Claude Platform & API by Michaël Bettan

Definition: The Claude API is a **stateless** HTTP interface. The core endpoint, `POST /v1/messages`, takes the whole conversation (system prompt, alternating `user` / `assistant` turns, tool definitions) and returns one assistant message made of typed **content blocks** plus a `stop_reason` and token `usage`. Official SDKs (Python, TypeScript, Go, Java and others) wrap the same API and add retries, streaming helpers and types.

What your code owns: the API remembers nothing between calls and never runs your tools. Your application owns **history** (what to re-send, prune or compact), **tool execution** (and authorization), **branching on** `stop_reason`, **retries**, **validation** of outputs, and **cost/latency instrumentation**. Almost every exam-style defect is one of these responsibilities done wrong.

> MENTAL MODEL
> Treat every response as a **state transition**: read `stop_reason` first, then decide whether to show text, run tools, continue, compact, fall back or stop. Code that reads `content[0].text` without checking `stop_reason` is the root of most production bugs.

Workflow: 1. Build the request (model ID, `max_tokens`, system, messages, tools, optional thinking/effort and output format) → **2. Call** (sync, stream or batch: pick the transport by who is waiting and how much work there is) → **3. Branch on** `stop_reason` (`end_turn`, `tool_use`, `max_tokens`, `refusal`, and the rest) → **4. Append & loop** (append the assistant turn unchanged, add tool results, repeat until done).

1. Model family, effort and choosing a model

Tier	Profile	Typical fit	Watch out
Haiku	Fastest, lowest cost per token	High-volume classification, routing, extraction, real-time triage, sub-agent workers	Validate on your eval set before trusting it on nuanced steps
Sonnet	Balanced capability, speed and cost	**Default starting point** for most apps and agents, coding, tool use	Don't upgrade the whole pipeline when only one step fails
Opus	Most capable, slowest, highest cost	Hardest reasoning, long multi-step agent work, planning, review of critical output	Paying top-tier prices on steps a smaller tier passes

Two independent levers:

- **Model tier** sets the capability ceiling and the price per token.
- **Reasoning depth** is a per-request setting: adaptive thinking plus `output_config.effort` (`low`, `medium`, `high`, `xhigh`, `max`; the accepted set varies by model).
- Tune them separately. A smaller tier at higher effort can beat a larger tier at low effort on some tasks, and vice versa; only an eval tells you which.

Selection rule:

- Start on the balanced tier with the least reasoning that passes.
- Move **individual steps** up or down only on eval evidence; route easy vs. hard inputs to different tiers.
- **Planner → Executor tier routing (Spec-Driven pattern):** Use **Opus** (or high effort) for a read-only planning/specification pass that produces a structured spec artifact (data models, architecture, acceptance criteria), then start a **fresh context window** on **Sonnet** passing *only* the finalized spec to execute tool calls. This avoids paying Opus rates across dozens of implementation turns and keeps exploratory research noise out of the execution window.
- Treat every model or effort swap as a release: pinned ID, baseline comparison, rollback target.
- Turn thinking off or effort down on high-volume classifiers: thinking tokens are billed as output.

Sampling parameters on current models: on Claude 4.7 and later, setting `temperature`, `top_p` or `top_k` to any non-default value returns **HTTP 400**. Steer behavior with prompting, examples, structured outputs and effort instead. Even on older models where `temperature: 0` is accepted, output was **never fully deterministic**: test properties, not exact strings.

> COMMON TRAP
> "Use the best model everywhere" is still a tier decision, just the most expensive one. And "set temperature to 0 for consistency" fails twice: it 400s on the newest models and never guaranteed identical text anyway.

2. Messages API: request shape and roles

Field	Required?	What it does
`model`	Yes	Model ID. Pin a fixed ID in production (see §17).
`max_tokens`	Yes	Cap on **output** tokens for this response. Does not create context room or fix validation errors.

Field	Required?	What it does
`messages`	Yes	Conversation as `user` / `assistant` turns; content is a string or an array of blocks.
`system`	No	**Top-level** system prompt (string or text blocks) for instructions that apply to the whole conversation. Newer models also accept **mid-conversation** `role: "system"` messages inside `messages`, placed **after a user turn**, to change instructions later without invalidating the cached prefix (check per-model support).
`tools` / `tool_choice`	No	Tool definitions and how Claude may use them (§4).
`thinking` / `output_config`	No	Adaptive thinking, `effort`, and `format` for structured outputs (§7, §9).
`stream`	No	Server-sent events instead of one JSON body (§6).
`stop_sequences`	No	Custom strings that end generation (`stop_reason: "stop_sequence"`).
`inference_geo`	No	First-party data residency: `"global"` (default) or `"us"` (§17).

- **Input content blocks:** `text`, `image`, `document`, `tool_result` (in user turns); prior `text`, `tool_use`, `thinking`, `redacted_thinking` (in assistant turns you echo back).
- **Response object:** `id`, `role: "assistant"`, `content[]`, `stop_reason`, `stop_sequence`, `usage` (`input_tokens`, `output_tokens`, `cache_creation_input_tokens`, `cache_read_input_tokens`).
- **Transport choice:** user waiting + short answer → sync. User waiting + long answer → streaming. Many concurrent in-app calls → async client. Bulk offline work → Message Batches.

```python
import anthropic
client = anthropic.Anthropic()  # reads ANTHROPIC_API_KEY from the environment

resp = client.messages.create(
    model=PINNED_MODEL_ID,
    max_tokens=1024,
    system="You are a support assistant for Acme. Answer only from the provided policy.",
    messages=[
        {"role": "user", "content": "Can I return an opened item?"},
    ],
)
if resp.stop_reason == "end_turn":
    answer = next(b.text for b in resp.content if b.type == "text")
```

System prompt design — the "Goldilocks zone" (prompt altitude):

Altitude	Characteristics	Failure mode
Too specific (brittle)	Treats the LLM as a deterministic state machine; hardcodes rigid branching ("if billing dispute, ask *exactly* 3 follow-up questions in this order") and tries to enumerate every edge case	Breaks on unscripted edge cases; high maintenance burden (if logic is 100% predetermined, use deterministic code, not an agent)
Goldilocks zone (heuristics)	Defines **1. Clear identity & scope**, **2. Goal-oriented tool guidance** (what tools exist and the goal, not rigid call sequences), **3. A repeatable reasoning framework** (*Understand* → *Gather context with tools* → *Act* → *Verify*), and **4. Explicit boundaries & escalation triggers** (e.g. legal, health/safety, or refunds above policy limits)	Generalizes cleanly to novel inputs while keeping behavior predictable and bounded
Too vague (underspecified)	High-level platitudes ("solve issues consistent with our brand essence; escalate if needed"); false assumption of shared context; undefined escalation boundaries	High variance across runs; model guesses company policy and either over-escalates or hallucinates rules

COMMON TRAP

Prefilling the final assistant turn to force a format is rejected with a 400 on Claude 4.6 and later. Use structured outputs (§7) instead. Also, `content[0]` is not always text: with thinking on, the first block is usually a `thinking` block. Select blocks by `type`.

3. Explicit criteria and few-shot examples

Why it matters: in review, triage and extraction pipelines, **precision** (how many reported items are real) decides whether people trust the output. Vague instructions produce noisy output; confidence wording does not fix it. Define *what qualifies*, then show it.

Vague instruction	Why it fails	Explicit replacement
"Check that comments are accurate"	Everything slightly imprecise gets flagged, so false positives pile up	"Flag a comment only when the behavior it claims **contradicts** what the code does"
"Be conservative" / "Only report high-confidence findings"	Doesn't define the boundary; the model's notion of "confident" is unchanged, so precision barely moves	Categorical **report / skip** lists: report bugs, security flaws, data loss; skip style, naming, and patterns used consistently in the repo

Vague instruction	Why it fails	Explicit replacement
"Rate the severity"	Labels drift between runs and reviewers	A severity rubric with **one concrete code example per level** (critical / major / minor)

Precision and trust:

- One noisy category makes users ignore the accurate categories too. Measure precision **per category**, not just overall.
- If one category is mostly noise, **temporarily disable it** (stop reporting it) while you rewrite its criteria and re-measure on an eval set; re-enable it once precision recovers.

Few-shot (multishot) examples are the most reliable fix when detailed instructions still give inconsistent format or judgment. Anthropic's prompting guidance: make examples relevant, diverse, and wrapped in `<example>` tags (inside `<examples>`); 3–5 examples usually work best.

Use few-shot to...	What the examples show
Handle **ambiguous cases** (which tool, escalate or not, is this a real gap)	The chosen action **and why** it beats the plausible alternative
Fix **output format**	The exact finding shape: location, issue, severity, suggested fix
Cut **false positives**	Contrast pairs: an acceptable pattern (not reported) next to a genuine issue (reported)
Reduce **extraction hallucination**	Correct extraction from varied layouts (inline citations vs. bibliography, informal amounts like "a couple of cups"), including an explicit `null` when a field is absent

- Examples teach a **judgment** that generalizes to new cases; a rule list matches only what you enumerated.
- Keep examples varied: near-identical examples get copied on surface features (length, wording, field order).

```
<examples>
  <example>
    <code>if user.is_admin or True:</code>
    <finding>{"location": "auth.py:42", "issue": "condition is
always true, so the admin check is bypassed",
              "severity": "critical", "suggested_fix": "remove 'or
True'"}</finding>
  </example>
  <example>
    <code>for i in range(len(items)):   # idiom used across this
repo</code>
    <finding>none: style preference consistent with the codebase,
not reported</finding>
  </example>
</examples>
```

COMMON TRAP

Fixing a noisy reviewer by adding "only report findings you are very confident about". It doesn't define what counts. Write categorical criteria, add contrast examples, and measure precision per category.

4. The tool-use loop

Workflow: 1. Define tools (`name`, `description` covering what it does, when to use it and when *not* to, `input_schema`) → **2. Claude requests** (response has `tool_use` blocks, each with an `id`, and `stop_reason: "tool_use"`) → **3. Your code executes** (authorize, validate inputs, run; add a human gate before irreversible actions) → **4. Return results** (append the assistant turn unchanged, then one user turn with a `tool_result` per call).

Pairing rule	What breaks if you ignore it
Every `tool_use` gets a `tool_result` whose `tool_use_id` equals the call's `id`	400: result references an unknown ID, or a call is left unanswered
Results go in the **very next** user message; nothing in between	400 on the next request
In that user message, `tool_result` blocks come **first**; any text goes after them	Validation error, or the turn ends early
Parallel calls in one turn are all answered in **one** user message	Orphaned calls; splitting results across turns fails
Failures are returned as a `tool_result` with `is_error: true` and a short message	An empty string reads as "no data", so Claude confidently reports a false result

```
messages.append({"role": "assistant", "content": resp.content})   # echo unchanged
results = []
for block in resp.content:
    if block.type == "tool_use":
        try:
            out = run_tool(block.name, block.input)  # your authz + validation inside
            results.append({"type": "tool_result", "tool_use_id": block.id, "content": out})
        except ToolError as e:
            results.append({"type": "tool_result", "tool_use_id": block.id,
                            "content": f"lookup failed: {e.short}", "is_error": True})
messages.append({"role": "user", "content": results})  # all results, one turn, first
```

`tool_choice`	Behavior	Notes
`{"type": "auto"}` (default)	Claude decides whether to call a tool	Add `"disable_parallel_tool_use": true` **inside** this object for at most one call per response

`tool_choice`	Behavior	Notes
`{"type": "any"}`	Must call some tool	Forced use: not allowed with manual extended thinking, and some of the newest models return 400

`tool_choice`	Behavior	Notes
`{"type": "tool", "name": "..."}`	Must call that tool	Same restrictions as `any`; prefer `auto` + `strict: true` or structured outputs where forcing is rejected
`{"type": "none"}`	No tool calls	Tools stay defined (keeps the cache prefix stable)

Tool design:

- Description = purpose + when to use + an **exclusion** sentence ("not for refunds").
- Small, well-named tool set; overlapping tools cause wrong picks.
- Mark truly required fields in `required`; use enums for closed sets.
- `strict: true` guarantees tool inputs match the schema.

Client vs. server tools & visual verification:

- **Client tools:** you execute them and return `tool_result` (which can include both `"type": "text"` and `"type": "image"` blocks, e.g., Playwright screenshots for closed-loop visual UI verification).
- **Server tools** (e.g. web search): Anthropic executes them inside the turn.
- A long server-tool loop can stop with `pause_turn`: send the assistant content back as-is to continue.

Agent taxonomy & scaling tool execution (Shallow ReAct vs. Deep Agents & CodeMode):

Architecture / Pattern	How it works	Trade-offs & when to use
Workflow / Hybrid RAG	LLM routes between predefined pipeline steps (rewrite query → retrieve → grade → answer)	Bounded pipelines with fixed control flow; cannot handle open-ended multi-step tasks
Shallow ReAct Loop (Direct JSON Tool Calling / Standard MCP)	Single `messages` loop: Reason → emit `tool_use` JSON (often via an **MCP Server** connected 1:1 to an **MCP Client** inside the **Host**, reducing M × N integrations to M + N) → append `tool_result` → repeat	Best for **1–5 tool iterations**. **4 scaling limits:** (1) **Upfront schema bloat** (50+ MCP tools burn ~50k tokens on turn 1), (2) **Ping-pong latency** (*K* chained calls = *K* full LLM inferences + intermediate `tool_result` bloat), (3) **JSON vs. code pretraining mismatch**, (4) **Schema limits** (JSON Schema enforces parameter types, not multi-step call order)
Deep Agent Harness	ReAct loop augmented with **(1) explicit planning tool (** `TodoWrite` **)**, **(2) isolated subagents**, **(3) virtual filesystem** (`glob`, `grep`, paginated `read_file(offset, limit)`, `write_file`), and **(4) heuristic system prompt**	Best for **long-horizon tasks** (30+ turns). Offloads bulky intermediate state to files/storage instead of `messages`, tracks `pending` / `in_progress` / `completed` steps via `TodoWrite`, and prevents blind retry loops
Code-orchestrated tool calling (Anthropic programmatic tool calling; Cloudflare Code Mode)	**Anthropic programmatic tool calling:** Claude writes Python that runs in Anthropic's code execution container; tools opt in via `allowed_callers`, and intermediate results stay out of context. **Cloudflare Code Mode** (separate third-party variant): converts MCP tool schemas into a typed **TypeScript API** (with docstrings) and exposes **one tool**: a sandboxed code runner. Model writes TypeScript that loops/chains multiple API calls; a **network-isolated V8 sandbox** routes calls via RPC bindings to MCP servers and returns `console.log()` output	Best for **multi-step tool chaining** and **large API surfaces**: collapses *K* sequential LLM round-trips into **1 inference pass** and keeps intermediate payloads out of `messages`

> COMMON TRAP
>
> "Unknown `tool_use_id`" is a **wiring** bug, not a schema or description bug. And `disable_parallel_tool_use` is not a top-level request field: it lives inside `tool_choice`.

5. stop_reason values and handling

`stop_reason`	Meaning	Correct handling
`end_turn`	Natural finish	Use the response
`tool_use`	Claude is calling one or more tools	Run them, return paired `tool_result`s, call again
`max_tokens`	Hit **your** `max_tokens` cap (or the model maximum)	Raise the cap or continue the response; never parse truncated JSON as complete
`model_context_window_exceeded`	Generation filled the **context window**	Treat as truncated; prune/compact history before the next turn
`stop_sequence`	One of your `stop_sequences` fired	Read `stop_sequence` to see which
`pause_turn`	A long server-tool loop was paused	Send the assistant content back unchanged to continue
`refusal`	Claude declined (safety classifiers)	Returned as **HTTP 200**. Read `stop_details`; don't blind-retry the same request; reset or rephrase the refused turn, or retry on a fallback model where offered

Refusals in practice:

- `stop_details` can carry a policy `category` (may be `null`) and an explanation that isn't guaranteed stable.
- Leaving the refused turn in history leads to **continued refusals**: remove or rephrase it, or clear history.
- When streaming, the refusal arrives on the `message_delta` event; discard the partial turn.
- An HTTP-status-only error check misses refusals entirely.

`max_tokens` vs. context window:

- **Input alone** larger than the window → 400 "prompt is too long", before any generation, on every model.

- Input fits, but input + `max_tokens` exceeds the window → newer models accept it and stop with `model_context_window_exceeded` if they run out.
- The API never silently drops old turns: history management is your job.

> **COMMON TRAP**
> A refusal is a content decision, not a transient error: exponential backoff on the same request just repeats it. And `max_tokens` limits output length only; raising it does nothing for an oversized prompt or a crowded context.

6. Streaming events

Event (in order)	Carries	Handler action
`message_start`	Message shell, initial usage	Open a buffer for the turn
`content_block_start`	Block `index` and type (text, tool_use, thinking)	Start a per-index accumulator
`content_block_delta`	`text_delta`, `input_json_delta` (partial tool JSON), thinking deltas	Render text; **append** JSON fragments, don't parse yet
`content_block_stop`	Block complete	Now parse tool input JSON and validate
`message_delta`	Top-level changes: `stop_reason`, cumulative output usage	Branch on `stop_reason` (refusal, tool_use, max_tokens…)
`message_stop`	Message complete	**Only now** commit the assistant turn to history
`ping`	Keep-alive	Ignore

Why stream:

- Improves **perceived** latency (time to first token) for a waiting user.
- Recommended for long requests: very long non-streamed calls risk timeouts, and the Python SDK requires streaming above roughly 21k `max_tokens`.
- Does **not** reduce total tokens, cost or request count.

Interrupted streams:

- "The read loop ended" ≠ "the message is complete". Only `message_stop` means complete.
- On a drop, discard the partial turn and retry from the last complete turn.
- SDK helpers (e.g. a final-message accumulator) do the bookkeeping for you.

> **COMMON TRAP**
> Saving a half-streamed assistant turn leaves a malformed `tool_use` in history, and every later request fails validation, often surfacing only on retries. Parse tool input at `content_block_stop`; commit the turn at `message_stop`.

7. Structured outputs

Approach	Guarantee	Use when
System-prompt output formatting (e.g. concise bullet points, markdown tables, structured sections)	None: best-effort prompt guidance (shapes structured reasoning and cuts conversational prose)	Interactive user experiences, human-readable summaries, streaming markdown responses
JSON outputs: `output_config.format = {"type": "json_schema", "schema": {...}}`	The response text parses and matches the schema (constrained decoding)	The answer itself must be machine-readable (extraction, classification, CI gates, reports)
Strict tool use: `"strict": true` on a tool	Tool name and inputs match `input_schema`	Agents whose tool calls feed real systems

```
resp = client.messages.create(
    model=PINNED_MODEL_ID, max_tokens=1024,
    messages=[{"role": "user", "content": email_text}],
    output_config={"format": {"type": "json_schema", "schema": {
        "type": "object",
        "properties": {"name": {"type": "string"},
"demo_requested": {"type": "boolean"}},
        "required": ["name", "demo_requested"],
        "additionalProperties": False}}},
)
if resp.stop_reason == "end_turn":                    # check
before parsing
    data = json.loads(next(b.text for b in resp.content if b.type
== "text"))
```

Production constraints:

- **Shape ≠ correctness.** A schema-valid object can still hold a wrong date or a hallucinated value. Keep evals and business-rule validation.
- **Still check `stop_reason`.** A `refusal` or `max_tokens` stop can leave output that doesn't conform.
- **First-call latency.** A new schema is compiled on first use and then cached, so the first request is slower.
- JSON outputs and strict tools can be used together in one request; both replace the old prefill trick, which newer models reject.

> **COMMON TRAP**
> When a parser breaks on "almost JSON", don't add a third paragraph of formatting instructions or a regex repair step. Move format enforcement into the API with `output_config.format` or `strict: true`.

Extraction via tool use: define a tool whose `input_schema` *is* the record you want, then read the `tool_use` block's `input` (already a parsed object). Nothing has to be executed: the tool call is the output.

`tool_choice` for extraction	Guarantee	Use it when
`{"type": "auto"}`	Claude may answer in **text** instead of calling the tool	You can tolerate a text reply, or the prompt states clearly when the tool applies
`{"type": "any"}`	Must call **one** of the tools; Claude picks which	Several extraction schemas (invoice, receipt, contract) and the document type is unknown

`tool_choice` for extraction	Guarantee	Use it when
`{"type": "tool", "name": "extract_metadata"}`	Must call **that** tool	A specific extraction must run first (e.g. metadata before enrichment steps)

- Forced choice (`any` / `tool`) errors with manual extended thinking, and some of the newest models reject it with a 400. There, use `auto` + `strict: true` + a prompt that says when to call the tool, or `output_config.format`.
- Where forced use is accepted, the response starts with the tool call and contains no `thinking` block.

Schema design for extraction:

Choice	Why	Example
Put in `required` only fields every source contains	A required field the document lacks pushes the model to **invent** a value	`invoice_number` required; `po_number` optional
Make may-be-missing fields **nullable** (`"type": ["string", "null"]`)	"Not stated" becomes a valid answer instead of a fabrication	`due_date` nullable + "use null if not stated". Required-but-nullable keeps the key always present
Enum + `"other"` + a detail string	Extensible categories without losing information	`category: [..., "other"]` + `category_detail`
Add an `"unclear"` enum value	The model can flag ambiguity instead of guessing	`sentiment: [positive, negative, neutral, unclear]`
Put **format-normalization rules** in the prompt	The schema fixes types, not how messy source formats are rendered	"Dates as YYYY-MM-DD; amounts as numbers; weights in grams"

```python
record_invoice = {
    "name": "record_invoice", "strict": True,
    "description": "Record fields from ONE invoice. Use null when a field is not stated.",
    "input_schema": {"type": "object", "additionalProperties": False,
        "properties": {
            "invoice_number":  {"type": "string"},
            "due_date":        {"type": ["string", "null"], "description": "YYYY-MM-DD, or null if absent"},
            "category":        {"type": "string", "enum": ["hardware", "software", "services", "other"]},
            "category_detail": {"type": ["string", "null"]},
            "stated_total":    {"type": "number"}},
        "required": ["invoice_number", "due_date", "category", "category_detail", "stated_total"]}}

resp = client.messages.create(model=PINNED_MODEL_ID, max_tokens=1024, tools=[record_invoice],
                              tool_choice={"type": "tool", "name": "record_invoice"},
                              messages=[{"role": "user", "content": invoice_text}])
record = next(b for b in resp.content if b.type == "tool_use").input  # dict, schema-valid
```

Strict-schema limits (verify against current docs): objects need `additionalProperties: false`; numeric (`minimum`, `maximum`) and string-length constraints aren't supported (400), so enforce ranges in your validator; complexity caps include 20 strict tools per request, 24 optional parameters and 16 union-type (e.g. nullable) parameters across strict schemas. Enum capitalization isn't guaranteed: compare case-insensitively.

> **COMMON TRAP**
> Marking every field `required` "for completeness". When the source lacks the value, a schema-valid but invented value comes back. Make it nullable or optional and say "null if not stated".

8. Validation, retry and feedback loops

Error class	Example	Removed by strict schema?	Fix
Syntax / shape	Broken JSON, wrong type, extra key	**Yes** (strict tools, JSON outputs)	Nothing more to do
Semantic	Line items don't sum to the total; value in the wrong field; impossible date	**No**	Validate in code → retry with error feedback → human review
Information absent	The total is only in an attachment you didn't send	**No**, and retries can't help	Return `null` / flag it, fetch the missing source, or route to a human

Retry with error feedback:

- The follow-up request carries the **original document**, the **failed extraction** and the **specific validation errors** ("line items sum to 1,140.00 but `stated_total` is 1,240.00; check for a missing line or misread digit").
- Retries work for format mismatches, structural mistakes, misplaced fields and slips the source can resolve.
- Retries **fail** when the information isn't in the input: pushing harder makes the model invent a value that passes your check. Cap attempts (1–2), then flag or route to review.

Fields that make errors visible:

- `calculated_total` vs. `stated_total`: the model sums the line items *and* copies the printed total; your code compares them and flags mismatches.
- `conflict_detected` (boolean + note): the source contradicts itself (two different dates for one event). Report it; don't let the model pick one silently.
- `detected_pattern` on review findings: name the construct that triggered the finding (e.g. `string-built-sql`, `unchecked-return`). When developers dismiss findings, group dismissals by pattern to see which criteria cause false positives, then rewrite those criteria (§3).

```python
def extract_with_retry(doc, max_retries=2):
    messages = [{"role": "user", "content": doc}]
    for attempt in range(max_retries + 1):
        resp = client.messages.create(model=PINNED_MODEL_ID, max_tokens=2048, tools=[record_invoice],
                                      tool_choice={"type": "tool", "name": "record_invoice"}, messages=messages)
        call = next(b for b in resp.content if b.type == "tool_use")
        errors = validate(call.input)  # sums, ranges, cross-field business rules
        if not errors:
            return call.input
        messages += [{"role": "assistant", "content": resp.content},
                     {"role": "user", "content": [{"type": "tool_result", "tool_use_id": call.id, "is_error": True,
                         "content": "Validation failed:\n" + "\n".join(errors) +
                                    "\nRe-read the document and fix only these fields. Use null if the value is not in it."}]}]
    return route_to_review(doc, call.input, errors)  # don't loop forever on absent data
```

COMMON TRAP

"Retry up to five times until validation passes." If the value isn't in the source, the loop eventually produces a fabricated value that *passes*. Absent data is an input problem: flag it or fetch the source.

9. Adaptive thinking

What changed: the older **extended thinking** mode made you allocate a fixed token budget. **Adaptive thinking** lets the model judge how much to reason per request; you steer depth with `effort`.

- API shape: `thinking={"type": "adaptive"}` plus `output_config={"effort": "high"}`.
- Manual `{"type": "enabled", "budget_tokens": N}` is deprecated on Claude 4.6 and returns **400 on 4.7 and later**.
- Migration: drop `budget_tokens`, switch the type to `adaptive`, control depth with `effort`.

Display and billing:

- `thinking.display`: `"summarized"` returns readable summaries; `"omitted"` returns an empty thinking field.
- You are billed for the **full** internal thinking as output tokens either way; display changes only what you see.
- Higher effort is not always better: it adds latency and cost. Match effort to task difficulty.

Rule in tool-use loops	Why
Echo the assistant message **exactly**, including `thinking` and `redacted_thinking` blocks, when you return tool results	Thinking blocks carry a cryptographic `signature`; editing, summarizing or filtering them gives a 400
Keep one thinking configuration per assistant turn	A tool-use loop is one assistant turn; change settings only between turns
Manual extended thinking allows only `tool_choice` `auto` or `none`	Forced tool use is incompatible with manual mode
To save context, prune or compact **older completed** turns	Never "save tokens" by stripping blocks from the active turn

COMMON TRAP

Changing `thinking` or `effort` between requests invalidates the messages prompt cache (and on some models the system and tool caches too). Pick a setting per workload and keep it stable.

10. Context window, compaction and prompt caching

Context strategy (Write / Select / Compress / Isolate)	How	Use when
Progressive disclosure & tool scoping (Select)	Expose lightweight ~100–200 token metadata (Skills / tool search) or task-scoped `tools[]`; load full schemas/instructions only on demand	Large tool catalogs or multi-MCP agents where 50+ eager `tools[]` schemas burn ~50k tokens on **turn 1**
Virtual filesystem offloading (Write & Select)	Write bulky intermediate outputs to files/DB (`write_file`); retrieve slices via `glob`, `grep`, and paginated `read_file(offset, limit)`	Long-horizon Deep Agents where keeping full file reads/search dumps in `messages` causes **over-retrieval**
Prune / Hard reset (Compress)	Drop stale tool results in code, or start a fresh `messages: []` array when switching tasks or after final spec approval	Tool outputs dominate history, or user switches to a new phase/unrelated task
Context editing (Compress)	Beta `context_management` edits that clear old tool uses (and thinking) server-side past a trigger	Long agent loops with bulky, one-time tool output

Context strategy (Write / Select / Compress / Isolate)	How	Use when
Compaction (Compress)	Beta `context_management` edit `compact_20260112`: older history is summarized into a compaction block (default trigger 150k input tokens; optional `pause_after_compaction` returns `stop_reason: "compaction"`)	Persistent agents that run for hours or days on a **single continuing task**
Sub-agent handoff (Isolate)	Give a bounded sub-task its own fresh `messages: []` array; return only a condensed summary or final artifact	Noisy exploration, parallel wave generation, or independent QA review
Measure	Token-counting endpoint before sending; watch `usage`	Always; production payloads are larger than test fixtures

> **DIAGNOSIS RULE (TURN 1 VS. TURN 5 FAILURE)**
> If an agent picks the wrong tool from **turn 1** after connecting multiple MCP servers, the culprit is **upfront tool-schema context pollution** (scope down `tools[]` or use progressive disclosure). If it picks the right tool on **turns 1–4** and fails from **turn 5**, the schemas are fine — **accumulated `tool_result` output** is crowding out instructions (prune, edit, compact, or offload to a virtual filesystem).

Context Rot taxonomy (3 long-context failure modes) & Multi-agent orchestration rules:

- **1. Context poisoning:** A hallucination, false assumption, or corrupted tool output enters `messages` and compounds across future turns. **Remedy:** Prune/rewind the poisoned turns or hard-reset `messages: []` (summarization/compaction often *preserves* the poisoned claim inside the summary!).
- **2. Context confusion:** Unnecessary or bulky context (full 1,500-line file reads — **over-retrieval**, irrelevant docs, or unused tool schemas) dilutes attention and steers the response. **Remedy:** Paginate reads (`offset` / `limit`), prune old `tool_result`s, or isolate in a subagent.
- **3. Context clash:** Conflicting rules or instructions appear across the prompt (e.g., system prompt rules vs. stale earlier turns or injected docs). **Remedy:** Deduplicate instructions and drop superseded turns.
- **Multi-agent context contracts (Subagent handoff, Wave fan-out & Builder–Reviewer separation):**
- **Self-contained handoff & final-message contract:** A subagent call (`messages.create`) starts with a clean context window (**zero parent history**), and its intermediate tool turns are discarded on return — the orchestrator must provide a self-contained task prompt, and the subagent's instructions must mandate returning the **verbatim output artifact or file reference** rather than a conversational completion note.
- **Batch concurrency control & when NOT to parallelize:** Because isolated subagents cannot inspect sibling executions, provide each worker with an explicit task bundle (target path, input snapshot, and boundary constraints), and dispatch parallel calls in **bounded batches (e.g. 3–5 concurrent calls)** to prevent HTTP 429 rate limits and orchestrator context bloat. **Never parallelize** sequentially dependent tasks (e.g., consumer and producer without a locked interface contract), shared stateful resources, trivial short operations, or exploratory debugging.
- **Builder–Reviewer isolation (bias mitigation):** Asking the builder to critique its own code/UI in the same `messages` thread anchors it to its prior `thinking` and rationalizations. Always run the Reviewer in a **separate API call with a fresh `messages` list** containing only the spec, rubric, and final diff/screenshot.

Prompt caching mechanics:

- Mark the end of a reusable prefix with `cache_control: {"type": "ephemeral"}`; up to **4 breakpoints** per request.
- Prefix order is `tools` → `system` → `messages`. A change at one level invalidates that level and everything after it.
- Cache hits need an **identical** prefix: static content first, dynamic content after the last breakpoint.
- Default TTL is 5 minutes (refreshed on use); `"ttl": "1h"` is optional.
- Track `cache_creation_input_tokens` vs. `cache_read_input_tokens` in `usage`.

Caching economics:

- Cache **write:** 1.25× base input (5-min TTL) or 2× (1-hour TTL).
- Cache **read:** about 0.1× base input (lower on some newer models; check pricing).
- Break-even: a 5-min entry pays off after 1 read; a 1-hour entry after 2 reads.
- Also cuts time to first token on long prefixes.
- Silent invalidators: editing tool definitions, changing `tool_choice`, toggling images, changing thinking/effort, or **swapping `system` output styles/personas mid-session** (which busts both `system` and the entire downstream `messages` cache — run separate parallel sessions per persona or place dynamic format shifts after the last `messages` breakpoint).

> **COMMON TRAP**
> A 0% hit rate with a timestamp, user ID or ticket text placed *before* the breakpoint. Each request writes a new entry and never reads one. A longer TTL only makes those writes cost more; move the dynamic text after the breakpoint.

11. Preserving critical facts in long sessions

The problem: the API is stateless, so every request must carry the conversation. As sessions grow, the tempting fixes (summarize, compact, drop turns) quietly lose the **exact** facts the task depends on.

Risk	Symptom	Fix
Progressive summarization	"Refund of 1,240.18 for order A-7731, promised by 14 March" becomes "customer wants a refund"	Keep a persistent **case facts** block outside the summarized history
Tool-output bloat	An order lookup returns 40+ fields when 5 matter; resent on every later turn	**Trim** tool output to the relevant fields in your wrapper *before* appending the `tool_result`

Risk	Symptom	Fix
Lost in the middle	Findings buried in the middle of a long aggregated input get skipped	Put a **key-findings summary first**; organize details under explicit section headers or XML tags
Partial history	Only the latest user message is sent; the model loses commitments and context	Send the complete (or deliberately managed) history every request

Case facts block:

- Extract transactional facts (amounts, dates, order IDs, statuses, promised actions, the customer's stated expectation) into a structured block your **code** owns and updates when tools return new data.
- Re-inject it on every request. For multi-issue sessions, keep one entry per issue keyed by order or ticket ID.
- Place it after the last cache breakpoint so updates don't invalidate the cached static prefix (§10).

```
case_facts = {"customer_id": "C-1029",
              "issues": {"A-7731": {"amount": 1240.18, "ask": "refund", "promised_by": "2026-03-14",
                                    "status": "return label sent"}}}
system = [{"type": "text", "text": STATIC_POLICY, "cache_control": {"type": "ephemeral"}},
          {"type": "text", "text": "<case_facts>\n" + json.dumps(case_facts) + "\n</case_facts>"}]
```

Long inputs (lost-in-the-middle mitigation):

- Models handle the start and end of a long input more reliably than the middle.
- Anthropic's long-context guidance: put long documents **at the top** and the query and instructions **at the end** (tests showed up to ~30% better answers on complex multi-document inputs); wrap each document in `<document>` tags with source metadata.
- When aggregating subagent or tool results, lead with a short summary of key findings, then the detail under headers.
- Ask upstream agents to return **structured data** (key facts, dates, source locations, citations, relevance scores) rather than verbose prose and reasoning when the downstream context budget is tight. Orchestration patterns live in the Claude Agent SDK guide.

> COMMON TRAP
> Trusting compaction or a rolling summary to keep amounts, dates and IDs. Summaries paraphrase. Keep authoritative facts in a structured block your code maintains, and trim tool output at the source.

12. Message Batches

- **What it is:** an asynchronous endpoint that accepts many Messages requests in one submission and processes them in the background at **50% of standard price** (input and output).
- **Limits:** capped at **256 MB total payload or 100,000 requests** per batch (whichever threshold is reached first). Most batches finish within an hour; anything not done in **24 hours** expires.
- `custom_id`: required, unique per request (1–64 chars, letters, digits, `_`, `-`). Results can come back **in any order**, so match on `custom_id`, never on position.
- **Lifecycle:** create → poll until `processing_status` is `ended` → fetch results (each succeeded, errored, canceled or expired). Results stay downloadable for 29 days.
- **No latency SLA:** "most finish within an hour" is typical, not promised; under heavy demand more requests can expire at 24 hours.
- **One response per request:** a batched request can include tools and prior turns, but your code can't run a tool mid-request and hand back the result. If a result stops on `tool_use`, the next step needs a new request. Batches suit single-shot work; forced extraction tools (§7) fit because the `tool_use` input *is* the output.

Workload	Best transport	Why
Nightly classification of 40k tickets, read next morning	**Message Batches**	Half price, no rate-limit fight, latency doesn't matter
Pre-merge PR check a developer is waiting on	Synchronous API	Blocking workflow; batch has no latency guarantee
Agent that must query your database between steps	Synchronous tool loop	Batch can't run your tools mid-request
Chat UI with long answers	Streaming	A person is waiting; perceived latency
Backend fan-out of a few hundred calls per user action	Async client with bounded concurrency	Needs results in seconds
Same 5k-token instructions on every batched item	Batches + prompt caching	The discounts can be combined (batch cache hits are best effort)

Operating a batch pipeline:

- **Refine on a sample first:** run the prompt on a representative sample, fix failures, *then* submit the large batch. First-pass success is what saves money.
- **Resubmit only failures:** collect `errored` / `expired` results by `custom_id` and resubmit just those, with fixes (e.g. split documents that exceeded the context window). Errored, canceled and expired requests aren't billed.
- **SLA math:** the worst case is *wait for the next submission + up to 24 h processing*. To promise results within 30 h of a document arriving, submit at least every 30 − 24 = 6 h, minus pre/post-processing time; a 4-hour cadence gives 4 + 24 = 28 h worst case.

> COMMON TRAP
> Splitting a list into chunks and calling `messages.create()` per item is **not** batching: it is serial synchronous traffic that hits rate limits. Smaller chunks or more backoff don't change that; the Batches API does. And the reverse trap: moving a **blocking** check (pre-merge, user waiting) to batches for the 50% saving, which removes any latency guarantee.

13. Multi-instance and multi-pass review

Why self-review is weak: the session that generated an artifact keeps its own reasoning in context and tends to defend it. "Now review your work critically" in the same thread, or more thinking, doesn't remove that anchor.

Pattern	How	Use it for
Independent reviewer instance	A separate `messages.create` with a fresh `messages` list holding only the spec/rubric and the artifact, **without** the generator's reasoning	Catching subtle bugs in generated code, contracts, extractions
Per-file local pass	One focused request per file: local bugs, security, logic	Large PRs where a single prompt dilutes attention
Cross-file integration pass	A separate pass over the per-file structured findings plus interfaces: data flow, contracts, consistency	Issues no single file shows; resolving contradictory local findings
Confidence-annotated verification	Each finding carries a self-reported confidence; route by threshold	Sending low-confidence findings to humans; thresholds must be calibrated (§16)

Workflow: 1. Generate (builder session produces the artifact) → **2. Local passes** (one fresh request per file; structured findings with `detected_pattern` and confidence) → **3. Integration pass** (new request over the findings and interfaces: cross-file data flow and contradictions) → **4. Route** (high-confidence findings auto-comment; low-confidence ones go to a human).

- Independence comes from **context**, not model choice; the reviewer can be the same model (a different tier is optional).
- Per-file passes are independent, so run them in parallel: sync for pre-merge checks, Message Batches for overnight audits (§12).
- See §10 (Builder–Reviewer isolation) for the context contract, and the Claude Agent SDK guide for orchestrating reviewers inside agent loops.

> **COMMON TRAP**
> Reviewing a 30-file PR in one prompt, or asking the generating session to "double-check". The first dilutes attention and yields contradictory findings; the second inherits the generator's blind spots. Split into local + integration passes and use an independent instance.

14. Errors and retries

Status / type	Class	Action
400 `invalid_request_error` (bad shape, prompt too long, unsupported param, spend limit you set)	Terminal	Fix the request; retrying the same body fails identically
401 authentication · 403 permission · 404 not found	Terminal	Fix key, permissions or model ID; fail fast
402 `billing_error` · 413 `request_too_large`	Terminal	Fix billing; shrink the payload below the endpoint's size limit
429 `rate_limit_error`	Retriable	Wait `retry-after`, then backoff + jitter. A spend-cap 429 has **no** `retry-after` and keeps failing
500 `api_error` · 504 `timeout_error`	Retriable	Exponential backoff; for 504 on long requests, stream instead
529 `overloaded_error`	Retriable	Backoff with jitter; consider a fallback model in the orchestration layer
HTTP 200 + `stop_reason: "refusal"`	Not an HTTP error	Content decision: handle as in §5, don't backoff-retry

SDK defaults:

- Official SDKs retry connection errors, 408, 409, 429 and ≥500 **2 times by default** with short exponential backoff, honoring `retry-after`.
- Configure with `max_retries` (client-wide or per request); `0` disables.
- Log the `request-id` response header for support.

Reliability placement:

- **Retry:** next to the call, owned by *one* layer.
- **Circuit breaker:** at the service boundary, on aggregate error rate.
- **Fallback** (other tier, cached answer, human queue): in orchestration.
- **Budgets:** max turns / tool calls per agent run.

> **COMMON TRAP**
> Wrapping SDK calls in your own 5-attempt loop with default SDK settings gives up to 15 attempts per request against a rate limit. One layer owns retries. And a bare `except Exception: pass` retries 400s pointlessly while hiding the real error.

15. Evals and cost levers

Output shape	Grader	Catches / misses
Label, number, enum, JSON field	Exact match / code assertion	Cheap, deterministic; can't judge free text
Structured object, tool trajectory	Code checks on properties (fields, ranges, expected/forbidden tools, call counts)	Robust to wording; misses tone and nuance
Free-text answer, summary, tone	LLM judge with a rubric	Scales; must be calibrated against human labels
High-stakes or novel behavior	Human review	Gold standard; slow and costly

Eval discipline:

- Define "done" first: success criteria, failure handling, cost and latency budgets.
- Golden set = representative inputs + edge cases + adversarial cases; refresh it whenever prompt, model or retrieval changes.
- Judge rules: reasoning **before** the score, anchored score bands, preferably a different model than the one graded, agreement checked against human labels.
- Change one variable per run; read per-case diffs, not just the mean.
- Assert properties, never exact strings: output varies even when correct.

Cost lever	Effect	Number / rule
Model tier per step	Largest price driver	Start balanced; move steps on eval evidence
Effort / thinking	Thinking billed as output	Low or off for classifiers and routers
Prompt caching	Cheaper repeated prefixes, faster first token	Write 1.25× / 2×, read ≈0.1×
Message Batches	Half price for offline work	50%, ≤24 h
Context size	Every resent token is billed again	Prune, edit, compact; count tokens
Call count / fan-out	Multi-agent runs multiply tokens	Fan out only for truly independent subtasks in bounded batches
Workspace spend caps (`settings/limits`)	Hard ceiling against runaway agent loops or leaked keys	Configure per-workspace USD caps in Claude Console to isolate dev/CI/subagent runs from prod
Streaming	Perceived latency only	No cost change
US-only residency (first party)	Premium	1.1× on Claude 4.6+ with `inference_geo: "us"`

COMMON TRAP

Quoting the per-call cost of a pilot. Instrument tokens, latency and errors per call from day one, multiply by production volume, size for the heavy tail and p95 rather than the median, and set the reliability floor before tuning cost.

16. Human review and confidence calibration

The core risk: an aggregate number hides segment failures. "97% field accuracy" can mean 99% on typed invoices and 70% on handwritten receipts, or one field (say `tax_id`) failing everywhere. Slice accuracy by **document type × field** before you reduce human review.

Practice	How	What it prevents
Segmented accuracy	Report accuracy per document type, per field, per source	Automating a segment that quietly fails
Field-level confidence	The model outputs a confidence per field (alongside `"unclear"` / `conflict_detected` flags)	One document-level score hiding one bad field
Calibration on labeled data	On a labeled validation set, measure real accuracy per confidence band; set thresholds per field / doc type where accuracy meets the target	Trusting raw self-reported confidence, which isn't calibrated until checked
Risk-based routing	Low confidence, conflicts, ambiguous or contradictory sources → human queue, highest-risk first	Spending scarce reviewer time on easy items
Stratified random sampling	Keep sampling **auto-approved, high-confidence** items within each stratum (doc type, field, source)	Unmeasured error rates and missed new error patterns or drift; plain random sampling under-samples rare types

Feedback loop: reviewer corrections become labeled data → recalibrate thresholds, add few-shot examples for the new error patterns (§3), extend the eval set (§15).

COMMON TRAP

"Overall accuracy is 97%, so remove human review", or "review only low-confidence items". The first ignores segments; the second never measures errors among auto-approved items. Validate per segment, calibrate thresholds on labeled data, and keep stratified sampling running.

17. Deploying: platforms, data residency and model-ID pinning

Platform	Operated by / identity	Residency control	Pick it when
Claude API (first party)	Anthropic; API keys and workspaces	`inference_geo` : `"global"` (default) or `"us"` (1.1×), Claude 4.6+ only. **No EU-only value**; older models 400 on the param	No binding cloud or EU constraint; newest features and betas first
Claude Platform on AWS	Anthropic-operated, AWS billing	Same `global` / `us` geographies	First-party features with AWS procurement
Amazon Bedrock	AWS-operated; IAM, AWS bill	Global endpoint, single-region endpoints, or geographic inference profiles (e.g. `eu.`-prefixed IDs); regional costs 10% more	Workload and compliance posture already live in AWS
Google Cloud (Vertex AI)	Google-operated; Cloud IAM and billing	Global, multi-region (`us` / `eu`) or regional (some models); 10% premium off global	Workload and compliance posture already in Google Cloud; EU residency
Microsoft Foundry	Anthropic-operated, on Azure	Varies by deployment type	Customer standardised on Azure

Model-ID pinning:

- **Claude 4.6 and later:** the dateless ID (e.g. `claude-sonnet-4-6`) *is* a fixed snapshot; updates ship under new IDs.
- **Earlier models:** dated IDs (`claude-haiku-4-5-20251001`); the short alias moves to the newest dated snapshot.
- Platform formats differ: Bedrock `anthropic.claude-…` (dated models end in `-v1:0`; optionally `global.` / `us.` / `eu.` prefixed), Google Cloud `…@YYYYMMDD` for dated models and the API format from 4.6 on.
- Every ID has its own deprecation and retirement date.

Release discipline:

- Pin the full model ID in config, not code.
- Keep the previous pin as the rollback target.
- Promote a new model only when evals meet or beat the pinned baseline.
- Feature availability differs by platform: verify each beta or tool you depend on.
- Measure latency from the customer's region with a real payload; compare total cost (fees, egress, integration), not just token price.

> **COMMON TRAP**
> "Inference must stay in the EU" cannot be met with first-party `inference_geo` (global or US only). Use Bedrock regional/EU routing or the Google Cloud `eu` multi-region, decided at scoping because residency is pass/fail. Encryption at rest does not change where inference runs.

18. Security for API apps

Risk	Control that actually bounds it	Not enough on its own
Indirect prompt injection via fetched pages, emails, files or tool output	Treat tool/fetched content as **data**; delimit it; scan it before it enters context; isolate readers from action tools	Filtering only the user's prompt
Excessive agency (agent sends, deletes, pays)	Least privilege at the **tool registry**: remove tools the task doesn't need; read-only credentials	"Never send email" in the system prompt
Side-effecting tool misuse	Deterministic authorization in your code **before** execution (allowlist, caller identity, amount scope), failing closed; human approval for irreversible actions	An output filter that runs after the tool already executed
Forged identity ("I am an admin")	Inject verified role and tenant server-side from your auth layer	Asking the model to verify the claim
Malformed or hostile tool inputs	`strict: true` plus server-side validation; parameterised queries; fixed write paths	Trusting model-generated arguments
Leaked secrets and PII	Keys in env/secret manager, never in prompts; pass reference IDs, not raw PII; per-tenant keys or workspaces	"Don't reveal secrets" instructions
Unsafe rendering of model output	Escape or sanitize before display; never execute model HTML/JS	Assuming model output is trusted
Untrusted MCP server or programmatic tool code (`Code Mode`)	Audit 3rd-party MCP code; execute model-generated tool orchestration inside a **network-isolated sandbox (e.g. V8 isolate)** with explicit RPC bindings only; restrict CI agents to isolated branches (`claude/issue/<id>`)	Running unreviewed `npx` MCP servers or `eval()`ing generated code on the host

Compliance scoping:

- Data retention, zero-data-retention and BAA eligibility vary **by feature and by platform**: confirm each feature you plan to use.
- "No training on API data" is not the same as zero retention.
- Audit-log every tool call and decision (request, response, context, outcome) with a shared trace ID.

> **COMMON TRAP**
> "All our users are trusted employees, so injection isn't a risk." The attacker is whoever wrote the web page, email or document the agent reads. Prompt rules and bigger models are probabilistic; capability removal and pre-action authorization are not.

19. Failure modes and find the defect

Symptom / scenario	Root cause → correct fix	Tempting wrong answer
"tool_result references unknown tool_use_id"	Result ID ≠ call ID, or result not in the next user turn → echo `block.id`; results first in the next message	Rewriting the tool description
400 after "trimming" history to save tokens with thinking on	Thinking block edited/dropped, signature mismatch → echo turns unchanged; compact older turns	Raising `max_tokens`
Validation error only on retries after a dropped stream	Half-streamed turn saved → commit at `message_stop`; discard partial turns	Changing the tool schema
Correct tool on turns 1–4, wrong tool from turn 5	Context crowded by tool output (**context confusion**) → prune, edit, compact, or offload to virtual FS; measure tokens	Clarifying descriptions or raising `max_tokens`
Wrong tool from **turn 1** after connecting 5 MCP servers, or 25 sequential round-trips on a loop	**Upfront schema bloat** (~50k tokens) and ping-pong latency → use progressive disclosure/scoped `tools[]` and **CodeMode** (TypeScript API in isolated V8 sandbox)	Raising `max_tokens` or setting `tool_choice: "any"`

Symptom / scenario	Root cause → correct fix	Tempting wrong answer
Agent repeats a hallucinated table/ID after compaction, or approves its own broken code in review	**Context poisoning** (survived compaction) and **self-confirmation bias** → hard-reset `messages: []` for poisoning; run Reviewer in a **separate `messages` array**	Running compaction again or prompting "be more critical" in the same thread
Support agent invents policies or breaks on unscripted tickets after a 40-page `if/else` rewrite	Prompt altitude outside the **Goldilocks zone** (too vague vs. brittle state machine) → define clear scope, 4-step reasoning framework, and explicit escalation triggers	Setting `temperature=0` (400s on 4.7+)
Model says "no orders found" during a DB outage	Failure returned as `""` → `tool_result` with `is_error: true`	"Never say no orders found" in the prompt
Answer silently missing; logs show HTTP 200	`stop_reason: "refusal"` or `max_tokens` not checked → branch on `stop_reason`	Retrying on HTTP status only
Same refusal repeats every turn	Refused turn kept in history → remove/rephrase it or reset	Exponential backoff
Nightly job hits 429 every night	Sync loop over items → Message Batches + `custom_id`	Smaller chunks
Batch results attached to the wrong tickets	Matched by position → match on `custom_id`	Sorting results by timestamp
429 storms despite "retries"	Own loop stacked on SDK retries → one retry owner, honor `retry-after`, jitter	More attempts
400 after model upgrade on `temperature` / `budget_tokens`	Unsupported on 4.7+ → omit sampling params; adaptive thinking + effort	Lowering the values
0% cache hit rate	Dynamic text before the breakpoint, or tool_choice/effort/system style changing per call → static prefix first, stable settings	1-hour TTL
Parser breaks after an upstream release; can't roll back	Moving alias → pin the full ID, keep prior pin, eval-gate promotion	Adding parser retries
EU-only residency requirement on first-party API	`inference_geo` has no EU value → Bedrock/Google Cloud EU routing	`inference_geo: "us"` + encryption
Summarizer agent forwards a thread externally	Injection via fetched email + unneeded send tool → treat content as data; remove the tool	Stricter user-input regex
Code reviewer floods PRs with style nits; developers start ignoring real bugs	Vague criteria → categorical report/skip rules + contrast few-shot examples; disable the noisiest category while fixing it	"Only report high-confidence findings"
Extractor returns plausible but invented `po_number` values	Field is `required` but often absent → nullable/optional + "null if not stated"	More retries
Retry loop never fixes `total`; the value is only in an attachment that wasn't sent	Information absent → stop retrying; flag, fetch the source or route to review	Raising the retry cap
Support agent forgets the refund amount and deadline after 30 turns	Rolling summary dropped exact facts → persistent case-facts block; trim tool outputs	Switching to a bigger model
97% overall accuracy, but handwritten receipts are often wrong	Aggregate hides segments → accuracy per doc type × field; stratified sampling; calibrated thresholds	Removing human review
Pre-merge check moved to Message Batches; developers wait hours	Batch has no latency SLA → sync API for blocking checks, batch for overnight	Polling the batch more often

Find the defect: a migrated tool-use turn. This code ran on an older model. After moving to a Claude 4.7+ pinned ID it fails or misbehaves. Find at least three defects.

```
resp = client.messages.create(
    model=NEW_PINNED_ID, max_tokens=4096,
    temperature=0,                                          #
(a)
    thinking={"type": "enabled", "budget_tokens": 8000},    #
(b)
    tools=TOOLS, messages=messages,
)
messages.append({"role": "assistant",                       #
(c)
                 "content": [b for b in resp.content if b.type ≠
"thinking"]})
call = next(b for b in resp.content if b.type == "tool_use")
try:
    out = run_tool(call.name, call.input)
except Exception:
    out = ""                                                #
(d)
messages.append({"role": "user", "content": [
    {"type": "text", "text": "Here is the result:"},        #
(e)
    {"type": "tool_result", "tool_use_id": call.id, "content":
out},
]})
```

Fixes:

- **(a)** Remove `temperature`: non-default sampling values return 400 on Claude 4.7+. Steer with the prompt instead.
- **(b)** Replace the manual budget with `thinking={"type": "adaptive"}` and `output_config={"effort": "high"}`; `budget_tokens` returns 400 on 4.7+.
- **(c)** Append `resp.content` unchanged: filtering thinking blocks breaks the signature check on the next call.
- **(d)** Return failures as a `tool_result` with `is_error: true` and a short message, never `""`.
- **(e)** Put `tool_result` blocks first; text after.
- **Bonus:** check `resp.stop_reason == "tool_use"` before looking for a call, and loop over *all* `tool_use` blocks in case of parallel calls.

20. Glossary

Term	Definition
Content block	Typed unit of a message: `text`, `image`, `document`, `tool_use`, `tool_result`, `thinking`, `redacted_thinking`.

Term	Definition
`stop_reason`	Why generation stopped: `end_turn`, `tool_use`, `max_tokens`, `stop_sequence`, `pause_turn`, `refusal`, `model_context_window_exceeded`.
`tool_use_id`	Field on a `tool_result` that must equal the `id` of the `tool_use` block it answers.
`is_error`	Flag on a `tool_result` telling Claude the tool failed, so it can retry, ask, or stop instead of trusting empty data.
`tool_choice`	`auto` (default), `any`, `tool`, `none`; holds `disable_parallel_tool_use`.
Strict tool use	`strict: true` on a tool: inputs are guaranteed to match its `input_schema`.
`output_config.format`	JSON outputs: constrains the response text to a JSON schema.
Adaptive thinking	`thinking: {type: "adaptive"}`; the model sizes its own reasoning, steered by `output_config.effort`.
Effort	Per-request reasoning depth: `low` → `max` (set varies by model). Higher = slower and costlier.
Signature	Cryptographic field on thinking blocks; any edit makes the next request fail.
`message_stop`	Final streaming event; the only safe point to commit a streamed turn.
`input_json_delta`	Streaming fragment of tool input JSON; parse only at `content_block_stop`.
Cache breakpoint	`cache_control` marker ending a reusable prefix; up to 4 per request; TTL 5m (default) or 1h.
Compaction	Server-side summarization of older context into a compaction block so long-running agents keep going.
`custom_id`	Your unique key per batched request; the only reliable way to match out-of-order results.
`inference_geo`	First-party residency parameter: `global` or `us` (Claude 4.6+).
Pinned model ID	An ID whose underlying model never changes; the unit of release and rollback.
Indirect prompt injection	Instructions hidden in content the app reads (pages, emails, tool output) rather than typed by the user.

Term	Definition
Goldilocks zone (altitude)	System prompt sweet spot between brittle state-machine rules (too specific) and platitudes (too vague): clear scope + reasoning framework + explicit boundaries.
Context poisoning / confusion / clash	Long-context failure modes: **poisoning** (hallucination in history compounds), **confusion** (bulky/irrelevant context steers output), **clash** (conflicting rules).
Deep Agent & CodeMode	**Deep Agents** combine `TodoWrite` planning, isolated subagents, and virtual-FS state offloading; **CodeMode** (Cloudflare) runs generated TypeScript against MCP tools in a V8 sandbox in 1 turn; Anthropic's equivalent is **programmatic tool calling** (Python in the code execution container).
Few-shot (multishot) examples	A few varied, tagged input → output examples that teach format and judgment for ambiguous cases.
Explicit criteria	Categorical report/skip rules (with examples per severity) that replace vague "be careful / be confident" wording.
Nullable field	Schema type such as `["string", "null"]` so "not stated" is a valid answer instead of a fabricated value.
Retry with error feedback	Re-asking with the document, the failed output and the specific validation errors; useless when the data is absent.
`calculated_total` / `stated_total`	Paired fields (computed vs. printed) whose mismatch your code flags as a semantic error.
`detected_pattern`	Field naming the construct behind a finding, so dismissals can be analyzed by pattern.
Case facts block	Structured, code-maintained facts (amounts, dates, IDs, statuses) re-injected every request outside summarized history.
Lost in the middle	Long-input effect where content in the middle is used less reliably than the start and end.
Independent reviewer	A separate API call with a fresh context (spec + artifact only) instead of self-review.
Multi-pass review	Per-file local passes followed by a cross-file integration pass.
Stratified sampling	Sampling auto-approved items within each segment (doc type, field) to keep measuring error rates.
Calibrated confidence	Confidence thresholds set from measured accuracy on a labeled validation set, per field or segment.