Claude Certified Architect – Foundations Certification

Exam Guide

Introduction

The Claude Certified Architect – Foundations certification validates that practitioners can make informed decisions about tradeoffs when implementing real-world solutions with Claude. This exam tests foundational knowledge across Claude Code, the Claude Agent SDK, the Claude API, and Model Context Protocol (MCP) — the core technologies used to build production-grade applications with Claude.

Questions on this exam are grounded in realistic scenarios drawn from actual customer use cases, including building agentic systems for customer support, designing multi-agent research pipelines, integrating Claude Code into CI/CD workflows, building developer productivity tools, and extracting structured data from unstructured documents. Candidates must demonstrate not only conceptual knowledge but practical judgment about architecture, configuration, and tradeoffs in production deployments.

This guide describes the exam content, lists the domains and task statements tested, provides sample questions, and recommends preparation strategies. Use it alongside hands-on experience to prepare effectively.

Exam Details at a Glance

Credential	Claude Certified Architect – Foundations
Number of questions	60
Time limit	120 minutes
Response format	Multiple choice; one correct answer and three incorrect options
Exam structure	4 scenarios drawn from a bank of 6
Content domains	5 (weightings in the Content Outline)
Delivery	Online proctored or at a test center
Exam fee	$125 USD
Scoring	Scaled score of 100–1,000; minimum passing score 720
Validity period	12 months from the date the credential is awarded
Result reporting	Pass or fail

Target Candidate Description

The ideal candidate for this certification is a solution architect who designs and implements production applications with Claude. This candidate has hands-on experience with:

- Building agentic applications using the Claude Agent SDK, including multi-agent orchestration, subagent delegation, tool integration, and lifecycle hooks
- Configuring and customizing Claude Code for team workflows using CLAUDE.md files, Agent Skills, MCP server integrations, and plan mode
- Designing Model Context Protocol (MCP) tool and resource interfaces for backend system integration
- Engineering prompts that produce reliable structured output, leveraging JSON schemas, few-shot examples, and extraction patterns
- Managing context windows effectively across long documents, multi-turn conversations, and multi-agent handoffs

- Integrating Claude into CI/CD pipelines for automated code review, test generation, and pull request feedback
- Making sound escalation and reliability decisions, including error handling, human-in-the-loop workflows, and self-evaluation patterns

The candidate typically has 6+ months of practical experience building with Claude APIs, Agent SDK, Claude Code, and MCP, understanding both the capabilities and limitations of large language models in production environments.

Exam Content

Response Types

All questions on the exam are multiple choice format. Each question has one correct response and three incorrect responses.

Select the single response that best completes the statement or answers the question. The incorrect options are designed to be plausible to a candidate with incomplete knowledge or experience.

The exam platform requires an answer to every question before you can advance, so no question is left unanswered.

Exam Results

The exam has a pass or fail designation. The exam is scored against a minimum standard established by subject matter experts.

Your results are reported as a scaled score of 100–1,000. The minimum passing score is 720. Scaled scoring models help equate scores across multiple exam forms that might have slightly different difficulty levels.

Content Outline

This exam guide includes weightings, content domains, and task statements for the exam. The exam has the following content domains and weightings:

- Domain 1: Agentic Architecture & Orchestration (27% of scored content)
- Domain 2: Tool Design & MCP Integration (18% of scored content)
- Domain 3: Claude Code Configuration & Workflows (20% of scored content)
- Domain 4: Prompt Engineering & Structured Output (20% of scored content)

- Domain 5: Context Management & Reliability (15% of scored content)

Exam Scenarios

The exam uses scenario-based questions. Each scenario presents a realistic production context that frames a set of questions. During the exam, 4 scenarios will be presented and picked at random from the full set of the 6 scenarios below.

Scenario 1: Customer Support Resolution Agent

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.

Primary domains: Agentic Architecture & Orchestration, Tool Design & MCP Integration, Context Management & Reliability

Scenario 2: Code Generation with Claude Code

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.

Primary domains: Claude Code Configuration & Workflows, Context Management & Reliability

Scenario 3: Multi-Agent Research System

You are building a multi-agent research system using the Claude Agent SDK. A coordinator agent delegates to specialized subagents: one searches the web, one analyzes documents, one synthesizes findings, and one generates reports. The system researches topics and produces comprehensive, cited reports.

Primary domains: Agentic Architecture & Orchestration, Tool Design & MCP Integration, Context Management & Reliability

Scenario 4: Developer Productivity with Claude

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.

Primary domains: Tool Design & MCP Integration, Claude Code Configuration & Workflows, Agentic Architecture & Orchestration

Scenario 5: Claude Code for Continuous Integration

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.

Primary domains: Claude Code Configuration & Workflows, Prompt Engineering & Structured Output

Scenario 6: Structured Data Extraction

You are building a structured data extraction system using Claude. The system extracts information from unstructured documents, validates the output using JavaScript Object Notation (JSON) schemas, and maintains high accuracy. It must handle edge cases gracefully and integrate with downstream systems.

Primary domains: Prompt Engineering & Structured Output, Context Management & Reliability

Domain 1: Agentic Architecture & Orchestration

Task Statement 1.1: Design and implement agentic loops for autonomous task execution

Knowledge of:

- The agentic loop lifecycle: sending requests to Claude, inspecting stop_reason ("tool_use" vs "end_turn"), executing requested tools, and returning results for the next iteration
- How tool results are appended to conversation history so the model can reason about the next action
- The distinction between model-driven decision-making (Claude reasons about which tool to call next based on context) and pre-configured decision trees or tool sequences

Skills in:

- Implementing agentic loop control flow that continues when stop_reason is "tool_use" and terminates when stop_reason is "end_turn"
- Adding tool results to conversation context between iterations so the model can incorporate new information into its reasoning
- Avoiding anti-patterns such as parsing natural language signals to determine loop termination, setting arbitrary iteration caps as the primary stopping mechanism, or checking for assistant text content as a completion indicator

Task Statement 1.2: Orchestrate multi-agent systems with coordinator-subagent patterns

Knowledge of:

- Hub-and-spoke architecture where a coordinator agent manages all inter-subagent communication, error handling, and information routing
- How subagents operate with isolated context—they do not inherit the coordinator's conversation history automatically
- The role of the coordinator in task decomposition, delegation, result aggregation, and deciding which subagents to invoke based on query complexity
- Risks of overly narrow task decomposition by the coordinator, leading to incomplete coverage of broad research topics

Skills in:

- Designing coordinator agents that analyze query requirements and dynamically select which subagents to invoke rather than always routing through the full pipeline
- Partitioning research scope across subagents to minimize duplication (e.g., assigning distinct subtopics or source types to each agent)
- Implementing iterative refinement loops where the coordinator evaluates synthesis output for gaps, re-delegates to search and analysis subagents with targeted queries, and re-invokes synthesis until coverage is sufficient
- Routing all subagent communication through the coordinator for observability, consistent error handling, and controlled information flow

Task Statement 1.3: Configure subagent invocation, context passing, and spawning

Knowledge of:

- The Task tool as the mechanism for spawning subagents, and the requirement that allowedTools must include "Task" for a coordinator to invoke subagents
- That subagent context must be explicitly provided in the prompt—subagents do not automatically inherit parent context or share memory between invocations
- The AgentDefinition configuration including descriptions, system prompts, and tool restrictions for each subagent type
- Fork-based session management for exploring divergent approaches from a shared analysis baseline

Skills in:

- Including complete findings from prior agents directly in the subagent's prompt (e.g., passing web search results and document analysis outputs to the synthesis subagent)
- Using structured data formats to separate content from metadata (source URLs, document names, page numbers) when passing context between agents to preserve attribution
- Spawning parallel subagents by emitting multiple Task tool calls in a single coordinator response rather than across separate turns
- Designing coordinator prompts that specify research goals and quality criteria rather than step-by-step procedural instructions, to enable subagent adaptability

Task Statement 1.4: Implement multi-step workflows with enforcement and handoff patterns

Knowledge of:

- The difference between programmatic enforcement (hooks, prerequisite gates) and prompt-based guidance for workflow ordering
- When deterministic compliance is required (e.g., identity verification before financial operations), prompt instructions alone have a non-zero failure rate
- Structured handoff protocols for mid-process escalation that include customer details, root cause analysis, and recommended actions

Skills in:

- Implementing programmatic prerequisites that block downstream tool calls until prerequisite steps have completed (e.g., blocking process_refund until get_customer has returned a verified customer ID)

- Decomposing multi-concern customer requests into distinct items, then investigating each in parallel using shared context before synthesizing a unified resolution
- Compiling structured handoff summaries (customer ID, root cause, refund amount, recommended action) when escalating to human agents who lack access to the conversation transcript

Task Statement 1.5: Apply Agent SDK hooks for tool call interception and data normalization

Knowledge of:

- Hook patterns (e.g., PostToolUse) that intercept tool results for transformation before the model processes them
- Hook patterns that intercept outgoing tool calls to enforce compliance rules (e.g., blocking refunds above a threshold)
- The distinction between using hooks for deterministic guarantees versus relying on prompt instructions for probabilistic compliance

Skills in:

- Implementing PostToolUse hooks to normalize heterogeneous data formats (Unix timestamps, ISO 8601, numeric status codes) from different MCP tools before the agent processes them
- Implementing tool call interception hooks that block policy-violating actions (e.g., refunds exceeding $500) and redirect to alternative workflows (e.g., human escalation)
- Choosing hooks over prompt-based enforcement when business rules require guaranteed compliance

Task Statement 1.6: Design task decomposition strategies for complex workflows

Knowledge of:

- When to use fixed sequential pipelines (prompt chaining) versus dynamic adaptive decomposition based on intermediate findings
- Prompt chaining patterns that break reviews into sequential steps (e.g., analyze each file individually, then run a cross-file integration pass)
- The value of adaptive investigation plans that generate subtasks based on what is discovered at each step

Skills in:

- Selecting task decomposition patterns appropriate to the workflow: prompt chaining for predictable multi-aspect reviews, dynamic decomposition for open-ended investigation tasks

- Splitting large code reviews into per-file local analysis passes plus a separate cross-file integration pass to avoid attention dilution
- Decomposing open-ended tasks (e.g., "add comprehensive tests to a legacy codebase") by first mapping structure, identifying high-impact areas, then creating a prioritized plan that adapts as dependencies are discovered

Task Statement 1.7: Manage session state, resumption, and forking

Knowledge of:

- Named session resumption using --resume <session-name> to continue a specific prior conversation
- fork_session for creating independent branches from a shared analysis baseline to explore divergent approaches
- The importance of informing the agent about changes to previously analyzed files when resuming sessions after code modifications
- Why starting a new session with a structured summary is more reliable than resuming with stale tool results

Skills in:

- Using --resume with session names to continue named investigation sessions across work sessions
- Using fork_session to create parallel exploration branches (e.g., comparing two testing strategies or refactoring approaches from a shared codebase analysis)
- Choosing between session resumption (when prior context is mostly valid) and starting fresh with injected summaries (when prior tool results are stale)
- Informing a resumed session about specific file changes for targeted re-analysis rather than requiring full re-exploration

Domain 2: Tool Design & MCP Integration

Task Statement 2.1: Design effective tool interfaces with clear descriptions and boundaries

Knowledge of:

- Tool descriptions as the primary mechanism LLMs use for tool selection; minimal descriptions lead to unreliable selection among similar tools

- The importance of including input formats, example queries, edge cases, and boundary explanations in tool descriptions
- How ambiguous or overlapping tool descriptions cause misrouting (e.g., analyze_content vs analyze_document with near-identical descriptions)
- The impact of system prompt wording on tool selection: keyword-sensitive instructions can create unintended tool associations

Skills in:

- Writing tool descriptions that clearly differentiate each tool's purpose, expected inputs, outputs, and when to use it versus similar alternatives
- Renaming tools and updating descriptions to eliminate functional overlap (e.g., renaming analyze_content to extract_web_results with a web-specific description)
- Splitting generic tools into purpose-specific tools with defined input/output contracts (e.g., splitting a generic analyze_document into extract_data_points,

summarize_content, and verify_claim_against_source)

- Reviewing system prompts for keyword-sensitive instructions that might override well-written tool descriptions

Task Statement 2.2: Implement structured error responses for MCP tools

Knowledge of:

- The MCP isError flag pattern for communicating tool failures back to the agent
- The distinction between transient errors (timeouts, service unavailability), validation errors (invalid input), business errors (policy violations), and permission errors
- Why uniform error responses (generic "Operation failed") prevent the agent from making appropriate recovery decisions
- The difference between retryable and non-retryable errors, and how returning structured metadata prevents wasted retry attempts

Skills in:

- Returning structured error metadata including errorCategory (transient/validation/permission), isRetryable boolean, and human-readable descriptions
- Including retriable: false flags and customer-friendly explanations for business rule violations so the agent can communicate appropriately
- Implementing local error recovery within subagents for transient failures, propagating to the coordinator only errors that cannot be resolved locally along with partial results and what was attempted

- Distinguishing between access failures (needing retry decisions) and valid empty results (representing successful queries with no matches)

Task Statement 2.3: Distribute tools appropriately across agents and configure tool choice

Knowledge of:

- The principle that giving an agent access to too many tools (e.g., 18 instead of 4-5) degrades tool selection reliability by increasing decision complexity
- Why agents with tools outside their specialization tend to misuse them (e.g., a synthesis agent attempting web searches)
- Scoped tool access: giving agents only the tools needed for their role, with limited cross-role tools for specific high-frequency needs
- tool_choice configuration options: "auto", "any", and forced tool selection ({"type": "tool", "name": "..."})

Skills in:

- Restricting each subagent's tool set to those relevant to its role, preventing cross-specialization misuse
- Replacing generic tools with constrained alternatives (e.g., replacing fetch_url with load_document that validates document URLs)
- Providing scoped cross-role tools for high-frequency needs (e.g., a verify_fact tool for the synthesis agent) while routing complex cases through the coordinator
- Using tool_choice forced selection to ensure a specific tool is called first (e.g., forcing extract_metadata before enrichment tools), then processing subsequent steps in follow-up turns
- Setting tool_choice: "any" to guarantee the model calls a tool rather than returning conversational text

Task Statement 2.4: Integrate MCP servers into Claude Code and agent workflows

Knowledge of:

- MCP server scoping: project-level (.mcp.json) for shared team tooling vs user-level (~/.claude.json) for personal/experimental servers
- Environment variable expansion in .mcp.json (e.g., ${GITHUB_TOKEN}) for credential management without committing secrets
- That tools from all configured MCP servers are discovered at connection time and available simultaneously to the agent

- MCP resources as a mechanism for exposing content catalogs (e.g., issue summaries, documentation hierarchies, database schemas) to reduce exploratory tool calls

Skills in:

- Configuring shared MCP servers in project-scoped .mcp.json with environment variable expansion for authentication tokens
- Configuring personal/experimental MCP servers in user-scoped ~/.claude.json
- Enhancing MCP tool descriptions to explain capabilities and outputs in detail, preventing the agent from preferring built-in tools (like Grep) over more capable MCP tools
- Choosing existing community MCP servers over custom implementations for standard integrations (e.g., Jira), reserving custom servers for team-specific workflows
- Exposing content catalogs as MCP resources to give agents visibility into available data without requiring exploratory tool calls

Task Statement 2.5: Select and apply built-in tools (Read, Write, Edit, Bash, Grep, Glob) effectively

Knowledge of:

- Grep for content search (searching file contents for patterns like function names, error messages, or import statements)
- Glob for file path pattern matching (finding files by name or extension patterns)
- Read/Write for full file operations; Edit for targeted modifications using unique text matching
- When Edit fails due to non-unique text matches, using Read + Write as a fallback for reliable file modifications

Skills in:

- Selecting Grep for searching code content across a codebase (e.g., finding all callers of a function, locating error messages)
- Selecting Glob for finding files matching naming patterns (e.g., **/*.test.tsx)
- Using Read to load full file contents followed by Write when Edit cannot find unique anchor text
- Building codebase understanding incrementally: starting with Grep to find entry points, then using Read to follow imports and trace flows, rather than reading all files upfront
- Tracing function usage across wrapper modules by first identifying all exported names, then searching for each name across the codebase

Domain 3: Claude Code Configuration & Workflows

Task Statement 3.1: Configure CLAUDE.md files with appropriate hierarchy, scoping, and modular organization

Knowledge of:

- The CLAUDE.md configuration hierarchy: user-level (~/.claude/CLAUDE.md), project-level (.claude/CLAUDE.md or root CLAUDE.md), and directory-level (subdirectory CLAUDE.md files)
- That user-level settings apply only to that user—instructions in ~/.claude/CLAUDE.md are not shared with teammates via version control
- The @import syntax for referencing external files to keep CLAUDE.md modular (e.g., importing specific standards files relevant to each package)
- .claude/rules/ directory for organizing topic-specific rule files as an alternative to a monolithic CLAUDE.md

Skills in:

- Diagnosing configuration hierarchy issues (e.g., a new team member not receiving instructions because they're in user-level rather than project-level configuration)
- Using @import to selectively include relevant standards files in each package's CLAUDE.md based on maintainer domain knowledge
- Splitting large CLAUDE.md files into focused topic-specific files in .claude/rules/ (e.g., testing.md, api-conventions.md, deployment.md)
- Using the /memory command to verify which memory files are loaded and diagnose inconsistent behavior across sessions

Task Statement 3.2: Create and configure custom slash commands and skills

Knowledge of:

- Project-scoped commands in .claude/commands/ (shared via version control) vs user-scoped commands in ~/.claude/commands/ (personal)
- Skills in .claude/skills/ with SKILL.md files that support frontmatter configuration including context: fork, allowed-tools, and argument-hint
- The context: fork frontmatter option for running skills in an isolated sub-agent context, preventing skill outputs from polluting the main conversation
- Personal skill customization: creating personal variants in ~/.claude/skills/ with different names to avoid affecting teammates

Skills in:

- Creating project-scoped slash commands in .claude/commands/ for team-wide availability via version control
- Using context: fork to isolate skills that produce verbose output (e.g., codebase analysis) or exploratory context (e.g., brainstorming alternatives) from the main session
- Configuring allowed-tools in skill frontmatter to restrict tool access during skill execution (e.g., limiting to file write operations to prevent destructive actions)
- Using argument-hint frontmatter to prompt developers for required parameters when they invoke the skill without arguments
- Choosing between skills (on-demand invocation for task-specific workflows) and CLAUDE.md (always-loaded universal standards)

Task Statement 3.3: Apply path-specific rules for conditional convention loading

Knowledge of:

- .claude/rules/ files with YAML frontmatter paths fields containing glob patterns for conditional rule activation
- How path-scoped rules load only when editing matching files, reducing irrelevant context and token usage
- The advantage of glob-pattern rules over directory-level CLAUDE.md files for conventions that span multiple directories (e.g., test files spread throughout a codebase)

Skills in:

- Creating .claude/rules/ files with YAML frontmatter path scoping (e.g., paths: ["terraform/**/*"]) so rules load only when editing matching files
- Using glob patterns in path-specific rules to apply conventions to files by type regardless of directory location (e.g., **/*.test.tsx for all test files)
- Choosing path-specific rules over subdirectory CLAUDE.md files when conventions must apply to files spread across the codebase

Task Statement 3.4: Determine when to use plan mode vs direct execution

Knowledge of:

- Plan mode is designed for complex tasks involving large-scale changes, multiple valid approaches, architectural decisions, and multi-file modifications
- Direct execution is appropriate for simple, well-scoped changes (e.g., adding a single validation check to one function)

- Plan mode enables safe codebase exploration and design before committing to changes, preventing costly rework
- The Explore subagent for isolating verbose discovery output and returning summaries to preserve main conversation context

Skills in:

- Selecting plan mode for tasks with architectural implications (e.g., microservice restructuring, library migrations affecting 45+ files, choosing between integration approaches with different infrastructure requirements)
- Selecting direct execution for well-understood changes with clear scope (e.g., a single-file bug fix with a clear stack trace, adding a date validation conditional)
- Using the Explore subagent for verbose discovery phases to prevent context window exhaustion during multi-phase tasks
- Combining plan mode for investigation with direct execution for implementation (e.g., planning a library migration, then executing the planned approach)

Task Statement 3.5: Apply iterative refinement techniques for progressive improvement

Knowledge of:

- Concrete input/output examples as the most effective way to communicate expected transformations when prose descriptions are interpreted inconsistently
- Test-driven iteration: writing test suites first, then iterating by sharing test failures to guide progressive improvement
- The interview pattern: having Claude ask questions to surface considerations the developer may not have anticipated before implementing
- When to provide all issues in a single message (interacting problems) versus fixing them sequentially (independent problems)

Skills in:

- Providing 2-3 concrete input/output examples to clarify transformation requirements when natural language descriptions produce inconsistent results
- Writing test suites covering expected behavior, edge cases, and performance requirements before implementation, then iterating by sharing test failures
- Using the interview pattern to surface design considerations (e.g., cache invalidation strategies, failure modes) before implementing solutions in unfamiliar domains

- Providing specific test cases with example input and expected output to fix edge case handling (e.g., null values in migration scripts)
- Addressing multiple interacting issues in a single detailed message when fixes interact, versus sequential iteration for independent issues

Task Statement 3.6: Integrate Claude Code into CI/CD pipelines

Knowledge of:

- The -p (or --print) flag for running Claude Code in non-interactive mode in automated pipelines
- --output-format json and --json-schema CLI flags for enforcing structured output in CI contexts
- CLAUDE.md as the mechanism for providing project context (testing standards, fixture conventions, review criteria) to CI-invoked Claude Code
- Session context isolation: why the same Claude session that generated code is less effective at reviewing its own changes compared to an independent review instance

Skills in:

- Running Claude Code in CI with the -p flag to prevent interactive input hangs
- Using --output-format json with --json-schema to produce machine-parseable structured findings for automated posting as inline PR comments
- Including prior review findings in context when re-running reviews after new commits, instructing Claude to report only new or still-unaddressed issues to avoid duplicate comments
- Providing existing test files in context so test generation avoids suggesting duplicate scenarios already covered by the test suite
- Documenting testing standards, valuable test criteria, and available fixtures in CLAUDE.md to improve test generation quality and reduce low-value test output

Domain 4: Prompt Engineering & Structured Output

Task Statement 4.1: Design prompts with explicit criteria to improve precision and reduce false positives

Knowledge of:

- The importance of explicit criteria over vague instructions (e.g., "flag comments only when claimed behavior contradicts actual code behavior" vs "check that comments are accurate")

- How general instructions like "be conservative" or "only report high-confidence findings" fail to improve precision compared to specific categorical criteria
- The impact of false positive rates on developer trust: high false positive categories undermine confidence in accurate categories

Skills in:

- Writing specific review criteria that define which issues to report (bugs, security) versus skip (minor style, local patterns) rather than relying on confidence-based filtering
- Temporarily disabling high false-positive categories to restore developer trust while improving prompts for those categories
- Defining explicit severity criteria with concrete code examples for each severity level to achieve consistent classification

Task Statement 4.2: Apply few-shot prompting to improve output consistency and quality

Knowledge of:

- Few-shot examples as the most effective technique for achieving consistently formatted, actionable output when detailed instructions alone produce inconsistent results
- The role of few-shot examples in demonstrating ambiguous-case handling (e.g., tool selection for ambiguous requests, branch-level test coverage gaps)
- How few-shot examples enable the model to generalize judgment to novel patterns rather than matching only pre-specified cases
- The effectiveness of few-shot examples for reducing hallucination in extraction tasks (e.g., handling informal measurements, varied document structures)

Skills in:

- Creating 2-4 targeted few-shot examples for ambiguous scenarios that show reasoning for why one action was chosen over plausible alternatives
- Including few-shot examples that demonstrate specific desired output format (location, issue, severity, suggested fix) to achieve consistency
- Providing few-shot examples distinguishing acceptable code patterns from genuine issues to reduce false positives while enabling generalization
- Using few-shot examples to demonstrate correct handling of varied document structures (inline citations vs bibliographies, methodology sections vs embedded details)
- Adding few-shot examples showing correct extraction from documents with varied formats to address empty/null extraction of required fields

Task Statement 4.3: Enforce structured output using tool use and JSON schemas

Knowledge of:

- Tool use (tool_use) with JSON schemas as the most reliable approach for guaranteed schema-compliant structured output, eliminating JSON syntax errors
- The distinction between tool_choice: "auto" (model may return text instead of calling a tool), "any" (model must call a tool but can choose which), and forced tool selection (model must call a specific named tool)
- That strict JSON schemas via tool use eliminate syntax errors but do not prevent semantic errors (e.g., line items that don't sum to total, values in wrong fields)
- Schema design considerations: required vs optional fields, enum fields with "other" + detail string patterns for extensible categories

Skills in:

- Defining extraction tools with JSON schemas as input parameters and extracting structured data from the tool_use response
- Setting tool_choice: "any" to guarantee structured output when multiple extraction schemas exist and the document type is unknown
- Forcing a specific tool with tool_choice: {"type": "tool", "name": "extract_metadata"} to ensure a particular extraction runs before enrichment steps
- Designing schema fields as optional (nullable) when source documents may not contain the information, preventing the model from fabricating values to satisfy required fields
- Adding enum values like "unclear" for ambiguous cases and "other" + detail fields for extensible categorization
- Including format normalization rules in prompts alongside strict output schemas to handle inconsistent source formatting

Task Statement 4.4: Implement validation, retry, and feedback loops for extraction quality

Knowledge of:

- Retry-with-error-feedback: appending specific validation errors to the prompt on retry to guide the model toward correction
- The limits of retry: retries are ineffective when the required information is simply absent from the source document (vs format or structural errors)
- Feedback loop design: tracking which code constructs trigger findings (detected_pattern field) to enable systematic analysis of dismissal patterns

- The difference between semantic validation errors (values don't sum, wrong field placement) and schema syntax errors (eliminated by tool use)

Skills in:

- Implementing follow-up requests that include the original document, the failed extraction, and specific validation errors for model self-correction
- Identifying when retries will be ineffective (e.g., information exists only in an external document not provided) versus when they will succeed (format mismatches, structural output errors)
- Adding detected_pattern fields to structured findings to enable analysis of false positive patterns when developers dismiss findings
- Designing self-correction validation flows: extracting "calculated_total" alongside "stated_total" to flag discrepancies, adding "conflict_detected" booleans for inconsistent source data

Task Statement 4.5: Design efficient batch processing strategies

Knowledge of:

- The Message Batches API: 50% cost savings, up to 24-hour processing window, no guaranteed latency SLA
- Batch processing is appropriate for non-blocking, latency-tolerant workloads (overnight reports, weekly audits, nightly test generation) and inappropriate for blocking workflows (pre-merge checks)
- The batch API does not support multi-turn tool calling within a single request (cannot execute tools mid-request and return results)
- custom_id fields for correlating batch request/response pairs

Skills in:

- Matching API approach to workflow latency requirements: synchronous API for blocking pre-merge checks, batch API for overnight/weekly analysis
- Calculating batch submission frequency based on SLA constraints (e.g., 4-hour windows to guarantee 30-hour SLA with 24-hour batch processing)
- Handling batch failures: resubmitting only failed documents (identified by custom_id) with appropriate modifications (e.g., chunking documents that exceeded context limits)
- Using prompt refinement on a sample set before batch-processing large volumes to maximize first-pass success rates and reduce iterative resubmission costs

Task Statement 4.6: Design multi-instance and multi-pass review architectures

Knowledge of:

- Self-review limitations: a model retains reasoning context from generation, making it less likely to question its own decisions in the same session
- Independent review instances (without prior reasoning context) are more effective at catching subtle issues than self-review instructions or extended thinking
- Multi-pass review: splitting large reviews into per-file local analysis passes plus cross-file integration passes to avoid attention dilution and contradictory findings

Skills in:

- Using a second independent Claude instance to review generated code without the generator's reasoning context
- Splitting large multi-file reviews into focused per-file passes for local issues plus separate integration passes for cross-file data flow analysis
- Running verification passes where the model self-reports confidence alongside each finding to enable calibrated review routing

Domain 5: Context Management & Reliability

Task Statement 5.1: Manage conversation context to preserve critical information across long interactions

Knowledge of:

- Progressive summarization risks: condensing numerical values, percentages, dates, and customer-stated expectations into vague summaries
- The "lost in the middle" effect: models reliably process information at the beginning and end of long inputs but may omit findings from middle sections
- How tool results accumulate in context and consume tokens disproportionately to their relevance (e.g., 40+ fields per order lookup when only 5 are relevant)
- The importance of passing complete conversation history in subsequent API requests to maintain conversational coherence

Skills in:

- Extracting transactional facts (amounts, dates, order numbers, statuses) into a persistent "case facts" block included in each prompt, outside summarized history

- Extracting and persisting structured issue data (order IDs, amounts, statuses) into a separate context layer for multi-issue sessions
- Trimming verbose tool outputs to only relevant fields before they accumulate in context (e.g., keeping only return-relevant fields from order lookups)
- Placing key findings summaries at the beginning of aggregated inputs and organizing detailed results with explicit section headers to mitigate position effects
- Requiring subagents to include metadata (dates, source locations, methodological context) in structured outputs to support accurate downstream synthesis
- Modifying upstream agents to return structured data (key facts, citations, relevance scores) instead of verbose content and reasoning chains when downstream agents have limited context budgets

Task Statement 5.2: Design effective escalation and ambiguity resolution patterns

Knowledge of:

- Appropriate escalation triggers: customer requests for a human, policy exceptions/gaps (not just complex cases), and inability to make meaningful progress
- The distinction between escalating immediately when a customer explicitly demands it versus offering to resolve when the issue is straightforward
- Why sentiment-based escalation and self-reported confidence scores are unreliable proxies for actual case complexity
- How multiple customer matches require clarification (requesting additional identifiers) rather than heuristic selection

Skills in:

- Adding explicit escalation criteria with few-shot examples to the system prompt demonstrating when to escalate versus resolve autonomously
- Honoring explicit customer requests for human agents immediately without first attempting investigation
- Acknowledging frustration while offering resolution when the issue is within the agent's capability, escalating only if the customer reiterates their preference
- Escalating when policy is ambiguous or silent on the customer's specific request (e.g., competitor price matching when policy only addresses own-site adjustments)
- Instructing the agent to ask for additional identifiers when tool results return multiple matches, rather than selecting based on heuristics

Task Statement 5.3: Implement error propagation strategies across multi-agent systems

Knowledge of:

- Structured error context (failure type, attempted query, partial results, alternative approaches) as enabling intelligent coordinator recovery decisions
- The distinction between access failures (timeouts needing retry decisions) and valid empty results (successful queries with no matches)
- Why generic error statuses ("search unavailable") hide valuable context from the coordinator
- Why silently suppressing errors (returning empty results as success) or terminating entire workflows on single failures are both anti-patterns

Skills in:

- Returning structured error context including failure type, what was attempted, partial results, and potential alternatives to enable coordinator recovery
- Distinguishing access failures from valid empty results in error reporting so the coordinator can make appropriate decisions
- Having subagents implement local recovery for transient failures and only propagate errors they cannot resolve, including what was attempted and partial results
- Structuring synthesis output with coverage annotations indicating which findings are well-supported versus which topic areas have gaps due to unavailable sources

Task Statement 5.4: Manage context effectively in large codebase exploration

Knowledge of:

- Context degradation in extended sessions: models start giving inconsistent answers and referencing "typical patterns" rather than specific classes discovered earlier
- The role of scratchpad files for persisting key findings across context boundaries
- Subagent delegation for isolating verbose exploration output while the main agent coordinates high-level understanding
- Structured state persistence for crash recovery: each agent exports state to a known location, and the coordinator loads a manifest on resume

Skills in:

- Spawning subagents to investigate specific questions (e.g., "find all test files," "trace refund flow dependencies") while the main agent preserves high-level coordination
- Having agents maintain scratchpad files recording key findings, referencing them for subsequent questions to counteract context degradation

- Summarizing key findings from one exploration phase before spawning sub-agents for the next phase, injecting summaries into initial context
- Designing crash recovery using structured agent state exports (manifests) that the coordinator loads on resume and injects into agent prompts
- Using /compact to reduce context usage during extended exploration sessions when context fills with verbose discovery output

Task Statement 5.5: Design human review workflows and confidence calibration

Knowledge of:

- The risk that aggregate accuracy metrics (e.g., 97% overall) may mask poor performance on specific document types or fields
- Stratified random sampling for measuring error rates in high-confidence extractions and detecting novel error patterns
- Field-level confidence scores calibrated using labeled validation sets for routing review attention
- The importance of validating accuracy by document type and field segment before automating high-confidence extractions

Skills in:

- Implementing stratified random sampling of high-confidence extractions for ongoing error rate measurement and novel pattern detection
- Analyzing accuracy by document type and field to verify consistent performance across all segments before reducing human review
- Having models output field-level confidence scores, then calibrating review thresholds using labeled validation sets
- Routing extractions with low model confidence or ambiguous/contradictory source documents to human review, prioritizing limited reviewer capacity

Task Statement 5.6: Preserve information provenance and handle uncertainty in multi-source synthesis

Knowledge of:

- How source attribution is lost during summarization steps when findings are compressed without preserving claim-source mappings
- The importance of structured claim-source mappings that the synthesis agent must preserve and merge when combining findings

- How to handle conflicting statistics from credible sources: annotating conflicts with source attribution rather than arbitrarily selecting one value
- Temporal data: requiring publication/collection dates in structured outputs to prevent temporal differences from being misinterpreted as contradictions

Skills in:

- Requiring subagents to output structured claim-source mappings (source URLs, document names, relevant excerpts) that downstream agents preserve through synthesis
- Structuring reports with explicit sections distinguishing well-established findings from contested ones, preserving original source characterizations and methodological context
- Completing document analysis with conflicting values included and explicitly annotated, letting the coordinator decide how to reconcile before passing to synthesis
- Requiring subagents to include publication or data collection dates in structured outputs to enable correct temporal interpretation
- Rendering different content types appropriately in synthesis outputs—financial data as tables, news as prose, technical findings as structured lists—rather than converting everything to a uniform format

Sample Questions

The following sample questions illustrate the format and difficulty level of the exam. These are drawn from the practice test and include explanations to aid learning.

Scenario: Customer Support Resolution Agent

Question 1: Production data shows that in 12% of cases, your agent skips get_customer entirely and calls lookup_order using only the customer's stated name, occasionally leading to misidentified accounts and incorrect refunds. What change would most effectively address this reliability issue?

A) Add a programmatic prerequisite that blocks lookup_order and process_refund calls until get_customer has returned a verified customer ID.

B) Enhance the system prompt to state that customer verification via get_customer is mandatory before any order operations.

C) Add few-shot examples showing the agent always calling get_customer first, even when customers volunteer order details.

D) Implement a routing classifier that analyzes each request and enables only the subset of tools appropriate for that request type.

Correct Answer: A

When a specific tool sequence is required for critical business logic (like verifying customer identity before processing refunds), programmatic enforcement provides deterministic guarantees that prompt-based approaches cannot. Options B and C rely on probabilistic LLM compliance, which is insufficient when errors have financial consequences. Option D addresses tool availability rather than tool ordering, which is not the actual problem.

Question 2: Production logs show the agent frequently calls get_customer when users ask about orders (e.g., "check my order #12345"), instead of calling lookup_order. Both tools have minimal descriptions ("Retrieves customer information" / "Retrieves order details") and accept similar identifier formats. What's the most effective first step to improve tool selection reliability?

A) Add few-shot examples to the system prompt demonstrating correct tool selection patterns, with 5-8 examples showing order-related queries routing to lookup_order.

B) Expand each tool's description to include input formats it handles, example queries, edge cases, and boundaries explaining when to use it versus similar tools.

C) Implement a routing layer that parses user input before each turn and pre-selects the appropriate tool based on detected keywords and identifier patterns.

D) Consolidate both tools into a single lookup_entity tool that accepts any identifier and internally determines which backend to query.

Correct Answer: B

Tool descriptions are the primary mechanism LLMs use for tool selection. When descriptions are minimal, models lack the context to differentiate between similar tools. Option B directly addresses this root cause with a low-effort, high-leverage fix. Few-shot examples (A) add token overhead without fixing the underlying issue. A routing layer (C) is over-engineered and bypasses the LLM's natural language understanding. Consolidating tools (D) is a valid architectural choice but requires more effort than a "first step" warrants when the immediate problem is inadequate descriptions.

Question 3: Your agent achieves 55% first-contact resolution, well below the 80% target. Logs show it escalates straightforward cases (standard damage replacements with photo evidence) while attempting to autonomously handle complex situations requiring policy exceptions. What's the most effective way to improve escalation calibration?

A) Add explicit escalation criteria to your system prompt with few-shot examples demonstrating when to escalate versus resolve autonomously.

B) Have the agent self-report a confidence score (1-10) before each response and automatically route requests to humans when confidence falls below a threshold.

C) Deploy a separate classifier model trained on historical tickets to predict which requests need escalation before the main agent begins processing.

D) Implement sentiment analysis to detect customer frustration levels and automatically escalate when negative sentiment exceeds a threshold.

Correct Answer: A

Adding explicit escalation criteria with few-shot examples directly addresses the root cause: unclear decision boundaries. This is the proportionate first response before adding infrastructure. Option B fails because LLM self-reported confidence is poorly calibrated—the agent is already incorrectly confident on hard cases. Option C is over-engineered, requiring labeled data and ML infrastructure when prompt optimization hasn't been tried. Option D solves a different problem entirely; sentiment doesn't correlate with case complexity, which is the actual issue.

Scenario: Code Generation with Claude Code

Question 4: You want to create a custom /review slash command that runs your team's standard code review checklist. This command should be available to every developer when they clone or pull the repository. Where should you create this command file?

A) In the .claude/commands/ directory in the project repository

B) In ~/.claude/commands/ in each developer's home directory

C) In the CLAUDE.md file at the project root

D) In a .claude/config.json file with a commands array

Correct Answer: A

Project-scoped custom slash commands should be stored in the .claude/commands/ directory within the repository. These commands are version-controlled and automatically available to all developers when they clone or pull the repo. Option B (~/.claude/commands/) is for personal commands that aren't shared via version control. Option C (CLAUDE.md) is for project instructions and context, not command definitions. Option D describes a configuration mechanism that doesn't exist in Claude Code.

Question 5: You've been assigned to restructure the team's monolithic application into microservices. This will involve changes across dozens of files and requires decisions about service boundaries and module dependencies. Which approach should you take?

A) Enter plan mode to explore the codebase, understand dependencies, and design an implementation approach before making changes.

B) Start with direct execution and make changes incrementally, letting the implementation reveal the natural service boundaries.

C) Use direct execution with comprehensive upfront instructions detailing exactly how each service should be structured.

D) Begin in direct execution mode and only switch to plan mode if you encounter unexpected complexity during implementation.

Correct Answer: A

Plan mode is designed for complex tasks involving large-scale changes, multiple valid approaches, and architectural decisions—exactly what monolith-to-microservices restructuring requires. It enables safe codebase exploration and design before committing to changes. Option B risks costly rework when dependencies are discovered late. Option C assumes you already know the right structure without exploring the code. Option D ignores that the complexity is already stated in the requirements, not something that might emerge later.

Question 6: Your codebase has distinct areas with different coding conventions: React components use functional style with hooks, API handlers use async/await with specific error handling, and database models follow a repository pattern. Test files are spread throughout the codebase alongside the code they test (e.g., Button.test.tsx next to Button.tsx), and you want all tests to follow the same conventions regardless of location. What's the most maintainable way to ensure Claude automatically applies the correct conventions when generating code?

A) Create rule files in .claude/rules/ with YAML frontmatter specifying glob patterns to conditionally apply conventions based on file paths

B) Consolidate all conventions in the root CLAUDE.md file under headers for each area, relying on Claude to infer which section applies

C) Create skills in .claude/skills/ for each code type that include the relevant conventions in their SKILL.md files

D) Place a separate CLAUDE.md file in each subdirectory containing that area's specific conventions

Correct Answer: A

Option A is correct because .claude/rules/ with glob patterns (e.g., **/*.test.tsx) allows conventions to be automatically applied based on file paths regardless of directory location— essential for test files spread throughout the codebase. Option B relies on inference rather than explicit matching, making it unreliable. Option C requires manual skill invocation or relies on Claude choosing to load them, contradicting the need for deterministic "automatic" application based on file paths. Option D can't easily handle files spread across many directories since CLAUDE.md files are directory-bound.

Scenario: Multi-Agent Research System

Question 7: After running the system on the topic "impact of AI on creative industries," you observe that each subagent completes successfully: the web search agent finds relevant articles, the document analysis agent summarizes papers correctly, and the synthesis agent produces coherent output. However, the final reports cover only visual arts, completely missing music, writing, and film production. When you examine the coordinator's logs, you see it decomposed the topic into three subtasks: "AI in digital art creation," "AI in graphic design," and "AI in photography." What is the most likely root cause?

A) The synthesis agent lacks instructions for identifying coverage gaps in the findings it receives from other agents.

B) The coordinator agent's task decomposition is too narrow, resulting in subagent assignments that don't cover all relevant domains of the topic.

C) The web search agent's queries are not comprehensive enough and need to be expanded to cover more creative industry sectors.

D) The document analysis agent is filtering out sources related to non-visual creative industries due to overly restrictive relevance criteria.

Correct Answer: B

The coordinator's logs reveal the root cause directly: it decomposed "creative industries" into only visual arts subtasks (digital art, graphic design, photography), completely omitting music, writing, and film. The subagents executed their assigned tasks correctly—the problem is what they were assigned. Options A, C, and D incorrectly blame downstream agents that are working correctly within their assigned scope.

Question 8: The web search subagent times out while researching a complex topic. You need to design how this failure information flows back to the coordinator agent. Which error propagation approach best enables intelligent recovery?

A) Return structured error context to the coordinator including the failure type, the attempted query, any partial results, and potential alternative approaches.

B) Implement automatic retry logic with exponential backoff within the subagent, returning a generic "search unavailable" status only after all retries are exhausted.

C) Catch the timeout within the subagent and return an empty result set marked as successful.

D) Propagate the timeout exception directly to a top-level handler that terminates the entire research workflow.

Correct Answer: A

Structured error context gives the coordinator the information it needs to make intelligent recovery decisions—whether to retry with a modified query, try an alternative approach, or proceed with partial results. Option B's generic status hides valuable context from the coordinator, preventing informed decisions. Option C suppresses the error by marking failure as success, which prevents any recovery and risks incomplete research outputs. Option D terminates the entire workflow unnecessarily when recovery strategies could succeed.

Question 9: During testing, you observe that the synthesis agent frequently needs to verify specific claims while combining findings. Currently, when verification is needed, the synthesis agent returns control to the coordinator, which invokes the web search agent, then re-invokes synthesis with results. This adds 2-3 round trips per task and increases latency by 40%. Your evaluation shows that 85% of these verifications are simple fact-checks (dates, names, statistics) while 15% require deeper investigation. What's the most effective approach to reduce overhead while maintaining system reliability?

A) Give the synthesis agent a scoped verify_fact tool for simple lookups, while complex verifications continue delegating to the web search agent through the coordinator.

B) Have the synthesis agent accumulate all verification needs and return them as a batch to the coordinator at the end of its pass, which then sends them all to the web search agent at once.

C) Give the synthesis agent access to all web search tools so it can handle any verification need directly without round-trips through the coordinator.

D) Have the web search agent proactively cache extra context around each source during initial research, anticipating what the synthesis agent might need to verify.

Correct Answer: A

Option A applies the principle of least privilege by giving the synthesis agent only what it needs for the 85% common case (simple fact verification) while preserving the existing coordination pattern for complex cases. Option B's batching approach creates blocking dependencies since synthesis steps may depend on earlier verified facts. Option C over-provisions the synthesis agent, violating separation of concerns. Option D relies on speculative caching that cannot reliably predict what the synthesis agent will need to verify.

Scenario: Claude Code for Continuous Integration

Question 10: Your pipeline script runs claude "Analyze this pull request for security issues" but the job hangs indefinitely. Logs indicate Claude Code is waiting for interactive input. What's the correct

approach to run Claude Code in an automated pipeline?

A) Add the -p flag: claude -p "Analyze this pull request for security issues"

B) Set the environment variable CLAUDE_HEADLESS=true before running the command

C) Redirect stdin from /dev/null: claude "Analyze this pull request for security issues" < /dev/null

D) Add the --batch flag: claude --batch "Analyze this pull request for security issues"

Correct Answer: A

The -p (or --print) flag is the documented way to run Claude Code in non-interactive mode. It processes the prompt, outputs the result to stdout, and exits without waiting for user input—exactly what CI/CD pipelines require. The other options reference non-existent features (CLAUDE_HEADLESS environment variable, --batch flag) or use Unix workarounds that don't properly address Claude Code's command syntax.

Question 11: Your team wants to reduce API costs for automated analysis. Currently, real-time Claude calls power two workflows: (1) a blocking pre-merge check that must complete before developers can merge, and (2) a technical debt report generated overnight for review the next morning. Your manager proposes switching both to the Message Batches API for its 50% cost savings. How should you evaluate this proposal?

A) Use batch processing for the technical debt reports only; keep real-time calls for pre-merge checks.

B) Switch both workflows to batch processing with status polling to check for completion.

C) Keep real-time calls for both workflows to avoid batch result ordering issues.

D) Switch both to batch processing with a timeout fallback to real-time if batches take too long.

Correct Answer: A

The Message Batches API offers 50% cost savings but has processing times up to 24 hours with no guaranteed latency SLA. This makes it unsuitable for blocking pre-merge checks where developers wait for results, but ideal for overnight batch jobs like technical debt reports. Option B is wrong because relying on "often faster" completion isn't acceptable for blocking workflows. Option C reflects a misconception—batch results can be correlated using custom_id fields. Option D adds unnecessary complexity when the simpler solution is matching each API to its appropriate use case.

Question 12: A pull request modifies 14 files across the stock tracking module. Your single-pass review analyzing all files together produces inconsistent results: detailed feedback for some files but superficial comments for others, obvious bugs missed, and contradictory feedback—flagging a

pattern as problematic in one file while approving identical code elsewhere in the same PR. How should you restructure the review?

A) Split into focused passes: analyze each file individually for local issues, then run a separate integration-focused pass examining cross-file data flow.

B) Require developers to split large PRs into smaller submissions of 3-4 files before the automated review runs.

C) Switch to a higher-tier model with a larger context window to give all 14 files adequate attention in one pass.

D) Run three independent review passes on the full PR and only flag issues that appear in at least two of the three runs.

Correct Answer: A

Splitting reviews into focused passes directly addresses the root cause: attention dilution when processing many files at once. File-by-file analysis ensures consistent depth, while a separate integration pass catches cross-file issues. Option B shifts burden to developers without improving the system. Option C misunderstands that larger context windows don't solve attention quality issues. Option D would actually suppress detection of real bugs by requiring consensus on issues that may only be caught intermittently.

Preparation Exercises

Complete these hands-on exercises to build practical familiarity with the topics covered on the exam. Each exercise is designed to reinforce knowledge across one or more exam domains.

Exercise 1: Build a Multi-Tool Agent with Escalation Logic

Objective: Practice designing an agentic loop with tool integration, structured error handling, and escalation patterns.

Steps:

1. Define 3-4 MCP tools with detailed descriptions that clearly differentiate each tool's purpose, expected inputs, and boundary conditions. Include at least two tools with similar functionality that require careful description to avoid selection confusion.
2. Implement an agentic loop that checks stop_reason to determine whether to continue tool execution or present the final response. Handle both "tool_use" and "end_turn" stop reasons correctly.

3. Add structured error responses to your tools: include errorCategory (transient/validation/permission), isRetryable boolean, and human-readable descriptions. Test that the agent handles each error type appropriately (retrying transient errors, explaining business errors to the user).
4. Implement a programmatic hook that intercepts tool calls to enforce a business rule (e.g., blocking operations above a threshold amount), redirecting to an escalation workflow when triggered.
5. Test with multi-concern messages (e.g., requests involving multiple issues) and verify the agent decomposes the request, handles each concern, and synthesizes a unified response.

Domains reinforced: Domain 1 (Agentic Architecture & Orchestration), Domain 2 (Tool Design & MCP Integration), Domain 5 (Context Management & Reliability)

Exercise 2: Configure Claude Code for a Team Development Workflow

Objective: Practice configuring CLAUDE.md hierarchies, custom slash commands, path-specific rules, and MCP server integration for a multi-developer project.

Steps:

6. Create a project-level CLAUDE.md with universal coding standards and testing conventions. Verify that instructions placed at the project level are consistently applied across all team members.
7. Create .claude/rules/ files with YAML frontmatter glob patterns for different code areas (e.g., paths: ["src/api/**/*"] for API conventions, paths: ["**/*.test.*"] for testing conventions). Test that rules load only when editing matching files.
8. Create a project-scoped skill in .claude/skills/ with context: fork and allowed-tools restrictions. Verify the skill runs in isolation without polluting the main conversation context.
9. Configure an MCP server in .mcp.json with environment variable expansion for credentials. Add a personal experimental MCP server in ~/.claude.json and verify both are available simultaneously.
10. Test plan mode versus direct execution on tasks of varying complexity: a single-file bug fix, a multi-file library migration, and a new feature with multiple valid implementation approaches. Observe when plan mode provides value.

Domains reinforced: Domain 3 (Claude Code Configuration & Workflows), Domain 2 (Tool Design & MCP Integration)

Exercise 3: Build a Structured Data Extraction Pipeline

Objective: Practice designing JSON schemas, using tool_use for structured output, implementing validation-retry loops, and designing batch processing strategies.

Steps:

11. Define an extraction tool with a JSON schema containing required and optional fields, an enum with an "other" + detail string pattern, and nullable fields for information that may not exist in source documents. Process documents where some fields are absent and verify the model returns null rather than fabricating values.
12. Implement a validation-retry loop: when Pydantic or JSON schema validation fails, send a follow-up request including the document, the failed extraction, and the specific validation error. Track which errors are resolvable via retry (format mismatches) versus which are not (information absent from source).
13. Add few-shot examples demonstrating extraction from documents with varied formats (e.g., inline citations vs bibliographies, narrative descriptions vs structured tables) and verify improved handling of structural variety.
14. Design a batch processing strategy: submit a batch of 100 documents using the Message Batches API, handle failures by custom_id, resubmit failed documents with modifications (e.g., chunking oversized documents), and calculate total processing time relative to SLA constraints.
15. Implement a human review routing strategy: have the model output field-level confidence scores, route low-confidence extractions to human review, and analyze accuracy by document type and field to verify consistent performance.

Domains reinforced: Domain 4 (Prompt Engineering & Structured Output), Domain 5 (Context Management & Reliability)

Exercise 4: Design and Debug a Multi-Agent Research Pipeline

Objective: Practice orchestrating subagents, managing context passing, implementing error propagation, and handling synthesis with provenance tracking.

Steps:

16. Build a coordinator agent that delegates to at least two subagents (e.g., web search and document analysis). Ensure the coordinator's allowedTools includes "Task" and that each subagent receives its research findings directly in its prompt rather than relying on automatic context inheritance.
17. Implement parallel subagent execution by having the coordinator emit multiple Task tool calls in a single response. Measure the latency improvement compared to sequential execution.
18. Design structured output for subagents that separates content from metadata: each finding should include a claim, evidence excerpt, source URL/document name, and publication date. Verify that the synthesis subagent preserves source attribution when combining findings.

19. Implement error propagation: simulate a subagent timeout and verify the coordinator receives structured error context (failure type, attempted query, partial results). Test that the coordinator can proceed with partial results and annotate the final output with coverage gaps.
20. Test with conflicting source data (e.g., two credible sources with different statistics) and verify the synthesis output preserves both values with source attribution rather than arbitrarily selecting one, and structures the report to distinguish well-established from contested findings.

Domains reinforced: Domain 1 (Agentic Architecture & Orchestration), Domain 2 (Tool Design & MCP Integration), Domain 5 (Context Management & Reliability)

Appendix

Technologies and Concepts

The following list contains technologies and concepts that might appear on the exam:

- Claude Agent SDK — Agent definitions, agentic loops, stop_reason handling, hooks (PostToolUse, tool call interception), subagent spawning via Task tool, allowedTools configuration
- Model Context Protocol (MCP) — MCP servers, MCP tools, MCP resources, isError flag, tool descriptions, tool distribution, .mcp.json configuration, environment variable expansion
- Claude Code — CLAUDE.md configuration hierarchy (user/project/directory), .claude/rules/ with YAML frontmatter path-scoping, .claude/commands/ for slash commands, .claude/skills/ with SKILL.md frontmatter (context: fork, allowed-tools, argument-hint), plan mode, direct execution, /memory command, /compact, --resume, fork_session, Explore subagent
- Claude Code CLI — -p / --print flag for non-interactive mode, --output-format json, --json-schema for structured CI output
- Claude API — tool_use with JSON schemas, tool_choice options ("auto", "any", forced tool selection), stop_reason values ("tool_use", "end_turn"), max_tokens, system prompts
- Message Batches API — 50% cost savings, up to 24-hour processing window, custom_id for request/response correlation, polling for completion, no multi-turn tool calling support
- JSON Schema — Required vs optional fields, enum types, nullable fields, "other" + detail string patterns, strict mode for syntax error elimination
- Pydantic — Schema validation, semantic validation errors, validation-retry loops
- Built-in tools — Read, Write, Edit, Bash, Grep, Glob — their purposes and selection criteria
- Few-shot prompting — Targeted examples for ambiguous scenarios, format demonstration, generalization to novel patterns
- Prompt chaining — Sequential task decomposition into focused passes

- Context window management — Token budgets, progressive summarization, lost-in-the-middle effects, context extraction, scratchpad files
- Session management — Session resumption, fork_session, named sessions, session context isolation
- Confidence scoring — Field-level confidence, calibration with labeled validation sets, stratified sampling for error rate measurement

In-Scope Topics

The following topics are explicitly tested on the exam:

- Agentic loop implementation: Control flow based on stop_reason, tool result handling, loop termination conditions
- Multi-agent orchestration: Coordinator-subagent patterns, task decomposition, parallel subagent execution, iterative refinement loops
- Subagent context management: Explicit context passing, structured state persistence, crash recovery using manifests
- Tool interface design: Writing effective tool descriptions, splitting vs consolidating tools, tool naming to reduce ambiguity
- MCP tool and resource design: Resources for content catalogs, tools for actions, description quality for adoption
- MCP server configuration: Project vs user scope, environment variable expansion, multi-server simultaneous access
- Error handling and propagation: Structured error responses, transient vs business vs permission errors, local recovery before escalation
- Escalation decision-making: Explicit criteria, honoring customer preferences, policy gap identification
- CLAUDE.md configuration: Hierarchy (user/project/directory), @import patterns, .claude/rules/ with glob patterns
- Custom commands and skills: Project vs user scope, context: fork, allowed-tools, argument-hint frontmatter
- Plan mode vs direct execution: Complexity assessment, architectural decisions, single-file changes
- Iterative refinement: Input/output examples, test-driven iteration, interview pattern, sequential vs parallel issue resolution
- Structured output via tool_use: Schema design, tool_choice configuration, nullable fields to prevent hallucination
- Few-shot prompting: Ambiguous scenario targeting, format consistency, false positive reduction

- Batch processing: Message Batches API appropriateness, latency tolerance assessment, failure handling by custom_id
- Context window optimization: Trimming verbose tool outputs, structured fact extraction, position-aware input ordering
- Human review workflows: Confidence calibration, stratified sampling, accuracy segmentation by document type and field
- Information provenance: Claim-source mappings, temporal data handling, conflict annotation, coverage gap reporting

Out-of-Scope Topics

The following related topics will NOT appear on the exam:

- Fine-tuning Claude models or training custom models
- Claude API authentication, billing, or account management
- Detailed implementation of specific programming languages or frameworks (beyond what's needed for tool and schema configuration)
- Deploying or hosting MCP servers (infrastructure, networking, container orchestration)
- Claude's internal architecture, training process, or model weights
- Constitutional AI, RLHF, or safety training methodologies
- Embedding models or vector database implementation details
- Computer use (browser automation, desktop interaction)
- Vision/image analysis capabilities
- Streaming API implementation or server-sent events
- Rate limiting, quotas, or API pricing calculations
- OAuth, API key rotation, or authentication protocol details
- Specific cloud provider configurations (AWS, GCP, Azure)
- Performance benchmarking or model comparison metrics
- Prompt caching implementation details (beyond knowing it exists)
- Token counting algorithms or tokenization specifics

Exam Preparation Recommendations

To prepare for this certification exam:

21. Build an agent with the Claude Agent SDK: Implement a complete agentic loop with tool calling, error handling, and session management. Practice spawning subagents and passing context between them.
22. Configure Claude Code for a real project: Set up CLAUDE.md with a configuration hierarchy, create path-specific rules in .claude/rules/, build custom skills with frontmatter options (context: fork, allowed-tools), and integrate at least one MCP server.
23. Design and test MCP tools: Write tool descriptions that clearly differentiate similar tools. Implement structured error responses with error categories and retryable flags. Test tool selection reliability with ambiguous requests.
24. Build a structured data extraction pipeline: Use tool_use with JSON schemas, implement validation-retry loops, design schemas with optional/nullable fields, and practice batch processing with the Message Batches API.
25. Practice prompt engineering techniques: Write few-shot examples for ambiguous scenarios. Define explicit review criteria to reduce false positives. Design multi-pass review architectures for large code reviews.
26. Study context management patterns: Practice extracting structured facts from verbose tool outputs, implementing scratchpad files for long sessions, and designing subagent delegation to manage context limits.
27. Review escalation and human-in-the-loop patterns: Understand when to escalate (policy gaps, customer requests, inability to progress) versus resolve autonomously. Practice designing human review workflows with confidence-based routing.

Version 0.2 Last Updated: June 30 2026