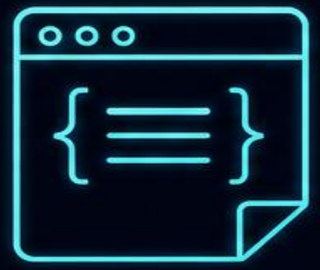


### Appending each reply back into messages is your job

You must add each assistant reply to the history before making the next call.



# How the SDK Maps to REST



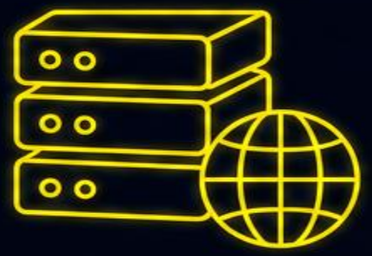
## Generated client

The SDKs are generated from one OpenAPI spec, so a new API feature shows up across every language client together.



## One-to-one fields

JSON field names map directly to SDK parameters, with only naming-style differences. What you read in the API docs is what you pass.



## Same wire behavior

client.messages.create sends the identical POST /v1/messages under the hood, so behavior including streaming matches a raw call.

# Why This Mental Model Helps

When you picture one REST call underneath, the whole platform gets simpler.




**You can debug** by inspecting the raw request and response

**Docs written** for the API apply to your SDK code

**You can drop** to raw HTTP anytime without losing features

The SDK is a convenience, not a different product.

# One Endpoint, Every Call



**Requests and responses** are JSON over HTTP




**Almost everything** routes through POST /v1/messages



**The API is stateless,** so you resend history each turn



**SDK calls** become the exact same REST call underneath

# What the SDK Does For You

If the SDK is just a wrapper over REST, what work does it actually save you? Quite a lot, as it turns out.

## Without the SDK: Manual Plumbing

your_app.py

```python
import os, json, httpx

API_KEY = os.getenv(
  "ANTHROPIC_API_KEY")

BASE = "https://api.anthropic.com/v1"

HEADERS = {
  "x-api-key": API_KEY,
  "anthropic-version": "2023-06-01",
  "content-type": "application/json"
}

payload = {
  "model": "claude-opus-5",
  "messages": [
    {"role": "user",
     "content": "Summarize Q2 revenue"}
  ],
  "max_tokens": 400
}

# Make the request
resp = httpx.post(
   f"{BASE}/messages",
   headers=HEADERS,
   json=payload,
   timeout=30.0
)

data = resp.json()
answer = data["content"][0]["text"]
print(answer)
```

1. **Manage API Key** — Load, store, rotate
2. **Build Request** — Headers, URL, payload JSON
3. **Send HTTP** — Handle auth, retries, timeouts, errors
4. **Parse Response** — Check status, parse JSON
5. **Handle Errors** — Network issues, rate limits, invalid
6. **Extract Data** — Pull text, usage, metadata

## With the SDK: Convenience Layer

your_app.py

```python
from anthropic import Anthropic
client = Anthropic()

msg = client.messages.create(
  model="claude-opus-5",
  max_tokens=400,
  messages=[
    {"role": "user",
     "content": "Summarize Q2 revenue"}
  ]
)
print(msg.content[0].text)
```




- **Auth** — API Key
- **HTTP** — Requests
- **Retries** — Timeouts
- **Parse** — Responses
- **Error** — Handling
- **Usage** — Metadata

# Four Jobs the SDK Handles



## Authentication

It reads your API key and sets the required headers on every request.



## Retries

It automatically retries transient failures with exponential backoff.



## Typing

It gives you typed arguments and response objects, so your editor catches mistakes early.



## Streaming parsing

It parses the server-sent event stream into clean message objects for you.

# Authentication in One Place

The SDK centralizes the credentials and headers every request needs.

### Your API key goes in once, at client setup

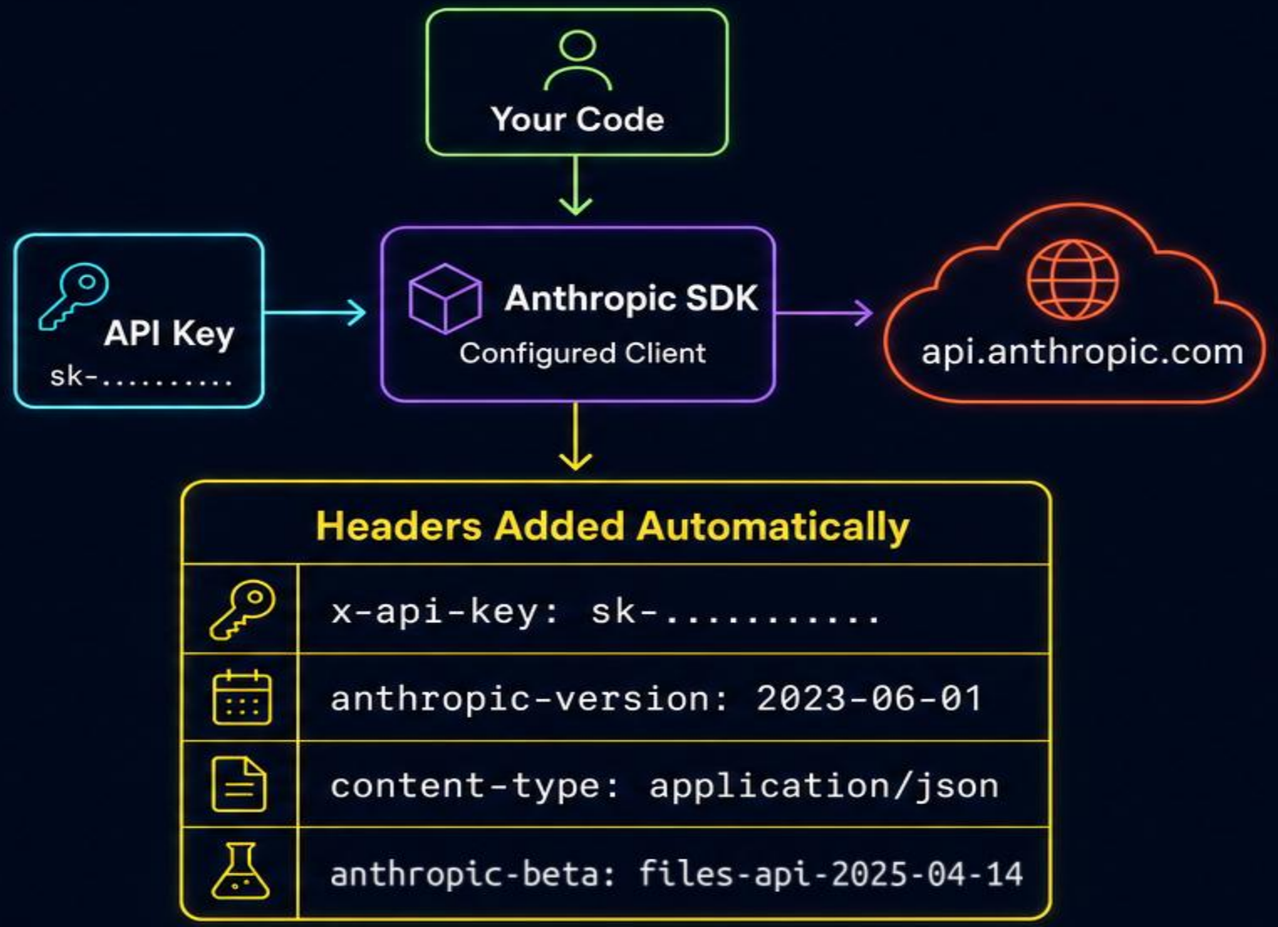
Store it once in the SDK client.
Everything uses it automatically.

### It sets x-api-key and the anthropic-version header

The SDK attaches your key and pins the API version.

### Content-type and beta headers are added for you

The SDK includes required headers so requests work every time.




**Get the client configured, and auth stops being a per-call chore.**

# Retries and Timeouts



## Automatic retries

The SDK retries transient errors, like 429 rate limits and 5xx server errors, with exponential backoff. By default it retries twice before giving up.



## Sensible timeouts

It applies a default request timeout, around ten minutes. For very long generations it expects you to stream, rather than block on one huge response.

# Typing Catches Bugs Early

Strong types turn a class of runtime errors into editor warnings.

sendMessage.ts

```ts
import Anthropic from "@anthropic-ai/sdk";
const client = new Anthropic();
const result = await client.messages.create({
  model: "claude-opus-5",
  max_
});
console.log(result.
content[0].text);
```

| | |
|---|---|
| max_tokens | number |
| metadata | Metadata |
| messages | MessageParam[] |
| stream | boolean |

Property 'max_' does not exist.
Did you mean 'max_tokens'?



**Request parameters** are typed, so typos fail before you run



**Responses** are objects, so you read fields, not raw JSON keys



**Autocomplete** surfaces what each call accepts and returns

---

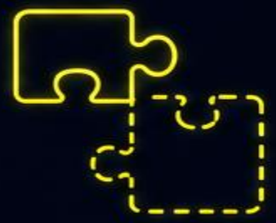
You lean on the type system instead of reading docs constantly.

# Convenience, Not New Powers



## No hidden features

The SDK exposes only what the REST API supports. If an endpoint can't do something, no SDK method can conjure it.



## Surface limits apply

A capability a given hosting surface excludes, like a particular server tool, is missing in that surface's SDK client too.



## Check the API first

If you're unsure whether something is possible, the question is always whether the REST endpoint supports it.

# Streaming Is Just Parsed for You

Streaming responses arrive as a sequence of small events over one connection.

### The API sends server-sent events as tokens generate

The model streams deltas to your app over one HTTP connection.

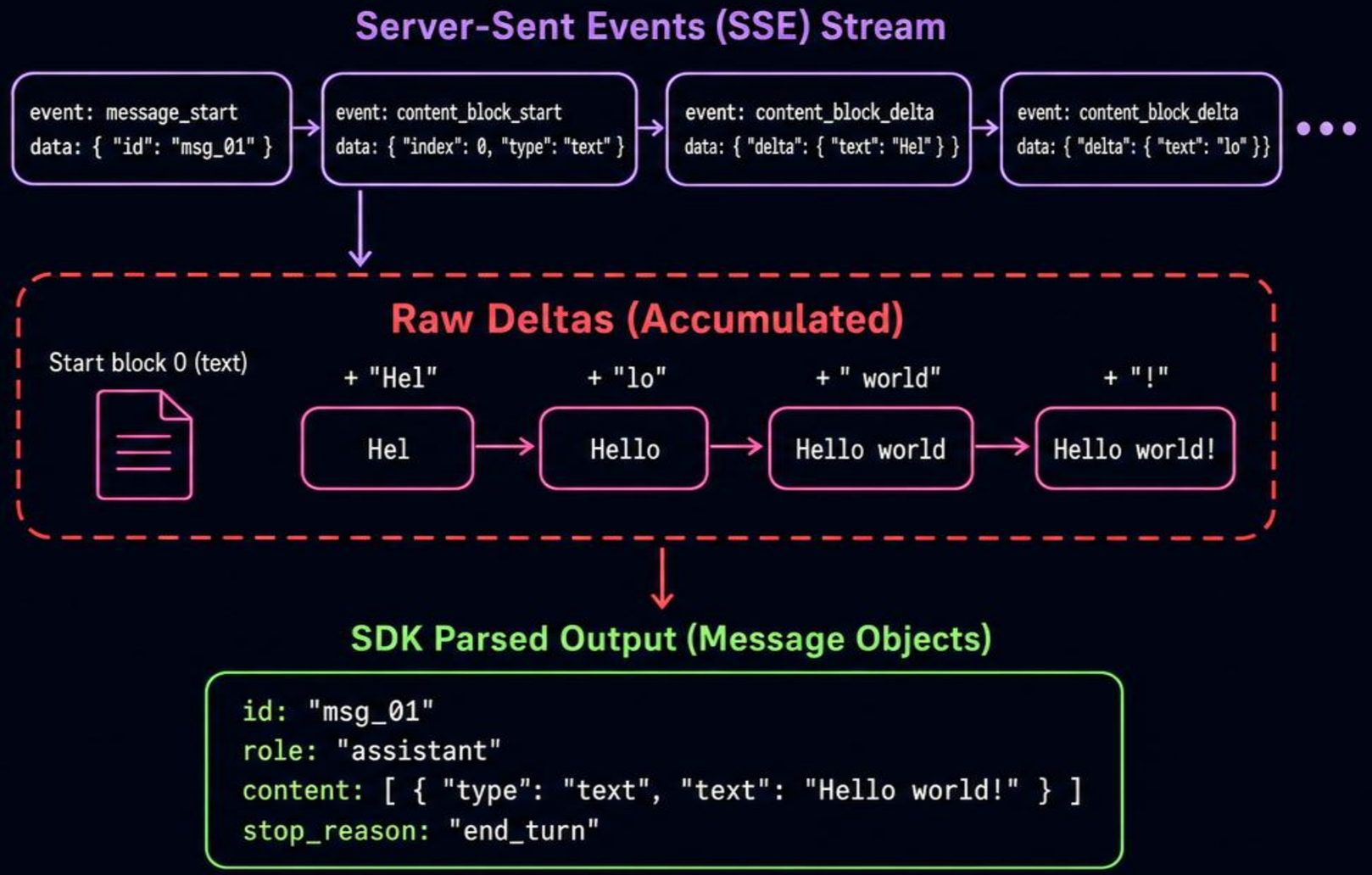
### Raw, you'd accumulate deltas and track block boundaries

You'd handle partial chunks, boundary markers, and event types yourself.

### The SDK assembles them into message objects for you

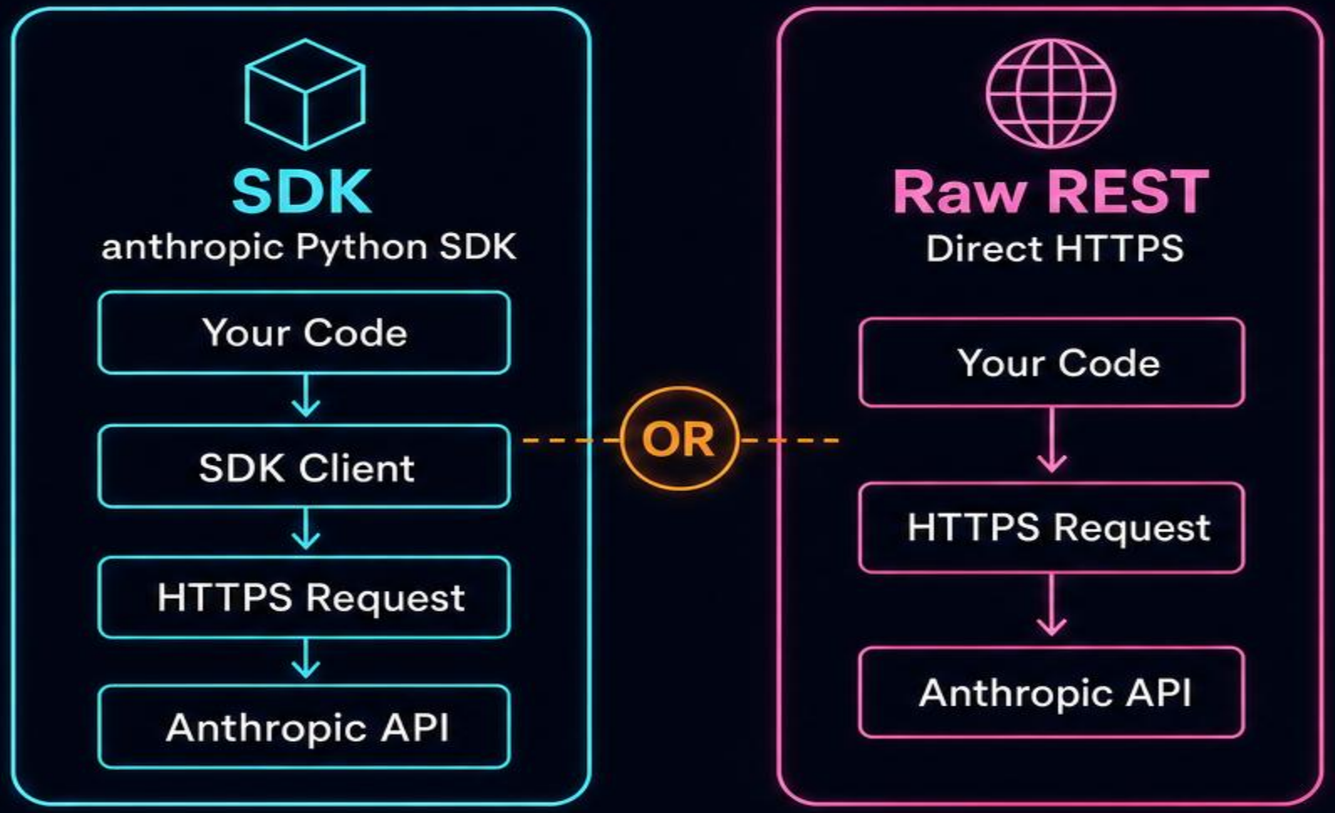
It builds complete messages, attaches metadata, and surfaces them through a simple API.

Same stream on the wire, far less parsing code to own.



# When to Use Raw REST

The thin-wrapper design means either path works. Pick by context.




**Use the SDK**
for speed, typing, and fewer moving parts

**Use raw REST**
in an unsupported language or tiny footprint

**Mix them,**
since both hit the same endpoint

Because they're interchangeable, switching later costs almost nothing.

# A Simple Decision Rule

## Reach for the SDK

This is your default. In Python or TypeScript, the SDK saves auth, retries, typing, and streaming parsing, with almost no downside for most apps.

## Reach for raw REST

Choose it when your language has no official SDK, you need a minimal dependency footprint, or you're debugging behavior right at the wire.

# Convenience Over the Same API



**It manages** auth, retries, typing, and streaming parsing



**It adds no capability** the REST endpoint lacks



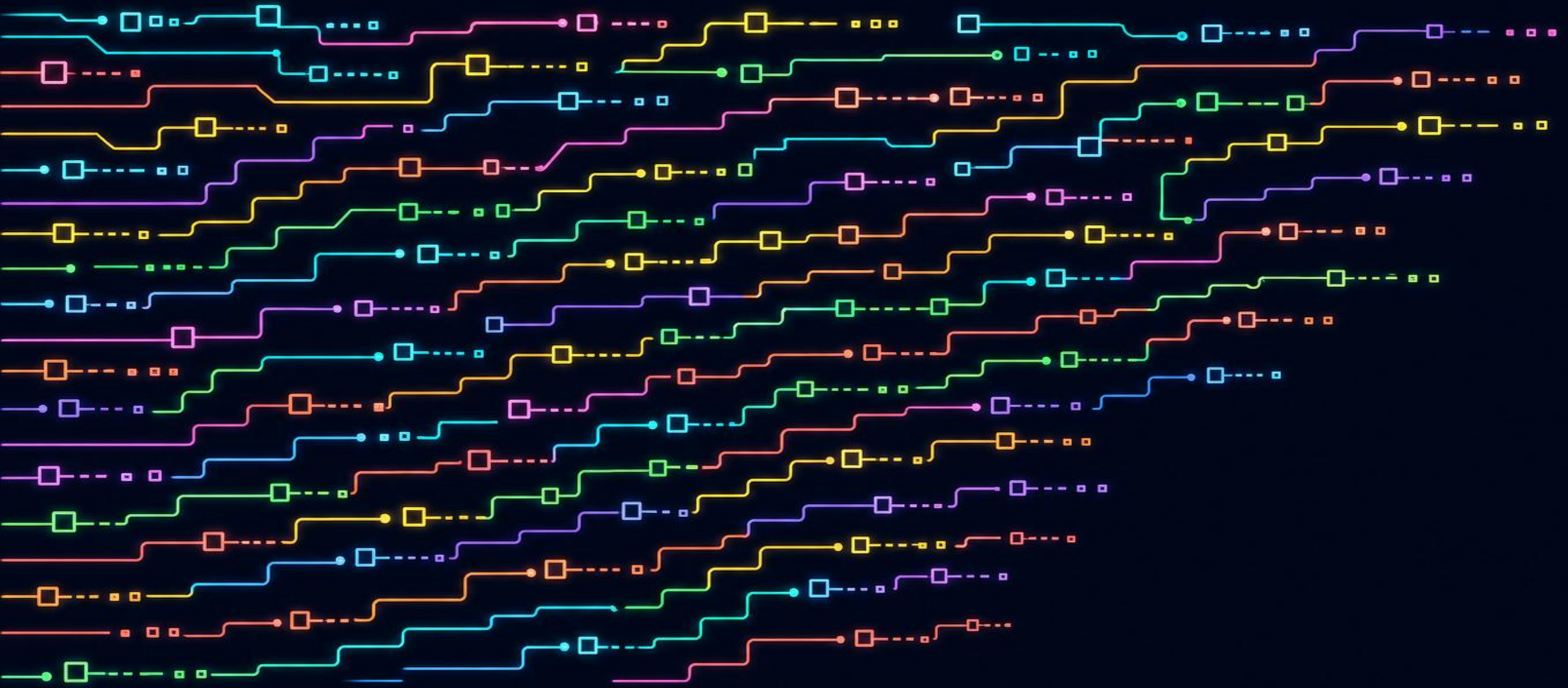
**Unsupported in the API** means unsupported in the SDK



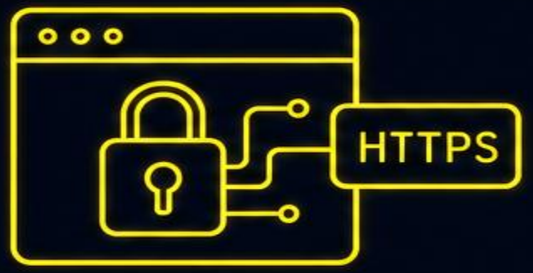
**SDK and raw REST** are interchangeable, so pick by context

# Streaming Over HTTP With SSE

Every call to the Claude API rides over a standard HTTPS request.



# Three Streaming Fundamentals




## HTTPS Request

The Claude API is reached over a normal encrypted HTTP request, so one request and response carries your call.




## Streaming Mode

Setting stream to true tells the server to send the response incrementally as it is generated, not all at once.




## Server-Sent Events

The format Claude uses to push those increments. Named events flow one way, from server to client.

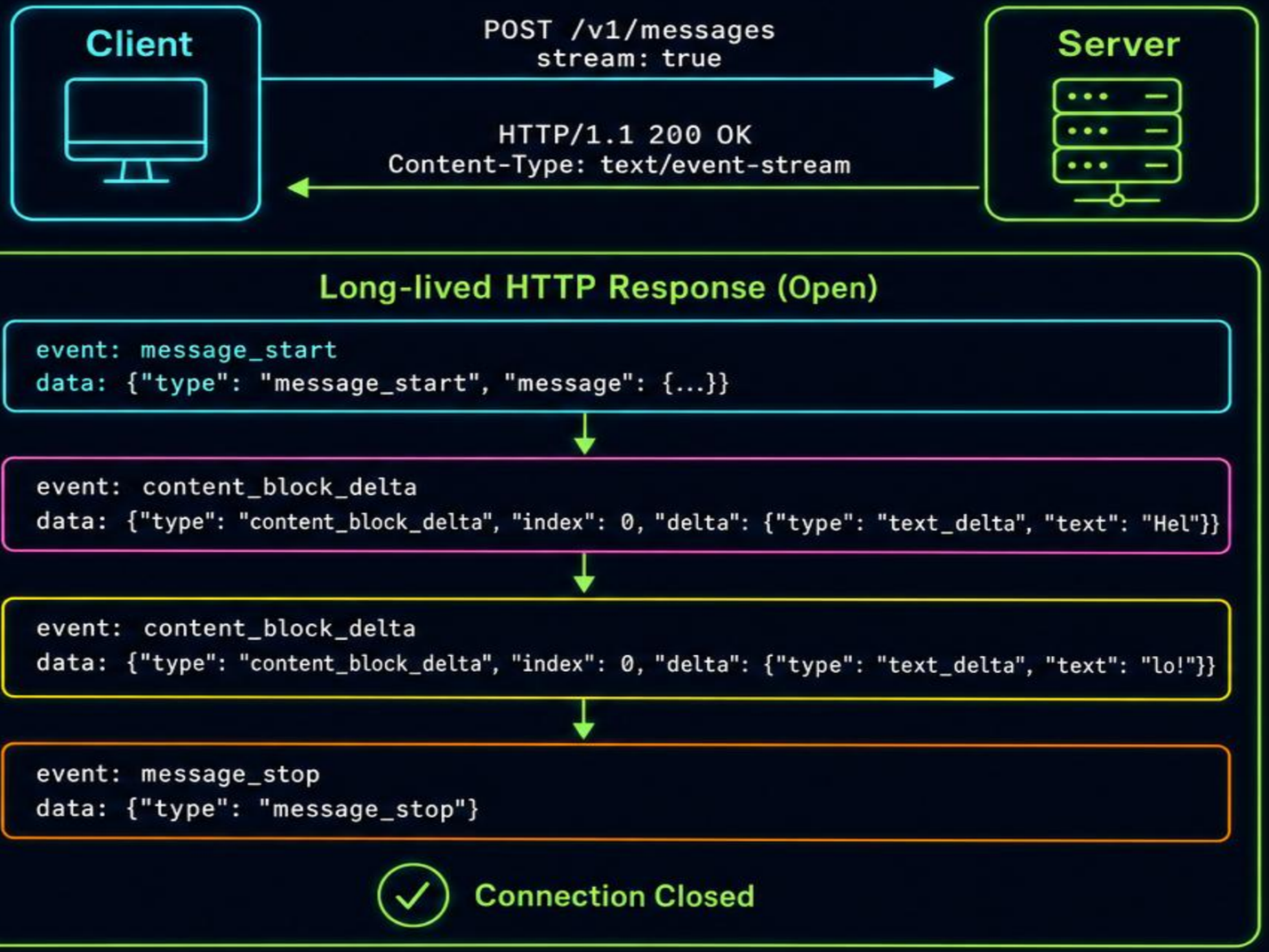
# How a Streaming Request Works

You open the stream with the same request you'd send normally, just with streaming turned on.

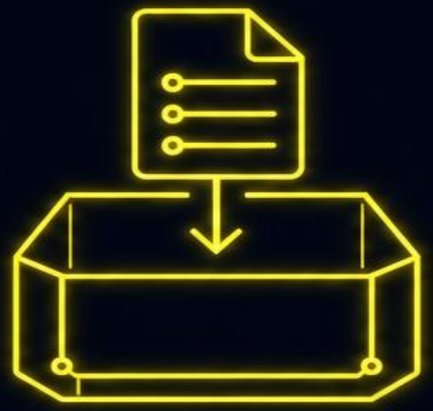
**The server keeps** one HTTP response open and writes to it over time

**Each chunk arrives** as a labeled event with a small JSON payload

**The connection stays** alive until the final event closes it



# Buffered vs. Streamed Responses



## Buffered

The default. The server generates the whole message, then returns it in one response. Simple to handle, but you wait for the full result before anything shows.



## Streamed

With stream true, tokens arrive as they're produced. You show output sooner and handle very long responses that a single blocking call would time out on.

# The Event Sequence Is Fixed

The events don't arrive randomly. They follow one guaranteed order every time.

## STREAM EVENT SEQUENCE

1

**message_start**

type: "message_start"

```
{ "type": "message_start",
  "message": { "id": "msg_01", "role": "assistant" } }
```

2

**content_block_delta**

type: "content_block_delta"

```
{ "type": "content_block_delta", "index": 0,
  "delta": { "type": "text_delta", "text": "Hello" } }
...
```

3

**message_stop**

type: "message_stop"

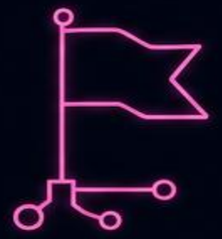
```
{ "type": "message_stop" }
```



**A message start event** opens the stream with an empty message



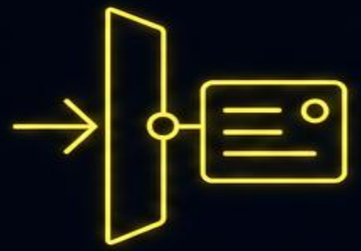
**Content block events** carry the actual text and tool deltas



**A message stop event** marks the response complete

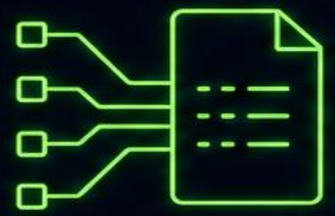
Knowing the order means you always know what to expect next.

# Events That Bracket the Stream



## message_start

Opens the stream and carries a message with empty content. The first event you receive, never the text itself.



## content_block_delta

The workhorse events. Each one adds a small piece of text or tool input to the current block.



## message_stop

The final event. Once you see it, the response is complete and the connection can close.

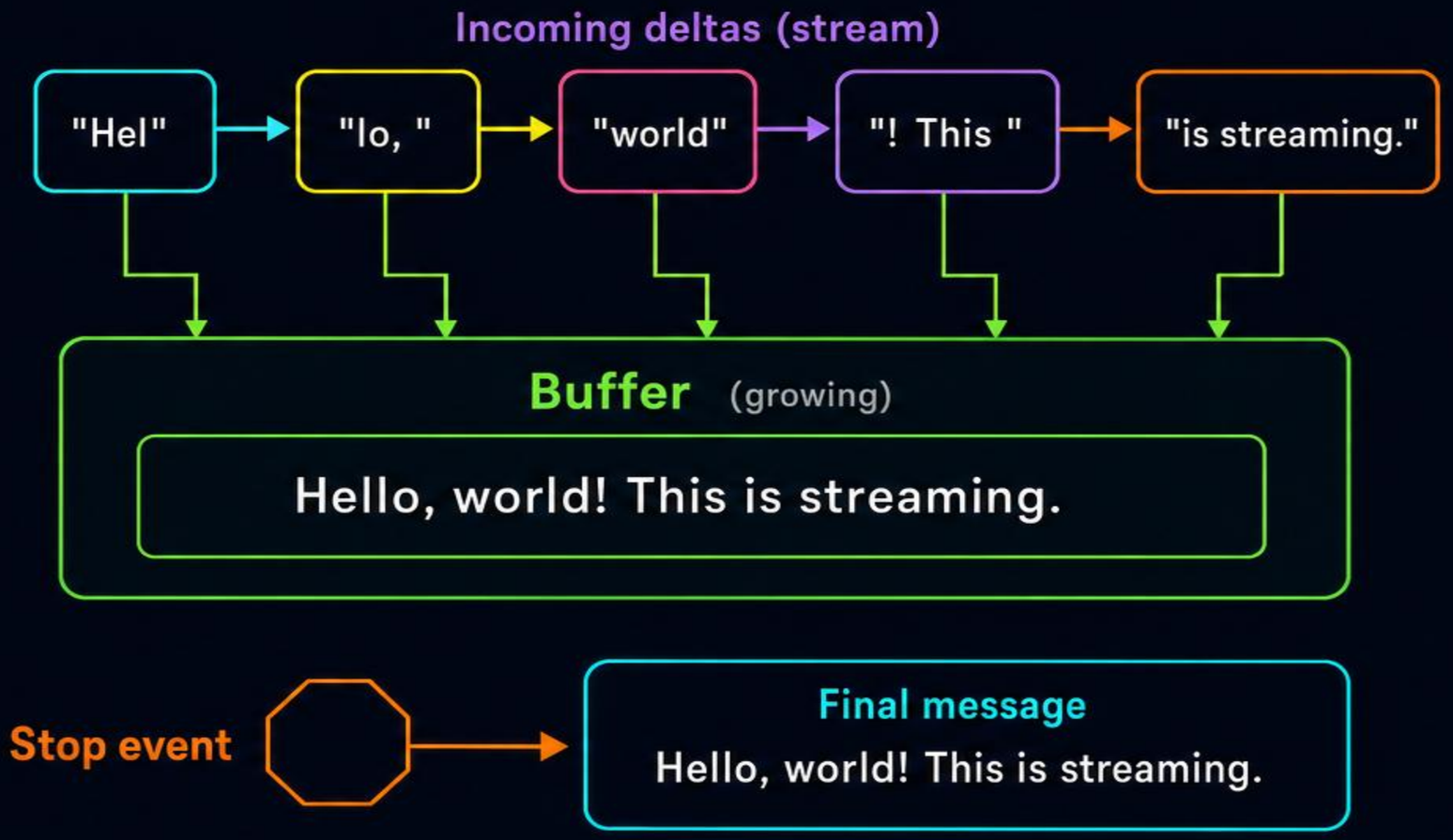
# Accumulating Deltas Into a Response

A streamed response arrives in fragments, so your code stitches it back together.

**Start an empty buffer**
when the message start event lands

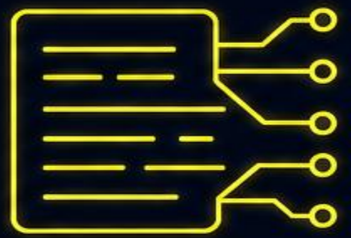
**Append each delta**
to the buffer as it comes in

**Close out the block**
when its stop event arrives



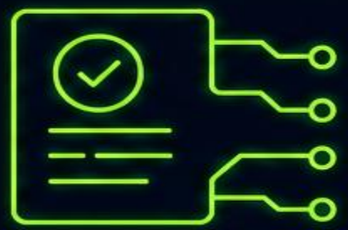

**By the final event, your buffer holds the complete message.**

# Three Delta-Level Terms



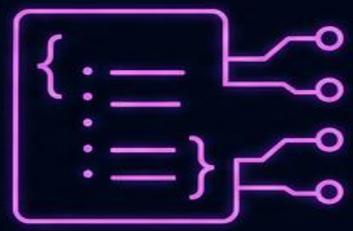
## content_block_delta

An event adding a chunk of output to one content block. Most streamed text arrives as a series of these.



## message_delta

Late events carrying top-level updates, like the stop reason and token usage for the whole response.

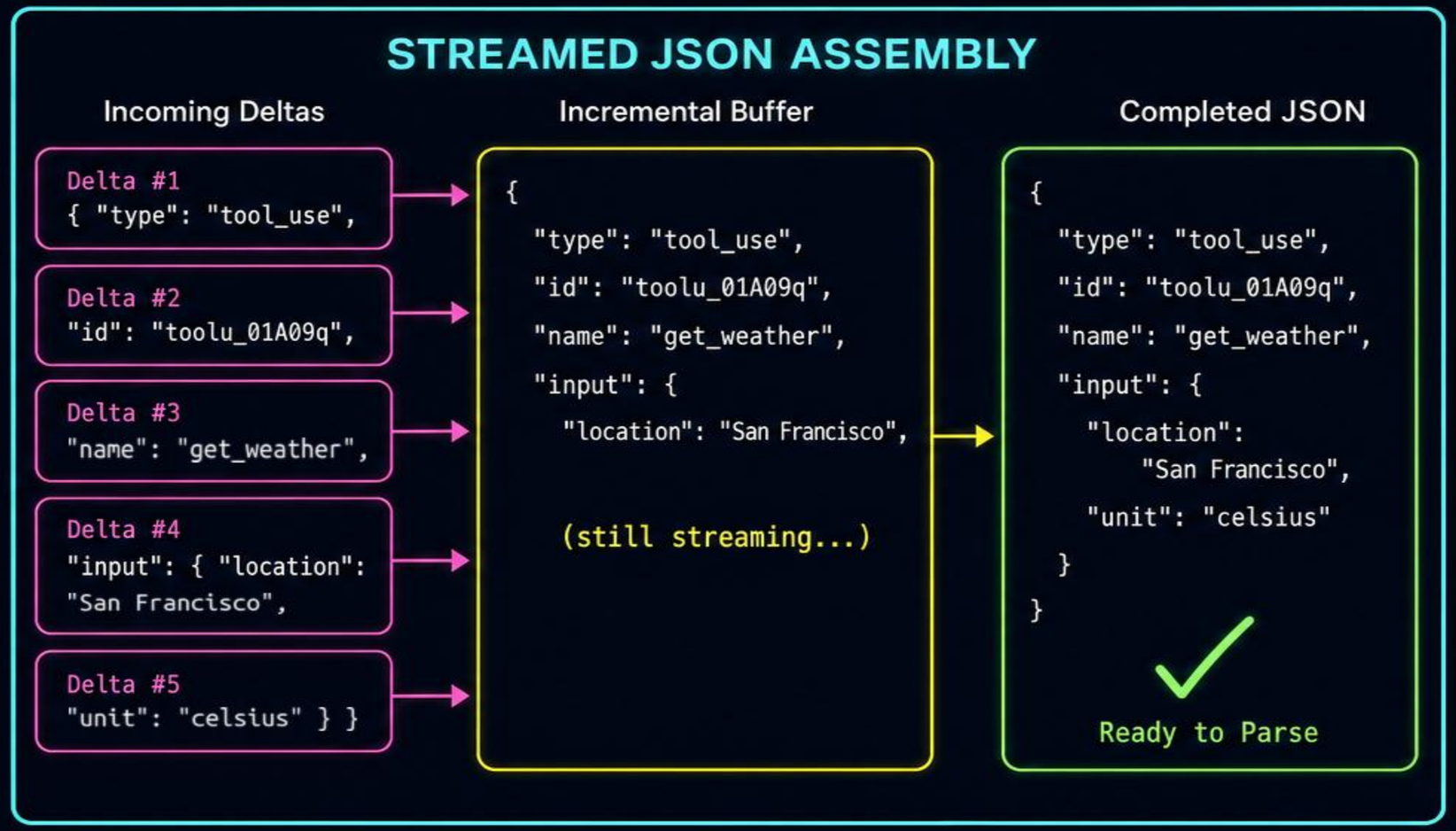



## input_json_delta

The delta type for tool calls. It carries partial JSON that builds a tool’s arguments piece by piece.

# Partial Tool Arguments

When Claude calls a tool, its arguments don't arrive all at once.




**They stream** as partial JSON fragments across several deltas

**A half-built fragment** won't parse alone, so don't try mid-stream

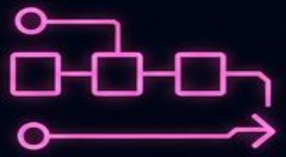
**Parse only** once the block's stop event arrives

At that point the tool input is always a complete, valid object.

# How Claude Streams



**Turn on streaming** and read events on a single open connection



**Events follow a fixed order** from message start to message stop



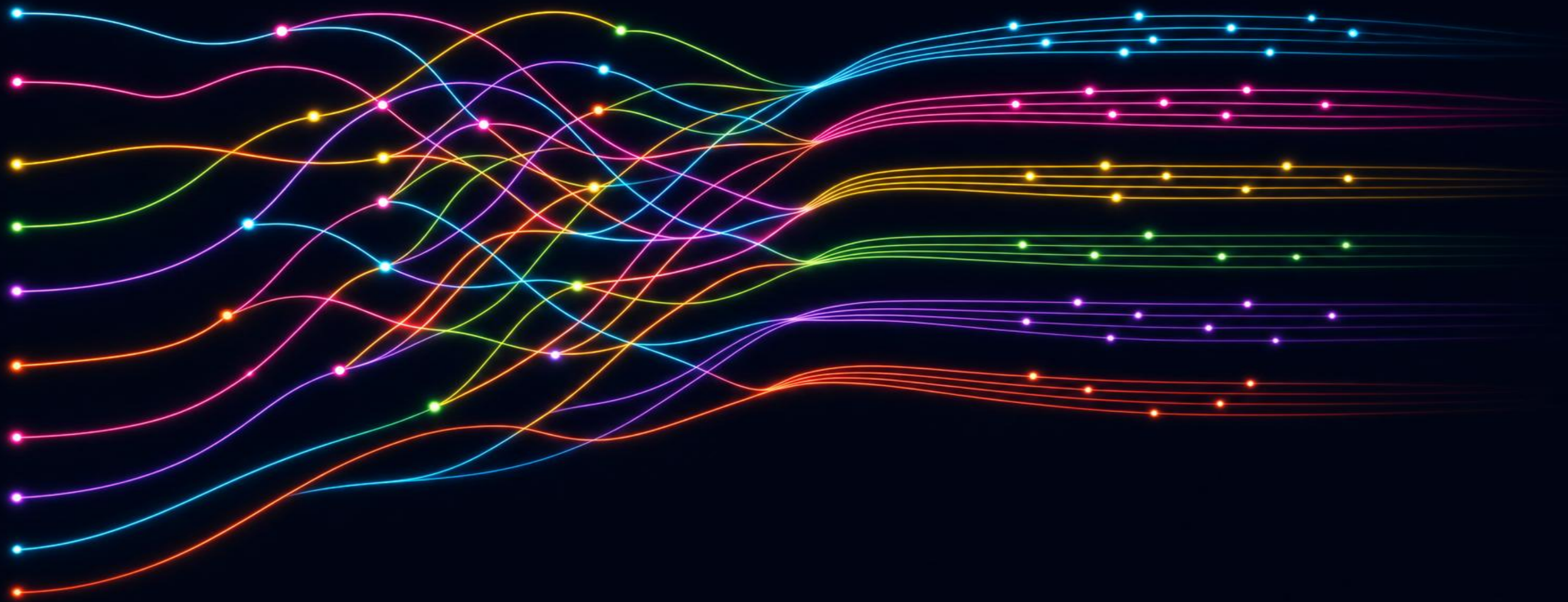
**Accumulate deltas** into a buffer to rebuild the full response



**Tool arguments stream** as partial JSON, parsed only at block close

# Websockets vs SSE for Claude

Developers often assume a real-time stream from Claude runs over a websocket. It doesn't.



# Three Connection Concepts



## WebSocket

A persistent, full-duplex connection. Once open, both client and server can send messages at any time, in either direction.



## Server-Sent Events

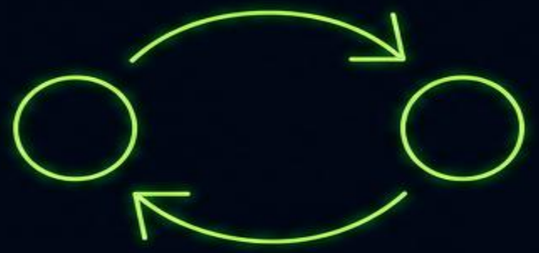
A one-way stream. The server pushes events to the client over one HTTP response, but the client can't send back on it.



## Full-Duplex

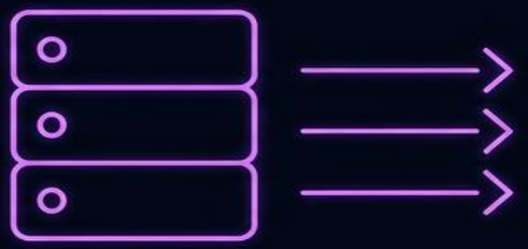
Traffic flowing both ways at once, like a phone call. A one-way channel allows only a single direction.

# WebSocket vs. SSE



## WebSocket

Bidirectional and persistent. Built for apps where both sides push messages constantly, like live chat or multiplayer games. More setup, and its own protocol handshake to open.



## Server-Sent Events

One-directional, server to client, riding over plain HTTP. Simpler to run and friendly to firewalls. This is what the Claude API uses for streaming.

# The Websocket Misconception

Here's the myth worth clearing up front. Claude does not stream over a websocket.



**Token streaming uses server-sent events, not a two-way socket**

Claude sends chunks via SSE over HTTP/HTTPS.




**You never open a websocket to talk to the Claude API**

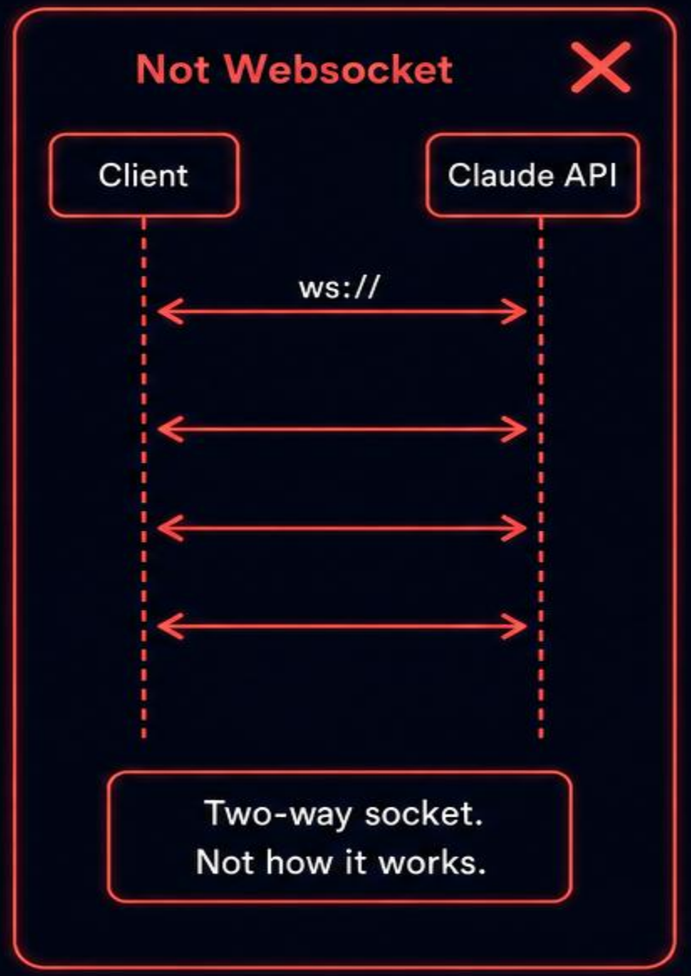
The API is HTTP-based only.
No ws://, ever.

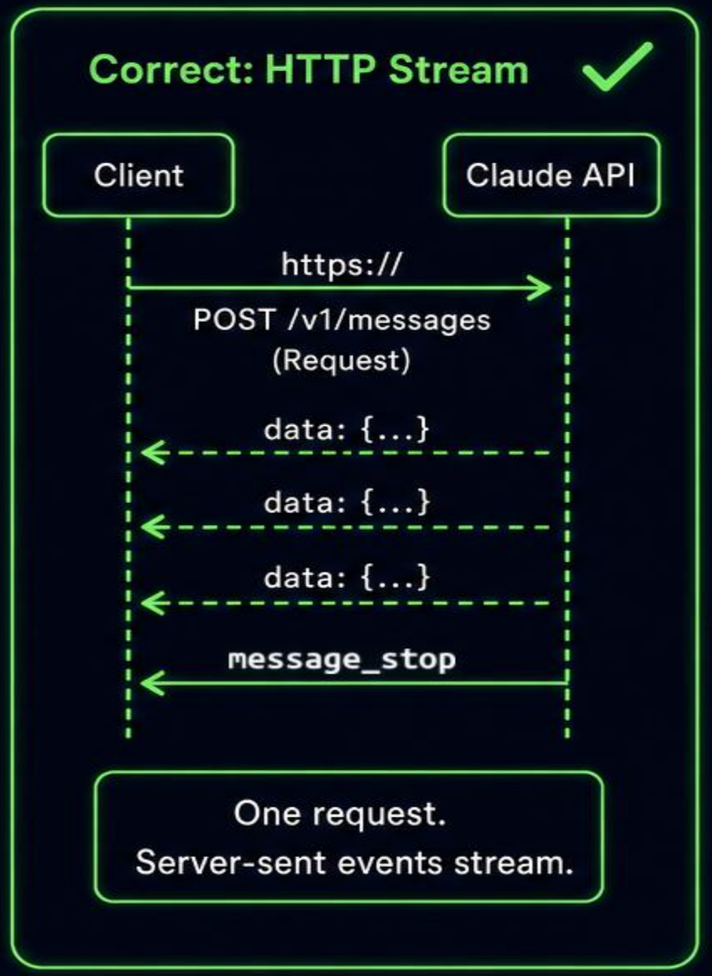


**Every call, streaming or not, is a plain HTTPS request**

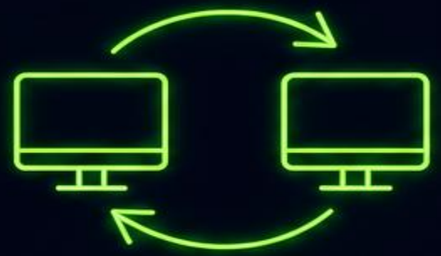
One request in, one response (or stream) out.

If a design assumes a websocket to Claude, that assumption is wrong.



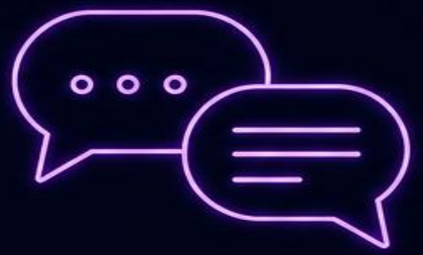



# Where Websockets Fit



## Good fit

Apps needing constant two-way traffic, like live collaboration or multiplayer games where both sides push continuously.



## Also a fit

A chat server that both sends and receives messages in real time over one persistent, open channel.

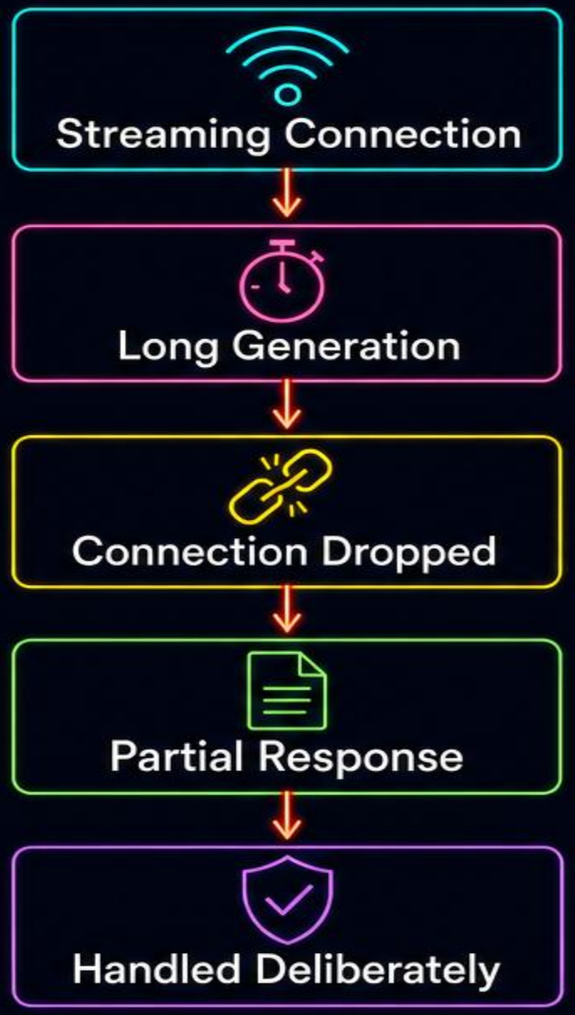


## Not how you reach Claude

The Claude API is request and response over HTTP, with streaming layered on as server-sent events.

# Planning for a Dropped Stream

A long-lived streaming connection can drop, so build expecting it.




**The SDK** sets a generous request timeout, around ten minutes by default

**Very long** generations have to stream, or a blocking call risks timing out

**A dropped** connection mid-stream means a partial, incomplete response

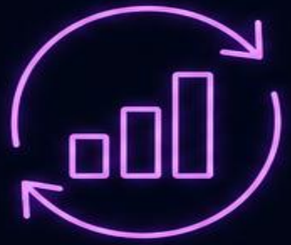
**Treat the stream as something that can fail, and handle it deliberately.**

# Three Connection Concerns



## Timeouts

A ceiling on how long a request can run before the client gives up. Large streamed responses can approach it.



## Retries With Backoff

Automatically re-sending a failed request, waiting longer between tries. The SDK retries transient errors a couple of times by default.



## Ping and Error Events

Events that can interleave with your stream. A ping is a keep-alive to ignore. An error signals a problem.

# Handling Ping and Error Events

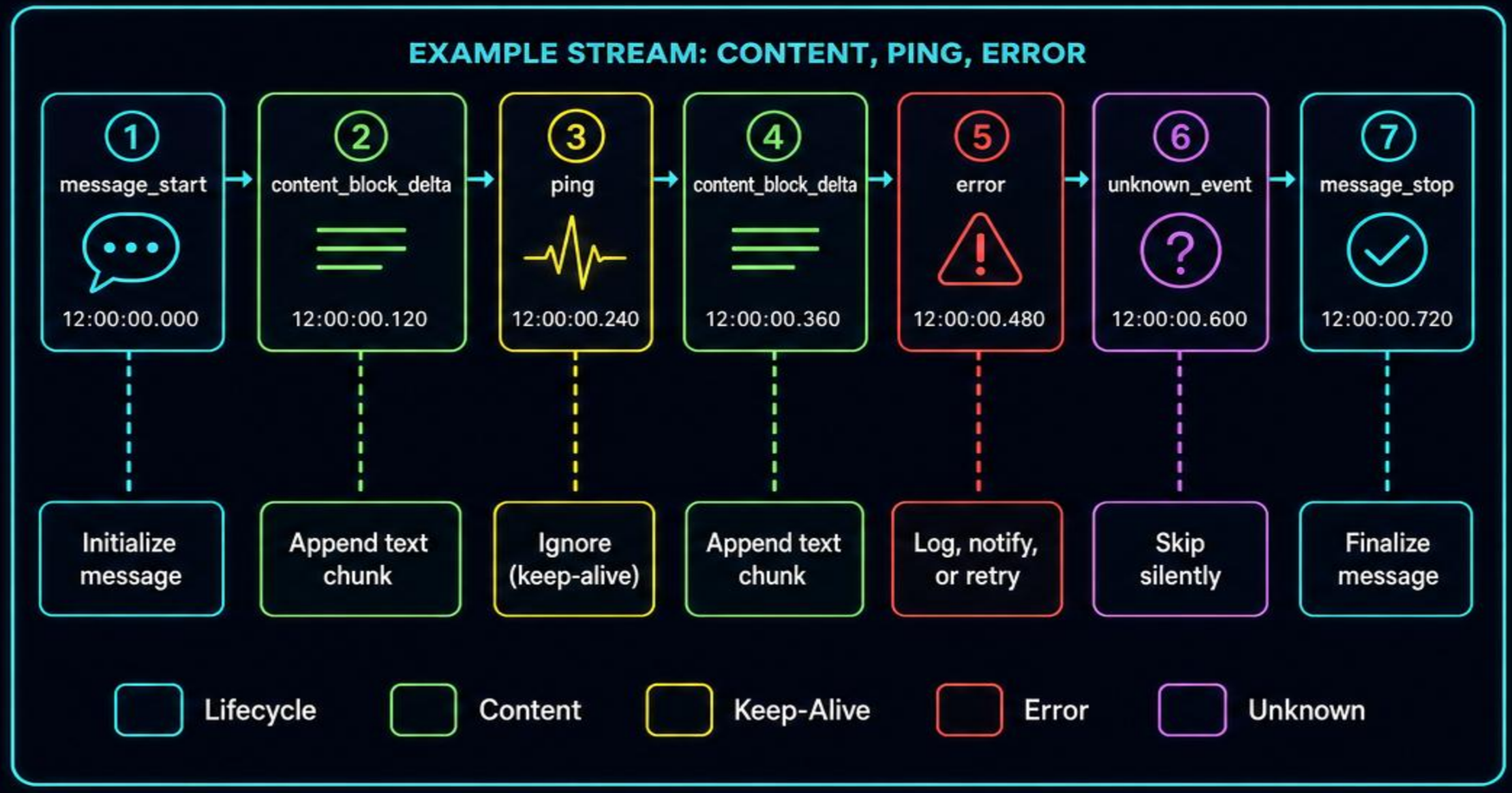
Your stream carries more than just content events, so your reader handles the extras.

**A ping event**
is a keep-alive with
no payload, safe to ignore

**An error event**
can arrive mid-stream
and needs real handling

**Unknown event types**
should be skipped,
not treated as failures

Read defensively, and an interleaved event won't crash your loop.



# Retry It vs. Fix It



## Transient errors

Timeouts, rate limits, network blips, and 5xx server errors. These are expected to clear on their own, so retry with exponential backoff.



## Permanent errors

Bad requests, invalid schemas, and auth failures. Retrying won't help, because the input itself is wrong. Fix the request instead of retrying.

# SSE, Not Websockets



**A websocket** is a persistent, two-way connection for constant back-and-forth



**Claude's streaming** is one-way server-sent events over plain HTTPS



**Plan for dropped streams** with sensible timeouts



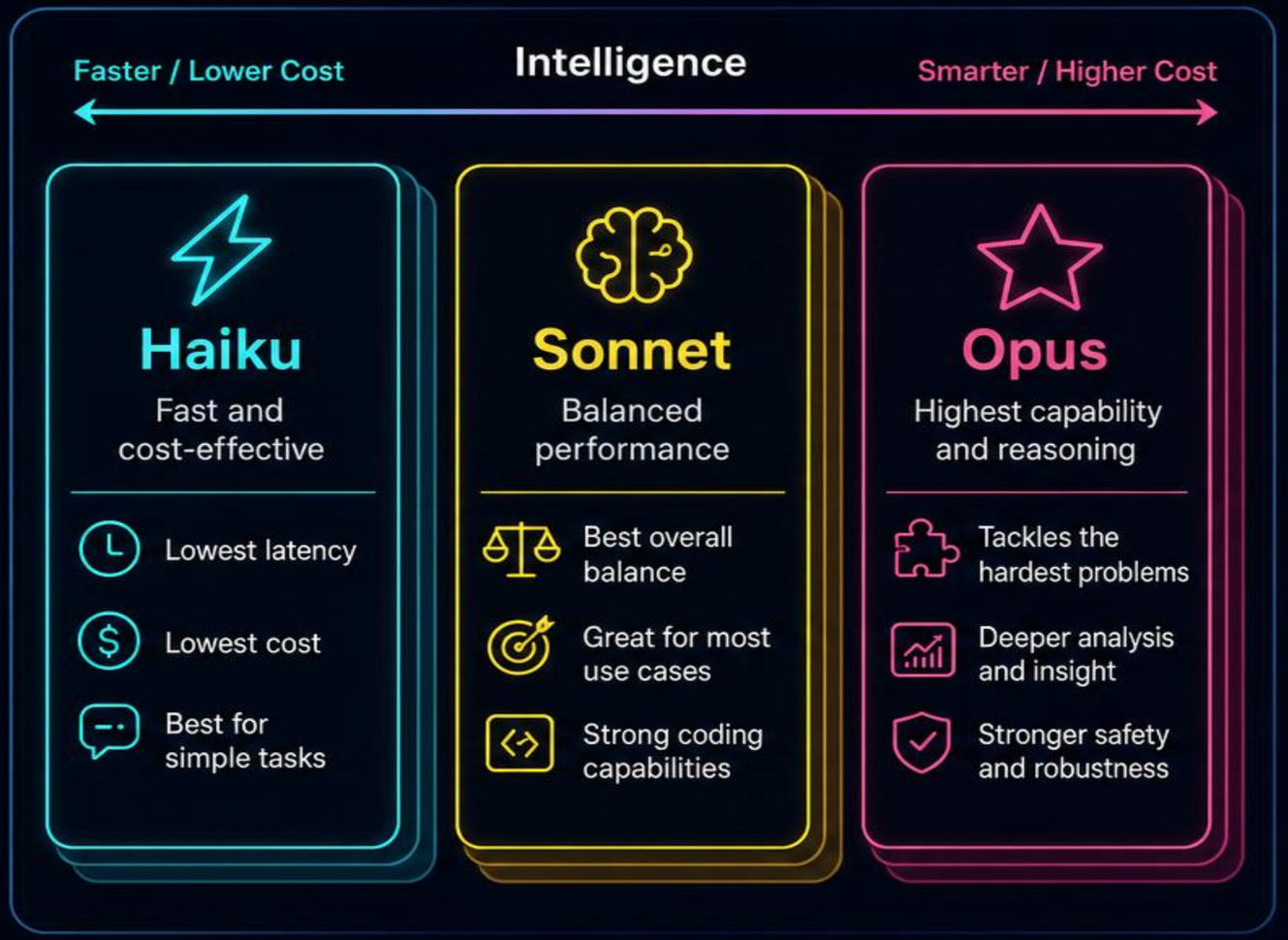
**Retry transient errors** with backoff, but fix permanent ones



**Ignore ping events** and handle error events on the stream

# The Three Model Tiers

Claude comes in a family of models, and picking the right one is a real engineering decision.




| Which to use, and when | Haiku | Sonnet | Opus |
|---|---|---|---|
| Chat and assistants | ✓ | ✓ | ✓ |
| Code generation | ✓ | ✓ | ✓ |
| Data analysis and research | – | ✓ | ✓ |
| Complex reasoning | – | – | ✓ |
| Mission-critical decisions | – | – | ✓ |

Start with **Sonnet.**
Scale up to **Opus** if the task demands it.

# Three Model Tiers



## Opus

The most capable tier for the hardest reasoning, coding, and agentic work, at the highest per-token cost and generally higher latency.



## Sonnet

The balanced tier, trading capability against speed and cost. It targets high-volume production work that needs strong quality.



## Haiku

The fastest, lowest-cost tier for high-throughput or latency-sensitive simple tasks like classification, routing, and extraction.

# Today's Named Models

Each tier has a specific current model you name in code.



**Opus 4.8**, the recommended default for complex work



**Sonnet 5**, the best combination of speed and intelligence



**Haiku 4.5**, the fastest model with near-frontier smarts



**Claude Fable 5**

Most capable model for long-running agents






**Claude Opus 4.8**

Premium power for the toughest problems






**Claude Sonnet 5**

Balanced performance for most workloads





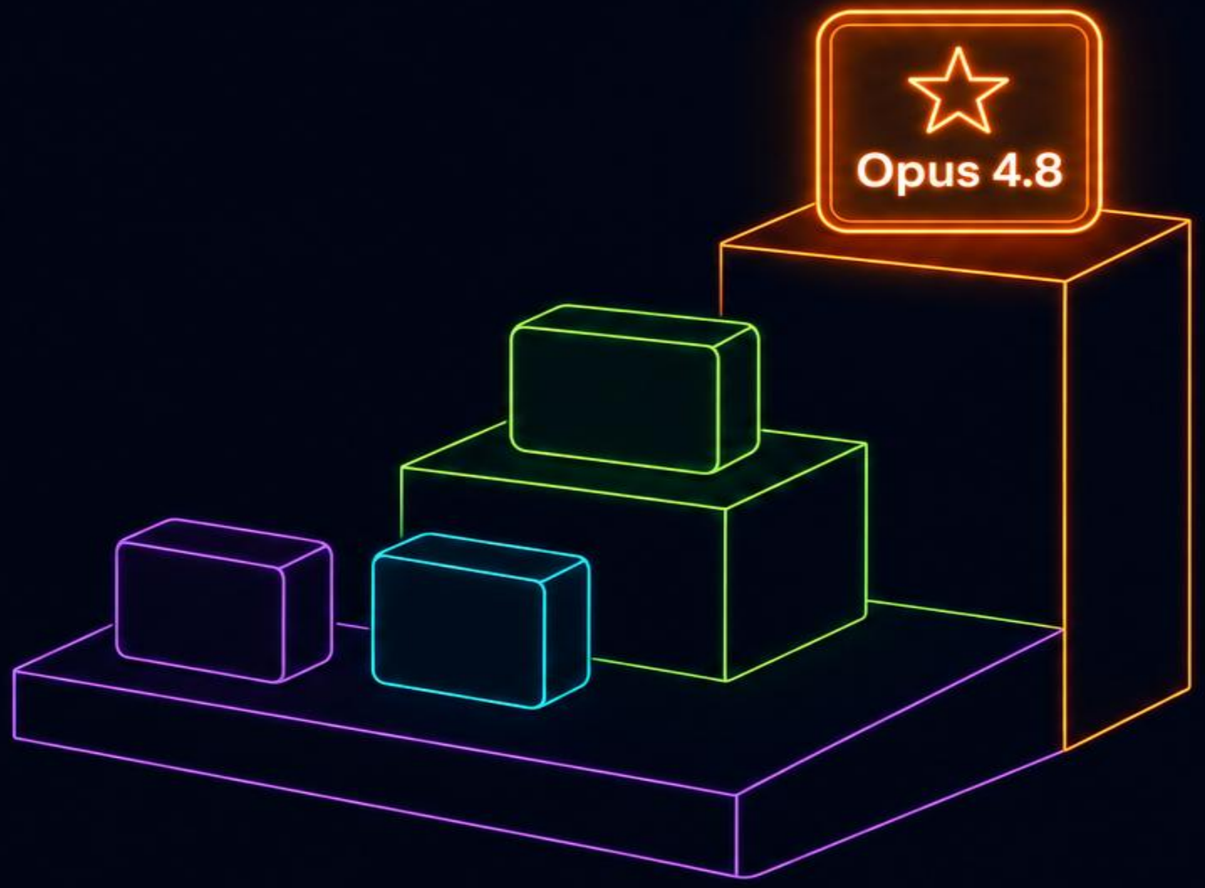

**Claude Haiku 4.5**

Fast, efficient, and built for scale



# Opus, the Everyday Capability Ceiling

Opus is where you go when the task is genuinely hard.




**It leads** on complex reasoning and multi-step coding

**It anchors** long-running autonomous agents and large refactors

**Opus 4.8** is Anthropic's recommended starting point

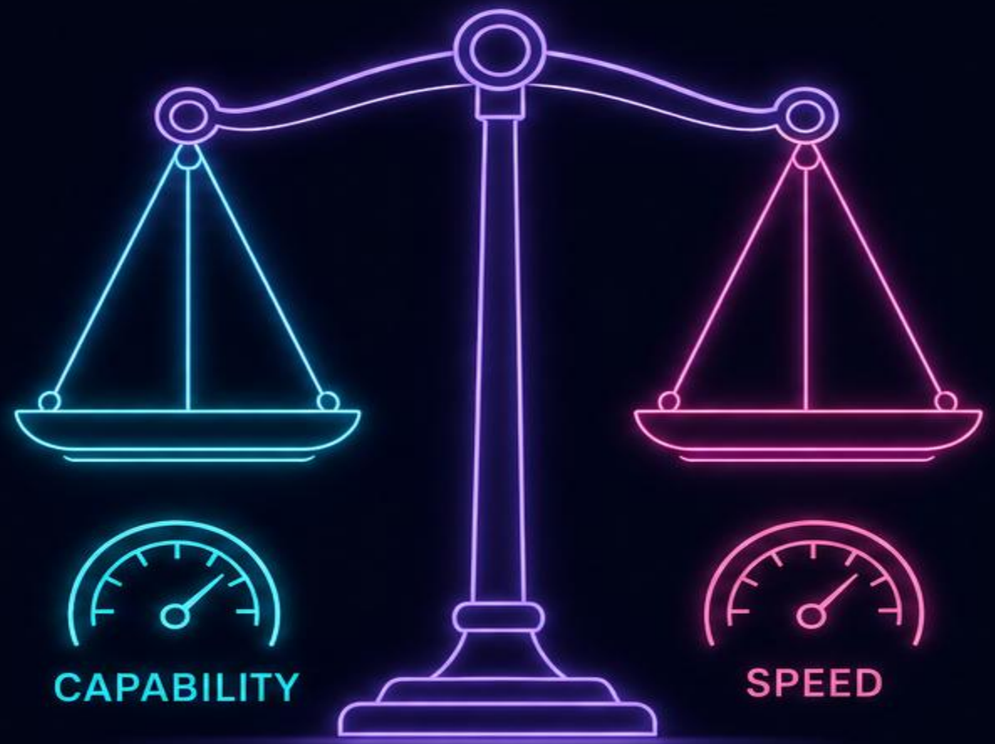
# Sonnet, the Balance Point

Sonnet sits right between raw capability and raw speed.

**Strong quality** for high-volume production workloads at scale

**Much lower cost** and latency than the Opus tier

**Sonnet 5** balances speed and intelligence well



# Haiku, Fast and Economical



## Fastest Tier

Haiku 4.5 is the fastest model with near-frontier intelligence, built for real-time apps and high-volume work.



## Lowest Cost

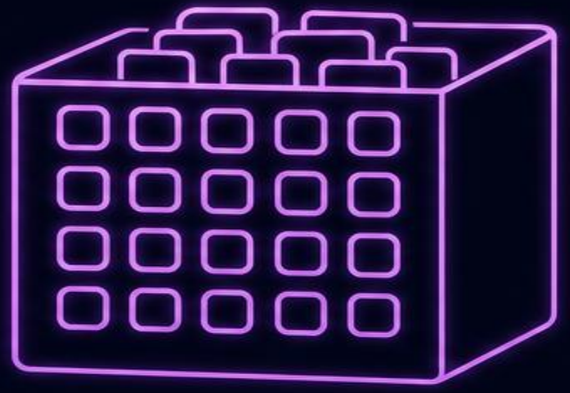
It carries the lowest per-token price of any tier, so cost-sensitive deployments naturally favor it.



## Simple Tasks

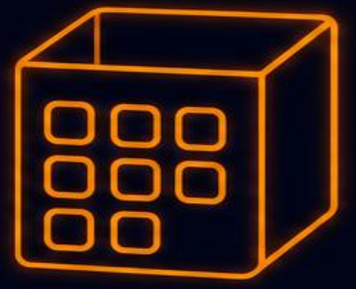
Classification, routing, extraction, and sub-agent steps are its sweet spot at high throughput and scale.

# Context Windows



## One Million Tokens

Fable 5, Opus 4.8, and Sonnet 5 each attend to up to one million tokens, a very large working space for big documents and long agent runs.

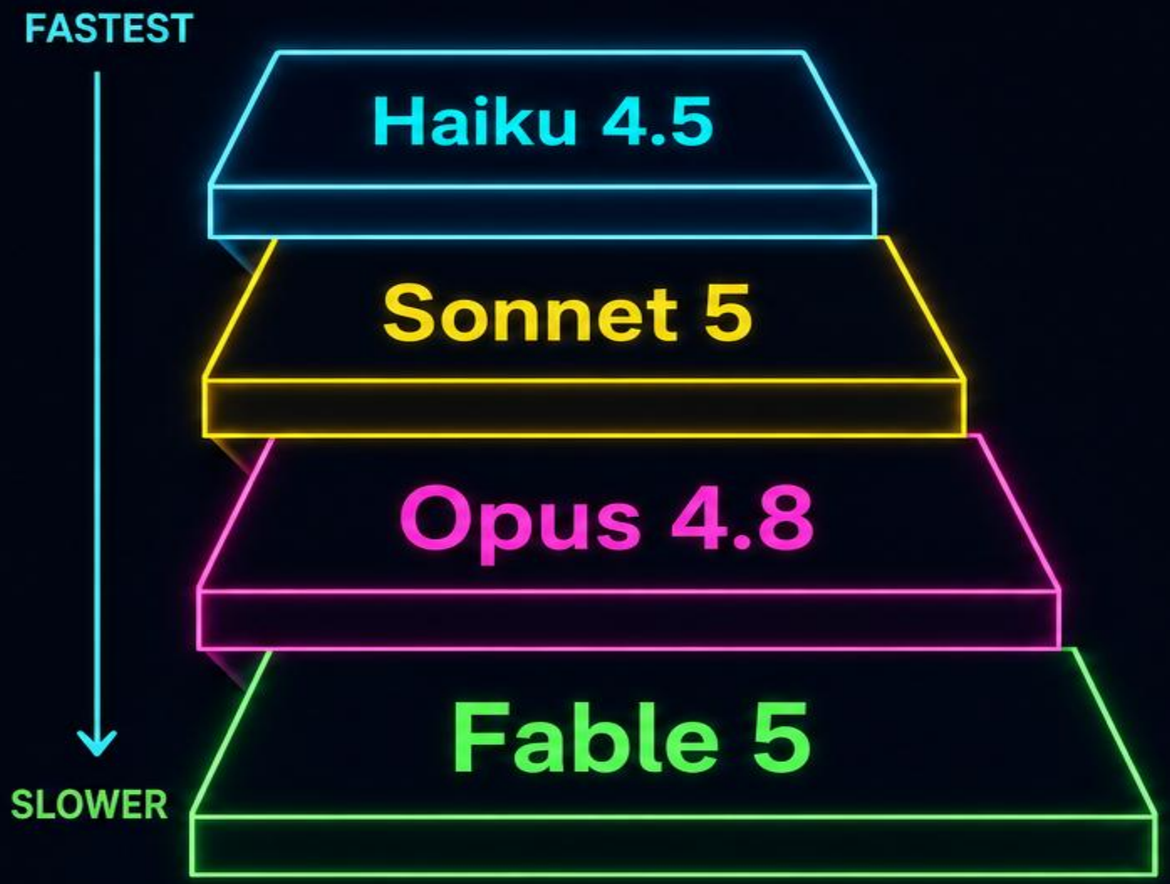


## Two Hundred Thousand

Haiku 4.5 attends to up to 200,000 tokens, a fifth of the larger window, so the same oversized prompt can fit Opus yet overflow Haiku.

# The Latency Ladder

Response speed climbs steadily as you move down the tiers.




**Haiku 4.5 is the fastest to respond**

Ideal for real-time UX, quick replies, and high-throughput workloads.

**Sonnet 5 is fast, just a step behind Haiku**

Great balance of speed and capability for most everyday tasks.

**Opus 4.8 is moderate, trading speed for depth**

Best for complex reasoning, long-form analysis, and high-stakes decisions.

**Fable 5 is the slowest, since it spends the most time reasoning.**

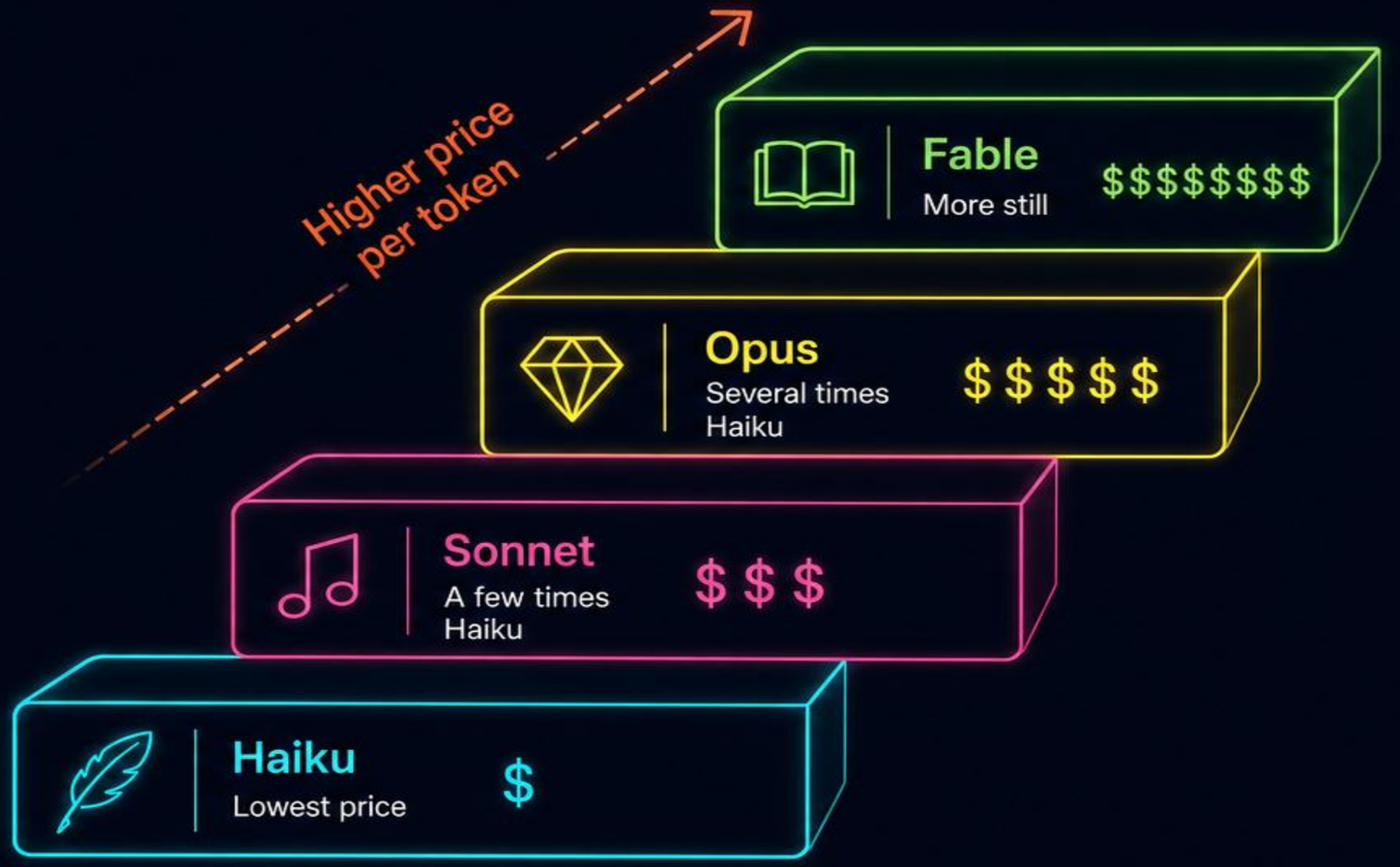
# The Cost Ladder

Per-token price rises as you climb the capability ladder.

**Haiku** sits at the lowest per-token price of the four

**Sonnet** costs a few times Haiku's input rate

**Opus** costs several times Haiku's input rate, Fable more still



# The Tiers at a Glance



**Opus 4.8** leads on capability at the highest cost



**Sonnet 5** balances quality, speed, and cost



**Haiku 4.5** is fastest and cheapest at 200k context



**Fable 5** sits above them for the very hardest work

# Matching a Tier to Your Workload

Knowing the three tiers is only half the job. The real skill is picking the right one for a given task.

## 1 Characterize the workload

**Goal**
What is the task trying to achieve?

**Predictability**
Can the path be pre-scripted?

**Tool needs**
Static set or dynamic choice?

**Risk tolerance**
How much risk is acceptable?

**Change rate**
How often does the environment shift?

## 2 Recommended tier

Best fit for this workload



**Agent**
Recommended
Confidence: 0.87

**Why this pick?**

- Goal is open-ended
- Path cannot be pre-scripted
- Dynamic tool selection needed
- Medium risk tolerance
- Environment changes frequently

## 3 Tier options and example workloads

**Chatbot**
Lowest autonomy
Responds to input. Does not act.

Fits workloads like:
- FAQ assistant
- Content summarizer
- Draft email helper

**Workflow**
Deterministic
Fixed sequence. Developer controls flow.

Fits workloads like:
- Report generation
- Data pipeline
- Onboarding flow

**Agent**
Highest autonomy
Selects tools dynamically at inference time.

Fits workloads like:
- Research analyst
- Incident responder
- Strategic planner

# Three Constraints to Read



## Task Difficulty

How hard the reasoning or coding is.
Harder problems push you toward Opus,
simpler ones toward Haiku.



## Latency Budget

How fast the response has to come back.
Tight real-time budgets favor the
faster, lighter tiers.



## Cost Ceiling

How much you can spend per call at volume.
High throughput at low cost points
toward Haiku.

# The Guiding Principle

One rule sits underneath every model choice.



## Pick the lightest tier that clears the quality bar

Start simple. Upgrade only when evidence shows you must.



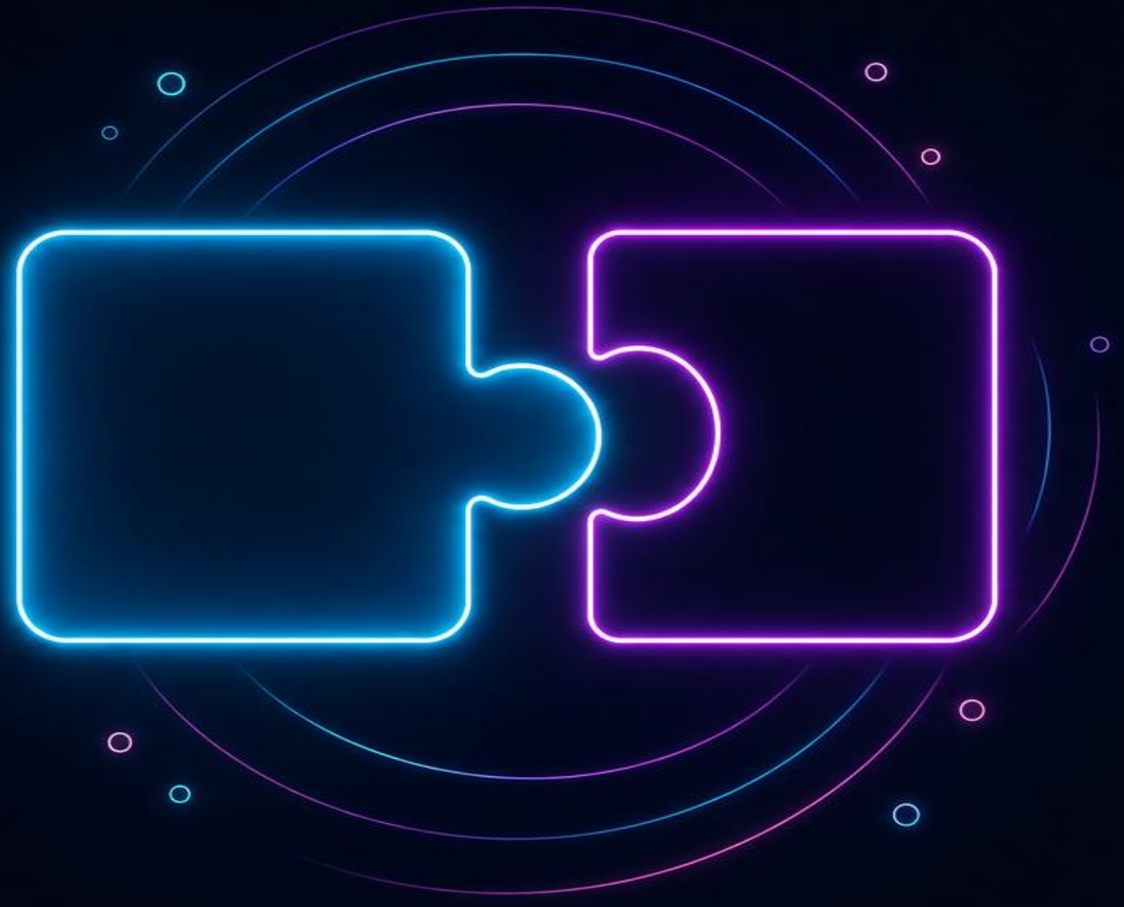
## Don't pay for capability the task doesn't need

Right-size to the job. Save cost, latency, and operational risk.



## Don't starve a hard task with a tier too small

Hard goals need autonomy, tools, and the headroom to adapt.



The goal is fit, not the most powerful model you can reach for.

# Two Selection Strategies



## Start Light, Upgrade

Begin with Haiku for prototyping, tight latency, or high volume. Move up a tier only where you hit a real capability gap.

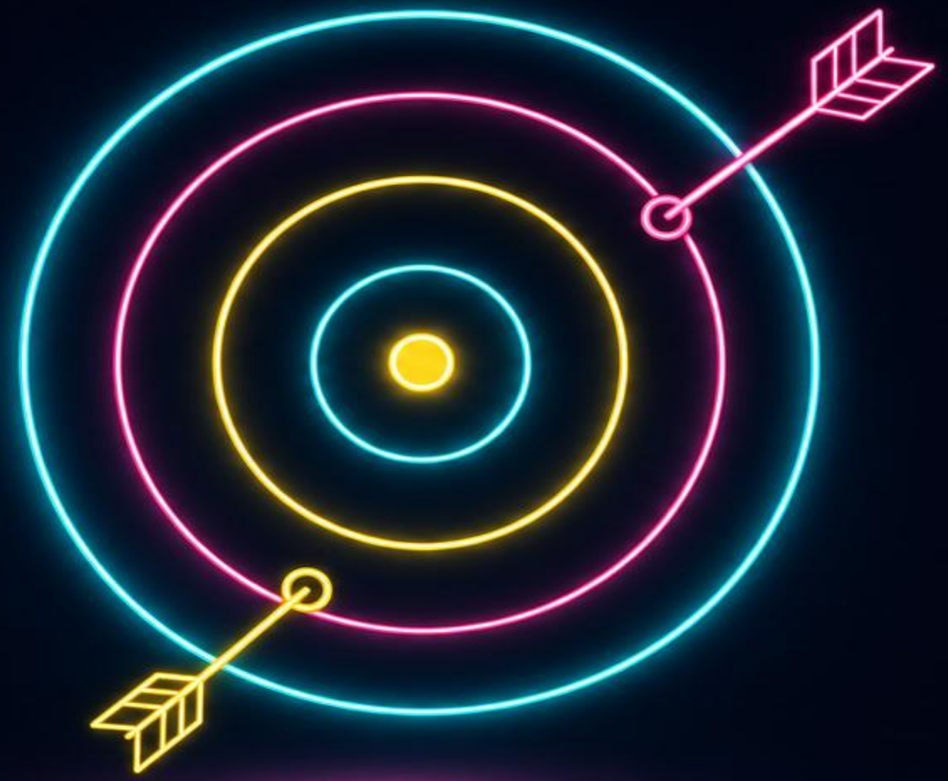


## Start Capable, Optimize Down

Begin with Opus for complex reasoning where accuracy leads. Once it works, step down to a lighter tier to cut cost and latency.

# Two Failure Modes

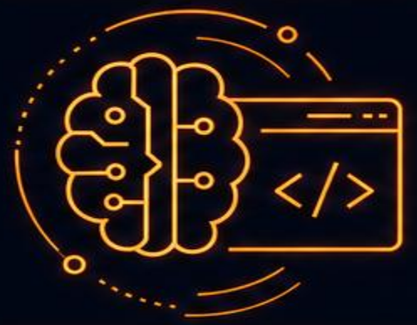
There are two classic ways developers get tier selection wrong.



**Over-provisioning,** defaulting to Opus everywhere and wasting cost and latency on simple work

**Under-serving,** always grabbing Haiku and starving tasks that genuinely need more capability

# Where Each Tier Wins



## Opus Wins

The hardest reasoning tasks, large-scale refactors, multi-hour autonomous coding agents, and vision-heavy or computer-use workflows.



## Sonnet Wins

High-volume production coding, data analysis, content creation, and everyday agentic tool use at real scale.



## Haiku Wins

Real-time apps, high-volume classification, routing, extraction, and lightweight sub-agent steps within a larger job.

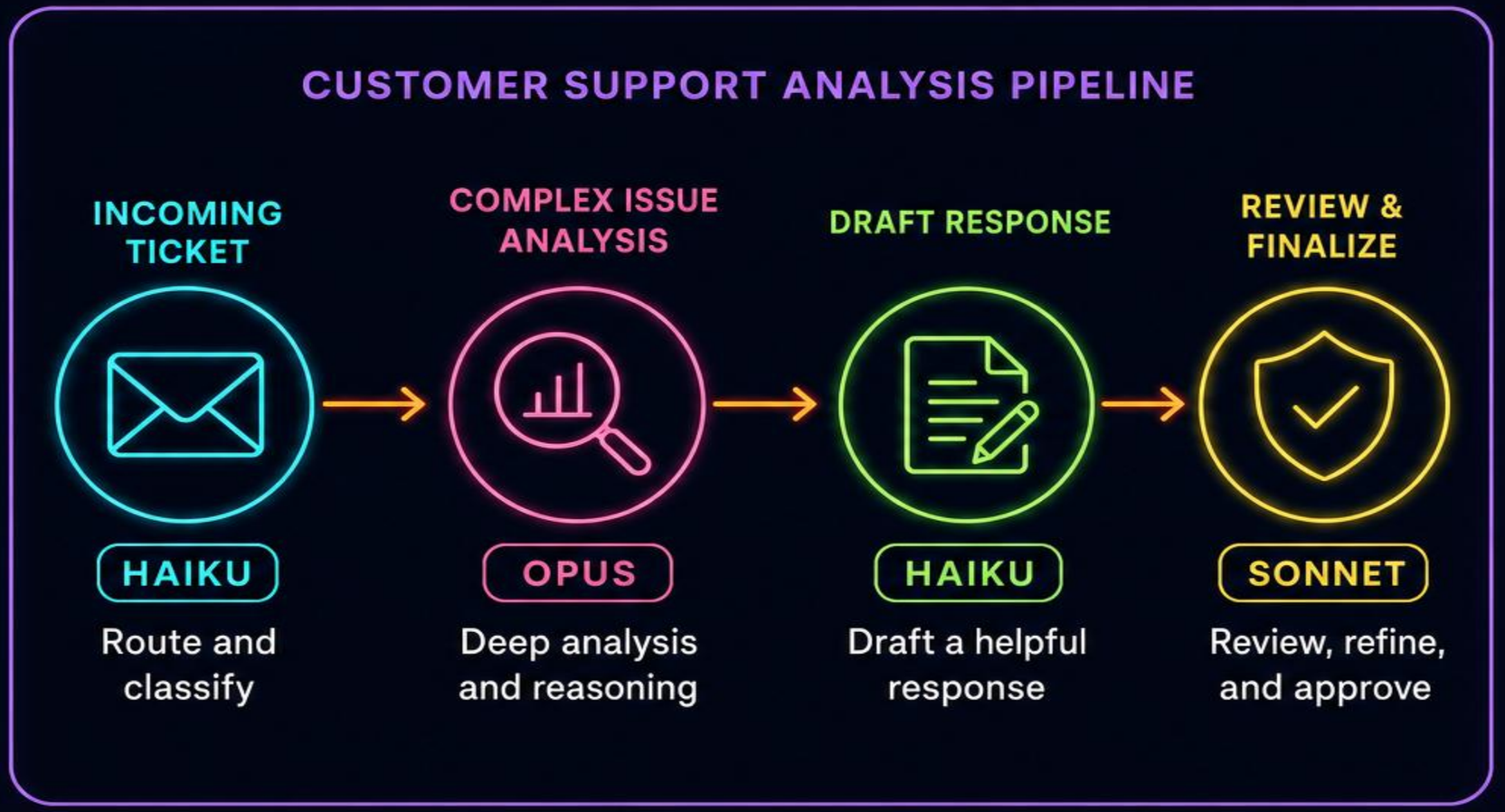
# Mixing Tiers in One Pipeline

You don’t have to pick one tier for the whole job.

**Use Haiku** to route, classify, or pre-filter inputs

**Hand the hard** sub-step to Opus or Sonnet

**Let a lighter** tier draft, a stronger tier review



# A Worked Example




## The Job

A support pipeline tags each incoming ticket, then drafts a reply for an agent to review.



## The Read

Tagging is simple and high-volume.
Drafting needs stronger quality but not the very top tier.



## The Pick

Haiku tags the tickets cheaply,
Sonnet drafts the replies.
Opus stays in reserve for escalations.

# When to Move Up a Tier

Start light, but know the signals that say upgrade.



**Opus**
Hardest reasoning

**Sonnet**
Production scale

**Haiku**
Lightest tier

**Quality misses**
the bar on real evaluation cases

**The task needs**
deeper reasoning than the tier gives

**Errors cluster**
on the hardest inputs, not easy ones

# Choosing the Right Tier



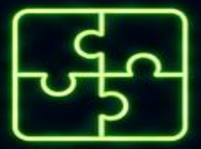
**Read** task difficulty, latency budget, and cost ceiling



**Pick** the lightest tier that clears the quality bar



**Start** light and upgrade, or start capable and optimize down



**Mix** tiers so each sub-step gets the model it needs

# Quality, Latency, and Cost

Every model choice is a balancing act between three forces that pull against each other.



**QUALITY**

Better accuracy, reliability, and depth



**LATENCY**

Faster responses and lower delay



**COST**

Lower spend and efficient usage



# Three Balancing Factors



## Quality

The capability of the model, its accuracy and depth on the reasoning, coding, or analysis the task demands.



## Latency

How long the model takes to respond, which matters most for real-time and interactive workloads.




## Cost

The per-token price you pay, multiplied across every call once the system runs at production volume.

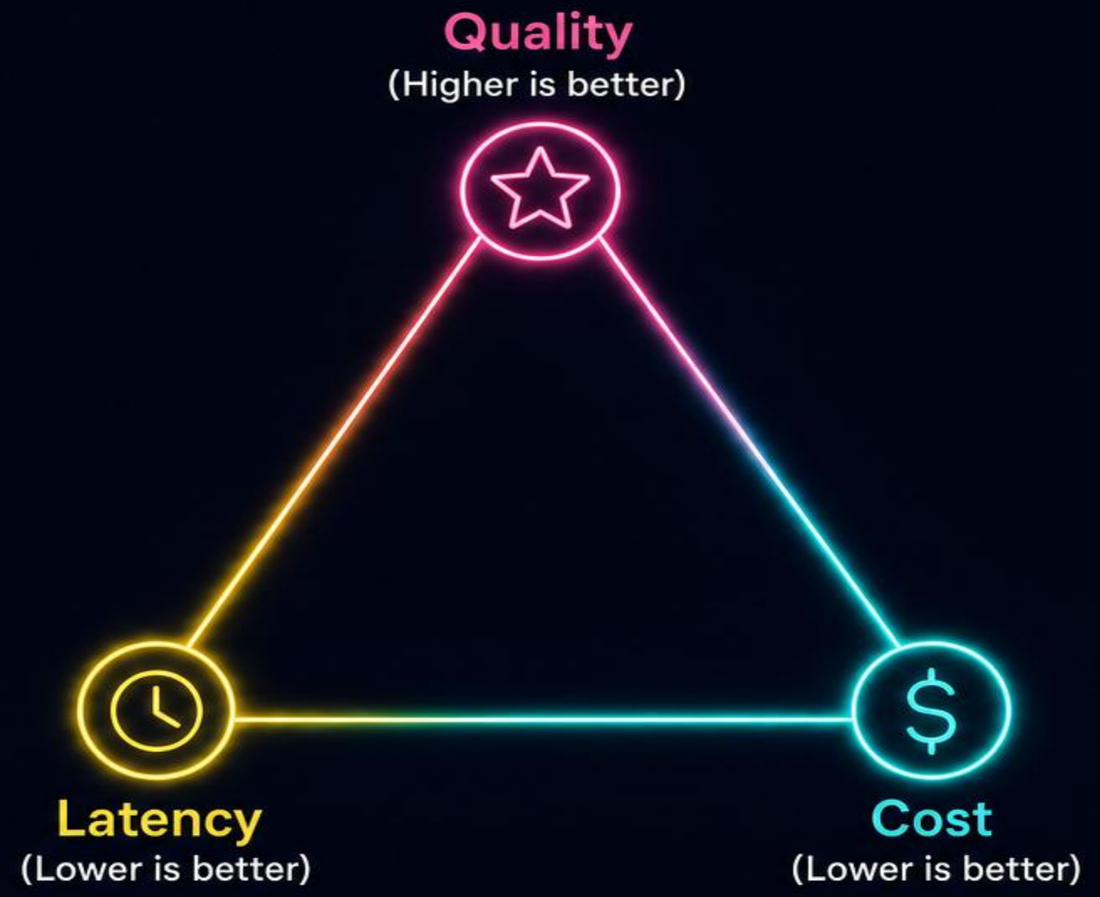
# The Tradeoff Triangle

You rarely get top quality, top speed, and lowest cost at once.

**Higher capability**
usually means
higher cost

**Higher capability**
usually means
more latency

**The lighter, cheaper**
tiers are the fastest



# Quality vs. Speed



## Higher Tiers

Opus and Fable deliver the top quality for hard problems, but they run at moderate to slower latency because they spend more effort on reasoning.



## Lighter Tiers

Haiku and Sonnet respond fastest, with Haiku leading, trading some capability ceiling for speed that suits real-time and interactive work.

# How Cost Scales



## Haiku, the Baseline

The lowest per-token price, so treat it as the cost baseline you compare against.



## Sonnet

Costs a few times Haiku's input rate, buying stronger quality for production work.



## Opus

Costs several times Haiku's input rate, the price of top-tier reasoning and coding.




## Output Costs More

Output tokens bill higher than input on every tier, so terse responses save money.

# Effort, the Same-Model Lever



## Same-Model Tuning

Effort controls how much work the model spends on a response, without changing which model you use.



## Often the Better Lever

Tuning effort is often a better first move than switching to a different, heavier model.




## Trades Within a Model

Lower effort means less latency and cost, higher effort means more reasoning depth on the same tier.

# Satisfy Every Constraint

The best pick isn't the strongest or the cheapest model.



### It's the tier that meets every stated constraint together

All constraints must be met at the same time.



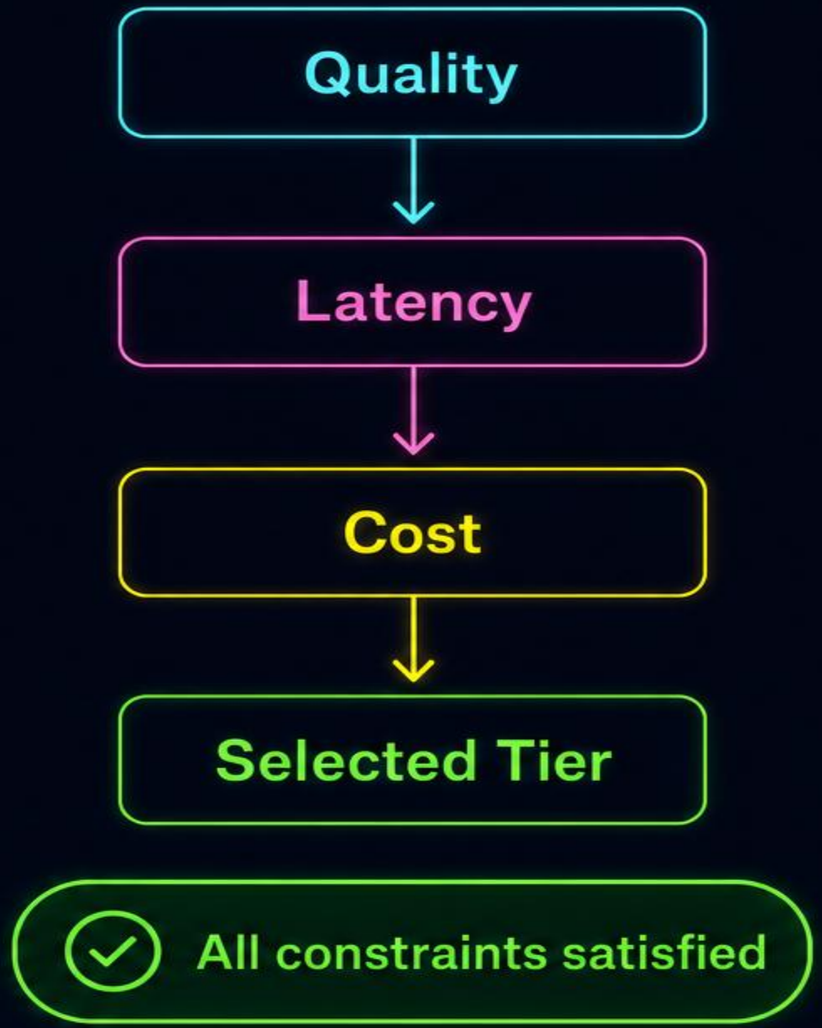
### Weigh quality, latency, and cost as a set

Optimize across all three dimensions together.



### A tier that fails any one of them is out

One failed constraint disqualifies the tier.



# Common Selection Traps



## Picking by Prestige

Always choosing Opus because it's the most capable, ignoring that most tasks don't need it.



## Picking by Price

Always choosing Haiku to save money, even when the task clearly needs more capability.



## Ignoring Latency

Optimizing only for quality or cost, then missing a real-time budget the workload actually had all along.

# Balancing the Tradeoffs



**Higher tiers** cost more and generally run slower



**Haiku is fastest and cheapest,** Opus and Fable the most capable



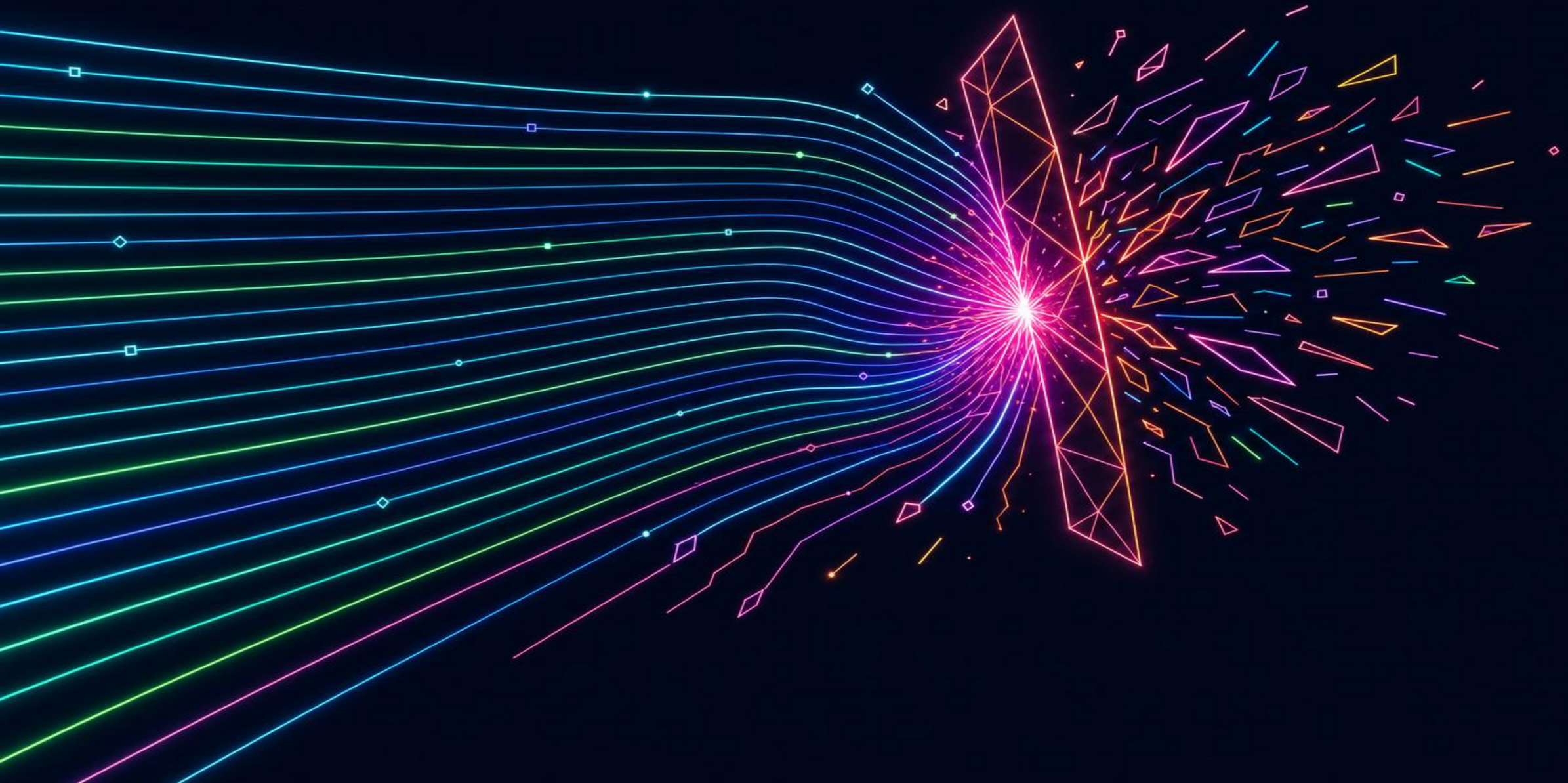
**Tune effort** before you switch to a heavier model



**The best pick** satisfies every constraint at once

# When a Newer Model Rejects Params

Migrating an app to a newer Claude model can turn a working call into an error.



# Three Terms for Breaking Changes



## Breaking Change

When a newer model rejects a parameter an older one accepted, so a request that once worked now fails.



## HTTP 400

The bad request status the API returns when a call carries a parameter the target model no longer supports.



## Model-Specific Behavior

Which parameters a call accepts depends on the exact model, so identical code can pass on one and fail on another.

# Why a Working Call Breaks

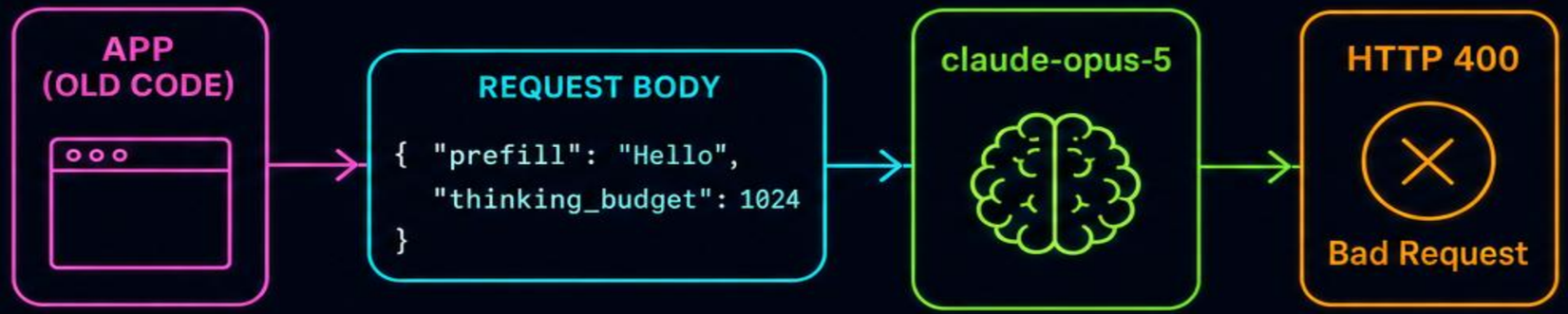
As newer models ship, some older parameters are retired in favor of cleaner mechanisms.

**Sampling controls, prefill,** and the old thinking budget were all removed

**Old code** still sends them in the request body

**The new model** answers with a 400 instead of a response




**The code never changed. The rules underneath it did.**

# Before Migration vs After



## Before Migration

On an older model, the request sets a temperature, prefills the assistant's reply to force a format, and returns a clean response every single time.




## After Migration

Ported unchanged to a current model like Opus 4.8, that exact same body returns a 400, because those parameters are no longer accepted.

# Three Classic Casualties



## Sampling Params

temperature, top-p, and top-k return a 400 on Fable 5, Opus 4.8, Opus 4.7, and Sonnet 5.



## Final-Turn Prefill

Prefilling the last assistant turn to shape output is rejected on the same four current models.

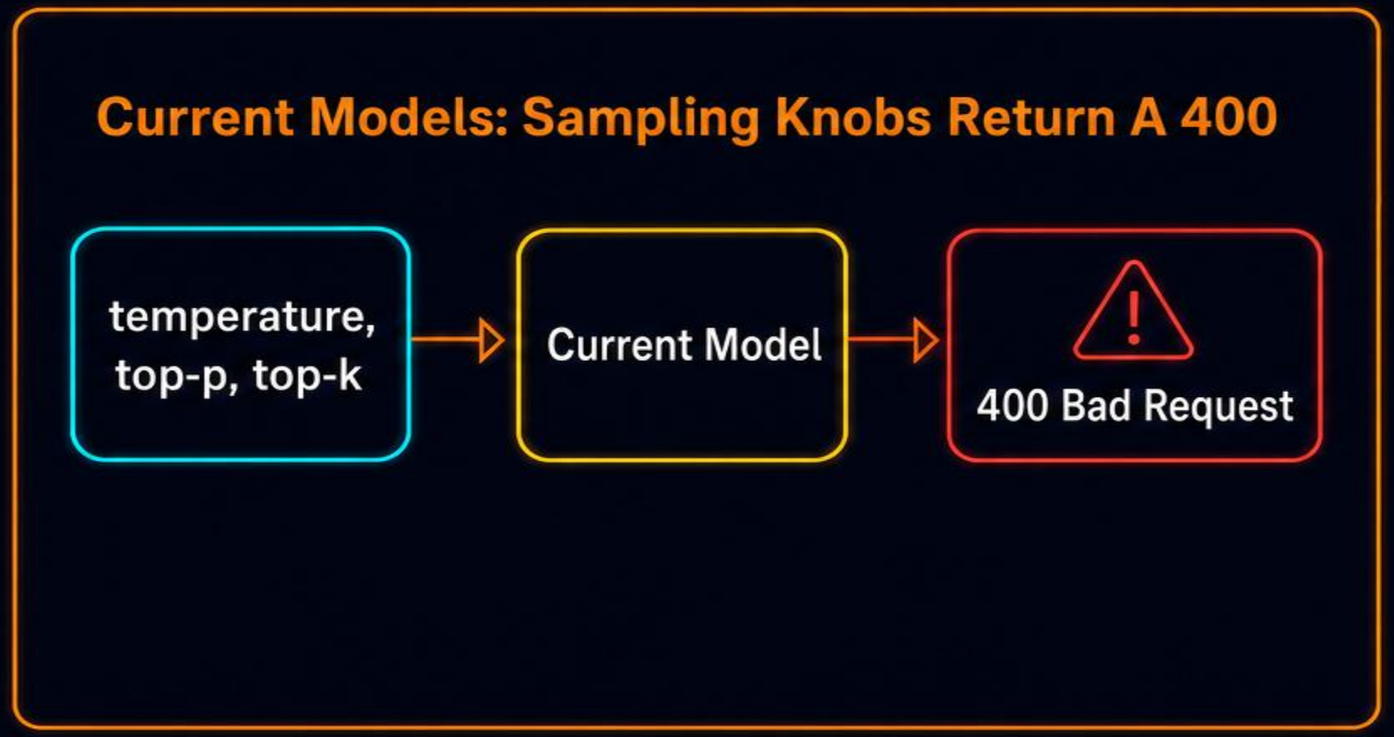


## Legacy Thinking Budget

Thinking set to enabled with a token budget returns a 400 on those same four models.

# Sampling Parameters Are Gone

On current models, the sampling knobs you may reach for out of habit no longer work.



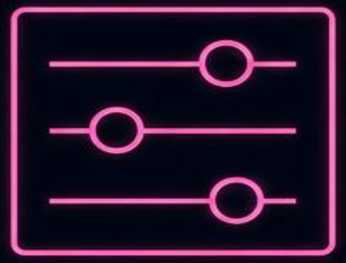

**temperature, top-p, and top-k** return a 400 on Fable 5, Opus 4.8, Opus 4.7, and Sonnet 5

**Steer behavior** with prompting and the effort level instead

**temperature zero** was never fully deterministic, even on old models

**So don’t reach for sampling to control output on current models.**

# Three Parameter Swaps



## Sampling to Effort

Replace the removed temperature and top-p controls by steering with your prompt and the effort level.



## Budget to Adaptive

Replace the legacy budget-tokens thinking mode with adaptive thinking, where the model decides how much to think.



## Prefill to Structured

Replace a final-turn prefill with a structured output format or an instruction in the system prompt.

# Thinking Is Model-Specific

Thinking behavior isn't uniform across the lineup, so it doesn't just port forward.



**Opus 4.8 and Sonnet 5** use adaptive thinking; the old budget style returns a 400

Opus 4.8 and Sonnet 5 → Adaptive Thinking → Budget Style Returns 400



**Haiku 4.5** still uses budget-tokens and has no adaptive mode

Haiku 4.5 → Budget-Tokens → No Adaptive Mode



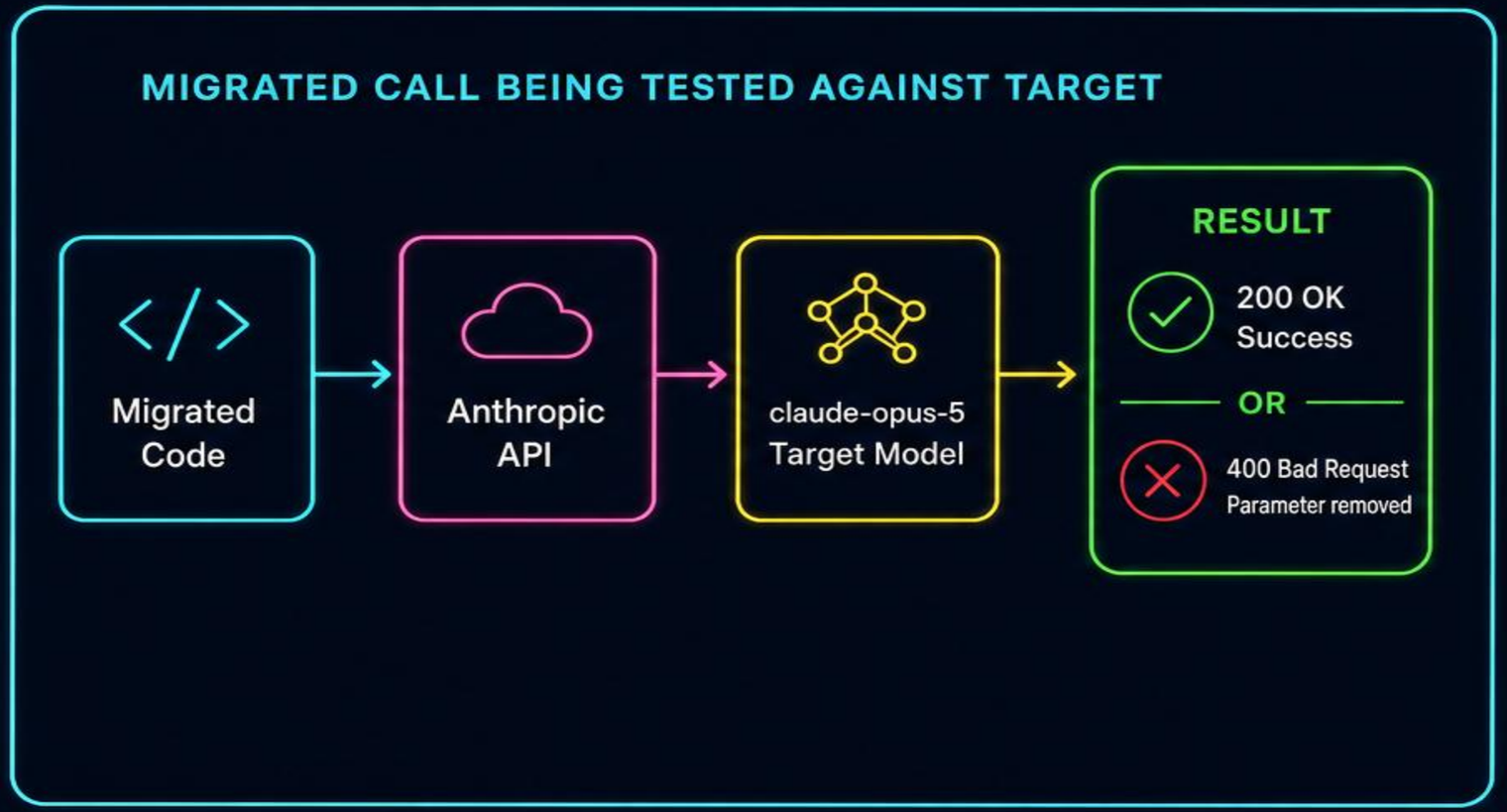
**Fable 5** keeps thinking always on; disabling it is rejected

Fable 5 → Thinking Always On → Disabling Is Rejected

Match your thinking config to the specific target model.

# Test Ported Code on the Target

The safe habit is simple. Never assume parameters carry forward untested.




**Run migrated code** against the exact model you'll use in production

**Read a post-migration 400** as a removed parameter, not your bug

**Check the migration guide** for the target before you cut over

**Catch these in testing, before your users ever see them.**

# What Breaks on Migration



**A once-valid call can return a 400 after migration**

API shape, required fields, and parameter names change.



**Sampling params, final-turn prefill, and legacy thinking budgets break first**

Old defaults or unsupported options cause silent drift or hard failures.



**Swap to prompting plus effort, adaptive thinking, and structured outputs**

Move control from tuning knobs into prompts and response schemas.



**Thinking behavior is model-specific, so test on the exact target**

Validation on this model and version prevents surprises in production.

# Migrating Safely With Version Pinning

Deprecation forces every Claude app to migrate models eventually, and a careless switch can break what works.

## Deliberate Migration Workflow

1. **Inventory And Assess**
   List apps, usage, and current model versions
2. **Choose Target Model**
   Review deprecation notices and select a target version
3. **Build Test Suite**
   Cover critical prompts and real-world scenarios
4. **Run Evaluation**
   Compare outputs, latency, and cost against baseline
5. **Migrate And Deploy**
   Update apps, monitor, and validate in production

## Version Pinning in Practice

**Client App**

```
model: claude-opus-4-20250514
```

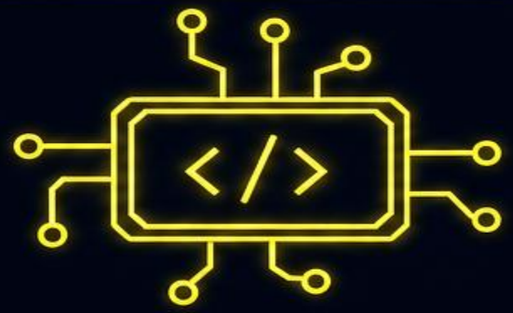
**Model Registry**

- claude-opus-4-20250514
- claude-opus-4-1-20250805 — Deprecated
- claude-opus-4-5-20251101
- claude-opus-5

**Opt-in Upgrade**

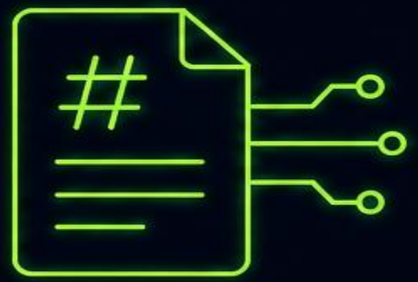
```
model: claude-opus-5
```

# Three Pinning Terms



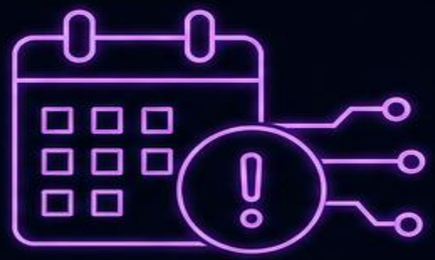
## Model ID

The exact string naming a model, like claude-opus-4-8, Opus 4.8's pinned snapshot. It selects both the weights and the behavior.



## Version Pinning

Naming an exact model ID instead of a floating alias, so a new release can't silently change your app's behavior.



## Deprecation

Anthropic marking a model for retirement on a set date, after which calls fail and you're forced to migrate.

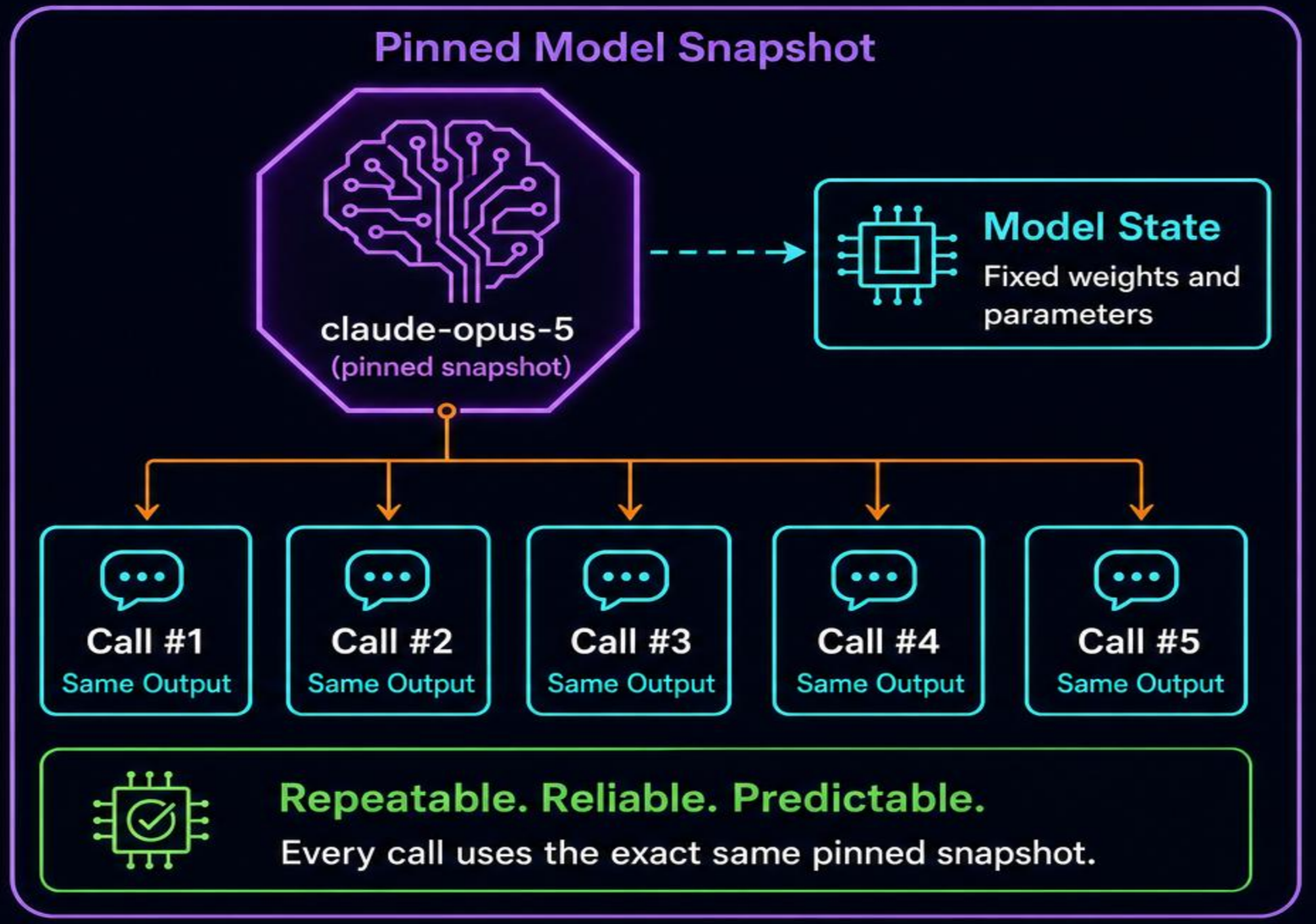
# Pinned Snapshots Don't Drift

On the 4.6 generation and later, a model ID is a fixed snapshot.

**Anthropic never updates** the weights under an existing ID

**A new version ships** under a new ID, not on top of the old one

**A pinned ID gives you** the same model on every call



# Pinned ID vs Floating Alias



## Pinned Snapshot ID

A fixed identifier naming one frozen model. Dateless from the 4.6 generation on, like claude-opus-4-8, dated on pre-4.6 models. Behavior stays stable until you deliberately migrate. Best for production.



## Floating Alias

A pre-4.6 moving string that resolves to whatever the latest snapshot is. Convenient, but a new release can shift behavior with no code change.

# Dateless vs. Dated IDs




## Dateless format

The 4.6 generation and later use claude, name, then major version, like claude-opus-4-8. This dateless ID is itself the pinned snapshot, not a moving latest pointer.

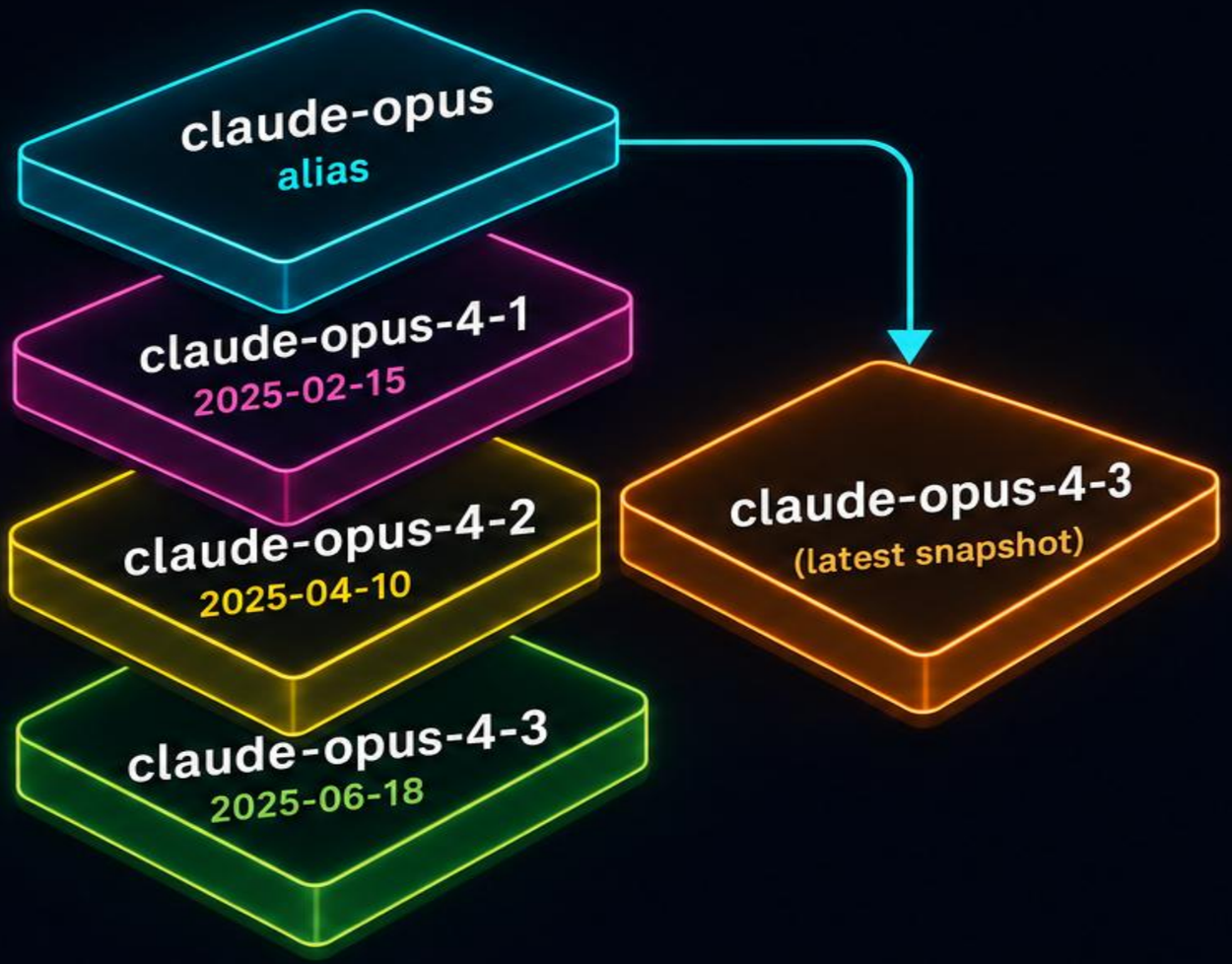



## Dated format

Older models end in a year-month-day stamp, like claude-sonnet-4-5 then the date. The trailing date marks the exact snapshot you are calling.

# Aliases Are a Pre-4.6 Convenience

Before the 4.6 generation, the API also offered shorter aliases for convenience.




**An alias points at the newest dated snapshot for that version**

It always resolves to the most recent snapshot available.

**It trades reproducibility for always getting the freshest fix**

Great for staying current, but harder to reproduce past runs.

**Aliases are a pre-4.6 convenience, not offered for newer IDs**

Starting with 4.6 and beyond, only dated IDs are available.

For a dateless ID, the ID itself is already the pin.

# Four Steps to Migrate Safely



## Pin the Current Model

Lock the app to the exact model ID you run today, so nothing shifts while you prepare.



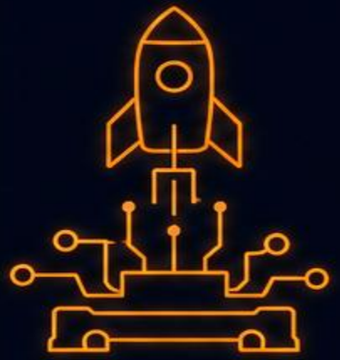
## Freeze an Eval Set

Gather a fixed set of representative prompts and expected outputs, and keep it unchanged.




## Test the New Model

Run the frozen eval against the candidate model and compare results side by side.

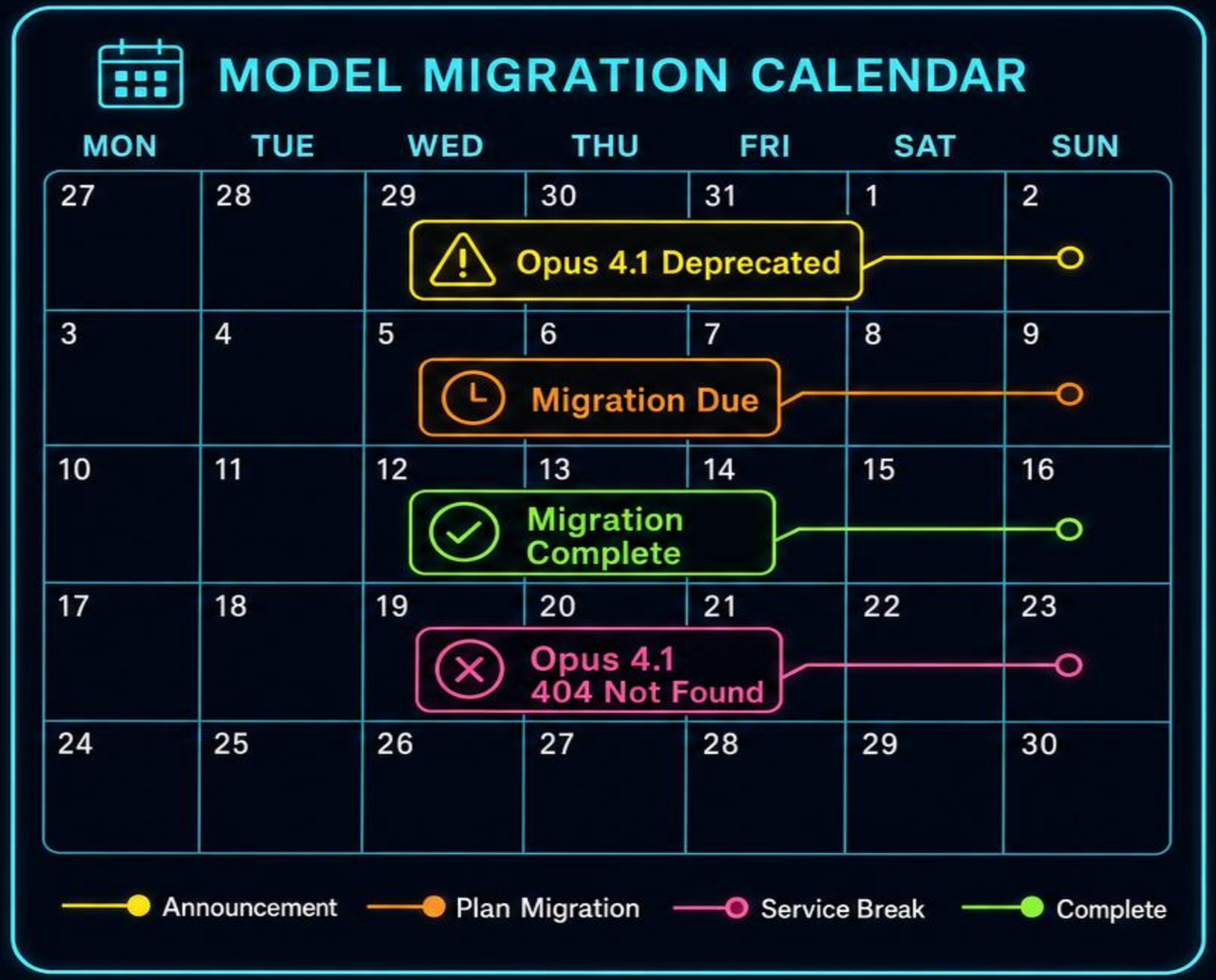


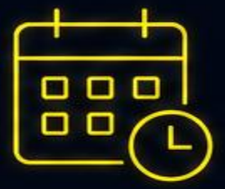
## Cut Over Deliberately

Once results hold, switch the pinned ID in one reviewed change and watch production.

# Migration Is Recurring Maintenance

Migrating models is not a one-time development task. It's ongoing maintenance driven by deprecation.



**A deprecated model gets** a retirement date and later returns a 404



**An app that never** migrated simply stops working



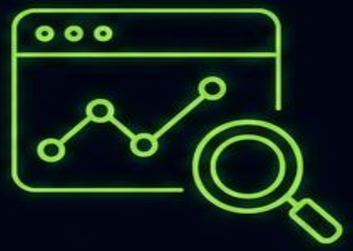

**Opus 4.1 is deprecated;** Sonnet 3.7 and Haiku 3.5 already retired

# Three Testing Terms



## Frozen Eval Set

A fixed set of prompts and expected outputs, held constant so two models are judged on identical inputs.



## Regression Test

Rerunning that eval against a new model to catch behavior that changed for the worse before users see it.




## Prompt Versioning

Tracking prompts as versioned artifacts tied to a model version, so a change is reviewable and reversible.

# Re-Baseline After a Switch

Some things change even when the call still succeeds, so a green light isn't the whole story.

**The newer tokenizer** counts roughly 30 percent more tokens on Opus 4.7 and later, plus Fable 5 and Sonnet 5

**Re-check your max output**, cost estimates, and context budget

**Thinking display** now defaults to omitted, so reasoning may look empty

## Example: Model Switch Impact

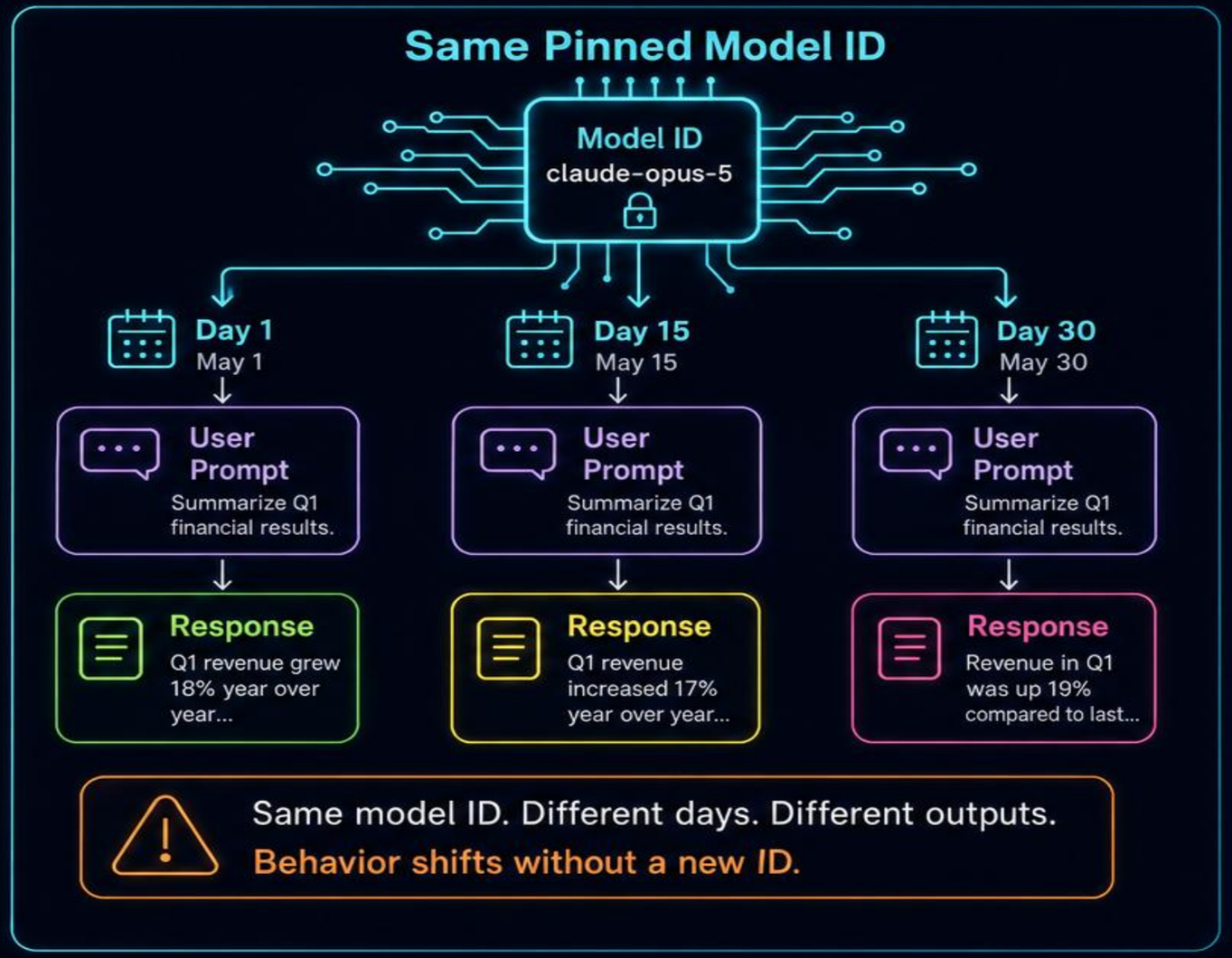
| | Before (Opus 4.5) | | After (Opus 4.7) |
|---|---|---|---|
| Input Tokens | 1,000 | +30% | 1,300 |
| Output Tokens | 2,000 | +30% | 2,600 |
| Estimated Cost (USD) | $0.120 | +30% | $0.156 |

Same call. Higher tokens. Higher cost.
Re-baseline your budgets and alerts.

Update alerts

# Behavior Shifts Without a New ID

Pinning fixes the weights, but not everything that shapes a response lives in the weights.




**Serving infrastructure like routing and safety classifiers can change**

Upstream systems, policies, and classifiers evolve over time.

**That can shift observable behavior** on the same pinned ID

Responses can drift even when the model version stays fixed.

**So rerun your eval set** on a schedule, not just at migration

Scheduled evals catch drift early before it impacts users.

# Migrate on Your Terms




**Pin an exact model ID** so releases don't change behavior silently



**Treat migration as recurring maintenance** driven by deprecation



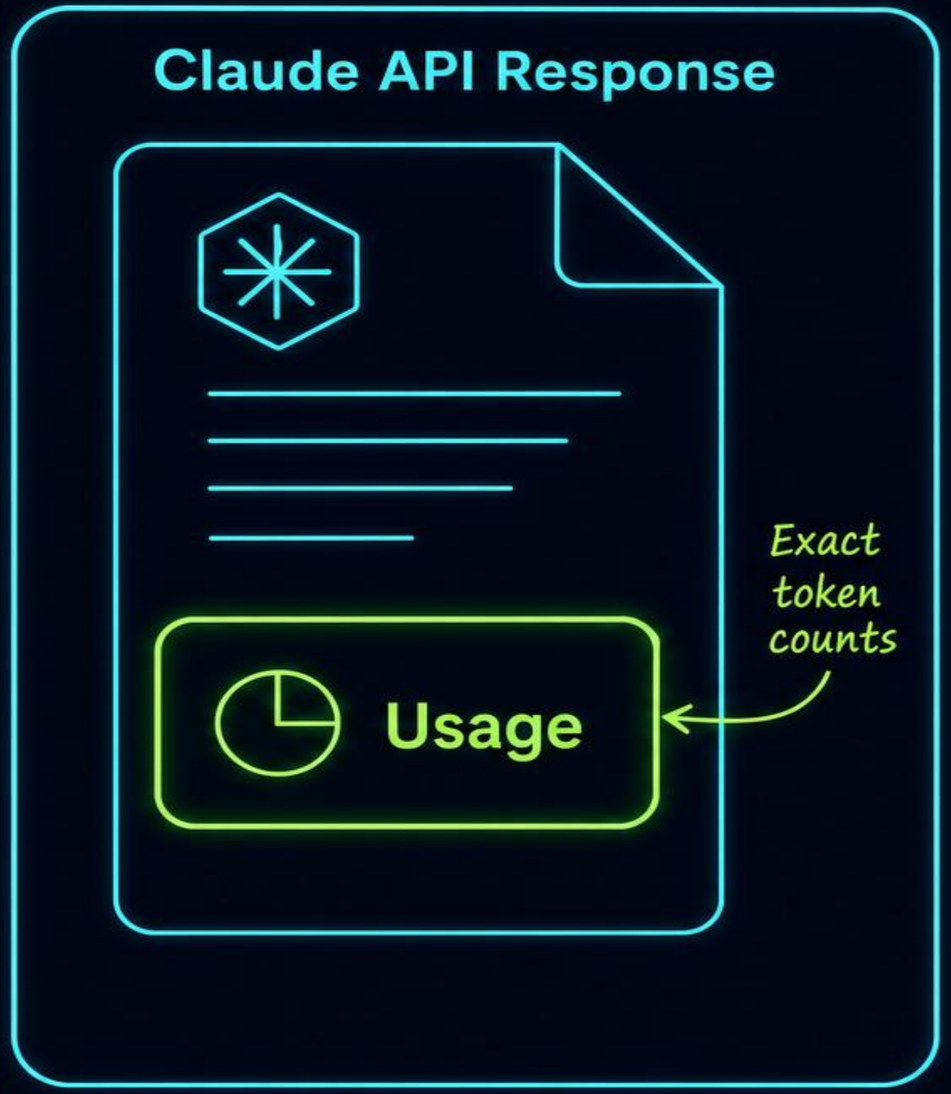
**Test a new model against a frozen eval set** before cutting over

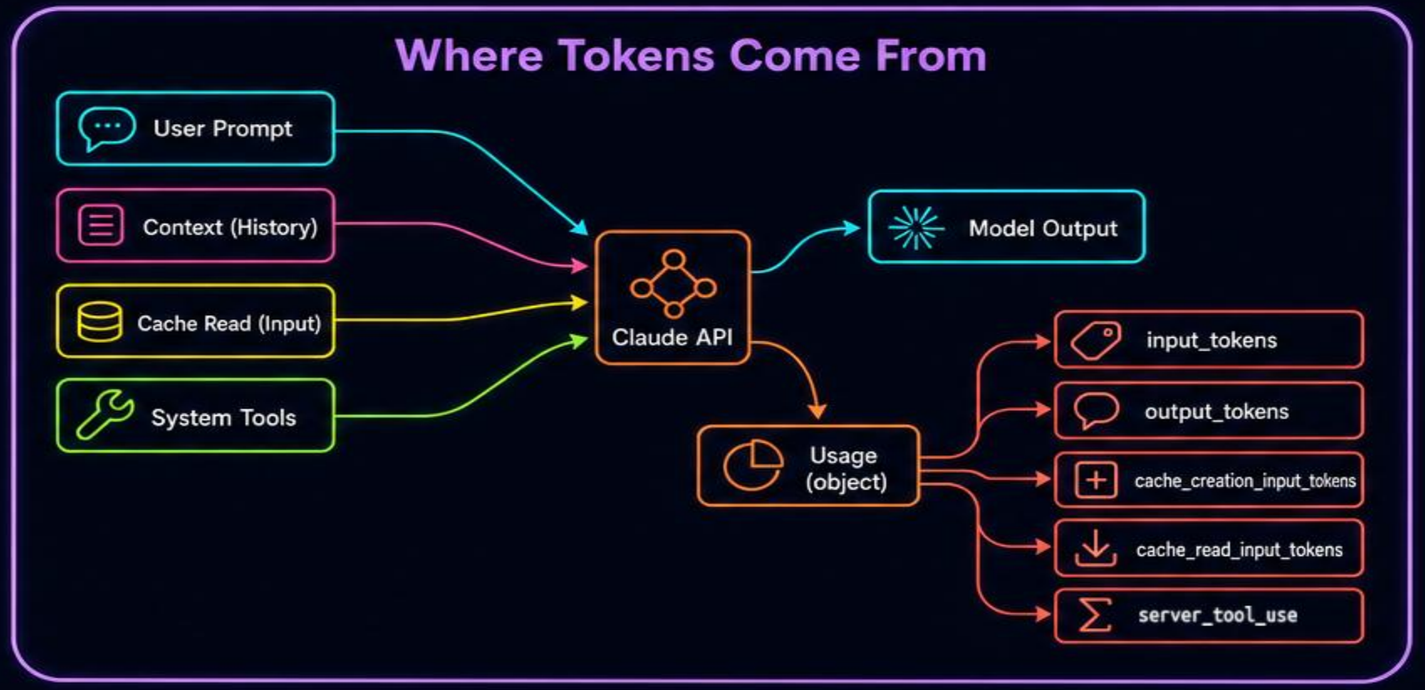


**Behavior can shift even on a pinned ID,** so keep monitoring

# Reading the Usage Fields

Every response from the Claude API carries a usage object with exact token counts.



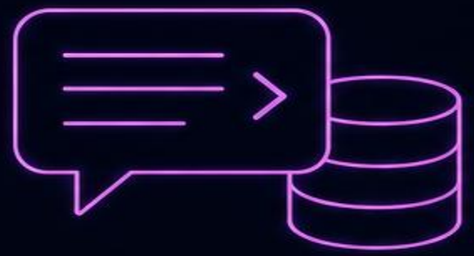



# Three Terms to Start With




## Usage Object

The block on every response reporting token counts for that one call. It is the basis for cost tracking.



## input_tokens

The uncached input tokens processed at full price. Not the whole prompt when caching is on, only the part read fresh.



## output_tokens

The tokens the model generated in its reply. These usually carry a higher per-token price than input tokens do.

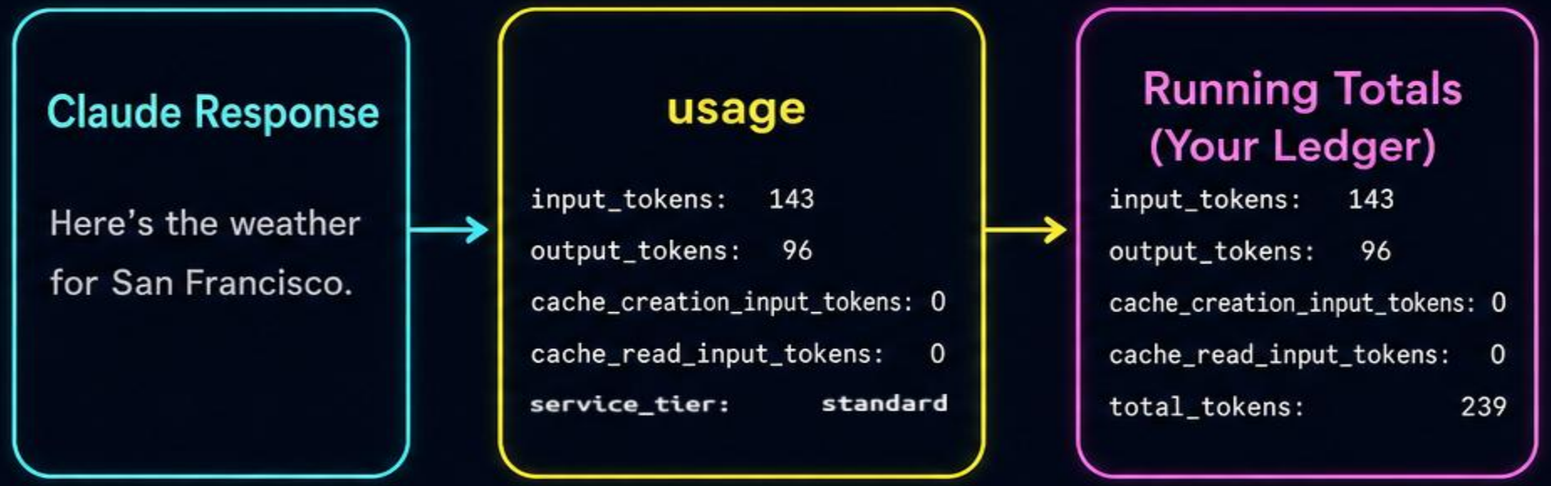
# Where Usage Lives

The usage object comes back on the same response as your content, no extra call needed.

**It reports counts** for that one request, not a running total

**Your code reads** it and adds the numbers to your own ledger

**Summing across calls** is on you, the API doesn't aggregate




**Track it per call, and your spend stays accurate.**

# Input Tokens vs. Output Tokens



## Input Tokens

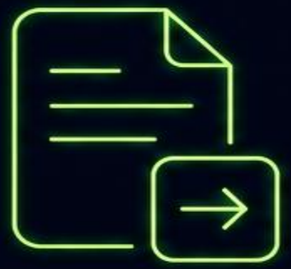
Everything the model reads, before caching splits it: your system prompt, conversation history, tool definitions, and the current message. Priced at the base input rate.



## Output Tokens

Everything the model generates in response, including the visible answer and thinking tokens. Priced higher than input on every current model, so they dominate many bills.

# The Four Usage Fields



## input_tokens

Uncached prompt tokens read at full input price. Excludes anything served from the cache.



## output_tokens

Tokens the model generated in its reply, billed at the higher output rate.



## cache_creation_input_tokens

Prompt tokens written into the cache on this call, billed at a small write premium.

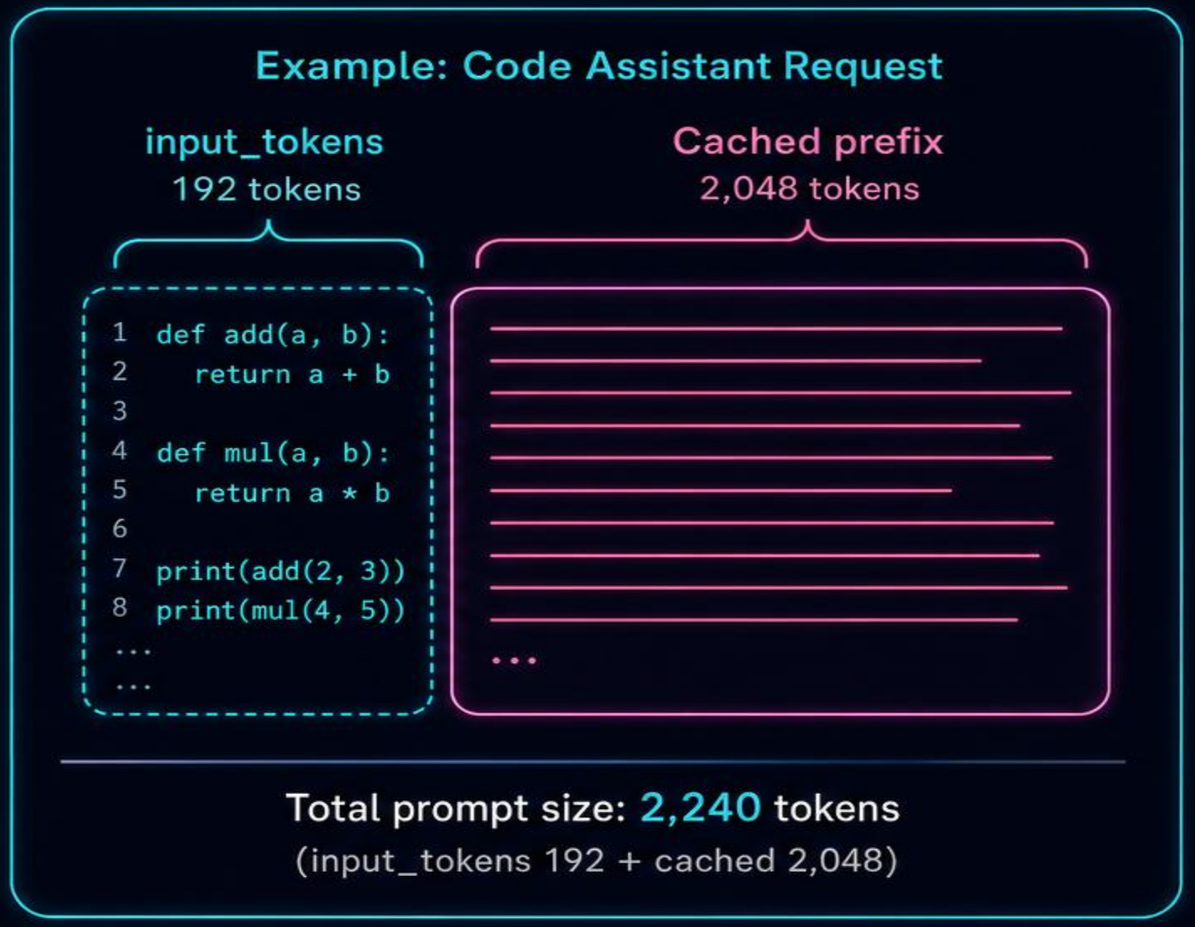


## cache_read_input_tokens

Tokens served from cache instead of reprocessed, billed at a deep discount to base input.

# Input Tokens Is Only the Remainder

When caching is on, input_tokens stops being the whole prompt. It counts only the fresh part.




### The cached tokens live in the two cache fields instead

They are stored in the system and reused across turns.

### Reading input_tokens alone undercounts your real prompt size

It reflects only the new tokens in this request.

### A big cached prefix can make input_tokens look tiny

The true context in play is much larger than it appears.

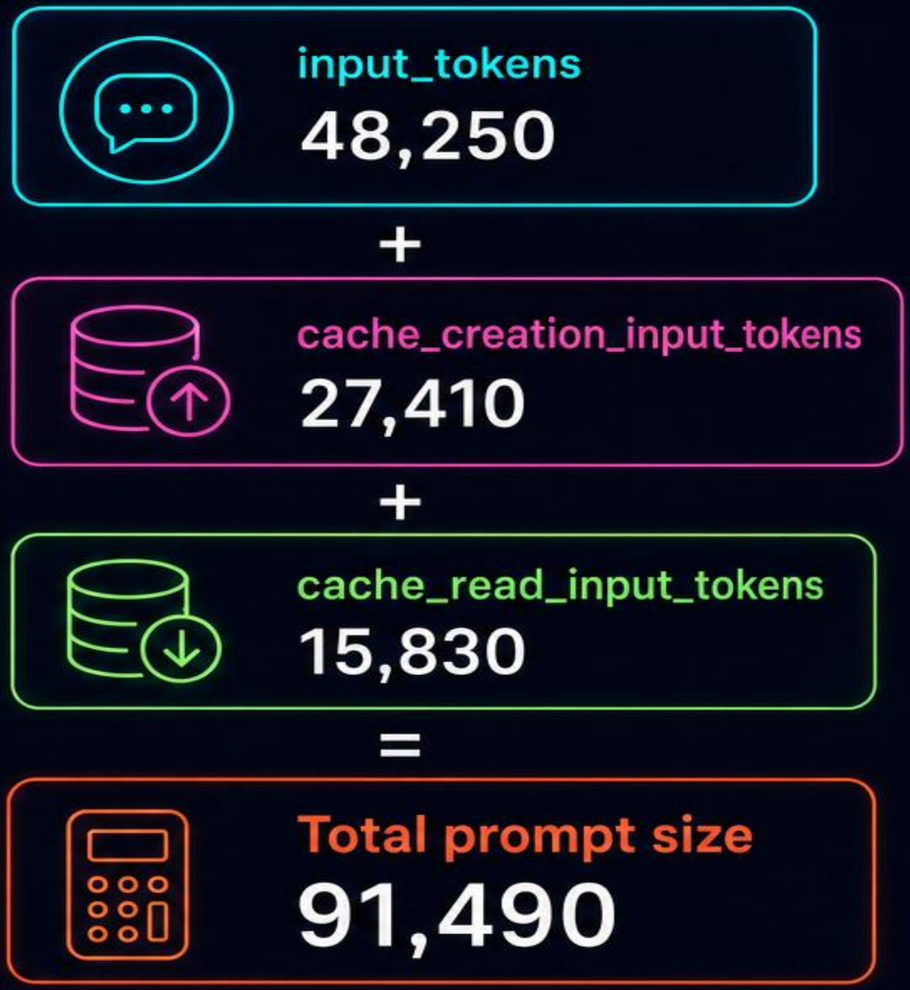
# Rebuilding Total Prompt Size

To get the real prompt size, you add three fields together, not just read one.

**Start with input_tokens,**
the uncached remainder

**Add cache_creation_input_tokens,**
the tokens written to cache

**Add cache_read_input_tokens,**
the tokens served from cache



# Counting Tokens Before You Call



## count_tokens

An endpoint that returns the input token count for a request before you send it. No generation, no output charge.



## Same Inputs

You pass the model, messages, system prompt, and tools you plan to use, so the estimate reflects the real call.

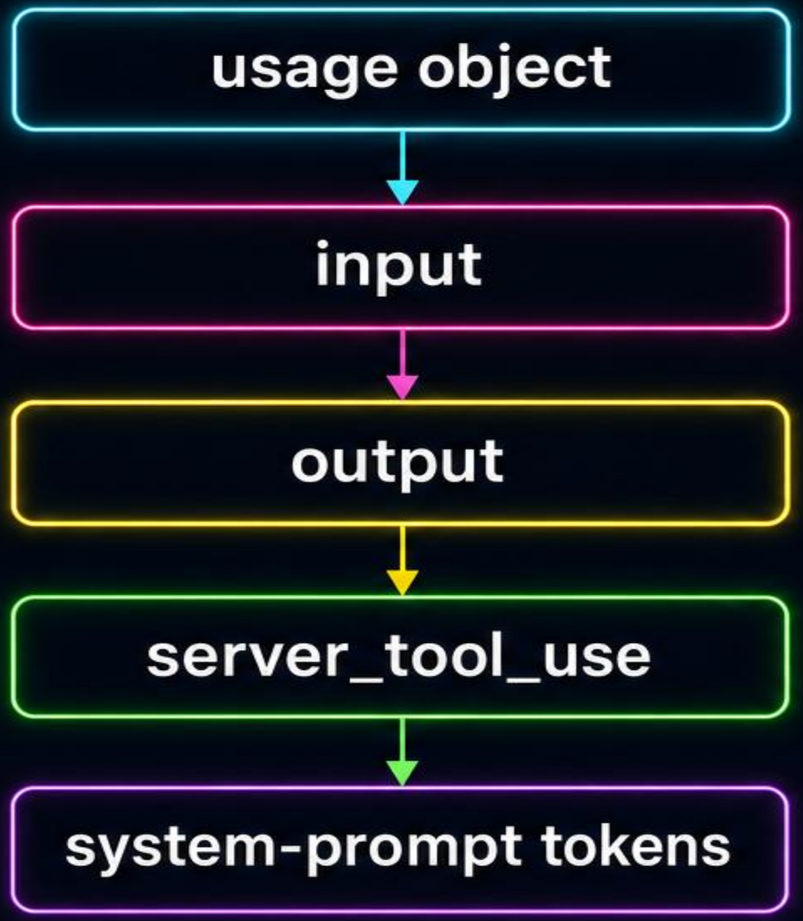


## Model-Specific

Counts are per model, so pass the same model you will infer with. A different model can tokenize the text differently.

# Usage Beyond Plain Tokens

The usage object can carry more than input and output when your call uses extra features.




**Server tools report** under a server_tool_use section

**Tool definitions** add system-prompt tokens to your input

**Those overhead tokens** vary by model and tool choice

**Fold every category in, or your estimate runs light.**

# Reading Usage Correctly



**Read input_tokens, output_tokens, and the two cache fields**

Understand what each field represents before you interpret usage.



**input_tokens is the uncached remainder, not the full prompt**

It excludes cached content that was reused.



**Add all three input fields for true prompt size**

Sum input_tokens + cache_creation_input_tokens
+ cache_read_input_tokens.



**Count tokens up front with count_tokens, per model**

Estimate before sending to avoid surprises on usage.

# Modeling and Controlling API Cost

Once your app works, the next question is what it costs to run at scale.

## Cost Model from First Principles

**Token Volume**
inputs + outputs per request

×

**Rates**
per 1M tokens (input / output)

×

**Requests**
per minute / hour / day / month

↓

**Cost per Request**
(input cost + output cost)

↓

**Cost over Time**
minute, hour, day, month

↓

**Projected Spend**
with scenarios and buffers

## Example Projection

Assumptions and projected monthly spend

| Avg Input | Avg Output | Requests | Rates |
|---|---|---|---|
| 1,200 tokens | 800 tokens | 10,000 / day | $3.00 / $15.00 per 1M tokens |

| Scenario | Tokens / Day (Input / Output) | Est. Cost / Day | Est. Cost / Month (30 days) |
|---|---|---|---|
| Base | 12.0M / 8.0M<br>20.0M total | $1.32 | $39.60 |
| Growth +50% | 18.0M / 12.0M<br>30.0M total | $1.98 | $59.40 |
| Peak 2x | 24.0M / 16.0M<br>40.0M total | $2.64 | $79.20 |
| Optimized -30% | 8.4M / 5.6M<br>14.0M total | $0.92 | $27.60 |

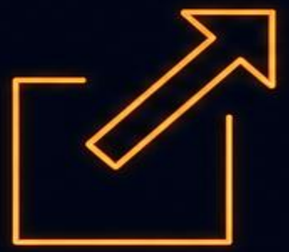
Add a buffer for variance and spikes (e.g., +20%).
Continuously track actual spend vs. projection.

# Three Pricing Facts



## Per-Token Pricing

Spend is tokens times a rate. Every token in and out has a price, so cost scales with usage.



## Output Costs More

On every current model, output tokens are priced several times higher than input tokens for the same count.



## Cache Changes Price

Cached reads bill at a fraction of base input, while cache writes bill at a premium. Same tokens, different rate.

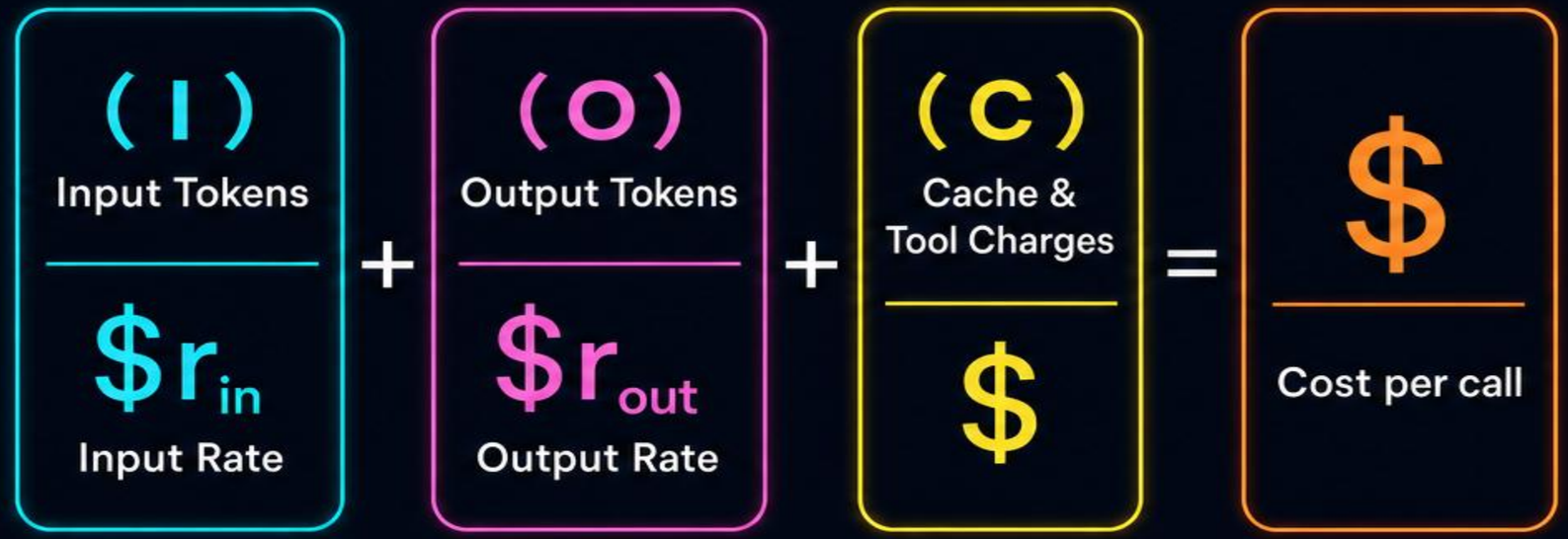
# The Basic Cost Formula

At its core, cost for one call is a short sum you can write on a napkin.

**Multiply** input tokens by the input rate

**Add** output tokens times the output rate

**Layer in** cache and any server-tool charges




Add those up and you have the cost of a single request.

# Input Rate vs. Output Rate



## Input Rate

The price for tokens the model reads.
It's the lower of the two rates, so a big
prompt is cheaper per token than a big reply.

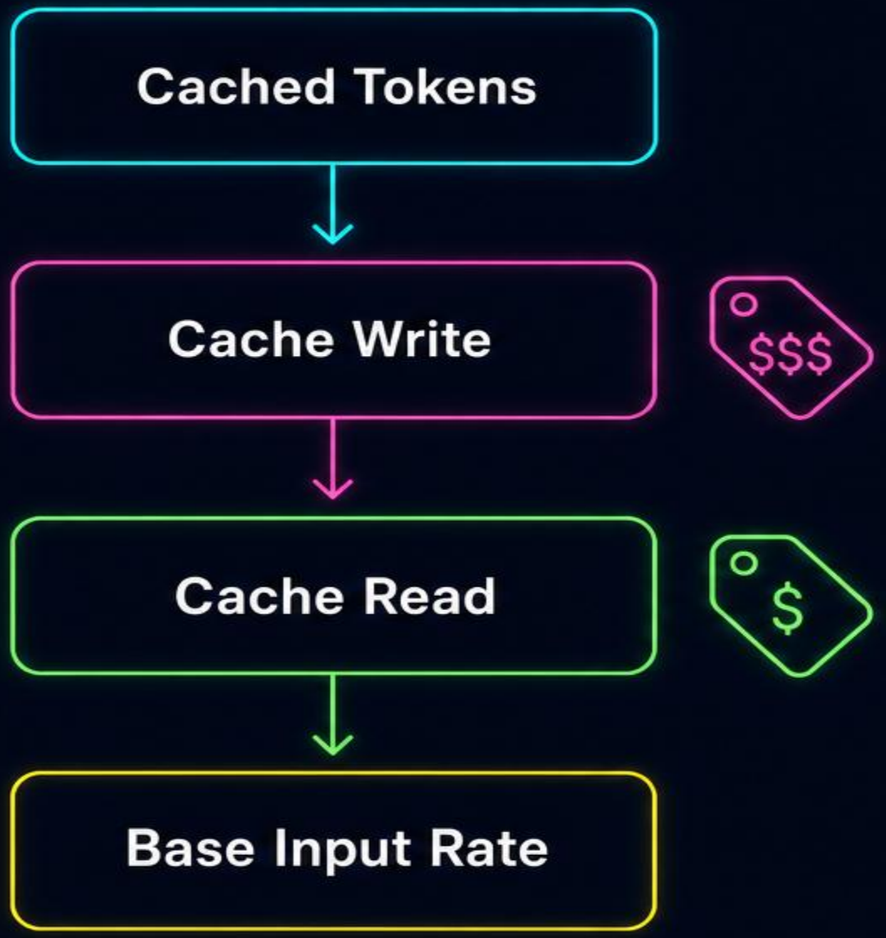


## Output Rate

The price for tokens the model writes.
It runs several times the input rate,
so a long reply is where cost piles up.

# Cache Tokens Change the Math

Caching doesn't just save tokens, it reprices them, so your model has to account for it.




### A cache read bills at a fraction of the base input rate

Reusing cached tokens is heavily discounted compared to fresh input.

### A cache write bills at a premium above base input

Storing new tokens adds cost beyond the standard input rate.

### Counting cached tokens at full price overstates cost

Treating cached tokens like fresh input leads to inflated cost estimates.

# Building a Cost Model

A workload cost model scales one call up to real traffic in three steps.

## Estimate tokens per request, input and output

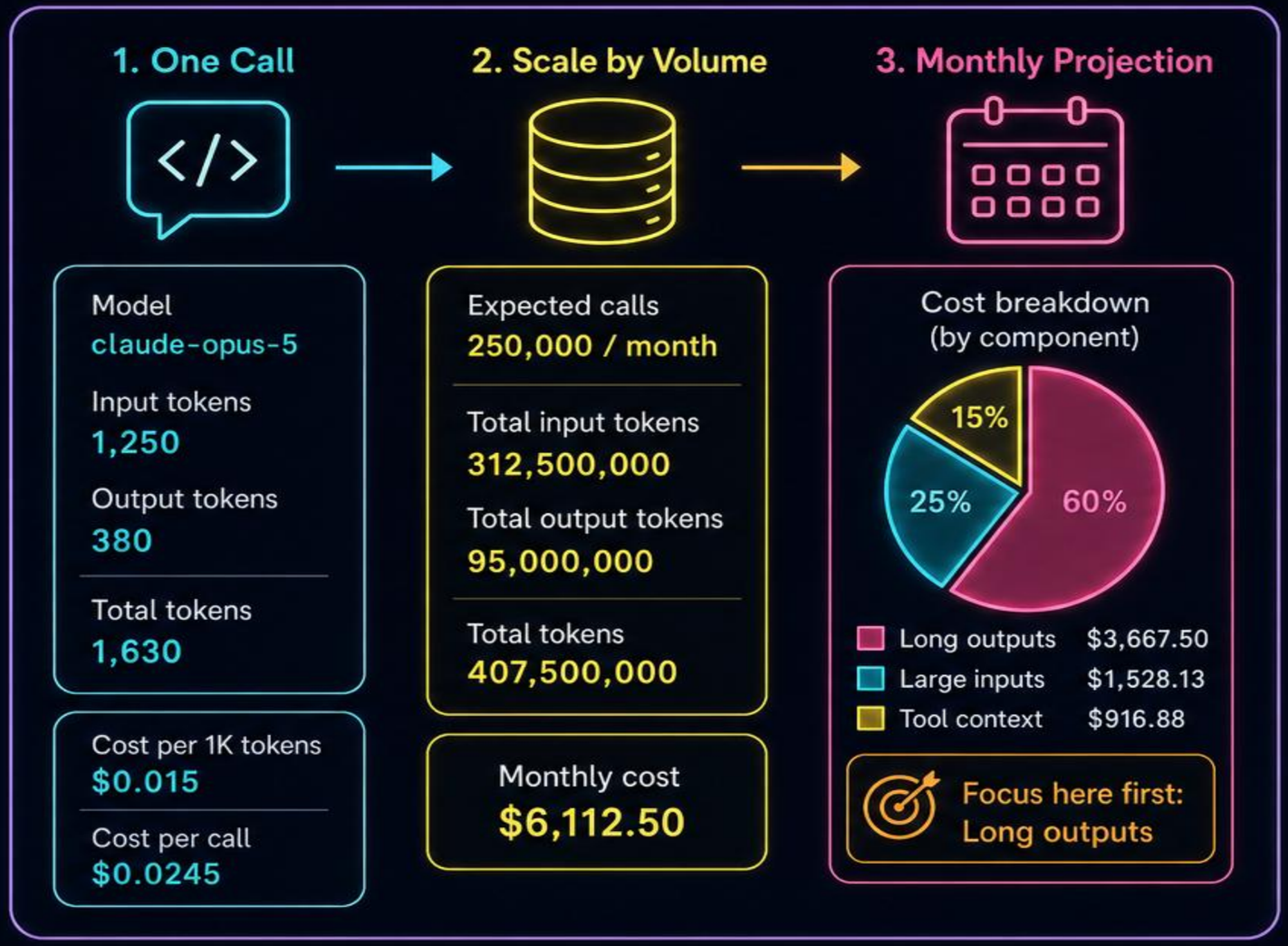
Measure or estimate the input and output tokens for a typical request.

## Multiply by your expected call volume

Project total tokens and cost based on the number of calls you expect.

## Find the dominant driver, the biggest slice, and focus there

Identify the largest cost component and optimize where it matters most.



# Four Levers to Cut Spend



## Tier Selection

Run the lightest model tier that still meets the quality bar.
Over-provisioning simple work wastes money.



## Smaller Prompts

Trim system prompts and history to the tokens the task needs.
Fewer input tokens means lower cost.



## Prompt Caching

Reuse a stable prefix so repeated tokens bill at
the cheap read rate instead of full input.



## Batch Processing

Send non-urgent work through the batch path for
a large discount when latency can wait.

# Cost by Tier, Relatively



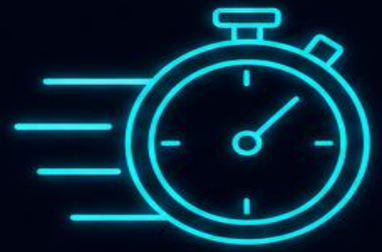
## Opus Tier

The most capable and the most expensive per token. Reach for it when the task genuinely needs top reasoning.



## Sonnet Tier

The balanced middle. Strong quality at a lower per-token price than Opus, sized for high-volume production work.



## Haiku Tier

The cheapest and fastest per token. Best for simple, high-throughput tasks like routing, extraction, and classification.

# Estimate Before You Ship

Don’t wait for the invoice to learn what a feature costs. Model it up front.



## Use count tokens to size the input before any spend

Estimate average input tokens per request using real samples and token counters.



## Combine that with a realistic output estimate

Estimate output tokens based on expected detail and format requirements.



## Multiply by volume for a monthly projection

Use your expected monthly request volume to project total token usage and cost.

A number on paper beats a surprise on the bill.

# Controlling What You Spend



**Cost is** input times rate plus output times rate, plus cache



**Output is** priced higher, so watch reply length



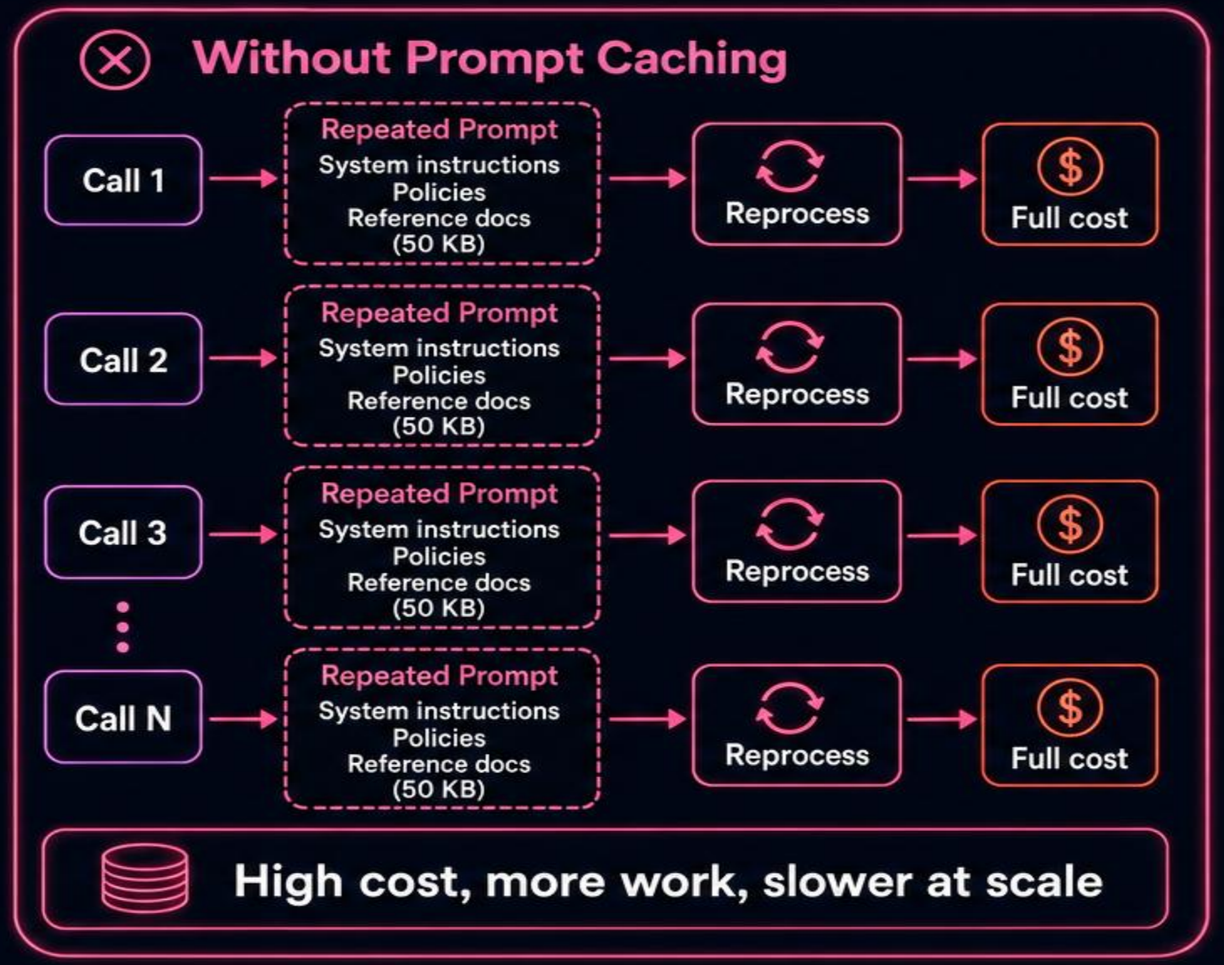
**Scale tokens** per request by volume to find the driver

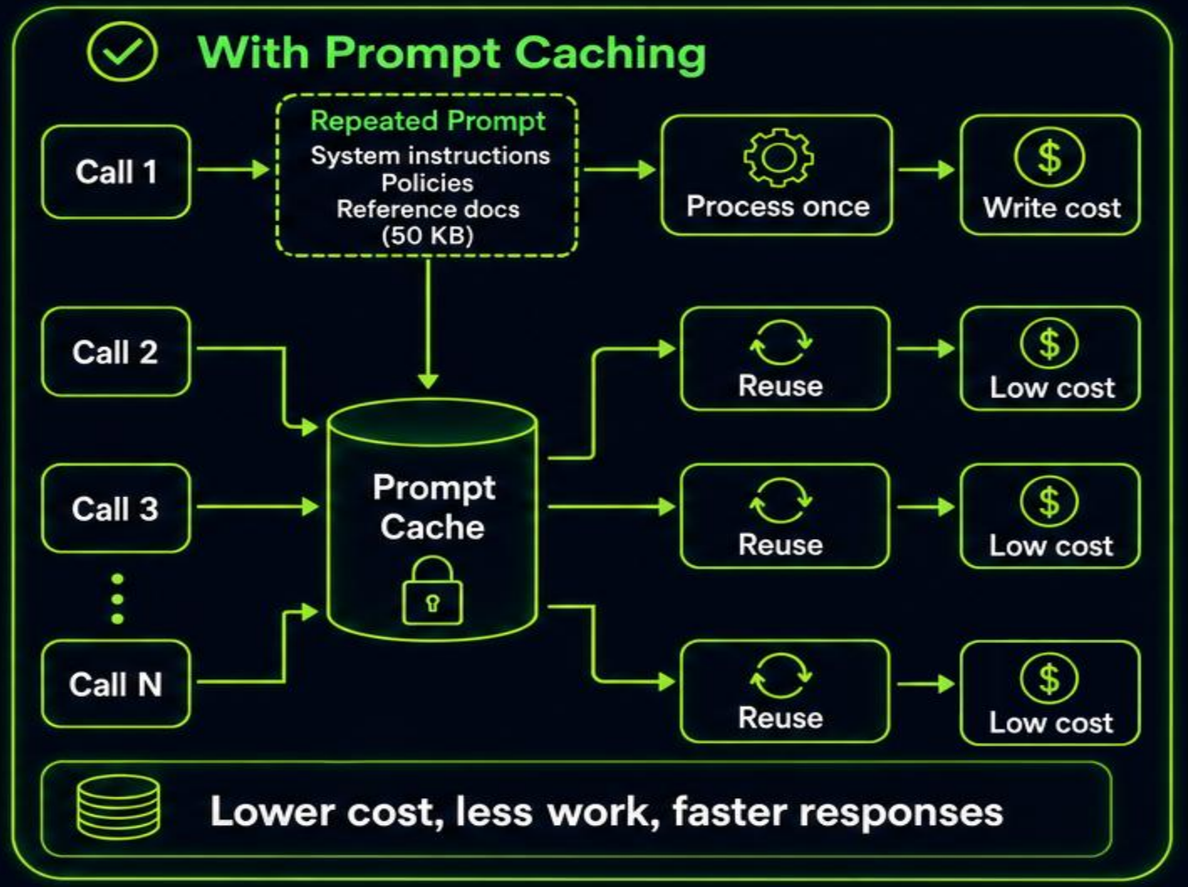


**Pull tier,** prompt size, caching, and batch to cut spend

# Prompt Caching to Cut Repeat Cost

Large, stable context that repeats on every call gets reprocessed and billed each time. Prompt caching reuses that work instead.



# Three Caching Concepts




## Prompt caching

Reusing an already-processed prompt prefix marked with cache_control, so a large, stable context is not reprocessed and rebilled on every call.



## Cache write

The one-time cost of storing a new prefix. It bills at a premium over base input, paid once when the entry is created.



## Cache read

The cost of serving a later call from that stored prefix. It bills at a deep discount, which is where the savings live.

# What You Actually Cache

Caching only helps when the same content rides along on call after call.

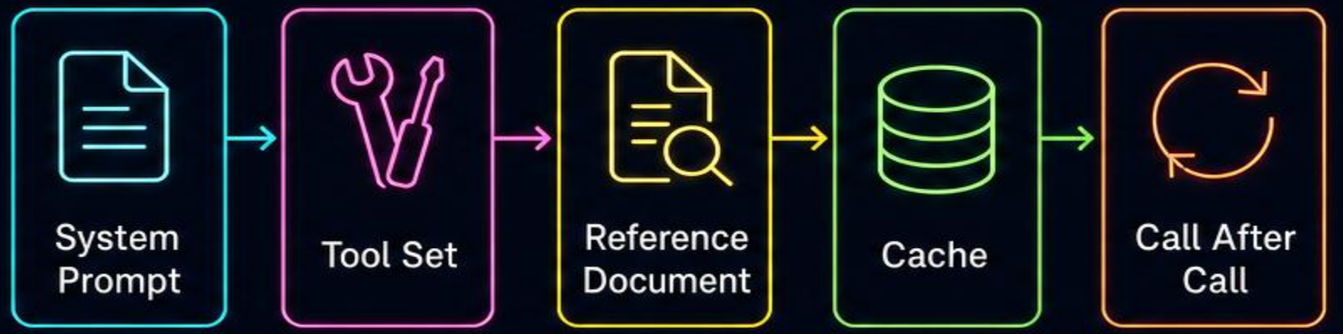
**A large system prompt**
that frames every
request the same way

**A fixed tool set**
that doesn't change
between calls

**A long reference document**
you ask questions
against repeatedly

## Stable Shared Context Reuse

Cached once, reused across many calls.



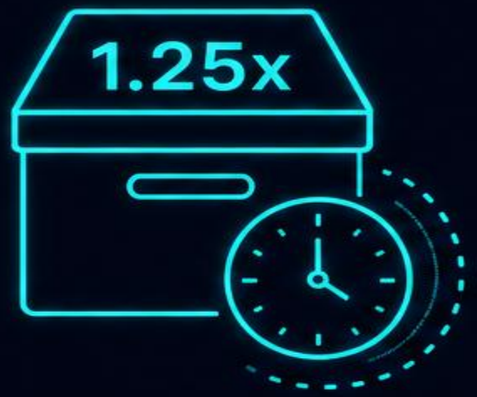

The bigger and more stable that block, the more each read saves.

# Cache Cost Multipliers



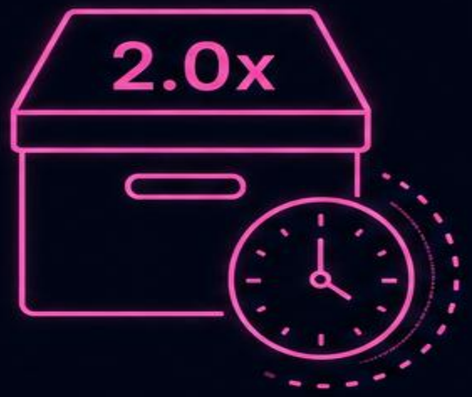

## Cache read

A hit bills at 0.1x base input price. Reuse is the entire source of savings.




## 5-minute write

Writing a default 5-minute entry costs 1.25x base input, a small one-time premium.




## 1-hour write

Writing a 1-hour entry costs 2.0x base input, the price of the longer window.

# 5-Minute vs. 1-Hour TTL




## 5-minute TTL

The default ephemeral window. Every read refreshes it, so steady back-to-back traffic keeps a hot prefix alive at the cheaper 1.25x write.




## 1-hour TTL

Set with type ephemeral and ttl one hour.
It survives longer gaps between calls, but the write costs 2.0x, so it fits spaced-out traffic.

# When Caching Pays Off

Caching is a bet that cheap reads will outweigh the write premium.




Hit a big, frozen prefix often enough and the math clearly wins.

**A 5-minute entry** pays for itself after roughly one reuse

**A 1-hour entry** needs two or more reads to come out ahead

**The stable prefix** should dominate the request size

# When Not to Cache

Caching can cost more than it saves, so know the losing cases.



## A prefix you hit only once still pays the write premium

Every unique prefix incurs the cost to store, even if it's never read again.



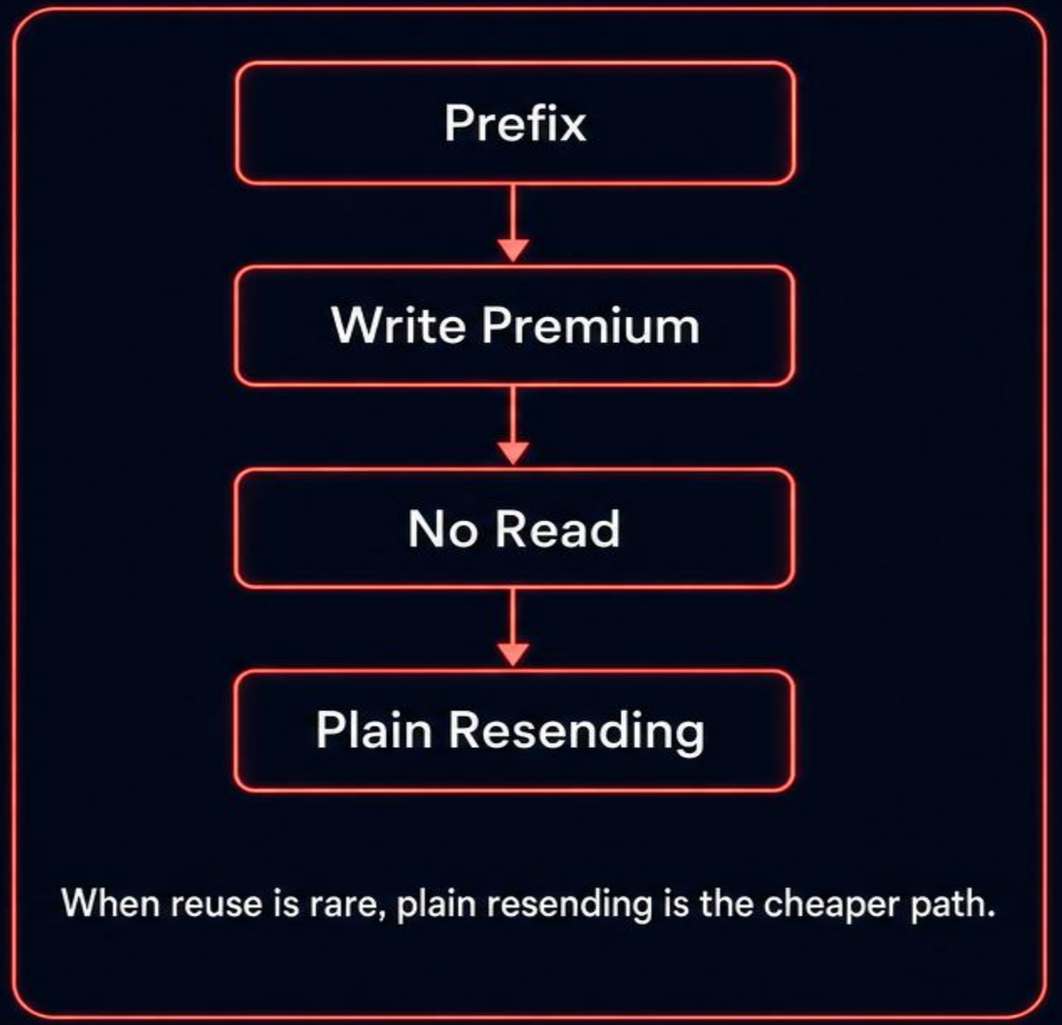
## Content that changes every call never produces a read

Dynamic, per-request content means nothing is reusable.



## A prefix under the model's token floor silently won't cache

If the prefix is below the minimum token threshold, it's not cached at all.




When reuse is rare, plain resending is the cheaper path.

# Break-Even in Practice



## First read wins it back

A 5-minute write at 1.25x plus one 0.1x read totals 1.35x, already under paying full price twice.




## The 1-hour threshold

A 1-hour write at 2.0x needs two or more reads before its cheap reads cover the heavier write.

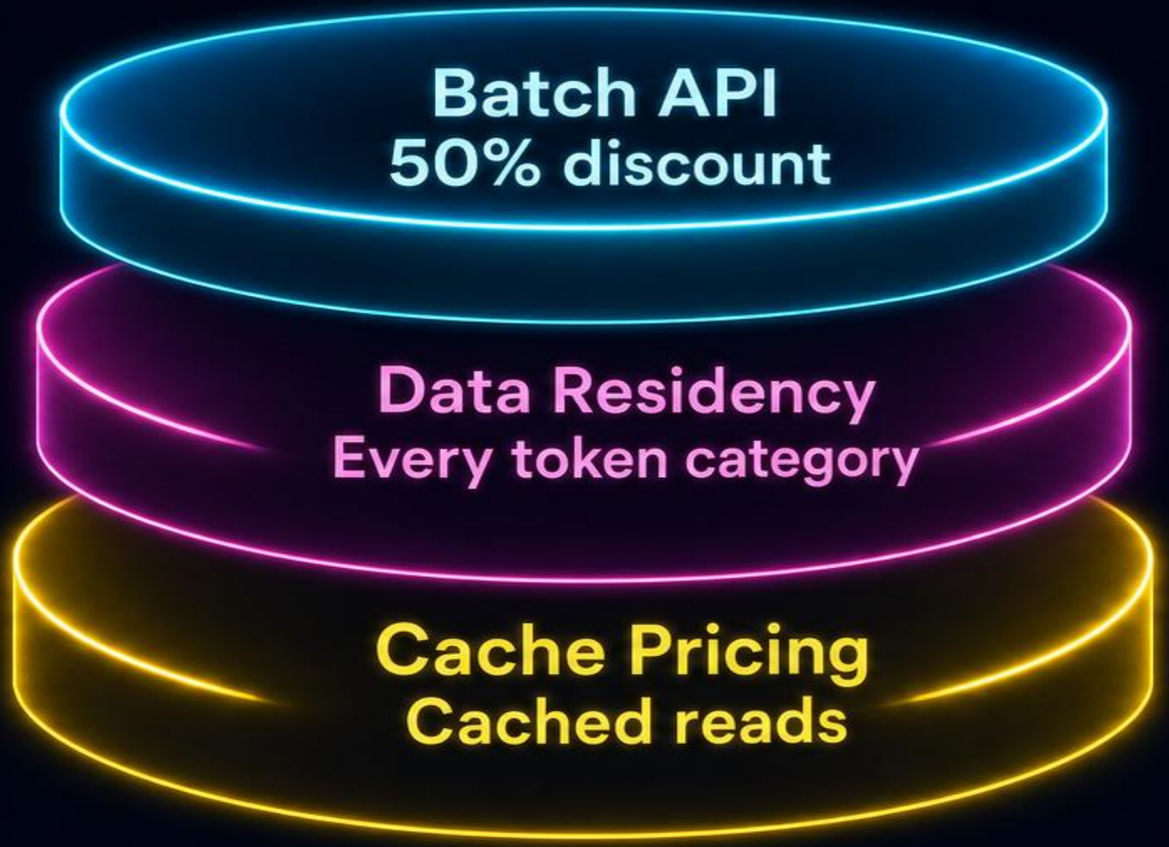


## Reads refresh the clock

Each cache read resets the entry's lifetime, so continuous traffic can keep a 5-minute prefix warm indefinitely.

# Stacking Other Discounts

Cache multipliers combine with the other cost levers you already use.




**The Batch API 50% discount** stacks on top of cache pricing

**A data-residency multiplier** applies across every token category

**Cached reads still show up** split out in the usage fields

**Layer the levers, then read usage to confirm the real savings.**

# Making Reuse Pay



**Reads bill** at 0.1x, 5-minute writes 1.25x, 1-hour writes 2.0x



**A 5-minute** entry pays off after about one reuse



**A 1-hour** entry needs two or more reads to win



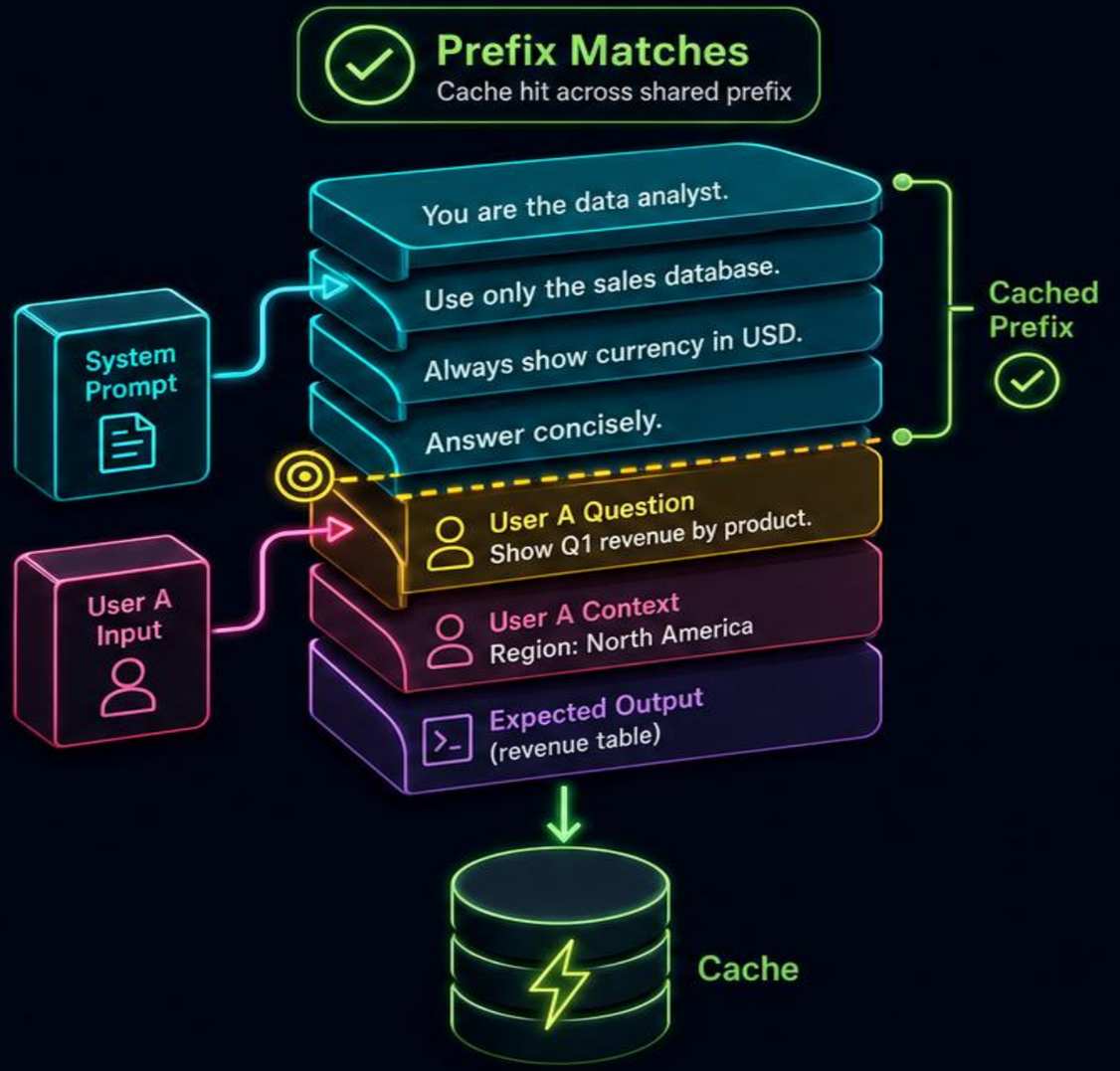
**Cache** only big, stable, frequently-hit prefixes

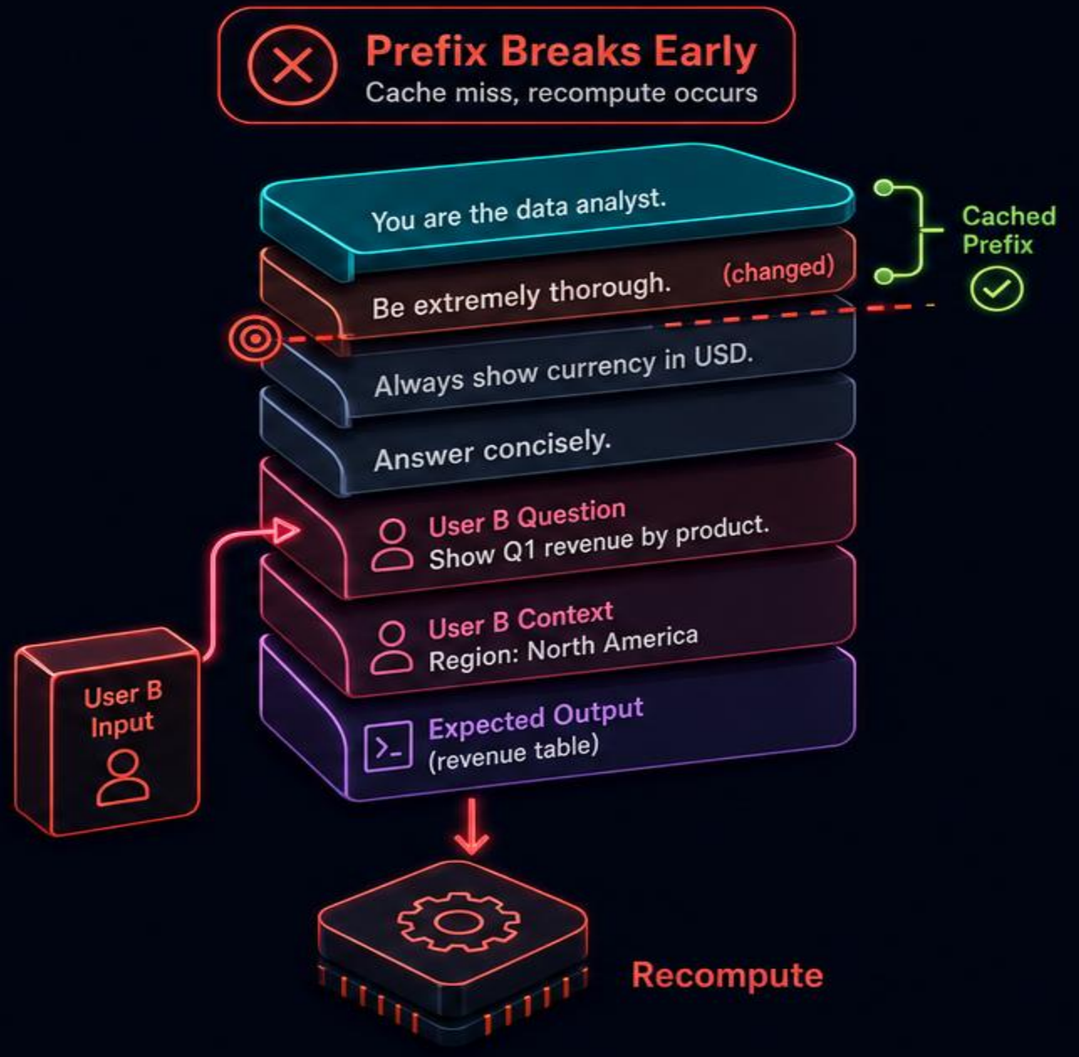


**Batch and residency** multipliers stack on cache pricing

# Cache Breakpoints and the Prefix Rule

Caching saves money only when your prefix actually matches, byte for byte.



# Three Prefix Concepts



## cache_control breakpoint

An explicit marker on a block that ends a cacheable prefix at that point. You get up to four per request.



## Rendered prefix

Everything from the start of the request up to a breakpoint, assembled in a fixed tools, system, messages order.




## Cache key

The exact rendered bytes of that prefix. Match is byte-for-byte, never fuzzy, and always prefix-only.

# The Render Order

Before matching, the request renders in one fixed order.

**tools render first,**
at the very front of the prefix

**the system prompt**
renders next, after the tools

**the messages render last,**
in conversation order

## 1 Tools
Render First

```
{
  "tools": [
    { "name": "get_weather", "input_schema": { "type": "object" } },
    { "name": "search_flights", "input_schema": { "type": "object" } }
  ]
}
```

## 2 System Prompt
Renders Next

ROLE
system

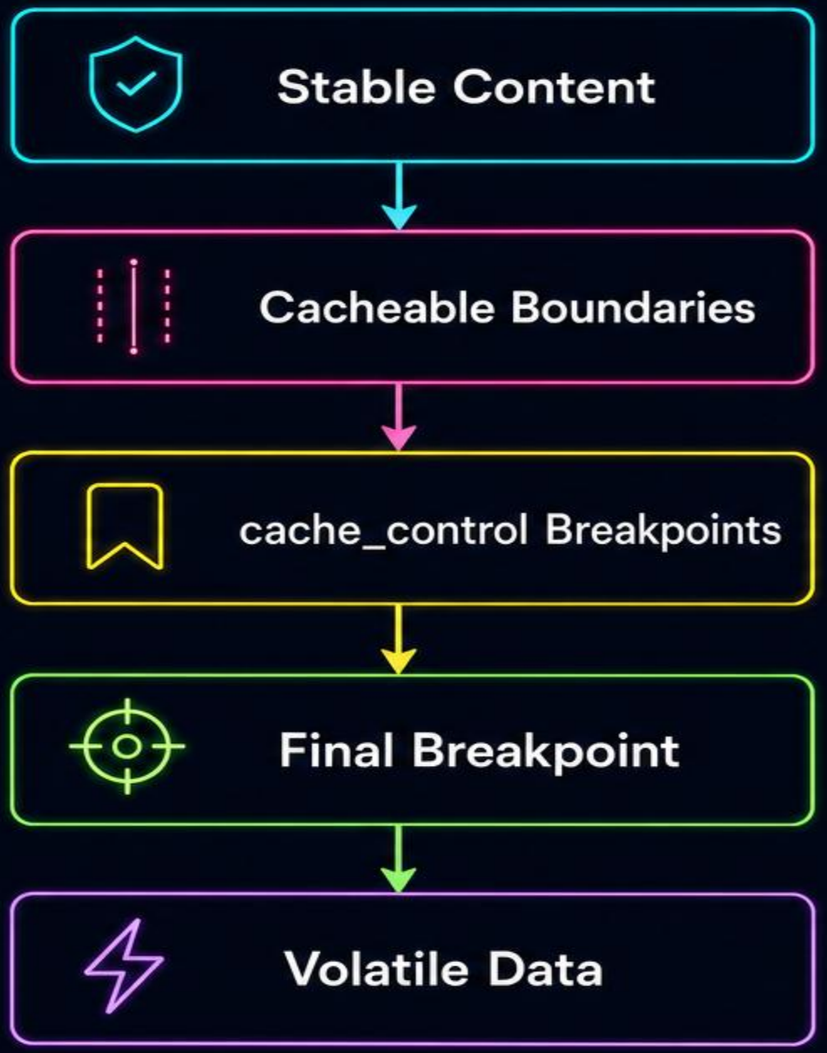
CONTENT
You are a helpful travel assistant.
Use the provided tools when needed.

## 3 Messages
Render Last

| | | |
|---|---|---|
| USER | Find flights from SFO to NYC tomorrow. | 10:15 |
| ASSISTANT | Checking options for you now. | 10:15 |
| TOOL | get_flights result: 8 options found. | 10:16 |

# Placing Breakpoints

You mark cacheable boundaries with cache_control breakpoints on stable blocks.







### You get up to four explicit breakpoints per request

Use up to four cache_control breakpoints in each request.



### Place each one at a boundary where content stops changing

Align breakpoints with stable content boundaries.



### Push volatile per-request data after your final breakpoint

Keep volatile data after the last breakpoint to protect cache hit rate.

# What Silently Invalidates a Cache



## Volatile system text

A timestamp, UUID, or session ID baked into the system prompt changes the bytes every call.



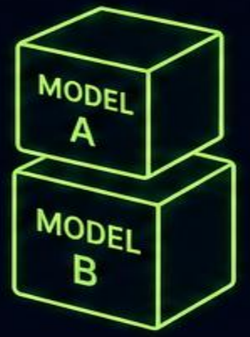
## Unsorted JSON

Serializing tools or data with unstable key order shifts the bytes even when the content matches.



## Shifting tools

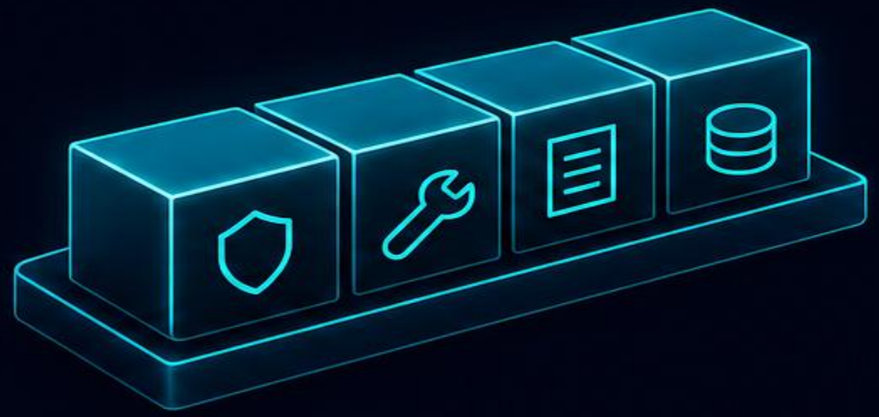
Tools sit at the front of the prefix, so changing the tool set invalidates everything after it.




## Model switch

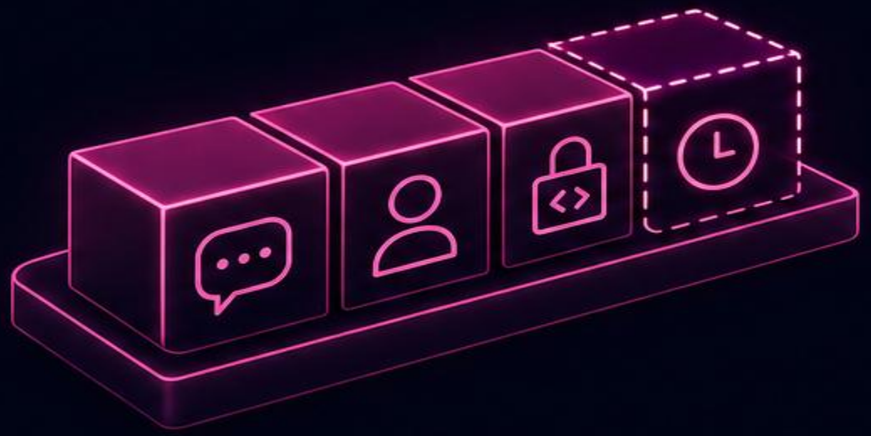
Caches are model-scoped. Move the same prefix to another model and you take a full miss.

# Frozen Prefix vs. Volatile Tail



## Frozen prefix

The stable front of the request, tools, system, and settled history. It caches once and gets reused byte-for-byte on later calls.



## Volatile tail

The changing part, the newest user turn and per-request data. It belongs after your last breakpoint so it never invalidates the cache.

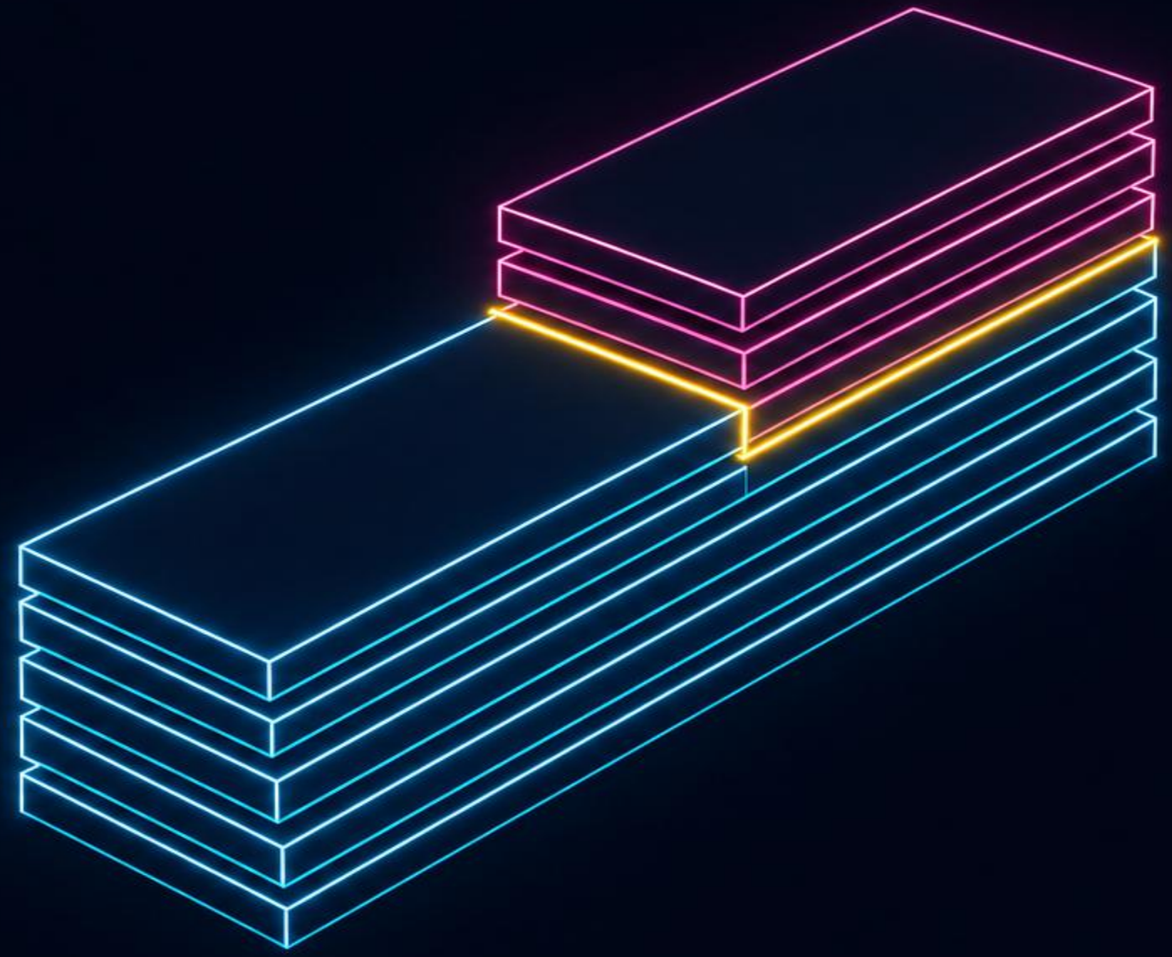
# Check-Pointing a Growing Chat

As a conversation grows, you re-checkpoint to keep caching the settled part.

**Move a breakpoint forward**
as earlier turns stop changing

**A breakpoint looks back**
at most 20 blocks for a prior entry

**Long tool loops**
past 20 blocks need intermediate breakpoints



# Per-Model Minimum Prefix



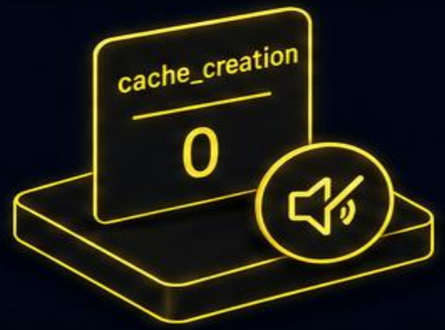

## A per-model floor

A prefix has to clear a token minimum to cache at all, and that floor is 1,024 tokens on Opus 4.8.




## It varies by tier

The floor is lower on some models, for example 512 tokens on Fable 5, so the same prefix can cache on one and not another.

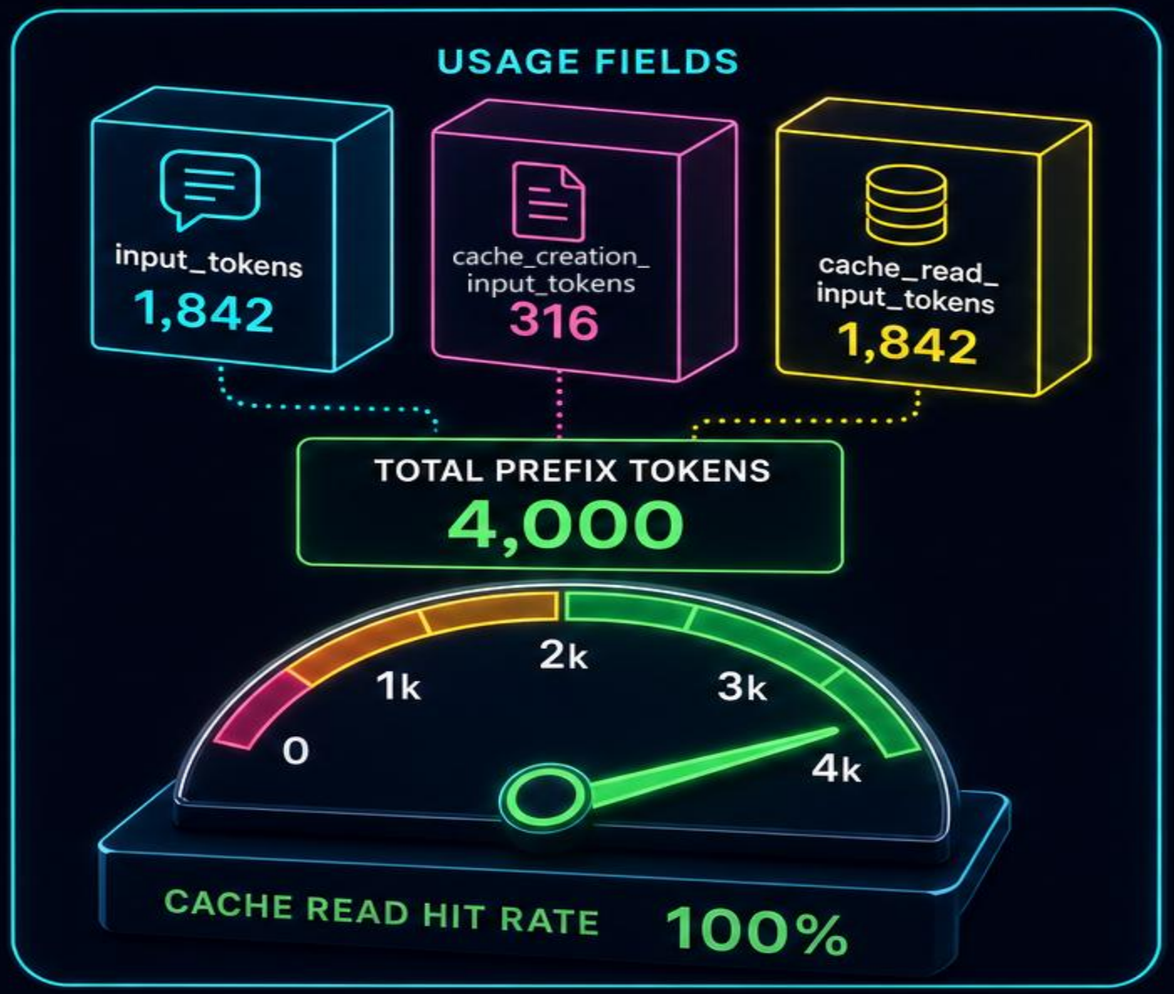



## Below the floor

A short prefix silently does not cache.
You get cache_creation of zero and
no error at all.

# Verifying Hits

You confirm caching works by reading the usage fields.




**Watch**
cache_read_input_tokens
on repeated identical calls

**A steady zero**
across repeats means
a silent invalidator

**Sum input, creation, and read tokens**
for the true prefix size

# Making the Prefix Match



**The prefix** renders tools, then system, then messages



**Place up to four** cache_control breakpoints at stable spots



**Timestamps, unsorted JSON, and model switches** invalidate silently



**A prefix clears a per-model floor,** 1,024 on Opus 4.8



**Verify hits** with cache_read_input_tokens