MockCert

Study guide | Anthropic | CCAR-F

Claude Certified Architect – Foundations

Notes for every domain of the Claude Certified Architect – Foundations exam, organized by the official exam guide's domains and weights.

By MockCert about 100 min read Updated November 13, 2026

CONTENTS

DOMAIN 1 · 27%
Agentic Architecture & Orchestration 2

DOMAIN 2 · 18%
Tool Design & MCP Integration 18

DOMAIN 3 · 20%
Claude Code Configuration & Workflows 30

DOMAIN 4 · 20%
Prompt Engineering & Structured Output 44

DOMAIN 5 · 15%
Context Management & Reliability 58

MockCert is an independent certification preparation platform. It is not affiliated with, endorsed by or operated by Anthropic. Exam domains and weights follow the official exam guide. These notes were last updated on November 13, 2026; the notes at mockcert.com are kept up to date as the exam changes.

Domain 1 | 27% of the exam | 24 min read

Agentic Architecture & Orchestration

Overview

This domain is about 27% of the exam, roughly 16 of its 60 questions.

Every point below comes from Anthropic's official documentation; each section lists its sources.

Design and implement agentic loops for autonomous task execution

Start here

THINK FIRST

How should an agentic loop decide whether to run another tool or stop and answer?

Why it matters: Questions show agent loop code or behavior and ask what decides the next iteration, or why the loop stops at the wrong time.

An agentic loop lets Claude choose its next tool from what it has learned so far. Your code runs the tools, feeds the results back, and reads `stop_reason` to decide whether to go around again.

Key terms

Term	Meaning
stop_reason	A field in every successful response that says why Claude stopped generating.
tool_use block	Claude's tool request, with an id, the tool name and its input.
tool_result block	Your reply to a tool request, matched to it by `tool_use_id`.
Model-driven decisions	Claude picks the next tool from context, instead of following a preset tree.

The loop in one picture

Check `stop_reason`, run the tool, send its result back. Each pass adds the tool results to the conversation history. Claude then reasons about its next action with the new information in view.

Let stop_reason decide

Continue on `tool_use`; stop on `end_turn`. On `tool_use`, run the requested tool and return its result. On `end_turn`, Claude finished naturally, so present the response.

What each stop_reason tells the loop to do

stop_reason	What it means	What your loop does
`tool_use`	Claude is calling a tool	Run the tool and send back the result
`end_turn`	Claude completed its response naturally	Stop the loop and use the response

Avoid fragile stop signals

Never end the loop by reading Claude's wording. Parsing natural language for a finished signal is an anti-pattern. So is treating assistant text as proof of completion. An arbitrary iteration cap should not be the main way the loop stops.

EXAM TRAP

A response that contains text is not a sign the task is complete. Base the loop on `stop_reason`: continue on `"tool_use"` and stop on `"end_turn"`.

Return results correctly

Answer every `tool_use` block in one user message. That message holds only `tool_result` blocks, and the request keeps the same tools array. Match each result to its call with `tool_use_id`.

EXAM TRAP

Do not add a text block right after tool results. It teaches Claude to expect user input after every tool use, so it ends its turn.

Model-driven, not scripted

Claude chooses each next tool from the current context. A preset decision tree fixes the tool order in advance. An agentic loop lets each result shape the next step. Where an order is critical business logic, enforce it in code instead.

Watch the context grow

Every turn and tool result stays in the context window. Tool definitions and piled-up results use up space. A long agent with many tools or turns can run out of context before it finishes.

Thinking inside the loop

With thinking on, return thinking blocks complete and unchanged. A tool-use loop counts as one assistant turn, so change the thinking setting only between turns. Interleaved thinking lets Claude reason about each tool result before its next action.

Key takeaways

- Keep looping while `stop_reason` is `tool_use`; stop when it is `end_turn`.
- Append each tool result to the conversation so Claude can reason about the next step.

- Do not stop on text content, parsed phrases or an iteration cap as the main signal.
- In an agentic loop, Claude picks the next tool; a decision tree fixes it in advance.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Stop reasons and fallback - Claude Platform Docs (opens in a new tab) · Handle tool calls - Claude Platform Docs (opens in a new tab) · Context windows - Claude Platform Docs (opens in a new tab) · Manage tool context - Claude Platform Docs (opens in a new tab) · Thinking in tool and multi-turn workflows - Claude Platform Docs (opens in a new tab)

Orchestrate multi-agent systems with coordinator-subagent patterns

Start here

THINK FIRST

What does the coordinator own in a hub-and-spoke multi-agent system, and what must it give each subagent?

Why it matters: Questions describe a research system with a coordinator and subagents, and ask how it should delegate, route and refine work.

One coordinator sits at the hub: it splits the task, picks subagents, routes every message and combines the results. Subagents start with isolated context, so everything they need must arrive in their prompt.

Key terms

Term	Meaning
Hub-and-spoke	A design where the coordinator manages all communication, error handling and routing between subagents.
Coordinator	The agent that decomposes the task, delegates, aggregates results and chooses which subagents to run.
Fork	A subagent whose own conversation does not start fresh. Any other subagent starts with a fresh conversation of its own.
Isolated context	A subagent that is not a fork gets no parent history or tool results, only its system prompt and the task prompt.
Iterative refinement loop	The coordinator checks synthesis for gaps, re-delegates targeted queries, then reruns synthesis.

The coordinator as hub

All subagent communication flows through the coordinator. Subagents report back to the conversation that spawned them rather than messaging each other. Routing everything through the coordinator gives you observability, consistent error handling and controlled information flow.

EXAM TRAP

Letting workers message each other directly is the agent team model, not hub-and-spoke. It gives up the coordinator's control of information flow.

Subagents start with isolated context

A subagent does not inherit the coordinator's history. Unless it is a fork, a subagent receives only its own system prompt and the prompt string it is given. Put the file paths, errors, decisions and prior findings it needs into that prompt. Only its final message comes back to the coordinator.

EXAM TRAP

A synthesis subagent does not inherit earlier search results from its parent. Pass the complete findings of prior agents directly in its prompt.

Choosing which subagents to run

Select subagents by what the query needs. A coordinator should analyze each query and invoke only the subagents it requires, not always the full pipeline. Its prompts should state research goals and quality criteria rather than step-by-step procedures, so subagents can adapt.

Partitioning the research scope

Give each subagent a distinct slice of the work. Assign distinct subtopics or source types to each subagent to minimize duplication. Independent subagents can then run in parallel, launched as several Task calls in one response.

EXAM TRAP

A very narrow split is not automatically thorough. Overly narrow decomposition leaves parts of a broad research topic uncovered.

Refining until coverage is sufficient

Check synthesis for gaps, then re-delegate. The coordinator reviews the synthesis output for gaps. It sends targeted queries back to the search and analysis subagents, then re-invokes synthesis, until coverage is sufficient. Synthesis can also annotate which findings are well supported and where sources were missing.

Handling subagent failures

Subagents send structured errors up to the coordinator. A subagent should recover from transient failures locally and escalate only what it cannot fix, with partial results and what it tried. A generic status like "search unavailable" hides that context. After a timeout, the coordinator can continue with partial results and note the gaps.

Key takeaways

- The coordinator decomposes the task, delegates, aggregates results and decides which subagents to invoke.
- Route all subagent communication through the coordinator for observability and consistent error handling.
- Partition scope by subtopic or source type, but not so narrowly that coverage suffers.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Subagents in the SDK - Claude Code Docs (opens in a new tab) · Run agents in parallel - Claude Code Docs (opens in a new tab)

Configure subagent invocation, context passing, and spawning

Start here

THINK FIRST

What does a subagent know when it starts, and how do you make sure it gets what it needs?

Why it matters: Questions show a coordinator whose subagents fail, run slowly or lose context, and ask which setting or prompt change fixes it.

A subagent starts with a fresh context and knows only what its prompt tells it. The coordinator must be allowed to spawn it, must pass it every finding it needs, and should launch independent subagents together.

Key terms

Term	Meaning
Task tool	The tool a coordinator uses to spawn subagents; it shows as Agent in tool_use blocks.
AgentDefinition	A subagent's setup: its description, its system prompt and its allowed tools.
Fork	A subagent that begins with a copy of the current conversation, not a blank context.
Coordinator	The hub agent that routes all information and errors between subagents.

Let the coordinator spawn

The coordinator's `allowedTools` must include `"Task"`. Without it, the coordinator cannot invoke any subagent. In the stream, the same tool shows as `Agent` in `tool_use` blocks and as `Task` in the `system:init` tools list.

Subagents start fresh

A subagent does not inherit the parent's conversation. Unless it is a fork, its context window starts without the parent conversation. Subagents also share no memory between invocations. The prompt string

is the only thing the parent passes in.

> **EXAM TRAP**
> Do not assume a subagent already knows what the coordinator learned. Put the findings, file paths and decisions it needs directly in its prompt.

Pass complete findings

Give the synthesis subagent the full results of earlier agents. Include web search results and document analysis outputs in its prompt. Structure each finding so content stays separate from metadata, keeping attribution intact.

Fields of a structured finding passed between agents

Field	Kind	Why it matters
Claim	Content	The point the finding makes
Evidence excerpt	Content	Shows what supports the claim
Source URL or document name	Metadata	Keeps the claim's attribution through synthesis
Publication date	Metadata	Travels with the claim into synthesis

Spawn in parallel

Emit several Task calls in one coordinator response. Independent subagents then run at the same time. The work finishes in the time of the slowest subagent, not the sum of all of them.

> **EXAM TRAP**
> Spawning one subagent per turn makes the work sequential. For parallel runs, put all the Task calls in a single response.

Define each subagent type

`description` **and** `prompt` **are required in** `AgentDefinition`**.** The description tells Claude when to use the agent, so keep it clear and specific. The prompt is its system prompt. Restrict `tools` to the agent's role; if you omit the field, it inherits every available tool.

Guide with goals, not steps

Tell subagents the research goal and the quality bar. Step-by-step procedures leave a subagent no room to adapt to what it finds. Goals and quality criteria let it choose its own path.

Fork to compare approaches

`fork_session` **branches from a shared analysis baseline.** Each branch explores a different direction from the same starting point. For example, compare two testing strategies or two refactoring approaches after one codebase analysis.

Key takeaways

- A coordinator needs `"Task"` in `allowedTools` to spawn subagents.
- Subagents inherit no parent context, so pass every needed finding in the prompt.
- Emit multiple Task calls in one response to run subagents in parallel.
- Use `fork_session` to explore divergent approaches from one shared baseline.

Sources: Subagents in the SDK - Claude Code Docs (opens in a new tab) · Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Parallel tool use - Claude Platform Docs (opens in a new tab)

Implement multi-step workflows with enforcement and handoff patterns

Start here

THINK FIRST

When is a prompt instruction enough to keep an agent's steps in order, and when must code enforce the order?

Why it matters: Questions describe an agent that skips or misorders a step, or hands off badly, and ask for the most reliable fix.

Key terms

Term	Meaning
Programmatic prerequisite	A code gate that blocks a tool call until an earlier step has returned what it needs.
Hook	Code that intercepts tool calls or results to enforce a rule or change the data.
Structured handoff	An escalation summary with customer details, root cause and recommended actions.
Escalation trigger	A condition that moves the case to a human, such as a policy gap.

Enforcement versus guidance

Code gives a deterministic guarantee; prompts stay probabilistic. A firmer system prompt or extra few-shot examples still rely on the model choosing to comply. When an error has financial consequences, that is not enough.

Ways to keep a required tool order

Approach	What it does	Guarantees the order?
Programmatic prerequisite	Blocks later tools until verification returns an ID	Yes, deterministically

Approach	What it does	Guarantees the order?
Stronger system prompt	Tells the agent verification is mandatory	No, compliance is probabilistic
Few-shot examples	Shows the agent verifying first	No, compliance is probabilistic
Routing classifier	Limits which tools each request can use	No, it controls availability, not order

EXAM TRAP

Limiting which tools a request can see does not fix a skipped step. It changes tool availability, while the problem is tool ordering.

Gate the risky tools

Block refund tools until `get_customer` returns a verified ID. In one support scenario, the agent skipped verification in **12%** of cases and looked up orders by name alone. That occasionally led to misidentified accounts and incorrect refunds. A programmatic prerequisite gives a deterministic guarantee that prompt-based fixes cannot.

Hooks enforce business rules

A hook can stop a tool call that breaks a rule. For example, it can block an operation above a set amount and route it to an escalation workflow. A `PostToolUse` hook works on the other side, reshaping tool results before the model reads them.

Order inside one turn

For dependent calls, a sequential run that stops at the first failure is valid. Execution order is your choice. Independent read-only calls are usually safe to run in parallel. If you stop early, return `is_error: true` for any call you did not run. If you run in parallel and a call fails because its prerequisite was not done yet, return the error and Claude reissues it next turn.

Split multi-concern requests

Break the request into separate items, then work on each. The agent handles each concern, and then it synthesizes the results into one unified response.

Know when to escalate

Escalate when asked, at a policy gap, or when progress stalls. A customer asking for a human is a clear trigger. So is a policy exception or gap, not merely a complex case. An agent that cannot make meaningful progress should also hand off.

EXAM TRAP

Complexity alone is not an escalation trigger. Policy exceptions or gaps are, along with a request for a human and a lack of progress.

Hand off a complete summary

The human agent cannot see the conversation transcript. So the handoff must stand on its own. Include the customer ID, the root cause, the refund amount and the recommended action.

EXAM TRAP

Do not assume the human can read the chat. Human agents receiving the escalation lack access to the conversation transcript, so compile a structured summary for them.

Key takeaways

- Use code, not prompts, when a step order must hold every time, such as verifying identity before refunds.
- Stronger prompts and few-shot examples still depend on probabilistic compliance by the model.
- A routing classifier controls which tools are available, not the order they run in.
- An escalation summary carries customer ID, root cause, refund amount and recommended action.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Parallel tool use - Claude Platform Docs (opens in a new tab)

Apply Agent SDK hooks for tool call interception and data normalization

Start here

THINK FIRST

When should a business rule live in an Agent SDK hook instead of in the system prompt?

Why it matters: Questions describe an agent that breaks a rule or gets messy tool data, and ask which hook pattern fixes it.

Hooks are your code, run when the SDK fires an event such as a tool call about to start or a result coming back. Code gives deterministic guarantees, while prompt instructions only make compliance likely.

Key terms

Term	Meaning
PreToolUse	The event fired when a tool is about to be called; a hook here can allow, deny or change the call.
PostToolUse	The event fired after a tool returns a result; a hook here can add to or replace that result.
permissionDecision	The field a PreToolUse hook sets to allow, deny, ask or defer the tool call.
Matcher	The pattern that picks which tools a hook fires for; it matches tool names only.

Hooks versus prompts

Hooks give deterministic guarantees; prompts do not. Stronger system prompt wording and few-shot examples still depend on the model choosing to comply. When a mistake costs money, such as a refund before identity checks, that non-zero failure rate is too high. Hooks and prerequisite gates enforce the rule in code instead.

EXAM TRAP

Few-shot examples that always call the verification tool first do not guarantee the order. They still rely on probabilistic compliance, so pick a hook.

Blocking a policy violation

A PreToolUse hook can stop a call and redirect it. The hook checks the call against the rule, such as a refund above a threshold amount. If the call breaks it, set `permissionDecision` to `deny` and route the case to a human escalation workflow. `permissionDecisionReason` tells the model why, and `systemMessage` tells the user. Hooks can also demand human approval for sensitive actions like database writes.

Normalizing tool results

A PostToolUse hook can fix data before the model reads it. Different MCP tools may return Unix timestamps, ISO **8601** dates or numeric status codes. A PostToolUse hook can convert them all into one format first, so the agent reasons over consistent data.

PostToolUse output fields compared

Field	What it does	Applies to
`additionalContext`	Appends information to the tool result	The tool result where the hook fired
`updatedToolOutput`	Replaces the output before Claude sees it	Any tool, in both SDKs
`updatedMCPToolOutput`	Replaces the output (the older field)	MCP tools only

EXAM TRAP

`additionalContext` does not change the tool's output; it only appends to it. To replace messy data, set `updatedToolOutput`.

Choosing which tools a hook sees

Matchers match tool names, never arguments. A regex such as `^mcp__` fires the hook for every MCP tool, because their names start with `mcp__`. To act only on certain file paths or values, read `tool_input` inside the hook and decide there.

EXAM TRAP

A matcher cannot filter on a file path or an amount. The hook must inspect `tool_input` itself.

When several hooks disagree

The most restrictive decision wins. Matching hooks run in parallel. Deny beats defer, defer beats ask, and ask beats allow, so one deny blocks the call. If a PreToolUse callback times out, Claude Code discards its output and blocks the call too.

EXAM TRAP

A PreToolUse hook that times out does not let the call through. Claude Code cancels the hook and blocks the tool call.

Async hooks and side effects

Async hook output cannot block or change anything. By the time an async hook returns, the agent has already moved on. Use async hooks only for logging, metrics or notifications, never for a compliance check.

EXAM TRAP

An async hook cannot enforce a refund limit, because its output cannot block the operation.

Key takeaways

- Use a hook, not prompt wording, when a business rule must hold every time.
- PreToolUse hooks intercept outgoing calls; PostToolUse hooks transform results before the model sees them.
- Set `permissionDecision` to `deny` to stop a call, and give the model a reason.
- One deny from any matching hook blocks the tool call.

Sources: Intercept and control agent behavior with hooks - Claude Code Docs (opens in a new tab) · Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Modifying system prompts - Claude Code Docs (opens in a new tab)

Design task decomposition strategies for complex workflows

Start here

THINK FIRST

When should a workflow follow a fixed chain of steps, and when should its plan grow from what each step finds?

Why it matters: Questions describe a review or investigation task and ask which decomposition pattern fits it best.

Predictable work splits into a fixed chain of steps you set in advance. Open-ended work needs a plan that

adds subtasks as each step reveals something new. Picking between the two is the core skill here.

Key terms

Term	Meaning
Prompt chaining	Breaking a task into sequential API calls, each a separate step you can inspect.
Dynamic decomposition	Creating subtasks from what each step discovers, instead of fixing them up front.
Integration pass	A separate review pass that follows data flow across files after the per-file passes.
Attention dilution	Lost focus when one review pass has to cover too many files at once.
Coordinator agent	An agent that reads a query and chooses which subagents to invoke for it.

Match the pattern to the task

Chain prompts for predictable reviews; decompose dynamically for investigations. A predictable review with known aspects can run as a chain of sequential API calls. An adaptive investigation plan instead generates subtasks from what each step discovers.

Prompt chaining and dynamic decomposition compared

Feature	Prompt chaining	Dynamic decomposition
Best fit	Predictable multi-aspect reviews	Open-ended investigation tasks
How steps are set	A fixed sequence of separate API calls	Subtasks generated from each step's findings
Main benefit	You can inspect outputs and enforce the pipeline	The plan adapts as new dependencies appear

EXAM TRAP

Dynamic decomposition is not always the better choice. For a predictable multi-aspect review, a fixed prompt chain is the expected answer.

Why chain prompts at all

Chaining helps when you must inspect or control each stage. With adaptive thinking and subagent orchestration, Claude handles most multistep reasoning internally. Explicit chaining still earns its place when you need to log, evaluate or branch between calls. The common shape is self-correction: draft, review against criteria, then refine.

Split big reviews into two layers

Review each file alone, then run a separate cross-file pass. Per-file passes catch local issues. The integration pass follows data flow between files. One workflow shape runs a reviewer per file, then hands

every finding to one agent that ranks and deduplicates them.

EXAM TRAP

Putting every file into one big review pass does not make it more thorough. It dilutes attention and can produce contradictory findings.

Let the plan grow with findings

Adaptive plans create subtasks from what each step uncovers. This suits investigation, where you cannot list the work in advance. A coordinator should also pick only the subagents a query needs, rather than always running the full pipeline.

EXAM TRAP

A coordinator that splits a broad research topic too narrowly does not gain focus. It leaves parts of the topic uncovered.

Open-ended tasks: map first

Map the structure, find high-impact areas, then plan by priority. Take adding comprehensive tests to a legacy codebase. Survey the code first, starting with broad questions and then narrowing. The Claude Code testing recipe likewise finds untested code before it generates test scaffolding. Let the plan change as you discover dependencies.

Keep a long plan alive

Plan mode proposes a plan and edits nothing until you approve. Claude writes that plan to a file. A long session compacts its context, but Claude Code puts the plan file back each time, so the plan is not lost. You can then switch to direct execution for the implementation.

Delegate where it helps

Use subagents for parallel, isolated or independent workstreams. Keep sequential steps that share context in the main thread. The Explore subagent can absorb verbose discovery, so the main context window does not run out during a multi-phase task.

EXAM TRAP

Subagents are not the default for every step. For sequential work that must keep context across steps, work directly instead of delegating.

Scripts for very large jobs

A workflow script fits when a task outgrows one agent's context. It also fits when the same step must run across many items. The script itself keeps the loop, branches and partial results, so Claude's context holds only the final answer. One shape searches in rounds and stops when a round finds nothing new.

Key takeaways

- Prompt chaining fits predictable multi-aspect reviews; dynamic decomposition fits open-ended investigations.
- Split a large review into per-file passes plus a separate cross-file integration pass.
- For open-ended work, map the structure and high-impact areas before building a plan that adapts.
- Explicit chaining still pays off when you must inspect intermediate outputs or enforce a pipeline.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Prompting best practices - Claude Platform Docs (opens in a new tab) · Orchestrate subagents at scale with dynamic workflows - Claude Code Docs (opens in a new tab) · Common workflows - Claude Code Docs (opens in a new tab) · Set up Claude Code in a monorepo or large codebase - Claude Code Docs (opens in a new tab)

Manage session state, resumption, and forking

Start here

THINK FIRST

When should you resume a past session, fork it, or start fresh with a summary?

Why it matters: Questions describe returning to earlier analysis and ask whether to resume, fork or start a new session.

Resuming keeps prior analysis, which helps only while it is still valid. Forking copies the history into an independent branch for a different approach. When old tool results are stale, a fresh session with a structured summary is more reliable.

Key terms

Term	Meaning
Named session	A session given a name with claude -n or /rename, so claude --resume can find it.
Fork	A separate session seeded with a copy of the original's history, under its own session ID.
Stale tool results	Earlier tool output that no longer matches the code, for example after files were modified.
Structured summary	A condensed record of results and decisions passed into a fresh session's prompt.

Naming and resuming sessions

Name a session, then resume it by that name. Name it at startup with `claude -n` or later with `/rename`. Resuming by name reopens it under the same session ID and appends to the conversation, so you can continue a named investigation across work sessions.

Example: Starts a named session, then later resumes that same session by its name.

```
claude -n my_session
claude --resume my_session
```

EXAM TRAP

An unnamed session's default display name is not a resume handle. Name the session if you plan to resume it.

Telling a resumed session what changed

Name the files you modified since the last analysis. Resuming lets the agent reuse its analysis without rereading files. If code changed in the meantime, tell it which previously analyzed files changed. That allows targeted re-analysis instead of a full re-exploration.

EXAM TRAP

A branch switch changes the files, not the history. Claude sees the new branch's files, but the conversation history stays the same. When you resume after code changes, tell the agent which analyzed files changed.

Resume or start fresh

Resume when prior context is mostly valid; start fresh when it is stale. Stale tool results mislead a resumed session, so a fresh start from a structured summary is the safer choice. Pass the analysis output, decisions and file diffs you need into the fresh session's prompt.

Resuming and starting fresh compared

Feature	Resume	Fresh session with summary
Fits when	Prior context is mostly valid	Prior tool results are stale
Starts with	The whole earlier conversation	A fresh context plus the summary you inject

Forking to explore alternatives

Fork to branch from a shared analysis baseline. A fork starts with a copy of the history and gets its own session ID, while the original stays unchanged. Use forks to compare two testing strategies or refactoring approaches from one codebase analysis. You can also fork with `/branch`.

Example: Resumes a named session as a new, independent fork with its own session ID.

```
claude --resume my_session --fork-session
```

EXAM TRAP

Forking does not isolate files. Edits by a forked agent are real and visible to every session in the same directory.

Two terminals, one session

Fork before you work in parallel. Resuming the same session in two terminals without forking interleaves both into one transcript. Forks give you two independent sessions that resume separately. Remember that a session keeps the conversation, not the filesystem.

Key takeaways

- Resume a named session with `claude --resume` when the earlier context is still mostly valid.
- Tell a resumed session which files changed so it re-analyzes only those.
- Start fresh with a structured summary when prior tool results are stale.
- Fork to compare approaches from one baseline; each fork has its own session ID.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Work with sessions - Claude Code Docs (opens in a new tab) · Manage sessions - Claude Code Docs (opens in a new tab) · How Claude Code works - Claude Code Docs (opens in a new tab)

Domain 2 | 18% of the exam | 18 min read

Tool Design & MCP Integration

Overview

This domain is about 18% of the exam, roughly 11 of its 60 questions.

Every point below comes from Anthropic's official documentation; each section lists its sources.

Design effective tool interfaces with clear descriptions and boundaries

Start here

THINK FIRST

When an agent keeps calling the wrong one of two similar tools, what should you change first, and why?

Why it matters: Questions describe an agent that misroutes requests and ask which change to its tools or prompt fixes it first.

Claude picks a tool by matching the user's request against each tool's name and description. Vague or overlapping descriptions cause misrouting. The recommended first fix is to expand each tool's description.

Key terms

Term	Meaning
Tool description	Text that says what a tool does; Claude reads it to decide when to call the tool.
Misrouting	Calling the wrong tool because two tools have vague or nearly identical descriptions.
MCP server	A server you connect to Claude that offers tools, each with its own tool definition. Claude calls them when a request maps to a tool's capability.
Keyword-sensitive instruction	A system prompt line whose wording links certain words to a tool, creating tool associations you did not intend.
Purpose-specific tool	A narrow tool with a defined input and output contract, split out of a generic tool.

How Claude chooses a tool

Tool descriptions are the main signal for tool selection. With the default `tool_choice` of `auto`, Claude calls a tool when the request matches its described capability and the answer is not already in context. With an MCP server attached, a general question about that service gets a direct answer, not a tool call.

EXAM TRAP

A clear tool name is not enough on its own. When descriptions are minimal, the model lacks the context to tell similar tools apart.

Parts of a tool definition

Name, title and description serve different readers. An MCP tool also carries an `inputSchema` for its parameters. If a tool declares an output schema, servers must return structured results that conform to it, and clients should validate them.

Three fields of an MCP tool definition compared

Field	What it holds	Main reader
`name`	Unique identifier of the tool	Programs and logic; display fallback when no title exists
`title`	Optional human-readable name	People using the interface
`description`	What the tool does, in plain language	Claude, when deciding whether to call it

What a strong description says

Cover inputs, example queries, edge cases and boundaries. Say what the tool is for, what it accepts and returns, and when to use it over a similar tool. Describe parameters too: the `get_weather` example asks for a city and state. A stateful server should put its retention policy in the creation tool's description.

Why overlap causes misrouting

Near-identical descriptions send requests to the wrong tool. In the guide's scenario, an agent calls `get_customer` for order questions instead of `lookup_order`. Both had one-line descriptions and took similar identifiers. The same risk applies to `analyze_content` and `analyze_document`.

EXAM TRAP

Few-shot examples, a keyword routing layer or merging the tools are not the first fix. Examples cost tokens without curing the cause, routing bypasses the model's language understanding, and merging is more work than a first step.

Rename, split or constrain tools

You can also rename, split or constrain the tools themselves. Renaming `analyze_content` to `extract_web_results`, with a web-specific description, removes its functional overlap. Splitting breaks

one generic tool into separate tools. Constraining swaps a generic tool for a narrower alternative.

Renaming, splitting and constraining a tool compared

Change	Before	After
Rename	`analyze_content`	`extract_web_results`, with a web-specific description
Split	A generic `analyze_document`	`extract_data_points`, `summarize_content`, `verify_claim_against_source`
Constrain	A generic `fetch_url`	`load_document`, which validates document URLs

Keep the tool set small

More tools make selection less reliable. The guide contrasts **18** tools with **4-5**: each extra tool adds decision complexity. For dozens of tools across several MCP servers, set `defer_loading` to true with the Tool search tool. Only relevant tools then reach each query; the default is false.

System prompt wording

Prompt wording can override well-written tool descriptions. Keyword-sensitive instructions can tie words to the wrong tool, so review the system prompt as well as the descriptions. Wording also shifts how readily Claude calls tools at all.

How system prompt wording shifts tool use

Wording	Example start	Effect on tool use
Light nudge	"Use the tools to investigate…"	Increases tool use
Strong rule	"Always call a tool first…"	Pushes tool use further
Open choice	"Use your judgment…"	Keeps tool calls conservative

EXAM TRAP

Rewriting descriptions will not help if a system prompt line keeps steering a keyword to one tool. Check the prompt for such lines too.

Key takeaways

- Descriptions are the main signal Claude uses to choose between similar tools.
- Fix misrouting first by expanding descriptions with input formats, examples, edge cases and boundaries.
- Rename or split overlapping tools so each has one purpose and clear inputs and outputs.
- Review the system prompt for keyword-sensitive lines that pull requests toward the wrong tool.

Sources: Tool use with Claude - Claude Platform Docs (opens in a new tab) · MCP connector - Claude Platform Docs (opens in a new tab) · Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July

2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Give Claude custom tools - Claude Code Docs (opens in a new tab) · Tools - Model Context Protocol (opens in a new tab) · Schema Reference - Model Context Protocol (opens in a new tab)

Implement structured error responses for MCP tools

Start here

THINK FIRST

What should an MCP tool return when it fails, so the agent can choose the right recovery?

Why it matters: Questions describe a tool failure that the agent mishandles and ask which error response or recovery pattern fixes it.

A failed tool call should say so with `isError` and explain why in structured fields. A category and a retryable flag let the agent retry transient failures, stop on business rule violations, and explain clearly to users.

Key terms

Term	Meaning
isError	The tool result flag set to true to signal a failure, so Claude can react to it.
errorCategory	A structured field that names the error type: transient, validation or permission.
isRetryable	A boolean that marks an error as retryable or not, which prevents wasted retry attempts.
Valid empty result	A successful query that found no matches, unlike an access failure.

Signal the failure with isError

Set `isError` to `true` so Claude knows the call failed. In the Messages API, the matching field on a `tool_result` block is `is_error`. Given an error result, Claude can retry, try another tool or explain the failure. A handler error does not stop the agent loop.

Kinds of tool error

Transient, validation, business and permission errors differ. Each type calls for a different move from the agent, so the tool should say which one occurred rather than just that something failed.

How three error types are handled

Error type	Example	Agent's next move
Transient	A timeout or the service is unavailable	Retry the call
Validation	Invalid or missing input	Retry with corrected input

Error type	Example	Agent's next move
Business	A policy violation	Do not retry; explain it to the customer

Business rule violations

Return `retriable: false` and a customer-friendly explanation. A policy violation will not succeed on a retry. The friendly explanation gives the agent words it can pass on, so it explains the business error to the customer appropriately.

Empty results are not errors

Separate access failures from valid empty results. A timeout needs a retry decision, while a query that ran and found no matches succeeded. Hiding errors by returning empty results as success is an anti-pattern. So is ending a whole workflow on one failure.

> **EXAM TRAP**
> An empty result caused by a timeout is not "no matches". Report it as an access failure so the coordinator can decide what to do.

Recover locally, escalate the rest

Subagents handle transient failures themselves. A subagent should recover from transient failures locally. It passes up only errors it cannot resolve, with what it attempted and any partial results. A generic status like "search unavailable" hides that context from the coordinator.

> **EXAM TRAP**
> Escalating every transient timeout to the coordinator is not the pattern. Recover locally first and propagate only unresolved errors.

Key takeaways

- Signal tool failures with `isError: true`, and add structured details.
- Include `errorCategory`, `isRetryable` and a human-readable description in error responses.
- Business rule violations get `retriable: false` and a customer-friendly explanation.
- A valid empty result is a success; an access failure needs a retry decision.

Sources: Give Claude custom tools - Claude Code Docs (opens in a new tab) · Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Handle tool calls - Claude Platform Docs (opens in a new tab)

Distribute tools appropriately across agents and configure tool choice

Start here

THINK FIRST

How do you decide which tools each agent gets, and when should you force a tool call?

Why it matters: Questions describe an agent that picks the wrong tool or skips one, and ask how to scope its tools or set tool_choice.

Each agent should hold only the few tools its role needs, because more tools mean harder choices and more misuse. The `tool_choice` setting then controls whether the model may call a tool, must call one, or must call a named one.

Key terms

Term	Meaning
Scoped tool access	Giving an agent only the tools its role needs, plus a few cross-role tools for frequent needs.
tool_choice	The request setting that controls whether Claude must call a tool, and which one.
Forced tool selection	A tool_choice of type tool with a name, which makes Claude call that exact tool.
Constrained tool	A narrow stand-in for a generic tool, such as load_document that validates URLs in place of fetch_url.

Why fewer tools work better

Too many tools hurt tool selection. The exam guide's example is **18** tools instead of **4** to **5**: every extra tool adds decision complexity. Anthropic notes that selection degrades past **30** to **50** tools. Merging related operations into fewer tools with an action parameter reduces ambiguity.

EXAM TRAP

Adding more specialized tools to one agent does not make it more reliable. More choices lower its tool selection reliability.

Scoping tools by role

Give each subagent only the tools its role needs. A synthesis agent that holds web search tends to misuse it. Restrict each subagent's tools through its `AgentDefinition`. For a frequent need, add one scoped cross-role tool, such as `verify_fact`, and route complex cases through the coordinator.

Constraining generic tools

Swap a generic tool for a narrow, validating one. For example, replace a generic `fetch_url` tool with `load_document`, a constrained alternative that validates document URLs. Anthropic gives similar advice for computer use, the most general option: prefer narrower tools when they cover the use case.

The tool_choice options

`auto` **may call a tool;** `any` **must;** `tool` **must call one named tool.** With `auto`, the default when tools are provided, the model may answer in text instead. `none` blocks tools and is the default when no tools are given. Set `any` to guarantee a tool call rather than conversational text.

The four tool_choice settings compared

Setting	Must Claude call a tool?	Which tool
`auto`	No, it may reply with text	Any provided tool, if it chooses one
`any`	Yes	Any one of the provided tools
`tool`	Yes	The single tool named in the setting
`none`	No, tools are blocked	None

EXAM TRAP

`any` does not pick a specific tool. To make sure `extract_metadata` runs, force it by name.

Forcing a first step

Force one tool first, then continue in follow-up turns. Forcing `extract_metadata` makes sure that extraction runs before any enrichment tools. Later steps then run in follow-up turns. A forced response starts with the `tool_use` block, with no text and no thinking block before it.

Works: A tool_choice value that forces Claude to call the extract_metadata tool.

```
{"type": "tool", "name": "extract_metadata"}
```

- `"type": "tool"`: forces one particular tool

Fails: `any` *requires some tool call, but it does not make* `extract_metadata` *the one Claude calls.*

```
{"type": "any"}
```

Model support and caching

Check the model before forcing tools. On Claude Opus **5.5**, Sonnet **5.5**, Fable **5.1** and Mythos **5.1**, `any` and `tool` return a **400** error. Changing `tool_choice` between requests can also cause prompt cache misses, so keep it stable.

EXAM TRAP

Do not assume forced tool choice works on every model. On those four models, a request with `any` or `tool` fails with a **400** error instead of forcing a tool call.

Key takeaways

- Give each agent a small, role-specific tool set; extra tools reduce selection reliability.
- Use `any` to guarantee a tool call, and `tool` with a name to guarantee a specific one.
- `auto` is the default when tools are provided, and the model may then reply with text.
- Add a scoped cross-role tool like `verify_fact` for frequent needs, and route complex cases via the coordinator.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Define tools - Claude Platform Docs (opens in a new tab) · Tool use with Claude - Claude Platform Docs (opens in a new tab) · Tool search tool - Claude Platform Docs (opens in a new tab) · Tool combinations - Claude Platform Docs (opens in a new tab) · Troubleshooting tool use - Claude Platform Docs (opens in a new tab)

Integrate MCP servers into Claude Code and agent workflows

Start here

THINK FIRST

Where should a team's MCP servers be configured, and how do you make sure the agent actually uses them well?

Why it matters: Questions describe a team adding MCP servers and ask where to configure them, how to handle secrets, or why the agent ignores a tool.

Shared servers live with the project and personal ones with the user. Every configured server's tools are available together once connected. Clear tool descriptions and resources then steer the agent to the right tool and data.

Key terms

Term	Meaning
.mcp.json	Project-scoped MCP configuration for servers the whole team shares.
~/.claude.json	User-scoped MCP configuration for personal or experimental servers.
MCP resource	Context data a server shares, such as files or schemas, each identified by a URI.
Environment variable expansion	Writing `${GITHUB_TOKEN}` in config so the real secret is never committed.

Project scope or user scope

Shared servers go in `.mcp.json`; personal ones in `~/.claude.json`. Put team tooling at the project level so everyone gets it. Keep experiments at the user level so they stay yours.

The two MCP configuration scopes compared

Feature	Project scope	User scope
File	`.mcp.json`	`~/.claude.json`
Used for	Shared team tooling	Personal or experimental servers

EXAM TRAP

A personal experimental server does not belong in the project's `.mcp.json`. Configure it in user-scoped `~/.claude.json` instead.

Keep secrets out of the repo

Reference tokens as `${GITHUB_TOKEN}` in `.mcp.json`. The config file can be shared while each person supplies the token through the environment. No secret is committed.

All servers at once

Tools from every configured server load when the agent connects. They are all available to the agent at the same time. A project server and a personal server can both be in use in one session.

Describe tools in detail

Rich MCP tool descriptions stop the agent from defaulting to Grep. Claude reads a tool's description to decide when to call it. Explain what the tool can do and what it returns, so the agent picks the more capable MCP tool.

EXAM TRAP

When the agent keeps choosing Grep over a more capable MCP tool, look at that tool's description first. Spell out its capabilities and outputs in detail.

Expose catalogs as resources

Resources show the agent what data exists without exploring. Issue summaries, documentation trees and database schemas make good resources. The agent sees what is available up front, so it makes fewer exploratory tool calls.

Build or reuse a server

Use a community server for standard integrations like Jira. Save custom servers for workflows that are specific to your team. For a standard integration, the existing community server is the better choice.

Key takeaways

- Configure shared team servers in project-scoped `.mcp.json` and personal ones in `~/.claude.json`.
- Use `${VAR}` expansion in `.mcp.json` so credentials are never committed.
- Describe MCP tools in detail so the agent picks them over built-ins such as Grep.
- Prefer community MCP servers for standard integrations; build custom ones for team-specific workflows.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Resources - Model Context Protocol (opens in a new tab) · Tools - Model Context Protocol (opens in a new tab) · Give Claude custom tools - Claude Code Docs (opens in a new tab) · MCP connector - Claude Platform Docs (opens in a new tab)

Select and apply built-in tools (Read, Write, Edit, Bash, Grep, Glob) effectively

Start here

THINK FIRST

Given a task in an unfamiliar codebase, how do you decide between Grep, Glob, Read, Write, Edit and Bash?

Why it matters: Questions describe a coding task and ask which built-in tool, or which order of tools, fits it best.

Each built-in tool has one job: Grep searches file contents, Glob matches file paths, Read and Write handle whole files, and Edit makes targeted changes. Edit needs a unique text match, so Read plus Write is its fallback. Explore step by step instead of reading everything upfront.

Key terms

Term	Meaning
Grep	Built-in tool that searches file contents for patterns such as function names, error messages or imports.
Glob	Built-in tool that finds files whose paths match a name or extension pattern.
Edit	Built-in tool for targeted changes; it locates the text to change by a unique match.
Incremental exploration	Learning a codebase step by step: Grep finds entry points, then Read follows imports and flows.

Search contents or match paths

Grep searches inside files; Glob matches file paths. Ask what you are looking for. Text inside code, such as a function name or an error message, calls for Grep. A file name or extension pattern calls for Glob.

Matching a search task to Grep or Glob

Task	Tool	What it searches
Find all callers of a function	Grep	File contents
Locate where an error message appears	Grep	File contents

Task	Tool	What it searches
Find every file matching `**/*.test.tsx`	Glob	File paths
Find files by name or extension	Glob	File paths

EXAM TRAP

Glob never looks inside files. If the task is to find where a function name appears in code, the answer is Grep, not Glob.

Whole files or targeted changes

Read and Write handle full files; Edit changes one part. Choose Edit for a small, targeted change that can anchor on text appearing once in the file. Choose Read to load a full file and Write to replace its full contents.

When Edit cannot find a unique match

Fall back to Read plus Write. Edit depends on a unique text match. If the text appears more than once, the edit fails. Read the full file, then Write the corrected version. The API text editor tool works the same way: a `str_replace` matching several places should return an error with `is_error: true`.

EXAM TRAP

Retrying Edit with the same non-unique text keeps failing. Give it more surrounding context to make a unique match, or switch to Read then Write.

The API text editor tool

Its commands are view, create and str_replace. `view` reads a whole file, a range of lines, or a directory listing. In `view_range`, line numbers start at **1**, and **-1** as the end line reads to the end. `create` writes a new file from `file_text`. `str_replace` needs `old_str` to match exactly, including whitespace and indentation.

Explore incrementally

Start with Grep for entry points, then Read to trace flows. Avoid reading every file upfront, because each file read adds to the context window. From an entry point, follow imports with Read. Large research reads can go to a subagent, which keeps them in its own context.

EXAM TRAP

Reading all files before starting is the wrong choice. It fills context with files that may not matter instead of following the flow from an entry point.

Trace usage across wrapper modules

List the exported names first, then search for each one. Identify every name the wrapper module exports, then Grep for each name across the codebase. In a large codebase, code intelligence plugins can also jump to definitions and find references through a language server.

Built-in search versus Bash search

Built-in Grep and Glob try to hide denied paths; Bash does not. Claude Code makes a best-effort attempt to omit Read-denied paths from built-in Grep and Glob results. A Bash `grep -r` or `find` still lists those files. Content searches respect `.gitignore` by default, so folders such as `node_modules/` stay out.

> **EXAM TRAP**
>
> Denying Read on a path does not hide it from a Bash `grep -r` or `find` over that directory.

The API bash tool

One shell stays alive across tool calls. The working directory, environment variables and created files carry over to the next command. Sending `restart: true` starts a clean session. Interactive commands such as vim or less cannot run. Validate commands with an allowlist, since a blocklist misses commands it did not foresee.

Key takeaways

- Grep finds text inside files; Glob finds files by path pattern.
- Edit needs a unique text match; when it fails on a non-unique match, use Read then Write.
- Explore with Grep first, then Read along the imports, rather than reading every file upfront.
- To trace a function through wrapper modules, list every exported name and search for each one.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Text editor tool - Claude Platform Docs (opens in a new tab) · Explore the context window - Claude Code Docs (opens in a new tab) · Set up Claude Code in a monorepo or large codebase - Claude Code Docs (opens in a new tab) · Bash tool - Claude Platform Docs (opens in a new tab)

Domain 3 | 20% of the exam | 20 min read

Claude Code Configuration & Workflows

Overview

This domain is about 20% of the exam, roughly 12 of its 60 questions.

Every point below comes from Anthropic's official documentation; each section lists its sources.

Configure CLAUDE.md files with appropriate hierarchy, scoping, and modular organization

Start here

THINK FIRST

When a teammate's Claude sessions ignore a team convention, where should that instruction live, and how do you check what loaded?

Why it matters: Questions describe Claude ignoring an instruction and ask which CLAUDE.md level, import or rule file fixes it.

CLAUDE.md files stack: user, project and subdirectory files all load and add together. Only project files committed to git reach teammates. Imports and `.claude/rules/` keep instructions modular, and `/memory` shows which memory files are in play.

Key terms

Term	Meaning
User-level CLAUDE.md	`~/.claude/CLAUDE.md`: personal preferences for all your projects, seen only by you.
Project CLAUDE.md	`./CLAUDE.md` or `./.claude/CLAUDE.md`: team instructions shared through version control.
@import	The `@path/to/import` syntax, which loads another file into context alongside the CLAUDE.md that names it.
Rule file	A markdown file in `.claude/rules/` covering one topic, optionally scoped to matching file paths.

The configuration hierarchy

User, project and directory CLAUDE.md files all add together. Discovered files are concatenated rather than overriding each other. Content runs from the filesystem root down to the working directory, so files nearer the launch directory are read last.

The three CLAUDE.md levels compared

Level	Location	Who gets it, and when
User	`~/.claude/CLAUDE.md`	Only you, across all your projects
Project	`./CLAUDE.md` or `./.claude/CLAUDE.md`	The whole team, through version control
Directory	A CLAUDE.md in a subdirectory	Loads once Claude reads, writes or edits a file there

Why a new teammate misses instructions

User-level instructions never travel through version control. A convention kept in your `~/.claude/CLAUDE.md` applies only to you. Move it into the project CLAUDE.md and commit it to git. Then the instruction applies consistently to every team member.

EXAM TRAP

Asking each person to copy the line into their own user-level file is not the fix. Team standards belong in the project CLAUDE.md, committed to git.

Conflicts between levels

No hard precedence rule decides between levels. If two levels disagree, the result depends on Claude's reading of them. Write rules that do not conflict, or state the precedence explicitly in the more specific file.

EXAM TRAP

A project CLAUDE.md does not silently replace a user CLAUDE.md. Both load, and Claude sees both.

Modular CLAUDE.md with @import

Import standards files instead of pasting them in. Use `@path/to/import` to include, for example, the standards each package needs. Relative paths resolve from the importing file, not the working directory, and nesting stops at **4** hops. Wrap a path in backticks to mention it without importing it. External imports first need approval in a dialog.

EXAM TRAP

Imports organize a long CLAUDE.md but do not lower its context cost. Imported files load together with the file that imports them.

Topic files in .claude/rules/

Split a large CLAUDE.md into one file per topic. Give each file a descriptive name, such as `testing.md` or `api-design.md`. Every `.md` file there is found recursively, so subfolders work. A rule without `paths` frontmatter loads at launch, with priority equal to `.claude/CLAUDE.md`. A rule with `paths` applies only to matching files.

Diagnose with /memory and /context

Check which memory files Claude actually has. `/memory` lists CLAUDE.md, CLAUDE.local.md and other memory files across user and project scopes, and opens them for editing. `/context` lists the memory files loaded at launch; one missing there is invisible to Claude. Each session starts fresh, so lasting instructions belong in CLAUDE.md.

CLAUDE.md in the Agent SDK

`settingSources` **decides which files the SDK loads.** With `"project"`, CLAUDE.md and `.claude/rules/*.md` load at session start. With `"user"`, `~/.claude/rules/*.md` loads too. Leaving `settingSources` out equals user, project and local together, while an empty list limits the agent to programmatic settings.

Key takeaways

- User-level `~/.claude/CLAUDE.md` reaches only you; team standards go in the committed project CLAUDE.md.
- All CLAUDE.md levels add together, with no hard precedence rule between them.
- Use `@import` to include each package's relevant standards files and keep CLAUDE.md modular.
- Split a large CLAUDE.md into one-topic files in `.claude/rules/`.
- Use `/memory` to see which memory files are in play when behavior differs between sessions.

Sources: Use Claude Code features in the SDK - Claude Code Docs (opens in a new tab) · How Claude remembers your project - Claude Code Docs (opens in a new tab) · Explore the .claude directory - Claude Code Docs (opens in a new tab) · Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · How Claude Code works - Claude Code Docs (opens in a new tab)

Create and configure custom slash commands and skills

Start here

THINK FIRST

Where should a reusable workflow live, and how do you keep its output out of your main session?

Why it matters: Questions describe a workflow a developer or team wants to reuse and ask where to store it and how to configure it.

Commands and skills live either in the project, shared through version control, or in your home directory, for

you alone. Skills add frontmatter that controls isolation, tool access and arguments, and they load only when needed, unlike CLAUDE.md.

Key terms

Term	Meaning
Project scope	Files under .claude/ in the repository, shared with every developer through version control.
User scope	Files under ~/.claude/ in your home directory, personal and not shared through version control.
SKILL.md	The file in each skill directory that holds YAML frontmatter and Markdown instructions.
context: fork	Frontmatter that runs a skill in an isolated sub-agent context, keeping its output out of the main conversation.

Project versus personal scope

Team commands go in the repo, personal ones in your home directory. Commands in `.claude/commands/` are version-controlled and reach everyone who clones or pulls the repo. Commands in `~/.claude/commands/` stay personal. Skills follow the same split between `.claude/skills/` and `~/.claude/skills/`.

Project and user scope compared

Feature	Project scope	User scope
Commands folder	`.claude/commands/`	`~/.claude/commands/`
Skills folder	`.claude/skills/`	`~/.claude/skills/`
Who gets it	Every developer who clones or pulls the repo	Only you, across all your projects

EXAM TRAP

Editing a shared project skill to suit yourself affects teammates. Create a personal variant in `~/.claude/skills/` under a different name.

Commands and skills

A command file's name becomes the command; skills are its successor. The file `.claude/commands/deploy.md` creates `/deploy`. Each skill is a directory with a `SKILL.md` file, whose frontmatter needs a name and a description. The description decides when Claude invokes the skill.

Isolating a skill with context: fork

Fork skills whose output would flood the main session. With `context: fork`, the skill runs in an isolated sub-agent context. Use it for verbose output, such as codebase analysis, or for exploratory work,

such as brainstorming alternatives.

Example: The SKILL.md frontmatter line that runs a skill in an isolated sub-agent context.

```
context: fork
```

Restricting tools and asking for arguments

`allowed-tools` **limits tools;** `argument-hint` **asks for parameters.** You might limit a skill to file write operations, for example, to prevent destructive actions. When someone runs the skill with no arguments, `argument-hint` asks them for the parameters it needs. A project skill can combine `context: fork` with `allowed-tools`.

EXAM TRAP

Do not count on `allowed-tools` pre-approvals everywhere. In SDK sessions, Claude Code ignores them when managed settings set `allowManagedPermissionRulesOnly`.

Skills versus CLAUDE.md

Skills load on demand; CLAUDE.md is always loaded. Put task-specific workflows in skills and universal standards in CLAUDE.md. At startup only each skill's name and description load, and Claude reads the full `SKILL.md` once the skill becomes relevant.

EXAM TRAP

A team-wide coding standard does not belong in a skill. Always-loaded universal standards go in CLAUDE.md.

Key takeaways

- Team commands go in `.claude/commands/`; personal ones go in `~/.claude/commands/`.
- Use `context: fork` for skills with verbose or exploratory output.
- Use `allowed-tools` to restrict tools and `argument-hint` to prompt for parameters.
- Skills are on-demand workflows; CLAUDE.md holds always-loaded standards.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Extend agents with skills - Claude Code Docs (opens in a new tab) · Skill authoring best practices - Claude Platform Docs (opens in a new tab) · Get started with Agent Skills in the API - Claude Platform Docs (opens in a new tab)

Apply path-specific rules for conditional convention loading

Start here

THINK FIRST

When should a convention live in a path-scoped rule instead of a CLAUDE.md file inside a subdirectory?

Why it matters: Questions describe where a convention's files sit in the codebase and ask which rule setup loads it only when it is needed.

A rule file in `.claude/rules/` can list glob patterns under `paths:` in its frontmatter. Claude then loads that rule only when it works with a matching file. Unrelated conventions stay out of context, wherever the matching files sit.

Key terms

Term	Meaning
Path-scoped rule	A rule file in `.claude/rules/` whose `paths:` frontmatter limits when it loads.
paths: frontmatter	The YAML field at the top of a rule file that holds the glob patterns for its trigger files.
Glob pattern	A file pattern such as `**/*.test.tsx` that matches files by name in any directory.
Per-directory CLAUDE.md	A CLAUDE.md inside a directory, next to its code, where that directory's owners keep their conventions.
Context rot	The drop in accuracy and recall as the number of tokens in context grows.

Rules as separate files

Split a large CLAUDE.md into topic files in `.claude/rules/`. Each file holds one topic, such as `testing.md`, `api-conventions.md` or `deployment.md`. Together they replace one monolithic CLAUDE.md. Rule files can live at project or global scope.

Scoping a rule with paths

A `paths:` field in YAML frontmatter holds the rule's glob patterns. Claude loads the rule only when it reads, writes or edits a matching file. The field is optional, so a rule without it stays unscoped.

Example: Rule frontmatter that loads API conventions only for files under the src/api folder.

```
paths: ["src/api/**/*"]
```

- `src/api/**/*`: matches every file below src/api

Why scoping saves context

Rules that match no current file stay out of context. That cuts irrelevant context and token usage. It matters because more context is not automatically better: accuracy and recall drop as tokens grow, which is called context rot.

EXAM TRAP

One big root CLAUDE.md is not the lean choice in a large codebase. It tends to spend context on unrelated instructions, or it stays too generic to help.

Matching files by type

A glob like `/.test.tsx` matches test files in any directory. The pattern targets a file type, not a folder. One rule can carry testing conventions for every test file, wherever it lives. A broader pattern such as `**/*.test.*` works the same way.

Rules or directory CLAUDE.md

Pick a path-scoped rule when one convention spans scattered paths. A per-directory CLAUDE.md sits beside its code and suits conventions that one directory's owners maintain. Test files spread across the codebase fit a glob rule better.

Path-scoped rule and per-directory CLAUDE.md compared

Feature	Path-scoped rule	Per-directory CLAUDE.md
Where the file lives	Centrally in `.claude/` at the repo root	Inside the directory, next to its code
When it loads	When Claude works with a file matching its `paths:` glob	At launch from that directory, or on demand
Best fit	One rule for many scattered paths, or all conventions in one place	Directory owners keeping their own conventions

EXAM TRAP

Directory CLAUDE.md files are the wrong tool for a convention that spans many directories, like scattered test files. A glob-pattern rule covers them from one place.

Scoped rules and compaction

Compaction summarizes the message history; scoped rules then reload on demand. A scoped rule loads into the message history, so compaction summarizes it away with everything else. Claude Code then reloads it on demand rather than re-injecting it. Unscoped rules and the project-root CLAUDE.md come back from disk.

EXAM TRAP

Do not count on a scoped rule surviving compaction. If it must persist, remove its `paths:` frontmatter or move it into the project-root CLAUDE.md.

Rules in the Agent SDK

The `project` setting source loads `.claude/rules/.md` files. It reads rules from the working directory and from every parent directory. User rules in `~/.claude/rules/` need the `user` source. Omitting `settingSources` loads user, project and local sources.

Key takeaways

- Put glob patterns under `paths:` in a rule's YAML frontmatter so it loads only for matching files.
- Choose a glob rule over directory CLAUDE.md files when a convention spans many directories.
- A pattern like `**/*.test.tsx` applies a convention by file type, in any folder.
- Scoped rules reload on demand after compaction; drop `paths:` if a rule must always persist.

Sources: Set up Claude Code in a monorepo or large codebase - Claude Code Docs (opens in a new tab) · Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Explore the context window - Claude Code Docs (opens in a new tab) · Explore the .claude directory - Claude Code Docs (opens in a new tab) · Context windows - Claude Platform Docs (opens in a new tab) · Use Claude Code features in the SDK - Claude Code Docs (opens in a new tab)

Determine when to use plan mode vs direct execution

Start here

THINK FIRST

How do you tell whether a coding task needs plan mode or can go straight to direct execution?

Why it matters: Questions describe a coding task and ask whether to plan first, execute directly, or combine the two.

Plan mode suits complex, multi-file or architectural work, where exploring first prevents costly rework. Direct execution suits simple, well-scoped changes. Big tasks often plan first, then execute the approved plan.

Key terms

Term	Meaning
Plan mode	A mode where Claude reads files and explores, then writes a plan, without editing your source.
Direct execution	Making the change right away, suited to well-understood changes with a clear scope.
Explore subagent	A built-in subagent that isolates verbose discovery output and returns a summary.
Plan file	The file plan mode writes, stored in a plans/ directory and re-injected after compaction.

When to plan first

Use plan mode for complex, multi-file or architectural work. Choose it for large-scale changes, several valid approaches or architectural decisions. Examples are restructuring microservices, a library migration touching **45+** files, or picking between integration approaches with different infrastructure needs.

EXAM TRAP

A task is not simple just because it is one feature. A new feature with several valid implementation approaches still calls for plan mode.

When to execute directly

Go direct for simple, well-scoped changes. A single-file bug fix with a clear stack trace, or one added date validation conditional, has a clear scope. Such changes are well understood, so you can make them without a planning step.

Plan mode and direct execution compared

Feature	Plan mode	Direct execution
Best for	Large-scale, multi-file or architectural changes	Simple, well-scoped changes
Example	A library migration affecting **45+** files	A single-file bug fix with a clear stack trace

EXAM TRAP

Plan mode is not needed for every edit. A single-file fix with a clear stack trace fits direct execution.

What plan mode does

Claude explores and proposes, but does not edit your source. It reads files and runs shell commands to explore, then writes a plan. Edits stay blocked until you approve it, except in interactive terminal sessions with bypass permissions available. Press Ctrl+G to edit the plan yourself first.

Entering and leaving plan mode

Press Shift+Tab, or prefix one prompt with `/plan`. You can also start the CLI in plan mode, or set `defaultMode` to `plan` in `.claude/settings.json` for a project. Approving the plan exits plan mode and switches to the permission mode you chose, so Claude starts editing.

Example: Starts a Claude Code session in plan mode, which blocks edits until you approve a plan.

```
claude --permission-mode plan
```

Keeping discovery out of context

Send verbose discovery to the Explore subagent. The Explore subagent works in its own context and returns only a summary. That prevents context window exhaustion during multi-phase tasks, while the main conversation keeps the high-level view.

EXAM TRAP

The Explore subagent cannot be resumed later. It is one-shot and returns no agentId, so use a custom or general-purpose agent instead.

Plan, then execute

Investigate in plan mode, then implement directly. For a library migration, plan the approach first, then execute it. For changes across several packages, the plan file is re-injected after each compaction, so the plan survives a long session.

Key takeaways

- Plan mode fits large-scale, multi-file, architectural or multi-approach tasks.
- Direct execution fits well-understood changes with a clear scope.
- Plan mode explores and proposes without editing your source until you approve.
- Use the Explore subagent to keep verbose discovery out of the main context.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Choose a permission mode - Claude Code Docs (opens in a new tab) · Set up Claude Code in a monorepo or large codebase - Claude Code Docs (opens in a new tab) · Explore the .claude directory - Claude Code Docs (opens in a new tab) · Subagents in the SDK - Claude Code Docs (opens in a new tab)

Apply iterative refinement techniques for progressive improvement

Start here

THINK FIRST

When Claude's output is close but not right, which kind of feedback moves it forward fastest?

Why it matters: Questions describe a refinement that is stuck and ask which technique, or which way of giving feedback, will fix it.

Good refinement replaces vague prose with something Claude can check against. Examples show the exact transformation, tests show what passes, and questions surface what you forgot. How you group your fixes depends on whether they interact.

Key terms

Term	Meaning
Input/output example	A concrete pair showing what goes in and exactly what should come out.
Test-driven iteration	Writing tests before the code, then improving it by sharing the failures.

Term	Meaning
Interview pattern	Claude asks you questions to surface issues you had not considered before it builds.
Interacting issues	Problems whose fixes interact with each other. Address them together in one detailed message.

Show, do not describe

Give 2-3 concrete input/output examples when prose is misread. If a written description of a transformation gets read in different ways, examples settle it. They are the most effective way to show the exact result you expect.

EXAM TRAP

Rewriting the prose description in more detail is the tempting fix. When descriptions are read inconsistently, concrete examples work better.

Fix edge cases with test cases

Give the failing input and the output you expect. For a migration script that mishandles null values, a specific test case pins down the right behavior. Explicit examples of how to handle an edge case also help where Claude might misclassify an input.

Write the tests first

Cover behavior, edge cases and performance before any code exists. Then iterate by sharing the test failures. Claude Code can run the suite, read the errors, edit the code and rerun the tests.

Let Claude interview you

Have Claude ask questions before it implements anything. This surfaces points you may not have anticipated. It helps most in an unfamiliar domain, where design issues such as cache invalidation or failure modes are easy to miss.

One message or several

Send interacting issues together; fix independent ones in turn. When fixes interact, describe all of them in one detailed message. Fix independent issues sequentially, one per iteration.

Grouping feedback by how the issues relate

Feature	Interacting issues	Independent issues
How fixes relate	One fix changes another	Each fix stands alone
How to send them	All in a single detailed message	Sequentially, one per iteration

EXAM TRAP

Sequential fixes suit independent issues only. When the fixes interact, describe all of them together in a single detailed message.

Iterate, do not restart

Refine with feedback when the first attempt misses. Claude Code is conversational, so you correct course instead of starting over. A message typed while Claude works is queued and read as soon as the running tool calls finish.

Refine before you scale

Tune the prompt on a sample set before a large batch. This raises the first-pass success rate and cuts the cost of resubmitting. Judge readiness against the success criteria and thresholds you set in advance.

Key takeaways

- When prose is read inconsistently, show **2-3** concrete input/output examples instead.
- Write tests for behavior, edge cases and performance first, then iterate on the failures.
- Use the interview pattern in unfamiliar domains to surface design issues before building.
- Send interacting issues in one detailed message; fix independent issues one at a time.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Ticket routing - Claude Platform Docs (opens in a new tab) · How Claude Code works - Claude Code Docs (opens in a new tab)

Integrate Claude Code into CI/CD pipelines

Start here

THINK FIRST

What does Claude Code need to run unattended in a pipeline and produce findings your CI can act on?

Why it matters: Questions describe a CI job that hangs, produces unusable output or weak reviews, and ask which flag or setup fixes it.

In CI, nobody is there to answer prompts, so Claude Code must run non-interactively. It needs machine-readable output for the pipeline, project context from CLAUDE.md, and a reviewer that did not write the code.

Key terms

Term	Meaning
-p flag	The flag that runs Claude Code non-interactively, so a pipeline never waits for input.

Term	Meaning
--json-schema	A flag used with `--output-format json` to get structured, machine-parseable findings.
CLAUDE.md	A project file that gives Claude persistent context such as conventions and standards.
Independent review instance	A fresh session, without the generating session's reasoning, that reviews the code.

Run without a human

Use `-p` so a CI run never hangs waiting for input. Non-interactive mode suits pipelines. Unattended runs differ in small ways: a background command is stopped at its time limit, unlike in a local interactive session.

EXAM TRAP

Running Claude Code in CI without `-p` is the mistake to spot. The run can then hang waiting for interactive input that never comes.

Make output machine-readable

Pair `--output-format json` with `--json-schema`. The findings then come back structured and parseable. Your pipeline can post them automatically as inline comments on the pull request.

Give context through CLAUDE.md

CLAUDE.md carries project context into CI runs. Record testing standards, what makes a test valuable, and the fixtures available. That raises test quality and cuts low-value tests. The review command follows CLAUDE.md even in a `-p` run.

EXAM TRAP

Do not look for a `/code-review` file on disk to hold review rules: it is a bundled skill with none. Put the rules every review must follow in CLAUDE.md.

Avoid duplicate tests

Put the existing test files in context before generating tests. Claude can then see which scenarios the suite already covers. It suggests new cases rather than repeating old ones.

Review with fresh eyes

The session that wrote the code is a weak reviewer of it. It keeps the reasoning that produced the code, so it rarely questions its own choices. An independent instance catches subtle issues better than self-review instructions or extended thinking.

Self-review compared with an independent review instance

Feature	Same-session self-review	Independent instance
Reasoning context	Keeps the reasoning from generation	Starts without that prior reasoning
Catching subtle issues	Weaker, even with self-review instructions	More effective

EXAM TRAP

Asking the same session to review its work, or turning on extended thinking, is not the fix. Use an independent review instance.

Reviews on later pushes

Automatic follow-up reviews focus on what changed. In managed Code Review, a review that starts after a push looks at changes since the last review. It posts fewer nits and pre-existing findings.

Key takeaways

- Run Claude Code in CI with `-p` so it never waits for interactive input.
- Use `--output-format json` with `--json-schema` for findings your pipeline can post as inline comments.
- Document testing standards and fixtures in CLAUDE.md to raise the quality of generated tests.
- Review generated code with an independent instance, not the session that wrote it.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Use Claude Code features in the SDK - Claude Code Docs (opens in a new tab) · Interactive mode - Claude Code Docs (opens in a new tab) · Review locally with /code-review - Claude Code Docs (opens in a new tab) · Read and request reviews on a pull request - Claude Code Docs (opens in a new tab)

Domain 4 | 20% of the exam | 20 min read

Prompt Engineering & Structured Output

Overview

This domain is about 20% of the exam, roughly 12 of its 60 questions.

Every point below comes from Anthropic's official documentation; each section lists its sources.

Design prompts with explicit criteria to improve precision and reduce false positives

Start here

THINK FIRST

Why do instructions like "be careful" fail to cut false positives in a review prompt, and what works instead?

Why it matters: Questions show a noisy review or classification prompt and ask which change improves precision and restores trust.

Precision comes from saying exactly what counts, not from asking for caution. Name the issues to report and the ones to skip, and define each severity level with examples. Noisy categories cost trust in the accurate ones too.

Key terms

Term	Meaning
False positive	A flagged finding that turns out not to be a real problem.
Explicit criterion	A concrete rule for when to flag, such as a comment contradicting the code.
Confidence-based filtering	Reporting only what the model feels sure about, instead of naming categories.
REVIEW.md	A file of review-only instructions: what to flag, at what severity, what to skip.

Say exactly what to flag

Replace vague checks with concrete, testable conditions. "Check that comments are accurate" leaves the model guessing. A rule to flag a comment only when its claim differs from what the code does tells it precisely when to report.

Caution is not a criterion

"Be conservative" does not raise precision; categories do. General advice such as "only report high-confidence findings" falls short of specific categorical criteria. A model's own sense of confidence is also poorly calibrated on hard cases.

EXAM TRAP

Asking the model to report only findings it is confident about seems safe. It does not improve precision the way specific categorical criteria do.

Name what to report and skip

Report bugs and security issues; skip minor style and local patterns. Write the categories down instead of filtering on confidence. Lists of paths to skip, such as generated code, lockfiles and vendored dependencies, remove more noise.

Vague instructions versus explicit criteria

Feature	Vague instruction	Explicit criterion
Comment accuracy	Check that comments are accurate	Flag only when claimed behavior contradicts the code
Which findings to post	Only report high-confidence findings	Report bugs and security; skip minor style
Behavior claims	Infer the behavior from naming	Require a file:line citation as evidence

Define every severity level

Give each severity level explicit criteria and a code example. That keeps classification consistent. For example, reserve Important for findings that break behavior, leak data or block a rollback. Style, naming and refactoring are Nit at most.

Protect developer trust

A noisy category undermines trust in the accurate ones too. When one category produces many false positives, developers doubt the accurate findings as well. Disable that category for a while to restore trust, and improve its prompt in the meantime.

EXAM TRAP

Leaving a high false-positive category running does not just waste time on that category. It erodes confidence in accurate findings too.

Raise the bar where needed

Demand evidence or near-certainty for risky finding classes. Behavior claims can require a file:line citation rather than a guess from naming. Paths that need only light review can carry a rule such as reporting only near-certain, severe issues.

Key takeaways

- Explicit criteria beat vague instructions like "check that comments are accurate".
- "Be conservative" and confidence-based filtering do not improve precision; categorical criteria do.
- High false-positive categories undermine trust in accurate ones, so pause them while fixing their prompts.
- Consistent severity needs explicit criteria with concrete code examples for each level.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Customize Code Review - Claude Code Docs (opens in a new tab)

Apply few-shot prompting to improve output consistency and quality

Start here

THINK FIRST

When detailed instructions still give uneven results, when are examples the right fix and when are they not?

Why it matters: Questions describe inconsistent or inaccurate output and ask whether few-shot examples, clearer criteria or something else fixes it.

Examples show Claude what good output looks like, which beats describing it in the abstract. Well-chosen examples also teach judgment that carries over to new cases. They do not fix a missing tool description or guarantee a required step.

Key terms

Term	Meaning
Few-shot example	An example of the output you want, given to Claude instead of abstract instructions.
Ambiguous scenario	A case where several actions look plausible and the right one needs judgment.
False positive	A reported issue that is really an acceptable pattern, not a genuine problem.
Categorical criteria	Specific, named rules for what counts, instead of general advice like being conservative.

Examples beat abstract rules

Show the output you want when instructions alone stay inconsistent. Few-shot examples are the most effective way to get consistently formatted, actionable output. For a code review, show each finding with its location, issue, severity and suggested fix.

Teach the hard calls

Use 2-4 targeted examples for ambiguous scenarios. Each example should show why one action was chosen over the plausible alternatives. Separately, few-shot examples let the model generalize judgment to novel patterns, rather than matching only the cases you listed.

EXAM TRAP

Do not assume examples only teach the model to match the cases they show. Few-shot examples let it generalize judgment to novel patterns as well.

Cut false positives

Contrast acceptable code patterns with genuine issues. Paired examples reduce false positives and still let the model generalize. Vague advice such as "only report high-confidence findings" does not improve precision the way specific categorical criteria do.

EXAM TRAP

Telling a reviewer to "be conservative" does not improve precision; specific categorical criteria do. Separately, examples that contrast acceptable patterns with genuine issues reduce false positives.

Make severity consistent

Define each severity level with concrete code examples. Explicit criteria plus an example per level give consistent classification across reviews.

Steady extraction across formats

Show correct extraction from documents with varied structures. Examples can cover inline citations versus bibliographies, or narrative text versus tables. They help with informal measurements too, reducing hallucination. They also address required fields that come back empty or null.

Where examples fall short

Examples do not replace good tool descriptions or code checks. If tool selection fails because descriptions are thin, examples add tokens without fixing the cause. If a step must always happen first, examples still rely on probabilistic compliance.

Matching the fix to the cause

Problem	Root cause	Better first fix
Inconsistent output format	Relying on detailed instructions alone	Few-shot examples of the desired format
Wrong tool picked	Minimal tool descriptions	Expand the tool descriptions
Poor escalation calibration	Unclear decision boundaries	Explicit escalation criteria with examples
Required step skipped	Prompts give only probabilistic compliance	Programmatic enforcement

Key takeaways

- Few-shot examples are the most effective fix when detailed instructions give inconsistent output.
- For ambiguous cases, use **2-4** examples that explain why one action beat the alternatives.
- Examples of acceptable code versus genuine issues reduce false positives.
- When tool descriptions are minimal, expand them first; examples only add token overhead.

Sources: Increase output consistency - Claude Platform Docs (opens in a new tab) · Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab)

Enforce structured output using tool use and JSON schemas

Start here

THINK FIRST

How do you guarantee an extraction always comes back as valid structured data, and what can still go wrong?

Why it matters: Questions describe an extraction task and ask which tool_choice setting or schema design gives reliable, honest output.

Defining the extraction as a tool with a JSON schema is the most reliable way to get well-formed output. `tool_choice` decides whether a tool call happens at all. Schema design decides whether the values are honest.

Key terms

Term	Meaning
Extraction tool	A tool whose input schema is the data shape you want; you read the data from the tool_use block.
tool_choice	The setting that controls whether Claude must call a tool, and which one.
strict: true	A tool setting that constrains sampling so tool inputs always match the schema.
Nullable field	A field that may be null, so the model can admit the value is not there.

Extraction as a tool call

Put the schema in the tool's input and read the data from tool_use. Tool use with a JSON schema is the most reliable route to schema-compliant output. With `strict: true`, the input always follows the schema and the tool name is always valid.

Pick the right tool_choice

`auto` **allows text,** `any` **forces a tool, forced selection names one.** The setting decides how much freedom Claude has to skip the tool or pick among several.

The three tool_choice settings compared

Setting	Must call a tool?	Typical use
`"auto"`	No, it may return text instead	Claude decides whether a tool is needed
`"any"`	Yes, but Claude chooses which tool	Several extraction schemas, unknown document type
`{"type": "tool", "name": "..."}`	Yes, the named tool	Make `extract_metadata` run before enrichment

EXAM TRAP

`auto` does not guarantee structured output, because Claude may answer in text. Use `any` when you have several schemas and do not know the document type.

Force a step to run first

Forced selection makes one named tool run before the others. For example, force `extract_metadata` before any enrichment tools. Handle the later steps in follow-up turns.

Valid JSON is not correct data

Tool use removes syntax errors, not semantic errors. Line items may still fail to add up to the total, or values may land in the wrong field. Checks such as a calculated total beside the stated total catch these gaps.

EXAM TRAP

A strict schema does not prove the numbers are right. It guarantees the shape, so semantic checks are still your job.

Make missing data optional

Use nullable fields for values a document may not contain. Some documents simply lack a value. A nullable field lets the model return null there, instead of fabricating something to fill the gap.

EXAM TRAP

Do not let the schema force a guess. When a field may be absent from the document, make it nullable so the model can return null.

Leave room in your enums

Add `"unclear"` and an `"other"` value with a detail field. The unclear value covers ambiguous cases. The other value plus a detail string lets categories grow without breaking the schema. Add format normalization rules to the prompt for messy inputs.

Strict schema limits

Strict schemas allow at most 24 optional parameters per request. Every parameter not listed in `required` counts toward that total. Programmatic tool calling does not support tools that set `strict: true`.

Key takeaways

- Tool use with a JSON schema is the most reliable way to get schema-compliant output.
- Use `tool_choice: "any"` when there are several schemas and the document type is unknown.
- Strict schemas stop syntax errors but not sums that fail or values in the wrong field.
- Make fields nullable when the document may lack them, so the model returns null.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Strict tool use - Claude Platform Docs (opens in a new tab) · Get structured output from agents - Claude Code Docs (opens in a new tab) · Structured outputs - Claude Platform Docs (opens in a new tab) · Programmatic tool calling - Claude Platform Docs (opens in a new tab)

Implement validation, retry, and feedback loops for extraction quality

Start here

THINK FIRST

When will sending a failed extraction back to the model fix it, and when is another try wasted effort?

Why it matters: Questions show a failed extraction and ask whether a retry can fix it, or how to design the check that catches it.

Schemas stop malformed output, but not wrong values. Validation catches the wrong values, and a retry that carries the exact error lets the model fix them. A retry only helps when the answer is actually in the document.

Key terms

Term	Meaning
Syntax error	Output that breaks the JSON schema itself, such as a bad field type or invalid JSON.
Semantic error	Output that fits the schema but is wrong, such as totals that do not add up.

Term	Meaning
Retry-with-error-feedback	A retry that adds the specific validation errors to the prompt to steer the fix.
detected_pattern	A field on each finding that records which code construct triggered it.

Two kinds of extraction error

Tool use removes syntax errors, not semantic ones. A strict JSON schema through tool use keeps the output well formed. It cannot check that line items match the total, or that each value sits in the right field.

Syntax errors and semantic errors compared

Feature	Syntax error	Semantic error
Example	Invalid JSON or a wrong field type	Line items that do not add up to the total
Stopped by strict schemas	Yes, through tool use or constrained decoding	No, the output still fits the schema
How you catch it	The schema enforces it	A check such as a calculated total beside the stated total

EXAM TRAP

A strict schema does not prove the data is right. Values can be in the wrong field or fail to sum and still pass the schema.

Retry with the error attached

Send the document, the failed extraction and the exact error. The follow-up carries all three pieces in one request. Adding the specific validation error tells the model what to fix, so it can correct itself.

Know when a retry cannot help

Retries fix format and structure, not missing information. A format mismatch or a structural slip is fixable on the next try. If the needed fact lives only in an outside document you never provided, no retry will find it. Track which errors are which.

EXAM TRAP

Retrying more times does not help when the value is absent from the document. If the task might lack that information, make the field optional.

Build checks into the output

Extract a calculated total next to the stated total. When the two differ, the gap flags a likely error for review. A `conflict_detected` boolean does the same job when the document itself disagrees with itself.

Learn from dismissed findings

A `detected_pattern` field shows which constructs cause findings. When developers dismiss findings, you can group them by pattern. That shows which patterns produce false positives, so you can tune the checks instead of guessing.

Retry limits in the Agent SDK

The SDK re-prompts on a schema mismatch, but only up to a limit. Past that limit, you get an error instead of structured data. Its subtype is `error_max_structured_output_retries`. Common causes are an overly complex schema or an ambiguous task.

Errors the model can act on

MCP tool execution errors are meant for self-correction. A bad date format comes back in the tool result with `isError: true`, and the model can retry with new parameters. Protocol errors, like an unknown tool, are less likely to be fixable by the model.

Key takeaways

- Strict schemas stop syntax errors, but not semantic errors such as totals that do not add up.
- A useful retry resends the source document and the failed output, plus the exact validation error.
- Retries fix format and structure problems, never information that is missing from the document.
- A `detected_pattern` field turns dismissed findings into data about false positives.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Structured outputs - Claude Platform Docs (opens in a new tab) · Get structured output from agents - Claude Code Docs (opens in a new tab) · Tools - Model Context Protocol (opens in a new tab)

Design efficient batch processing strategies

Start here

THINK FIRST

How do you decide whether a workload belongs in the Message Batches API, and how do you run it reliably?

Why it matters: Questions describe a workload's latency needs and ask whether to batch it, how often to submit, or how to handle failures.

The Message Batches API trades speed for cost: a **50%** discount, but up to **24** hours to process and no latency SLA. Use it for work nobody is waiting on. Track every request by `custom_id`, so you can match results and resubmit only the failures.

Key terms

Term	Meaning
Message Batches API	An asynchronous API that processes many requests at a **50%** discount, with no guaranteed latency.
custom_id	A unique ID on each batch request, used to match each result to its request.
Blocking workflow	Work that cannot move on until the answer arrives, such as a pre-merge check.
Latency SLA	A commitment on how quickly results arrive; the Message Batches API offers none.

What batching costs and gives

Half the price, with no promise on speed. All batch usage, including input, output and special tokens, gets a **50%** discount. Most batches finish within **1** hour, but processing can take up to **24** hours, and there is no guaranteed latency SLA.

EXAM TRAP

A batch is not certain to finish in **24** hours. If processing does not complete in that time, the batch expires.

Match the API to the workflow

Blocking work goes synchronous; latency-tolerant work goes to batches. Ask whether anything waits for the answer. A pre-merge check blocks the merge, so it needs the synchronous API. Overnight and weekly analysis can wait, so batching fits.

Which API fits which workload

Workload	API	Reason
Pre-merge check	Synchronous API	Blocking: the merge waits for the result
Overnight report	Message Batches API	Non-blocking and latency-tolerant
Weekly audit	Message Batches API	Non-blocking and latency-tolerant
Nightly test generation	Message Batches API	Non-blocking and latency-tolerant

EXAM TRAP

Moving pre-merge checks to batches to save money is the wrong answer. With no latency SLA, a blocking check could wait for hours.

No tool loops inside a batch

A batch request cannot run multi-turn tool calling. Inside a single batch request, tools cannot run mid-request and have their results returned. A task where Claude must call a tool, read its result and

continue does not fit one batch request.

Correlate results with custom_id

Results can come back in any order. Give each request a meaningful, unique `custom_id` to match results to requests. It must be **1** to **64** characters of letters, digits, hyphens and underscores. Results are available once all messages complete, or after **24** hours if that comes first.

Plan the submission frequency

Fit the submission schedule to the SLA you must meet. With up to **24** hours of batch processing, the exam guide's example submits in **4**-hour windows to guarantee a **30**-hour SLA. Work out total processing time against the SLA before choosing how often to submit.

Handle failures by custom_id

Resubmit only the failed requests, after fixing them. One failed request does not affect the others in its batch. Errored requests, such as invalid requests or internal server errors, create no message and are not billed. Find failures by `custom_id`, then fix them, for example by chunking documents that exceeded context limits.

EXAM TRAP

Resubmitting the whole batch is unnecessary when a few requests fail, because the others were processed normally. Send only the failed documents again.

Refine before you scale

Test the prompt on a sample before sending large volumes. Refining the prompt on a sample set raises first-pass success and cuts the cost of repeated resubmissions. Also test one request shape first with the Messages API, so validation errors appear before the batch.

Batching documents

The Message Batches API can process many PDFs at once. Split large PDFs into chunks when needed. Size and page limits cover the whole request payload. Upload large PDFs with the Files API and reference them by `file_id` to keep payloads small.

Key takeaways

- Batches cost **50%** less but can take up to **24** hours, with no latency SLA.
- Use the synchronous API for blocking pre-merge checks and batches for overnight or weekly analysis.
- A batch request cannot execute tools mid-request and continue with their results.
- Match results by `custom_id`, and resubmit only the failed requests after fixing them.
- Refine the prompt on a sample set before batch-processing large volumes.

Sources: Batch processing - Claude Platform Docs (opens in a new tab) · Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · PDF support - Claude Platform Docs (opens in a new tab)

Design multi-instance and multi-pass review architectures

Start here

THINK FIRST

Why does a second, independent Claude instance catch problems that self-review misses?

Why it matters: Questions describe a review setup that misses subtle bugs or contradicts itself, and ask how to restructure the review.

A model that wrote the code keeps its own reasoning in context, so it rarely questions it. Review works better with a fresh, independent instance, with large reviews split into focused passes, and with confidence attached to each finding for routing.

Key terms

Term	Meaning
Self-review	Asking the same session that generated code to review it, with its reasoning still in context.
Independent review instance	A separate Claude instance that reviews the output without the generator's reasoning context.
Integration pass	A review pass that analyzes data flow across files, run after the per-file passes.
Attention dilution	The loss of focus one pass suffers when it reviews a large multi-file change at once.

Limits of self-review

A model rarely questions its own reasoning in the same session. It keeps the reasoning context from generation, so its earlier choices look sound to it. That is why the same session reviews its own changes less effectively than an independent instance does.

EXAM TRAP

Telling the generating session to review itself harder, or giving it extended thinking, is not the fix. An independent instance without that context catches more.

Using an independent reviewer

Review with an instance that lacks the generator's context. A new Claude Code session starts with a fresh context window and none of the earlier history. A non-fork subagent also starts fresh. Either can review the generated code without inheriting the reasoning that produced it.

EXAM TRAP

A forked session or forked subagent is not independent. It copies the conversation so far, including the generator's reasoning.

Splitting large reviews

Run per-file passes, then a cross-file integration pass. One pass over many files dilutes attention and can produce contradictory findings. Focused passes catch local issues, and a separate pass traces data flow across files. Prompt chaining can run these steps in sequence.

The two kinds of review pass compared

Feature	Per-file pass	Integration pass
Scope	One file at a time	Across files
Looks for	Local issues	Cross-file data flow problems
Order	First, for each file	After the per-file passes

Confidence for review routing

Have the verification pass report confidence per finding. Each finding then carries a confidence level, so review routing can be calibrated. Self-reported confidence is poorly calibrated on its own, so tune routing thresholds with labeled validation sets.

EXAM TRAP

A high self-reported confidence does not prove a finding is right. The agent can be confidently wrong on hard cases.

Precise criteria over vague caution

Tell reviewers what to report, not just to be careful. Instructions like "be conservative" improve precision less than specific categorical criteria. Requiring evidence, such as a file:line citation, before a class of finding is posted also cuts false positives.

Multi-agent review in practice

Claude's managed Code Review chains independent passes. Several agents find issues in parallel, and a verification step filters out false positives. The remaining findings are deduplicated and ranked by severity, then posted. Ultrareview goes further and independently reproduces every finding.

Key takeaways

- Self-review in the same session is weak because the model keeps its generation reasoning.
- Use an independent instance, not a fork, to review generated code.
- Split large reviews into per-file local passes plus a cross-file integration pass.
- Attach self-reported confidence to findings, but calibrate it before routing on it.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · How Claude Code works - Claude Code Docs (opens in a new tab) · Customize Code Review - Claude Code Docs (opens in a new tab) · Code Review - Claude Code Docs (opens in a new tab) · Find bugs with ultrareview - Claude Code Docs (opens in a new tab)

Domain 5 | 15% of the exam | 18 min read

Context Management & Reliability

Overview

This domain is about 15% of the exam, roughly 9 of its 60 questions.

Every point below comes from Anthropic's official documentation; each section lists its sources.

Manage conversation context to preserve critical information across long interactions

Start here

THINK FIRST

How do you keep exact amounts, dates and key findings intact as a conversation grows long?

Why it matters: Questions describe a long support or research session that loses details, and ask how to structure context so they survive.

Tool results use tokens out of proportion to their relevance, and models may omit findings from the middle of long inputs. Keep exact facts in a persistent block outside summarized history, and put key findings summaries first.

Key terms

Term	Meaning
Case facts block	A persistent block of amounts, dates, order numbers and statuses in each prompt, kept outside summarized history.
Lost in the middle	Models handle the start and end of long inputs reliably but may omit findings from the middle.
Context layer	A separate store of structured issue data, such as order IDs and statuses, for multi-issue sessions.
Context editing	Removing old tool_result blocks from history once they are used, without restarting the conversation.

Keep exact facts out of summaries

Put transactional facts in a persistent case facts block. Amounts, dates, order numbers and statuses go in a block included in every prompt, outside the summarized history. In multi-issue sessions, keep each issue's

order ID, amount and status in a separate context layer.

EXAM TRAP

Do not rely on summarized history to carry exact amounts and dates. Keep them in a case facts block outside that history.

Tool results fill the context

Tool results use tokens out of proportion to their relevance. An order lookup may return **40+** fields when only **5** are relevant. In long runs, tool definitions and piled-up tool results can exhaust the context window. Context editing removes old tool results once used, and programmatic tool calling keeps intermediate results out of history.

Beat the lost-in-the-middle effect

Put a key findings summary at the start of long inputs. Models handle the start and the end of a long input well, yet they may drop findings placed in the middle. Lead with the key findings, then organize the detailed results under explicit section headers.

EXAM TRAP

A key finding buried in the middle of a long aggregated input may be skipped. Place a summary of key findings at the beginning.

Keeping context across turns

Context retention matters for conversation continuity. That includes a user who navigates away and later returns. In the support chatbot example, each user message is appended to a message list, and that whole list goes into the API call.

Structured output from subagents

Have subagents return findings with their metadata. In the research exercise, each finding carries a claim, an evidence excerpt, a source URL or document name, and a publication date. The parent may summarize a subagent's report, so ask explicitly when you need it kept verbatim.

Key takeaways

- Keep amounts, dates, order numbers and statuses in a case facts block in every prompt.
- Tool results can use tokens far out of proportion to their relevance, such as **40+** fields when **5** matter.
- Put key findings first and use explicit section headers in long inputs.
- Retaining context when a user leaves and comes back matters for continuity.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Manage tool context - Claude Platform Docs (opens in a new tab) · Customer support agent - Claude Platform Docs (opens in a new tab) · Subagents in the SDK - Claude Code Docs (opens in a new tab)

Design effective escalation and ambiguity resolution patterns

Start here

THINK FIRST

How should a support agent decide between resolving a case itself, asking a clarifying question, and handing it to a human?

Why it matters: Questions describe a support agent's escalation behavior and ask which trigger, prompt change or clarification fixes it.

Escalate on clear triggers: the customer asks for a human, policy has a gap, or the agent cannot progress. Sentiment and self-reported confidence do not measure complexity. When facts are ambiguous, ask rather than guess, and teach the boundaries with explicit criteria and few-shot examples.

Key terms

Term	Meaning
Escalation trigger	A condition that sends a case to a human: a request for one, a policy gap, or no meaningful progress.
Policy gap	A request the policy is silent or ambiguous about, such as competitor price matching.
First-contact resolution	The support scenario's goal: **80%** or more, while still knowing when to escalate.
Structured handoff	A summary for the human agent with customer details, root cause and recommended action.

The real escalation triggers

Escalate for a human request, a policy gap or no progress. Policy exceptions and gaps are triggers, not just complex cases. Everything else the agent should try to resolve itself. In the guide's support scenario, the agent handles returns, billing disputes and account issues.

Common support situations and the right response

Situation	Response	Reason
Customer explicitly asks for a human	Escalate immediately	Honor the request without investigating first
Frustrated customer, issue within the agent's ability	Acknowledge, then offer to resolve	Escalate only if the customer repeats the request
Competitor price match, policy covers own-site only	Escalate	Policy is silent on this request

Situation	Response	Reason
Lookup returns several matching customers	Ask for more identifiers	Do not pick one by heuristic

Explicit demand or frustration

An explicit request for a human is honored at once. Do not investigate first when the customer clearly asks for a person. Frustration alone is different: if the issue is within the agent's ability, acknowledge the frustration and offer a resolution. Escalate if the customer then repeats the preference.

EXAM TRAP

Trying to solve the problem before honoring an explicit request for a human is wrong. The request itself is the trigger.

Unreliable escalation signals

Sentiment and self-reported confidence do not measure complexity. Sentiment does not correlate with case complexity. An LLM's self-reported confidence is poorly calibrated, and the agent is already confidently wrong on hard cases.

EXAM TRAP

A confidence threshold sounds precise but fails, because the agent can be confidently wrong exactly where escalation matters most.

Fixing poor calibration

Add explicit escalation criteria and few-shot examples to the prompt. In the guide's sample, an agent at **55%** first-contact resolution escalated simple damage replacements but handled policy exceptions alone. The root cause is unclear decision boundaries. Explicit criteria with examples are the proportionate first fix; a trained classifier needs labeled data and ML infrastructure.

EXAM TRAP

A separate classifier trained on historical tickets is over-engineered as a first step. Prompt optimization has not been tried yet.

Why few-shot examples work

Examples teach judgment that generalizes to new cases. Few-shot examples help the model handle novel patterns instead of matching only the listed cases. They also show how to treat ambiguous requests.

Ambiguous matches need clarification

Ask for another identifier when a lookup returns several matches. Instruct the agent to request more identifiers rather than pick a match by heuristic.

Enforce and hand off

Hooks can block policy-violating actions and route them to a human. For example, an interception hook can block refunds over **$500** and redirect them to escalation. Human agents may not see the transcript, so the handoff summary gives customer ID, root cause, refund amount and recommended action.

Key takeaways

- Escalate for an explicit human request, a policy gap or exception, or no meaningful progress.
- Honor an explicit request for a human immediately; for frustration alone, offer to resolve first.
- Sentiment and self-reported confidence are poor proxies for case complexity.
- With several matching records, ask for more identifiers instead of choosing by heuristic.
- Fix poor escalation calibration first with explicit criteria and few-shot examples in the prompt.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab)

Implement error propagation strategies across multi-agent systems

Start here

THINK FIRST

When a subagent fails partway through a task, what should it tell the coordinator so the whole system recovers well?

Why it matters: Questions describe a failing subagent or coordinator and ask which error handling design lets the system recover.

A coordinator can only recover from errors it understands. Subagents fix transient failures locally and pass up only what they cannot resolve, with structured context and partial results. The coordinator then decides what to retry, reroute or report as a coverage gap.

Key terms

Term	Meaning
Structured error context	Failure type, attempted query, partial results and alternative approaches, returned together with an error.
Valid empty result	A query that ran successfully but found no matches; it is not an error.
Coverage annotation	A note in synthesis output on which findings are well supported and which areas have gaps.
Transient error	A temporary failure, such as a timeout or service unavailability, unlike validation or permission errors.

What a useful error carries

Return the failure type, the attempt, partial results and alternatives. This structured context lets the coordinator make an informed recovery decision. Error metadata can also carry an `errorCategory` of transient, validation or permission, an `isRetryable` boolean, and a human-readable description. Marking errors retryable or not prevents wasted retries.

> **EXAM TRAP**
> A generic status such as "search unavailable" or "Operation failed" is not enough. It hides the context the coordinator needs to choose a recovery path.

Two anti-patterns

Neither hide errors nor stop the whole workflow over one failure. Both are anti-patterns. Instead, pass the error up with structured context and let the coordinator proceed with partial results. The guide's timeout exercise tests exactly this.

Recover locally first

Subagents handle transient failures themselves. Only errors a subagent cannot resolve go to the coordinator, together with what it attempted and any partial results. Claude Code itself makes up to **10** retries with exponential backoff on transient failures before it shows an error.

Classify the error

Transient, validation, business and permission errors need different handling. Timeouts and service unavailability are transient. Invalid input is a validation error, and a policy violation is a business error. For business rule violations, return `retriable: false` with a customer-friendly explanation, so the agent can tell the user what happened. MCP tools report failures with the `isError` flag.

Annotate coverage in synthesis

Show which findings are well supported and where gaps remain. When a source was unavailable, the synthesis should say which topic areas have gaps. In the guide's exercise, a simulated subagent timeout still gives the coordinator structured context. It proceeds with partial results and marks the gaps. Conflicting credible sources keep both values with attribution.

The coordinator's role

In hub-and-spoke designs, the coordinator owns error handling. All subagent communication goes through the coordinator, which gives observability, consistent error handling and controlled information flow. It can evaluate synthesis for gaps and send targeted queries back to subagents until coverage is sufficient.

Partial and failed runs in Claude Code

A run stopped at its turn limit comes back marked partial. `maxTurns` is the turn limit set for a subagent. When a subagent hits that limit, Claude Code flags its output as partial, which tells Claude the work is

unfinished. An API error that ends a subagent early, such as a rate limit, is different: it is never delivered as the subagent's result.

Key takeaways

- Errors that reach the coordinator carry failure type, attempted query, partial results and alternatives.
- Suppressing errors as empty success and ending a whole workflow on one failure are both anti-patterns.
- Subagents retry transient failures locally and propagate only what they cannot resolve.
- Synthesis output marks well-supported findings and the gaps left by unavailable sources.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Error reference - Claude Code Docs (opens in a new tab) · Subagents in the SDK - Claude Code Docs (opens in a new tab)

Manage context effectively in large codebase exploration

Start here

THINK FIRST

How do you keep an agent accurate while it explores a large codebase over a long session?

Why it matters: Questions describe a long exploration that starts to drift or crashes, and ask which context technique fixes it.

Long sessions fill the context with verbose discovery output until answers drift. Keep the main agent's context lean: delegate exploration to subagents, write findings to files, summarize between phases, and persist state so work survives a crash.

Key terms

Term	Meaning
Context degradation	Drift in long sessions: inconsistent answers and vague typical patterns instead of the specific classes found earlier.
Scratchpad file	A file where an agent records key findings and rereads them for later questions.
Manifest	A structured export of agent state that the coordinator loads on resume after a crash.
Subagent	A helper agent that works in its own context and returns only its final message to the parent.

Spotting context degradation

Drift shows up as vague, inconsistent answers. In a long session, the model starts citing "typical patterns" instead of the specific classes it found earlier. Its answers also stop agreeing with each other. Scratchpad files of key findings help counteract this drift.

Delegating exploration to subagents

Let subagents absorb verbose output. A subagent's tool calls and results stay in its own context; only its final message returns. Spawn one for a specific question, such as finding all test files or tracing refund flow dependencies. The main agent keeps the high-level coordination.

EXAM TRAP

A subagent does not see the parent's conversation unless it is a fork, which inherits the full context. Otherwise, put the file paths, errors and decisions it needs into its prompt.

Scratchpad files

Write key findings to a file and reread them. An agent that records its discoveries in a scratchpad file can look them up for later questions instead of trusting fading context.

Summarizing between phases

Summarize one phase before spawning the next. When an exploration phase ends, condense its key findings. Inject that summary into the initial context of the next phase's subagents, so they start from what is already known. A fresh start with a structured summary beats resuming with stale tool results.

Crash recovery with manifests

Each agent exports its state to a known location. On resume, the coordinator loads the manifest and injects the saved state into agent prompts. Passing captured results into a fresh session is often more robust than relying on session resume.

EXAM TRAP

Resuming the old session is not the robust default after a crash. Load the structured state exports and build fresh prompts from them.

Compacting a crowded context

Run `/compact` when discovery output fills the context. During extended exploration, `/compact` reduces context usage once verbose output piles up. Compaction keeps the active context small, so pair it with memory for anything that must survive the summary.

EXAM TRAP

Compaction alone does not preserve every finding. Keep must-keep facts in memory so they outlast the summary.

Key takeaways

- Inconsistent answers and talk of "typical patterns" signal context degradation.
- Delegate verbose exploration to subagents; only their final message returns to the parent.
- Persist findings in scratchpad files and phase summaries, and inject them into the next phase.

- For crash recovery, export state to manifests that the coordinator loads on resume.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab) · Create custom subagents - Claude Code Docs (opens in a new tab) · Subagents in the SDK - Claude Code Docs (opens in a new tab) · Run agents in parallel - Claude Code Docs (opens in a new tab) · Work with sessions - Claude Code Docs (opens in a new tab) · Memory tool - Claude Platform Docs (opens in a new tab)

Design human review workflows and confidence calibration

Start here

THINK FIRST

When is it safe to let high-confidence extractions skip human review?

Why it matters: Questions describe an extraction pipeline with limited reviewers and ask how to measure accuracy and route review.

A single overall accuracy number can hide weak document types or fields. Measure accuracy per segment, calibrate the model's field-level confidence against labeled data, and send only uncertain or conflicting cases to people.

Key terms

Term	Meaning
Aggregate accuracy	One overall accuracy figure for all documents, which can mask weak document types or fields.
Field-level confidence	A confidence score the model outputs for each extracted field, used to route review.
Calibration	Setting review thresholds by checking confidence scores against a labeled validation set.
Stratified random sampling	Sampling high-confidence extractions to measure their error rate and spot new error patterns.

Overall accuracy can mislead

A 97% overall score can hide weak segments. One document type or field may perform poorly while the average still looks high. Before you reduce human review, break accuracy down by document type and by field. Automate only when every segment performs consistently.

EXAM TRAP

A high overall accuracy is not enough to automate extractions. Check each document type and field first.

Field-level confidence scores

Have the model score its confidence for each field. Field-level scores let you route review attention to the uncertain fields. Self-reported confidence is poorly calibrated on its own, though, so an agent can be confidently wrong on hard cases.

EXAM TRAP

Raw self-reported confidence is not a safe threshold. Calibrate it against a labeled validation set first.

Calibrating review thresholds

Set thresholds against a labeled validation set. Compare the model's confidence scores with fields whose correct values you already know. Then pick the score below which a field goes to review, so attention lands where errors are likely.

Routing to human review

Send low-confidence and conflicting cases to people. Route an extraction to a reviewer when the model's confidence is low, or when the source documents are ambiguous or contradictory. This spends limited reviewer capacity on the cases that most need it.

Sampling high-confidence output

Keep sampling the extractions you automate. Use stratified random sampling of high-confidence extractions. It gives you an ongoing measure of their error rate and detects novel error patterns as they appear.

EXAM TRAP

Automating high-confidence extractions does not end measurement. Sampling them keeps their error rate visible.

Specific criteria beat confidence filters

Name what to report instead of asking for high confidence. An instruction like "only report high-confidence findings" improves precision less than specific categorical criteria. Say which issues to report and which to skip. Categories with many false positives also erode trust in the accurate ones.

Key takeaways

- Validate accuracy per document type and field before cutting human review.
- Calibrate field-level confidence thresholds with a labeled validation set.
- Route low-confidence or contradictory-source extractions to human reviewers first.
- Keep stratified random sampling running on high-confidence output to track errors.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab)

Preserve information provenance and handle uncertainty in multi-source synthesis

Start here

THINK FIRST

How does a multi-agent research system keep track of where each claim came from, and what should it do when sources disagree?

Why it matters: Questions describe a research pipeline that lost citations or mishandled conflicting data and ask which design fixes it.

Every claim should travel with its source from the first agent to the final report. Summaries that drop that link lose attribution. When sources disagree, the system keeps both values, labeled, and lets the coordinator decide how to reconcile them before synthesis.

Key terms

Term	Meaning
Claim-source mapping	A structured link from each claim to its evidence, URL or document name, and date.
Provenance	Where a claim came from, kept with the claim as metadata through each step.
Contested finding	A finding that credible sources disagree on, reported apart from settled ones.
Coverage annotation	A note showing which findings are well supported and where sources were missing.

Where attribution gets lost

Summaries drop sources when claims lose their mappings. Each time findings are compressed, the link from claim to source can disappear. Numbers, percentages and dates can also blur into vague wording.

EXAM TRAP

A shorter summary is not a safer one. Compressing findings without their claim-source mappings is exactly how attribution gets lost.

Structure every finding

Keep content and metadata in separate fields. Each finding carries a claim, an evidence excerpt, a source URL or document name, and a publication date. The synthesis agent must preserve that attribution when it merges findings.

What a structured finding carries

Field	Kind	What it protects
Claim and evidence excerpt	Content	What was found and what supports it
Source URL or document name	Metadata	Attribution through every later step
Publication or collection date	Metadata	Correct reading of changes over time

Dates prevent false conflicts

Require publication or collection dates in every output. Two figures from different years may both be right for their time. Without dates, that difference can be misread as a contradiction.

Keep conflicting values

Report both values with their sources; never pick one at random. When two credible sources disagree, annotate the conflict. Document analysis should finish with both values marked, and the coordinator decides how to reconcile them before synthesis.

EXAM TRAP

Choosing the more recent or more likely value silently is the wrong move. Keep both values, each with its source, and flag the conflict.

Separate settled from contested

Give well-established and contested findings their own sections. Keep each source's own characterization and its method context. Coverage notes can also show where gaps exist because sources were unavailable.

Fit the format to the content

Tables for financial data, prose for news, lists for technical findings. Give each content type the form that suits it, instead of one uniform format for everything in the synthesis.

Key takeaways

- Attribution is lost when findings are summarized without their claim-source mappings.
- Each finding should carry a claim, evidence excerpt, source URL or document name, and date.
- When credible sources conflict, keep both values with attribution instead of choosing one.
- Dates in structured outputs stop differences over time from looking like contradictions.

Sources: Claude Certified Architect – Foundations Exam Guide Version 1.0 · Effective July 2026 · Exam code: CCAR-F · This guide i (opens in a new tab)

MockCert

Take a Claude Certified Architect – Foundations mock exam the way the real one feels

5 timed mock exams · 60 questions in 120 minutes each · one is free

- Free to start: study notes, explained quizzes, flashcards and the first mock exam
- Timed mock exams built to the official exam guide
- A verifiable Certificate of Assessment when you pass a premium mock

Explore Claude Certified Architect – Foundations on MockCert

Open it on your phone

mockcert.com/ccar-f/

Scan the code with your phone's camera.