FREE PREVIEW · ~30% OF THE FULL GUIDE

Unofficial Study Guide for the CCAR-F Exam

Claude Certified Architect — Foundations · sample edition

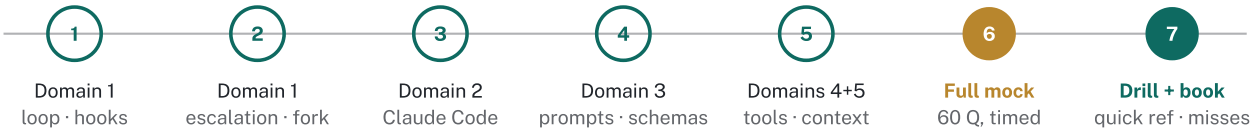
CCAR-F | 60 Q · 120 MIN | PASS 720/1000

An independent study resource by Zehon.com · August 2026
This free preview contains complete, unabridged sections of the full guide: Day 1 of the 7-day plan, all of Domain 1, and 7 practice questions with full answer explanations.

The 7-day plan

Read a domain, get your hands dirty, checkpoint with the quiz — every day.

~2 hours a day for 7 days. Every day pairs reading with hands-on work — CCAR-F rewards muscle memory for flags, paths, and parameters. Don't burn the 36 quiz questions early: days 2–5 use per-domain slices as checkpoints, day 6 is a full mock, day 7 re-runs your misses.



Materials: this guide · the 36-question quiz bank · the community 60-question mock (github.com/paullarionov)

Study in domain-weight order; the mock on day 6 tells you whether to book.

Day 1 — Orientation + Domain 1 core

Study (~75 min): "Exam snapshot" + guide sections 1.1–1.4 — agent loop, stop_reason table, hub-and-spoke, Task tool, hooks vs prompts. Longer worked examples: community guide chapters 1 and 3.

Hands-on (~40 min): in a Python/TypeScript scratch project, write a minimal agent loop against the Messages API: 2 dummy tools, loop on `stop_reason == "tool_use"`, append `tool_result` blocks in a user message, exit on `end_turn`. Deliberately trigger `max_tokens` once. This one exercise cements ~8 exam facts.

✓ **CHECKPOINT**

From memory: all 7 stop_reason values, the exact JSON of a tool_result message, and the three agent-loop anti-patterns.

🔒 **IN THE FULL EDITION — THE REST OF THE PLAN**

- Day 2 — Domain 1 finish
- Day 3 — Domain 2: Claude Code configuration
- Day 4 — Domain 3: prompts and structured output
- Day 5 — Domains 4 + 5: tools, MCP, context

- Day 6 — Full mock + gap repair
- Day 7 — Drill, final quiz, logistics

The study guide

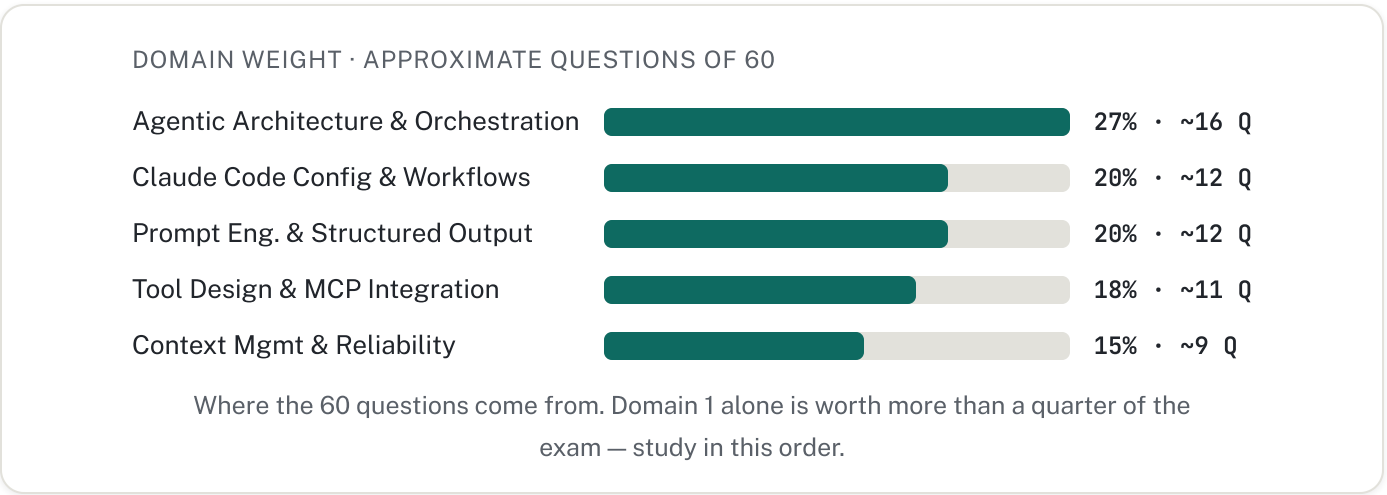
Five weighted domains — the most memorization-heavy exam of the suite.

Everything on this page maps to the five weighted domains of the official exam blueprint, cross-checked against Anthropic's docs. It is the most memorization-heavy exam of the suite — the tables and trap pairs are flashcard material, and every diagram shows a mechanism you need to know cold.

EXAM SNAPSHOT

60 Q 1 CORRECT OF 4	**4** RANDOM SCENARIOS PER SITTING	**120 min** CLOSED BOOK	**720** PASS, 100–1000 SCALE
$125 LIST PRICE — DISCOUNTS MAY APPLY	**0** GUESSING PENALTY — ANSWER ALL		

Pearson OnVUE, online proctored, booked via the Anthropic Partner Academy. Credly badge valid 12 months. Retakes: 14 days after the 1st attempt, 30 after the 2nd, 90 after the 3rd; max 4 attempts per rolling 12 months. Logistics reflect the official program pages as of August 2026 — verify before booking.



Where the 60 questions come from. Domain 1 alone is worth more than a quarter of the exam — study in this order.

The signature question style

A realistic production scenario, then four options that **all look legitimate** — one real flag, path, or parameter and three plausible fakes (`--batch` instead of `-p` , `~/.claude/commands/` instead of `.claude/commands/` , a `"role": "tool"` message instead of a `tool_result` block). Read every option to the end.

⚖ **FOUR META-RULES DECIDE MOST QUESTIONS**

Root cause beats symptom patch — fix the mechanism, don't add a layer. **Deterministic beats probabilistic when stakes are high** — hooks for money and compliance, prompts for preferences. **Cheapest sufficient fix first** — description rewrite < few-shot < architecture change. **Preserve information, annotate uncertainty** — partial results with coverage notes beat silent gaps.

✕ **OUT OF SCOPE — DON'T STUDY THESE**

Fine-tuning, auth/billing details, streaming/SSE, rate limits and cost math, prompt-caching internals, embeddings/vector DBs, computer use, vision, MCP server hosting, model internals.

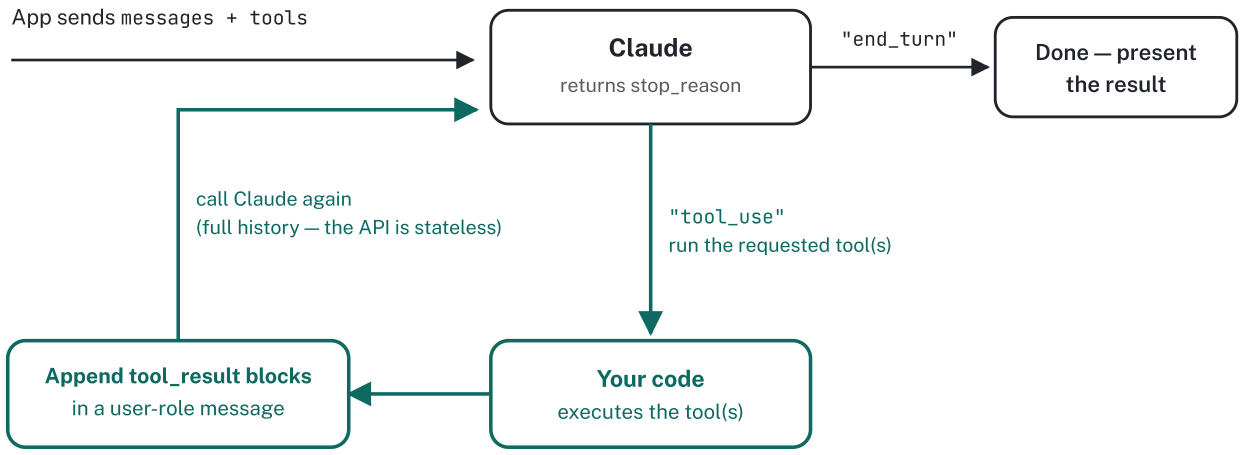
Official technology checklist: Claude Agent SDK · MCP · Claude Code · Claude API · Message Batches API · JSON Schema · Pydantic · CLAUDE.md · built-in tools (Read/Write/Edit/Bash/Grep/Glob).

Domain 1 · Agentic Architecture & Orchestration 27% · ~16 QUESTIONS

The agentic loop, coordinator/subagent design, hooks vs prompts, decomposition, escalation, and sessions. The biggest domain — most of its questions hang off two diagrams.

1.1 · The agentic loop

Every Agent SDK question assumes this cycle. The model decides the next step; your code executes tools and feeds results back.



The loop's only exit is `stop_reason == "end_turn"`. The teal path is the tool cycle: execute → append `tool_result` in a *user* message → call again.

THREE COMPLETION-DETECTION DISTRACTORS — REJECT ON SIGHT

Parsing assistant text for "Task completed" (brittle heuristic) · an iteration cap like `max_iterations=10` as the *primary* stop signal (safety net at most) · treating "response contains text" as done (Claude emits text alongside `tool_use` blocks). The only reliable signal is `stop_reason == "end_turn"`.

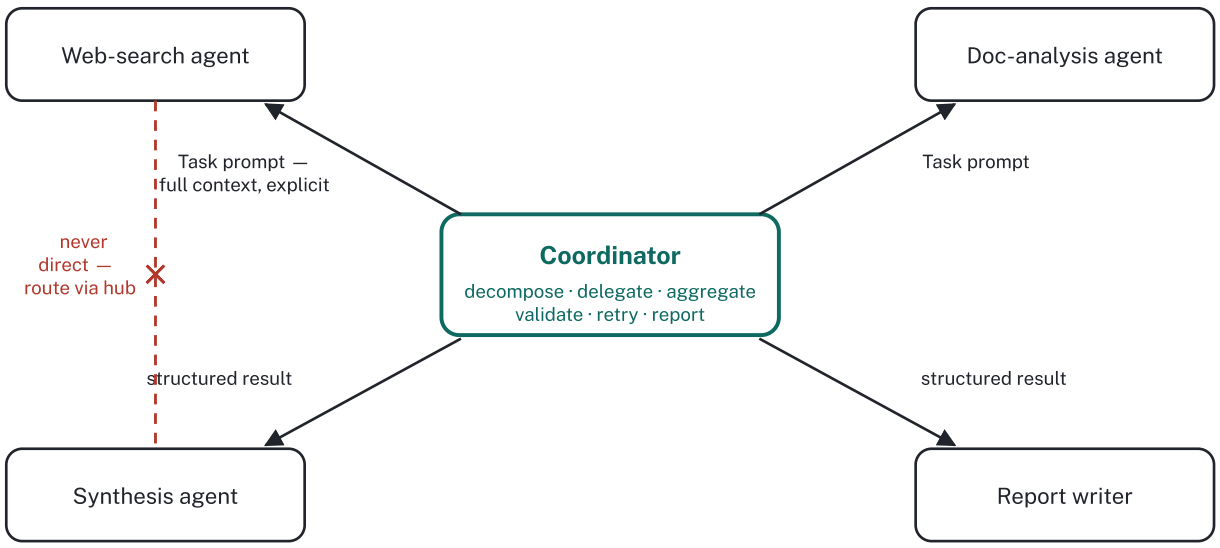
Full `stop_reason` set — `tool_use` vs `end_turn` carry the most weight; the rest appear as distractors:

VALUE	MEANING	CORRECT ACTION
`end_turn`	Model finished naturally	Present result; exit loop
`tool_use`	Model wants tool(s) executed	Run tools, return `tool_result`, loop
`max_tokens`	Hit the `max_tokens` cap	Truncated — raise limit or continue
`stop_sequence`	A custom stop sequence fired	Check the `stop_sequence` field
`pause_turn`	Server-tool loop paused	Send assistant content back to continue
`refusal`	Model declined	Handle/retry per policy

`model_context_window_exceeded`	Output filled the window	Treat as truncated

This is **model-driven** orchestration — Claude picks the next tool from context — unlike a hard-coded decision tree where the sequence is fixed in code.

1.2 · Hub-and-spoke: coordinator and subagents



All communication routes through the coordinator — the rationale: observability, uniform error handling, and control over what each subagent receives. "Analysis sends results straight to synthesis" is always a wrong option.

★ **THE KEY PRINCIPLE IN DOMAIN 1**

Subagents have isolated context. They do not inherit the coordinator's history, and they don't share memory across calls. Every fact a subagent needs — the document text, prior results, the output format — must be explicitly included in its `Task` prompt. `Task: "Analyze the document"` is wrong when the document lives only in the coordinator's history.

Two classic root-cause questions:

- **Narrow reports:** a report on "AI impact on creative industries" covers only visual art; logs show the coordinator decomposed into digital art / graphic design / photography. Every subagent succeeded — the defect is the **coordinator's decomposition was too narrow**, not any subagent.

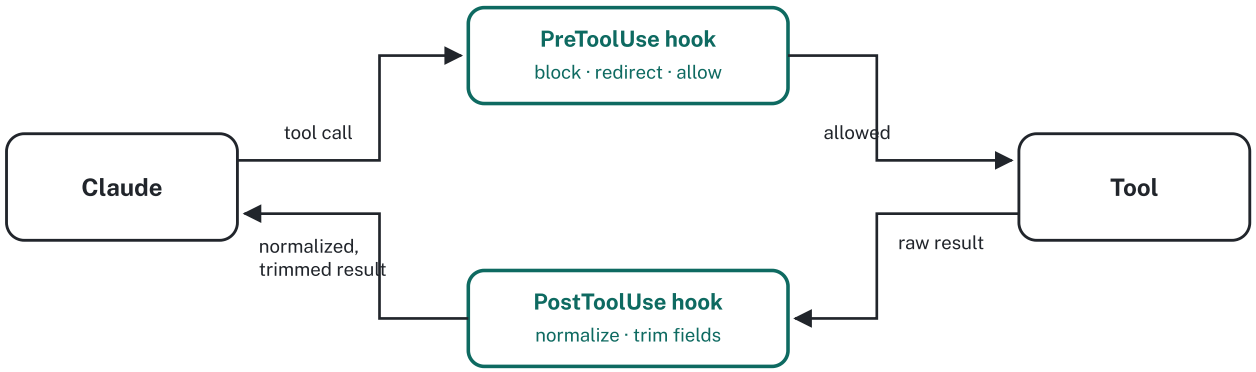
- **Duplicated research:** two subagents cover the same subtopics and tokens double → the coordinator must **explicitly partition the research space before delegating** — not post-hoc dedup, not shared state.

1.3 · Task tool, AgentDefinition, and spawning

- Subagents are spawned via the `Task` tool — the coordinator's `allowed_tools` must include `"Task"` or it cannot delegate at all.
- `AgentDefinition` configures an agent: `name`, `description`, system prompt, `allowed_tools` (least privilege — fewer tools means more reliable selection).
- **Parallel spawning:** multiple `Task` calls in a single response run concurrently — the answer to "run independent research streams in parallel."
- Write coordinator prompts as **goals and quality criteria**, not step-by-step micro-instructions.

1.4 · Hooks vs prompt instructions

The single most important distinction in the domain: code is deterministic, prompts are probabilistic.



PreToolUse gates the *outgoing call* (block a >$500 refund, enforce tool ordering); PostToolUse transforms the *result* (Unix timestamps → ISO 8601, 40 fields → the 5 that matter) — especially for third-party MCP servers you can't modify.

HOOKS — DETERMINISTIC, 100%

- Financial thresholds ("block refunds > $500")
- Compliance and safety requirements
- Mandatory tool sequencing (verify ID before refund)

PROMPTS — PROBABILISTIC, NEVER 100%

- Preferences and tone
- General guidance ("try to resolve before escalating")

- Anything where occasional misses are acceptable

⚖ **DECISION RULE**

If failure has financial, legal, or safety consequences, enforce it **in code** — hooks or programmatic preconditions — never in the prompt. "12% of runs skip `get_customer` before `process_refund`" → programmatic precondition; not a stronger system prompt, not few-shot, not a routing classifier. For confirmation flows, the guaranteed fix is **two-tool token binding**: `preview_X` returns a single-use token that `execute_X` requires.

1.5 · Task decomposition

Fixed pipeline (prompt chaining)

Predefined sequential steps. For predictable, repeatable structure: code-review templates, extract → validate → enrich.

Dynamic adaptive decomposition

Subtasks generated from intermediate findings. For open-ended investigations where scope is unknown up front.

✕ **ATTENTION DILUTION (10–14 FILE PR, SINGLE REVIEW PASS)**

Symptoms: deep comments on some files, shallow on others, contradictions, missed obvious bugs. Fix: **per-file passes for local issues + one integration pass for cross-file flows**. Distractors: "a larger context window can attend to everything" (false), forcing devs to split PRs, majority-voting three full passes.

1.6 · Escalation and human-in-the-loop

VALID TRIGGERS — MEMORIZE AS A SET

- Customer **explicitly asks for a human** → escalate immediately, no "let me try first"
- **Policy gap** — request falls outside written policy; never invent policy
- No progress after reasonable attempts
- Financial op over threshold (hook-enforced)
- Multiple customer matches → ask for another identifier; never guess

UNRELIABLE TRIGGERS — ALWAYS DISTRACTORS

- Sentiment analysis (mood ≠ complexity)
- Model self-rated confidence 1–10 (poorly calibrated — confidently wrong)
- A separately trained escalation classifier (overengineering)

Anger pattern: first frustration → acknowledge, offer a concrete resolution; escalate only if the customer *reiterates* wanting a human. **Structured handoff** must be self-contained (the human never sees the transcript): customer ID, issue summary, order ID, root cause, actions taken, recommended action, escalation reason.

1.7 · Sessions: resume, continue, fork

- `claude --resume` (`-r`) resumes a specific session; `--continue` (`-c`) loads the most recent conversation in the current directory.
- Resume risk: **stale tool results** — if files changed since, prior Read/Grep output is wrong → start a **new session seeded with a short structured summary** instead.
- **Forking** (`--fork-session` with resume/continue) branches a session: both forks share context up to the branch point, then diverge — the tool for comparing two approaches (Redux vs Context API) from one investigation baseline.

Practice bank — 36 questions

36 original, exam-style practice questions with full explanations — written to the published objectives, not taken from any exam.

Work them by domain as checkpoints (the 7-day plan says when), then re-run your misses on day 7. The answer key starts on its own page after the last question — answers follow on the next page.

Agentic Architecture & Orchestration

Q1 EASY

You are implementing the agent loop for a customer support agent built on the Claude API. After each API call you must decide whether to execute requested tools and call Claude again, or stop and present the final answer. What should drive this decision?

A. Parse Claude's text output for completion phrases such as 'Task completed' or 'Is there anything else?'

B. Check the response's stop_reason field: continue the loop on "tool_use", stop on "end_turn"

C. Set max_iterations=10 and stop when the counter is reached, regardless of Claude's response

D. Check whether the response contains any assistant text content; if it does, the task is finished

Q2 EASY

Your agent loop received a response with stop_reason "tool_use" and executed the requested lookup_order call. How must the tool's result be sent back to Claude in the next API request?

A. As a message with role "tool" containing the raw result string

B. As an assistant-role message with a tool_result content block appended to Claude's prior turn

C. As a user-role message whose content includes a tool_result block referencing the tool_use_id

D. In the top-level system field, so the result has priority over the conversation history

Q3 MEDIUM

A coordinator in a multi-agent research system dispatches a document-analysis subagent with the prompt "Analyze the document and extract key statistics." The subagent responds that no document was provided, even though the coordinator discussed the document extensively in its own conversation. What is the root cause?

A. Subagents run with isolated context and do not inherit the coordinator's history, so the document must be included explicitly in the Task prompt

B. The subagent's context window was exceeded by the document, so it was silently dropped

C. The coordinator must first write the document to shared agent memory that all subagents read automatically

D. The Task call is missing a session_id parameter linking the subagent to the coordinator's session

Q4 EASY

In the Claude Agent SDK, you define a coordinator agent that must be able to delegate work to specialized subagents. What does the coordinator's configuration require for delegation to work?

A. A subagents list enumerating the AgentDefinition of every child agent

B. The delegate_to() method enabled via the orchestration: true flag

C. The "Task" tool included in the coordinator's allowed tools list

D. A shared context_pool configured so subagents can read the coordinator's state

Q5 MEDIUM

Your research coordinator must run three independent investigations (web search on topic X, document analysis of report Y, web search on topic Z) and total latency matters. How do you run the subagents concurrently?

A. Set parallel: true in each subagent's AgentDefinition

B. Have the coordinator emit multiple Task tool calls in a single response; those subagents execute in parallel

C. Spawn each subagent from a separate coordinator session and merge results manually

D. Mark the subagents async in the SDK and await them with a gather() call inside the coordinator prompt

Q6 MEDIUM

Compliance requires that your support agent never processes a refund above $500 — any such request must go to a human. Occasional violations carry regulatory penalties. Where should this rule be enforced?

A. In the system prompt, stated as a strict rule with the dollar threshold in bold

B. As few-shot examples demonstrating escalation for refunds of $501, $750, and $1,200

C. In the process_refund tool description, warning the model not to call it above $500

D. In a PreToolUse hook that intercepts process_refund calls and redirects any amount over $500 to escalation

Q7 MEDIUM

Your agent consumes data from several MCP servers, some third-party and unmodifiable: one returns Unix timestamps, another ISO 8601 dates, a third numeric status codes (1=pending, 2=shipped). The agent regularly misinterprets these values. What is the most maintainable fix?

A. A PostToolUse hook that intercepts all tool results and normalizes formats before the model processes them

B. A normalize_data tool the agent is instructed to call after every retrieval

C. Documentation of each tool's format conventions added to the system prompt

D. Wrapper MCP servers around each third-party server that reformat responses

Answer key & explanations

Read every explanation, even for questions you got right — the distractor rationales are training too.

Q1 → B stop_reason is Claude's explicit structured loop-control signal: "tool_use" means execute the tools and return tool_result blocks; "end_turn" means the model finished. Parsing text (A) is a brittle natural-language heuristic; an iteration cap (C) is at most a safety net, never the primary stop condition; and (D) fails because Claude routinely emits explanatory text alongside tool_use blocks in the same response.

Q2 → C Tool results are returned as a tool_result content block inside a user-role message, carrying the tool_use_id from the matching tool_use block. There is no "tool" role in the Messages API (A is a plausible-looking invention borrowed from other APIs). Results are not appended to the assistant turn (B) — the assistant turn contains the tool_use request. The system field (D) is for behavioral instructions, not tool outputs.

Q3 → A Subagents have isolated context: nothing from the coordinator's conversation carries over unless it is explicitly placed in the subagent's prompt (document text, prior results, output requirements). (B) invents a silent-drop behavior — nothing was ever sent. (C) describes shared memory that does not exist between subagents. (D) invents a session_id linkage parameter; no such mechanism passes context to Task subagents.

Q4 → C Subagents are spawned via the Task tool, so the coordinator's allowedTools must include "Task" — without it the coordinator cannot delegate at all. (A) and (B) are fabricated configuration surfaces, and (D) contradicts the isolation model: subagents never read coordinator state; context is passed explicitly in each Task prompt.

Q5 → B Parallelism is achieved by the coordinator issuing multiple Task calls in one response — the runtime executes them concurrently. There is no parallel flag on AgentDefinition (A) and no async/gather construct expressible inside a prompt (D). Separate coordinator sessions (C) sacrifice the single point of aggregation, observability, and error handling that the hub-and-spoke pattern exists to provide.

Q6 → D Prompt-based approaches (A, B, C) are probabilistic — high compliance but never 100% — which is unacceptable when failure has financial or regulatory consequences. A PreToolUse hook intercepts the outgoing tool call in code and blocks or redirects it deterministically, guaranteeing the threshold is enforced regardless of what the model decides. The rule: money, legal, or safety constraints belong in hooks, not prompts.

Q7 → A A PostToolUse hook is a single deterministic interception point that transforms every tool result — including output from third-party servers you cannot modify — before the model sees it. A normalize_data tool (B) relies on the model remembering to call it every time (probabilistic). Prompt documentation (C) relies on the model performing conversions correctly on every turn. Per-server wrappers (D) work but multiply maintenance across servers instead of centralizing the logic.

Ready for the rest?

You've just read about 30% of the full guide — at full depth, nothing dumbed down.

The full Unofficial CCAR-F Study Guide includes

- Every domain at the same depth as Domain 1 you just read — diagrams, decision tables, trap pairs, and cue boxes throughout
- The complete 7-day plan: 7 days of guided study, hands-on drills, and checkpoints
- All 36 original practice questions with full answer-key explanations — including the distractor rationales
- The quick-reference sheets and night-before cram material

Get the full edition at `zehon.com`