Claude Certified Architect – Foundations

Unofficial community study guide – https://github.com/dnacenta/claude-certified-architect

Claude Certified Architect – Foundations

Anthropic's technical certification for solution architects building production applications with Claude (exam code **CCAR-F**). 60 multiple-choice, scenario-based questions in 120 minutes, proctored and closed-book. Passing score: 720/1000. Candidates answer questions from 4 of 6 randomly selected scenarios. No penalty for guessing.

Part of a four-credential family: Claude Certified Associate – Foundations (CCAO-F), **Architect – Foundations (CCAR-F)** – this guide, Architect – Professional (CCAR-P), and Developer – Foundations (CCDV-F). Delivered via Pearson VUE (OnVUE online or test center), registered through the Anthropic Partner Academy. Per Exam Guide v1.0 (effective July 2026): $125 per attempt, certification valid 12 months, up to 4 attempts per rolling 12 months (14/30/90-day waits after attempts 1–3). Renewal is free when done on time – review what changed since you certified and pass a non-proctored assessment; let it lapse and the full exam fee applies again. Access remains gated to the Claude Partner Network (registration needs a partner-domain email); the earlier "$99, first 5,000 partner employees free" beta terms are superseded.

Target: Solution architect with 6+ months experience with Claude APIs, Agent SDK, Claude Code, and MCP.

Current Model Lineup (September 2026)

The exam is model-agnostic, but scenario questions and code samples assume a current lineup. This guide's samples use `claude-opus-5`.

Model	ID	Context	Max output	Price (in / out per MTok)	Notes
Claude Fable 5.1	`claude-fable-5-1`	1M	128k	$10 / $50 (cache reads $0.25)	Most capable widely released model (2026-09-01); thinking always on; **no forced** `tool_choice`; thinking blocks bound to model + history
Claude Opus 5	`claude-opus-5`	1M	128k	$5 / $25	**Default for complex agentic coding and enterprise work**
Claude Sonnet 5	`claude-sonnet-5`	1M	128k	$2 / $10	Best speed/intelligence balance; no mid-conversation system messages, no task budgets
Claude Haiku 4.5	`claude-haiku-4-5`	200k	64k	$1 / $5	Fastest; only current model still using `budget_tokens` thinking; no `effort`

Claude Mythos 5.1 (`claude-mythos-5-1`) shares Fable 5.1's specs and is invitation-only under Project Glasswing, for defensive cybersecurity work; unlike Mythos 5 it runs safety classifiers, so `refusal` handling applies there too. **Legacy but still available:** Fable 5, Opus 4.8, Opus 4.7, Opus 4.6, Opus 4.5, Sonnet 4.6, Sonnet 4.5. Opus 4.1 retired on 2026-08-05. Anthropic now publishes retirement floors: Fable 5.1 not before 2027-09-01, Opus 5 not before 2027-07-24, Sonnet 5 not before 2027-06-30, Haiku 4.5 not before 2026-10-15.

Model IDs from the 4.6 generation onward are **dateless but still pinned snapshots**, not evergreen pointers. Query `client.models.list()` / `.retrieve(id)` for live `max_input_tokens`, `max_tokens`, and `capabilities` rather than hardcoding a table like this one.

What Fable 5.1 changes for architects (the same API surface as Fable 5, plus three breaking changes – details in Domain 2 §2.3, Domain 4 §4.7, Domain 5 §5.7):

1. `tool_choice: "any"` and `{"type": "tool", ...}` return **400**. Use `auto` plus an explicit instruction, `strict: true`, or structured outputs.
2. Thinking blocks are readable only by the model that produced them or a newer one — a fallback or router switch to an older model silently drops them (unbilled) and that model re-plans.
3. Editing, reordering, or removing earlier turns invalidates every later thinking block. Harnesses must be **append-only**: freeze `system` and `tools`, move mid-session changes into `role: "system"` messages, and trim context server-side (compaction / context editing) rather than on the client.

Fable 5.1 also requires 30-day data retention (no zero-data-retention orgs unless authorized) and is excluded from Priority Tier, as are Opus 5 and Sonnet 5.

> **Docs (since 2026-07):** Anthropic split its documentation. API docs live at `platform.claude.com/docs/en/*`, Claude Code docs at `code.claude.com/docs/en/*`; the old `docs.anthropic.com` URLs still redirect. The SDK was renamed from "Claude Code SDK" to the **Claude Agent SDK** (`pip install claude-agent-sdk`, `npm install @anthropic-ai/claude-agent-sdk`).
>
> **Program (Exam Guide v1.0, effective July 2026):** four credentials (CCAO-F, CCAR-F, CCAR-P, CCDV-F), proctored via Pearson VUE, registered through the Anthropic Partner Academy. $125 for CCAR-F, 12-month validity — superseding the launch-period "$99 / first 5,000 partner employees free" terms.
>
> Refresh history: CHANGELOG.md.

Exam Domains

Domain	Weight	Deep Dive
Agentic Architecture & Orchestration	27%	domains/d1-agentic-architecture.md
Tool Design & MCP Integration	18%	domains/d2-tool-design-mcp.md
Claude Code Configuration & Workflows	20%	domains/d3-claude-code-config.md
Prompt Engineering & Structured Output	20%	domains/d4-prompt-engineering.md
Context Management & Reliability	15%	domains/d5-context-reliability.md

---

The 6 Exam Scenarios

1. **Customer Support Resolution Agent** — Agent SDK, MCP tools, escalation logic
2. **Code Generation with Claude Code** — CLAUDE.md, plan mode, slash commands
3. **Multi-Agent Research System** — Coordinator-subagent patterns, error handling
4. **Developer Productivity with Claude** — Built-in tools, MCP servers, codebase exploration
5. **Claude Code for CI/CD** — Structured output, batch API, multi-pass review
6. **Structured Data Extraction** — JSON schemas, tool_use, validation-retry

---

The 10 Critical Anti-Patterns

These show up as wrong answers on the exam:

1. Parsing natural language for loop termination instead of checking `stop_reason`

2. Arbitrary iteration caps as primary stopping mechanism
3. Prompt-based enforcement for critical business rules (should use hooks)
4. Self-reported confidence scores for escalation decisions
5. Sentiment-based escalation (sentiment ≠ complexity)
6. Generic error messages hiding diagnostic context
7. Silently suppressing errors (returning empty results as success)
8. Too many tools per agent (18 vs 4-5 recommended) — see Domain 2 §2.3 for how tool search changes this in the current API
9. Same-session self-review (retains reasoning context bias)
10. Aggregate accuracy metrics masking per-document-type failures

---

Key Decision Frameworks

Decision	Option A	Option B
Enforcement	**Programmatic hooks** — financial/safety	**Prompt-based** — best-effort
Execution mode	**Plan mode** — multi-file, architectural	**Direct** — single-file, obvious
tool_choice	**"any"** — guarantee tool call	**Forced** — specific tool first. *Both return 400 on Fable 5.1 / Mythos 5.1* — there, `auto` + instruction + `strict: true`, or `output_config.format`
API type	**Synchronous** — blocking workflows	**Batch** — latency-tolerant
Error handling	**Structured context** — category, retry, partial	Never generic or silent
Escalation	**Immediate** — explicit customer request	**Resolve first** — within capability
Code review	**Multi-pass** — large PRs	**Single-pass** — small, focused
Context passing	**Always explicit** in subagent prompts	Never rely on auto-inheritance
Thinking	**Adaptive** (`{"type": "adaptive"}`) on every current model	`budget_tokens` is removed — 400 on Fable 5.x / Opus 5 / Sonnet 5 / 4.7 / 4.8
Effort	`high` (default) then sweep — `xhigh` for the hardest coding/agentic work on Opus 5 / Fable 5.1, and the recommended start on Opus 4.8 / 4.7	`low` / `medium` — subagents, mechanical and routine tasks
Tool exposure	**Load upfront** — under ~10 tools	**Tool search +** `defer_loading` — 10+ tools or multi-server MCP
Context reduction	**Clear** (context editing) — verbose tool results	**Compact** — long conversations that need their narrative
State that must persist	**Memory tool / files**	Never rely on the conversation surviving compaction
Agent harness	**Tool Runner / Agent SDK** — you host	**Managed Agents** — Anthropic hosts loop + sandbox
Conversation history	**Append-only** — freeze `system`/`tools`, changes via `role: "system"` messages	Never edit or snip earlier turns on Fable 5.1 — it invalidates thinking blocks
Many-agent orchestration (Claude Code)	**Subagents / forks** — a few delegated tasks per turn	**Dynamic workflows** — a script the runtime executes for dozens to hundreds of agents

---

4-Week Study Plan

Assumes daily study, 1.5–2 hours per day.

Week	Focus	Topics	Daily Goal
1	Core Architecture	Agentic loops, `stop_reason`, multi-agent orchestration, hooks, session management, tool design, MCP config	1 domain section + practice questions
2	Applied Skills	CLAUDE.md hierarchy, plan mode, CI/CD, Batches API, explicit criteria, few-shot, validation-retry, structured output	1 domain section + build a small project using each concept
3	Reliability + Hands-On	Context management, escalation, error propagation, provenance, hands-on exercises across all domains	Review weak areas + build an end-to-end scenario
4	Exam Prep	Practice exams, anti-pattern drills, scenario walkthroughs, timed question sets	1 full practice exam per day + targeted review

---

Resources

- Official CCAR-F Exam Guide (PDF, v1.0) — the March 2026 launch guide is superseded
- Anthropic Partner Academy — CCAR-F registration · Prep courses · Certification FAQ (renewal, retakes, eligibility)
- Pearson VUE — Anthropic certification program
- Anthropic Skilljar — Building with Claude API
- 12-Week Training Program (GitHub)
- Building Effective Agents (Anthropic Research)
- Claude Partner Network

API (platform.claude.com)

- Tool use overview
- Tool search tool
- Programmatic tool calling
- Memory tool
- MCP connector
- Thinking · Preserved thinking · Effort · Task budgets
- Mid-conversation system messages
- Context editing · Compaction
- Prompt caching · Cache diagnostics
- Structured outputs
- Stop reasons and fallback · Refusals and fallback
- Managed Agents overview · `ant` CLI · Admin API
- Models overview · Claude Fable 5.1 migration guide · Model deprecations · Pricing

Claude Code (code.claude.com)

- Overview · CLI reference
- Subagents · Dynamic workflows · Agent teams · Skills · Plugins · Hooks
- MCP · Memory and CLAUDE.md · Routines · Commands · Changelog
- Claude Agent SDK overview

Background reading

- MCP Introduction
- Advanced tool use (Anthropic Engineering)

- Effective context engineering for AI agents

Domain 1: Agentic Architecture & Orchestration (27%)

The heaviest domain on the exam. Covers 7 task statements focused on how agentic systems are built, how loops operate, how agents coordinate, and how enforcement works.

1.1 The Agentic Loop Lifecycle

At its core, an agentic system is a loop. Claude receives a prompt with tools, decides whether to call a tool or respond, and the loop continues until Claude signals it's done.

How the Loop Works

```
User sends prompt with tool definitions
         ↓
    Claude responds
         ↓
  Check stop_reason
         ↓
┌──────────────────────────────────────┐
│ "tool_use"     → Execute tool        │
│                 → Return tool_result │
│                 → Send next request  │
│                 → Loop continues     │
│                                      │
│ "end_turn"     → Claude is done      │
│                 → Present final text │
│                 → Loop ends          │
│                                      │
│ "stop_sequence"→ Custom stop matched │
│                 → Loop ends          │
│                                      │
│ "pause_turn"   → Server-side tool    │
│                   loop hit its cap   │
│                 → Send response back │
│                 → Loop continues     │
│                                      │
│ "max_tokens"   → Hit token limit     │
│                 → May need continue  │
│                                      │
│ "refusal"      → Declined on safety  │
│                 → Inspect stop_details│
│                 → Loop ends          │
│                                      │
│ "model_context_window_exceeded"      │
│                → Hit context window  │
│                → Valid but truncated │
└──────────────────────────────────────┘
```

The `stop_reason` Values

Value	Meaning	Action
`"tool_use"`	Claude wants to call a tool	Execute it, return result, continue loop
`"end_turn"`	Claude has finished reasoning	Present the final text response
`"stop_sequence"`	Output matched a configured `stop_sequences` string	Loop ends; inspect which sequence was hit
`"pause_turn"`	Server-side tool loop (e.g., web_search) hit its internal iteration limit	Send the response back to continue
`"max_tokens"`	Response hit the `max_tokens` limit	May need to request continuation
`"refusal"`	Claude declined to respond on safety grounds	Loop ends; inspect `stop_details` (see below), rephrase or route the request
`"model_context_window_exceeded"`	Generation hit the model's context window before `max_tokens`	Response is valid but truncated; trim input or continue. Default in Sonnet 4.5+

> **`stop_details` on refusals (Opus 4.7+):** A `refusal` response also carries a `stop_details` object (no beta header needed). `stop_details.type` is always `"refusal"`; `stop_details.category` is the policy category (e.g. `"cyber"`, `"bio"`, `"reasoning_extraction"`, `"frontier_llm"`, or `null` — the set has grown over time); `stop_details.explanation` is a human-readable string (don't parse it). `stop_details` is `null` for every other stop reason. Use the category to route or log specific refusals differently.

Server-Side Refusal Fallbacks

A `refusal` arrives as **HTTP 200**, not an exception — code that reads `content` without checking `stop_reason` first will silently process an empty or partial response. For production agents on Claude Opus 5, Fable 5 / 5.1, and Mythos 5.1 (which, unlike Mythos 5, runs safety classifiers), don't just log the refusal: opt into server-side fallback so the request is re-routed automatically by refusal category.

```
response = client.messages.create(
    model="claude-opus-5",
    max_tokens=16000,
    betas=["server-side-fallback-2026-07-01"],
    fallbacks="default",          # server routes by refusal category
    messages=[...],
)
```

`fallbacks="default"` means you never maintain a model list. The older array form (`betas=["server-side-fallback-2026-06-01"]` + `fallbacks=[{"model": "claude-opus-4-8"}]`) still works. Server-side fallback is **Claude API only** — on Bedrock, Vertex, and Foundry use the SDKs' client-side `BetaRefusalFallbackMiddleware` instead.

One Fable 5.1 consequence: a fallback lands the conversation on an *older* model, which cannot read Fable 5.1's thinking blocks. The API drops them (unbilled), the request succeeds, and the fallback model re-plans without that reasoning — expect a slower, costlier first turn after the switch. Send the `thinking-binding-controls-2026-08-01` beta header to get an `input_transformations` list of what was dropped.

Loop rule: always branch on `stop_reason` *before* reading `content`. `refusal` is a terminal stop reason for that request, not a retryable tool error. And on Fable 5.1, keep the `messages` array **append-only** — editing or snipping earlier turns invalidates every later thinking block (Domain 5 §5.7).

API Response Structure

When Claude wants to use a tool, the response contains both text and a tool_use block:

```
{
  "id": "msg_01Aq9w938a90dw8q",
  "model": "claude-opus-5",
  "stop_reason": "tool_use",
  "role": "assistant",
  "content": [
    {
      "type": "text",
      "text": "I'll check the current weather in San Francisco for you."
    },
    {
      "type": "tool_use",
      "id": "toolu_01A09q90qw90lq917835lq9",
      "name": "get_weather",
      "input": { "location": "San Francisco, CA", "unit": "celsius" }
    }
  ]
}
```

The `content` array can contain multiple blocks — Claude may explain what it's doing (text) and call a tool (tool_use) in the same response. It can also call multiple tools simultaneously by including multiple tool_use blocks.

Returning Tool Results

Tool results must reference the exact `tool_use_id` from Claude's request:

```
{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
      "content": "15 degrees celsius, partly cloudy"
    }
  ]
}
```

For multiple simultaneous tool calls, return multiple `tool_result` blocks in the same message, each matching its `tool_use_id`.

Error Results

When a tool execution fails:

```
{
  "type": "tool_result",
  "tool_use_id": "toolu_01A09q90qw90lq917835lq9",
  "is_error": true,
  "content": "API timeout after 30 seconds"
}
```

Setting `is_error: true` helps Claude understand the tool failed and reason about recovery strategies.

Model-Driven Decision Making

The exam tests that you understand: **Claude decides** which tool to call and when to stop. You don't pre-configure decision trees or fixed tool sequences. The model reasons about what information it needs, selects the appropriate tool, processes the result, and decides the next step.

This is fundamentally different from traditional workflow automation where steps are predetermined.

Anti-Patterns (These Are Wrong Answers)

1. **Parsing natural language** for loop termination — "If Claude says 'I'm done', exit the loop." Wrong. Always check `stop_reason`.
2. **Arbitrary iteration caps** as primary stopping — "Run max 5 loops then stop." Wrong as a primary mechanism. You may use safety caps, but `stop_reason` drives the loop.
3. **Checking for text content** as completion — "If the response has text, it's done." Wrong. Responses can contain both text and tool_use blocks.

> **Beyond the exam guide — task budgets (beta):** the modern answer to "how do I bound an agentic loop without a hard iteration cap" is a **task budget**. `output_config.task_budget` gives Claude a token ceiling it can *see*, so it paces itself and finishes gracefully instead of being cut off mid-work. This is different from `max_tokens`, which is an enforced per-response ceiling the model is unaware of.
>
> ```
> with client.beta.messages.stream(
>     model="claude-opus-5", max_tokens=128000,
>     betas=["task-budgets-2026-03-13"],
>     output_config={"effort": "high",
>                    "task_budget": {"type": "tokens", "total": 64000}},
>     messages=[...], tools=[...],
> ) as stream:
>     response = stream.get_final_message()
> ```
>
> Minimum `total` is 20,000. Available on Claude Fable 5.1 / Mythos 5.1, Fable 5 / Mythos 5, Opus 5, and Opus 4.8/4.7 — **not** on Sonnet 5, Opus 4.6, or Haiku 4.5. Stream it — a large `max_tokens` on a non-streaming request hits HTTP timeouts. Size the budget against your real task-length distribution: a budget that is obviously too small makes Claude decline, scope down, or stop early with a partial result, which looks like a refusal but isn't one. The exam's anti-pattern still holds: a budget paces the loop, `stop_reason` still terminates it.

Who Runs the Loop: Four Ways to Build an Agent

Two independent questions separate the options: **who supplies the harness** (the loop plus context management) and **who supplies the deployment** (the infrastructure it runs on). Tool Runner and the Claude Agent SDK are easy to conflate because both supply a harness only — you still host them.

Approach	You write	Harness & deployment	Tools available
Manual loop (`client.messages.create`)	The `while stop_reason == "tool_use"` loop	You build the harness; you host	Only tools you define
Tool Runner (`client.beta.messages.tool_runner`)	Just the tool functions	SDK supplies the loop; you host	Only tools you define
Managed Agents (beta, `/v1/agents` + `/v1/sessions`)	Agent config + your tool results	Anthropic supplies the loop **and** hosts a per-session sandbox	Hosted bash / files / code execution, plus Skills, MCP, and your tools
Claude Agent SDK (`claude-agent-sdk`)	A prompt + options	Claude Code harness + built-in tools; you host	Read/Write/Edit/Bash/Glob/Grep/WebSearch/WebFetch + MCP + subagents

Tool Runner ≠ Claude Agent SDK. Tool Runner ships inside the regular Anthropic SDK (`anthropic` / `@anthropic-ai/sdk`) and only loops over tools *you* define — no built-in tools, no filesystem, no sandbox. It exposes per-turn hooks for approval gates, error interception, result modification (e.g. attaching `cache_control`), and retries. The Claude Agent SDK is Claude Code packaged as a library.

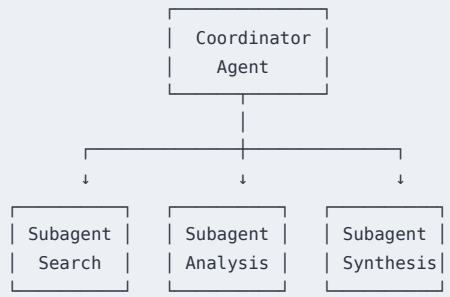
Managed Agents (beta) is the newest surface and the only one that adds managed *deployment*. The mandatory flow is Agent (created once, versioned, persisted) → Session (one per run). `model`, `system`, and `tools` live on the **agent**, never the session. Each session provisions a container that acts as the agent's workspace and streams events back; you send messages and tool results in. It also adds scheduled deployments (cron-fired sessions), vault-stored credentials substituted at egress, dollar-denominated session budgets, and multiagent rosters. Beta header: `managed-agents-2026-04-01`. Not available on Bedrock / Vertex / Foundry — use Claude API + tool use there. The recommended control-plane flow is now the `ant` **CLI**: define agents and environments as version-controlled YAML (`ant beta:agents create < agent.yaml`), and let application code own only the data plane (`sessions.create` with the stored agent ID). `ant auth login` also gives the SDKs an OAuth profile, so a bare `Anthropic()` client works with no API key in the environment.

Choosing: stay at the simplest tier that works. A single call or a code-controlled workflow handles most tasks; reach for an agent only when the task is genuinely open-ended and model-driven, the value justifies the latency and cost, and errors are catchable (tests, review, rollback).

1.2 Multi-Agent Orchestration

Hub-and-Spoke Architecture

The standard pattern for multi-agent systems is hub-and-spoke: one coordinator agent manages multiple specialized subagents.



Coordinator Responsibilities

The coordinator:

- **Decomposes** the task into subtasks
- **Delegates** to appropriate subagents
- **Routes** all inter-subagent communication (subagents never talk to each other directly)
- **Aggregates** results from subagents
- **Handles errors** from subagent failures
- **Evaluates** synthesis quality and re-delegates if gaps exist

Subagent Isolation

Critical concept: Subagents have isolated context. They do NOT inherit the coordinator's conversation history. Everything a subagent needs must be explicitly passed in its prompt.

This means:

- The coordinator must include all relevant context in the subagent's task description
- Subagents can't reference earlier parts of the main conversation
- Each subagent starts fresh with only what the coordinator provides

Dynamic Subagent Selection

Not every query needs every subagent. The coordinator should:

- Assess what the query actually requires
- Only invoke relevant subagents
- Not route everything through a full pipeline

Example: A research query about a single topic doesn't need web search + document analysis + synthesis if a single focused search would suffice.

Decomposition Risks

Overly narrow decomposition: The exam specifically tests this. Example scenario: coordinator is asked to research "creative industries" and decomposes it into only visual arts subtasks, missing music, writing, film, and design.

The fix: broader initial decomposition with explicit coverage checks.

Iterative Refinement Loops

The coordinator evaluates synthesis output and identifies gaps:

```
Coordinator → Subagents (round 1)
         ↓
   Evaluate results
         ↓
   Identify gaps
         ↓
Coordinator → Subagents (round 2, targeted)
         ↓
   Final synthesis
```

1.3 Subagent Spawning and Context Passing

The Agent Tool

Subagents are spawned with the `Agent` tool — the canonical name in both Claude Code and the Claude Agent SDK. The coordinator must have `Agent` in its allowed tools to spawn subagents, and access can be narrowed to specific subagent types with `Agent(worker, researcher)`.

> **Naming note:** older material (including the exam guide's wording) calls this the **Task** tool. `Task` was Claude Code's original name for the same mechanism; current Claude Code and SDK docs use `Agent`. If an exam item says "Task tool", it means this.
>
> **SDK naming:** the SDK was renamed from "Claude Code SDK" to the **Claude Agent SDK**. Python: `pip install claude-agent-sdk`. TypeScript: `npm install @anthropic-ai/claude-agent-sdk`. Top-level entry point: `query()` + `ClaudeAgentOptions`.

Context Must Be Explicit

This is tested heavily: **context does not automatically inherit.** When the coordinator spawns a subagent, it must include everything the subagent needs in the prompt.

Bad:

```
"Analyze the customer's issue"
# Subagent has no idea who the customer is or what the issue is
```

Good:

```
"Customer #12345 reported that their order #67890 (placed 2024-01-15,
total $149.99) arrived damaged. They are requesting a full refund.
Analyze the order history and return policy applicability."
```

Parallel Subagent Execution

To run subagents in parallel, emit multiple `Agent` tool calls in a **single coordinator response**. Not across separate turns — that would be sequential.

```
{
  "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" }
    }
  ]
}
```

Claude Code caps concurrent subagents at 20 (`CLAUDE_CODE_MAX_CONCURRENT_SUBAGENTS`) and spawn depth at 3 (`CLAUDE_CODE_MAX_SUBAGENT_SPAWN_DEPTH`); at the depth limit the `Agent` tool is withheld from the subagent (a fork keeps it listed, but it errors instead of spawning).

> **Beyond the exam guide — when the fan-out outgrows a turn:** Claude Code now has four orchestration units, and the exam's hub-and-spoke reasoning maps onto the first one. The difference is *who holds the plan*.
>
> | | **Subagents** | **Skills** | **Agent teams (experimental)** | **Dynamic workflows** |
> |---|---|---|---|---|
> | What it is | A worker Claude spawns | Instructions Claude follows | A lead session supervising peer sessions | A JavaScript script the runtime executes |
> | Who decides what runs next | Claude, turn by turn | Claude, following the prompt | The lead agent | The script (`agent()`, `parallel()`, `pipeline()`) |
> | Intermediate results live in | Claude's context | Claude's context | A shared task list | Script variables |
> | Scale | A few delegated tasks per turn | Same | A handful of long-running peers | Dozens to hundreds of agents per run, resumable |
>
> A workflow moves the coordinator's plan out of the model and into code — the deterministic answer to "the coordinator forgot half the subtasks" (§1.2). Claude writes the script; `/workflows` watches it; `ultracode` (an effort setting) lets Claude decide when to reach for one. Agent teams need `CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS=1`.

AgentDefinition in the Claude Agent SDK

```python
from claude_agent_sdk import query, ClaudeAgentOptions, AgentDefinition

async for message in query(
    prompt="Use the code-reviewer agent to review this codebase",
    options=ClaudeAgentOptions(
        allowed_tools=["Read", "Glob", "Grep", "Agent"],
        agents={
            "code-reviewer": AgentDefinition(
                description="Expert code reviewer for Python and TypeScript",
                prompt="Analyze code quality, identify bugs, suggest improvements.",
                tools=["Read", "Glob", "Grep"],
            ),
            "test-writer": AgentDefinition(
                description="Test generation specialist",
                prompt="Write comprehensive unit tests for the provided code.",
                tools=["Read", "Write", "Bash"],
            ),
        },
    ),
):
    print(message)
```

Messages emitted from inside a subagent's context carry a `parent_tool_use_id` field — use it to attribute messages to the right subagent execution.

Forks vs Fresh Subagents

A **fork** is the opposite trade-off from an isolated subagent: it inherits the *entire* conversation — history, system prompt, tools, model — and shares the prompt cache, so it is cheaper than a fresh subagent. Only its final result returns to the main conversation; its tool calls stay isolated. Forks can't spawn sub-forks.

	Fresh subagent	**Fork**
Conversation history	Not inherited — pass everything explicitly	Fully inherited
System prompt / tools	The subagent's own	Same as the parent session
Prompt cache	Cold	Shared with the parent (cheaper)
Use for	Bounded, well-specified tasks; noisy exploration	Divergent branches off a shared baseline

In Claude Code, Claude requests a fork through the `Agent` tool with `subagent_type: "fork"` (fork mode is on by default in interactive sessions, and subagents it spawns then run in the background). You can start one yourself with `/subtask <prompt>`. Two neighbours are easy to confuse: `/fork` now *copies* the whole session into a separate background session (agent view), and `/branch` switches you into a copy of the conversation to try a different direction. The SDK/CLI also exposes `--fork-session` for resuming a session under a new ID; `fork_session` is the older API-level name for the same "branch from a shared baseline" idea.

Structured Context Passing

Best practice: separate content from metadata when passing context between agents.

```
{
  "findings": [
    {
      "claim": "Revenue increased 15% YoY",
      "evidence": "Q4 earnings report, page 3",
      "source_url": "https://example.com/earnings",
      "document_name": "Q4 2024 Earnings Report",
      "publication_date": "2025-01-15"
    }
  ]
}
```

This preserves attribution and allows the receiving agent to evaluate source quality.

1.4 Workflow Enforcement and Handoff

The Critical Distinction

This is one of the most heavily tested concepts on the exam:

Enforcement Type	When to Use	Guarantee Level
Programmatic (hooks, prerequisites)	Financial/safety consequences	Deterministic — 100% enforced
Prompt-based (instructions)	Best-effort acceptable	Probabilistic — non-zero failure rate

Example: Financial Operations

Scenario: A customer support agent must verify identity before processing refunds.

Wrong approach (prompt-based):

```
"Always verify the customer's identity before processing any refund."
```

This will work most of the time, but has a non-zero failure rate. For financial operations, "most of the time" is not acceptable.

Correct approach (programmatic): A PreToolUse hook on `process_refund` that checks whether `get_customer` has been called and returned a verified customer ID. If not, the hook blocks the refund with exit code 2.

Handoff to Humans

When an agent escalates to a human, the human agent typically does NOT have access to the full conversation transcript. The handoff must include a structured summary:

```
Customer: Jane Doe (#12345)
Issue: Damaged product received
Order: #67890, $149.99, placed 2024-01-15
Root cause: Shipping damage
Actions taken: Verified order, confirmed damage via photos
Recommended action: Full refund + replacement shipment
Refund amount: $149.99
```

1.5 Claude Code / Agent SDK Hooks

Hook Events (Current Catalog)

The hook system expanded substantially in 2026 — the current catalog is **33 events** across eight groups. Exam-relevant ones are marked ⋆.

Group	Events
Setup	`Setup` (runs once per session)
Session & turn	`SessionStart` ⋆, `SessionEnd` ⋆, `UserPromptSubmit` ⋆, `UserPromptExpansion`, `Stop` ⋆, `StopFailure`
Tool / agentic loop	`PreToolUse` ⋆, `PostToolUse` ⋆, `PostToolUseFailure`, `PostToolBatch`, `PermissionRequest` ⋆, `PermissionDenied`
Agent & task	`SubagentStart`, `SubagentStop` ⋆, `TaskCreated`, `TaskCompleted`, `TeammateIdle`
File & config	`FileChanged`, `CwdChanged`, `DirectoryAdded`, `ConfigChange`, `InstructionsLoaded`
Compaction	`PreCompact` ⋆, `PostCompact`
Model	`PreModelSwitch`, `PostModelSwitch` (block, confirm, or annotate a model change — new in v2.1.251)
Context & worktree	`Notification`, `MessageDisplay`, `Elicitation`, `ElicitationResult`, `WorktreeCreate`, `WorktreeRemove`

The rough lifecycle you should carry into the exam:

```
SessionStart
  → UserPromptSubmit
    → [Agentic Loop]:
        PreToolUse
          → PermissionRequest  (may fire PermissionDenied)
            → PostToolUse  /  PostToolUseFailure
              → SubagentStart / SubagentStop
                → TaskCreated / TaskCompleted
    → Stop  /  StopFailure
  → PreCompact → PostCompact
  → PreModelSwitch → PostModelSwitch   (only when the model changes)
→ SessionEnd
```

Hook Types

Five types are supported. `mcp_tool` is new in 2026.

Type	How It Works	Use Case
`command`	Runs a shell command. Exit 0 = pass, exit 2 = block	File validation, linting gates
`http`	POST to a URL endpoint	External audit logging, webhooks
`mcp_tool`	Calls a tool on a configured MCP server	Structured validation via a reusable service
`prompt`	Single LLM call that returns a yes/no decision	Context-dependent approval
`agent`	Multi-turn subagent with tool access	Complex compliance checks

PreToolUse Hooks (Current JSON Shape)

Each matcher holds an **array** of `hooks` (the old singular `hook` field is gone). Hooks can be `async`, gated on `if` permission rules, and carry a `statusMessage` shown in the UI.

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "process_refund",
        "hooks": [
          {
            "type": "command",
            "if": "Bash(git *)",
            "command": "\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/check-refund-authorization.sh",
            "timeout": 600,
            "statusMessage": "Verifying refund authorization...",
            "shell": "bash"
          },
          {
            "type": "mcp_tool",
            "server": "compliance",
            "tool": "check_refund_policy",
            "input": { "amount": "${tool_input.amount}" },
            "timeout": 60
          }
        ]
      }
    ]
  },
  "disableAllHooks": false
}
```

Decision control output (unchanged):

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

`permissionDecision` values: `"allow"`, `"deny"`, `"ask"`.

PostToolUse Hooks

Intercept tool results before Claude processes them. Useful for:

- Normalizing timestamps, date formats, status codes
- Scrubbing sensitive data from results
- Adding metadata to results

If the tool itself errored, `PostToolUseFailure` fires instead — handle retries and diagnostics there.

Hooks in Skills and Subagents

Hooks can now be scoped to a **skill's lifecycle** via the `hooks:` frontmatter field in `SKILL.md`, and subagents can carry their own hook definitions. This lets you enforce rules locally (e.g., a `commit` skill enforces `cargo fmt` before `git commit`) without polluting project-wide settings.

Hook Configuration Locations

Location	Scope	Shared?
`~/.claude/settings.json`	All projects for this user	No (personal)
`.claude/settings.json`	This project	Yes (version control)
`.claude/settings.local.json`	This project, this machine	No (gitignored)
Managed policy settings	Organization-wide	Yes (admin-managed)

1.6 Task Decomposition Strategies

Prompt Chaining (Fixed Sequential)

For predictable, multi-aspect tasks where you know the steps upfront:

```
Step 1: Analyze each file individually (local analysis)
Step 2: Cross-file integration pass (data flow, consistency)
Step 3: Generate final report
```

Each step's output feeds into the next. The sequence is predetermined.

Dynamic Adaptive Decomposition

For open-ended investigation where subtasks emerge from discoveries:

```
Initial query: "Why is the app slow?"
  → Profile CPU usage
    → Discovery: database queries are the bottleneck
      → Analyze query patterns
        → Discovery: N+1 queries in the user listing endpoint
          → Generate fix for specific queries
```

The subtasks are generated based on what each step reveals.

Multi-Pass Review Architecture

For code review of large PRs:

Pass 1 — Per-file local analysis:

- Each file reviewed independently
- Consistent depth across all files
- Catches local issues (bugs, style, types)

Pass 2 — Cross-file integration:

- Separate pass looking at how files interact
- Catches data flow issues, interface mismatches, cross-file consistency
- Different prompt focused on integration concerns

Why two passes? Single-pass review of large PRs leads to:

- Attention dilution (model focuses on early files, skimps on later ones)
- Contradictory findings (inconsistent standards across files)
- Missed integration issues (can't see cross-file problems when reviewing one file)

1.7 Session Management

Resuming Sessions

`--resume <session-name>` continues a named prior conversation with full context.

When to resume:

- Ongoing investigation where tool results are still valid
- Iterative development within the same session

When to start fresh:

- Tool results are stale (files changed, services restarted)
- Context has drifted too far from current task
- In these cases, pass a structured summary of prior findings rather than resuming

Forking a Session

`--fork-session` resumes an existing session under a **new** session ID, so the original stays untouched and each branch explores a different approach without contaminating the others. In an interactive session, `/subtask <prompt>` spawns a fork that inherits the full conversation and returns only its result, `/fork` copies the session into a separate background session, and `/branch` switches you into a copy (see §1.3).

Use case: Testing two different refactoring strategies from the same starting point.

Moving a Session Between Surfaces

Sessions are no longer tied to one machine. `claude --cloud "<task>"` starts the work in a cloud session (`--remote` is a deprecated alias); `claude --teleport` pulls a web session back into the terminal; `/desktop` hands the current terminal session to the desktop app for visual diff review. Remote Control drives a running local session from a phone or another browser. Locally, `claude --bg "<task>"` starts a **background session** and returns immediately; `claude agents` opens the agent view, and `claude attach / logs / stop / respawn / rm <id>` manage it from the shell (`respawn --all` restarts every running session, e.g. to pick up a new binary). This matters architecturally: a "session" is a portable unit of state, so anything the agent must not lose belongs in files or memory, not in scrollback.

Informing Resumed Sessions

When resuming a session after file changes, you must explicitly tell Claude what changed:

```
"Since our last session, I've updated auth.py to use JWT tokens instead of
session cookies. Please re-analyze the authentication flow with these changes."
```

Don't assume the resumed session will notice changes on its own — tool results from the previous session may reflect old file contents.

Domain 1 Practice Questions

Q1: 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

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

Q2: 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

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

Q3: 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

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

Domain 2: Tool Design & MCP Integration (18%)

Covers how to design tool interfaces that LLMs can reliably use, how to handle errors from tools, how to distribute tools across agents, and how MCP servers are configured and operate.

2.1 Tool Interface Design

Tool Descriptions Are Everything

Tool descriptions are the **primary mechanism** Claude uses to decide which tool to call. The description is not just documentation — it's the selection criteria.

Bad description (leads to unreliable selection):

```
{
  "name": "search",
  "description": "Search for things"
}
```

Good description:

```
{
  "name": "search_knowledge_base",
  "description": "Search the internal knowledge base for support articles, FAQs, and troubleshooting guides. Use this whe
}
```

What to Include in Tool Descriptions

1. **Purpose** — What the tool does and when to use it
2. **Input format** — What the input should look like, with examples
3. **Output format** — What the tool returns
4. **Edge cases** — What happens with invalid inputs or no results
5. **Differentiation** — When to use THIS tool vs similar tools

The Full Tool Definition Schema

```
{
  "name": "get_weather",
  "description": "Get the current weather in a given location. Returns temperature, conditions, humidity, and wind speed.
  "input_schema": {
    "type": "object",
    "properties": {
      "location": {
        "type": "string",
        "description": "The city and state/country, e.g. 'San Francisco, CA' or 'London, UK'"
      },
      "unit": {
        "type": "string",
        "enum": ["celsius", "fahrenheit"],
        "description": "Temperature unit. Defaults to celsius if not specified."
      }
    },
    "required": ["location"]
  }
}
```

Strict Mode

Strict mode guarantees Claude's tool inputs conform exactly to the schema:

```
{
  "name": "extract_invoice",
  "strict": true,
  "input_schema": {
    "type": "object",
    "properties": {
      "vendor_name": { "type": "string" },
      "invoice_number": { "type": "string" },
      "total_amount": { "type": "number" },
      "currency": { "type": "string", "enum": ["USD", "EUR", "GBP"] }
    },
    "required": ["vendor_name", "invoice_number", "total_amount", "currency"],
    "additionalProperties": false
  }
}
```

Requirements for strict mode:

- `additionalProperties: false` must be set
- All properties must be in `required`
- Only a subset of JSON Schema is supported (see Domain 4)

Anti-Pattern: Ambiguous Tool Overlap

Having two tools with near-identical descriptions causes misrouting:

```
Tool 1: "analyze_content" — "Analyze content for insights"
Tool 2: "analyze_document" — "Analyze a document for insights"
```

Claude can't reliably distinguish between these. Fix: rename to clearly differentiate purpose, or merge into one tool.

2.2 Structured Error Responses

MCP Error Structure

When tools fail, the error response should be structured, not a generic string:

```
{
  "isError": true,
  "errorCategory": "transient",
  "isRetryable": true,
  "description": "Database connection timed out after 30 seconds. The query was for customer #12345's order history.",
  "partial_results": null,
  "attempted_action": "SELECT * FROM orders WHERE customer_id = 12345"
}
```

Error Categories

Category	Retryable?	Examples	Agent Action
Transient	Yes	Timeouts, service unavailability, rate limits	Retry with backoff
Validation	No (without input changes)	Invalid email format, missing required field	Fix input, retry
Business	No	Refund exceeds policy limit, account frozen	Escalate or inform user
Permission	No	Insufficient access, expired token	Escalate

What the Agent Should Do with Errors

1. **Transient errors:** Subagents should implement local recovery (retry with backoff). Only propagate if recovery fails.
2. **Validation errors:** Claude should examine the error, fix the input, and retry.
3. **Business errors:** Claude should explain the policy to the user or escalate.
4. **Permission errors:** Claude should escalate to a human.

Anti-Patterns (Wrong Answers)

1. **Generic error messages:** `"Operation failed"` — hides the cause and prevents intelligent recovery
2. **Silent suppression:** Returning `{ "results": [] }` when the search engine is down. This makes it look like there are no results when actually the search didn't work. The agent will confidently tell the user "I found nothing" when the truth is "I couldn't search."
3. **Generic status messages:** `"search unavailable"` without context about what was attempted
4. **Terminating on single failure:** Killing an entire multi-step workflow because one step failed, instead of using partial results or trying alternatives

Distinguishing Empty from Failed

This is a specific exam concept:

Situation	What Happened	Correct Response
Search returns `[]` with status 200	Valid query, no matches	"No results found for that query"
Search returns timeout error	Query didn't execute	"I wasn't able to search right now, let me try again"

These must be handled differently. The first is information. The second is an error.

2.3 Tool Distribution and tool_choice

The 4-5 Tool Maximum

Each agent should have **4-5 tools maximum**. Giving an agent 18 tools degrades selection reliability — Claude may pick the wrong tool, call unnecessary tools, or struggle to reason about which to use.

Solution for complex systems: Distribute tools across specialized subagents.

```
Instead of:
  One agent with 18 tools

Do:
  Coordinator with: Agent, AskUser
  Search subagent with: search_web, search_docs, search_db
  Analysis subagent with: analyze_text, extract_entities, summarize
  Action subagent with: send_email, create_ticket, update_record
```

This is the exam's answer, and the underlying principle (one agent, one focused job) is still correct. The current numbers from Anthropic's docs are worth knowing alongside it.

Beyond the Exam Guide: Tool Search

Anthropic's current published guidance is that **tool-selection accuracy degrades once an agent exceeds roughly 30–50 available tools**, and that a typical multi-server MCP setup (GitHub, Slack, Sentry, Grafana, Splunk) burns ~55k tokens of context in tool definitions before any work happens. The fix is no longer "split into subagents" alone — it's the **tool search tool**, which is GA on the Claude API.

```
"tools": [
  { "type": "tool_search_tool_regex_20251119", "name": "tool_search_tool_regex" },
  { "name": "get_weather", "description": "...", "input_schema": { ... },
    "defer_loading": true }
]
```

How it works:

1. You still send **every** tool definition on every request. `defer_loading` controls what enters the *context window*, not what you transmit.
2. Non-deferred tools load into context immediately; deferred ones don't.
3. When Claude needs something else, it calls the search tool. The API returns `tool_reference` blocks and expands them into full definitions inline — the system-prompt prefix is untouched, so **prompt caching survives**.
4. Two variants: `tool_search_tool_regex_20251119` (Claude writes Python `re.search()` patterns, max 200 chars) and `tool_search_tool_bm25_20251119` (natural-language queries, max 500 chars). Both search names, descriptions, argument names, and argument descriptions.

Rule	Detail
Never defer everything	At least one tool must have `defer_loading: false`, normally the search tool itself. Otherwise: **400, "All tools cannot be deferred"**
Keep a hot set	Leave your **3–5 most-used tools** non-deferred so Claude can call them without a search round-trip
Limits	Up to 10,000 deferred tools per request; searches return 5 matches by default (Claude may set `limit`, 1–10,000)
Cache	A tool with `defer_loading: true` can't also carry `cache_control` — 400. Put the breakpoint on a non-deferred tool
Composability	Works with `strict: true`; the grammar builds from the full toolset
MCP	Don't set `defer_loading` per tool — set it on the `mcp_toolset` entry's `default_config`

When to use it: 10+ tools, tool definitions over 10k tokens, accuracy dropping as the toolset grows, or aggregating multiple MCP servers. **When not to:** fewer than 10 tools, every tool used every request, or definitions totalling under ~100 tokens.

Reconciling the two: the exam's "4-5 tools" is about how many tools are *in front of the model at decision time*. Tool search doesn't raise that number — it keeps it at 3–5 while letting the catalog behind it grow to thousands.

tool_choice Parameter

Controls how Claude interacts with tools:

Value	Behavior	Use Case
`"auto"` (default)	Claude may call a tool OR respond with text	General conversation with optional tool use
`"any"`	Claude MUST call a tool, but can choose which	Guarantee structured output
`{"type": "tool", "name": "specific_tool"}`	Claude MUST call this exact tool	Force a specific operation first
`"none"`	Claude cannot call any tool; text only	Force a text answer while keeping tool definitions in context

Any `tool_choice` value can also carry `"disable_parallel_tool_use": true` to cap Claude at a single tool call per response (by default it may emit several `tool_use` blocks at once).

> **Fable 5.1 / Mythos 5.1 (since 2026-09-01): forced tool use is gone.** `{"type": "any"}` and `{"type": "tool", "name": ...}` return `400 invalid_request_error` (`tool_choice: type "tool" and "any" are not supported for this model`) — on `count_tokens` and in the Batches API too. `auto` and `none` still work, and `disable_parallel_tool_use` still works with `auto`. The replacements, in order of preference:
>
> 1. `tool_choice: {"type": "auto"}` plus an explicit instruction naming the tool (in the `user` turn, or in a mid-conversation `role: "system"` message when the app requires the call), with `strict: true` on the tool so the arguments still match your schema.
> 2. Structured outputs (`output_config.format`, Domain 4 §4.3) when the forced call only existed to get JSON back.
>
> Opus 5, Sonnet 5, and the 4.x family keep all four `tool_choice` values, so the exam's "`any` guarantees a tool call" answer is still correct — just not model-agnostic anymore. Check `client.models.retrieve(id).capabilities` before assuming.

When to Use Each

`"auto"` — Most conversations. Claude decides whether a tool is needed.

`"any"` — When you need guaranteed structured output. Example: You have multiple extraction schemas (invoice, receipt, contract) and you want Claude to pick the right one and always return structured data.

Forced specification — When a specific tool must run first. Example: Always call `extract_metadata` before any enrichment step.

`"none"` — When a turn must produce prose (e.g., a final summarization pass) without stripping the tool definitions from the request.

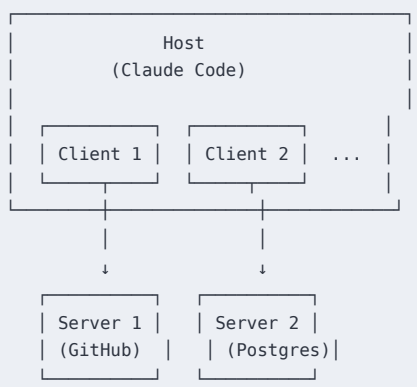
```python
# Force Claude to call a specific tool (Opus 5 / Sonnet 5 / 4.x — 400 on Fable 5.1)
response = client.messages.create(
    model="claude-opus-5",
    messages=[...],
    tools=[...],
    tool_choice={"type": "tool", "name": "extract_metadata"}
)
```

2.4 MCP Server Configuration

What is MCP?

The Model Context Protocol is an open standard for connecting AI models to external data sources and tools. It provides a standardized way for Claude to interact with databases, APIs, file systems, and other services.

Architecture



- **Host**: The AI application (Claude Code, an IDE) that coordinates MCP clients
- **Client**: A component within the host that maintains a 1:1 connection to an MCP server
- **Server**: A program that provides context (tools, resources, prompts) to the client

Transport Types

Transport	`type` in config	How It Works	When to Use
stdio	`"stdio"` (implied when the entry has `command`)	Standard input/output	Local process communication. No network overhead. Most common for local tools.
Streamable HTTP	`"http"` (alias: `"streamable-http"`)	HTTP POST for client→server, optional SSE for streaming	Remote servers, cloud-hosted tools. The only remote transport that supports OAuth
SSE (legacy)	`"sse"`	Server-Sent Events endpoint	Services that still expose only an SSE endpoint
WebSocket	`"ws"`	Persistent bidirectional connection	Remote servers that push events unprompted. Header-only auth, no OAuth

> **Config gotcha:** an entry with a `url` but no `type` is a configuration error — Claude Code reads a typeless entry as a stdio server, skips it, and reports `has a "url" but no "type"`.

> Always set `type` on remote servers.

MCP Protocol Details

- Uses **JSON-RPC 2.0** message format
- Lifecycle:
  1. `initialize` — Capability negotiation (what the server supports)
  2. `notifications/initialized` — Server confirms ready
  3. `tools/list` — Client discovers available tools
  4. `tools/call` — Client invokes a tool

Server Primitives

Primitive	What It Is	Example
Tools	Executable functions	`search_database`, `create_issue`
Resources	Data sources (read-only content catalogs)	File listings, database schemas
Prompts	Interaction templates	Pre-built prompt workflows

Client Primitives

Primitive	What It Is
Sampling	Server requests an LLM completion from the client
Elicitation	Server requests user input from the client
Roots	Server asks the client which URIs or filesystem boundaries it may operate within

Installation Scopes

Three scopes, and the file a server lands in depends on the scope — not on which file you happened to edit.

Scope	Loads in	Shared with the team?	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 (v2.1.259+) — org-provided servers users can't remove

Note the naming trap: MCP *local scope* lives in `~/.claude.json` (home directory), while general *local settings* live in `.claude/settings.local.json` (project directory). They are unrelated files.

Adding servers from the CLI:

```
# Remote HTTP server, shared with the team
claude mcp add --transport http --scope project shared-api https://example.com/mcp

# Remote server with a static auth header
claude mcp add --transport http secure-api https://api.example.com/mcp \
  --header "Authorization: Bearer $TOKEN"

# Local stdio server — everything after -- is passed to the server untouched
claude mcp add airtable --env AIRTABLE_API_KEY=YOUR_KEY -- npx -y @example/mcp-server

# Paste a config block written for another MCP client
claude mcp add-json example '{"type":"http","url":"https://mcp.example.com/mcp"}'
```

Project-Level Configuration (`.mcp.json`)

This file lives in the project root and is version-controlled:

```
{
  "mcpServers": {
    "shared-api": {
      "type": "http",
      "url": "https://example.com/mcp",
      "headers": { "Authorization": "Bearer ${API_TOKEN}" },
      "timeout": 600000
    },
    "postgres": {
      "command": "npx",
      "args": ["@modelcontextprotocol/server-postgres"],
      "env": {
        "DATABASE_URL": "${DATABASE_URL}"
      }
    }
  }
}
```

Key details:

- `${DATABASE_URL}` — Environment variable expansion. The actual secret is NOT in the config file. Use `${VAR:-default}` when the variable may be unset.
- `command` + `args` — How to start the MCP server process (stdio transport)
- `timeout` — per-server tool execution timeout in milliseconds; overrides `MCP_TOOL_TIMEOUT` for that server
- Claude Code **prompts for approval** before using project-scoped servers from `.mcp.json` in interactive sessions. `claude -p`, Agent SDK sessions, and cloud sessions can't show that prompt and load them without asking — use `disabledMcpjsonServers` or `--setting-sources` to keep a server out of headless runs
- Some server names are reserved (`workspace`, `claude-in-chrome`, `computer-use`, `Claude Preview`, `Claude Browser`) and will be skipped with a warning

Authentication for Remote Servers

HTTP and SSE servers support **OAuth** — run `/mcp` in-session (or `claude mcp add` then authenticate) and Claude Code handles the browser flow and token refresh. For services without OAuth, pass a static token via `headers`, or generate one at connect time with `headersHelper` (a command whose stdout becomes the header value) when the credential is short-lived. WebSocket servers are header-only.

Tool Search Is On by Default in Claude Code

MCP tool definitions are **deferred** by default: only tool names and server instructions load at session start, and Claude searches for the rest on demand. That is why Claude Code

imposes no per-server tool cap — the practical limit is your context budget.

`ENABLE_TOOL_SEARCH`	**Behavior**
(unset)	All MCP tools deferred, loaded on demand
`true`	Force deferral (sends the beta header even through proxies)
`auto` / `auto:N`	Load upfront while deferred definitions total under 10% (or N%) of the context window; defer once past it
`false`	Load every MCP tool upfront

Set `"alwaysLoad": true` on a server entry to exempt it from deferral when Claude needs its tools on every turn. Server authors should write good **server instructions** — with tool search on, those instructions are how Claude decides whether to search your server at all. Claude Code truncates tool descriptions and server instructions at 2 KB each.

Output Limits

Claude Code warns when MCP tool output exceeds **10,000 tokens** and truncates at **25,000 tokens** by default. Raise the ceiling with `MAX_MCP_OUTPUT_TOKENS`; the warning threshold is fixed. This is the mechanical reason tool-output trimming (Domain 5) matters — a chatty server can silently lose the tail of its own response.

MCP Resources

Resources expose content catalogs to reduce exploratory tool calls. Instead of Claude blindly searching, it can browse available resources:

```
{
  "resources": [
    {
      "uri": "file:///project/docs/api-reference.md",
      "name": "API Reference",
      "mimeType": "text/markdown"
    }
  ]
}
```

2.5 Built-in Tools

Claude Code's built-in tools and when to use each:

Tool	Purpose	When to Use
Grep	Search file contents for patterns	Finding function definitions, error messages, imports, string patterns
Glob	Find files by name/extension	Locating files: `**/*.test.ts`, `src/**/config.*`
Read	Load full file contents	Reading a specific file after finding it with Grep/Glob
Write	Create new files	Creating new files that don't exist
Edit	Targeted modifications	Changing specific sections of existing files (uses unique text matching)
Bash	Run terminal commands	git, npm, running scripts, system commands
WebSearch / WebFetch	Search the web; fetch and read a URL	Looking up current docs, checking an API's behavior
Agent	Spawn a subagent	Delegating bounded or noisy work (see Domain 1 §1.3)
Skill	Invoke a skill by name	Running a packaged workflow
NotebookEdit	Edit Jupyter notebook cells	`.ipynb` files
AskUserQuestion	Ask the user a blocking question	Decisions only the user can make
ToolSearch	Load a deferred tool's schema on demand	The client-side counterpart of the API's tool search — MCP and rarely used built-in tools ship as names only until Claude searches for them
SendMessage / ListAgents	Message a running or completed subagent, teammate, or another session	Resuming a subagent with its context intact (Domain 3 §3.4); cross-session messaging
Workflow	Run a dynamic-workflow script that orchestrates many subagents	Only when the user opts in ("use a workflow", `ultracode`) — see Domain 1 §1.3

Codebase Exploration Pattern

The correct pattern for exploring an unfamiliar codebase:

```
1. Grep for entry points (main functions, route definitions, exports)
2. Read the entry point files
3. Follow imports — Grep for referenced modules
4. Read those files
5. Build understanding iteratively
```

Anti-pattern: Reading all files upfront. This wastes context and doesn't build understanding.

When Edit Fails

The Edit tool works by matching unique text strings. If the text isn't unique in the file, the edit fails.

Fallback: Use Read to get the full file, then Write to replace the entire file with the modified version.

2.6 Anthropic-Defined and Server-Side Tools

Not every tool is one you write. Some are **Anthropic-defined** (schema-less: you declare a `type` and a `name`, and Claude knows the interface), and some are **server-side** (they execute on Anthropic's infrastructure — results come back as content blocks in the same response, with no client-side execution loop).

Tool	`type`	Client- or server-side	Result block
Web search	`web_search_20260209`	Server	`web_search_tool_result`
Web fetch	`web_fetch_20260209`	Server	`web_fetch_tool_result`
Code execution	`code_execution_20260521`	Server	`bash_code_execution_tool_result` (`.content.stdout`)
Tool search (regex / BM25)	`tool_search_tool_regex_20251119` / `..._bm25_20251119`	Server	`tool_search_tool_result`
Memory	`memory_20250818`	Client (you implement the file ops)	standard `tool_result`
Bash	`bash_20250124`	Client	standard `tool_result`
Text editor	`text_editor_20250728`	Client	standard `tool_result`

Details worth carrying into an exam or a design review:

- **Version the type string.** `web_search_20260209` / `web_fetch_20260209` add dynamic filtering and need Opus 4.6+ / Sonnet 4.6+; older models use the basic `web_search_20250305` / `web_fetch_20250910`. Don't also declare `code_execution` alongside the `_20260209` variants — they run code under the hood, and a second execution environment confuses the model.
- **Web fetch only fetches URLs already present in the conversation.** It is not a crawler.
- **Server-tool errors do not raise.** They return HTTP 200 with an error object inside the result block (e.g. `{"error_code": "max_uses_exceeded"}`). For web search, a success `content` is a *list* and an error `content` is an *object* — branch on that before indexing. This is the same "empty vs failed" distinction from §2.2, at the API layer.
- `pause_turn` is how a server-side tool loop tells you it hit its internal cap. Send the response back to continue (Domain 1 §1.1).
- **Bash and text editor are schema-less.** Declaring a custom tool of your own named `"bash"` with an `input_schema` creates a *different* tool.
- **Advisor tool:** its `model` must be at least as capable as the request's top-level `model` (e.g. executor `claude-sonnet-5` → advisor `claude-opus-5`). An invalid pair returns 400.

Parallel Tool Use

By default Claude may emit several `tool_use` blocks in one assistant message. Execute them concurrently and return **all** `tool_result` blocks in a **single** user message. Splitting them across multiple messages silently teaches Claude to stop making parallel calls. For a tool that failed, return its `tool_result` with `is_error: true` — never drop it.

Programmatic Tool Calling

Claude can call *your* custom tool from inside the code execution sandbox instead of round-tripping through the conversation — useful when a tool returns large results that would otherwise flood the context. Declare `{"type": "code_execution_20260120", "name": "code_execution"}` and set `"allowed_callers": ["code_execution_20260120"]` on the custom tool. When responding to a pending programmatic call, the user message must contain **only** `tool_result`

blocks — no text. Not compatible with `strict: true`, `disable_parallel_tool_use`, forced `tool_choice`, or MCP tools.

MCP from the API Side: the MCP Connector

Domain 2's MCP material is written from Claude Code's perspective (a host that manages its own clients). The Messages API can also connect to remote MCP servers directly. It needs **both halves** — passing `mcp_servers` alone is a validation error:

```
client.beta.messages.create(
    model="claude-opus-5",
    betas=["mcp-client-2025-11-20"],
    mcp_servers=[{"type": "url", "url": "https://mcp.example.com/mcp", "name": "example"}],
    tools=[{"type": "mcp_toolset", "mcp_server_name": "example"}],
    messages=[...],
)
```

Set `defer_loading` once on the `mcp_toolset` entry's `default_config` (or per tool in `configs`) rather than on individual tool definitions.

Changing Tools Mid-Conversation Without Breaking the Cache

The `tools` array sits *earlier* in the cached prefix than `system`, so editing it between turns invalidates the cache for the whole conversation (Domain 4 §4.8). On the models that support mid-conversation system messages (Fable 5 / 5.1, Mythos 5 / 5.1, Opus 4.8, Opus 5 — not Sonnet 5), declare the full tool set up front and then *offer or withdraw* tools with `tool_addition` / `tool_removal` blocks inside a `role: "system"` message (beta header `mid-conversation-tool-changes-2026-07-01`). A tool declared with `defer_loading: true` stays withheld until a `tool_addition` surfaces it. On Fable 5.1 this is also the only history-safe way to change tools, because rebuilding `tools` counts as editing earlier turns and invalidates later thinking blocks.

Domain 2 Practice Questions

Q1: 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

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

Q2: 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

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

Q3: 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`

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

Q4: 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 and mark all but 3-5 frequently used tools `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

Answer: B — 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. D 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.

Q5: 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

Answer: C — 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 and Sonnet 5 still accept forced tool use, so this is a per-model check, not a global rule.

Domain 3: Claude Code Configuration & Workflows (20%)

Covers CLAUDE.md hierarchy and loading, custom commands and skills, path-specific rules, plan mode vs direct execution, iterative refinement, and CI/CD integration.

3.0 Surfaces: Where Claude Code Runs

Claude Code is no longer just a CLI. Every surface connects to the **same engine**, so CLAUDE.md files, settings, skills, and MCP servers work identically across all of them — which is the architecturally relevant point: configuration is portable, and so is a session.

Surface	What it is	Install / entry point
Terminal	Full-featured CLI	`curl -fsSL https://claude.ai/install.sh | bash`, then `claude`
VS Code / Cursor	Extension with inline diffs, @-mentions, plan review	Extensions marketplace
JetBrains	Plugin for IntelliJ, PyCharm, WebStorm (requires the CLI)	JetBrains Marketplace
Desktop app	Standalone app: visual diffs, parallel sessions, scheduled tasks, cloud sessions	macOS / Windows download
Web	Browser sessions with no local setup; long-running tasks on repos you don't have locally	claude.ai/code
Mobile	iOS / Android app for the same cloud sessions	Claude app

Session Portability

Goal	Mechanism
Start locally, finish in the cloud	`claude --cloud "<task>"` (`--remote` is a deprecated alias)
Pull a web/cloud session into the terminal	`claude --teleport`
Run a task as a local background session	`claude --bg "<task>"`, then `claude agents` / `claude attach <id>` / `logs` / `stop` / `respawn` / `rm`
Copy the current session into a background session	`/fork [prompt]` (a forked *subagent* that reports back is `/subtask`; `/branch` switches you into a copy)
Hand the current terminal session to the desktop app	`/desktop`
Drive a running local session from a phone or another browser	Remote Control
Push external events (Telegram, Discord, iMessage, webhooks) into a session	Channels
Route work from team chat	`@Claude` in Slack

Scheduling

Three distinct mechanisms — the exam-relevant distinction is *where they run*:

Mechanism	Runs	Use for
Routines (`/schedule`, web, or desktop)	In the cloud — keeps running with your machine off; can also trigger on API calls or GitHub events	Morning PR reviews, overnight CI failure analysis, weekly dependency audits
Desktop scheduled tasks	On your machine, with local file and tool access	Anything that needs the local filesystem or private network
`/loop`	Inside the current CLI session	Quick polling within one sitting — omit the interval and Claude self-paces between iterations

---

3.1 CLAUDE.md Hierarchy

What is CLAUDE.md?

CLAUDE.md files are instruction files that tell Claude Code how to behave in a specific context. They're like `.editorconfig` or `.eslintrc` but for an AI agent — defining coding standards, project conventions, tool restrictions, and workflow rules.

Loading Order (Broader → More Specific)

Claude Code loads CLAUDE.md files in a hierarchy. More specific files override broader ones:

```
1. Managed Policy (org-wide, admin-controlled)
   └── /etc/claude-code/CLAUDE.md                          (Linux, WSL)
   └── /Library/Application Support/ClaudeCode/CLAUDE.md   (macOS)
   └── C:\Program Files\ClaudeCode\CLAUDE.md               (Windows)

2. User-Level (personal, NOT version-controlled)
   └── ~/.claude/CLAUDE.md

3. Project-Level (shared via version control)
   └── ./CLAUDE.md  or  ./.claude/CLAUDE.md

4. Local project (personal, gitignored)
   └── ./CLAUDE.local.md

5. Directory-Level (loaded on demand)
   └── ./src/api/CLAUDE.md (loaded when Claude reads files in src/api/)
   └── ./tests/CLAUDE.md (loaded when Claude reads files in tests/)
```

Key distinction: Project-level CLAUDE.md is committed to git and shared with the team. User-level is personal and local. `CLAUDE.local.md` is personal per project (gitignore it). Directory-level loads dynamically when Claude works in that directory.

AGENTS.md Interop

During normal sessions Claude Code reads `CLAUDE.md`, **not** `AGENTS.md` — though `/init` does read an existing `AGENTS.md` when scaffolding. If your repo already uses `AGENTS.md` for other coding agents, keep one source of truth by importing it:

```
# CLAUDE.md
@AGENTS.md

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

@import Syntax

CLAUDE.md files can import other files for modular configuration:

```
# Project Configuration

@docs/coding-standards.md
@docs/api-conventions.md
@~/.claude/my-personal-preferences.md

See @README.md for project overview.
```

Resolution rules:

- Relative paths resolve relative to the file containing the import
- `@~/` resolves to the user's home directory
- Maximum import depth: 5 hops (to prevent circular imports)
- Imported files can themselves contain @imports (up to depth limit)

.claude/rules/ Directory

For topic-specific rules organized as separate files:

```
.claude/
  rules/
    testing.md       — Testing conventions
    api-design.md    — API endpoint patterns
    security.md      — Security requirements
    typescript.md    — TypeScript-specific rules
```

Without frontmatter: Rules load unconditionally at session start. **With frontmatter:** Rules load conditionally based on file paths.

Auto Memory

Claude Code maintains automatic memory at:

```
~/.claude/projects/<project>/memory/MEMORY.md
```

- **First 200 lines OR 25 KB** (whichever comes first) loaded at session start
- Machine-local (not version controlled)
- All worktrees in the same git repo share one memory directory
- Claude writes notes here automatically about project patterns, decisions, preferences
- Requires Claude Code v2.1.59 or later
- `autoMemoryEnabled: false` disables it; `autoMemoryDirectory` relocates it
- Topic files (e.g., `debugging.md`, `api-conventions.md`) alongside `MEMORY.md` are loaded on demand — the 200-line cap applies only to `MEMORY.md` itself
- Current layout: `MEMORY.md` is an **index** (one line per memory) and each memory is its own file with YAML frontmatter — `name`, `description`, and a `type` of `user` / `feedback` / `project` / `reference`; Claude Code stamps a `modified` timestamp on every write (v2.1.214+)
- Subagents can maintain their own auto memory

Monorepo and Multi-Directory Controls

Setting / flag	Purpose
`claudeMdExcludes` (in settings)	Glob-list of CLAUDE.md files to skip — useful when another team's nested CLAUDE.md would otherwise load
`--add-dir <path>`	Grant Claude access to additional working directories
`CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD=1`	Also load CLAUDE.md / rules from `--add-dir` paths
`CLAUDE_CODE_NEW_INIT=1`	Interactive multi-phase `/init` that scaffolds CLAUDE.md, skills, and hooks

---

3.2 Skills (and Legacy Slash Commands)

> **Major change (2026):** Custom slash commands have been **merged into skills**. A file at `.claude/commands/deploy.md` and a skill at `.claude/skills/deploy/SKILL.md` both create `/deploy` and behave the same way. The old `/project:name` and `/user:name` namespacing is gone — commands are just `/name`, and higher-priority locations win when names collide.

Skill Locations (Priority: Enterprise > Personal > Project > Plugin)

Location	Path	Applies to
Enterprise	Managed settings path	All users in the org
Personal	`~/.claude/skills/<name>/SKILL.md`	All your projects
Project	`.claude/skills/<name>/SKILL.md`	This project only
Plugin	`<plugin>/skills/<name>/SKILL.md`	Where the plugin is enabled (namespaced `plugin-name:skill-name`)

Legacy `.claude/commands/*.md` still works and supports the same frontmatter — but skills add a directory for supporting files, dynamic context injection, and subagent execution. When a skill and a command share a name, the skill wins. Claude Code skills follow the open Agent Skills standard.

Example SKILL.md

```
---
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 Skill

Deploy the application to $ARGUMENTS.

1. Run the test suite
2. Build the application
3. Push to the deployment target
4. Verify the deployment succeeded
```

Frontmatter Reference (Current)

Field	Purpose
`name`	`/slash-command` name (defaults to directory name; lowercase, digits, hyphens, max 64 chars)
`description`	When to use the skill — Claude uses this to decide whether to auto-load. Capped at 1,536 chars combined with `when_to_use`
`when_to_use`	Additional trigger phrases / example requests, appended to `description`
`argument-hint`	Hint shown in autocomplete, e.g. `[issue-number]`
`arguments`	Named positional arguments for `$name` substitution
`disable-model-invocation`	`true` → only the user can invoke via `/name`; Claude can't auto-trigger
`user-invocable`	`false` → hidden from `/` menu; Claude can still invoke (background knowledge)
`allowed-tools`	Tools pre-approved while this skill is active
`model`	Model override for this skill's turn
`effort`	`low` / `medium` / `high` / `xhigh` / `max` — effort-level override
`context`	`fork` runs the skill in an isolated subagent
`agent`	Which subagent type executes when `context: fork` (e.g. `Explore`, `Plan`, `general-purpose`)
`hooks`	Hooks scoped to this skill's lifecycle
`paths`	Glob patterns — auto-load only when working with matching files
`shell`	`bash` (default) or `powershell` for inline command injection

Substitution Variables

Variable	Resolves To
`$ARGUMENTS`	Full argument string passed by the user
`$ARGUMENTS[N]` / `$N`	0-indexed positional argument (`$0`, `$1`, …)
`$name`	Named argument declared in `arguments:` frontmatter
`${CLAUDE_SESSION_ID}`	Current session identifier
`${CLAUDE_SKILL_DIR}`	Directory containing the SKILL.md file

Dynamic Context Injection

Skills can preprocess shell output and file contents before Claude sees them:

```
---
name: pr-summary
description: Summarize changes in a pull request
context: fork
agent: Explore
allowed-tools: Bash(gh *)
---

## PR context
- Diff: !`gh pr diff`
- Comments: !`gh pr view --comments`
- Changed files: !`gh pr diff --name-only`

## Your task
Summarize this pull request...
```

`!`cmd`` runs the command *before* the skill body is sent. Multi-line blocks use ```` ```! ````. File inclusion with `@path` also works. Disable globally with `"disableSkillShellExecution": true`.

Context Isolation (`context: fork`)

- Spawns the skill as a subagent with its own context
- The `agent:` field picks the execution environment (built-in `Explore`, `Plan`, `general-purpose`, or a custom `.claude/agents/*` subagent)
- CLAUDE.md still loads; main conversation history does not
- Useful for noisy operations (deep research, deployments) that would pollute the main context

Skill Lifecycle and Compaction

When invoked, a skill's rendered body enters the conversation as a single message and stays there. Claude Code does **not** re-read the file on later turns — write standing instructions, not one-time steps. Auto-compaction re-attaches the most recently invoked skills (first 5 000 tokens of each, 25 000-token combined budget), dropping older ones if you used many.

Bundled Skills

Claude Code ships with a set of bundled skills that varies by release. Recent versions include:

- `/claude-api` — Build and migrate Claude API / Anthropic SDK apps; subcommands `migrate`, `prompt-audit`, `upgrade` (Python SDK 0.x → 1.x), `cost-optimize`
- `/code-review` — Multi-agent review of pending changes or a PR (`ultra` runs it in the cloud; `/ultrareview` is the legacy alias)
- `/diff` — Review the working-tree changes, including Claude's edits, before shipping
- `/security-review` — Security review of the pending changes on the branch
- `/simplify` — Reuse / simplification / efficiency cleanups on changed code
- `/dataviz` — Charts and dashboards with consistent design
- `/design` — Multi-artboard visual design canvas
- `/init` — Scaffold a CLAUDE.md for the codebase
- `/run` — Launch the project's app to confirm a change works
- `/loop` — Run a prompt or slash command on a recurring interval
- `/schedule` — Create and manage cloud routines
- `/deep-research` — A bundled dynamic *workflow*: fan out web searches, cross-check sources, synthesize a cited report
- `/workflows` / `/workflow-authoring` — Watch running workflows; load the script-writing reference (when dynamic workflows are enabled)
- `/advisor` — Consult a second model (`fable`, `opus`, `sonnet`) at key moments of a task
- `/fast` — Toggle fast mode (Opus 5 / Opus 4.8 only; premium pricing, same model, faster output)
- `/update-config` — Configure settings.json, hooks, and permissions
- `/fewer-permission-prompts` — Build a permission allowlist from your usage
- `/skill-doctor` — Show what each skill costs in context and how often it is used (v2.1.252+); every listed skill costs tokens on every turn
- `/doctor` — Diagnose Claude Code setup problems; `/feedback` sends a report with session context

Check `/help` in your installed version (or the commands page in the docs) for the current list — the set changes release to release.

Plugins

A **plugin** is a distributable bundle of skills, subagents, hooks, and MCP servers, installed from a marketplace and enabled per project. Plugin-provided items are namespaced (`plugin-name:skill-name` for skills, `plugin:<plugin>:<server>` for MCP servers) so they can't collide with your own. Plugins sit at the lowest priority in the skill resolution order (Enterprise > Personal > Project > Plugin).

Plugins are also the unit that `claude plugin eval` tests: you write an eval suite against the plugin's skills and run it in a sandbox or in CI, and it emits a JSON report. That is the packaging answer to "how does a team ship and regression-test its Claude Code conventions?"

3.3 Path-Specific Rules

Conditional Rule Loading

Rules in `.claude/rules/` can specify which files they apply to using glob patterns in YAML frontmatter:

```
---
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
- Have corresponding integration tests
```

```
---
paths:
  - "**/*.test.tsx"
  - "**/*.test.ts"
  - "**/*.spec.ts"
---

# Testing Conventions

- Use React Testing Library, not Enzyme
- Test behavior, not implementation
- Mock external APIs, not internal modules
- Each test file must have a describe block matching the component/function name
```

When Path-Specific Rules Beat Directory CLAUDE.md

Scenario: Testing conventions that apply to test files scattered throughout the entire codebase.

Directory CLAUDE.md approach:

```
src/
  components/
    CLAUDE.md  ← Would need testing rules here
    Button.test.tsx
  api/
    CLAUDE.md  ← And here
    users.test.ts
  utils/
    CLAUDE.md  ← And here
    format.test.ts
```

Path-specific rules approach:

```
.claude/
  rules/
    testing.md  ← One file with paths: ["**/*.test.*"]
```

The rules approach is superior because:

- Single source of truth for testing conventions
- Applies everywhere test files exist, even new directories
- No need to duplicate CLAUDE.md files across directories

3.4 Plan Mode vs Direct Execution

Permission Modes (Current)

Plan mode is now one entry in a broader permission-mode system. Start in a specific mode with `--permission-mode <mode>`, or cycle modes with **Shift+Tab** in an interactive session.

Mode	Behavior
`default`	Standard permission prompts
`acceptEdits`	Auto-accept file edits; still prompt for other tools
`plan`	Exploration-only: Claude designs an approach and presents it before touching files
`auto`	Classifier decides per-tool whether to prompt — see `claude auto-mode defaults`
`dontAsk`	Never prompt for the pre-approved list
`bypassPermissions`	Skip all prompts (dangerous; use only for short, contained tasks)

When to Use Plan Mode

Indicator	Example
Multi-file changes	Refactoring auth across 15 files
Architectural decisions	Choosing between REST and GraphQL
Multiple valid approaches	Several ways to implement caching
Large scope	Library migration affecting 45+ files
Unclear requirements	"Make the app faster" — needs investigation first

In plan mode, Claude:

1. Explores the codebase (Glob, Grep, Read)
2. Identifies affected files and patterns
3. Designs an approach

4. Presents the plan for approval
5. Only implements after user approval

When to Use Direct Execution

Indicator	Example
Single-file fix	Bug fix with clear stack trace
Obvious scope	Adding a null check
Clear instructions	"Add a `createdAt` field to the User model"
Small, contained	Fixing a typo, updating a constant

Built-in Subagents

Four subagents ship with Claude Code and are registered by default in interactive sessions:

Agent	Purpose	Tools
Explore	Fast file discovery and codebase search	Read-only; skips CLAUDE.md and git status
Plan	Research for plan mode before proposing changes	Read-only; skips CLAUDE.md and git status
general-purpose	Multi-step tasks needing exploration *and* action	All subagent tools
claude	Catch-all fallback	All subagent tools

The Explore agent is the canonical context-preservation move:

```
Main conversation context is preserved
     ↓
Explore subagent runs (Glob, Grep, Read — lots of output)
     ↓
Returns concise summary to main conversation
     ↓
Main context stays clean
```

Without it, a deep codebase exploration would fill the main context window with raw file contents and search results.

Custom Subagents (`.claude/agents/`)

Custom subagents are markdown files with YAML frontmatter, resolved in the order: managed settings > `--agents` CLI JSON > project (`.claude/agents/`) > user (`~/.claude/agents/`) > plugin.

```yaml
---
name: db-reader
description: Execute read-only database queries        # how Claude decides to delegate
tools: Read, Grep, Bash                                # allowlist
disallowedTools: Write, Edit                           # denylist, applied after `tools`
model: sonnet                                          # inherit | sonnet | opus | haiku | full ID
effort: low                                            # low | medium | high | xhigh | max
permissionMode: plan                                   # default | acceptEdits | auto | dontAsk | bypassPermissions | pla
maxTurns: 10
skills: [api-conventions]                              # preloaded skill content
mcpServers: [github]
memory: project                                        # user | project | local — persistent across sessions
isolation: worktree                                    # run in its own git worktree
background: false
experimental:
  cacheTtl: 1h                                         # prompt-cache lifetime for this subagent (v2.1.248+)
hooks:
  PreToolUse:
    - matcher: "Bash"
      hooks:
        - type: command
          command: ./scripts/validate-readonly-query.sh
---

Execute SELECT queries only. Never issue writes.
```

Three ways to invoke: name it in natural language ("use the db-reader to…"), `@`-mention it to guarantee that specific agent runs, or start a whole session as it with `claude --agent db-reader`.

What a fresh subagent loads: its own system prompt, the delegation prompt, the CLAUDE.md hierarchy, a git status snapshot, preloaded skills, and environment details. **What stays out:** your conversation history, the main session's auto memory, and other subagents' results — which is exactly the "context does not inherit" rule from Domain 1 §1.3.

Constraints worth remembering: permission modes flow *down* — if the parent runs `bypassPermissions`, `acceptEdits`, or `auto`, the subagent inherits it and cannot override. Subagents nest 3 deep by default and 20 can run concurrently. Completed subagents can be resumed by ID or name via `SendMessage`, retaining their prior context; one-shot agents (Explore, Plan) cannot be resumed.

Prompt-injection note: Claude Code scans subagent reports for instruction-shaped patterns before the parent reads them, escaping text that imitates harness output. It's a mitigation, not a permission boundary — every tool call still goes through the session's permission checks.

Beyond Subagents: Agent Teams and Dynamic Workflows

Subagents are one worker per delegation, decided turn by turn. Two larger units exist when that isn't enough, and the exam-style question is *who holds the plan*:

	Subagents	Agent teams (experimental)	Dynamic workflows
What it is	A worker Claude spawns	A lead session supervising peer Claude Code sessions with a shared task list and inter-agent messaging	A JavaScript script Claude writes and a runtime executes in the background
Who decides what runs next	Claude, turn by turn	The lead agent	The script (`agent()`, `parallel()`, `pipeline()`, `phase()`)
Scale	A few delegated tasks per turn	A handful of long-running peers	Dozens to hundreds of agents per run; resumable in-session
Enable	Built in	`CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS=1`	Paid plans and API access; Pro turns it on in `/config`; `workflowSizeGuideline` caps the advisory agent count

A workflow only runs when the user opts in — the `ultracode` effort level, an explicit "use a workflow", or a bundled workflow skill such as `/deep-research`. `/workflows` watches, pauses, resumes, or saves a run. Codifying the orchestration is the point: the plan becomes a reviewable, rerunnable script instead of a coordinator's memory (Domain 1 §1.2).

Combining Modes

Plan mode for investigation → Direct execution for implementation:

```
1. Enter plan mode
2. Explore the codebase
3. Design the approach
4. Present plan to user
5. User approves
6. Exit plan mode
7. Implement directly, file by file
```

3.5 Iterative Refinement Techniques

1. Concrete Examples

When prose instructions produce inconsistent results, provide 2-3 input/output examples:

```
Instead of: "Format dates consistently"

Use: "Format dates as YYYY-MM-DD. Examples:
  Input: 'January 5th, 2024' → Output: '2024-01-05'
  Input: '12/31/23' → Output: '2023-12-31'
  Input: 'Mar 2024' → Output: '2024-03-01'"
```

2. TDD Iteration

Write tests first, then iterate by sharing test failures:

```
1. Write the test: expect(validateEmail("bad")).toBe(false)
2. Run tests → FAIL
3. Share failure with Claude
4. Claude fixes the implementation
5. Run tests → PASS or new failures
6. Repeat until green
```

3. Interview Pattern

Claude asks clarifying questions before implementing:

```
User: "Add authentication to the app"
Claude: "Before I implement, a few questions:
  - JWT or session-based?
  - Do you need refresh tokens?
  - What's the session duration?
  - Should I add role-based access control?"
```

4. Multiple vs Independent Issues

Multiple interacting issues: Address in a single message because fixes may conflict.

```
"The auth middleware has two issues:
1. It doesn't check token expiry
2. It returns 500 instead of 401 for invalid tokens
Fix both together."
```

Independent issues: Fix sequentially to keep each change clean and reviewable.

3.6 CI/CD Integration

CLI Flags for CI

Flag	Purpose	Critical?
`-p` / `--print`	Non-interactive mode	**MUST use in CI** or the process hangs waiting for input
`--output-format`	`text`, `json`, or `stream-json` for machine-parseable output	Needed for downstream processing
`--input-format stream-json`	Stream input events back into the agent	For pipeline composition
`--json-schema '{...}'`	Validate final output against a JSON Schema (print mode only)	Enforces CI output shape
`--max-turns N`	Hard cap on agentic turns; exits with error if hit	Guardrail
`--max-budget-usd X`	Stop once API spend hits $X (print mode only)	Cost guardrail
`--bare`	Skip auto-discovery (hooks, skills, plugins, MCP, auto memory, CLAUDE.md) — fast scripted starts	Low-overhead scripted calls
`--no-session-persistence`	Don't save the session to disk	Ephemeral CI runs
`--fallback-model sonnet`	Auto-fall back when the primary model is overloaded	Reliability
`--include-hook-events`	Emit all hook lifecycle events into the stream	Observability
`--include-partial-messages`	Emit partial streaming events	Debugging
`--permission-mode <mode>`	Start in `default` / `acceptEdits` / `plan` / `auto` / `dontAsk` / `bypassPermissions`	See §3.4
`--permission-prompt-tool <mcp>`	Delegate permission prompts to an MCP tool in headless mode	Non-interactive approvals
`--permission-prompts none`	Deny anything that would prompt when nobody can answer (v2.1.259+); the active mode still decides everything else	Unattended runners
`--restricted`	Remove command/code-running tools and `WebFetch`, confine file tools to the working directories, load only managed settings, refuse `bypassPermissions` (v2.1.248+)	Eval harnesses on shared machines
`--safe-mode`	Start with every customization disabled (CLAUDE.md, skills, hooks, MCP, plugins, workflows, auto memory) — unlike `--bare`, permissions still apply	Troubleshooting a broken config
`--session-id`, `--resume`, `--continue`, `--fork-session`	Session lifecycle control	Multi-step pipelines
`--setting-sources user,project,local`	Restrict which settings layers are loaded	Reproducible runs
`--effort <level>`	`low` / `medium` / `high` / `xhigh` / `max` / `ultracode`	Tune cost vs quality per run
`--agent <name>` / `--agents '<json>'`	Run the session as a named subagent, or define subagents inline	Specialized CI jobs
`--append-subagent-system-prompt` / `-file`	Append text (or a file, v2.1.261+) to every subagent's prompt except forks (non-interactive only)	Consistent citation/format rules

`--forward-subagent-text`	Emit subagent messages into the stream	Observability across delegated work
`--system-prompt` / `--system-prompt-file` / `--append-system-prompt`	Replace or extend the system prompt	Purpose-built CI personas
`--exclude-dynamic-system-prompt-sections`	Move per-machine sections out of the system prompt	Prompt-cache reuse across runners
`--strict-mcp-config`	Use only the servers from `--mcp-config`	Hermetic MCP surface
`--init` / `--init-only`	Run Setup / SessionStart hooks (and exit)	Warm a CI workspace before the real run
`--bg` / `--background` + `--exec`	Start as a background agent; run a shell command as a background job	Long tasks that shouldn't block
`--environment ccpool_*` / `--ref`	Target a self-hosted environment; check out a named ref	Fleet / remote execution
`--betas`	Send beta headers with API requests	Opt into preview features

Example CI Usage

```
# GitHub Actions example
- name: Code Review
  run: |
    claude -p "Review this PR for bugs and security issues" \
      --output-format json \
      --json-schema '{"type":"object","properties":{"issues":{"type":"array"}}}'
```

Managed CI Integrations

You don't always wire the CLI up by hand:

Integration	What it does
GitHub Actions	Run Claude Code in a workflow — PR review, issue triage, `@claude` mentions
GitLab CI/CD	Same model on GitLab pipelines
GitHub Code Review	Automatic review on every PR, no workflow file to maintain
Routines	Cloud-scheduled runs that can also fire on GitHub events or API calls

The `-p` + `--output-format json` pattern below is still the right primitive when you need full control; these integrations are the managed path.

Cost, Cache, and Fleet Controls (Settings)

A few settings that turn up in "how does the platform team keep 200 developers' sessions cheap and consistent" scenarios:

Setting	Purpose
`promptCacheTtl` / `subagentPromptCacheTtl`	Choose the 5-minute or 1-hour prompt-cache lifetime for the session and for subagents (per-agent override: `experimental.cacheTtl` in frontmatter)
`modelPicker` / `modelPricing` (managed)	Curate the `/model` list; supply custom per-token rates so `/cost` is accurate behind a gateway
`managedMcpServers` (managed)	Org-provided MCP servers every session gets
`bashOutputMaxChars` / `taskOutputMaxChars`	Cap inline tool output (up to 128K chars) before it floods the context
`permissions.blockReadsOutsideWorkingDirectories`	Keep file reads inside the working directories
`workflowSizeGuideline`	Advisory agent-count ceiling for dynamic workflows

`/cost` now reports per-session prompt-cache statistics and cache-miss diagnostics — the Claude Code view of the same prefix-stability problem described in Domain 4 §4.8.

Message Batches API

For non-blocking, cost-optimized batch operations:

Feature	Detail
Cost savings	50% compared to synchronous API
Processing window	Up to 24 hours, no latency SLA
Correlation	`custom_id` per request for matching responses — results arrive in **any order**, so key by `custom_id`, never by position
Multi-turn	NOT supported within a single batch request
Extended output	Up to 300k output tokens on Opus 5 / 4.8 / 4.7 / 4.6 and Sonnet 5 / 4.6 with beta header `output-300k-2026-03-24` (vs 128k synchronous)

Good for:

- Overnight code analysis reports
- Weekly audit summaries
- Nightly test generation
- Batch document processing

NOT good for:

- Pre-merge checks (blocking, needs immediate response)
- Real-time user interactions
- Anything requiring tool calling loops within a single request

Handling Batch Failures

```
# Process batch results
for result in batch_results:
    if result.status == "failed":
        failed_ids.append(result.custom_id)

# Resubmit only failures
resubmit_batch = [r for r in original_requests if r.custom_id in failed_ids]
```

Session Context Isolation for Reviews

Problem: The same Claude session that wrote code is less effective at reviewing it because it retains the reasoning context (it already "knows" why it made each decision, making it less likely to question them).

Solution: Use independent review instances that don't have the generation context.

```
Generation session: Claude writes the code
                          ↓
              (new session, no shared context)
                          ↓
Review session: Claude reviews the code fresh
```

Domain 3 Practice Questions

Q1: 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

Answer: B — 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.

Q2: 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

Answer: B — 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.

Q3: 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`

Answer: B — 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.

Domain 4: Prompt Engineering & Structured Output (20%)

Covers writing explicit criteria, few-shot prompting, getting guaranteed structured output via tool_use and JSON schemas, validation-retry loops, batch processing, and multi-pass review architecture.

4.1 Explicit Criteria

The Problem with Vague Instructions

Vague instructions produce inconsistent results because the model interprets them differently each time.

Bad (vague):

```
"Be conservative when flagging issues."
"Only report high-confidence findings."
"Use your best judgment on severity."
```

Good (explicit):

```
"Flag a comment as outdated ONLY when the claimed behavior directly
contradicts the actual code behavior. Do NOT flag comments that are
merely incomplete or could be more detailed."
```

Explicit Criteria Structure

For each type of judgment the model must make, define:

1. **What qualifies** — Specific conditions that trigger the judgment
2. **What doesn't qualify** — Boundary cases that should NOT trigger it
3. **Examples** — Concrete input/output pairs showing the boundary

```
Flag as SECURITY ISSUE when:
  - User input is passed to SQL queries without parameterization
  - User input is rendered in HTML without escaping
  - Credentials are hardcoded in source files

Do NOT flag:
  - ORM queries (already parameterized)
  - Rendering user input in server-side templates with auto-escaping
  - Credentials loaded from environment variables
```

False Positive Impact

High false positive rates in one category undermine trust in ALL categories, even accurate ones. Developers start ignoring all warnings.

Solution: When a category has high false positives, temporarily disable it and improve the criteria prompts before re-enabling. Don't let one noisy category degrade the value of accurate ones.

4.2 Few-Shot Prompting

When to Use Few-Shot

- Ambiguous scenarios where prose instructions aren't precise enough
- Tasks where the format or reasoning pattern needs demonstration
- Reducing hallucination in extraction tasks
- Showing how to handle edge cases

How Many Examples

2-4 targeted examples is the sweet spot. Too few may not cover the pattern. Too many wastes context and may overfit to the examples.

What Good Examples Include

Each example should demonstrate:

1. **Input** — What the model receives
2. **Output** — What it should produce
3. **Reasoning** — WHY this output is correct (not just what to do, but why)

```
Example 1:
Input: "// Returns the user's full name"
Code: function getName(user) { return user.firstName; }
Output: {
  "issue": "Comment claims full name but function only returns firstName",
  "severity": "medium",
  "location": "line 1",
  "reasoning": "The comment says 'full name' but the code returns only
    firstName, not firstName + lastName. This is a factual contradiction."
}

Example 2:
Input: "// Validates the input"
Code: function validate(input) { return input.length > 0 && input.length < 100; }
Output: null
Reasoning: "The comment says 'validates input' and the function does validate
  input. The comment is vague but not wrong — it doesn't claim specific
  validation rules that contradict the implementation."
```

Few-Shot Enables Generalization

Good examples teach the pattern, not just the specific cases. The model should generalize to novel situations by understanding the reasoning behind each example.

4.3 Structured Output via tool_use

The Problem

When you need Claude to return structured data (JSON), free-form text generation can produce:

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

Solution 1: tool_use with Schema

Define a tool that represents your desired output structure:

```
{
  "name": "extract_invoice_data",
  "description": "Extract structured data from an invoice document",
  "input_schema": {
    "type": "object",
    "properties": {
      "vendor_name": {
        "type": "string",
        "description": "The company or person who issued the invoice"
      },
      "invoice_number": {
        "type": "string",
        "description": "The unique invoice identifier"
      },
      "line_items": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "description": { "type": "string" },
            "quantity": { "type": "number" },
            "unit_price": { "type": "number" },
            "total": { "type": "number" }
          },
          "required": ["description", "quantity", "unit_price", "total"]
        }
      },
      "total_amount": { "type": "number" },
      "currency": {
        "type": "string",
        "enum": ["USD", "EUR", "GBP", "other"]
      }
    },
    "required": ["vendor_name", "invoice_number", "line_items", "total_amount", "currency"]
  }
}
```

Use `tool_choice: {"type": "tool", "name": "extract_invoice_data"}` to force Claude to call this tool, guaranteeing structured output — on Opus 5, Sonnet 5, and the 4.x family. **Fable 5.1 / Mythos 5.1 reject forced tool use with a 400** (Domain 2 §2.3); there, prefer Solution 3 below, or `auto` plus an instruction naming the tool with `strict: true`.

Solution 2: Strict Mode

Add `"strict": true` to the tool definition for guaranteed schema compliance:

```
{
  "name": "extract_invoice_data",
  "strict": true,
  "input_schema": {
    "type": "object",
    "properties": { ... },
    "required": ["vendor_name", "invoice_number", "line_items", "total_amount", "currency"],
    "additionalProperties": false
  }
}
```

Strict mode requirements:

- `additionalProperties: false` must be set
- ALL properties must be listed in `required`

Solution 3: output_config (Direct JSON Output)

Instead of tool_use, you can request JSON output directly. The parameter is `output_config.format`. The older top-level `output_format` is **deprecated** — don't use it in new code — and the previous `structured-outputs-2025-11-13` beta header is no longer required:

```
response = client.messages.create(
    model="claude-opus-5",
    messages=[{"role": "user", "content": "Extract data from this invoice: ..."}],
    output_config={
        "format": {
            "type": "json_schema",
            "schema": {
                "type": "object",
                "properties": {
                    "vendor_name": { "type": "string" },
                    "total_amount": { "type": "number" }
                },
                "required": ["vendor_name", "total_amount"],
                "additionalProperties": false
            }
        }
    }
)
```

> **SDK helper:** `client.messages.parse()` (Python/TypeScript) wraps this pattern — pass a Pydantic model or Zod schema and it sends `output_config.format`, validates the response against your schema, and returns the parsed object via `response.parsed_output`. Prefer it over hand-parsing JSON when using structured outputs.

Schema Compliance vs Semantic Correctness

Important exam concept: Structured output guarantees **schema compliance** (correct types, required fields present, valid enums) but does NOT guarantee **semantic correctness**.

Schema compliance: ✅ `"total_amount": 150.00` (correct type, field exists) Semantic error: ❌ `"total_amount": 150.00` when line items sum to 175.00

You still need validation logic to catch semantic errors.

Schema Design Patterns

Required vs Optional Fields:

```
{
  "purchase_order_number": {
    "type": ["string", "null"],
    "description": "PO number if present on the invoice, null if not found"
  }
}
```

Use nullable fields when source documents may not contain the information. Making everything required can force Claude to fabricate data.

Enum with "other":

```
{
  "document_type": {
    "type": "string",
    "enum": ["invoice", "receipt", "contract", "purchase_order", "other"]
  },
  "document_type_detail": {
    "type": ["string", "null"],
    "description": "If document_type is 'other', describe the document type"
  }
}
```

Enum with "unclear":

```
{
  "payment_status": {
    "type": "string",
    "enum": ["paid", "unpaid", "partial", "unclear"]
  }
}
```

Adding `"unclear"` as an option prevents Claude from guessing when the information is ambiguous.

JSON Schema Limitations in Strict Mode

Strict mode supports a subset of JSON Schema:

Supported	NOT Supported
`type` (object, array, string, integer, number, boolean, null)	`minimum`, `maximum`, `multipleOf`
`properties`, `required`	`minLength`, `maxLength`
`enum`, `const`	`pattern` (regex), `patternProperties`
`items` (for arrays)	`oneOf`, `if`/`then`/`else`
`anyOf`, `allOf`, `$ref`/`$defs`	Recursive schemas
`additionalProperties: false` (required on every object)	`additionalProperties` set to anything other than `false`
`format` (`date-time`, `time`, `date`, `duration`, `email`, `hostname`, `uri`, `ipv4`, `ipv6`, `uuid`)	Complex array constraints

> Earlier versions of structured outputs did not support `anyOf`/`allOf` — they do now. The Python and TypeScript SDKs strip unsupported constraints from the schema they send and validate them client-side instead.

Implication: Numeric range validation, string length validation, regex patterns, and complex conditional schemas must be handled in your validation code, not in the schema.

4.4 Validation-Retry Loops

The Pattern

```
max_retries = 3

for iteration in range(max_retries):
    response = extract_data(document, prompt)
    errors = validate(response)

    if not errors:
        return response  # Success

    # Append specific errors to prompt for next attempt
    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: {errors}")
```

What Validation Catches

Schema validation (can catch reliably):

- Missing required fields
- Wrong data types
- Invalid enum values
- Malformed dates or emails

Semantic validation (needs custom logic):

- Line items don't sum to total
- Dates are in the future when they shouldn't be
- Referenced entities don't exist
- Cross-field contradictions

Self-Correction Patterns

Calculated vs Stated:

```
{
  "stated_total": 150.00,
  "calculated_total": 175.00,
  "conflict_detected": true,
  "conflict_note": "Stated total ($150) doesn't match sum of line items ($175)"
}
```

By extracting both the stated value and a calculated value, you can detect discrepancies automatically.

When Retries Don't Work

Retries are ineffective when the information is **absent from the source document**. If the invoice doesn't have a PO number, retrying won't make one appear.

Solution: Use nullable/optional fields and accept null for missing data rather than retrying.

Tracking False Positives

Add a `detected_pattern` field to track which code constructs trigger findings:

```
{
  "issue": "Potential SQL injection",
  "detected_pattern": "string concatenation in query",
  "file": "users.py",
  "line": 42
}
```

This allows analysis of dismissal patterns — if developers consistently dismiss "string concatenation in query" findings, the prompt criteria for that pattern may need refinement.

4.5 Batch Processing

Message Batches API

Feature	Detail
Cost	50% savings vs synchronous
Latency	Up to 24 hours, no SLA
Correlation	`custom_id` per request — results come back in **any order**, so key by `custom_id`, never by position
Multi-turn	NOT supported in a single batch request
Max output	128k normally; up to **300k** with beta header `output-300k-2026-03-24` on Opus 5 / 4.8 / 4.7 / 4.6 and Sonnet 5 / 4.6
Result states	`succeeded`, `errored`, `canceled`, `expired` — poll `batches.retrieve(id).processing_status` until `"ended"`, then stream `batches.results(id)`

Creating a Batch

```
batch = client.messages.batches.create(
    requests=[
        {
            "custom_id": "invoice-001",
            "params": {
                "model": "claude-opus-5",
                "max_tokens": 4096,
                "messages": [
                    {"role": "user", "content": f"Extract data from: {invoice_001_text}"}
                ],
                "tools": [extract_tool],
                "tool_choice": {"type": "tool", "name": "extract_invoice_data"}
            }
        },
        {
            "custom_id": "invoice-002",
            "params": { ... }
        }
    ]
)
```

When to Use Batch vs Synchronous

Synchronous	Batch
Pre-merge code review checks	Overnight code analysis
Real-time user interactions	Weekly audit reports
Blocking CI steps	Nightly test generation
Interactive debugging	Bulk document processing

Failure Handling

Don't resubmit the entire batch when some requests fail. Use `custom_id` to identify and resubmit only failures:

```
failed = [r for r in results if r.status == "failed"]
resubmit = [orig for orig in original if orig.custom_id in [f.custom_id for f in failed]]
```

Pre-Batch Refinement

Always refine your prompts on a small sample before batch-processing large volumes. This catches prompt issues before you waste credits on 10,000 documents.

4.6 Multi-Pass Review

The Self-Review Problem

When Claude generates code and then reviews it in the same session, it retains the reasoning context from generation. It "remembers" why it made each decision, making it less likely to question those decisions.

This is called **reasoning context bias**.

The Solution: Independent Review Instances

```
Session 1: Claude generates code
                    ↓
            (completely separate session)
                    ↓
Session 2: Claude reviews the code
           (no knowledge of why decisions were made)
```

Multi-Pass Architecture for Large PRs

Pass 1 — Per-file local analysis:

- Review each file independently
- Consistent depth across all files
- Catches: bugs, style issues, type errors, security vulnerabilities
- Separate prompt for each file

Pass 2 — Cross-file integration:

- Review how files interact with each other
- Different prompt focused on integration concerns
- Catches: data flow issues, interface mismatches, missing error handling at boundaries

- Receives results from Pass 1 for context

Why Not Single-Pass?

For large PRs (10+ files), single-pass review suffers from:

1. **Attention dilution** — Model focuses on early files, gives less attention to later ones
2. **Inconsistent standards** — Different quality of review across files
3. **Missing integration issues** — Can't see cross-file problems when reviewing one file at a time in a single pass

---

4.7 Thinking, Effort, and the Current Request Surface

The request shape for prompting Claude changed substantially across the 4.6 → 5 generations. Several parameters that older material treats as standard now return **400** on current models.

Adaptive Thinking Replaced Thinking Budgets

```
response = client.messages.create(
    model="claude-opus-5",
    max_tokens=16000,
    thinking={"type": "adaptive", "display": "summarized"},
    output_config={"effort": "high"},
    messages=[...],
)
```

Model	Thinking config	Omitting `thinking`	`budget_tokens`	`temperature` / `top_p` / `top_k`
Claude Fable 5.1 / Mythos 5.1	`{"type": "adaptive"}` or omit; `disabled` → 400	Runs adaptive (always on)	**400**	**400**
Claude Fable 5	`{"type": "adaptive"}` or omit	Runs adaptive (always on)	**400**	**400**
Claude Opus 5	`{"type": "adaptive"}` or omit	Runs **adaptive** by default	**400**	**400**
Claude Opus 4.8 / 4.7	`{"type": "adaptive"}`	Runs **without** thinking	**400**	**400**
Claude Sonnet 5	`{"type": "adaptive"}`	Runs adaptive	**400**	**400**
Claude Haiku 4.5	`{"type": "enabled", "budget_tokens": N}`	No thinking	Required for thinking	Allowed

The subtlety worth flagging: on **Opus 4.8/4.7 you must set `adaptive` explicitly** or you get no thinking at all, while on **Opus 5 thinking is on by default**. Code carried forward from 4.8 that disables thinking will behave differently on Opus 5.

Effort

`effort` is GA (no beta header) and lives **inside** `output_config`, not at the top level:

```
output_config={"effort": "xhigh"}   # low | medium | high | xhigh | max
```

It defaults to `high` on the Claude API (equivalent to omitting it) and to `xhigh` in Claude Code; Haiku 4.5 rejects it, and `xhigh` is not available on Opus 4.6 / Sonnet 4.6. Anthropic's current per-model guidance: on **Opus 5 and Fable 5.1 start at `high` and sweep** — step up to

`xhigh`/`max` only where evals show headroom, and use `low`/`medium` liberally for routine or latency-sensitive routes (lower effort on these models often beats `xhigh` on the previous generation); on **Opus 4.8 / 4.7 start at `xhigh`** for coding and agentic work. Lower effort means fewer, more-consolidated tool calls and terser output. Effort matters more on these models than on any prior generation, so **re-tune it when you migrate**, don't carry the old value across. Changing the top-level `effort` mid-conversation invalidates the message cache; on Fable 5.1, Mythos 5.1, and Opus 5, a **per-message effort** change — a `role: "system"` message carrying `output_config.effort` (beta `mid-conversation-output-config-2026-07-01`) — keeps the prefix intact.

Thinking Display

`display: "omitted"` is now the **default** on Fable 5 / 5.1, Opus 5, Opus 4.8/4.7, and Sonnet 5 — a silent change from Opus 4.6 and Sonnet 4.6, where it was `"summarized"`. Thinking still happens and is still billed identically; only visibility changes. If you stream reasoning to users, the default looks like a long pause before any output, so set `display: "summarized"` explicitly. The raw chain of thought is never exposed on any model. On Fable 5.x the text Claude writes *between tool calls* comes back as progress-update `thinking` blocks rather than `text` blocks, so under the default it is empty; `display: "updates"` (beta `thinking-display-updates-2026-08-18`) returns those progress notes as readable text while the reasoning itself stays hidden.

Assistant Prefill Is Gone

Prefilling the last assistant turn to force a response format returns **400** on Fable 5, Opus 5, Sonnet 5, and the whole 4.6/4.7/4.8 family. Use structured outputs (`output_config.format`) or system-prompt instructions instead. This retires a prompt-engineering technique that a lot of older material still recommends.

Mid-Conversation System Messages

On Claude Opus 5, Opus 4.8, Fable 5 / 5.1, and Mythos 5 / 5.1 (not Sonnet 5, no beta header) you can append `{"role": "system", "content": "..."}` to the **`messages` array** rather than editing the top-level `system` field. This is the prompt-injection-safe operator channel, and — because it doesn't touch the cached prefix — the cheap one. Constraints: it must follow a `user` message (or an assistant message ending in server-tool use), can't be `messages[0]`, and must either be last or be followed by an assistant turn.

Two extensions arrived with Fable 5.1 (both beta): **turn-scoped** messages with `clear_at: "next_user_message"` (header `mid-conversation-system-clear-at-2026-08-21`) render for one turn, then stay in the transcript cleared — the right shape for a per-turn reminder such as "batch independent tool calls", which Fable 5.1 needs more often than Fable 5 in long loops; and **tool changes** via `tool_addition` / `tool_removal` blocks (header `mid-conversation-tool-changes-2026-07-01`), covered in Domain 2 §2.6. Never delete earlier copies of these messages: on Fable 5.1 that counts as editing history.

Preserved Thinking: The History Is Now Part of the Contract

Every `thinking` block Fable 5.1 emits records the model that produced it *and* is valid only against the exact `system`, `tools`, and message prefix that preceded it. Two rules follow:

1. **Model binding.** Fable 5.1 reads its own blocks and those of Opus 5, Fable 5, Mythos 5, and earlier models; none of those can read Fable 5.1's. When a fallback, router, or retry moves a conversation to an older model, the API silently drops the unreadable blocks (unbilled) and that model re-plans.
2. **Prefix binding.** Editing, reordering, or removing earlier turns — including client-side compaction that keeps recent turns verbatim behind a summary — invalidates every later

> thinking block. Where enforced (organizations created on or after 2026-08-31 today; every organization on later models), a request that replays such a block returns a 400. Append-only histories, server-side compaction, and context editing are all safe, because the check compares the conversation *as you sent it*.

If Claude Code, claude.ai, Managed Agents, or the Agent SDK own your history, they already comply. If your code builds `messages` itself, run the three-step check from the migration guide: capture consecutive request bodies, diff `system` / `tools` / shared `messages` prefix, and fix every non-append change — using `thinking-binding-controls-2026-08-01` with `prefix_mismatch_behavior: "drop_block"` to log `input_transformations` while you do. Domain 5 §5.7 covers the compaction shapes that stay valid.

4.8 Prompt Caching

Caching is a prompt-*structure* problem, which is why it belongs here rather than in an ops runbook.

Prefix match: the cache keys on an exact prefix. Any byte change anywhere in the prefix invalidates everything after it. Render order is `tools` → `system` → `messages`, so a tool list that reorders between requests invalidates the system prompt and the whole conversation behind it.

Design rule: stable content first (frozen system prompt, deterministically ordered tool list), volatile content (timestamps, per-request IDs, the actual question) after the last `cache_control` breakpoint.

Mechanic	Detail
Breakpoints	Max 4 per request (`cache_control: {"type": "ephemeral"}`)
Minimum prefix	Model-dependent — 512 tokens on Fable 5.x / Mythos 5.x / Opus 5; 1,024 on Opus 4.8, Sonnet 5, Sonnet 4.6/4.5; 2,048 on Opus 4.7; 4,096 on Opus 4.6/4.5 and Haiku 4.5. Shorter prefixes silently don't cache
Pricing	Cache reads cost 10% of the input price (2.5% on Fable 5.1 / Mythos 5.1 — $0.25/MTok); 5-minute writes 1.25×, 1-hour writes 2×
Verification	`usage.cache_read_input_tokens`. Zero across repeated requests means something is invalidating the prefix. **Cache diagnostics** (beta `cache-diagnosis-2026-04-07`, `diagnostics: {previous_message_id}`) has the API report exactly where consecutive requests diverged
Pre-warming	`max_tokens: 0` warms a cache entry without generating — with the same thinking and `effort` configuration your real traffic uses, or the entry is never hit

Silent invalidators to audit for: `datetime.now()` in the system prompt, JSON serialized with non-deterministic key order, a tool set that varies per request, a session ID or user name interpolated into the prefix, switching `speed` (fast mode) mid-conversation, changing the top-level `effort` or thinking configuration between requests (use per-message effort on Fable 5.1 / Opus 5), and changing a task-budget value (it is rendered into the prompt).

Interaction with tool search: deferred tools are excluded from the system-prompt prefix and discovered tools are appended inline, so `defer_loading` *preserves* the cache. A tool with `defer_loading: true` cannot also carry `cache_control` — put the breakpoint on a non-deferred tool.

Domain 4 Practice Questions

Q1: 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

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

Q2: 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

Answer: C — 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.

Q3: 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

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

Q4: 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

Answer: B — `budget_tokens` and the sampling parameters (`temperature` / `top_p` / `top_k`) are removed on Fable 5, Opus 5, Sonnet 5, and the 4.7/4.8 family, and return a 400. Adaptive thinking plus `effort` replaces the fixed-thinking-budget concept. Assistant prefills also 400 on these models.

Domain 5: Context Management & Reliability (15%)

Covers conversation context management, escalation patterns, error propagation, large codebase context strategies, human review workflows, and information provenance.

5.1 Conversation Context Management

The Context Window Is Finite

Claude has a limited context window. Every message, tool result, and system prompt consumes tokens. Long conversations degrade as earlier information gets compressed or lost.

Current sizes, because "1M tokens" changes the shape of the problem without eliminating it:

Model	Context window	Max output
Claude Fable 5.1 / Fable 5 / Opus 5 / Sonnet 5	1M tokens	128k
Claude Opus 4.8 / 4.7 / 4.6, Sonnet 4.6	1M tokens	128k
Claude Haiku 4.5	200k tokens	64k

Two traps that come with a 1M window:

1. **A bigger window is not free.** Every token in it is billed on every turn, and the "lost in the middle" effect below gets *worse* with length, not better. Filling a 1M window because you can is a cost and a quality regression.
2. **Token counts changed.** Opus 4.7 introduced a new tokenizer, carried by Opus 4.8, Opus 5, Sonnet 5, and Fable 5 / 5.1: the same text produces roughly 30% more tokens than on pre-4.7 models (1M tokens ≈ 555k words now, versus ≈ 750k before). Any budget, chunk size, or threshold calibrated on an older model needs re-baselining with `messages.count_tokens` — never with `tiktoken`.

Progressive Summarization Risks

When context is compressed (via `/compact` or automatic compression), the system summarizes earlier conversation. This introduces risks:

What gets lost in summarization:

- Exact numerical values (`$149.99` becomes "approximately $150")
- Specific dates (`2024-01-15` becomes "mid-January")
- Precise percentages (`23.7%` becomes "about 24%")
- Order numbers, customer IDs, reference codes
- Exact error messages and stack traces

Solution: Case Facts Block

Extract critical transactional data into a persistent structured block that survives summarization:

```
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)
```

The "Lost in the Middle" Effect

Models process the **beginning** and **end** of long inputs more reliably than the **middle**. Information placed in the middle of a large context is more likely to be overlooked.

Mitigation strategies:

1. Place key findings summaries at the **beginning** of aggregated inputs
2. Use explicit section headers to create navigation structure
3. Keep critical information out of the middle of long documents
4. If you must include lots of content, bookend the important parts

Tool Output Trimming

Tool results can be verbose. A customer order lookup might return 40 fields when you only need 5.

Bad: Return all 40 fields, consuming context for nothing.

Good: Trim to relevant fields before returning to Claude:

```
{
  "order_id": "67890",
  "status": "delivered",
  "delivery_date": "2024-01-20",
  "total": 149.99,
  "items": ["Widget Pro (x2)"]
}
```

What Survives /compact

Survives	May Be Lost
CLAUDE.md (re-read 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

---

5.2 Escalation Patterns

When to Escalate Immediately

Trigger	Why	Action
Customer explicitly requests a human	Respecting user autonomy	Escalate immediately, no further investigation
Policy gap or exception	Agent can't make policy decisions	Escalate with structured context
No progress after reasonable attempts	Agent is stuck	Escalate with summary of what was tried

When NOT to Escalate

Bad Trigger	Why It's Wrong
High sentiment/emotion	Sentiment ≠ complexity. An angry customer may have a simple issue.
Self-reported low confidence	Claude's confidence scores are poorly calibrated. A "low confidence" response might be correct, while a "high confidence" response might be wrong.
Long conversation	Length doesn't indicate need for escalation. Some issues legitimately take multiple turns.

The Sentiment ≠ Complexity Distinction

This is a specifically tested concept:

```
"I'M SO FRUSTRATED WITH THIS!!!" + simple billing error
→ Agent can resolve. Don't escalate based on caps and exclamation marks.

"I have a question about the interaction between your enterprise SLA
and the new EU data residency requirements" + calm tone
→ May need escalation due to policy complexity, despite calm sentiment.
```

Multiple Customer Matches

When a customer lookup returns multiple matches:

Wrong: Pick the most likely match using heuristics. **Right:** Ask the customer for additional identifying information to disambiguate.

```
"I found multiple accounts matching that name. Could you provide
your email address or the last four digits of your phone number
so I can locate the right account?"
```

Structured Escalation Handoff

When escalating, provide a structured summary because the human agent typically can't see the conversation transcript:

```
ESCALATION SUMMARY
==================
Customer: Jane Doe (#12345)
Contact: jane@example.com | 555-0123
Issue type: Refund request — damaged product
Order: #67890 ($149.99, delivered 2024-01-20)

Actions taken by agent:
1. Verified customer identity
2. Confirmed order and delivery details
3. Customer provided photos of damage
4. Refund amount ($149.99) exceeds automated approval limit ($100)

Recommended action: Manual approval for full refund + replacement
Urgency: Standard (customer is frustrated but not at risk of churn)
```

5.3 Error Propagation

Structured Error Context

When errors occur, propagate them with full context:

```
{
  "failure_type": "timeout",
  "service": "knowledge_base_search",
  "attempted_query": "return policy for electronics over $500",
  "timeout_duration_ms": 30000,
  "partial_results": null,
  "alternative_approaches": [
    "Try searching with simpler terms",
    "Check the FAQ section directly",
    "Ask the customer for the specific product"
  ],
  "is_retryable": true,
  "retry_after_ms": 5000
}
```

This gives Claude enough information to make an intelligent recovery decision.

Anti-Patterns

1. Generic error messages:

```
{ "error": "Operation failed" }
```

Claude can't recover intelligently because it doesn't know what failed or why.

2. Silent suppression:

```
{ "results": [] }  // Returned when the search engine was actually down
```

Claude will confidently tell the user "I couldn't find any information about that" when the truth is "I wasn't able to search." These are fundamentally different situations.

3. Workflow termination:

```
Step 1: Get customer info → Success
Step 2: Search knowledge base → Timeout
Step 3: ABORT ENTIRE WORKFLOW
```

Better: Continue with partial results, note the gap, and offer alternatives.

Access Failure vs Empty Result

Situation	HTTP Status	Meaning	Agent Response
Valid search, no matches	200 + empty array	No relevant content exists	"I didn't find any matching articles"
Search service down	503 / timeout	Search didn't execute	"I wasn't able to search right now, let me try another approach"

Subagent Error Recovery

Subagents should implement **local recovery** for transient failures:

```
Subagent: Search for "return policy"
  → Timeout
  → Retry with backoff (local recovery)
  → Success on second attempt
  → Return results to coordinator
```

Only propagate errors the subagent cannot resolve. When propagating, include:

- What was attempted
- What partial results were obtained
- What the failure was
- Whether it's retryable

5.4 Large Codebase Context Management

Context Degradation in Extended Sessions

In long sessions, Claude's reliability degrades:

- References "typical patterns" instead of specific code found earlier
- Gives inconsistent answers about the same code
- Forgets earlier discoveries
- Starts making assumptions instead of checking

Mitigation Strategies

1. Scratchpad Files

Persist findings to files that survive context compression:

```
# scratchpad.md
## Auth System Findings
- Entry point: src/auth/middleware.ts:23
- Uses JWT with RS256
- Token expiry: 1 hour
- Refresh token: 7 days
- Role check: src/auth/roles.ts:45
- Known issue: No token revocation mechanism
```

When context is compressed, Claude can re-read the scratchpad to recover findings.

2. Subagent Delegation

Isolate verbose exploration in subagents:

```
Main context: Clean, focused on the task
  ↓
Explore subagent: Reads 50 files, searches codebase extensively
  ↓
Returns: Concise summary of findings
  ↓
Main context: Receives only the summary, stays clean
```

3. `/compact` Proactive Use

During extended sessions, proactively compact context before it degrades naturally. Better to compact with a structured summary than to let automatic compression lose information unpredictably.

4. Crash Recovery Pattern

For long-running multi-agent workflows:

```
# Each agent exports state to a known location
agent.save_state("/tmp/workflow/agent-1-state.json")

# Coordinator maintains a manifest
manifest = {
    "agent-1": {"status": "complete", "state": "/tmp/workflow/agent-1-state.json"},
    "agent-2": {"status": "in_progress", "state": "/tmp/workflow/agent-2-state.json"},
    "agent-3": {"status": "pending"}
}

# On resume, coordinator loads manifest and picks up where it left off
```

5.5 Human Review Workflows

Stratified Random Sampling

Don't just sample randomly from all outputs. Stratify by risk or category:

```
Category A (high-risk): Sample 20% of outputs
Category B (medium-risk): Sample 10% of outputs
Category C (low-risk): Sample 5% of outputs
```

This concentrates human review effort on areas most likely to have errors.

Field-Level Confidence Scores

Instead of a single confidence score for the entire extraction, provide per-field confidence:

```
{
  "vendor_name": { "value": "Acme Corp", "confidence": 0.99 },
  "total_amount": { "value": 149.99, "confidence": 0.95 },
  "purchase_order": { "value": "PO-2024-001", "confidence": 0.72 },
  "tax_id": { "value": "12-3456789", "confidence": 0.45 }
}
```

Calibration: Confidence scores must be calibrated against labeled validation sets. A 0.9 confidence should mean the field is correct ~90% of the time. Without calibration, confidence scores are meaningless.

Accuracy by Document Type

The hidden failure pattern:

```
Overall accuracy: 97% ← Looks great!

Breakdown:
  Standard invoices: 99.5% ← Excellent
  Handwritten notes: 45%  ← Terrible
  Foreign language docs: 62% ← Bad
```

Aggregate metrics mask poor performance on specific document types. Always break down accuracy by type and field.

Per-Document-Type Reporting

Report accuracy metrics for each document type AND each field within that type:

```
Standard Invoices:
  vendor_name: 99.8%
  total_amount: 99.2%
  line_items: 97.5%
  tax_id: 94.1%

Handwritten Notes:
  vendor_name: 85.0%
  total_amount: 42.0% ← Critical failure
  line_items: 38.0% ← Critical failure
```

This reveals where the system actually needs improvement rather than hiding behind an aggregate number.

5.6 Information Provenance

Claim-Source Mappings

When subagents produce findings, they should output structured provenance:

```
{
  "claim": "Revenue increased 15% year-over-year in Q4 2024",
  "evidence": "Q4 earnings report, page 3, paragraph 2",
  "source_url": "https://example.com/earnings/q4-2024",
  "document_name": "Q4 2024 Earnings Report",
  "publication_date": "2025-01-15",
  "source_type": "primary"
}
```

Handling Conflicting Data

When sources disagree, **annotate the conflict** rather than silently picking one:

```
{
  "field": "Q4 revenue",
  "values": [
    {
      "value": "$2.3B",
      "source": "Company press release (2025-01-15)",
      "note": "Preliminary figures"
    },
    {
      "value": "$2.28B",
      "source": "SEC filing (2025-02-28)",
      "note": "Audited figures"
    }
  ],
  "resolution": "SEC filing is the authoritative source (audited)",
  "conflict_type": "minor_discrepancy"
}
```

Anti-pattern: Silently selecting one value without noting the disagreement. This hides potential data quality issues.

Temporal Data

Include publication and collection dates to prevent temporal confusion:

```
{
  "claim": "Unemployment rate is 3.7%",
  "data_as_of": "2024-11-01",
  "source_published": "2024-12-06",
  "retrieved_on": "2025-01-10"
}
```

Without dates, a reader might think two different unemployment figures from different months represent a contradiction when they're actually both correct for their respective time periods.

Synthesis Output Quality

When producing research synthesis:

1. **Distinguish certainty levels:**
    - "Well-established: Revenue grew 15% (confirmed by SEC filing and two analyst reports)"
    - "Contested: Market share estimates range from 12% to 18% depending on methodology"
    - "Single-source: Only one report mentions the planned acquisition"
2. **Preserve source characterization:**
    - Don't merge or reinterpret what sources say
    - Quote or closely paraphrase the original claim
    - Note the source's own caveats and qualifications
3. **Render appropriately by content type:**
    - Financial data → tables with source attribution
    - News/events → chronological prose
    - Technical specifications → structured lists
    - Disputed claims → side-by-side comparison

5.7 Server-Side Context Management

Sections 5.1 and 5.4 cover context management as something *you* do — case-facts blocks, scratchpads, trimming, delegating to subagents. The API now offers three server-side mechanisms that do part of this work for you. They are complementary, not alternatives, and knowing which one solves which problem is the point.

Mechanism	What it does	Where state goes
Context editing	*Clears* old tool results or thinking blocks before the model sees them	Deleted — gone
Compaction	*Summarizes* earlier context when it approaches a threshold	Replaced by a summary in the conversation
Memory tool	Lets Claude read and write persistent files	Outside the conversation, survives everything

Context Editing (beta `context-management-2025-06-27`)

Clears, does not summarize.

```
client.beta.messages.create(
    model="claude-opus-5",
    betas=["context-management-2025-06-27"],
    context_management={"edits": [
        {"type": "clear_thinking_20251015",
         "keep": {"type": "thinking_turns", "value": 2}},
        {"type": "clear_tool_uses_20250919",
         "trigger": {"type": "input_tokens", "value": 30000},
         "keep": {"type": "tool_uses", "value": 3},
         "clear_at_least": {"type": "input_tokens", "value": 5000},
         "exclude_tools": ["web_search"]},
    ]},
    tools=[...], messages=[...],
)
```

- `clear_tool_uses_20250919` — clears old tool results (`clear_tool_inputs: true` also clears the parameters). Default trigger: 100,000 input tokens; default `keep`: 3.
- `clear_thinking_20251015` — clears thinking blocks. When combining both, **`clear_thinking` must be listed first**.
- `clear_at_least` exists to protect the prompt cache: clearing a trivial amount invalidates the cache for no benefit.
- The response reports what happened in `context_management.applied_edits`.

Compaction (beta `compact-2026-01-12`)

Summarizes rather than clears. Available on Fable 5 / 5.1, Mythos 5 / 5.1, Opus 5, Opus 4.8/4.7/4.6, Sonnet 5, and Sonnet 4.6; default trigger around 150k tokens. An `instructions` parameter accepts your own summarization prompt.

> **The critical integration detail:** append the **whole** `response.content` back to your `messages` on every turn, not just the extracted text. The compaction blocks in the response are what the API uses to replace compacted history on the next request. Pulling out the text string and appending that silently destroys the compaction state — and it fails quietly, which is the worst failure mode for something you only notice at turn 90.

Client-side SDK compaction (`compaction_control` on the tool runner) is **deprecated** in favor of this.

Preserved Thinking: Compaction Shapes That Stay Valid

Fable 5.1 adds a constraint the other mechanisms didn't have: its thinking blocks are valid only against the exact history that preceded them, so **editing earlier turns invalidates every later block** (Domain 4 §4.7). Server-side compaction and context editing don't count as edits — the check compares the conversation *as you sent it* — which is now the strongest argument for moving trimming to the server. If you must compact on the client, only three shapes survive:

Shape	Rule
Simple compaction (recommended)	Replace the whole history with one summary message plus the new user turn; replay nothing else. No thinking blocks carry over, so nothing fails
Keep-tail compaction	If recent turns stay verbatim behind a summary, strip their `thinking` / `redacted_thinking` blocks (text and tool calls can stay), or send `prefix_mismatch_behavior: "drop_block"`
Background compaction	A summary swapped in later invalidates every block produced in between — send `"drop_block"` on each request that still carries pre-swap thinking, or compact synchronously

What never works: snipping individual turns out of the middle of the transcript, deleting old tool results by hand, or rebuilding `system` / `tools` between requests. Use a mid-conversation `role: "system"` message for the instruction change you were making, and server-side context editing for selective removal. Dropping blocks once at a compaction boundary is cheap; invalidating them on *every* request restarts the prompt cache each time.

Memory Tool (`memory_20250818`)

A client-side tool — Anthropic defines the interface, you implement the file operations. Declared as `{"type": "memory_20250818", "name": "memory"}`, no `input_schema`.

Its role in this domain: it's the persistence layer that makes clearing safe. Combine it with context editing and Claude receives a warning to write anything important to memory *before* its tool results are cleared. That is the API-level version of the scratchpad pattern in §5.4.

Choosing

Situation	Mechanism
Agentic loop with dozens of verbose tool results	Context editing (`clear_tool_uses`)
Extended thinking plus a need to keep cache hits high	Context editing (`clear_thinking`)
Long research/analysis conversation that must keep its narrative	Compaction
Findings that must survive across sessions, not just turns	Memory tool
Facts that must never be paraphrased	Case-facts block (§5.1) — no mechanism protects exact numbers better than restating them

---

5.8 Bounding and Recovering Long Runs

Task Budgets vs `max_tokens`

`max_tokens` is a ceiling the model can't see; hitting it truncates output mid-thought. A **task budget** (`output_config.task_budget`, beta `task-budgets-2026-03-13`, minimum 20,000) is a ceiling the model *can* see, so it paces itself and lands the work. Available on Claude Fable 5.1 / Mythos 5.1, Fable 5 / Mythos 5, Opus 5, and Opus 4.8/4.7 — **not** on Sonnet 5, Opus 4.6, or Haiku 4.5. The budget counts what Claude generates plus the tool results it reads this turn — not the full history you resend. Leave `remaining` unset in a normal loop; only pass it when you rewrite or compact history yourself and the server can no longer derive prior spend. Two reliability footnotes: a budget that is obviously too small for the task makes Claude decline, scope down, or stop early with a partial result (raise the budget before debugging anything else), and the budget value is rendered into the prompt, so changing it mid-task is a cache miss.

Managed Agents **session budgets** are a different thing: hard, dollar-denominated, platform-enforced caps on one session. A task budget is advisory and token-denominated.

Refusals Are a Reliability Concern, Not Just a Safety One

A `refusal` arrives as HTTP 200 with `stop_reason: "refusal"` and a `stop_details.category`. Code that reads `content` without checking `stop_reason` treats a refusal as a successful empty answer — the same class of bug as treating a failed search as zero results (§5.3). For

production paths, enable server-side fallback (`betas=["server-side-fallback-2026-07-01"]`, `fallbacks="default"`) so the request is re-routed by category rather than dropped. See Domain 1 §1.1. Mythos 5.1 now runs classifiers too (Mythos 5 did not), so the same handling applies under Project Glasswing; and a fallback *from* Fable 5.1 lands on a model that can't read its thinking blocks, so budget for a re-planning turn.

Domain 5 Practice Questions

Q1: 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

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

Q2: 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

Answer: C — 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.

Q3: 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

Answer: B — 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.

Q4: 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

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

Q5: 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

Answer: B — 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.

Q6: 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

Answer: B — `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.

Q7: 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`) or compaction, which don't count as edits
- C) Switch `thinking` to `{"type": "disabled"}`
- D) Delete the tool results *and* the thinking blocks that follow them

Answer: B — 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. A always works but throws away the reasoning and restarts the prompt cache on every request; C is a 400 on Fable 5.1 (thinking is always on); D still edits the middle of the transcript.

Claude Certified Associate — Foundations (CCAO-F)

Overview guide for the **Claude Certified Associate — Foundations** exam — the entry-level, non-developer credential in the Claude certification family. Facts below are taken from the official Exam Guide v1.0 (effective July 2026).

> Looking for the Architect exam? The full deep-dive study guide in this repo covers CCAR-F. This page is an overview of a sibling certification.

What It Validates

That you can apply Claude to complete business and productivity tasks with minimal guidance: using built-in platform features and tools to streamline workflows, identifying opportunities to improve processes, selecting approaches that balance quality, efficiency, and cost, and recognizing limitations — escalating more complex or technical work to Claude Architects and Developers.

Intended for: professionals who use Claude as a productivity tool — operations, marketing, project management, education, communications, and general knowledge work. Candidates typically have limited to moderate technical expertise, positioned between casual prompt users and technical AI practitioners.

Not intended for: software developers building against APIs or designing agentic systems — that scope belongs to the Developer and Architect credentials.

Exam at a Glance

Exam code	CCAO-F
Items	60 (multiple-choice and multiple-response; each item states how many responses to select)
Time limit	120 minutes
Delivery	Proctored — online proctored and/or Pearson VUE test center
Passing score	720 scaled (scale 100–1,000)
Exam fee	$99 USD
Validity	12 months from award
Result reporting	Pass/fail with scaled score, plus percent-correct by domain
Prerequisites	None mandatory — recommended experience only

Domain Blueprint

#	Domain	Weight
1	Prompting and Task Execution	14%
2	Output Evaluation and Validation	21%
3	Product and Model Selection	12%
4	Workflow Integration and Solution Design	16%
5	Configuration and Knowledge Management	12%
6	Governance, Risk, and Responsible Use	15%
7	Troubleshooting and Optimization	10%

Note the emphasis: output evaluation (21%), workflow design (16%), and governance (15%) together make up more than half the exam — this credential tests **judgment about using AI responsibly**, more than prompting technique.

Key objectives per domain

1. **Prompting and Task Execution** — effective prompts for business tasks, task decomposition, iterating prompts, adapting strategy by task type (analysis, research, drafting, brainstorming).
2. **Output Evaluation and Validation** — evaluating accuracy and completeness, spotting hallucinations/inconsistencies/bias, fact-checking, knowing when human review is required, adapting outputs for an audience, choosing output formats (artifacts, inline, structured data).
3. **Product and Model Selection** — choosing product features (Projects, research mode, chat, artifacts), differentiating model tiers (Haiku, Sonnet, Opus), aligning model choice with cost/speed/quality, managing context limits and memory (when to restart, summarize, or persist).
4. **Workflow Integration and Solution Design** — applying Claude to requirements analysis, research, planning, process optimization; integrating Claude into existing workflows; communicating Claude's value and limitations to stakeholders.
5. **Configuration and Knowledge Management** — configuring Claude Projects with instructions and knowledge sources, managing connectors (e.g. Google Drive, Gmail), writing effective system-level instructions, maintaining configurations over time.
6. **Governance, Risk, and Responsible Use** — appropriate vs inappropriate use cases, data sensitivity/regulatory/privacy considerations, organizational AI policy compliance, ethical implications.
7. **Troubleshooting and Optimization** — diagnosing underperforming prompts and poor outputs, adjusting based on feedback, optimizing workflows.

How to Prepare (per the official guide)

- Study the blueprint and self-assess against each objective
- Review official documentation for Projects, Artifacts, Memory, Skills, and Code Execution
- Practice structuring prompts, decomposing tasks, and iterating on outputs
- Build a real workflow: configure a Project with instructions and knowledge sources, then evaluate outputs for accuracy and bias
- Practice responsible-use judgment: data sensitivity, appropriate use cases, when to escalate or seek human review

Much of this repo's Domain 4 (explicit criteria, few-shot) and Domain 5 (context management, escalation, provenance) material transfers directly — at a less technical depth.

Registration and Policies

1. Register via the exam's page on the **Anthropic Partner Academy** (partner-tier discounts apply at checkout), then schedule through **Pearson VUE** — online proctoring or a test center.
2. Cancel/reschedule up to 24 hours before the appointment; changes within 24 hours forfeit the fee.
3. Retakes: up to 4 attempts per rolling 12 months, with waiting periods after each failed attempt (14 / 30 / 90 days), per Pearson VUE program policy.
4. Government-issued photo ID required; the name must exactly match your registration.
5. Renewal: the credential is valid 12 months; renewing on time is free — review what changed since you certified and pass a non-proctored assessment. If it lapses, the full exam fee applies again. Registration requires a partner-domain email address.

Resources

- Official CCAO-F Exam Guide (PDF, v1.0)
- Pearson VUE — Anthropic certification program
- Anthropic Skilljar courses

Claude Certified Architect – Professional (CCAR-P)

Overview guide for the **Claude Certified Architect – Professional** exam – the advanced tier above the Architect Foundations exam this repo covers in depth. Facts below are taken from the official Exam Guide v1.0 (effective July 2026).

> Preparing for Foundations first? Start with the full CCAR-F study guide – Anthropic's Professional blueprint assumes you operate comfortably at that level.

What It Validates

That you can design, build, and deliver production-grade AI solutions on the Claude platform as an architect who owns the full lifecycle: selecting models, architectures, and API patterns; prompt and context engineering; integrating Claude into enterprise systems; and incorporating evaluation, security, compliance, and governance into designs – plus the distinctly Professional-level skill of **stakeholder communication and lifecycle management**.

Intended for: mid- to senior-level solution architects, AI/ML engineers, and technical leads. Recommended (not required): 3+ years in systems architecture or platform engineering, 6+ months hands-on with Claude (or comparable LLM systems) in production, experience delivering end-to-end systems from discovery through operationalization.

Not intended for: entry-level developers, casual users, or roles limited to isolated tasks (e.g. prompt writing) without broader system-design responsibility.

Exam at a Glance

Exam code	CCAR-P
Items	63 (multiple-choice and multiple-response; each item states how many responses to select)
Time limit	120 minutes
Delivery	Proctored – online proctored and/or Pearson VUE test center
Passing score	720 scaled (scale 100–1,000)
Exam fee	$175 USD
Validity	12 months from award
Result reporting	Pass/fail with scaled score, plus percent-correct by domain
Prerequisites	None mandatory – recommended experience only

Domain Blueprint

Seven domains vs Foundations' five. The Foundations themes reappear at higher altitude, and two areas are effectively new: RAG-heavy Integration, and Stakeholder Communication & Lifecycle Management.

#	Domain	Weight
1	Solution Design & Architecture	17%
2	Claude Models, Prompting & Context Engineering	13%
3	Integration	19%
4	Evaluation, Testing & Optimization	16%
5	Governance, Safety & Risk Management	14%
6	Stakeholder Communication & Lifecycle Management	14%
7	Developer Productivity & Operational Enablement	7%

Key objectives per domain

1. **Solution Design & Architecture (17%)** — translating business problems into Claude-based solutions; end-to-end architectures (input → processing → output → feedback loops); choosing among workflow, agentic, and augmented-LLM patterns; multi-agent systems and orchestration; decomposition; aligning to business value pillars (efficiency, cost, performance SLAs).
2. **Claude Models, Prompting & Context Engineering (13%)** — model selection by trade-off; system prompts, templates, guardrails; zero-shot/few-shot/chain-of-thought; context-window and token optimization; prompt reuse (caching, modular prompts, Skills).
3. **Integration (19%)** — the heaviest domain: tool/agent capability-bloat evaluation, authn/authz gap analysis, accuracy-latency trade-offs, observability at scale, **RAG pipeline design** (chunking, indexing, retrieval matched to data shape and query pattern), choosing integration mechanisms (MCP vs API/CLI vs agent-to-agent), progressive-discovery vs monolithic context strategy.
4. **Evaluation, Testing & Optimization (16%)** — evaluation metrics (accuracy, latency, cost, safety, security), eval datasets and test frameworks, A/B testing, diagnosing prompt failure/hallucination/model mismatch, token/latency/cost optimization, logging and observability.
5. **Governance, Safety & Risk Management (14%)** — guardrails and safety controls, LLM failure modes, human-in-the-loop validation, regulatory compliance (GDPR, HIPAA, FedRAMP), ethical AI (bias, fairness, transparency).
6. **Stakeholder Communication & Lifecycle Management (14%)** — structured discovery and requirements gathering, communicating architectural decisions and trade-offs, feedback loops and expectation/SLA alignment, architecture documentation, supporting lifecycle phases (discovery, design, handoff, monitoring, iteration). This domain does not exist on CCAR-F.
7. **Developer Productivity & Operational Enablement (7%)** — configuring Claude tooling for teams (e.g. Claude Code), improving developer workflows with AI-assisted tooling, supporting debugging and operational issues.

Sample-question flavor (from the official guide)

The published samples test least-privilege tool configuration (remove unneeded refund/delete tools rather than log or confirm), prompt-caching architecture (stable prefix first + caching to cut latency *and* cost), and RAG debugging (confident-but-wrong answers after a document refresh → suspect retrieval/indexing first). Expect judgment-under-tradeoffs questions, not recall.

Relationship to CCAR-F and This Repo

- All five CCAR-F domains reappear inside CCAR-P domains 1–5 — the existing deep-dives remain the right base layer.
- Study the deltas on top: **RAG architecture** (chunking/indexing/retrieval strategy), **evaluation frameworks and A/B testing**, **regulated-industry compliance**, and **stakeholder communication / lifecycle management** — none of which the Foundations guide covers today.
- Anthropic recommends sitting CCAR-F first unless you already have deep, current production experience.

How to Prepare (per the official guide)

- Study the blueprint and self-assess against each objective
- Review official documentation for the Claude API, models, prompt engineering, MCP, and Skills
- Build and operate at least one end-to-end Claude solution including RAG, evaluation, and observability
- Practice architectural decision-making: model selection, integration protocols, security trade-offs

---

Registration and Policies

1. Register via the exam's page on the **Anthropic Partner Academy** (partner-tier discounts apply at checkout), then schedule through **Pearson VUE** — online proctoring or a test center.
2. Cancel/reschedule up to 24 hours before the appointment; changes within 24 hours forfeit the fee.
3. Retakes: up to 4 attempts per rolling 12 months, with waiting periods after each failed attempt (14 / 30 / 90 days), per Pearson VUE program policy.
4. Renewal: the credential is valid 12 months; renewing on time is free — review what changed since you certified and pass a non-proctored assessment. If it lapses, the full exam fee applies again. Registration requires a partner-domain email address.

Resources

- Official CCAR-P Exam Guide (PDF, v1.0)
- Pearson VUE — Anthropic certification program
- CCAR-F study guide in this repo — the base layer

Claude Certified Developer — Foundations (CCDV-F)

Overview guide for the **Claude Certified Developer — Foundations** exam — the hands-on implementation credential in the Claude certification family. Facts below are taken from the official Exam Guide v1.0 (effective July 2026).

> Looking for the Architect exam? The full deep-dive study guide in this repo covers CCAR-F. This page is an overview of a sibling certification — with heavy content overlap, mapped below.

What It Validates

That you can build, integrate, and ship production-grade applications, agents, and workflows on the Claude platform at a foundational level: building agents with the Claude Agent SDK and custom agent loops, integrating Claude through the API and client SDKs (streaming, error handling, multi-format input), operating Claude Code (Skills, CLAUDE.md, settings.json, plugins), prompt and context engineering, designing and running evals, model/cost/latency optimization, secure-by-design practices and hook-based guardrails, and building custom tools and MCP servers.

Intended for: AI/ML engineers, technical leads, and senior software engineers. Recommended (not required): 1–5 years of software engineering, 6+ months hands-on with Claude or comparable LLM systems, Python and/or TypeScript, REST APIs and CLI fluency.

Not intended for: non-technical users, or roles limited to prompt writing without broader application-development responsibility.

Exam at a Glance

Exam code	CCDV-F
Items	53 (multiple-choice and multiple-response; each item states how many responses to select)
Time limit	120 minutes
Delivery	Proctored — online proctored and/or Pearson VUE test center
Passing score	720 scaled (scale 100–1,000)
Exam fee	$125 USD
Validity	12 months from award
Result reporting	Pass/fail with scaled score, plus percent-correct by domain
Prerequisites	None mandatory — recommended experience only

Domain Blueprint

Unlike the Architect Foundations exam (5 evenly weighted domains), CCDV-F has 8 domains with very uneven weights — a third of the exam is Applications and Integration:

#	Domain	Weight
1	Agents and Workflows	14.7%
2	Applications and Integration	33.1%
3	Claude Code	3.1%
4	Eval, Testing, and Debugging	2.6%
5	Model Selection and Optimization	16.8%
6	Prompt and Context Engineering	11.0%
7	Security and Safety	8.1%
8	Tools and MCPs	10.6%

What each domain covers

1. **Agents and Workflows (14.7%)** — workflow-vs-agent decision criteria, manager/supervisor hierarchies, subagents; building with the Claude Agent SDK, custom loops and harnesses, hosted vs self-hosted deployment, hooks for deterministic actions; agent design patterns (tool-use loops, sub-agents, memory, context-window management) and frameworks.
2. **Applications and Integration (33.1%)** — the big one. Requirements and systems-lifecycle basics; Claude API mechanics (messages, tools, streaming, vision, thinking, caching, third-party vendors, batch vs realtime); software-engineering foundations (REST, JSON, async, version control, refactoring); application design (how Claude interprets instructions across Claude Code / Desktop / claude.ai / API / SDKs, content boundaries, schema design, session hygiene); configuration management (CLAUDE.md, settings.json, model pinning, prompt versioning).
3. **Claude Code (3.1%)** — core components (Rules, Skills, Commands, Agents, Agent Memory), session management, slash commands, headless mode, streaming mode, auto-mode, the CLAUDE.md hierarchy, settings.json.
4. **Eval, Testing, and Debugging (2.6%)** — error-type identification, recovery strategies, trace analysis, isolating integration-layer vs model-output problems.
5. **Model Selection and Optimization (16.8%)** — LLM fundamentals (tokens, context windows, sampling, non-determinism), model options (fast mode, extended/adaptive thinking, effort levels), model-tier tradeoffs and breaking changes across releases, token budgeting, prompt caching for cost.
6. **Prompt and Context Engineering (11.0%)** — context-drift/bloat prevention (tool-output pruning, compaction), context isolation via subagents, prompt engineering principles, structured-output patterns, defensive parsing, skepticism toward confident output.
7. **Security and Safety (8.1%)** — prompt-injection awareness and mitigation, jailbreak defense, untrusted input, PII, guardrail layering, hooks as safety controls, secrets and key management.
8. **Tools and MCPs (10.6%)** — tool use and function calling, tool-description writing, client-side vs server-side tools, approval patterns, MCP server authoring and deployment, stdio vs socket transports.

Overlap with This Repo's CCAR-F Material

A large share of CCDV-F content is already covered in the Architect guide, at architect altitude:

CCDV-F domain	Covered in
Agents and Workflows	d1 — Agentic Architecture
Tools and MCPs	d2 — Tool Design & MCP
Claude Code	d3 — Claude Code Configuration
Prompt and Context Engineering	d4 — Prompt Engineering + d5 — Context Management
Applications and Integration / Model Selection	Partially in d1/d4; the API-mechanics and cost-optimization depth is CCDV-F-specific

The developer exam goes deeper on API mechanics, SDK usage, evals, security, and cost optimization than the Architect Foundations blueprint does.

How to Prepare (per the official guide)

- Study the blueprint and self-assess against each skill
- Build and ship at least one real Claude application end-to-end: API integration, an agent loop or Agent SDK build, custom tools or an MCP server, and an eval
- Review official documentation for the Claude API, Agent SDK, Claude Code, and MCP
- Practice cost levers hands-on: model-tier selection, prompt caching, batch processing

---

Registration and Policies

1. Register via the exam's page on the **Anthropic Partner Academy** (partner-tier discounts apply at checkout), then schedule through **Pearson VUE** — online proctoring or a test center.
2. Cancel/reschedule up to 24 hours before the appointment; changes within 24 hours forfeit the fee.
3. Retakes: up to 4 attempts per rolling 12 months, with waiting periods after each failed attempt (14 / 30 / 90 days), per Pearson VUE program policy.
4. Renewal: the credential is valid 12 months; renewing on time is free — review what changed since you certified and pass a non-proctored assessment. If it lapses, the full exam fee applies again. Registration requires a partner-domain email address.

Resources

- Official CCDV-F Exam Guide (PDF, v1.0)
- Pearson VUE — Anthropic certification program
- Claude API docs · Claude Code docs · Agent SDK · MCP