DUMPSBOSS.

Claude Certified Architect – Foundations

Anthropic CCAR-F

Version Demo

Total Demo Questions: 16

Total Premium Questions: 169

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

QUESTION NO: 1

You are integrating Claude Code into your Continuous Integration/Continuous Deployment (CI/CD) pipeline. The system runs automated code reviews, generates test cases, and provides feedback on pull requests. You need to design prompts that provide actionable feedback and minimize false positives.

Your pipeline includes a release-notes generation step that classifies and summarizes approximately 200 commits at the end of each weekly release cycle. Each commit is currently sent as a separate Messages API request using a Sonnet-tier Claude model. The release notes are not needed until the following morning, providing approximately 12 hours of acceptable latency.

Your team must reduce the per-token API cost while retaining the same model, prompts, and output quality.

Which approach satisfies all these constraints?

A. Issue the 200 Messages API requests concurrently because parallel execution reduces the per-token price.

B. Submit the 200 requests through the Message Batches API with unique custom_id values and retrieve the results after the batch finishes.

C. Concatenate all 200 commit messages into one Messages API request because reducing the number of requests always reduces token costs.

D. Replace the Sonnet-tier model with a Haiku-tier model to obtain a lower per-token price.

ANSWER: B

Explanation:

Submit the 200 requests through the Message Batches API with unique custom_id values and retrieve the results after the batch finishes. This is the appropriate cost-optimization mechanism because Message Batches accepts a collection of independent Messages API requests while preserving the request-level model selection, message content, parameters, and expected response behavior. The team can therefore continue using the same Sonnet-tier model and the same proven commit-classification and summarization prompts, maintaining the established output-quality profile.

Anthropic offers batch processing at a 50% discount on both input and output token usage compared with standard Messages API pricing. This directly addresses the requirement to lower per-token cost, rather than merely changing execution timing or reducing the amount of work performed. The deferred release-notes workflow is suitable for asynchronous processing because its results are not required during an interactive pull-request review. Each request should have a unique `custom_id`, allowing the pipeline to reliably map completed results, failures, and retries back to the relevant commit even if results are returned in a different order. See Anthropic's Message Batches documentation and pricing documentation.

QUESTION NO: 2

You are integrating Claude Code into your Continuous Integration/Continuous Deployment (CI/CD) pipeline. The system runs automated code reviews, generates test cases, and provides feedback on pull requests. You need to design prompts that provide actionable feedback and minimize false positives.

Your automated code review is missing genuine bugs in pull requests. Investigation reveals that your review prompt includes the instruction: "Only flag critical issues that would definitely cause production failures. Ignore minor concerns and anything you are uncertain about." Developers confirm that some missed bugs are genuine logic errors that the model investigated but chose not to report. The team requires the review output to remain structured, with each finding tagged with metadata, and actionable.

Which prompt change both removes the cause of the suppressed findings and preserves structured, tagged output for downstream filtering?

A. Add a second review pass that rereads the diff using the same prompt, looking for anything the first pass may have missed.

B. Instruct the model to report all findings with confidence and severity tags, deferring filtering to a downstream step.

C. Remove all severity-related instructions from the prompt and let the model use its default judgment about what to report.

D. Enable extended thinking and instruct the model to reason step by step about every code change before producing its review.

ANSWER: B

Explanation:

Instructing the model to report all findings with confidence and severity tags, deferring filtering to a downstream step, directly removes the instruction that caused genuine issues to be withheld. The original prompt explicitly tells the model to discard concerns that are not judged certain or critical, so a model can identify a plausible logic error during review but omit it from its final response. Asking it to surface findings rather than pre-filter them improves recall while retaining useful context about the model's assessment of each issue.

Confidence and severity metadata preserve the structured contract required by the CI/CD workflow. For example, each finding can include a concise description, affected file and line, suggested remediation, severity, and confidence. A deterministic downstream policy can then decide which findings block a pull request, create a ticket, or are shown as advisory feedback. This separates detection from enforcement: Claude provides comprehensive, actionable review evidence, while the pipeline applies organization-specific risk thresholds consistently. Anthropic's guidance on structured outputs supports defining an explicit response schema for reliable machine consumption, and its prompting guidance recommends clear instructions and output formats. See Anthropic Structured Outputs documentation and Anthropic Prompt Engineering overview.

QUESTION NO: 3

A customer-support agent can use `get_customer`, `lookup_order`, `process_refund`, and `escalate_to_human`. In evaluation, it handles routine refund requests correctly but sometimes continues searching after all required facts are known, making unnecessary tool calls before responding. In other cases, it continues retrying after receiving a clearly non-retriable business-policy error.

What instruction design most directly addresses both behaviors?

A. Increase the maximum number of agent turns so the agent has more opportunity to discover a successful path before responding.

B. Require the agent to call every available customer and order tool before it provides a final response.

C. Define explicit completion and stopping criteria: respond once the issue is resolved or a clear next action is available; do not repeat non-retriable failed actions; and escalate when authorized resolution is impossible.

D. Instruct the agent to minimize tool use and answer the customer immediately whenever a request appears to be a refund inquiry.

ANSWER: C

Explanation:

Define explicit completion and stopping criteria: respond once the requested issue is resolved or a clear next action is available; do not repeat non-retriable failed actions; and escalate when authorized resolution is impossible. These criteria give the agent a decision boundary for ending investigation, distinguishing useful additional evidence from redundant work, and transferring responsibility when it cannot safely proceed.

Increasing the turn limit permits more unnecessary calls. Requiring every available tool to be used would worsen the over-exploration. A blanket instruction to answer quickly could cause premature responses before required customer or order information has been verified.

QUESTION NO: 4

When analyzing complex legal cases that cite multiple precedents, the document-analysis subagent processes each precedent sequentially. A landmark case citing 12 precedents takes more than three minutes to analyze completely. What is the most effective way to reduce this latency while preserving the coordinator's ability to monitor and debug the system?

A. Have the coordinator spawn parallel document-analysis subagents, each handling a subset of precedents, and then aggregate the results before synthesis.

B. Enable the document-analysis subagent to spawn its own specialized subagents dynamically when it encounters cases with many citations.

C. Create a recursive agent hierarchy where analysis agents subdivide work among child agents until reaching single-precedent granularity.

D. Implement a message queue where precedent-analysis tasks are processed asynchronously by a pool of worker agents.

ANSWER: A

Explanation:

Have the coordinator spawn parallel document-analysis subagents, each handling a subset of precedents, and then aggregate the results before synthesis. This approach addresses the actual bottleneck: the cited precedents are independent analysis units, so they can be examined concurrently rather than forcing the system to wait for one precedent's review before beginning the next. The coordinator retains explicit ownership of decomposition, assigning balanced batches and a consistent output contract—for example, holdings, jurisdiction, relevance, treatment by the landmark case, and supporting citations.

Centralized orchestration also preserves the observability required for production legal-analysis workflows. The coordinator can record the prompt and inputs supplied to each worker, track progress and failures, retry a failed task, validate that every precedent has a result, and associate each structured response with its source before passing the complete set to a synthesis step. This creates a clear execution trace without adding unnecessary layers of delegation. Anthropic describes this orchestrator-worker pattern as useful when a lead agent can break a task into parallel, independent subtasks and then combine the results. See Anthropic's Building a multi-agent research system and the Claude Agent SDK overview for guidance on coordinating agentic workflows and tool use.

QUESTION NO: 5

When researching "renewable-energy adoption," the web-search agent returns recent statistics showing 35% adoption in 2024, while the document-analysis agent extracts an 18% adoption figure from an internal 2021 report. The synthesis agent incorrectly treats the figures as contradictory instead of recognizing that they may show growth over time. What change would best enable the synthesis agent to interpret such temporal differences correctly?

A. Require subagents to include publication dates and data-collection periods in their structured outputs.

B. Configure the web-search agent to return only results published during the previous six months.

C. Add a conflict-resolution agent that automatically discards older data whenever a newer value exists for the same metric.

D. Instruct the synthesis agent to treat the newest value as authoritative and place all older findings in a separate historical section.

ANSWER: A

Explanation:

Require subagents to include publication dates and data-collection periods in their structured outputs. This gives the synthesis agent the essential provenance needed to distinguish observations made at different times from genuinely conflicting claims about the same period. In this case, identifying one metric as pertaining to 2021 and the other to 2024 lets the agent evaluate the values as possible evidence of increasing adoption, rather than automatically classifying them as a contradiction. Data-collection or observation periods are particularly important because a document's publication date may differ substantially from when the underlying measurement occurred.

A robust schema can also capture source URLs, geographic scope, population, metric definition, unit, and methodology. With these fields, the synthesis agent can make a qualified conclusion: adoption appears to have risen from 18% in 2021 to 35% in 2024, provided both sources measure a comparable population using compatible definitions. Anthropic recommends using structured outputs to constrain responses to a defined schema, making metadata consistently available to downstream

steps in a multi-agent workflow. See Anthropic Structured Outputs documentation and Anthropic prompt engineering guidance.

QUESTION NO: 6

You are using Claude Code to accelerate software development. Your team uses it for code generation, refactoring, debugging, and documentation. You need to integrate it into your development workflow with custom slash commands, CLAUDE.md configurations, and understand when to use plan mode vs direct execution.

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

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

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

ANSWER: D

Explanation:

Place the workflow in `.claude/skills/migrate-component/SKILL.md` at the repository root and commit it to version control. Claude Code treats a project skill's directory name as its slash-command name, so the `migrate-component` directory makes the skill available as `/migrate-component`. A skill is the appropriate format for this use case because it packages a reusable, multi-step procedure that Claude loads when a developer explicitly invokes it, rather than placing the full migration process into every conversation.

Keeping the skill in the project's `.claude` directory gives it repository-level scope. When the directory and its `SKILL.md` file are committed, every team member receives the same migration workflow through the normal clone, pull-request, review, and versioning process. This also keeps workflow updates synchronized with the codebase to which they apply. Claude Code distinguishes these project-scoped skills from skills stored in a user's home directory, which are intended for an individual user across projects. Anthropic documents the project skill location, naming convention, and invocation behavior in the Claude Code skills documentation. See also the guidance on shared, repository-level configuration in the Claude directory documentation.

QUESTION NO: 7

Your team changes an extraction prompt to improve recognition of handwritten delivery addresses. On the same 80 documents used to develop the prompt, address accuracy rises from 76% to 94%. However, the team also repeatedly inspected incorrect outputs from those documents while refining the instructions and examples.

What is the most reliable next step before deploying the change?

A. Deploy the revised prompt because the measured increase on the development documents is large enough to demonstrate reliability.

B. Evaluate the revised prompt on a separate, labeled holdout set that was not used to inspect failures or choose examples.

C. Add the remaining incorrect development documents as few-shot examples until the accuracy reaches 100%.

D. Compare the old and new prompts by measuring average response latency and input-token usage on the development documents.

ANSWER: B

Explanation:

Evaluate the revised prompt on a separate, labeled holdout set that was not used to inspect failures or choose examples. The development set is no longer an unbiased measure of generalization because the prompt was iteratively tailored using its errors. A holdout evaluation provides evidence of whether the improvement transfers to new documents, including varied handwriting and layouts.

Adding more examples from the same 80 documents may further overfit the prompt to known patterns. Comparing only token use or response latency does not measure extraction correctness. Production monitoring is useful after release, but it should complement rather than replace a pre-deployment evaluation on unseen representative data.

QUESTION NO: 8

You are using Claude Code to accelerate software development. Your team uses it for code generation, refactoring, debugging, and documentation. You need to integrate it into your development workflow with custom slash commands, CLAUDE.md configurations, and understand when to use plan mode vs direct execution.

You're implementing a caching layer for API responses to speed up the /products endpoint. You have a rough idea—Redis with a 5-minute TTL—but you're new to production caching and aren't sure what other considerations a robust implementation requires.

What's the most effective way to start your iterative workflow?

A. Ask Claude to interview you about the caching requirements before implementing, surfacing considerations like invalidation strategies, cache layers, consistency guarantees, and failure modes.

B. Use plan mode to analyze the current /products endpoint implementation, then provide your caching requirements once Claude explains how the existing code is structured.

C. Start with a minimal request: "Add Redis caching to /products with 5-minute TTL." Add features and fix issues through follow-up prompts as problems surface during testing.

D. Write a specification with your known requirements and "TBD" markers for uncertain areas, having Claude propose solutions for each TBD as it implements.

ANSWER: A

Explanation:

Ask Claude to interview you about the caching requirements before implementing, surfacing considerations like invalidation strategies, cache layers, consistency guarantees, and failure modes. This is the most effective first step because the core uncertainty is not how to write Redis integration code; it is defining the behavior that makes caching safe and appropriate for this particular product endpoint. A short requirements interview can establish what events invalidate product data, whether temporarily stale responses are acceptable, how cache keys must isolate users or tenants, how to handle cache outages, and which metrics or logs will verify that the cache is improving performance without causing incorrect responses.

Claude Code supports interactive clarification through its question-asking capability, which is especially useful when a request has important unstated constraints. Once those decisions are captured, they can be turned into a clear, testable implementation specification. Plan Mode can then be used to inspect the repository, identify the appropriate integration points, and propose a scoped change before files are modified. This sequence reduces rework and makes the eventual implementation easier to review and validate. See Anthropic's guidance on planning and working effectively with Claude Code in the Claude Code best practices and its interactive mode documentation.

QUESTION NO: 9

You are building developer productivity tools using the Claude Agent SDK. The agent helps engineers explore unfamiliar codebases, understand legacy systems, generate boilerplate code, and automate repetitive tasks. It uses the built-in tools (Read, Write, Bash, Grep, Glob) and integrates with Model Context Protocol (MCP) servers.

A developer asks the agent to investigate why a specific API endpoint intermittently returns 500 errors. The codebase has 200+ files and the developer doesn't know which components are involved. The agent must trace the error through routing, middleware, business logic, and database layers.

What task decomposition approach would be most effective?

A. Have the agent first create a comprehensive plan mapping all code paths through the endpoint before beginning any file exploration or code reading.

B. Define a fixed sequence of investigation steps upfront—grep for error patterns, then read error handlers, then check database queries, then examine middleware—executing each step regardless of intermediate findings.

C. Run parallel worker agents that simultaneously investigate all four layers, then synthesize their findings to identify where the error originates.

D. Have the agent dynamically generate investigation subtasks based on what it discovers at each step, adapting its exploration plan as new information about the error path emerges.

ANSWER: D

Explanation:

Have the agent dynamically generate investigation subtasks based on what it discovers at each step, adapting its exploration plan as new information about the error path emerges. This is the most effective approach because an intermittent server error in an unfamiliar, large codebase is an open-ended investigation: the relevant execution path, owning modules, runtime conditions, and dependencies are not known in advance. The agent should begin with available evidence, such as the endpoint route, request logs, stack traces, error messages, or references found through search. It can then use tool results to form and test progressively narrower hypotheses—for example, following a route handler to a service call, identifying conditional middleware behavior, or inspecting a query and its surrounding error handling.

Anthropic recommends agentic systems for tasks where the number and sequence of steps cannot be reliably prescribed beforehand. Agents use tools in a loop, observe the resulting environmental feedback, and decide what action is most useful next. For coding investigations, this enables the process to follow the actual code and evidence rather than an assumed architecture. The agent should also maintain clear stopping criteria, record evidence for each hypothesis, and surface uncertainty or request additional telemetry when the available repository evidence cannot establish the cause. See Anthropic's Building Effective Agents guidance and the Claude Code overview.

QUESTION NO: 10

You are building developer productivity tools using the Claude Agent SDK. The agent helps engineers explore unfamiliar codebases, understand legacy systems, generate boilerplate code, and automate repetitive tasks. It uses the built-in tools (Read, Write, Bash, Grep, Glob) and integrates with Model Context Protocol (MCP) servers.

An engineer asks your agent to identify untested code paths in a legacy payment processing module spanning 45 files. After reading the first 8 source files, the agent's responses are becoming noticeably less accurate—it's forgetting previously discussed code patterns and hasn't yet located all test files or traced critical payment flows.

What's the most effective approach to complete this investigation?

A. Spawn subagents to investigate specific questions (e.g., "find all test files for payment processing," "trace refund flow dependencies") while the main agent coordinates findings and preserves high-level understanding.

B. Clear context with /clear , then selectively re-read only the most critical files discovered so far, writing key findings to a scratchpad file that persists between context resets.

C. Switch to using Grep to search for specific function names instead of reading full files, reducing the content loaded into context for remaining exploration.

D. Document all current findings in a summary report, clear context completely, then use that report as the sole reference for continuing the investigation.

ANSWER: A

Explanation:

Spawn subagents to investigate specific questions (e.g., “find all test files for payment processing,” “trace refund flow dependencies”) while the main agent coordinates findings and preserves high-level understanding. This is the most effective approach because the task naturally separates into bounded, evidence-driven investigations: discovering relevant tests, tracing payment and refund call chains, identifying conditionals and error handling, and mapping external service dependencies. Each subagent can use file-searching and code-reading tools within an isolated context, then return concise findings such as file paths, relevant symbols, uncovered branches, and supporting call-chain details. The coordinating agent retains enough context to compare results, resolve overlaps, identify gaps, and produce a coherent coverage assessment.

This architecture directly addresses degradation caused by loading large volumes of source material into one agent context. Rather than asking the primary agent to retain every implementation detail from dozens of files, it delegates focused exploration and consumes compact summaries. Anthropic documents that subagents operate with separate context windows and are useful for tasks that require focused investigation, while the parent agent manages delegation and synthesis. The Claude Code workflow guidance also recommends delegating independent research paths so findings can be combined efficiently. Clear subagent deliverables—especially concrete file references, branch conditions, and associated tests—make the final determination of untested payment paths auditable and actionable.

References: Anthropic Docs: Subagents; Anthropic Docs: Common workflows.

QUESTION NO: 11

You are building developer-productivity tools using the Claude Agent SDK. The agent helps engineers explore unfamiliar codebases, understand legacy systems, generate boilerplate code, and automate repetitive tasks. It uses the built-in tools—Read, Write, Bash, Grep, and Glob—and integrates with Model Context Protocol (MCP) servers.

After adding an MCP server with specialized code-refactoring tools—extract_function, rename_variable, and inline_function—you notice that the agent still uses basic text manipulation through Write and Bash sed commands for refactoring tasks. The MCP server is connected and healthy. Examining the configuration, you find that each MCP tool has a minimal description such as, “extract_function: Extracts a function from code.”

What is the most effective way to improve adoption of the MCP refactoring tools?

A. Implement a request classifier that detects refactoring intent and automatically routes those requests to the MCP server before the agent processes them.

B. Accept this as expected behavior because simpler tools such as sed are more predictable than specialized refactoring tools.

C. Enhance the MCP tool descriptions to explain when each tool is preferable to text manipulation and clarify expected inputs and outputs.

D. Remove the Write tool from the agent’s configuration for refactoring sessions so it must use the MCP tools for code modifications.

ANSWER: C

Explanation:

Enhance the MCP tool descriptions to explain when each tool is preferable to text manipulation and clarify expected inputs and outputs. Claude selects among available tools using the tool definitions supplied in its context, especially each tool’s name, description, and input schema. A short description such as “Extracts a function from code” provides too little decision-making guidance when general-purpose alternatives such as Write or Bash are also available. The model needs a clear indication that the refactoring tool performs syntax-aware or language-aware transformations, preserves code semantics where applicable, and should be preferred over string-based edits for the relevant operation.

Useful descriptions should state the tool’s purpose, ideal use cases, constraints, required identifiers or source locations, and the shape of its result. For example, an extract-function description can specify that it should be used to extract a selected contiguous code region into a new function instead of editing source text manually, and identify the file path, symbol/range, and desired function name as inputs. Detailed tool guidance improves tool selection without fragile external routing or unnecessarily removing general-purpose capabilities. Anthropic recommends clear, detailed tool descriptions that explain what a tool does and when to use it: Anthropic tool use documentation. MCP likewise defines tools with human-readable descriptions intended to help models understand their functionality: MCP tools specification.

QUESTION NO: 12

Your code-review prompts include both implementation changes and the corresponding test file, but the review comments fail to identify untested code paths. The model correctly flags functions that have no tests at all, but it fails to recognize when conditional branches or error-handling paths within tested functions lack coverage. What is the most effective way to improve branch-level gap detection without overcomplicating the pipeline?

A. Interleave the implementation and tests in the prompt, presenting each function immediately before its test cases.

B. Add explicit instructions requiring Claude to enumerate every conditional branch and exception path, then verify that each path has a corresponding test assertion.

C. Implement a two-pass pipeline in which one model call extracts all conditional branches and another cross-references them against test assertions.

D. Include few-shot examples showing code with an uncovered branch and the corresponding review comment identifying the missing test case.

ANSWER: B

Explanation:

Add explicit instructions requiring Claude to enumerate every conditional branch and exception path, then verify that each path has a corresponding test assertion. This directly addresses the observed failure by changing the review task from a high-level check for whether a function has tests into a structured coverage analysis. The instruction should have Claude identify decision points such as `if`/`else` conditions, switch alternatives, validation failures, early returns, thrown exceptions, and error handlers. For each identified path, Claude can then locate the test case and assertion that demonstrates the expected behavior, or report the specific path for which evidence is absent.

This approach is effective because it gives the model a concrete, repeatable evaluation procedure while retaining a single review call. The requested output can also be made auditable by requiring each finding to name the condition, the unexercised outcome, and the missing test behavior. Anthropic's guidance emphasizes writing clear, specific instructions for the desired task and using structured steps when a task requires systematic reasoning. See Anthropic's prompt engineering overview and its guidance on chain-of-thought prompting.

QUESTION NO: 13

You are building a customer support resolution agent using the Claude Agent SDK. The agent handles high-ambiguity requests like returns, billing disputes, and account issues. It has access to your backend systems through custom Model Context Protocol (MCP) tools (get_customer, lookup_order, process_refund, escalate_to_human). Your target is 80%+ first-contact resolution while knowing when to escalate.

After expanding the agent's MCP tools with delivery-specific capabilities (check_delivery_status, contact_driver, issue_credit, apply_promo_code, update_delivery_address, reschedule_delivery), the total tool count has grown from 4 to 10. Your evaluation suite shows tool selection accuracy has dropped from 88% to 71%. Log analysis reveals the majority of errors involve the agent selecting between semantically overlapping tools—calling issue_credit when process_refund was correct, and calling check_delivery_status when lookup_order already returns the needed data.

Which approach structurally eliminates the semantic overlap identified in the logs as the error source?

A. Split the tools across two sub-agents—a "financial resolution" agent with process_refund, issue_credit, and apply_promo_code, and a "delivery operations" agent with the remaining delivery tools—with a coordinator routing between them.

B. Consolidate semantically overlapping tools—merge issue_credit and process_refund into a single resolve_compensation tool with an action parameter, and fold check_delivery_status into lookup_order with an optional include_tracking flag.

C. Enable the tool search tool with defer_loading on the six new tools, keeping the original four always loaded, so the agent dynamically discovers specialized tools only when needed.

D. Add few-shot examples to the system prompt demonstrating correct selection for each ambiguous tool pair, such as showing when issue_credit applies versus when process_refund is appropriate.

ANSWER: B

Explanation:

Consolidate semantically overlapping tools—merge issue_credit and process_refund into a single resolve_compensation tool with an action parameter, and fold check_delivery_status into lookup_order with an optional include_tracking flag. This is correct because it changes the interface that Claude must reason over, rather than attempting to train around ambiguity. A unified compensation tool can expose a constrained `action` field, such as `refund` or `credit`, with descriptions, eligibility inputs, and validation rules that make the intended business operation explicit. Likewise, retrieving delivery tracking is naturally an optional facet of order lookup, so an `include_tracking` parameter avoids presenting two tools that can both appear to answer a delivery-status question.

Anthropic recommends designing tools around clear, distinct capabilities, with precise names, descriptions, and schemas. Reducing duplicate or competing affordances produces a smaller, more deterministic tool surface and lets the model select one business domain operation before supplying the specific requested action. This directly addresses the observed source of selection errors while preserving the backend functions needed to execute each outcome. The resulting MCP interface is easier to evaluate, document, authorize, and evolve as support workflows grow. See Anthropic's tool-use implementation guidance and the Model Context Protocol architecture documentation.

QUESTION NO: 14

You are building a customer support resolution agent using the Claude Agent SDK. The agent handles high-ambiguity requests like returns, billing disputes, and account issues. It has access to your backend systems through custom Model Context Protocol (MCP) tools (get_customer , lookup_order , process_refund , escalate_to_human). Your target is 80%+ first-contact resolution while knowing when to escalate.

During testing, you find that when a customer says "I need a refund for my recent purchase," the agent calls process_refund immediately—but populates the required order_id parameter with a plausible-looking but fabricated value instead of first calling lookup_order to retrieve the actual order ID. The refund call fails because the fabricated ID doesn't exist.

Which change directly addresses the root cause of the agent fabricating the order_id value?

A. Update the process_refund tool description to explicitly state that order_id must be obtained from a prior lookup_order call and must never be assumed or invented.

B. Switch tool_choice from " auto " to " any " to force the agent to make a tool call on every turn.

C. Add server-side validation that checks whether the order_id exists in your database before executing the refund, returning an error to the agent if not found.

D. Pre-parse incoming customer messages to extract any order IDs mentioned, and inject them into the conversation context before passing to Claude.

ANSWER: A

Explanation:

Update the `process_refund` tool description to explicitly state that `order_id` must be obtained from a prior `lookup_order` call and must never be assumed or invented. This directly fixes the missing instruction governing parameter provenance and the dependency between the two tools. The model needs a clear, actionable contract: it must identify the customer's actual order through the authoritative backend lookup before attempting a refund, and it must use only the identifier returned by that lookup. Specifying both the required sequence and the prohibition on guessed values gives Claude the information needed to select the appropriate tool workflow for an ambiguous request.

Anthropic's tool-use guidance emphasizes that tool definitions, particularly their descriptions, are important instructions for the model. Descriptions should make clear what a tool does, when to use it, and constraints that govern safe and correct operation. In this workflow, the prerequisite lookup and the trusted source of the identifier are essential constraints, so they belong in the tool contract. This improves reliable orchestration while preserving the agent's ability to ask for clarification or use `escalate_to_human` when the lookup cannot resolve the customer's purchase. See Anthropic's guidance on implementing tool use and defining tools.

QUESTION NO: 15

Your automated code review is missing genuine bugs in pull requests. Investigation reveals that the review prompt includes this instruction: “Only flag critical issues that would definitely cause production failures. Ignore minor concerns and anything you are uncertain about.” Developers confirm that some missed findings are genuine logic errors that the model investigated but chose not to report. The team requires the review output to remain structured, with every finding tagged with metadata, and actionable. Which prompt change both removes the cause of the suppressed findings and preserves structured, tagged output for downstream filtering?

A. Enable extended thinking and instruct the model to reason step by step about every code change before producing its review.

B. Instruct the model to report all findings with confidence and severity tags, deferring filtering to a downstream step.

C. Remove all severity-related instructions and allow the model to use its default judgment about which findings to report.

D. Add a second review pass that rereads the diff using the same prompt and looks for anything the first pass may have missed.

ANSWER: B

Explanation:

Instruct the model to report all findings with confidence and severity tags, deferring filtering to a downstream step. This directly removes the reporting constraint responsible for the missed defects: the model is no longer told to discard concerns based on a high certainty or production-impact threshold after it has identified them. Separating detection from filtering improves recall, because plausible logic errors and lower-severity issues can be emitted for review rather than silently suppressed.

Confidence and severity metadata preserve the information needed to make the output useful and manageable. A structured response can require each finding to include fields such as location, description, evidence, confidence, severity, and suggested remediation. Downstream automation or a human reviewer can then apply transparent, adjustable rules—for example, escalating high-severity findings immediately while routing lower-confidence findings for verification. This design keeps the detection stage broad while allowing acceptance criteria to be tuned independently without changing what the model is permitted to report.

Anthropic recommends using clear output formats and explicitly defining the information that should be returned, including structured formats such as JSON when systems need reliable downstream processing. See the Anthropic prompt engineering overview and Anthropic structured outputs documentation.

QUESTION NO: 16

You are building developer productivity tools using the Claude Agent SDK. The agent helps engineers explore unfamiliar codebases, understand legacy systems, generate boilerplate code, and automate repetitive tasks. It uses the built-in tools (Read, Write, Bash, Grep, Glob) and integrates with Model Context Protocol (MCP) servers.

1.5An engineer asks the agent to understand how the caching layer works before adding a new cache invalidation trigger. After initial Grep searches, the agent has identified that caching logic spans 15 files including decorators, middleware, and service classes (~6,000 lines total).

What’s the most effective next step for building understanding while managing context constraints?

A. Use Grep to search for “invalidate” and “expire” patterns across all files, then Read only those specific line ranges with minimal surrounding context.

B. Use the Read tool to sequentially load all 15 files, building complete understanding across the full caching implementation.

C. Use Glob to find files matching common caching patterns (cache*.py , caching/), prioritize the largest files by reading them first, then check smaller files for gaps.

D. Analyze imports and class hierarchies to identify the base cache class. Read that file to understand the interface, then trace specific invalidation implementations.

ANSWER: D

Explanation:

“Analyze imports and class hierarchies to identify the base cache class. Read that file to understand the interface, then trace specific invalidation implementations.” is the most effective approach because it builds an architectural model before spending context on implementation detail. A base cache abstraction typically establishes the contract for cache creation, lookup, eviction, invalidation, key handling, and lifecycle behavior. Starting there lets the agent determine which methods and extension points matter for a new invalidation trigger.

From that interface, the agent can use targeted searches to locate concrete implementations, subclasses, dependency-injection registrations, middleware integration points, and callers of the invalidation contract. This produces an evidence-driven reading sequence: each additional file is read because a relationship from the cache abstraction indicates that it is relevant. The agent therefore preserves room in its context for the code paths, configuration, and tests that directly affect the requested change.

This matches Anthropic guidance to investigate codebases with focused searches and incremental exploration rather than indiscriminately loading large amounts of source material. It also aligns with the agentic workflow of using tools to gather only the information needed for the current task. See Anthropic’s Claude Code best practices and the Claude Code overview.