DUMPSBOSS.

Claude Certified Architect - Foundations

Anthropic CCA-F

Version Demo

Total Demo Questions: 16

Total Premium Questions: 168

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

Topic Break Down

Topic	No. of Questions
Topic 1, Claude Fundamentals	17
Topic 2, Prompt Engineering	46
Topic 3, Claude API	27
Topic 4, Tool Use	74
Topic 5, Responsible AI	4
Total	168

QUESTION NO: 1

Your document-classification workflow has separate tools for classify_invoice, classify_contract, and classify_receipt. A request sometimes contains a document that does not fit any of these categories, but tool_choice is set to any. Claude then selects the closest tool and produces misleading structured data. What is the most effective design change?

A. Add an explicit unsupported_document or needs_review tool for documents that do not match an extraction schema.

B. Keep the three extraction tools and instruct Claude to select the schema with the largest number of applicable fields.

C. Set tool_choice to auto so Claude can return free-form text when a document is unfamiliar.

D. Merge invoice, contract, and receipt fields into one universal extraction tool.

ANSWER: A

Explanation:

Add an explicit unsupported_document or needs_review tool with a schema for the reason and observed document characteristics. When tool choice requires a tool call, the available tool set must represent every valid outcome, including the possibility that no extraction schema applies. An explicit abstention path is more reliable than forcing the closest of several incompatible schemas.

The review tool can return a concise classification rationale, source identifier, and recommended routing queue. The application can then send unsupported documents to human review or a specialized workflow while preserving guaranteed structured output. Anthropic documents that tool definitions and schemas should clearly describe the available actions and expected inputs. See Anthropic's tool use documentation.

QUESTION NO: 2

You're implementing a new payment processing module that must follow your project's established patterns for database transactions, error handling, and audit logging.

You've identified three existing modules that exemplify these patterns: db_utils.py, error_handlers.py, and audit_logger.py. This is a one-off integration task—these patterns are well-documented in your team wiki and don't need additional project-level documentation. What's the most effective approach?

A. Describe the patterns from the three modules in natural language in your prompt, explaining the transaction handling approach, error format, and logging conventions Claude should follow.

B. Use @references to include the three modules directly in your prompt, giving Claude concrete code examples of the patterns to follow.

C. Ask Claude to explore your codebase to find and understand the transaction, error handling, and logging patterns before generating the new module.

D. Add documentation of each pattern to your CLAUDE.md file, establishing them as project conventions that Claude will apply automatically.

ANSWER: B

Explanation:

Use @references to include the three modules directly in your prompt, giving Claude concrete code examples of the patterns to follow. This provides the most relevant and authoritative context for a bounded implementation task. The source files show the exact transaction scopes and rollback conventions, exception classes and response structures, audit-event payloads, logging calls, naming choices, and local code organization that the new payment module must match. Concrete examples are more reliable than a prose restatement when consistency with an existing codebase matters, because Claude can work from the project's actual implementation rather than an abbreviated interpretation of it.

This approach is also appropriately scoped. Supplying the files in the task prompt gives Claude the context only for this payment-processing work, while preserving the established team wiki as the durable documentation source. Claude Code supports explicitly adding files and directories to a prompt with `@` references, allowing users to focus Claude on relevant

codebase context. Anthropic's prompting guidance also recommends providing examples when a specific structure, style, or behavior should be reproduced. The generated implementation should still be reviewed and tested, particularly around rollback paths and audit-event coverage. See Claude Code common workflows and Anthropic's multishot prompting guidance.

QUESTION NO: 3

Your agent receives a tool result from search_orders containing five orders with stable order IDs. The user asks to cancel the most recent order. Before calling cancel_order, the agent selects an ID based on a date displayed in the tool result. Because several orders have the same date and different fulfillment states, the wrong order is occasionally cancelled. What change most effectively improves safety?

A. Present candidate orders with differentiating details and require confirmation of the selected order before cancellation.

B. Instruct the agent to choose the order ID associated with the latest date returned by search_orders.

C. Modify cancel_order to accept an order date instead of an order ID so the agent does not select among records.

D. Automatically cancel every order sharing the most recent date, then allow the user to restore unwanted cancellations.

ANSWER: A

Explanation:

Have the application present the candidate orders with differentiating details and require the user to confirm the specific order before cancellation. "Most recent" can be ambiguous when multiple records share the same displayed date, and fulfillment state, item details, total, or timestamp may materially affect the user's intent. The user can resolve that ambiguity using the actual external records rather than relying on the agent to infer a target.

The confirmation should be bound to the selected order ID and expire after a short period so it cannot be reused for another cancellation. This creates a focused approval step for a consequential action while preserving a low-friction workflow. See Anthropic's tool-use implementation guidance.

QUESTION NO: 4

Your application sends a 60,000-token product-policy reference document and a stable system prompt with every customer-support request. Most requests differ only in the final user message. API costs and time to first token are increasing. What is the most effective way to reduce repeated processing of the stable context?

A. Use prompt caching for the stable system prompt and product-policy reference document, placing the changing customer request after the cached prefix.

B. Move the product-policy reference document into the user message so the system prompt remains small and can be processed more quickly.

C. Store each customer conversation in a separate Messages API session so the reference document is retained automatically by the API.

D. Send only the policy sections that mention words found in the customer's latest message.

ANSWER: A

Explanation:

Use prompt caching for the stable system prompt and product-policy reference document, placing the changing customer request after the cached prefix. Prompt caching is designed for workloads in which a large, repeated prefix is reused across multiple requests. The first request creates a cache entry for the shared content, and later requests that match the cached prefix can reuse it rather than repeatedly processing the full reference material.

The cacheable content must remain identical through the configured cache breakpoint. Putting dynamic information, such as the current customer message or request-specific account details, after that stable prefix preserves cache reuse while

ensuring Claude receives the current task context. This is more effective than summarizing policy material on every turn because it retains the complete approved reference while reducing repeated input processing. See Anthropic's prompt caching documentation.

QUESTION NO: 5

Your automated reviewer uses a single prompt covering security issues, API design, and business logic correctness. Your evaluation suite shows strong recall findings (82%) but poor recall for business logic edge cases in quiz scoring (34%). When you add few-shot examples of logic bugs to the prompt, logic recall is 41% but API design recall drops to 68%. How should you address this trade-off to improve detection across both categories?

A. Split the review into separate focused prompts - one for security and API design, another for business logic - each with dedicated examples, then combine findings before posting.

B. Upgrade to a more capable model tier, since its stronger reasoning will handle both concern types in a single prompt and eliminate the recall trade-off.

C. Provide the full repository as context instead of just the changed files and surrounding code, giving the model deeper visibility into business logic.

D. Replace the few-shot examples with a detailed checklist of specific logic edge cases to verify, such as division-by-zero in score calculation or grading thresholds.

ANSWER: A

Explanation:

Split the review into separate focused prompts - one for security and API design, another for business logic - each with dedicated examples, then combine findings before posting. The evaluation results show that a single general-purpose reviewer prompt has competing objectives: adding business-logic examples modestly improves logic-edge-case recall, but it reduces API-design recall. Separate reviewer passes allow each prompt to have an unambiguous task, relevant review criteria, and examples that closely match its target failure modes. The business-logic pass can concentrate on quiz-scoring invariants, boundary values, state transitions, rounding, and expected outcomes. The security and API-design pass can retain the instructions and examples that already support its stronger results. This design also enables independent, category-specific evaluation, so the team can measure recall and precision for each pass and iterate without obscuring regressions in another review area. After execution, the system can normalize, deduplicate, and merge findings into one developer-facing review. Anthropic recommends clear instructions and relevant examples for the task at hand, while its evaluation guidance emphasizes measuring the specific behaviors an application needs to perform. See Anthropic's multishot prompting guidance and guidance on developing evaluations.

QUESTION NO: 6

Production logs show that when the agent handles complex billing disputes requiring 6+ tool calls, it sometimes exhausts its max_turns limit after gathering data and before completing resolution or escalating. The team's goal is to guarantee that every customer interaction ends with either a completed resolution or a human escalation, regardless of how the agent loop terminates. Which approach achieves this guarantee?

A. Add orchestration-layer code that checks the agent's outcome after each loop termination - if the loop ended without a completed resolution or escalation, programmatically call escalate_to_human with the accumulated conversation context and tool results.

B. Implement a pre-tool-use hook that counts tool invocations and terminates the loop with an automatic escalation once the agent reaches 80% of its remaining actions.

C. Add system prompt instructions telling the agent to call escalate_to_human with a summary of its findings whenever it determines it cannot resolve the dispute.

D. Split the workflow into two sequential agent invocations — a first agent gathers information via get_customer and lookup_order, then the second agent uses that data and handles process_refund or escalate_to_human, each with separate turn budgets.

ANSWER: A

Explanation:

Add orchestration-layer code that checks the agent's outcome after each loop termination - if the loop ended without a completed resolution or escalation, programmatically call escalate_to_human with the accumulated conversation context and tool results. This enforces the required terminal-state invariant independently of model behavior and independently of the reason the agent loop stopped. A turn budget is a runtime safeguard, so an agent may reach its limit after successfully gathering information but before it can issue its intended final tool call or response.

The application orchestrator owns the execution loop and can reliably inspect the resulting state. It can determine whether a successful resolution was recorded or whether the human-handoff workflow was invoked. When neither terminal state exists, the orchestrator can deterministically trigger escalation and pass along the conversation history, retrieved customer details, order information, and tool outputs. This both prevents abandoned interactions and preserves work already completed by the agent for the human reviewer. Anthropic's tool-use guidance describes the client application's responsibility for executing tools, returning results, and managing the iterative workflow; this is the proper control point for a guaranteed fallback. See Implement tool use and Tool use with Claude.

QUESTION NO: 7

Your team frequently migrates React components to Vue. You've written a step-by-step workflow for Claude Code to follow during each migration, and you want every developer on the team to invoke it by typing /migrate-component. The workflow should stay in sync as the team iterates on it. Where should you place the skill file?

A. In ~/.claude/skills/migrate-component/SKILL.md on each developer's machine

B. In the project's .claude/settings.json using a skillOverrides entry to register and define the workflow

C. In .claude/skills/migrate-component/SKILL.md at the project root, committed to version control

D. As a detailed instruction block in the project's root CLAUDE.md file

ANSWER: C

Explanation:

In `.claude/skills/migrate-component/SKILL.md` at the project root, committed to version control is correct because Claude Code loads project skills from the repository's `.claude/skills/` directory. The skill directory name defines its slash-command name, so the `migrate-component` directory makes the workflow available to developers as `/migrate-component`. The `SKILL.md` file is designed to hold a reusable, structured procedure, including migration steps, validation expectations, conventions, and relevant context. This makes it well suited to a repeated React-to-Vue component migration process. Committing the skill with the project is equally important: it gives every team member the same workflow when they work in the repository, while normal version-control practices allow updates to be reviewed, versioned, and distributed alongside the code and conventions that the workflow supports. Project-scoped skills therefore provide both the desired slash-command interface and a shared source of truth for evolving team procedures. Anthropic documents project skill discovery, skill naming, and invocation in the Claude Code skills documentation. For related guidance on sharing repository-level Claude Code configuration and instructions with a team, see the Claude Code memory documentation.

QUESTION NO: 8

The coordinator agent has AgentDefinitions configured for all four specialized subagents, each with appropriate descriptions, prompts, and tool restrictions. During testing, you notice the coordinator correctly reasons about when to delegate—it generates messages like “I’ll ask the web search agent to find sources on this topic”—but no subagent execution ever occurs. The coordinator then proceeds as if the delegation happened and continues with incomplete information. Logs show no errors. What is the most likely cause?

A. Subagent context isolation means task descriptions from the coordinator don’t automatically reach subagents; you need to configure explicit context forwarding in Claude AgentOptions.

B. The coordinator’s max_tokens setting is too low, causing the Task tool invocation to be truncated before the subagent type parameter can be specified.

C. The coordinator's allowed Tools configuration doesn't include "Task", so while it can reason about delegation, cannot invoke the tool required to spawn subagents.

D. The AgentDefinitions are configured correctly, but the coordinator's system prompt doesn't explicitly list the available subagent types, preventing the model from knowing they can be invoked.

ANSWER: C

Explanation:

The coordinator's allowed Tools configuration doesn't include "Task", so while it can reason about delegation, cannot invoke the tool required to spawn subagents. In Anthropic's subagent architecture, defining specialized agents supplies their names, descriptions, system prompts, and permitted capabilities, but delegation requires the parent to make a Task tool call. This call specifies which subagent should run and provides the task for that agent to perform. A model can still write natural-language text expressing an intent to delegate when its instructions and available agent definitions make that workflow apparent. However, an intention stated in generated text is not an execution. If the coordinator cannot access the Task tool, it has no mechanism to launch a subagent and receive that subagent's result in its context. The absence of an execution error is therefore consistent with the observed behavior: no tool call is attempted, and the coordinator simply continues generating an answer from the information it already has. Allowing Task for the coordinating agent while limiting each specialist to only the tools required for its role implements a clear delegation boundary and least-privilege design. See Anthropic's subagents documentation and tool use overview.

QUESTION NO: 9

Your application submits a request with three independent tool calls: retrieve_account, retrieve_orders, and retrieve_support_tickets. The client executes all three successfully, but returns only the retrieve_account result in the next message. Claude then asks to retrieve orders and tickets again. How should the client return results for the original response?

A. Return a tool_result block for each executed tool call, with each block referencing its corresponding tool_use_id.

B. Combine all three results into one tool_result block using the tool_use_id from the first call.

C. Return the account result first, then send orders and tickets as separate assistant messages after Claude responds.

D. Include the three tool outputs as a JSON object in the system prompt on the next request.

ANSWER: A

Explanation:

Return a `tool_result` block for each executed tool call, with each block referencing its corresponding `tool_use_id`. When Claude emits multiple tool-use blocks, each call is a distinct requested operation. The client should execute the requested calls and supply every available result in the following user message so Claude has the complete evidence needed to continue the workflow.

Returning only one result leaves the other requested operations unresolved from Claude's perspective. It may therefore request them again or proceed without data that was actually retrieved. Associating each result with the exact tool-use identifier is essential because tool names can repeat and multiple calls can be present in a single assistant message. See Anthropic's tool-use implementation guidance.

QUESTION NO: 10

Your application receives natural-language requests to send invoices. Before calling send_invoice, it must obtain a customer ID, an approved invoice ID, and confirmation that the invoice has not already been sent. The current agent sometimes calls send_invoice using a customer name and an invoice title, causing duplicate or misdirected emails. What tool design change most effectively improves reliability?

A. Require send_invoice to accept canonical customer and invoice identifiers, and have the application validate approval status and idempotency before sending.

B. Allow send_invoice to accept names and titles, but add examples showing how the agent should format each value.

C. Have send_invoice search for the closest customer name and invoice title internally, then send to the first matching record.

D. Increase the detail in the system prompt instructing the agent to confirm customer and invoice names before every send.

ANSWER: A

Explanation:

Require `send_invoice` to accept canonical customer and invoice identifiers, and have the application validate approval status and idempotency before sending. Names and titles are ambiguous presentation values; they are not dependable identifiers for a consequential action. Search or lookup tools can return candidate records with stable IDs and differentiating metadata, allowing the user or agent to identify the intended customer and invoice before execution.

The send operation should independently verify that the invoice is approved, belongs to the specified customer, and has not already been delivered or processed with the same idempotency key. These conditions are business and state-validation rules that must be enforced in application code rather than delegated to an instruction. This design gives the agent clear tool inputs while protecting against duplicate and incorrectly targeted sends. See Anthropic's tool-use implementation guidance.

QUESTION NO: 11

Your MCP server implements a check_availability tool that queries an external calendar API. During testing, you encounter three error conditions:

(1) the tool is called with a malformed request, missing the required user_email parameter

(2) the calendar API returns a 404 because the specified user doesn't exist in the calendar system

(3) the calendar API returns a 503 because the service is temporarily unavailable. How should each error be reported according to MCP's error handling design?

A. Report all three as tool results with isError: true

B. Report errors 1 and 2 as JSON-RPC protocol errors, report error 3 as a tool result with isError: true

C. Report error 1 as a JSON-RPC protocol error, report errors 2 and 3 as tool results with isError: true

D. Report all three as JSON-RPC protocol errors.

ANSWER: C

Explanation:

Report error 1 as a JSON-RPC protocol error, report errors 2 and 3 as tool results with isError: true. The missing `user_email` field means the `tools/call` request does not satisfy the input schema declared for `check_availability`. Because the server cannot validate the request parameters required to invoke the tool, this is a request-level failure and should be returned as a JSON-RPC error, ordinarily the Invalid Params error. This communicates that the invocation itself was invalid rather than that a successfully started tool operation encountered an expected runtime problem.

Once a call has passed validation and the tool begins its calendar lookup, failures reported by the calendar system are execution outcomes. A nonexistent calendar user and a temporary service outage are both useful results for the MCP client and model to receive in the normal tool-result flow. The server should therefore return a `CallToolResult` with meaningful content describing the condition and set `isError` to `true`. This distinction preserves JSON-RPC errors for protocol and parameter handling while allowing the model to react to operational conditions, such as asking for a different user or trying again later. See the MCP Tools specification and the JSON-RPC 2.0 specification.

QUESTION NO: 12

During testing, you observe that in extended exploration sessions (30+ minutes), the agent starts giving inconsistent answers about context discussed earlier. Engineers report having to repeat context about modules they've already explored. What's the most effective approach to address this?

A. Create summaries of all source files before exploration begins, loading only these compressed representations into context.

B. Switch to a higher-capacity model tier to provide more context window space for accumulated exploration data.

C. Implement automatic context clearing every 15 minutes to ensure the agent starts with fresh, uncontaminated context.

D. Have the agent maintain a scratchpad file that records key findings, referencing it for subsequent questions.

ANSWER: D

Explanation:

Have the agent maintain a scratchpad file that records key findings, referencing it for subsequent questions. This creates durable external memory for an exploration that can outgrow the practically useful portion of the active prompt. The scratchpad should capture concise, evidence-based information such as modules examined, interfaces and dependencies, decisions, assumptions, unresolved questions, and paths or links to relevant code. Before undertaking a new investigative step or responding to a related question, the agent can retrieve the relevant entries; after meaningful work, it should update the artifact. This preserves continuity while keeping the immediate context focused on the information needed for the current task instead of repeatedly retaining every raw tool output, file, and conversational turn. A maintained scratchpad also makes the agent's evolving understanding inspectable: engineers can review, correct, or augment recorded findings without needing to reconstruct a lengthy interaction. Anthropic's guidance describes context as a finite resource and recommends deliberately managing it through techniques such as external memory, retrieval, and compaction for long-running agentic work. See the Anthropic context windows documentation and Anthropic's effective context engineering guidance.

QUESTION NO: 13

Your fitness coaching assistant uses a system prompt with detailed conditional logic: "If the user mentions being a beginner, provide step-by-step form instructions. If they use terms like 'progressive overload' or 'superset', respond concisely. If they ask about injury history, always recommend consulting a physician." During evaluation, you find the assistant correctly adapts to explicit expertise declarations but struggles when users don't clearly state their level—often defaulting to overly detailed responses regardless of contextual cues like technical terminology. Which change to the system prompt would most directly address this failure to pick up on implicit expertise signals?

A. Add an explicit instruction for the model to ask a clarifying question about experience level whenever the user's expertise isn't immediately clear from their first message.

B. Replace most conditionals with a general principle: "Adapt explanation depth to match user expertise, mirroring their terminology." Keep only the safety-critical conditional about injury consultations.

C. Implement a pre-conversation intake that asks users to rate their experience level, then inject that rating into the system prompt as context for all subsequent responses.

D. Add more conditional branches to cover additional expertise signals, such as "If user mentions specific rep ranges or asks about periodization, treat as advanced."

ANSWER: B

Explanation:

Replace most conditionals with a general principle: "Adapt explanation depth to match user expertise, mirroring their terminology." Keep only the safety-critical conditional about injury consultations. This directly addresses the stated problem because it instructs Claude to assess expertise from the user's overall language and context, rather than relying on a short, predefined list of phrases. A user may demonstrate familiarity through the specificity of a question, use of domain vocabulary, the structure of a training plan, or an implicit assumption about exercise concepts. A general adaptation principle enables the assistant to use those signals together and choose an appropriate level of detail even when the user never explicitly labels themselves as a beginner or advanced trainee. It also gives a clear desired outcome—calibrate depth and vocabulary to the person—while avoiding brittle keyword-trigger logic that cannot feasibly cover every way expertise may be

expressed. The physician-consultation instruction should remain explicit because it is a health-related safety constraint where consistent handling is important. Anthropic recommends giving Claude clear, direct instructions focused on the desired behavior and using prompt structure that supports reliable behavior across varied inputs. See Anthropic's prompt engineering overview and guidance on being clear and direct.

QUESTION NO: 14

Monitoring shows 12% of extractions fall Pydantic validation with specific errors like "expected float for quantity, got '2 to 3". Retrying these requests without modification produces failures. What's the most effective approach to recover from these validation failures?

A. Set temperature to 0 to eliminate output variability and ensure consistent formatting

B. Send a follow-up request including the validation error, asking the model to correct its output

C. Pre-process source documents to standardize problematic formats before sending them for extraction

D. Implement a secondary pipeline using a larger model tier to reprocess documents that fail validation

ANSWER: B

Explanation:

Send a follow-up request including the validation error, asking the model to correct its output is the most effective recovery mechanism because it turns a failed parse into actionable feedback for Claude. The validation message identifies the exact schema mismatch—here, that `quantity` must be a float but the extracted value is a range expressed as text. Including that error, together with the prior response or the relevant extraction context, lets the model revise the invalid field and return a response that conforms to the required Pydantic schema. This creates a targeted repair loop: generate an extraction, validate it, and, if validation fails, issue a bounded correction request containing the validation failures and an explicit instruction to return schema-compliant output. The corrected result can then be validated again before it enters downstream systems. This approach is particularly useful when most extractions already succeed and the failures are localized formatting or normalization issues rather than evidence that the underlying document cannot be processed. Anthropic's structured-output documentation describes defining schemas so responses are reliably shaped for programmatic use, while its prompting guidance emphasizes supplying clear constraints and relevant context to obtain a usable corrected result. See Anthropic Structured Outputs documentation and Anthropic Prompt Engineering overview.

QUESTION NO: 15

Your automated review calls the Claude API for each PR, using tool_use with a report_findings tool that returns a JSON array of finding objects (each with file_path, line_number, severity, category, and description). During testing on a large PR touching 30+ files, the response hits the max_tokens limit and the output is truncated mid-JSON,

causing your pipeline's parser to fail. What is the most effective way to handle this?

A. Increase max_tokens to the model's maximum and instruct Claude to keep finding descriptions under 50 words each.

B. Switch from tool_use to prompting Claude to return findings as a markdown list.

C. Split the review into multiple API calls that each analyze a subset of the changed files, then merge the resulting findings arrays.

D. Add retry logic that detects truncated JSON and re-sends the request with instructions to report only critical and high severity findings.

ANSWER: C

Explanation:

Split the review into multiple API calls that each analyze a subset of the changed files, then merge the resulting findings arrays. This handles the root cause: the findings for a large pull request exceed the response-token budget, causing

generation to stop before the tool input is complete. Batching the changed files into bounded groups keeps both the code context and the expected structured output at a manageable size, so each tool-use call is much more likely to produce a complete JSON array. Each batch result can be validated independently against the required finding schema before it is added to the consolidated review report. If a batch does fail, the pipeline can retry or subdivide only that batch rather than repeating analysis of the entire pull request. The retained file paths and line numbers make it straightforward to merge, sort, and deduplicate findings downstream. Batch sizes should be selected based on diff size, file complexity, and expected finding density; especially large files can be submitted separately. Anthropic documents that responses may stop upon reaching the configured maximum-token limit, identified through the response stop reason. Designing requests so that their complete structured result fits in the available output budget is therefore the reliable production approach. See the Messages API documentation and Anthropic's tool use documentation.

QUESTION NO: 16

Your codebase exploration tool stores session IDs to allow engineers to continue investigations across work sessions. An engineer spent an hour yesterday analyzing a legacy authentication module, building context about its architecture and dependencies. They want to continue today. The session ID is valid, but version control shows 3 of the 12 files the agent previously read were modified overnight by a teammate's merge. What approach best balances efficiency and accuracy?

A. Resume the session and immediately have the agent re-read all 12 previously analyzed files

B. Resume the session without informing the agent about the changed files

C. Resume the session and inform the agent which specific files changed for targeted re-analysis

D. Start a fresh session to ensure the agent works with current codebase state without stale assumptions

ANSWER: C

Explanation:

Resume the session and inform the agent which specific files changed for targeted re-analysis is the best approach because it retains the valuable investigative context accumulated in the prior session while explicitly grounding the continued work in the repository's current state. The agent can first inspect the modified files, compare their present behavior and interfaces with the assumptions established previously, and then reassess conclusions that may be affected by those changes. This makes the refresh proportional to the known change scope: it avoids repeating analysis of unchanged files unless the updated code reveals dependencies, call paths, tests, or contracts that also need review. Effective Claude workflows depend on supplying relevant and current context, especially when prior context may no longer fully reflect the source of truth. Clear instructions should identify the changed files and request validation of any earlier findings dependent on them. This preserves efficiency without treating session history as an authoritative snapshot of a mutable codebase. Anthropic recommends giving Claude direct, specific instructions and providing the contextual information needed for the task; see its guidance on being clear and direct and context window documentation.