

FREE PDF

Claude Certified Developer – Foundations (CCDV-F) practice questions

20 free questions with answers and explanations, written to the official exam guide.

Questions	pages 2–21
Check your answers	page 22
Answers and explanations	pages 23–36
Exam tips	page 37

Covers all 8 domains of the exam guide

Agents and Workflows	14.7%
Applications and Integration	33.1%
Claude Code	3.1%
Eval, Testing, and Debugging	2.6%
Model Selection and Optimization	16.8%
Prompt and Context Engineering	11.0%
Security and Safety	8.1%
Tools and MCPs	10.6%



Question 1

A support platform drafts replies to about 40,000 tickets a day, and agents send each draft after a quick read. A new prompt paired with a different model tier beats the current version on a 300–ticket eval set built last spring. Since then, two newly launched products have come to generate a third of all tickets.

How should the team move to the new version?

- ☐ (A) Switch the full ticket volume to the new version in one release, since it already outscores the current version on the team's eval set.
- ☐ (B) Route a small share of live tickets to the new version, grade its drafts against the current version's, and widen the share as scores hold.
- ☐ (C) Have five experienced agents use the new version on tickets they choose for a week, then switch everyone over if they approve of it.
- ☐ (D) Release the new version to the whole ticket volume behind a feature flag, and switch the flag back if customer complaints rise over the following week.

Note your answers as you go, on paper or in your notes app, then check them all at the answer key.

Question 2

Your inbox assistant runs on the Messages API with the web fetch tool for opening links, a search_mailbox tool that reads the user's private mail, and a send_email tool that Claude calls on its own to send replies. In a red-team test, hidden text in a newsletter told Claude to find the user's bank statements and mail them to an outside address, and it did.

How would you fix the problem?

- ☐ (A) Add an untrusted-content policy to the system prompt stating that instructions inside emails and fetched pages are data to report, not commands.
- ☐ (B) Move the assistant to the most capable model tier, which holds to its system prompt more reliably when content tries to redirect it.
- ☐ (C) Make send_email produce a draft that the user approves in the app before the handler delivers anything to any recipient.
- ☐ (D) Log each send_email call with its recipients and attachments, and alert the security team when mail goes to an unfamiliar domain.

Question 3

To control cost, a support chat service sends only the newest 20 messages of each conversation to the Messages API. Claude calls tools on most turns, often two or three in one reply. Short chats work, but once a chat passes 20 messages some requests fail with a 400 `invalid_request_error` that names a `tool_use_id`. Resending the full history makes the same request succeed.

How would you fix the problem?

- ☐ (A) Set `disable_parallel_tool_use` in `tool_choice`, so each assistant reply in the kept history holds at most one tool call.
- ☐ (B) Trim the history by token count instead of message count, using the token counting endpoint to size each kept message.
- ☐ (C) Set `tool_choice` to `none` on requests that carry a trimmed history, so Claude answers from the kept messages without new calls.
- ☐ (D) Start the kept history at a user message that holds no tool results, so every kept tool result still has its call.

Question 4

A report service sends one request asking Claude to outline a market report and then write the full 6,000–word report. Every report must contain the five sections compliance requires, and later sections build on findings in earlier ones. About one report in ten misses a required section, which the team discovers after paying for the full draft.

Which approach would you take?

- ☐ (A) Hand each report to an agent with a section-checking tool and let it decide for itself when the report is complete.
- ☐ (B) Split the work into an outline call and a drafting call, with code that checks the outline for the five sections in between.
- ☐ (C) Write each of the five required sections in its own parallel call, then join the sections into the finished report.
- ☐ (D) Add a final validation call that reads each finished report and sends any report missing a section back for regeneration.

Question 5

Model Selection and Optimization

A translation service quotes customers a price before sending each document to Claude, estimating input tokens as characters divided by four. Quotes for English contracts match the invoice closely, but quotes for Japanese manuals are far too low, and many manuals mix Japanese prose with English part numbers and tables. Customers pay the quoted price, so quotes must stay close to the actual charge.

How should the developer estimate tokens for each quote?

- ☐ (A) Keep a separate characters-per-token ratio for each language, calibrated from the usage figures of past documents in that language
- ☐ (B) Send each document to the Messages API first, read the input token count from the response usage, and quote from that figure
- ☐ (C) Count the words in each document and multiply by a fixed tokens-per-word ratio measured on a sample of past documents
- ☐ (D) Send each document to the token counting endpoint with the model the service uses, and quote from the input token count returned

Question 6

A developer's pipeline sends each supplier contract to the Messages API with an output limit of 2,000 tokens and parses Claude's reply as a JSON array of clauses. Contracts under 20 pages parse cleanly. Longer contracts fail with an unterminated-string error, and the logged reply text stops partway through a clause.

How would you fix the problem?

- ☐ (A) Retry each failed contract with exponential backoff, and pass the reply to the parser once a retry returns text that parses.
- ☐ (B) Add a system prompt instruction telling Claude to keep each clause entry brief and to close each JSON object it opens.
- ☐ (C) Check the stop_reason of each reply before parsing, and on a max_tokens stop raise the limit or split the contract into sections.
- ☐ (D) Enable structured outputs with a JSON schema for the clause array, so each reply Claude returns is constrained to that schema's shape.

Question 7

A contract-review feature sends one user message: output instructions and five questions first, then a 100-page services agreement inside document tags. The agreement fits well within the context window. On 10-page agreements the answers are accurate; on this one they miss relevant clauses and often ignore the output instructions. The clauses a question depends on can sit anywhere in the agreement.

How would you restructure the request?

- ☐ (A) Split the agreement into its sections, send each section with the five questions in its own request, and merge the answers into one reply.
- ☐ (B) Place each of the five questions directly after the section of the agreement that it is most likely to concern, within the message.
- ☐ (C) Put the agreement first in the user message inside document tags, followed by the output instructions and then the five questions.
- ☐ (D) Move the output instructions and the five questions into the system prompt, leaving only the agreement in the user message.

Question 8

A summarization service runs the same code in two regions: US traffic goes to the Claude API, and EU traffic goes to Amazon Bedrock under a data rule. The model ID is a constant in the code, and the EU deployment fails with an invalid model identifier error. After next quarter's eval, the operations team, which changes configuration but never code, will move summaries to another model tier.

Which approach would you take?

- ☐ (A) Add a platform check to the summarization code that passes either the Bedrock model ID or the Claude API model ID written beside it.
- ☐ (B) Set one model ID in configuration for both regions, written in the form that the Claude API uses for that model.
- ☐ (C) Point both regions at a moving model alias, so that each platform picks up the newest model with no ID change.
- ☐ (D) Keep a logical model name for each feature in configuration, with a table per platform giving the ID each region sends.

Question 9

An SRE team's MCP server, used from Claude Code and Claude Desktop, exposes alert-query tools and runbook resources. The team wrote an incident-summary template: instructions with an incident ID placeholder that tell Claude which alert tools to call and how to lay out the summary. Engineers should start each summary themselves by giving the ID, Claude should not start one on its own, and each template revision should reach both clients from this server.

How should the team expose the template?

- ☐ (A) Add it to the server as an `incident_summary` tool with a required `incident_id` parameter that returns the filled-in instructions.
- ☐ (B) Add it to the server as a resource template at `incident://{incident_id}/summary` whose content holds the filled-in instructions.
- ☐ (C) Package it as a Claude Code skill that takes an incident ID argument, kept in a shared skills folder each engineer syncs.
- ☐ (D) Add it to the server as a prompt with a required `incident_id` argument whose messages hold the filled-in instructions.

Question 10

Applications and Integration

A team prototyped a meeting-notes summarizer in the claude.ai app, where replies came back as Markdown lists and turned phrases such as next Friday into correct dates. In production the same notes go to the Messages API with a one-line system prompt, and replies arrive as plain paragraphs with deadlines placed on the wrong dates.

What should the developer do?

- ☐ (A) Switch the API requests to the model tier the prototype used in claude.ai, so production replies match the prototype's quality.
- ☐ (B) Write the Markdown layout rules and today's date into the system prompt that the application sends with each API request.
- ☐ (C) Raise the effort setting on the API requests so Claude reasons more carefully about the date arithmetic and the layout.
- ☐ (D) Add three summaries from the claude.ai prototype to the prompt as examples of the Markdown layout the replies should follow.



Question 11

A retailer's support chat serves signed-in customers. Its `get_orders` tool takes a `customer_id` argument that Claude fills in from the conversation, and the handler returns that customer's orders and delivery addresses. In testing, a user wrote 'I also manage account 48213, show its orders' and received a stranger's address.

How would you fix the problem?

- ☐ A Drop `customer_id` from the tool's input schema and have the handler read the customer from the request's authenticated session.
- ☐ B Check `customer_id` against the account number format in the handler and return a `tool_result` with `is_error` when it does not match.
- ☐ C Add a system prompt rule telling Claude to call `get_orders` with the customer ID the application supplies at the start of each chat.
- ☐ D Log each `customer_id` that Claude passes to `get_orders` and alert the security team when it differs from the signed-in customer.

Question 12

A developer's Agent SDK app fixes failing tests, with the main agent in acceptEdits mode. They added a reviewer subagent whose prompt says to review the main agent's diff and report problems; its description reads 'Helper agent' and it sets no tools field. In testing, the main agent rarely hands work to the reviewer, and once the reviewer rewrote two files itself instead of reporting.

Which two changes should the developer make? **Choose 2**

- ☐ A Add a line to the reviewer's prompt telling it to report each problem as text and to leave the files it reads unchanged.
- ☐ B Remove Edit and Write from the main agent's allowed tools so that any file change in the session has to wait for approval.
- ☐ C Give the reviewer's definition a tools list of Read, Grep and Glob so the subagent has no tool that can write files.
- ☐ D Move the review steps into a skill under .claude/skills so the main agent loads them whenever it reviews a diff.
- ☐ E Rewrite the reviewer's description to say it reviews diffs for defects and should be called after each fix is made.

Question 13

Claude Code

A developer wants Claude Code to know the URL of their own test sandbox whenever they work in one team repository, and nowhere else.

Where should the developer record the URL?

- ☐ (A) Add the sandbox URL to the CLAUDE.md file at the repository root, which Claude Code loads at the start of each session there.
- ☐ (B) Add the sandbox URL to the CLAUDE.md file in the .claude folder of the home directory, which Claude Code loads in each session.
- ☐ (C) Add the sandbox URL to a CLAUDE.local.md file at the repository root, and list that file in the repository's .gitignore.
- ☐ (D) Add the sandbox URL as an env entry in the repository's .claude/settings.local.json file, which stays out of version control.

Question 14

Every 15 minutes, a job sends about 300 new support tickets to Claude for sentiment tagging, and the operations dashboard must show them within 10 minutes of each run. A teammate proposes the Message Batches API for its lower cost, noting that test batches of this size finished in under five minutes.

Which approach would you take?

- ☐ (A) Send each run's tickets as concurrent Messages API requests, so the tags return while that run's job is still working.
- ☐ (B) Submit each run as a message batch, since test batches of this size came back well inside the 10-minute window.
- ☐ (C) Run the tagging once a night as one large message batch, and show the previous day's sentiment on the dashboard.
- ☐ (D) Submit each run as a message batch with a cache breakpoint after the shared instructions, so each batch processes faster.

Question 15

Model Selection and Optimization

A law firm's tool reviews about 20 commercial contracts a day and lists each indemnity, liability cap and termination clause for a lawyer to act on. A clause it missed in one signed deal cost the firm more than a year of the tool's model spend. A partner proposes the smallest model to cut costs. The firm holds 300 past contracts whose clauses lawyers have already listed.

Which approach would you take?

- (A) Run the reviews on the smallest model, and have a lawyer spot-check a random sample of its clause lists each week.
- (B) Test each model on the lawyer-reviewed contracts, and deploy the cheapest model that misses only a few more clauses than the best.
- (C) Test each model on the lawyer-reviewed contracts, and deploy the one missing the fewest clauses, likely the most capable.
- (D) Pick the model that leads public reasoning benchmarks for contract work, and move to each newer release as it launches.



Question 16

A developer's custom loop on the Messages API runs each reply's `tool_use` blocks, but it treats any reply containing a text block as the final answer and returns that text. Users of the order assistant now often get only 'Let me look up that order for you.' The logged replies are complete and also hold a `lookup_order` `tool_use` block.

How would you fix the loop?

- ☐ (A) Decide on `stop_reason` instead: run the requested tools and call again while it is `tool_use`, and finish on any other value.
- ☐ (B) Raise `max_tokens` on each request so Claude has room to finish the lookup and write the full answer in one reply.
- ☐ (C) Set `tool_choice` to any on each request so Claude must call a tool in its turn instead of writing a text reply.
- ☐ (D) Add a system prompt rule telling Claude to write no text in a turn where it calls a tool, so each reply holds just the tool calls.

Question 17

A dashboard streams Claude's replies so users watch the explanation appear, and Claude then calls a `create_chart` tool in the same reply. To draw the chart early, the front end runs `JSON.parse` on each `input_json_delta` of the tool call. The console fills with parse errors during tool calls, although the tool input in the final message is always valid.

How would you fix the problem?

- ☐ A Turn off streaming for requests that include `create_chart`, and read the tool input from the complete response object instead.
- ☐ B Append each `input_json_delta` to a buffer for that content block, and parse the buffer once its `content_block_stop` arrives.
- ☐ C Set `strict: true` on `create_chart`, so that every `input_json_delta` the stream delivers is valid JSON on its own.
- ☐ D Wrap `JSON.parse` in a try block that skips any `input_json_delta` that fails, and draw from the deltas that parse.

Question 18

An agent books meeting rooms overnight from employees' emailed requests. Its `book_room` tool has a string date parameter described only as the meeting date, and the booking service accepts whatever arrives through a lenient date parser. Logs show inputs such as next Friday, the 3rd and Nov 3, and some bookings land on the wrong day with no error.

Which change would you make?

- ☐ (A) Describe date as an ISO 8601 calendar date with an example, and have the service reject other input with an `is_error` tool_result.
- ☐ (B) Replace the lenient parser with a natural-language date library that resolves relative phrases such as next Friday entirely on its own.
- ☐ (C) Add the booking service's date rules and several example requests to the system prompt, so Claude writes dates the same way.
- ☐ (D) Set `strict: true` on `book_room` without changing its schema, so that every date the agent sends is guaranteed to be valid.

Question 19

A clinic's appointment assistant sends Claude's replies to patients by SMS, which shows raw characters. The system prompt says: Do not use markdown. Do not use bullet points. About one reply in five still arrives with asterisks and dashed lists. The product owner wants replies that read as natural sentences, and a test that stripped the symbols left clipped list fragments.

What should the developer do?

- ☐ (A) Extend the prohibitions to name asterisks, dashes, pound signs and numbered lists as characters the reply must not contain.
- ☐ (B) Strip markdown symbols from each reply in code before the SMS is sent, using a markdown-to-plain-text converter library.
- ☐ (C) Return replies through structured outputs with a schema of one plain string field, and send that field's value as the SMS.
- ☐ (D) Replace the prohibitions with the target style: short paragraphs of complete sentences, as a patient reads them in an SMS.

Question 20

Your assistant's system prompt opens with a line holding the current time to the second, followed by a 9,000-token policy manual, with a `cache_control` breakpoint on the last system block. Requests arrive every few seconds. In every response, `cache_creation_input_tokens` shows about 9,000 and `cache_read_input_tokens` shows zero, and the bill has risen since caching was enabled.

What should the developer do?

- ☐ (A) Switch the cache entry to the one-hour lifetime so the cached system prompt survives the gaps between consecutive requests from users.
- ☐ (B) Move the current-time line out of the system prompt's opening and send it after the breakpoint, in the user turn of each request.
- ☐ (C) Add a second cache breakpoint directly after the current-time line so the policy manual that follows is cached as its own entry.
- ☐ (D) Expand the cached system prompt with more policy text so its length reaches the model's minimum cacheable size and starts producing reads.

Check your answers

Tick each one you got right. Explanations start on the next page.

Count a multiple-response question as right only if you picked every correct letter and nothing else.

☐ Q1	B	☐ Q2	C	☐ Q3	D	☐ Q4	B	☐ Q5	D
☐ Q6	C	☐ Q7	C	☐ Q8	D	☐ Q9	D	☐ Q10	B
☐ Q11	A	☐ Q12	C, E	☐ Q13	C	☐ Q14	A	☐ Q15	C
☐ Q16	A	☐ Q17	B	☐ Q18	A	☐ Q19	D	☐ Q20	B

Your score ____ / 20

By domain

Agents and Workflows	Q4, Q12, Q16	____ / 3
Applications and Integration	Q1, Q6, Q8, Q10, Q14, Q17	____ / 6
Claude Code	Q13	____ / 1
Eval, Testing, and Debugging	Q3	____ / 1
Model Selection and Optimization	Q5, Q15, Q20	____ / 3
Prompt and Context Engineering	Q7, Q19	____ / 2
Security and Safety	Q2, Q11	____ / 2
Tools and MCPs	Q9, Q18	____ / 2

Question 1

- B** Route a small share of live tickets to the new version, grade its drafts against the current version's, and widen the share as scores hold.

Why B is right

An offline eval predicts quality only for the kinds of input it contains. This one was built before the two new products launched, so a third of today's tickets are untested. A staged rollout measures the new version on the live mix, with both versions graded side by side. Only a small share of tickets gets a weaker draft while the evidence comes in.

Why the other options are wrong

- A** Switch the full ticket volume to the new version in one release, since it already outscores the current version on the team's eval set.
A third of today's tickets are not covered by the old eval, and every customer would get the new drafts at once.
- C** Have five experienced agents use the new version on tickets they choose for a week, then switch everyone over if they approve of it.
Five agents choosing their own tickets is a small, self-selected sample with no grading against the current version.
- D** Release the new version to the whole ticket volume behind a feature flag, and switch the flag back if customer complaints rise over the following week.
Complaints arrive late and only from customers who notice, so a week of weaker drafts reaches customers before the flag flips back.

Exam tip: Before you swap a prompt or model in production, check that your eval set still matches today's traffic.

Source: platform.claude.com/docs/en/test-and-evaluate/develop-tests

Question 2

- C** Make `send_email` produce a draft that the user approves in the app before the handler delivers anything to any recipient.

Why C is right

Private data, untrusted content and a way to send data out, together, let any email sender steal the user's mail. With an approval gate in application code, an injected instruction can still produce a draft, but nothing leaves until the user sees the recipient and the attachment. Prompt rules and stronger models lower the odds of an attack working. One success is still enough to send the statements out.

Why the other options are wrong

- A** Add an untrusted-content policy to the system prompt stating that instructions inside emails and fetched pages are data to report, not commands.
The policy is worth having, but it only asks. The newsletter already persuaded Claude once, and `send_email` still mails whatever Claude decides.

Question 2, continued

- (B) Move the assistant to the most capable model tier, which holds to its system prompt more reliably when content tries to redirect it.
Stronger models resist injection better, yet no model makes it impossible, and one success is enough.
- (D) Log each send_email call with its recipients and attachments, and alert the security team when mail goes to an unfamiliar domain.
The alert fires after the statements have left the mailbox. Detection does not stop the theft.

Exam tip: When an agent reads private data and untrusted content, list every tool that can send data out, then remove it or put a user confirmation in front of it.

Source: platform.claude.com/docs/en/test-and-evaluate/strengthen-guardrails/mitigate...

Question 3

- (D) Start the kept history at a user message that holds no tool results, so every kept tool result still has its call.

Why D is right

The API rejects a tool_result whose tool_use is missing, with a 400 that names the tool_use_id. A fixed cut at 20 messages can fall between an assistant message with tool calls and the user message holding their results. Moving the cut to a user message with no tool results keeps every tool_use and tool_result pair whole. If no such message falls inside the newest 20, as in a long tool loop, cut further back or summarise the older turns.

Why the other options are wrong

- (A) Set disable_parallel_tool_use in tool_choice, so each assistant reply in the kept history holds at most one tool call.
This limits how many calls each new reply makes, but the cut can still fall between a call and its result.
- (B) Trim the history by token count instead of message count, using the token counting endpoint to size each kept message.
Measuring in tokens moves where the cut lands but still ignores tool pairs, so the same 400 returns on other chats.
- (C) Set tool_choice to none on requests that carry a trimmed history, so Claude answers from the kept messages without new calls.
This only stops new tool calls. The API still checks the history it receives, so an orphaned tool_result is rejected all the same.

Exam tip: When you shorten a tool-using conversation, choose the cut by turn structure so no tool_result loses its tool_use.

Source: platform.claude.com/docs/en/agents-and-tools/tool-use/handle-tool-calls

Question 4

- B** Split the work into an outline call and a drafting call, with code that checks the outline for the five sections in between.

Why B is right

The work has a fixed path: an outline, then a full draft that builds on it. Prompt chaining splits the job so each call does one easier task. A programmatic gate between the two calls checks the outline for the five required sections in code. A failing outline is retried at the price of a short call, so the 6,000-word draft is only written from an outline known to be complete.

Why the other options are wrong

- A** Hand each report to an agent with a section-checking tool and let it decide for itself when the report is complete.
The path is known in advance, so an autonomous agent adds unpredictable turns and cost, and its check still comes during or after the long draft.
- C** Write each of the five required sections in its own parallel call, then join the sections into the finished report.
Later sections build on findings in earlier ones, so independent parallel calls cannot use each other's analysis and the joined report does not hang together.
- D** Add a final validation call that reads each finished report and sends any report missing a section back for regeneration.
The check runs on the finished report, so every failure has already paid for a full draft and then pays again to regenerate it.

Exam tip: Put the check at the earliest step where the defect can show, before the costly call runs.

Source: anthropic.com/engineering/building-effective-agents

Question 5

- D** Send each document to the token counting endpoint with the model the service uses, and quote from the input token count returned

Why D is right

Characters per token vary with the model and the language. A divisor tuned on English undercounts Japanese, and these manuals mix the two. The token counting endpoint counts the document for the model you pass and returns an estimate that closely matches the billed input. Each quote then follows that document's own mix of prose, part numbers and tables.

Why the other options are wrong

- A** Keep a separate characters-per-token ratio for each language, calibrated from the usage figures of past documents in that language
A per-language ratio is an average, and these manuals mix Japanese prose with English part numbers and tables in varying shares.

Question 5, continued

- (B) Send each document to the Messages API first, read the input token count from the response usage, and quote from that figure
That call performs the translation and is billed before the customer sees a price, so every declined quote still costs a full job.
- (C) Count the words in each document and multiply by a fixed tokens-per-word ratio measured on a sample of past documents
Japanese is written without spaces between words, so a word count says little about the Japanese prose.

Exam tip: Base prices and budgets on counts from the token counting endpoint for the model you will call, and recount when you change models.

Source: platform.claude.com/docs/en/build-with-claude/token-counting

Question 6

- (C) Check the stop_reason of each reply before parsing, and on a max_tokens stop raise the limit or split the contract into sections.

Why C is right

Claude stopped because it ran out of output tokens, and the response says so with a stop_reason of max_tokens. Checking stop_reason before parsing turns truncation into its own path instead of a parser error. A higher limit, or smaller sections, gives the long contracts room to finish. Size the limit, or the chunk, to the longest document the job will see.

Why the other options are wrong

- (A) Retry each failed contract with exponential backoff, and pass the reply to the parser once a retry returns text that parses.
The same contract with the same limit produces the same cut-off reply, so each retry fails the same way and only adds cost. Backoff is for transient errors.
- (B) Add a system prompt instruction telling Claude to keep each clause entry brief and to close each JSON object it opens.
Shorter entries move the cut-off later but do not remove it, and Claude cannot close objects in text it never got to write.
- (D) Enable structured outputs with a JSON schema for the clause array, so each reply Claude returns is constrained to that schema's shape.
Structured outputs constrain the format, but a reply that hits the token limit is still cut off and may not match the schema.

Exam tip: Branch on stop_reason before you parse a reply: max_tokens means the output is incomplete.

Sources: platform.claude.com/docs/en/build-with-claude/handling-stop-reasons; platform.claude.com/docs/en/build-with-claude/structured-outputs

Question 7

- C** Put the agreement first in the user message inside document tags, followed by the output instructions and then the five questions.

Why C is right

Anthropic's long-context guidance puts long documents near the top of the prompt and the query after them. It reports that queries at the end can raise response quality markedly on complex inputs. The symptoms fit: short agreements work, while the long one loses clauses and the output instructions. Reordering to agreement, then instructions, then questions fixes this without splitting the document.

Why the other options are wrong

- A** Split the agreement into its sections, send each section with the five questions in its own request, and merge the answers into one reply.
The agreement already fits in one request, and splitting it cuts links between sections, such as an early definition that changes a later clause.
- B** Place each of the five questions directly after the section of the agreement that it is most likely to concern, within the message.
Questions scattered through the text still come before most of it, and tying each to one guessed section points Claude away from the rest.
- D** Move the output instructions and the five questions into the system prompt, leaving only the agreement in the user message.
The system prompt is read before the user message, so the instructions and questions still come before the hundred pages.

Exam tip: For long documents, order the prompt as document first, then instructions, then the question.

Source: platform.claude.com/docs/en/build-with-claude/prompt-engineering/claude-prom...

Question 8

- D** Keep a logical model name for each feature in configuration, with a table per platform giving the ID each region sends.

Why D is right

Bedrock expects its own model ID format, so the Claude API string fails in the EU region. Configuration can hold a logical name for the summary feature, with a table giving the ID each platform sends. The code asks only for the logical name. Next quarter's tier change then becomes a configuration edit that the operations team can make without a code release.

Why the other options are wrong

- A** Add a platform check to the summarization code that passes either the Bedrock model ID or the Claude API model ID written beside it.
This fixes the EU error, but both IDs stay in the code, so the operations team cannot change tier without a code release.

Question 8, continued

- (B) Set one model ID in configuration for both regions, written in the form that the Claude API uses for that model.
Bedrock expects its own ID format, so the EU requests keep failing with the same invalid model identifier error.
- (C) Point both regions at a moving model alias, so that each platform picks up the newest model with no ID change.
A model change should follow the team's eval, not happen on its own, and Bedrock has no such alias.

Exam tip: When one service reaches Claude through more than one platform, keep platform model IDs in configuration, never in code.

Sources: platform.claude.com/docs/en/build-with-claude/claude-in-amazon-bedrock;
platform.claude.com/docs/en/models/overview

Question 9

- (D) Add it to the server as a prompt with a required `incident_id` argument whose messages hold the filled-in instructions.

Why D is right

MCP splits control by primitive: the model invokes tools, the application attaches resources, and the user chooses prompts. A prompt with a required `incident_id` argument fits a summary that an engineer starts by giving the ID. Claude Code lists it as a slash command, and the filled-in messages become the request Claude carries out with the alert tools. Because the template lives on the server, each revision reaches both clients.

Why the other options are wrong

- (A) Add it to the server as an `incident_summary` tool with a required `incident_id` parameter that returns the filled-in instructions.
Tools are model-controlled, so Claude can start a summary by itself whenever alerts come up.
- (B) Add it to the server as a resource template at `incident://{incident_id}/summary` whose content holds the filled-in instructions.
A resource is context the application attaches, not a request Claude carries out, and in Claude Code, Claude also gets tools to read resources unasked.
- (C) Package it as a Claude Code skill that takes an incident ID argument, kept in a shared skills folder each engineer syncs.
The skill lives in engineers' folders, not on the MCP server, so revisions do not reach both clients from it, and Claude can load a skill by itself.

Exam tip: Remember the MCP control model: tools for the model, resources for the application, prompts for the user.

Sources: modelcontextprotocol.io/specification/latest/server/prompts; code.claude.com/docs/en/mcp

Question 10

- B** Write the Markdown layout rules and today's date into the system prompt that the application sends with each API request.

Why B is right

The claude.ai app adds a system prompt to each conversation. It supplies the current date and encourages formatting habits such as Markdown. That prompt does not apply to the Claude API, where Claude sees only what the request carries. Production gets the prototype's behaviour back once the application's own system prompt states the layout rules and inserts the date on each call.

Why the other options are wrong

- A** Switch the API requests to the model tier the prototype used in claude.ai, so production replies match the prototype's quality.
A different tier still gets the same one-line system prompt through the API, so it has no date to resolve next Friday against and no instruction to use Markdown lists.
- C** Raise the effort setting on the API requests so Claude reasons more carefully about the date arithmetic and the layout.
More effort buys more reasoning, but Claude cannot work out a deadline from a current date the request never gives it.
- D** Add three summaries from the claude.ai prototype to the prompt as examples of the Markdown layout the replies should follow.
Examples can teach the list layout, but they carry the dates of past meetings, so relative deadlines in new notes still have no current date.

Exam tip: When moving from claude.ai to the API, list what the app was supplying quietly, such as the date and formatting habits, and put each item in your own system prompt.

Source: platform.claude.com/docs/en/release-notes/system-prompts

Question 11

- A** Drop customer_id from the tool's input schema and have the handler read the customer from the request's authenticated session.

Why A is right

Identity and authorisation belong to the application, not to arguments the model writes. The handler already knows who signed in, so it should read the customer from the session. This is least privilege applied to tool inputs: if Claude cannot choose the account, neither a user's request nor an injected instruction can widen what the tool returns. Claude still decides when to call the tool, but no longer whose data comes back.

Why the other options are wrong

- B** Check customer_id against the account number format in the handler and return a tool_result with is_error when it does not match.
Account 48213 is a valid number, so the format check passes it and the stranger's address is still returned.

Question 11, continued

- C** Add a system prompt rule telling Claude to call `get_orders` with the customer ID the application supplies at the start of each chat.
The rule only asks Claude to behave, and the handler still trusts whatever ID arrives.
- D** Log each `customer_id` that Claude passes to `get_orders` and alert the security team when it differs from the signed-in customer.
The alert fires after the address has been shown. It helps an investigation but does not stop the disclosure.

Exam tip: For a tool that acts for the signed-in user, keep identity fields out of its schema and resolve the user on the server.

Source: platform.claude.com/docs/en/test-and-evaluate/strengthen-guardrails/mitigate...

Question 12

- C** Give the reviewer's definition a tools list of Read, Grep and Glob so the subagent has no tool that can write files.
- E** Rewrite the reviewer's description to say it reviews diffs for defects and should be called after each fix is made.

Why C and E are right

Claude decides when to hand work to a subagent from its description, and 'Helper agent' says nothing about when a review is due. A description saying it reviews diffs for defects after each fix lets Claude match the task to it. With no tools field, a subagent inherits the tools available to subagents and runs in the parent's `acceptEdits` mode, so its edits were approved automatically. A tools list of Read, Grep and Glob leaves it no tool that can write files.

Why the other options are wrong

- A** Add a line to the reviewer's prompt telling it to report each problem as text and to leave the files it reads unchanged.
The reviewer would still hold the edit tools under `acceptEdits`, and a prompt line only asks it not to use them.
- B** Remove Edit and Write from the main agent's allowed tools so that any file change in the session has to wait for approval.
In `acceptEdits` mode the mode approves file edits before allow rules are checked, so this changes nothing, and the main agent still needs to edit to fix tests.
- D** Move the review steps into a skill under `.claude/skills` so the main agent loads them whenever it reviews a diff.
The review would then run inside the main agent with its full tool set, so the separate read-only reviewer is lost and edits stay possible.

Exam tip: Give each subagent an explicit tools list matched to its job and a description that says when to use it.

Sources: code.claude.com/docs/en/agent-sdk/subagents; code.claude.com/docs/en/agent-sdk/permissions

Question 13

- C** Add the sandbox URL to a `CLAUDE.local.md` file at the repository root, and list that file in the repository's `.gitignore`.

Why C is right

Claude Code loads `CLAUDE.local.md` alongside the project `CLAUDE.md` in every session started in that repository. Listed in `.gitignore`, it is never committed, so teammates never receive the note. That fits both facts in the question: the URL belongs to one developer, and it matters in one repository. Keep the committed `CLAUDE.md` for what the whole team should share.

Why the other options are wrong

- A** Add the sandbox URL to the `CLAUDE.md` file at the repository root, which Claude Code loads at the start of each session there.
The project `CLAUDE.md` is committed and shared, so every teammate's Claude would be told about one person's sandbox.
- B** Add the sandbox URL to the `CLAUDE.md` file in the `.claude` folder of the home directory, which Claude Code loads in each session.
The user-level `CLAUDE.md` loads in every project on the machine, so the URL would follow the developer into unrelated repositories.
- D** Add the sandbox URL as an env entry in the repository's `.claude/settings.local.json` file, which stays out of version control.
Settings files configure Claude Code itself, such as permissions and environment variables. They are not read as instructions, so Claude is never told the sandbox exists.

Exam tip: Match the memory file to who should see it: user `CLAUDE.md` for you everywhere, project `CLAUDE.md` for the team, `CLAUDE.local.md` for you in one project.

Source: code.claude.com/docs/en/memory

Question 14

- A** Send each run's tickets as concurrent Messages API requests, so the tags return while that run's job is still working.

Why A is right

The dashboard must show each run's tags within 10 minutes. A synchronous Messages API call returns its answer as soon as it is generated. A message batch costs less, but it is asynchronous and offers no completion time shorter than its processing window, and requests still unprocessed when that window closes expire. Batch processing slows with demand, so a batch cannot back a 10-minute service level however fast the test batches were.

Why the other options are wrong

- B** Submit each run as a message batch, since test batches of this size came back well inside the 10-minute window.
Test runs can be fast, but batch processing slows with demand and nothing shorter than the processing window is guaranteed, so some runs will miss the deadline.

Question 14, continued

- (C) Run the tagging once a night as one large message batch, and show the previous day's sentiment on the dashboard.
This is the cheapest option, but day-old tags break the need to show each run's tickets within 10 minutes.
- (D) Submit each run as a message batch with a cache breakpoint after the shared instructions, so each batch processes faster.
Caching lowers input cost. It does not change when the batch is scheduled or when it completes.

Exam tip: Use batches only for work that can wait the whole processing window. A fast test batch says nothing about the next one.

Source: platform.claude.com/docs/en/build-with-claude/batch-processing

Question 15

- (C) Test each model on the lawyer-reviewed contracts, and deploy the one missing the fewest clauses, likely the most capable.

Why C is right

Anthropic's model-selection guidance lists applications where accuracy outweighs cost among the cases for starting capability-first. Here one missed clause cost more than a year of model spend, and volume is low. The lawyer-reviewed contracts let the firm measure each model on its own task before it commits. The eval then shows which model misses fewest clauses, likely the most capable one.

Why the other options are wrong

- (A) Run the reviews on the smallest model, and have a lawyer spot-check a random sample of its clause lists each week.
A weekly sample checks a few reviews after lawyers have acted on them, so a clause missed in an unchecked contract still reaches a signed deal.
- (B) Test each model on the lawyer-reviewed contracts, and deploy the cheapest model that misses only a few more clauses than the best.
Trading a small quality gap for a lower price suits high volume and cheap errors. Here the saving is small and each extra miss is costly.
- (D) Pick the model that leads public reasoning benchmarks for contract work, and move to each newer release as it launches.
Public benchmarks do not measure clause recall on the firm's contracts, and switching to each new release untested could lower recall unnoticed.

Exam tip: When errors are costly and volume is low, choose the model by its measured recall on your own documents.

Source: platform.claude.com/docs/en/about-claude/models/choosing-a-model

Question 16

- A** Decide on `stop_reason` instead: run the requested tools and call again while it is `tool_use`, and finish on any other value.

Why A is right

One reply can hold a short text block and a `tool_use` block together. A `stop_reason` of `tool_use` is what says Claude is waiting for a tool result. Keying the loop on `stop_reason` runs `lookup_order`, sends the `tool_result` back, and ends only on `end_turn` or another stop reason the app handles. Inferring the end of a turn from which blocks happen to be present cuts the answer short.

Why the other options are wrong

- B** Raise `max_tokens` on each request so Claude has room to finish the lookup and write the full answer in one reply.
The replies are not truncated. They end because Claude asked for a tool, and no single reply can hold the lookup's result before the app runs the tool.
- C** Set `tool_choice` to any on each request so Claude must call a tool in its turn instead of writing a text reply. Forcing a tool call on every request leaves Claude no turn in which to give the final answer.
- D** Add a system prompt rule telling Claude to write no text in a turn where it calls a tool, so each reply holds just the tool calls.
Text beside a tool call is normal, and a prompt only asks. Any reply that still includes a sentence ends the loop early again.

Exam tip: In a custom tool loop, branch on `stop_reason`, never on which content blocks are present.

Sources: platform.claude.com/docs/en/build-with-claude/handling-stop-reasons; platform.claude.com/docs/en/agents-and-tools/tool-use/handle-tool-calls

Question 17

- B** Append each `input_json_delta` to a buffer for that content block, and parse the buffer once its `content_block_stop` arrives.

Why B is right

During streaming, a tool's input arrives as fragments of a JSON string spread across delta events. One fragment may be half a key, so parsing each delta fails. Accumulating the fragments per content block and parsing at `content_block_stop` gives the same valid object the final message holds, while the explanation text keeps streaming to the user. The SDK streaming helpers do this accumulation for you.

Why the other options are wrong

- A** Turn off streaming for requests that include `create_chart`, and read the tool input from the complete response object instead.
Users watch the explanation stream in before the tool call, so turning streaming off freezes that text until the whole reply ends.

Question 17, continued

- (C) Set `strict: true` on `create_chart`, so that every `input_json_delta` the stream delivers is valid JSON on its own.
Strict mode makes the completed tool input match the schema, but the stream still delivers it in fragments that are not JSON on their own.
- (D) Wrap `JSON.parse` in a try block that skips any `input_json_delta` that fails, and draw from the deltas that parse.
Nearly every delta is a fragment, so skipping failures throws away most of the input and the chart is drawn from missing or wrong values.

Exam tip: Treat streamed tool input as complete only at the block's stop event, or let the SDK helpers collect it.

Source: platform.claude.com/docs/en/build-with-claude/streaming

Question 18

- (A) Describe date as an ISO 8601 calendar date with an example, and have the service reject other input with an `is_error` tool_result.

Why A is right

Tool input is typed data, and the parameter description tells Claude what a value means and how to format it. Naming ISO 8601 with an example tells Claude to resolve phrases such as next Friday itself and send an unambiguous date. When other input still arrives, an `is_error` tool_result with an actionable message lets Claude retry. The service stops guessing and booking the wrong day.

Why the other options are wrong

- (B) Replace the lenient parser with a natural-language date library that resolves relative phrases such as next Friday entirely on its own.
The library still guesses, now against the server's own date and time zone, and still books silently with no signal to the agent.
- (C) Add the booking service's date rules and several example requests to the system prompt, so Claude writes dates the same way.
Prompt rules ask but do not define the parameter, and the lenient service still accepts any string without telling the agent.
- (D) Set `strict: true` on `book_room` without changing its schema, so that every date the agent sends is guaranteed to be valid.
The schema says only that date is a string, so strict mode just ensures a string such as next Friday, which is what already arrives.

Exam tip: Define every format-sensitive parameter precisely in its description, and let the tool reject bad input loudly instead of guessing.

Sources: platform.claude.com/docs/en/agents-and-tools/tool-use/define-tools; platform.claude.com/docs/en/agents-and-tools/tool-use/handle-tool-calls

Question 19

- D** Replace the prohibitions with the target style: short paragraphs of complete sentences, as a patient reads them in an SMS.

Why D is right

Anthropic's formatting guidance says to tell Claude what to do instead of what not to do. Describing the target style steers the shape of the whole reply, which also removes the list structure. A ban gives no picture of a good reply, so Claude can switch to another list style. The test that stripped symbols showed the problem is the structure, not the characters.

Why the other options are wrong

- A** Extend the prohibitions to name asterisks, dashes, pound signs and numbered lists as characters the reply must not contain.
A longer list of bans is still only a list of bans. Claude can switch to another list style, and nothing says what a good SMS looks like.
- B** Strip markdown symbols from each reply in code before the SMS is sent, using a markdown-to-plain-text converter library.
The test showed that stripping symbols leaves clipped fragments, because the reply was written as a list.
- C** Return replies through structured outputs with a schema of one plain string field, and send that field's value as the SMS.
A schema controls the JSON around the text, not the text inside the string, so the field can still hold asterisks and dashed lists.

Exam tip: When Claude keeps producing a format you banned, rewrite the ban as a description of the format you want.

Source: platform.claude.com/docs/en/build-with-claude/prompt-engineering/claude-prom...

Question 20

- B** Move the current-time line out of the system prompt's opening and send it after the breakpoint, in the user turn of each request.

Why B is right

Prompt caching matches the prefix exactly, in the order tools, system, messages, up to the cache breakpoint. A time line at the very top changes on every request, so each request writes a new entry and none is read. That is why cache writes show on every call while reads stay at zero. With the changing line after the breakpoint, later requests read the policy manual from the cache.

Why the other options are wrong

- A** Switch the cache entry to the one-hour lifetime so the cached system prompt survives the gaps between consecutive requests from users.
Requests arrive every few seconds, well inside the default lifetime. A longer lifetime only raises the write price, and the changing time line still blocks every read.

Question 20, continued

- Ⓒ Add a second cache breakpoint directly after the current-time line so the policy manual that follows is cached as its own entry.
The cache key at any breakpoint covers everything before it, so a breakpoint after the time line still includes a value that changes every second.
- Ⓓ Expand the cached system prompt with more policy text so its length reaches the model's minimum cacheable size and starts producing reads.
A write of about 9,000 tokens on every call shows the prompt is already long enough to cache.

Exam tip: When cache writes appear on every call but reads stay at zero, look for a per-request value inside the cached prefix.

Source: platform.claude.com/docs/en/build-with-claude/prompt-caching

SAVE THIS PAGE

CCDV-F exam tips

- 1 In a custom tool loop, branch on stop_reason, never on which content blocks are present.
- 2 Give each subagent an explicit tools list matched to its job and a description that says when to use it.
- 3 Use batches only for work that can wait the whole processing window. A fast test batch says nothing about the next one.
- 4 Match the memory file to who should see it: user CLAUDE.md for you everywhere, project CLAUDE.md for the team, CLAUDE.local.md for you in one project.
- 5 When cache writes appear on every call but reads stay at zero, look for a per-request value inside the cached prefix.
- 6 For long documents, order the prompt as document first, then instructions, then the question.
- 7 When an agent reads private data and untrusted content, list every tool that can send data out, then remove it or put a user confirmation in front of it.
- 8 Remember the MCP control model: tools for the model, resources for the application, prompts for the user.



Good luck on exam day

Keep practising with our CCDV-F question set. There are more free questions on CertMadeEasy.com and in the app.

[More free CCDV-F questions →](#)

