

# Claude Certified Architect (CCA-F): Quick Reference

This is a condensed companion to the excellent study guide at [github.com/paullarionov/claude-certified-architect](https://github.com/paullarionov/claude-certified-architect). Start there for full, structured coverage of every exam topic - this sheet won't make much sense unless you already understand the topics.

The purpose here is to provide a tight collection of the key facts and concepts worth holding in working memory, aimed at people who already understand the underlying architecture.

*Caveat emptor*: it is deliberately not exhaustive. I left out anything I already knew cold, so use it as one input alongside your wider study, not as the whole of it.

## Claude API: stop\_reason

---

Every response carries a `stop_reason` telling you why model generation halted. For agent loops, the `tool_use` vs `end_turn` distinction is what drives the loop.

- `end_turn` — Claude has finished; surface the reply to the user.
- `tool_use` — Claude wants a tool run; execute it and feed the result back.
- `max_tokens` — the output cap was hit; the reply is truncated, so consider raising the limit.
- `stop_sequence` — a configured stop string was encountered.

## Tools: tool\_choice

---

Controls whether and how the model reaches for a tool:

- `auto` — the model's decision: answer in text or invoke a tool (the usual default).
- `any` — the model *must* use *some* tool; handy when you need guaranteed structured output.
- `{"type": "tool", "name": ...}` — pins the first call to one named tool, forcing an execution order (e.g. extract before enrich).
- Spawning sub-agents: the coordinator's tool list must include `Task` before it can delegate.

## JSON schemas for structured output

---

Calling a tool with a JSON schema is the sturdiest route to structured output. It pins down the shape but not the meaning:

- Output is always well-formed JSON — braces close, no stray commas.
- The structure is enforced, so required fields are present.
- Values are not fact-checked, though — they can still be wrong.

Schema design rules:

- Required vs optional — mark a field required only when it is genuinely always present; otherwise the model may invent a value to fill the gap.
- Allow nulls — `"type": ["string", "null"]` lets the model return null for missing data instead of hallucinating one.
- Open enums — pair your categories with an `"other"` option plus a free-text detail so anything outside the list is not silently dropped.
- An `"unclear"` enum value — give the model an honest escape hatch when it cannot confidently classify; better than a confident wrong label.

## MCP — configuration and errors

---

Where server config lives:

- `.mcp.json` — shared/team servers, at the project root, committed to version control.
- `~/.claude.json` — personal or throwaway servers, in your home directory, kept out of version control.

Error signalling: a failed call returns `isError: true`. Pair it with a structured error object (category, retryable flag, message) so the agent can decide whether to retry, rephrase, or escalate — a bare "operation failed" string gives it nothing to act on.

## Claude Code — CLAUDE.md hierarchy

---

Three levels of instruction:

- User — `~/.claude/CLAUDE.md`: applies to you alone, never committed; personal style and preferences.
- Project — `.claude/CLAUDE.md` or a root `CLAUDE.md`: applies to everyone on the repo, committed; coding and testing standards, architectural decisions.
- Directory — a `CLAUDE.md` inside a subfolder: applies while editing files there; conventions local to that part of the tree.
- Classic trap: putting project instructions in the user-level file, so new teammates never receive them.

## @path imports

---

`CLAUDE.md` can pull in other files with `@path`, keeping configuration modular:

- Both relative and absolute paths are supported.
- Imports nest up to 5 levels deep.

## The .claude/rules/ directory

---

A topic-by-topic alternative to one monolithic `CLAUDE.md` — individual `.md` files. The real lever is YAML frontmatter with a `paths` glob: a rule loads only when Claude edits a file matching the pattern, which saves context and tokens.

- Use `paths` rules when a convention spans files scattered across the tree (tests, migrations).
- Use a directory-level `CLAUDE.md` when the convention is tied to one specific folder.

## Slash commands

---

Reusable `/name` prompt templates written in markdown, committed so the team shares them:

- Legacy location: `.claude/commands/`.
- Current location: `.claude/skills/`.
- Personal, uncommitted variants live under `~/.claude/...` for individual workflows.

## Skills

---

Richer commands defined by `SKILL.md` frontmatter. Reach for them for on-demand, task-specific work, and keep always-on standards in `CLAUDE.md`. Key frontmatter:

- `context: fork` — runs in an isolated sub-agent so noisy output stays out of the main session.
- `allowed-tools` — whitelists the tools the skill may use (a safety boundary, e.g. no deletes).
- `argument-hint` — prompts for an argument when the skill is invoked bare.

## Planning mode vs direct execution

---

- Planning mode: investigate and propose only, with no edits, yielding a plan you approve. Best for large or multi-file changes, weighing several approaches, and architectural calls.
- Direct execution: just make the change. Best for single-file fixes and changes you already understand.
- Typical flow: plan first, approve, then execute.
- Explore sub-agent: a dedicated explorer that keeps its verbose digging out of the main context and returns only a summary, so long multi-phase tasks don't exhaust the main context window.

## /compact and /memory

---

- `/compact` — compresses the running context by summarising history; useful when verbose tool output fills the window, but summarising can drop precise details like exact numbers and dates.
- `/memory` — opens CLAUDE.md to edit notes, preferences and context that persist between sessions and load automatically at startup, so you aren't re-explaining the same things each time.

## CLI for CI/CD

---

- `claude -p (--print)` — non-interactive run: prompt in, result to stdout, exit. The only way to drive Claude in a pipeline such as CI/CD.
- `--output-format json` — emit JSON; `--json-schema` — validate that output against a schema.
- Do code reviews with a fresh instance: the session that wrote the code is a weaker reviewer of it, being anchored to its own reasoning. Spin up an independent one.
- To avoid repeat comments on re-review: feed in the prior review and instruct Claude to flag only new or still-open issues.

## Sessions — resume and fork

---

- `claude -r / --resume <name>` — picks a named session back up; be sure to watch for stale tool results if files have moved on since.
- `fork_session` — branches the current context into independent lines that share everything up to the split, then diverge. Good for trialling two different approaches side by side.

## Message Batches API

---

Asynchronous bulk requests with up to a 24-hour turnaround and no multi-turn tool calling. The `custom_id` field ties each result back to its source request, so when a few items fail you can resubmit just those by id rather than the whole batch.

## Production context management

---

- Don't lean on raw conversation history — it frays under summarisation. Pull the must-keep facts into a structured block and re-include that block in every prompt.
- Mind the lost-in-the-middle effect: park the critical bits at the very top or bottom of the message, not buried in the middle.
- For long investigations, have the agent write findings to a `scratchpad.md`; when context degrades or a new session starts, it reads the scratchpad instead of rediscovering everything.