FIELD GUIDE DECK · LAST REFRESH 14 OCTOBER 2026

Claude Certified Architect.

— Foundations

The CCAR-F exam one idea per slide: five domains, the wrong answers it tempts you with, and the patterns it rewards.



Unofficial · not affiliated with Anthropic

dnacenta.github.io/claude-certified-architect

THE EXAM AT A GLANCE

Sixty questions, two hours, 720 to pass.

60

multiple-choice and multiple-response questions, all scenario-based

120min

proctored and closed-book

720/1000

passing score

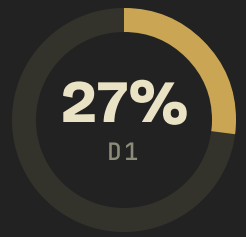
4of 6

scenarios, randomly selected for each exam

No penalty for guessing. Delivered by Pearson VUE, online through OnVUE or at a test center, and registered through the Anthropic Partner Academy.

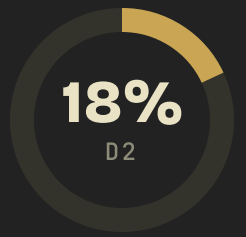
FIVE DOMAINS

Agentic architecture carries the most weight.



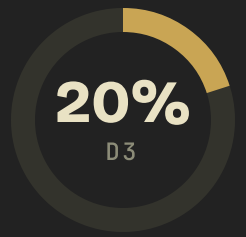
Agentic Architecture & Orchestration

Agentic loops, `stop_reason`, multi-agent orchestration, hooks and session management.



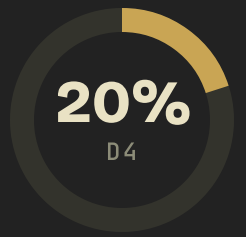
Tool Design & MCP Integration

Tool design, structured error responses, tools per agent and MCP configuration.



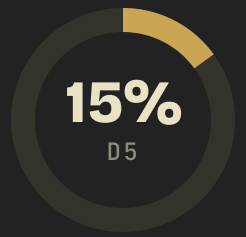
Claude Code Configuration & Workflows

CLAUDE.md hierarchy, plan mode vs direct execution, and Claude Code in CI/CD.



Prompt Engineering & Structured Output

Explicit criteria, few-shot, structured output, validation-retry and batch.



Context Management & Reliability

Context management, escalation, error propagation and provenance.

HOW TO USE THIS DECK

Every domain follows the same five steps.

STEP 1

Divider

The domain's weight and the task statements it covers.

STEP 2

Concepts

One idea per slide, with a diagram where the mechanism matters.

STEP 3

Traps

The wrong answers that look right, set against the pattern the exam rewards.

STEP 4

Practice

One question per slide. Commit to an answer first: the answer and the why are on the next slide.

STEP 5

Cheat sheet

The whole domain on one page, plus what it tests most.

Before the domains: the six scenarios, the ten anti-patterns, the decision frameworks and a four-week study plan.

THE 6 EXAM SCENARIOS

Each exam draws 4 of these 6 scenarios at random.

SCENARIO 1

Customer Support Resolution Agent

Agent SDK, MCP tools and escalation logic.

SCENARIO 2

Code Generation with Claude Code

CLAUDE.md, plan mode and slash commands.

SCENARIO 3

Multi-Agent Research System

Coordinator-subagent patterns and error handling.

SCENARIO 4

Developer Productivity with Claude

Built-in tools, MCP servers and codebase exploration.

SCENARIO 5

Claude Code for CI/CD

Structured output, the batch API and multi-pass review.

SCENARIO 6

Structured Data Extraction

JSON schemas, `tool_use` and validation-retry.

You can't know which four you'll get, so prepare all six.

These show up as wrong answers on the exam.

ANTI-PATTERN 1

Parsing text to end the loop

Natural language is not a termination signal. Check `stop_reason` instead.

ANTI-PATTERN 2

An iteration cap as the stop

An arbitrary cap is wrong as the primary stopping mechanism. `stop_reason` drives the loop.

ANTI-PATTERN 3

Prompts for critical rules

Prompt-based enforcement is best-effort. Financial and safety rules belong in programmatic hooks.

ANTI-PATTERN 4

Self-reported confidence

Don't escalate on the model's own confidence score. Escalate on an explicit customer request; otherwise resolve within capability.

ANTI-PATTERN 5

Sentiment-based escalation

Sentiment ≠ complexity. An upset tone is not a reason to escalate an issue that is within capability.

Hidden errors, crowded agents and flattering metrics.

ANTI-PATTERN 6

Generic error messages

They hide diagnostic context. Return structure: the error category, whether it's retryable, and partial results.

ANTI-PATTERN 7

Silently suppressed errors

Returning empty results as success hides the failure. Error handling is never generic or silent.

ANTI-PATTERN 8

Too many tools per agent

18 tools where 4–5 are recommended. Past ~10 tools, tool search with `defer_loading` changes the picture (Domain 2).

ANTI-PATTERN 9

Same-session self-review

The session retains its own reasoning context, so its review inherits that bias. Review from a fresh session.

ANTI-PATTERN 10

One aggregate accuracy number

A good overall metric can mask a document type that keeps failing. Measure per document type.

Match the mechanism to the stakes and the scope.

ENFORCEMENT

- **Programmatic hooks** — financial/safety
- **Prompt-based** — best-effort

ESCALATION

- **Immediate** — explicit customer request
- **Resolve first** — within capability

ERROR HANDLING

- **Structured context** — category, retry, partial
- Never generic or silent

EXECUTION MODE

- **Plan mode** — multi-file, architectural
- **Direct** — single-file, obvious

CODE REVIEW

- **Multi-pass** — large PRs
- **Single-pass** — small, focused

API TYPE

- **Synchronous** — blocking workflows
- **Batch** — latency-tolerant

Context never travels or survives on its own.

CONTEXT PASSING

- **Always explicit** in subagent prompts
- Never rely on auto-inheritance

AGENT HARNESS

- **Tool Runner / Agent SDK** — you host
- **Managed Agents** — Anthropic hosts the loop; the sandbox is Anthropic's cloud or self-hosted

MANY-AGENT ORCHESTRATION

- **Subagents / forks** — a few delegated tasks per turn
- **Dynamic workflows** — a script the runtime executes for dozens to hundreds of agents

CONTEXT REDUCTION

- **Clear** (context editing) — verbose tool results
- **Compact** — long conversations that need their narrative; prefer on-demand compaction

STATE THAT MUST PERSIST

- **Memory tool / files**
- Never rely on the conversation surviving compaction

CONVERSATION HISTORY

- **Append-only** — freeze `system`/`tools`, changes via `role: "system"` messages
- Never edit or snip earlier turns — it invalidates thinking blocks

Defer big tool sets; sweep effort from the default.

TOOL EXPOSURE

- **Load upfront** — under ~10 tools
- **Tool search +** `defer_loading` — 10+ tools or multi-server MCP

TOOL_CHOICE

- **"any"** — guarantee tool call
- **Forced** — specific tool first
- On models where both return 400: `auto` + instruction + `strict: true`, or `output_config.format`

THINKING

- **Adaptive** — `{"type": "adaptive"}`
- `budget_tokens` is removed — it returns 400

EFFORT

- **Start at the model's default and sweep**; `xhigh` for the hardest coding/agentic work
- `low` / `medium` — subagents, mechanical and routine tasks

4-WEEK STUDY PLAN

Build the core, apply it, harden it, then drill.

Assumes daily study, 1.5–2 hours per day.

WEEK 1

Core Architecture

- Agentic loops, `stop_reason`, multi-agent orchestration, hooks, session management, tool design, MCP config
- Daily: 1 domain section + practice questions

WEEK 2

Applied Skills

- CLAUDE.md hierarchy, plan mode, CI/CD, Batches API, explicit criteria, few-shot, validation-retry, structured output
- Daily: 1 domain section + build a small project using each concept

WEEK 3

Reliability + Hands-On

- Context management, escalation, error propagation, provenance, hands-on exercises across all domains
- Daily: review weak areas + build an end-to-end scenario

WEEK 4

Exam Prep

- Practice exams, anti-pattern drills, scenario walkthroughs, timed question sets
- Daily: 1 full practice exam + targeted review

THE CERTIFICATION FAMILY

Four credentials, one program.

CCAO-F

Associate — Foundations

Business / productivity users (non-developer)

$99

CCAR-F

Architect — Foundations

Solution architects

$125

CCAR-P

Architect — Professional

Senior architects owning the full solution lifecycle

$175

CCDV-F

Developer — Foundations

Engineers shipping Claude apps, agents, and workflows

$125

MODELS IN THE CODE SAMPLES

The current lineup

Code samples use `claude-opus-5-5` — Anthropic's current default for most workloads (Claude Opus 5.5 launched 2026-09-22). Samples that rely on forced `tool_choice` use `claude-haiku-5-5`, the only current model that still accepts it (Fable 5.1, Opus 5.5, and Sonnet 5.5 return 400).

MODEL	ID	CONTEXT	MAX OUTPUT	PRICE (IN / OUT PER MTOK)
Claude Fable 5.1	`claude-fable-5-1`	1M	128k	$10 / $50 (cache reads $0.25)
Claude Opus 5.5	`claude-opus-5-5`	1M	128k	$4 / $20 (cache reads $0.20)
Claude Sonnet 5.5	`claude-sonnet-5-5`	1M	128k	$2 / $10 (cache reads $0.10)
Claude Haiku 5.5	`claude-haiku-5-5`	1M	128k	$0.10 / $0.50 for prompts up to 100k tokens; $0.50 / $2.50 above



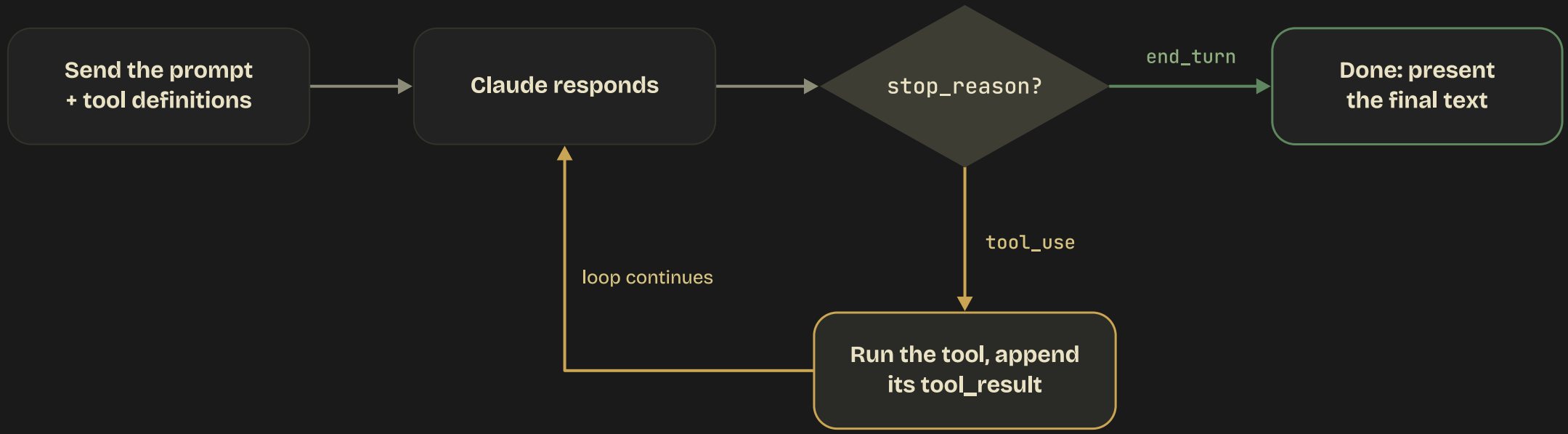
DOMAIN 1 OF 5 · THE HEAVIEST

Agentic Architecture & Orchestration

How agentic systems are built, how loops run, how agents coordinate, and how enforcement works. Seven task statements:

- 1.1 The agentic loop
- 1.2 Multi-agent orchestration
- 1.3 Subagents & context passing
- 1.4 Workflow enforcement & handoff
- 1.5 Hooks
- 1.6 Task decomposition
- 1.7 Session management

The loop ends on `stop_reason`, never on text.



THE OTHER FIVE VALUES

`stop_sequence`	`pause_turn`	`max_tokens`	`refusal`	`model_context_window_exceeded`
Ends. A stop string matched.	Send it back. A server tool hit its cap.	Hit the limit. May need a continuation.	Ends. Read stop_details.	Valid but truncated. Trim or continue.

Three wrong answers that look right.

THE RIGHT ANSWER

Branch on `stop_reason` every turn.

- `tool_use` → run the tool, return its tool_result, continue.
- `end_turn` → present the final text.
- A safety cap can exist, but `stop_reason` drives the loop.

Parsing natural language

"If Claude says *I'm done*, exit the loop."

An iteration cap as the main stop

"Run max 5 loops, then stop." Fine as a safety net, wrong as the mechanism.

Treating text as "done"

"If the response has text, it's finished." Text and tool_use blocks can arrive together.

Claude decides the next step; `stop_reason` tells your code how to act.

VALUE	MEANING	ACTION
`tool_use`	Claude wants to call a tool	Execute it, return the result, continue the loop
`end_turn`	Claude has finished reasoning	Present the final text response
`pause_turn`	A server-side tool loop hit its internal limit	Send the response back to continue
`max_tokens`	Response hit the `max_tokens` limit	May need to request continuation
`stop_sequence`	Output matched a configured stop string	Loop ends; inspect which sequence hit
`refusal`	Claude declined on safety grounds	Loop ends; inspect `stop_details`, rephrase or route

A `refusal` arrives as HTTP 200, not an exception: branch on `stop_reason` before reading `content`.

A `tool_result` must echo the exact `tool_use_id`.

JSON

```
// Claude's response, stop_reason "tool_use"
{ "type": "tool_use", "id": "toolu_01A", "name": "get_weather",
  "input": { "location": "San Francisco, CA" } }

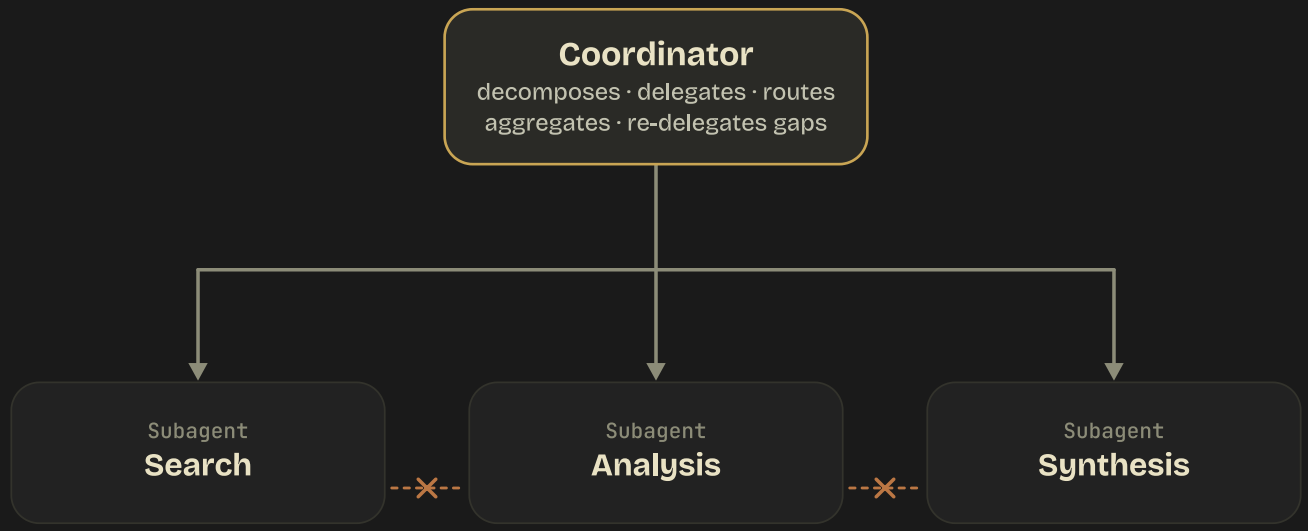
// Your next request, role "user"
{ "type": "tool_result", "tool_use_id": "toolu_01A",
  "content": "15 degrees celsius, partly cloudy" }

// When the tool fails
{ "type": "tool_result", "tool_use_id": "toolu_01A",
  "is_error": true, "content": "API timeout after 30 seconds" }
```

1. One response can hold text and several `tool_use` blocks.
2. Several calls → several `tool_result` blocks in one message, each matching its id.
3. `is_error: true` tells Claude the tool failed so it can reason about recovery.

One coordinator routes everything; subagents never talk to each other.

- The coordinator **decomposes** the task and **delegates** subtasks to specialized subagents.
- It **routes** all inter-subagent communication. Subagents never talk directly.
- It **aggregates** results, handles subagent failures and re-delegates when gaps exist.
- Not every query needs every subagent. Invoke only the relevant ones.



Subagents never talk to each other: every message goes through the coordinator.

Decompose broadly; a narrow split misses whole subtopics.

THE RIGHT ANSWER

Decompose broadly, then check coverage.

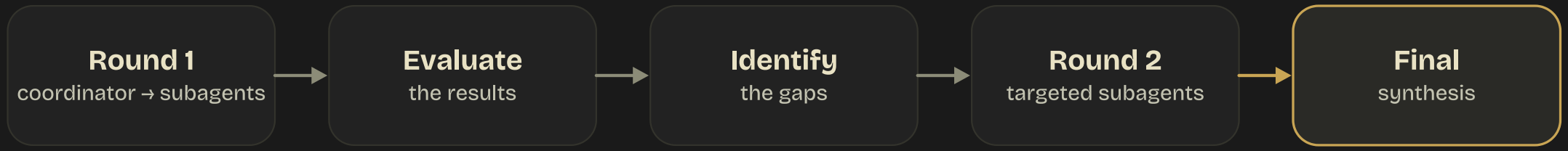
- Start with a broad initial decomposition of the whole topic.
- Add explicit coverage checks so no part of the topic is missed.
- Assess what each query actually requires and invoke only those subagents.

Overly narrow decomposition
"Creative industries" split into visual arts only, missing music, writing, film and design.

Every query through the full pipeline
Web search + document analysis + synthesis when a single focused search would suffice.

Subagents talking directly
Results passed from subagent to subagent. The coordinator routes all of it.

The coordinator evaluates the synthesis and re-delegates the gaps.



Coverage gaps found in round 1 get a second, narrower pass before synthesis.

- Round 1: the coordinator delegates to subagents and collects their results.
- It evaluates the results and identifies gaps in coverage or quality.
- Round 2: targeted subtasks go out for the gaps only.
- Then the coordinator produces the final synthesis.

A subagent knows only what its prompt tells it.

BAD

“Analyze the customer's issue”

The subagent has no idea who the customer is or what the issue is. It inherits none of the coordinator's history.

GOOD

Everything it needs, explicitly

“Customer #12345 reported that order #67890 ($149.99) arrived damaged and wants a full refund. Analyze the order history and return policy applicability.”

Pass findings as structured data (claim, evidence, source, date) so the receiver can judge source quality.

Parallel subagents means several `Agent` calls in one response.

JSON

```
{ "content": [
  { "type": "tool_use", "name": "Agent",
    "input": { "prompt": "Search for recent papers on...",
               "subagent_type": "search" } },
  { "type": "tool_use", "name": "Agent",
    "input": { "prompt": "Analyze the document at...",
               "subagent_type": "analysis" } }
] }
```

1. Both calls in a **single** coordinator response run in parallel. Separate turns run sequentially.
2. The coordinator needs `Agent` in its allowed tools. `Agent(worker, researcher)` narrows the types.
3. Older material, including the exam guide, calls it the **Task** tool. Same mechanism.

A fork inherits everything; a fresh subagent inherits nothing.

Fresh subagent

- History not inherited: pass everything explicitly.
- Its own system prompt and tools.
- Cold prompt cache.
- Use for bounded, well-specified tasks and noisy exploration.

Fork

- Inherits the entire conversation, system prompt, tools and model.
- Shares the parent's prompt cache, so it's cheaper.
- Only its final result returns. Forks can't spawn sub-forks.
- Use for divergent branches off a shared baseline.

Money or safety at stake? Enforce it in code, not in the prompt.

PROBABILISTIC

Prompt-based

- Instructions like *Always verify the customer's identity before any refund*.
- Works most of the time, with a non-zero failure rate.
- Fine only where best-effort is acceptable.

DETERMINISTIC

Programmatic

- Hooks and prerequisites, enforced every time.
- Required when there are financial or safety consequences.
- A PreToolUse hook on `process_refund` blocks with exit code 2 until `get_customer` returns a verified ID.

A human handoff needs a structured summary, not the transcript.

- The human agent typically has no access to the full conversation transcript.
- Lead with the facts: customer, issue, order details and root cause.
- List the actions already taken, such as *verified order, confirmed damage via photos*.
- Close with the recommended action and amount, such as *full refund + replacement*.

PreToolUse gates the call; PostToolUse cleans up the result.

BEFORE THE TOOL RUNS

PreToolUse

Can block the call with exit code 2 or a `deny` decision. Use it for prerequisites, like a verified identity before a refund.

AFTER IT SUCCEEDS

PostToolUse

Intercepts results before Claude processes them: normalize timestamps, dates and status codes, scrub sensitive data, add metadata.

AFTER IT FAILS

PostToolUseFailure

Fires instead of PostToolUse when the tool itself errored. Handle retries and diagnostics here.

Hooks live in user, project, local or managed settings, and can be scoped to a skill or a subagent.

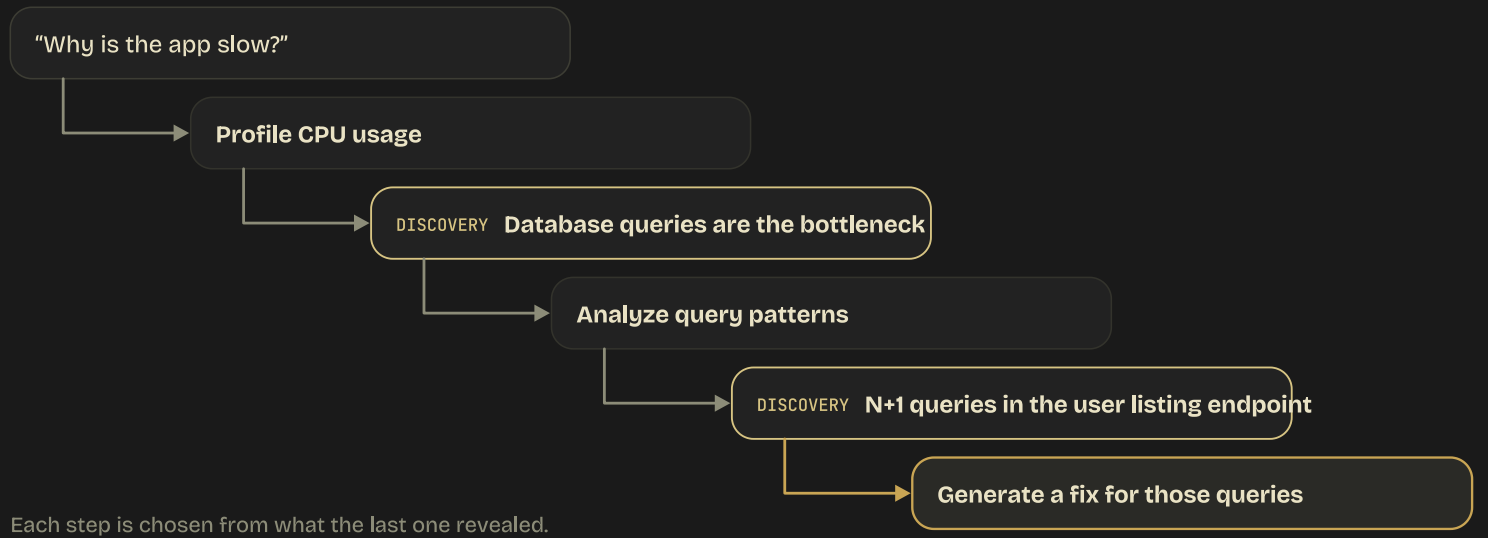
A PreToolUse hook blocks the call with a deny decision.

JSON

```
{
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "deny",
    "permissionDecisionReason": "Customer identity not yet verified",
    "additionalContext": "Call get_customer first to verify identity"
  }
}
```

1. Decisions are `allow`, `deny`, `ask` and `defer`. When hooks disagree, `deny` wins.
2. A `command` hook passes with exit 0 and blocks with exit 2.
3. A hook's `allow` never overrides a matching deny permission rule.
4. By default a hook that fails to start or times out lets the action go ahead. Make policy gates fail closed.

Known steps get a chain; open-ended work gets dynamic decomposition.



Each step is chosen from what the last one revealed.

- **Prompt chaining**: steps fixed upfront, each step's output feeds the next.
- Fits predictable, multi-aspect tasks: per-file analysis, cross-file pass, final report.
- **Dynamic decomposition**: each step's discoveries generate the next subtask.
- Fits open-ended investigation, such as *Why is the app slow?*

One pass over a large PR fails three ways. Split it in two.

THE RIGHT ANSWER

A per-file pass, then a cross-file pass.

- Pass 1: review each file independently, at consistent depth, for bugs, style and types.
- Pass 2: a separate prompt for data flow, interface mismatches and cross-file consistency.

Attention dilution
A single pass focuses on early files and skimps on later ones.

Contradictory findings
Standards vary from file to file, so findings contradict each other.

Missed integration issues
Reviewing one file at a time can't see cross-file problems.

Resume while tool results are valid; start fresh once they're stale.

--RESUME

Resume

- Continues a named prior conversation with full context.
- Ongoing investigation or iterative work, tool results still valid.

--FORK-SESSION

Fork

- Resumes under a **new** session ID; the original stays untouched.
- Try two refactoring strategies from the same starting point.

FRESH + SUMMARY

Start fresh

- Tool results are stale: files changed, services restarted.
- Context has drifted too far from the current task.
- Pass a structured summary of prior findings instead.

Resuming after edits? Say what changed, such as *I've updated auth.py to use JWT tokens*. It won't notice on its own.

An agentic loop is processing customer requests. After Claude responds, what should the system check to determine the next action?

A. Whether the response contains text content

B. The `stop_reason` field in the response

C. The length of the response

D. Whether Claude expressed confidence in its answer

B

The `stop_reason` field in the response

A Whether the response contains text content

C The length of the response

D Whether Claude expressed confidence in its answer

WHY

The `stop_reason` field determines whether to continue the loop (`tool_use`), end it (`end_turn`), or handle edge cases (`pause_turn`, `max_tokens`).

A coordinator agent needs to pass customer information to a subagent. What is the correct approach?

A. The subagent will automatically inherit the coordinator's conversation context

B. Include all relevant customer information explicitly in the subagent's prompt

C. Store the information in a shared database that both agents access

D. Use environment variables to pass the context

B

Include all relevant customer information explicitly in the subagent's prompt

A The subagent will automatically inherit the coordinator's conversation context

C Store the information in a shared database that both agents access

D Use environment variables to pass the context

WHY

Subagents have isolated context and do not inherit the coordinator's conversation. Context must be explicitly passed in the prompt.

A financial services application needs to ensure identity verification occurs before any refund processing. Which enforcement mechanism should be used?

A. Add instructions to the system prompt requiring verification first

B. Use a PreToolUse hook that blocks `process_refund` until verification is complete

C. Train the model on examples where verification comes first

D. Set a high temperature to encourage creative verification approaches

B

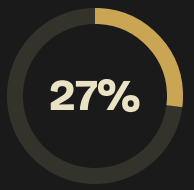
Use a PreToolUse hook that blocks `process_refund` until verification is complete

- **A** Add instructions to the system prompt requiring verification first
- **C** Train the model on examples where verification comes first
- **D** Set a high temperature to encourage creative verification approaches

WHY

Financial operations require deterministic, programmatic enforcement. Prompt-based instructions have a non-zero failure rate and are inappropriate for financial/safety-critical workflows.

Agentic Architecture & Orchestration



1.1 The loop

Branch on `stop_reason`. tool_use → run, return tool_result, continue. end_turn → done.

1.2 Orchestration

Hub-and-spoke. The coordinator decomposes, delegates, routes every message and aggregates. Subagents never talk directly.

1.3 Subagents

Isolated context: they inherit nothing. Put everything they need in the prompt.

1.4 Enforcement

Money or safety → programmatic (hooks, prerequisites), deterministic. Prompts are probabilistic.

1.5 Hooks

A PreToolUse hook can block a tool call (exit code 2) until a prerequisite is met.

1.6 Decomposition

Known steps → prompt chaining. Open-ended → dynamic decomposition. Watch for splits that are too narrow.

1.7 Sessions

`--resume` while tool results are valid. Stale? Start fresh with a summary. `--fork-session` to branch.

★ Most tested

Programmatic vs prompt-based enforcement. If a prompt can fail and the cost is real, use a hook.



DOMAIN 2 OF 5 · TOOLS & MCP

Tool Design & MCP Integration

How to design tool interfaces LLMs use reliably, handle tool errors, distribute tools across agents, and configure MCP servers. Six task statements:

- 2.1 Tool interface design
- 2.2 Structured error responses
- 2.3 Tool distribution & tool_choice
- 2.4 MCP server configuration
- 2.5 Built-in tools
- 2.6 Anthropic-defined & server tools

Claude picks a tool by its description, so make it answer five questions.

The description is the selection criteria, not documentation. “Search for things” leads to unreliable selection.

Purpose

What the tool does and when to use it.

Input format

What the input should look like, with examples.

Output format

What the tool returns.

Edge cases

What happens with invalid inputs or no results.

Differentiation

When to use this tool instead of a similar one.

A good one, `search_knowledge_base`, says what it searches, when to use it, what to send, and what comes back.

Strict mode guarantees schema-valid inputs, once the schema is closed.

JSON

```
{
  "name": "extract_invoice",
  "strict": true,
  "input_schema": {
    "type": "object",
    "properties": {
      "vendor_name": { "type": "string" },
      "total_amount": { "type": "number" }
    },
    "required": ["vendor_name", "total_amount"],
    "additionalProperties": false
  }
}
```

1. `strict: true` guarantees Claude's tool inputs conform exactly to the schema.
2. `additionalProperties: false` must be set.
3. Every property must be listed in `required`.
4. Only a subset of JSON Schema is supported (Domain 4).

Vague or overlapping tools get misrouted.

THE RIGHT ANSWER

Make every tool distinct.

- A specific name, plus what it does, what it takes and what it returns.
- Say when to use **this** tool versus similar ones.
- Two tools overlap? Rename them to differentiate purpose, or merge them into one.

A one-line description

`search`: “Search for things.” Nothing to select on, so selection is unreliable.

Near-identical descriptions

`analyze_content` (“Analyze content for insights”) vs `analyze_document` (“Analyze a document for insights”). Claude can't reliably tell them apart.

The error category decides the move: retry, fix, explain, or escalate.

CATEGORY	RETRYABLE?	EXAMPLES	AGENT ACTION
Transient	Yes	Timeouts, service unavailability, rate limits	Retry with backoff
Validation	Not without input changes	Invalid email format, missing required field	Examine the error, fix the input, retry
Business	No	Refund exceeds policy limit, account frozen	Explain the policy to the user or escalate
Permission	No	Insufficient access, expired token	Escalate to a human

Subagents recover from transient errors locally and propagate them only if recovery fails.

Errors that hide, lie, or give up are the wrong answers.

THE RIGHT ANSWER

Return a structured error.

- Set `isError`, an `errorCategory` and `isRetryable`.
- Describe the cause and the `attempted_action`, so the agent can recover intelligently.
- Use partial results or try alternatives instead of ending the workflow.

Generic messages

"Operation failed" or "search unavailable" hides the cause and what was attempted.

Silent suppression

Returning `{ "results": [] }` when search is down. The agent confidently says "I found nothing."

Terminating on a single failure

Killing a whole multi-step workflow because one step failed.

An empty result is information; a timeout is an error.

[] WITH STATUS 200

Valid query, no matches

- The search ran and found nothing.
- That is information, so report it.
- Say: “No results found for that query.”

TIMEOUT ERROR

The query didn't execute

- The search didn't work, so there is no result to report.
- That is an error, so handle it as one.
- Say: “I wasn't able to search right now, let me try again.”

Check the response status before answering. The two must be handled differently.

Give each agent four or five tools, and split the rest across subagents.

Eighteen tools on one agent degrades selection: wrong picks, unnecessary calls, trouble reasoning about which to use.

COORDINATOR

`Agent`, `AskUser`

SEARCH SUBAGENT

`search_web`, `search_docs`, `search_db`

ANALYSIS SUBAGENT

`analyze_text`, `extract_entities`, `summarize`

ACTION SUBAGENT

`send_email`, `create_ticket`, `update_record`

The principle behind the number is one agent, one focused job.

Tool search keeps a few tools in view while the catalog behind them grows.

- Tools marked `defer_loading: true` are still sent every request but stay out of the context window.
- Claude calls the search tool, and matches expand into full definitions inline, so prompt caching survives.
- Never defer everything: at least one tool, normally the search tool, must stay loaded, or it's a 400.
- Leave the 3–5 most-used tools non-deferred so Claude calls them without a search round-trip.

The exam's “4–5 tools” means tools in front of the model at decision time. Tool search keeps it there.

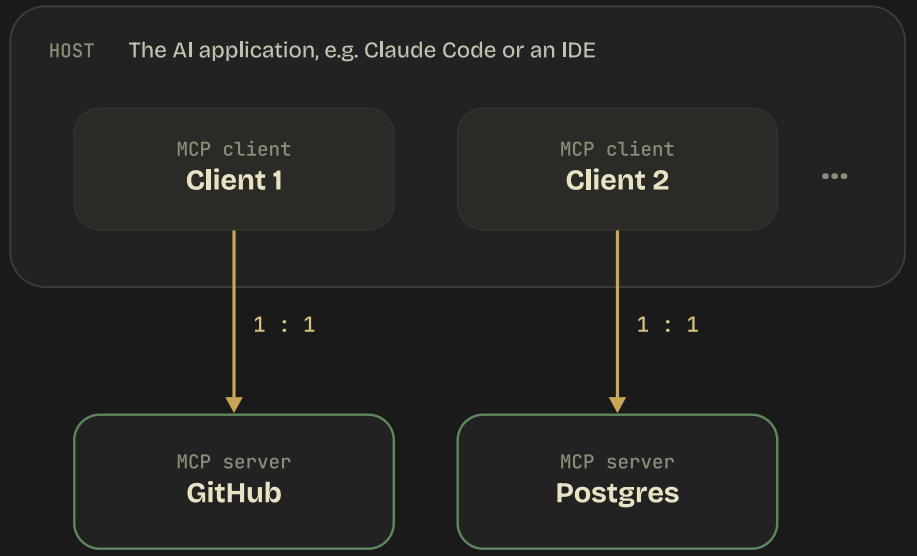
`tool_choice` decides whether Claude may, must, or can't call a tool.

VALUE	BEHAVIOR	USE IT FOR
`"auto"` **(default)**	May call a tool or answer in text	General conversation with optional tool use
`"any"`	Must call a tool, but chooses which	Guaranteed structured output, e.g. picking among invoice, receipt and contract schemas
`{"type": "tool", "name": …}`	Must call that exact tool	Forcing a step first, e.g. `extract_metadata` before enrichment
`"none"`	Text only; tool definitions stay in context	A prose turn, e.g. a final summarization pass

Some current models reject forced `any` and `tool` with a 400. Use `auto` plus an instruction naming the tool and `strict: true`, or structured outputs.

A host runs one MCP client per server, and servers supply the context.

- **Host**: the AI app, like Claude Code or an IDE, that coordinates MCP clients.
- **Client**: lives in the host, with a 1:1 connection to one server.
- **Server**: provides tools, resources and prompts.
- **Resources** are read-only catalogs Claude can browse instead of blindly searching.



Each client keeps one connection to one server.

Servers provide tools, resources and prompts.

MCP is an open standard for connecting AI models to data and tools, over JSON-RPC 2.0.

Use stdio for local servers and Streamable HTTP for remote ones.

TRANSPORT	TYPE IN CONFIG	WHEN TO USE
stdio	`"stdio"`, implied by `command`	Local processes. No network overhead; most common for local tools
Streamable HTTP	`"http"`	Remote and cloud-hosted servers; supports OAuth
SSE (legacy)	`"sse"`	Services that still expose only an SSE endpoint
WebSocket	`"ws"`	Remote servers that push events unprompted; header-only auth, no OAuth

Gotcha: an entry with a `url` but no `type` is read as stdio and skipped. Always set `type` on remote servers.

The scope decides where a server is stored and who gets it.

SCOPE	LOADS IN	SHARED?	STORED IN
Local (default)	Current project only	No	`~/.claude.json`, keyed by project path
Project	Current project only	Yes, via version control	`.mcp.json` in the project root
User	All your projects	No	`~/.claude.json`
Managed (admin)	Every session in the org	Yes, pushed by IT	`managedMcpServers` in managed settings

Naming trap: MCP local scope lives in `~/.claude.json`, not `.claude/settings.local.json`. The two are unrelated.

`.mcp.json` is committed, so secrets come in through `${VAR}` expansion.

JSON

```
{
  "mcpServers": {
    "shared-api": {
      "type": "http",
      "url": "https://example.com/mcp",
      "headers": { "Authorization": "Bearer ${API_TOKEN}" }
    },
    "postgres": {
      "command": "npx",
      "args": ["@modelcontextprotocol/server-postgres"],
      "env": { "DATABASE_URL": "${DATABASE_URL}" }
    }
  }
}
```

1. `${VAR}` expands from the environment, so the secret is never in the file. `${VAR:-default}` covers unset ones.
2. `command` + `args` start a stdio server; remote entries set `type` and `url`.
3. Interactive sessions ask before using project servers; `claude -p` and Agent SDK runs load them without asking.
4. Remote auth: OAuth via `/mcp`, else a static token in `headers`, or `headersHelper` for short-lived ones.

Grep searches inside files, Glob finds files, Edit changes them in place.

TOOL	PURPOSE	WHEN TO USE
Grep	Search file contents for patterns	Function definitions, error messages, imports, string patterns
Glob	Find files by name or extension	`**/*.test.ts`, `src/**/config.*`
Read	Load full file contents	A specific file after Grep or Glob found it
Write	Create new files	Files that don't exist yet
Edit	Targeted changes by unique text matching	Specific sections of existing files
Bash	Run terminal commands	git, npm, scripts, system commands

Also built in: Agent, Skill, WebSearch and WebFetch, AskUserQuestion, and ToolSearch for deferred tools.

Explore from entry points outward, and fall back to Read + Write when Edit fails.

EXPLORING A CODEBASE

Grep, read, follow, repeat

- Grep for entry points like main functions, routes and exports.
- Read those files, then Grep for the modules they import.
- Build understanding iteratively.

ANTI-PATTERN

Reading every file upfront

Wastes context and doesn't build understanding.

WHEN EDIT FAILS

Read it, then Write it whole

- Edit matches a unique text string. If the text isn't unique, the edit fails.
- Fallback: Read the full file, then Write the entire modified version.

Server-side tools run on Anthropic's side, and their errors arrive as HTTP 200.

- Anthropic-defined tools are schema-less: declare a `type` and a `name`, and Claude knows the interface.
- Server-side tools such as web search and code execution return results as content blocks in the same response.
- A server-tool error returns 200 with an error object in the result block. Branch on its shape before indexing.
- Your own tool named `bash` with an `input_schema` is a *different* tool from Anthropic's schema-less one.

Web fetch only fetches URLs already in the conversation. `pause_turn` means a server tool loop hit its cap: send it back.

Parallel tool calls get all their results back in one message.

THE RIGHT ANSWER

Run them together, answer them together.

- Execute every `tool_use` block in the message concurrently.
- Return all `tool_result` blocks in a single user message.
- A tool that failed still gets its `tool_result`, with `is_error: true`.

Splitting results across messages
Silently teaches Claude to stop making parallel calls.

Dropping a failed result
Never drop it. Return the failure with `is_error: true`.

An agent has 18 tools configured and is frequently selecting the wrong tool. What is the most effective solution?

A. Improve all 18 tool descriptions to be more detailed

B. Distribute the tools across specialized subagents with 4-5 tools each

C. Set tool_choice to "any" to force tool usage

D. Add examples to the system prompt showing correct tool selection

B

Distribute the tools across specialized subagents with 4-5 tools each

WHY

The 4-5 tool maximum per agent is the key architectural principle. Even with better descriptions, 18 tools degrades selection reliability.

A Improve all 18 tool descriptions to be more detailed

C Set tool_choice to "any" to force tool usage

D Add examples to the system prompt showing correct tool selection

A search tool returns an empty array. What should the agent communicate to the user?

A "The search service is currently unavailable"

B "No results were found matching your query"

C Check the response status to distinguish between "no results" and "search failed"

D Retry the search with different parameters

C

Check the response status to distinguish between "no results" and "search failed"

A "The search service is currently unavailable"

B "No results were found matching your query"

D Retry the search with different parameters

WHY

The agent must distinguish between a valid empty result (status 200, no matches) and a failed search (error/timeout). These require fundamentally different responses.

Where should MCP server credentials be stored in a project using `.mcp.json`?

A. Directly in the `.mcp.json` file

B. In environment variables referenced via `${VAR_NAME}` syntax

C. In a separate `.mcp-secrets.json` file

D. In the project's `package.json`

B

In environment variables referenced via `${VAR_NAME}` syntax

A Directly in the `.mcp.json` file

C In a separate `.mcp-secrets.json` file

D In the project's `package.json`

WHY

`.mcp.json` supports environment variable expansion. Credentials should never be committed to version control.

WHY NOT A

`.mcp.json` lives in the project root and is version-controlled, so anything written in it gets committed.

A platform team aggregates six MCP servers, exposing about 200 tools. Tool definitions consume ~55k tokens before any work starts, and tool selection has become unreliable. What is the current recommended fix?

A Raise `max_tokens` so there is room for the definitions

B Enable the tool search tool, keep it and the 3-5 most-used tools loaded, and mark the rest `defer_loading: true`

C Set `tool_choice: "any"` so Claude is forced to commit to a tool

D Set `defer_loading: true` on every tool including the search tool

B

Enable the tool search tool, keep it and the 3-5 most-used tools loaded, and mark the rest `defer_loading: true`

A Raise `max_tokens` so there is room for the definitions

C Set `tool_choice: "any"` so Claude is forced to commit to a tool

D Set `defer_loading: true` on every tool including the search tool

WHY

Tool search keeps only a small hot set plus the search tool in context and loads the rest on demand via `tool_reference` blocks, cutting definition tokens by ~85% while preserving prompt caching. Deferring every tool, the search tool included, is a 400 error: at least one tool must stay non-deferred. Splitting across subagents (§2.3) is still valid, but tool search is what makes a 200-tool catalog workable inside one agent.

An extraction service forces its schema with `tool_choice: {"type": "tool", "name": "extract_invoice"}` and is being moved from Claude Opus 5 to Claude Fable 5.1. What happens, and what is the right fix?

A. Nothing changes — forced tool use works on every current model

B. The request returns 400; switch to `tool_choice: "any"` instead

C. The request returns 400; use `tool_choice: "auto"` with an explicit instruction and `strict: true`, or structured outputs via `output_config.format`

D. The call silently falls back to text output

C

The request returns 400; use `tool_choice: "auto"` with an explicit instruction and `strict: true`, or structured outputs via `output_config.format`

A Nothing changes — forced tool use works on every current model

B The request returns 400; switch to `tool_choice: "any"` instead

D The call silently falls back to text output

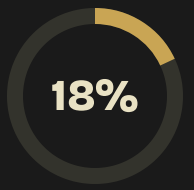
WHY

Fable 5.1 and Mythos 5.1 reject both forced forms (`any` and `tool`) with a 400. `auto` plus an instruction keeps the tool call, `strict: true` keeps the arguments schema-valid, and structured outputs replace the pattern entirely when the tool only existed to return JSON. Opus 5.5 and Sonnet 5.5 reject it exactly as Fable 5.1 does, while Haiku 5.5 (and legacy Sonnet 5 / Opus 5) still accept it — so this is a per-model check, not a global rule, and three of the four current models fail it.

WHY NOT B

`any` is forced tool use too. The same models reject it with a 400, just like a named tool.

Tool Design & MCP Integration



2.1 Tool design

Descriptions are the selection criteria: purpose, input, output, edge cases, when to use vs similar. Overlap → rename or merge.

2.2 Errors

Category + retryable. Transient → retry with backoff. Validation → fix input. Business → explain. Permission → escalate. Empty ≠ failed.

2.3 Distribution

Four or five tools per agent; split the rest across subagents. `auto` may, `any` must, a named tool forces it, `none` is text only.

2.4 MCP architecture

A host runs one client per server. stdio for local, Streamable HTTP for remote. A `url` always needs a `type`.

2.4 MCP config

`.mcp.json` is project scope and committed, so secrets go in `${VAR}`. Local and user scope live in `~/.claude.json`.

2.5 Built-in tools

Grep for contents, Glob for names. Explore from entry points, never read everything upfront. Edit fails on non-unique text → Read + Write.

2.6 API-side tools

Schema-less: just `type` + `name`. Server-tool errors arrive inside a 200. Parallel results go back in one message.



DOMAIN 3 OF 5

Claude Code Configuration & Workflows

How Claude Code is configured and driven: instruction files, skills, rules, plan mode, refinement and CI/CD. Seven task statements:

- 3.0 Surfaces & scheduling
- 3.1 CLAUDE.md hierarchy
- 3.2 Skills & commands
- 3.3 Path-specific rules
- 3.4 Plan mode vs direct execution
- 3.5 Iterative refinement
- 3.6 CI/CD integration

Schedulers differ by where they run: the cloud, your machine, or this session.

IN THE CLOUD

Routines

- Keep running with your machine off.
- Can also trigger on API calls or GitHub events.
- Morning PR reviews, overnight CI failure analysis, weekly dependency audits.

ON YOUR MACHINE

Desktop scheduled tasks

- Run locally, with local file and tool access.
- For anything that needs the local filesystem or a private network.

IN THIS SESSION

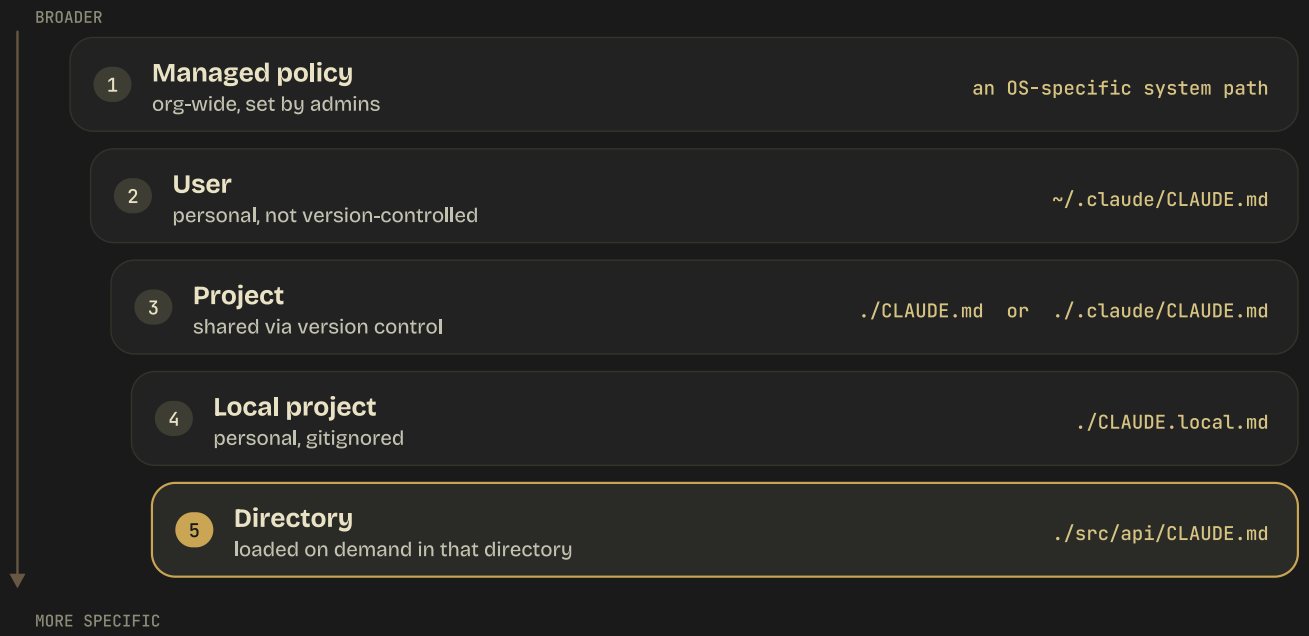
`/loop`

- Runs inside the current CLI session.
- Quick polling within one sitting.
- Omit the interval and Claude self-paces between iterations.

Every surface (terminal, IDE, desktop, web, mobile) runs the same engine, so CLAUDE.md, settings, skills and MCP servers work identically everywhere.

CLAUDE.md files stack up; none of them overrides another.

- Every file found is concatenated into context, broadest first.
- Files nearer the launch directory come last; `CLAUDE.local.md` follows `CLAUDE.md`.
- Directory files load on demand, when Claude works on a file there.
- Contradictions leave Claude to pick one. Resolve them; don't rely on precedence.



Monorepo with another team's nested CLAUDE.md? Skip it with `claudeMdExcludes` in settings.

Only the project CLAUDE.md is shared; every other layer stays with you.

SHARED · IN GIT

Project CLAUDE.md

- `./CLAUDE.md` or `./.claude/CLAUDE.md`, committed to version control.
- The team's coding standards, conventions and workflow rules.

PERSONAL

User & local files

- `~/.claude/CLAUDE.md` applies to all your projects and is never version-controlled.
- `CLAUDE.local.md` is personal per project: gitignore it.

MACHINE-LOCAL

Auto memory

- Notes Claude writes itself: project patterns, decisions, preferences.
- The start of `MEMORY.md` loads each session; topic files load on demand.
- All worktrees of one repo share it.

Imports keep CLAUDE.md modular, with one source of truth.

MARKDOWN

```
# CLAUDE.md
@AGENTS.md
@docs/coding-standards.md
@~/.claude/my-personal-preferences.md

See @README.md for project overview.

## Claude-specific overrides
Use plan mode for changes under `src/billing/`.
```

1. Relative paths resolve from the file that holds the import; `@~/` is your home directory.
2. Imported files can import too, up to a depth limit that stops circular imports.
3. No CLAUDE.md in the project? Claude Code falls back to reading `AGENTS.md`.
4. A personal `CLAUDE.local.md` counts as having one and silently stops `AGENTS.md` loading. Import it with `@AGENTS.md`.

Custom commands are now skills: same `/name`, more power.

- `.claude/commands/deploy.md` and `.claude/skills/deploy/SKILL.md` both create `/deploy` and behave the same.
- On a name clash, Enterprise > Personal > Project > Plugin, and a skill beats a same-named command.
- Skills add a folder for supporting files, dynamic context injection, and subagent execution.
- The body enters the conversation once and is never re-read, so write standing instructions.

The old `/project:name` and `/user:name` namespacing is gone. Plugin skills are namespaced `plugin-name:skill-name`.

Frontmatter decides who can run a skill and where it runs.

YAML

```yaml
---
name: deploy
description: Deploy the application to a specified environment
context: fork
agent: Explore
allowed-tools: Bash(git *) Bash(npm run deploy *)
argument-hint: [environment]
disable-model-invocation: true
---

Deploy the application to $ARGUMENTS.
```

1. `description` is how Claude decides whether to auto-load the skill.
2. `disable-model-invocation: true` means only the user can run it. `user-invocable: false` hides it from the menu instead.
3. `context: fork` runs it in an isolated subagent, chosen by `agent`. CLAUDE.md loads; chat history doesn't.
4. `allowed-tools` pre-approves tools for the invoking turn only. `$ARGUMENTS` is what the user typed.

Pick the extension by what it has to change.

SKILL

What Claude knows and does

A `SKILL.md` of instructions. Pick it when you keep pasting the same instructions.

MCP SERVER

Which tools Claude has

An external process that gives Claude tools. Pick it when Claude needs to reach an external system.

SETTINGS HOOK

Whether an event goes ahead

A shell command, HTTP request or prompt run on a lifecycle event. Pick it to block, allow or log with a script.

MOD

Events and the interface

Plugin functions run inside Claude Code, unsandboxed. Pick it for a pane, a custom command, or rewriting an event.

A plugin bundles these and more, namespaced so they can't collide with yours. Plugins sit lowest in skill priority.

Conventions that span many directories belong in one path-scoped rule.

DIRECTORY CLAUDE.MD

A copy in every folder

- Testing rules repeated in `components/`, `api/`, `utils/`…
- Every copy has to be kept in sync by hand.
- A new test directory gets nothing until someone adds another copy.

PATH-SPECIFIC RULE

One file in `.claude/rules/`

- `testing.md` with `paths: ["**/*.test.*"]` in its frontmatter.
- A single source of truth for testing conventions.
- Applies wherever test files exist, even in new directories.

A `paths` glob turns a rule from always-on into on-demand.

YAML

```yaml
---
paths:
  - "src/api/**/*.ts"
  - "src/api/**/*.test.ts"
---

# API Development Rules

All API endpoints must:
- Use Zod for request validation
- Return standardized error responses
- Include rate limiting middleware
```

1. Rules live in `.claude/rules/`, one topic per file: testing, API design, security.
2. No frontmatter means the rule loads unconditionally at session start.
3. With `paths`, it loads when Claude reads, writes or edits a matching file.
4. Skills take the same `paths` field to auto-load only for matching files.

Plan when the scope or approach is open; execute directly when it's obvious.

PLAN MODE

Explore, design, then ask

- Multi-file changes, like refactoring auth across 15 files.
- Architectural decisions, or several valid approaches.
- Large scope, like a library migration touching 45+ files.
- Unclear requirements, like “make the app faster”.

DIRECT EXECUTION

Just make the change

- A single-file fix with a clear stack trace.
- Obvious scope, like adding a null check.
- Clear instructions: “Add a `createdAt` field to the User model”.
- Small and contained, like a typo or a constant.

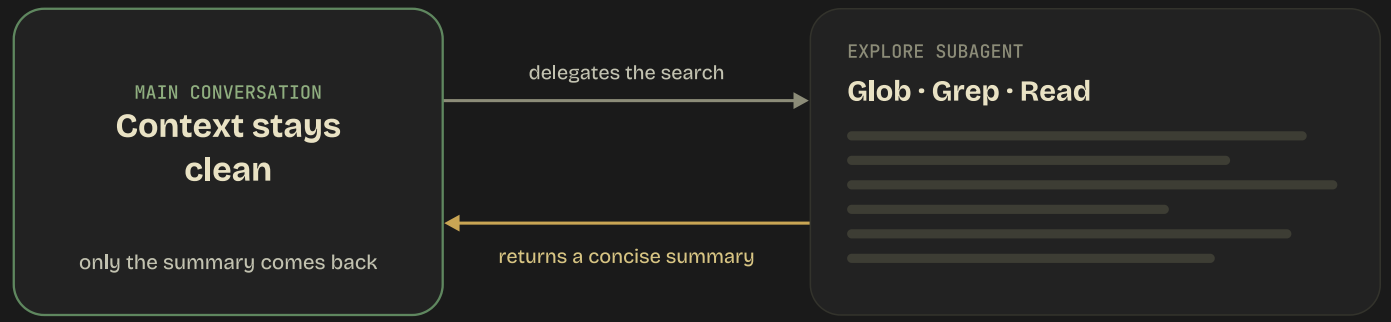
Combine them: plan mode explores and designs, you approve, then Claude exits plan mode and implements file by file.

Plan mode is one permission mode among several.

MODE	BEHAVIOR
`default`	Shown as Manual. Reads run; edits, shell commands and network access prompt.
`acceptEdits`	Auto-accepts file edits; other tools still prompt.
`plan`	Exploration only: designs an approach and presents it before touching files.
`auto`	A classifier decides per tool call whether to prompt.
`dontAsk`	Only in-directory reads and pre-approved tools run; anything else is denied.
`bypassPermissions`	Skips all prompts. Dangerous: short, contained tasks only.

Set one with `--permission-mode` or `permissions.defaultMode`, or cycle with Shift+Tab. `dontAsk` suits locked-down CI.

Hand noisy exploration to an Explore subagent and keep only its summary.



Without it, raw file contents and search results would fill the main context window.

- Explore runs Glob, Grep and Read in its own context, then returns a concise summary.
- Without it, raw file contents and search results fill the main context window.
- Explore and Plan are read-only built-ins; general-purpose can explore *and* act.
- Every subagent starts without your conversation history; Explore and Plan even skip CLAUDE.md.

Define your own in `.claude/agents/`: markdown plus frontmatter such as `tools`, `model` and `permissionMode`.

Refinement works on concrete evidence, not more prose.

INCONSISTENT OUTPUT

Concrete examples

Prose like “format dates consistently” keeps drifting. Give 2–3 input/output examples instead.

CORRECTNESS

TDD iteration

Write the test first, run it, share the failure. Claude fixes the code; rerun and repeat until green.

VAGUE REQUEST

Interview pattern

Claude asks clarifying questions before implementing: JWT or sessions? Refresh tokens? Role-based access?

SEVERAL ISSUES

Interacting vs independent

Interacting issues go in one message, since fixes may conflict. Independent ones go one at a time, each reviewable.

Three CI/CD answers that look right.

THE RIGHT ANSWER

`claude -p`, parseable output, hard limits.

- `-p` / `--print` runs non-interactively. Mandatory in CI.
- `--output-format json` makes the output parseable.
- `--json-schema` enforces its shape.
- `--max-turns` and `--max-budget-usd` as guardrails.

`--permission-prompts none` when nobody can answer a prompt.

Running Claude Code without `-p`

The process hangs, waiting for input that never comes.

The Batch API for pre-merge checks

A blocking check needs an immediate response. Batches have no latency SLA.

Reviewing in the session that wrote the code

It keeps its reasoning context, so it is less likely to question its own decisions.

The Batch API is for work nobody is waiting on.

GOOD FOR

Overnight, non-blocking jobs

- Overnight code analysis reports and weekly audit summaries.
- Nightly test generation and batch document processing.
- 50% cheaper than synchronous calls; results within 24 hours.

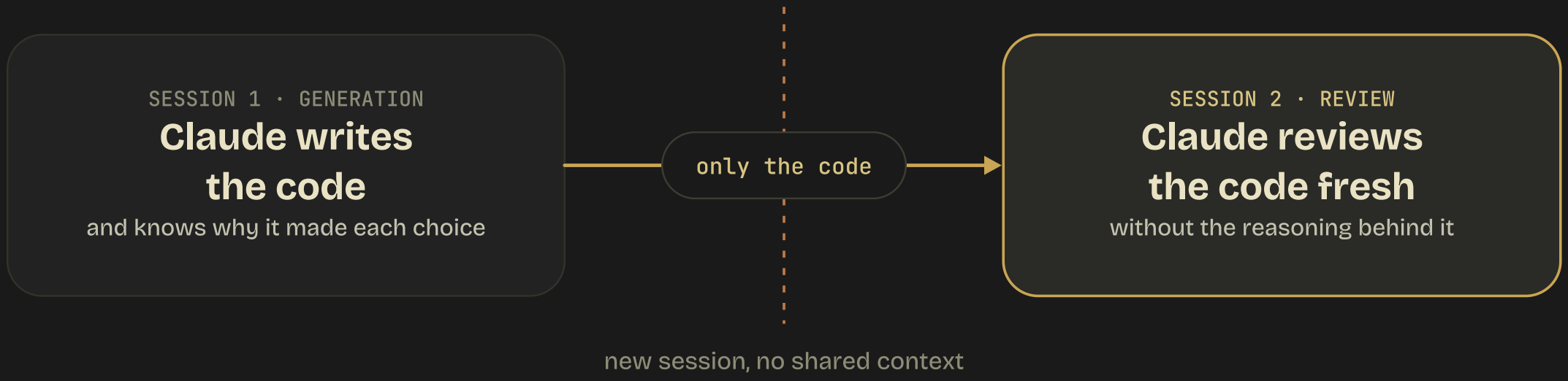
NOT GOOD FOR

Anything that blocks

- Pre-merge checks that need an immediate response.
- Real-time user interactions.
- Tool-calling loops. Multi-turn isn't supported inside one request.

Results arrive in any order: match them by `custom_id`, never by position, and resubmit only the failures.

Review code in a fresh session, not the one that wrote it.



The writing session already “knows” why it made each choice, so it rarely questions them. A new session with no shared context reviews the code fresh.

A team wants testing conventions to apply to all `*.test.ts` files throughout their codebase, which has tests scattered across many directories. What is the most maintainable approach?

A. Add a CLAUDE.md in every directory that contains test files

B. Create a single rule file in `.claude/rules/` with `paths: ["**/*.test.ts"]` frontmatter

C. Add testing rules to the project-level CLAUDE.md

D. Create a `tests/` directory and put all tests there with a CLAUDE.md

B

Create a single rule file in `.claude/rules/` with `paths: ["**/*.test.ts"]` frontmatter

A Add a CLAUDE.md in every directory that contains test files

C Add testing rules to the project-level CLAUDE.md

D Create a `tests/` directory and put all tests there with a CLAUDE.md

WHY

Path-specific rules in `.claude/rules/` are superior to directory-level CLAUDE.md when conventions span multiple directories. A single rule file with glob patterns covers all matching files without duplication.

WHY NOT A

A CLAUDE.md in every test directory duplicates the rules, and a new test directory gets nothing until someone adds another copy.

When should Claude Code use plan mode instead of direct execution?

A. When fixing a single bug with a clear stack trace

B. When migrating a library that affects 45+ files across the project

C. When adding a null check to a function

D. When updating a configuration constant

B

When migrating a library that affects 45+ files across the project

A When fixing a single bug with a clear stack trace

C When adding a null check to a function

D When updating a configuration constant

WHY

Plan mode is for multi-file changes with architectural implications and multiple valid approaches. A library migration affecting 45+ files clearly fits. The other options are small, obvious-scope changes suited for direct execution.

WHY NOT A

A single bug with a clear stack trace is small, obvious scope, the textbook case for direct execution.

A CI pipeline needs to run Claude Code for automated code review. Which flag is essential?

A `--verbose`

B `-p` / `--print`

C `--force`

D `--no-cache`

B

`-p` / `--print`

A `--verbose`

C `--force`

D `--no-cache`

WHY

The `-p`/`--print` flag enables non-interactive mode, which is mandatory in CI. Without it, Claude Code hangs waiting for user input. Pair it with `--output-format json` (or `stream-json`), `--max-turns` / `--max-budget-usd` for production-grade guardrails, and `--permission-prompts none` when no host can answer a permission prompt.

Claude Code Configuration & Workflows



3.0 Surfaces

One engine everywhere, so config travels. Routines run in the cloud, desktop tasks locally, `/loop` in the session.

3.1 CLAUDE.md

Managed → user → project → local → directory, all concatenated. Only project is shared. `@import` to split.

3.2 Skills

Commands merged into skills. `disable-model-invocation` = user-only. `context: fork` = isolated subagent.

3.3 Path rules

`.claude/rules/` with a `paths` glob: one file for conventions spread across directories.

3.4 Plan mode

Multi-file, architectural or unclear → plan. Small and obvious → direct. Explore keeps the main context clean.

3.5 Refinement

Examples beat prose. Share test failures. Interview first. Fix interacting issues together.

3.6 CI/CD

`-p` or it hangs. JSON output plus a schema. Batch only for non-blocking work. Review in a fresh session.



DOMAIN 4 OF 5

Prompt Engineering & Structured Output

How to write criteria the model can't misread, get structured output you can validate, and shape requests for current models. Eight task statements:

- 4.1 Explicit criteria
- 4.2 Few-shot prompting
- 4.3 Structured output
- 4.4 Validation-retry loops
- 4.5 Batch processing
- 4.6 Multi-pass review
- 4.7 Thinking & effort
- 4.8 Prompt caching

Vague instructions get read differently every time.

THE RIGHT ANSWER

Define each judgment explicitly.

- **What qualifies**: the specific conditions that trigger the judgment.
- **What doesn't**: the boundary cases that must not trigger it.
- **Examples**: concrete input/output pairs that show the boundary.

"Flag a comment as outdated ONLY when the claimed behavior directly contradicts the actual code behavior."

Asking for caution
"Be conservative when flagging issues."

Asking for confidence
"Only report high-confidence findings."

Asking for judgment
"Use your best judgment on severity."

One noisy category erodes trust in every category.

- High false positive rates in one category undermine trust in all of them, even the accurate ones.
- Developers start ignoring every warning, not just the noisy kind.
- The fix: temporarily disable the noisy category, improve its criteria, then re-enable it.
- Tag each finding with a `detected_pattern` (4.4) to see which constructs developers keep dismissing.

Few-shot examples show what prose can't pin down.

- Use them for ambiguous cases, formats or reasoning to demonstrate, edge cases, and extraction hallucination.
- **2–4 targeted examples** is the sweet spot. Too few miss the pattern; too many waste context and overfit.
- Each example shows the input, the output, and the reasoning for why that output is correct.
- Good examples teach the pattern, so the model generalizes to situations it hasn't seen.

Include a boundary case too: a comment that is vague but not wrong, whose correct output is `null`.

Don't ask for JSON in text; give Claude a schema.

FRAGILE

JSON requested in the prompt

- Malformed JSON
- Missing required fields
- Unexpected field names
- Wrong data types

TOOL_USE

A tool whose `input_schema` is the output

- Define a tool that represents the structure you want back.
- `strict: true` guarantees schema compliance.

DIRECT JSON

`output_config.format`

- Request JSON against a `json_schema`, no tool needed.
- The SDK's `messages.parse()` takes a Pydantic or Zod schema and returns the validated object.
- The older top-level `output_format` is deprecated.

A strict tool schema is a contract for the output.

JSON

```json
{
  "name": "extract_invoice_data",
  "description": "Extract structured data from an invoice document",
  "strict": true,
  "input_schema": {
    "type": "object",
    "properties": {
      "vendor_name": { "type": "string" },
      "total_amount": { "type": "number" }
    },
    "required": ["vendor_name", "total_amount"],
    "additionalProperties": false
  }
}
```

1. The `input_schema` is the structure you want back.
2. `strict: true` guarantees the output matches it.
3. Strict mode needs `additionalProperties: false` and every property listed in `required`.
4. Some current models reject forced `tool_choice` with a 400. There, use `output_config.format`, or `auto` plus `strict: true`.

Structured output guarantees the shape, not the truth.

GUARANTEED

Schema compliance

- Correct types
- Required fields present
- Valid enum values
- `"total_amount": 150.00` is a number in the right field.

NOT GUARANTEED

Semantic correctness

- `"total_amount": 150.00` while the line items sum to 175.00.
- Valid against the schema, and still wrong.
- You still need validation logic to catch semantic errors.

Leave room for missing or unclear data, and check the rest in code.

NULLABLE FIELDS

Missing data becomes `null`

Use `"type": ["string", "null"]` when the source may not contain it. Making everything required can force Claude to fabricate data.

ENUM + OTHER

An escape hatch

Add `"other"` to the enum, paired with a nullable detail field that describes what the document actually is.

ENUM + UNCLEAR

Permission not to guess

Add `"unclear"` so Claude doesn't guess when the information is ambiguous.

STRICT MODE SUBSET

Some rules live in your code

No `minimum`/`maximum`, length limits, `pattern`, `oneOf` or recursive schemas. Validate ranges, lengths and regex yourself.

Feed the specific validation errors back into the retry.

PYTHON

```python
max_retries = 3

for iteration in range(max_retries):
    response = extract_data(document, prompt)
    errors = validate(response)
    if not errors:
        return response  # success
    prompt += f"""
    Your previous extraction had these validation errors:
    {json.dumps(errors, indent=2)}
    Please correct these specific issues and try again.
    """

raise ExtractionError(f"Failed after {max_retries} attempts")
```

1. Validate every response and return the first clean one.
2. Schema checks catch missing fields, wrong types and bad enums. Sums, dates and cross-field contradictions need custom logic.
3. Extract a stated and a calculated total side by side to detect conflicts automatically.
4. Cap the attempts and raise when they run out.

Retries can't recover data the document doesn't contain.

THE RIGHT ANSWER

Accept `null` for missing data.

- Make fields nullable or optional when the source may not contain the information.
- No PO number on the invoice? Return `null` instead of retrying.

Retrying until the field is filled
If the invoice doesn't have a PO number, retrying won't make one appear.

Making every field required
Requiring data the document may lack can force Claude to fabricate it.

Batch what can wait; keep anything blocking synchronous.

SYNCHRONOUS

Results needed now

- Pre-merge code review checks
- Real-time user interactions
- Blocking CI steps
- Interactive debugging

MESSAGE BATCHES API

Results can wait

- 50% cheaper than synchronous calls.
- Up to 24 hours, with no SLA.
- Overnight code analysis, weekly audit reports, nightly test generation, bulk document processing.

Multi-turn conversations are not supported inside a single batch request.

Key every batch result by `custom_id`, and resubmit only the failures.

THE RIGHT ANSWER

Correlate by `custom_id`.

- Give every request a `custom_id`. Results come back in any order.
- Poll `processing_status` until `ended`, then stream the results.
- Resubmit only the requests that didn't succeed.

Result states: `succeeded`, `errored`, `canceled`, `expired`.

Resubmitting the whole batch
Wastes credits and reprocesses documents that already succeeded.

Matching results by position
Results arrive in any order. Key by `custom_id`, never by position.

Skipping the trial run
Refine prompts on a small sample before spending credits on thousands of documents.

Review code in a fresh session, not the one that wrote it.



- In the same session, Claude keeps the reasoning context from generation.
- It “remembers” why it made each decision, so it is less likely to question them.
- This is **reasoning context bias**.
- The fix: a completely separate review session with no knowledge of why decisions were made.

Review large PRs file by file first, then across files.

SINGLE PASS

Breaks down on large PRs

- Attention dilution: early files get more focus than later ones.
- Inconsistent standards across files.
- Missed cross-file integration issues.

PASS 1

Per-file local analysis

- A separate prompt for each file, at consistent depth.
- Catches bugs, style issues, type errors and security vulnerabilities.

PASS 2

Cross-file integration

- A different prompt focused on how files interact, with Pass 1 results as context.
- Catches data flow issues, interface mismatches and missing error handling at boundaries.

Older request shapes now return a 400 on current models.

THE RIGHT ANSWER

Adaptive thinking, steered by effort.

- `thinking={"type": "adaptive"}` replaces fixed thinking budgets.
- Control depth with `effort` inside `output_config`, not at the top level.
- Force a format with `output_config.format` or system-prompt instructions.

`budget_tokens`
Fixed thinking budgets return a 400.

`temperature`, `top_p`, `top_k`
The sampling parameters return a 400 too.

Assistant prefill
Prefilling the last assistant turn to force a format returns a 400, though older material still recommends it.

Four request-surface rules to carry through every migration.

EFFORT

Set it explicitly, re-tune on migration

Defaults differ by model, so an omitted `effort` can run shallower after a migration. Re-run your sweep; don't carry the old value across.

THINKING DISPLAY

Hidden by default

Thinking still happens and is still billed, but `display: "omitted"` hides it. Streaming reasoning to users? Set `"summarized"`.

SYSTEM MESSAGES

A mid-conversation operator channel

Append a `"role": "system"` message to `messages` instead of editing `system`. It is prompt-injection-safe and leaves the cached prefix intact.

PRESERVED THINKING

History is part of the contract

Thinking blocks are bound to the model, prefix and account that produced them. Editing earlier turns invalidates later blocks; append-only histories are safe.

The cache keys on an exact prefix, so put stable content first.

THE RIGHT ANSWER

Stable first, volatile last.

- Render order is `tools` → `system` → `messages`. Any byte change invalidates everything after it.
- Frozen system prompt and deterministically ordered tools first.
- Timestamps, per-request IDs and the question go after the last `cache_control` breakpoint.
- Verify with `usage.cache_read_input_tokens`. Zero across repeats means the prefix is breaking.

`datetime.now()` in the system prompt

The prefix changes on every request, invalidating everything behind it.

A tool list that varies or reorders

Tools render first, so it invalidates the system prompt and the whole conversation.

Changing effort or thinking between requests

Switching the top-level `effort` or thinking configuration invalidates the cache. Use a per-message effort change.

A code review system flags "use your best judgment" as a criterion for reporting issues. What should be changed?

A. Add more examples of past reviews

B. Replace with explicit criteria defining exactly what qualifies as a reportable issue

C. Increase the model's temperature for more nuanced judgment

D. Add a confidence score threshold

B

Replace with explicit criteria defining exactly what qualifies as a reportable issue

A Add more examples of past reviews

C Increase the model's temperature for more nuanced judgment

D Add a confidence score threshold

WHY

Vague instructions like "use your best judgment" produce inconsistent results. Replace with explicit criteria that define exactly what qualifies and what doesn't.

An invoice extraction system uses strict mode but sometimes returns a total that doesn't match the sum of line items. What's happening?

A. Strict mode is broken

B. The schema is misconfigured

C. Strict mode guarantees schema compliance, not semantic correctness

D. The model needs fine-tuning

C

Strict mode guarantees schema compliance, not semantic correctness

A Strict mode is broken

B The schema is misconfigured

D The model needs fine-tuning

WHY

Strict mode ensures the output conforms to the JSON schema (correct types, required fields) but cannot validate semantic correctness (whether the math adds up). Custom validation logic is needed for that.

A batch processing job has 1,000 documents. 50 fail. What's the correct approach?

A. Resubmit the entire batch of 1,000

B. Use `custom_id` to identify and resubmit only the 50 failed documents

C. Switch to synchronous processing

D. Increase the batch timeout

B

Use `custom_id` to identify and resubmit only the 50 failed documents

- **A** Resubmit the entire batch of 1,000
- **C** Switch to synchronous processing
- **D** Increase the batch timeout

WHY

Use `custom_id` to correlate requests with responses and resubmit only the failures. Resubmitting the entire batch wastes credits and reprocesses already-successful documents.

An extraction service written for an older model sets `thinking={"type": "enabled", "budget_tokens": 8000}` and `temperature=0`. It is being moved to Claude Opus 5. What happens?

A. It works unchanged; both parameters are still supported

B. Both parameters return a 400 — use `thinking={"type": "adaptive"}` and control depth with `output_config.effort`

C. `budget_tokens` is ignored silently and `temperature` still applies

D. It works, but thinking is disabled

B

Both parameters return a 400 — use `thinking={"type": "adaptive"}` and control depth with `output_config.effort`

A It works unchanged; both parameters are still supported

C `budget_tokens` is ignored silently and `temperature` still applies

D It works, but thinking is disabled

WHY

`budget_tokens` and the sampling parameters (`temperature` / `top_p` / `top_k`) are removed on Fable 5 / 5.1, Opus 5.5, Opus 5, Sonnet 5.5 / 5, Haiku 5.5, and the 4.7/4.8 family, and return a 400 (Haiku 5.5 tolerates only the defaults, `temperature: 1` and `top_p: 0.99`). Moving the same service to Haiku 5.5 — the natural successor for high-volume extraction — fails the same way. Adaptive thinking plus `effort` replaces the fixed-thinking-budget concept. Assistant prefills also 400 on these models.

Prompt Engineering & Structured Output



4.1 Explicit criteria

Define what qualifies, what doesn't, and examples. One noisy category? Disable it, fix its criteria, re-enable.

4.2 Few-shot

2–4 targeted examples, each with input, output and reasoning. For ambiguity, formats and edge cases.

4.3 Structured output

A tool's `input_schema` or `output_config.format`, plus `strict: true`. It guarantees the shape, not the truth: validate semantics in code.

4.4 Validation-retry

Feed the specific errors back and retry. Data absent from the source won't appear: accept `null`.

4.5 Batch

Cheaper, up to 24 hours, no SLA. Key by `custom_id`; resubmit only failures. Keep blocking work synchronous.

4.6 Multi-pass review

Review in a fresh session to avoid reasoning context bias. Large PRs get a per-file pass, then a cross-file pass.

4.7 Thinking & effort

Adaptive thinking plus `output_config.effort`. `budget_tokens`, sampling parameters and prefill return a 400.

4.8 Prompt caching

Exact prefix match, in `tools` → `system` → `messages` order. Stable first, volatile after the last breakpoint.



DOMAIN 5 OF 5 · THE LAST

Context Management & Reliability

How long conversations keep their facts, when an agent hands off to a human, how errors travel, and how outputs stay checkable. Eight task statements:

- 5.1 Conversation context
- 5.2 Escalation patterns
- 5.3 Error propagation
- 5.4 Large codebase context
- 5.5 Human review workflows
- 5.6 Information provenance
- 5.7 Server-side context
- 5.8 Bounding long runs

Long context costs more and loses the middle, so keep it lean and ordered.

- Every token is billed on every turn. Filling a big window just because you can costs money and quality.
- Models read the beginning and end of long inputs best, and the middle gets worse with length.
- Put key-findings summaries first, add explicit section headers, and bookend the important parts.
- Trim tool results before returning them: an order lookup may send 40 fields when you need 5.

Exact numbers survive summarization only in a case-facts block.

TEXT

```
CASE FACTS (do not summarize):
- Customer ID: #12345
- Order: #67890
- Order date: 2024-01-15
- Total: $149.99
- Issue: Damaged product on arrival
- Refund requested: Full refund ($149.99)
```

1. Summaries round things off: `$149.99` becomes “approximately $150”, `2024-01-15` becomes “mid-January”.
2. Order numbers, customer IDs, reference codes and exact error messages get lost too.
3. Extract them into a persistent structured block that survives summarization.
4. No compaction mechanism protects exact numbers better than restating them.

Nested CLAUDE.md files and path-scoped rules don't survive `/compact`.

SURVIVES /COMPACT	MAY BE LOST
Project-root CLAUDE.md and unscoped rules, re-injected from disk	Earlier conversation details
System prompts	Specific tool results
Recent messages	Nuanced reasoning from earlier turns
Auto-memory (MEMORY.md)	Exact quotes and numbers from early context
The plan-mode plan, recently edited files, invoked skill bodies (capped)	Nested CLAUDE.md files and `paths:`-scoped rules, until Claude next touches a matching file

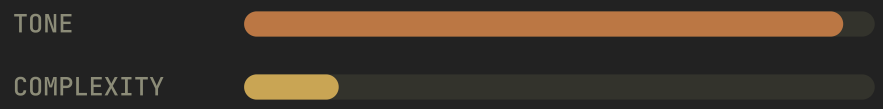
Put a rule that must survive compaction in the project-root CLAUDE.md, or drop its `paths:` frontmatter.

Escalate on issue complexity, never on emotional tone.

CUSTOMER

“I'M SO FRUSTRATED WITH THIS!!!”

+ a simple billing error



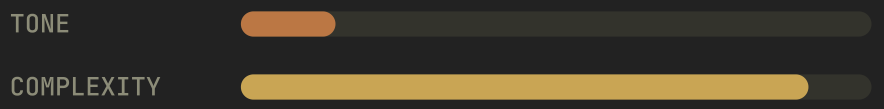
Resolve

Don't escalate on caps and exclamation marks.

CUSTOMER

“...the interaction between your enterprise SLA and the new EU data residency requirements”

+ a calm tone



Escalate

Policy complexity, despite calm sentiment.

Caps and exclamation marks over a simple billing error: resolve it. A calm question about how policies interact: it may need a human.

Mood, confidence and length are not reasons to escalate.

THE RIGHT ANSWER

Escalate on these three triggers.

- The customer explicitly asks for a human: escalate immediately, no further investigation.
- A policy gap or exception: the agent can't make policy decisions.
- No progress after reasonable attempts: hand off a summary of what was tried.

High sentiment

An angry customer may have a simple issue. Sentiment ≠ complexity.

Self-reported low confidence

Claude's confidence scores are poorly calibrated: a “low confidence” answer may be right, a “high confidence” one wrong.

A long conversation

Length doesn't signal a need to escalate. Some issues legitimately take multiple turns.

Ask instead of guessing, and hand off a summary that stands on its own.

MULTIPLE CUSTOMER MATCHES

Ask, don't pick by heuristics

“I found multiple accounts matching that name. Could you provide your email address or the last four digits of your phone number?”

ESCALATION HANDOFF

Send a structured summary

- Customer, contact details, issue type and order.
- Every action the agent already took.
- Recommended action and urgency.

The human agent typically can't see the conversation transcript.

An error Claude can't read is an error Claude can't recover from.

THE RIGHT ANSWER

Propagate structured context, then keep going.

- Say what failed, the service, the attempted query, partial results, alternatives, and whether it's retryable.
- Service down (503 or timeout) → "I wasn't able to search right now."
- Valid search, no matches (200 + empty array) → "I didn't find any matching articles."
- One step times out → continue with partial results, note the gap, offer alternatives.

Generic error messages

`{ "error": "Operation failed" }` hides what failed and why, so Claude can't recover intelligently.

Silent suppression

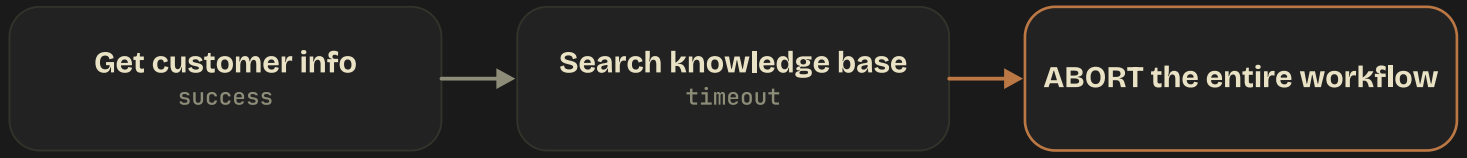
Returning `{ "results": [] }` while search is down makes Claude say "nothing found" when the truth is "I couldn't search."

Aborting the whole workflow

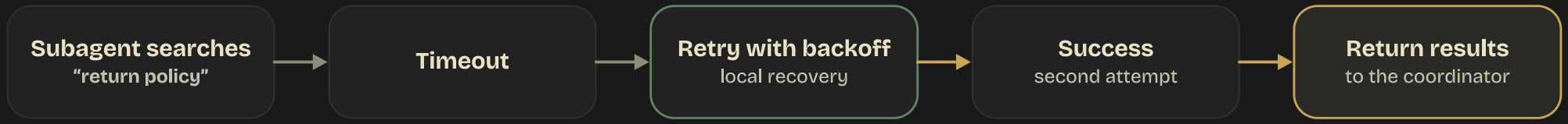
One timeout in step 2 throws away a run whose step 1 already succeeded.

Subagents recover locally from transient failures and propagate only what they can't fix.

WRONG · ONE TIMEOUT ENDS EVERYTHING



RIGHT · RECOVER LOCALLY, PROPAGATE ONLY WHAT YOU CAN'T FIX



No retry possible? Continue with partial results, note the gap, and offer alternatives.

A propagated error says what was attempted, what partial results exist, what failed, and whether it's retryable.

Long sessions drift, so persist what you learn before compression takes it.

Signs of drift: Claude cites “typical patterns” instead of code it found, gives inconsistent answers, forgets discoveries, and assumes instead of checking.

SCRATCHPAD FILES

Write findings down

Persist findings such as entry points, key facts and known issues to a file. After compression, Claude re-reads it to recover them.

PROACTIVE /COMPACT

Compact on your terms

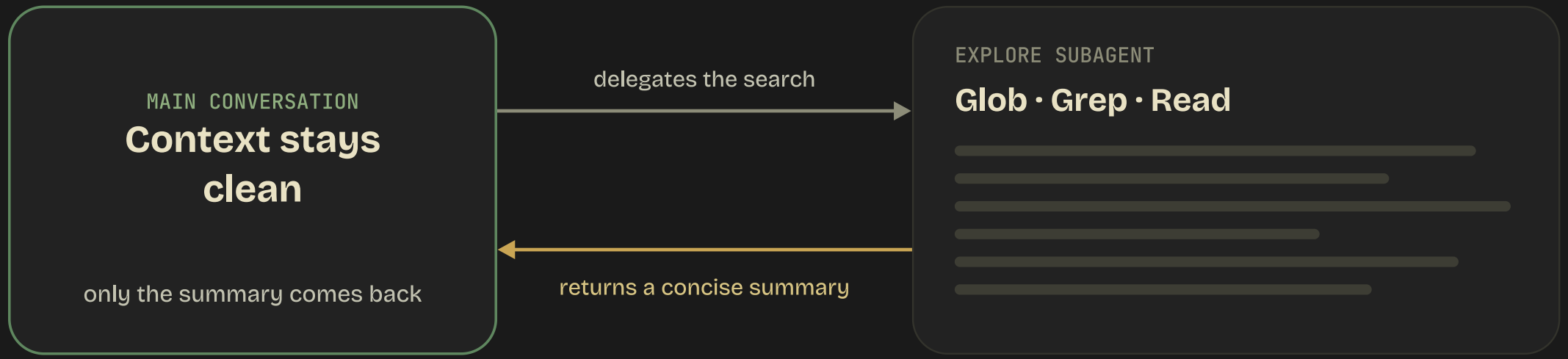
Compact with a structured summary before context degrades, rather than letting automatic compression lose information unpredictably.

CRASH RECOVERY

Resume from a manifest

In long multi-agent runs, each agent exports its state to a known location. The coordinator keeps a manifest and picks up where it left off.

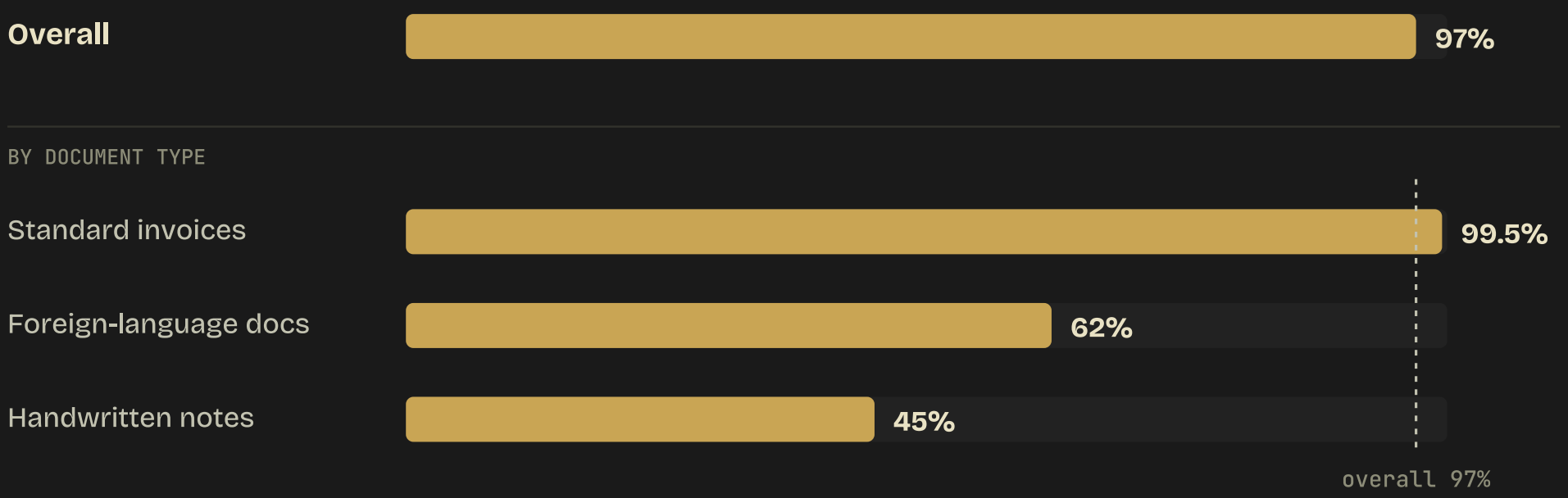
An Explore subagent reads widely so the main context gets only the summary.



Without it, raw file contents and search results would fill the main context window.

A mitigation for drift in long sessions. Verbose exploration stays in the subagent, and the main context stays focused on the task.

A strong overall accuracy can hide the document types that fail.



Report accuracy for each document type and each field within it before calling a system production-ready.

Aim human review by risk, and trust only calibrated per-field confidence.

SAMPLING

Stratify by risk or category

- Don't just sample randomly from all outputs.
- For example: 20% of high-risk, 10% of medium-risk and 5% of low-risk outputs.
- Review effort lands where errors are most likely.

CONFIDENCE

Score each field, then calibrate

- One confidence score per field, not one for the whole extraction.
- Calibrate against labeled validation sets: 0.9 should mean correct about 90% of the time.
- Without calibration, confidence scores are meaningless.

When sources disagree, show the disagreement instead of resolving it silently.

THE RIGHT ANSWER

Keep provenance and annotate conflicts.

- Every claim carries its evidence, source, document name, publication date and source type.
- Conflicting values stay side by side, each with its source and a note.
- State the resolution and why: an audited filing outranks preliminary figures.
- Label claims as well-established, contested or single-source.

Silently picking one value
Hides potential data quality issues. Nobody downstream can evaluate the discrepancy.

Leaving out dates
Two figures from different months look like a contradiction when both are correct for their period.

Merging or reinterpreting sources
Quote or closely paraphrase the original claim, and keep the source's own caveats.

Context editing clears, compaction summarizes, the memory tool persists.

MECHANISM	WHAT HAPPENS TO OLD CONTEXT	REACH FOR IT WHEN
Context editing	Cleared: old tool results or thinking blocks are deleted	Dozens of verbose tool results, or thinking that must keep cache hits high
Compaction	Summarized into a signed block that replaces it	A long conversation must keep its narrative
Memory tool	Written to files outside the conversation	Findings must survive across sessions, not just turns
Case-facts block	Restated word for word	Facts must never be paraphrased

Complementary, not alternatives. Paired with context editing, the memory tool is what makes clearing safe.

Compaction and trimming fail silently when the client mangles history.

THE RIGHT ANSWER

Send compaction state back intact, and trim on the server.

- Threshold compaction: append the whole `response.content` every turn, not the extracted text.
- On demand: the block goes first, in place of the messages it summarized, on every later request.
- Keep history append-only. Server-side editing and compaction don't count as edits.

Appending only the text

Destroys the compaction state without an error, so you notice only deep into a long run.

Leaving summarized messages after the block

Claude gets them twice, and every request still returns 200.

Deleting old tool results client-side

On models with preserved thinking, it invalidates every later thinking block (a 400 by default).

`max_tokens` truncates blindly; a task budget lets the model pace itself.

MAX_TOKENS

A ceiling the model can't see

- Enforced, but the model is unaware of it.
- Hitting it truncates output mid-thought.

TASK BUDGET

A ceiling the model can see

- The model paces itself and lands the work.
- Counts what Claude generates plus tool results it reads this turn, not resent history.
- Too small, and Claude declines, scopes down or stops early: raise it first.

SESSION BUDGET

A hard cap on one session

- A Managed Agents feature.
- Dollar-denominated and platform-enforced, where a task budget is advisory and counts tokens.

A refusal arrives as HTTP 200, so check `stop_reason` before reading content.

- It carries `stop_reason: "refusal"` and a `stop_details.category`.
- Reading `content` blindly turns a refusal into a successful empty answer, like a failed search read as zero results.
- In production, enable server-side fallback so the request is re-routed by category, not dropped.
- Refusals are a first-class path, not just a flagship issue, and they count against rate limits.

An agent is processing a customer support request. After context compression, the customer's order number ($149.99 order #67890) was summarized as "a recent order." What should have been done to prevent this?

A. Increase the context window size

B. Extract critical transactional data into a persistent case facts block

C. Disable context compression

D. Repeat the order details in every message

B

Extract critical transactional data into a persistent case facts block

WHY

Extracting transactional facts (amounts, order numbers, dates) into a structured block that persists through compression prevents loss of critical details.

A Increase the context window size

C Disable context compression

D Repeat the order details in every message

A customer says "I AM SO ANGRY RIGHT NOW!!!" about a simple billing charge of $5. Should the agent escalate to a human?

A. Yes, high sentiment indicates a complex issue

B. Yes, capital letters indicate urgency

C. No, sentiment intensity does not indicate issue complexity — the billing issue is simple and resolvable

D. No, but reduce the agent's confidence score

C

No, sentiment intensity does not indicate issue complexity — the billing issue is simple and resolvable

A Yes, high sentiment indicates a complex issue

B Yes, capital letters indicate urgency

D No, but reduce the agent's confidence score

WHY

Sentiment ≠ complexity. An angry customer with a simple $5 billing issue doesn't need human escalation. The agent should resolve the simple issue while acknowledging the customer's frustration.

An extraction system reports 97% overall accuracy. A stakeholder asks if it's ready for production. What additional information is needed?

A. The system is ready at 97%

B. Check if accuracy is consistent across all document types and fields

C. Run more documents through to increase the sample size

D. Compare against competitor systems

B

Check if accuracy is consistent across all document types and fields

A The system is ready at 97%

C Run more documents through to increase the sample size

D Compare against competitor systems

WHY

Aggregate accuracy can mask poor performance on specific document types. 97% overall might hide 45% accuracy on handwritten documents. Break down by type and field before declaring production readiness.

Two research subagents return different values for the same metric. What should the coordinator do?

A. Average the two values

B. Use the most recent value

C. Annotate the conflict with source attribution and let the user decide

D. Discard both and search again

C

Annotate the conflict with source attribution and let the user decide

A Average the two values

B Use the most recent value

D Discard both and search again

WHY

Conflicting data should be annotated with source attribution rather than silently resolved. This preserves transparency and lets downstream consumers evaluate the discrepancy.

A long-running agent enables server-side compaction. The integration appends only `response.content[0].text` to its message history each turn. What goes wrong?

A. Nothing — text is all the API needs

B. Compaction state is lost silently, because the compaction blocks in `response.content` are what the API uses to replace compacted history

C. The request fails with a 400 on the first compaction

D. Compaction triggers too early

B

Compaction state is lost silently, because the compaction blocks in `response.content` are what the API uses to replace compacted history

- **A** Nothing — text is all the API needs
- **C** The request fails with a 400 on the first compaction
- **D** Compaction triggers too early

WHY

Append the whole `response.content`, not the extracted text. Dropping the non-text blocks destroys the compaction state without raising an error, so the failure only surfaces deep into a long conversation.

An agentic workflow keeps getting truncated mid-task when it hits `max_tokens`. Which mechanism lets the model pace itself instead?

A. Raise `max_tokens` and hope for the best

B. A task budget (`output_config.task_budget`), which the model can see and plan around

C. A lower `effort` setting

D. An iteration cap in the client loop

B

A task budget (`output_config.task_budget`), which the model can see and plan around

WHY

`max_tokens` is an enforced ceiling the model is unaware of; a task budget injects a countdown the model sees during generation, so it finishes gracefully. Lower `effort` reduces spend but does not communicate a ceiling, and a client-side iteration cap is the anti-pattern from Domain 1 §1.1.

A Raise `max_tokens` and hope for the best

C A lower `effort` setting

D An iteration cap in the client loop

A custom harness keeps context small by deleting old tool results from the middle of the `messages` array before each request. It worked on Claude Opus 5. After moving to Claude Fable 5.1, requests start failing with a 400 that mentions thinking blocks. What is the correct fix?

A Strip all `thinking` blocks from every request

B Keep the history append-only and move the trimming to server-side context editing (`clear_tool_uses_20250919`) or compaction, which don't count as edits

C Switch `thinking` to `{"type": "disabled"}`

D Delete the tool results *and* the thinking block right after each one

B

Keep the history append-only and move the trimming to server-side context editing (`clear_tool_uses_20250919`) or compaction, which don't count as edits

A Strip all `thinking` blocks from every request

C Switch `thinking` to `{"type": "disabled"}`

D Delete the tool results *and* the thinking block right after each one

WHY

Fable 5.1's thinking blocks are valid only against the exact prefix that preceded them, so any client-side edit to earlier turns invalidates every later block. Server-side context editing and compaction are exempt because the check compares the conversation as you sent it. Stripping every thinking block always works but throws away the reasoning and restarts the prompt cache on every request; disabling thinking is a 400 on Fable 5.1 and Opus 5.5 (thinking is always on); deleting the tool results with only the thinking block right after each one still leaves the later thinking blocks behind an edited prefix, so the 400 stays.

An application uses on-demand compaction. It puts the returned `compaction` block first in `messages`, but leaves the messages the block summarized in place after it. Requests keep returning 200. What is happening?

A The request should have returned `compaction_block_misplaced`; the API is being lenient

B Nothing was saved — the API sends the summarized messages to Claude again after the summary, and no error is raised

C The API deduplicates the summarized messages against the block automatically

D The block is dropped because its signature no longer matches the messages

B

Nothing was saved — the API sends the summarized messages to Claude again after the summary, and no error is raised

A The request should have returned `compaction_block_misplaced`; the API is being lenient

C The API deduplicates the summarized messages against the block automatically

D The block is dropped because its signature no longer matches the messages

WHY

After the swap, the block stands in for the messages it summarized, and the API passes every message after it to Claude unchanged, so leaving them in place re-sends them with no error. `compaction_block_misplaced` fires only when summarized messages remain *in front of* the block. Leaving the block out of a later request is the other silent mistake: Claude simply gets no summary. Signatures are checked, but only against edits to the block itself.

Context Management & Reliability

15%

5.1 Context

Exact numbers go in a case-facts block. Key facts at the edges, not the middle.

5.2 Escalation

Escalate on a human request, a policy gap or a stall. Hand off a structured summary.

5.3 Errors

Say what failed and if it's retryable. “Couldn't search” ≠ “found nothing.”

5.4 Codebases

Scratchpad files survive compression. Subagents return summaries.

5.5 Human review

Stratify by risk. Calibrate per-field confidence. Break accuracy down by type.

5.6 Provenance

Every claim keeps its source and date. Annotate conflicts; never pick silently.

5.7 Server-side

Editing clears, compaction summarizes. Append the whole `response.content`.

5.8 Long runs

A task budget is visible, `max_tokens` isn't. A refusal is a 200: check `stop_reason`.

★ Most tested

Sentiment ≠ complexity. Escalate on policy complexity, not an angry tone.

BEFORE EXAM DAY

Every domain rewards the same instincts.

MECHANISMS OVER WORDS

Decide on signals, not prose

Loops end on `stop_reason`. Critical rules live in hooks. Escalation follows an explicit request, not sentiment or self-reported confidence.

VISIBLE FAILURES

Errors carry structure

Return the category, whether it's retryable, and partial results. Never generic, never silent, never empty results passed off as success.

EXPLICIT CONTEXT

Nothing is inherited

Subagent prompts carry everything explicitly. State that must persist goes in the memory tool or files, not the conversation.

FRESH EYES

Review outside the session

Same-session self-review keeps its reasoning bias. Use multi-pass review for large PRs and break accuracy down per document type.

RIGHT-SIZED

Scale to the job

Plan mode for multi-file or architectural work, batch for latency-tolerant jobs, tool search once an agent passes about ten tools.

THE EXAM ITSELF

Prepare all six scenarios

Each exam draws 4 of 6 at random, and there is no penalty for guessing, so answer every question.

LAST REFRESH 14 OCTOBER 2026

Good luck on exam day.

Read online · dnacenta.github.io/claude-certified-architect

Full text · guide_en.pdf

Source · github.com/dnacenta/claude-certified-architect

Unofficial community study guide. Claude Certified Architect is a certification program by Anthropic.