UNOFFICIAL — INDEPENDENT STUDY GUIDE

CLAUDE ARCHITECT HANDBOOK

Mastering Agentic Architecture, Tool Design, MCP & Prompt Engineering for Anthropic's Claude Platform

- Comprehensive Preparation Guide
- Covers All 5 Exam Domains in Detail
- 100+ Pages of In-Depth Content, Scenarios & Practice Questions.

Author

Romi Nur Ismanto

rominur@gmail.com

Practice Exam & Study Resources:
https://claude.jekardah.com/

Version 1.0 — March 2026

UNOFFICIAL — INDEPENDENT STUDY GUIDE

AI SOLUTION ARCHITECT
FOUNDATIONS HANDBOOK

Mastering Agentic Architecture, Tool Design, MCP & Prompt Engineering for Anthropic's Claude Platform

Comprehensive Preparation Guide

Covers All 5 Exam Domains in Detail

100+ Pages of In-Depth Content, Scenarios & Practice Questions

Author

Romi Nur Ismanto

rominur@gmail.com

Practice Exam & Study Resources:

https://claude.jekardah.com/

Version 1.0 — March 2026

DISCLAIMER

This handbook is an independently authored study guide. It is NOT an official publication of Anthropic, PBC and is NOT endorsed, sponsored, or affiliated with Anthropic in any way.

"Claude," "Claude Code," "Model Context Protocol (MCP)," and the Anthropic logo are trademarks or registered trademarks of Anthropic, PBC. All other product names, logos, and brands mentioned in this handbook are the property of their respective owners. The use of these names and trademarks in this publication is solely for educational and identification purposes and does not imply any endorsement or affiliation.

The content in this handbook is based on publicly available documentation, official exam guides, technical references, and the author's independent research and professional experience. While every effort has been made to ensure accuracy, the author makes no guarantees regarding the completeness or correctness of the information presented. Readers should always refer to Anthropic's official documentation for the most current and authoritative information.

This handbook is intended for personal educational use. No part of this publication may be reproduced, distributed, or transmitted in any form without the prior written permission of the author, except for brief quotations in reviews and educational contexts.

For questions, corrections, or feedback, contact: **rominur@gmail.com**

Practice exams available at: **https://claude.jekardah.com/**

Table of Contents

Part I: Introduction & Exam Overview

Chapter 1: About This Certification

The AI Solution Architect – Foundations certification validates professional competence for solution architects who design and implement production applications with Anthropic's Claude platform. Unlike certifications that test rote memorization, this exam evaluates your practical judgment about architecture, configuration, and tradeoffs in real-world production deployments. Note: This handbook is an independent, unofficial study resource and is not produced or endorsed by Anthropic, PBC.

This certification validates that you can make informed decisions when building with four core technologies: **Claude Code**, the **Claude Agent SDK**, the **Claude API**, and the **Model Context Protocol (MCP)**. Every question is grounded in realistic scenarios drawn from actual customer use cases, meaning you need hands-on experience—not just theoretical knowledge.

Who Should Take This Exam?

The ideal candidate is a solution architect with 6+ months of practical experience building with Claude. You should have hands-on familiarity 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

Exam Format

The exam is entirely multiple-choice. Each question has one correct response and three distractors. There is no penalty for guessing, so always answer every question. The exam uses a scaled scoring model from 100 to 1,000, with a minimum passing score of 720.

During the exam, you will encounter 4 scenarios selected randomly from a pool of 6 possible scenarios. Each scenario presents a realistic production context and frames a set of questions. The scenarios cover customer support agents, code generation, multi-agent research systems, developer productivity tools, CI/CD integration, and structured data extraction.

Aspect	Detail
Format	Multiple choice (1 correct, 3 distractors)
Scoring	Scaled 100–1,000; pass at 720
Scenarios	4 of 6 randomly selected
Penalty for guessing	None — always answer
Content type	Scenario-based, practical judgment

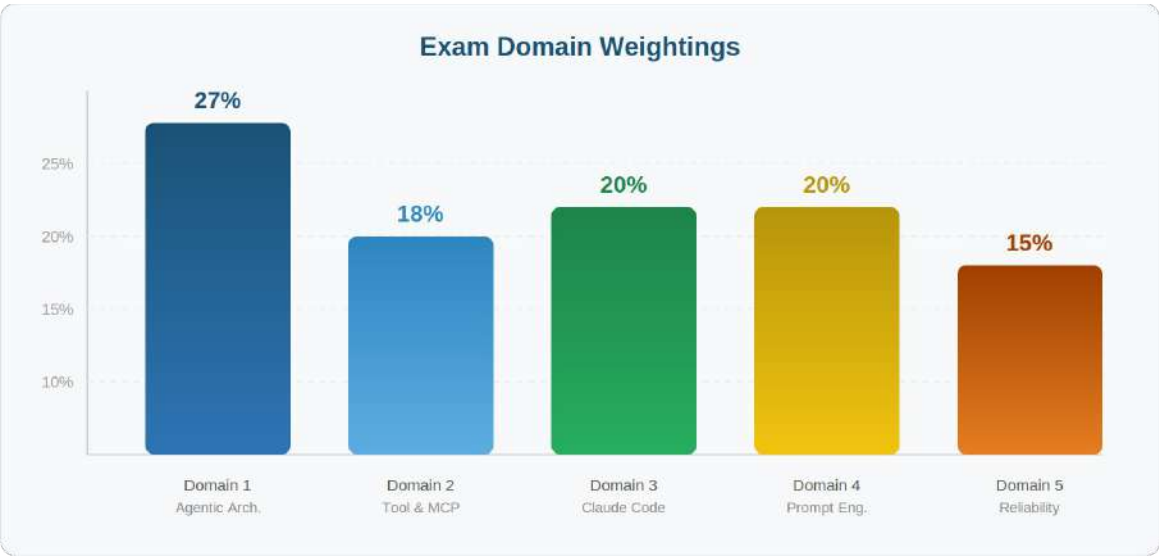
Content Domains & Weightings

The exam covers five domains, each weighted differently. Your study time should roughly proportional to these weightings:

Domain	Weight	Focus Areas
Domain 1: Agentic Architecture & Orchestration	27%	Agentic loops, multi-agent systems, hooks, task decomposition, session management
Domain 2: Tool Design & MCP Integration	18%	Tool descriptions, error handling, tool distribution, MCP servers, built-in tools
Domain 3: Claude Code Configuration & Workflows	20%	CLAUDE.md hierarchy, slash commands, skills, rules, plan mode, CI/CD
Domain 4: Prompt Engineering & Structured Output	20%	Explicit criteria, few-shot examples, JSON schemas, validation, batch processing
Domain 5: Context Management & Reliability	15%	Context preservation, escalation, error propagation, human review, provenance

EXAM STRATEGY TIP

Domain 1 is the heaviest at 27%. Prioritize mastering agentic loops, multi-agent orchestration, and hooks. Combined with Domains 3 and 4 (20% each), these three domains account for 67% of the exam.



Chapter 2: The Six Exam Scenarios

Understanding the scenarios is critical because every exam question is framed within one. Here is a deep dive into each scenario, what it tests, and the mental model you should use when encountering questions from it.

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 connects to backend systems through MCP tools: **get_customer**, **lookup_order**, **process_refund**, and **escalate_to_human**. Your target is 80%+ first-contact resolution.

This scenario primarily tests Domains 1, 2, and 5. Key themes include: ensuring customer identity verification before processing financial operations (programmatic enforcement vs. prompt-based guidance), designing structured handoff summaries for human escalation, handling ambiguous customer requests that span multiple concerns, and calibrating when to escalate versus resolve autonomously.

> **KEY CONCEPT**
>
> When the exam presents a reliability problem in the customer support scenario (e.g., agent skipping verification steps), always consider programmatic enforcement first. Prompt instructions have a non-zero failure rate for critical business logic.

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.

This scenario primarily tests Domains 3 and 5. Key themes include: the CLAUDE.md configuration hierarchy, creating shared vs. personal commands, path-specific rules for different code areas, and choosing between plan mode and direct execution based on task complexity.

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 produces comprehensive, cited reports.

This scenario primarily tests Domains 1, 2, and 5. It is one of the most complex scenarios and covers coordinator-subagent patterns, context passing between agents, parallel execution, error propagation, iterative refinement loops, and provenance tracking in multi-source synthesis.

> **COMMON TRAP**
>
> When reports have coverage gaps, the exam often tests whether you can identify the root cause at the correct level. If the coordinator decomposes a topic too narrowly, the problem is at the coordinator level —not at the subagent level. Each agent may be working correctly within its assigned scope.

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 built-in tools (Read, Write, Bash, Grep, Glob) and integrates with MCP servers.

This scenario tests Domains 2, 3, and 1. Key themes include: selecting the right built-in tool for each task, building codebase understanding incrementally, and integrating MCP servers for enhanced functionality.

Scenario 5: Claude Code for Continuous Integration

You are integrating Claude Code into your 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.

This scenario tests Domains 3 and 4. Key themes include: the -p flag for non-interactive mode, structured output with --output-format json and --json-schema, session context isolation (why a generator shouldn’t review its own code), and batch processing strategies.

Scenario 6: Structured Data Extraction

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

This scenario tests Domains 4 and 5. Key themes include: tool_use with JSON schemas for guaranteed structured output, schema design with optional/nullable fields, validation-retry loops, few-shot examples for varied document formats, and batch processing with the Message Batches API.

Chapter 3: How to Use This Handbook

This handbook is organized into seven parts that follow the exam's domain structure. Each domain chapter includes:

- **Conceptual Foundation**: The underlying principles and mental models you need
- **Deep-Dive Explanations**: Detailed coverage of every task statement, knowledge point, and skill listed in the exam guide
- **Code Examples & Patterns**: Concrete examples showing correct implementations
- **Anti-Patterns**: Common mistakes and why they fail
- **Exam-Style Questions**: Practice questions with detailed explanations
- **Decision Frameworks**: Flowcharts and decision trees for making production tradeoffs

Read the handbook sequentially for first-time preparation, or use it as a reference guide for specific topics. Each chapter is self-contained enough to study independently, though cross-references are provided where concepts overlap.

Part II: Domain 1 — Agentic Architecture & Orchestration (27%)

This is the heaviest domain on the exam. It covers how to design, implement, and manage autonomous agent systems built on Claude. Mastering this domain requires understanding the agentic loop lifecycle, multi-agent coordination patterns, hooks for enforcement, task decomposition strategies, and session management.

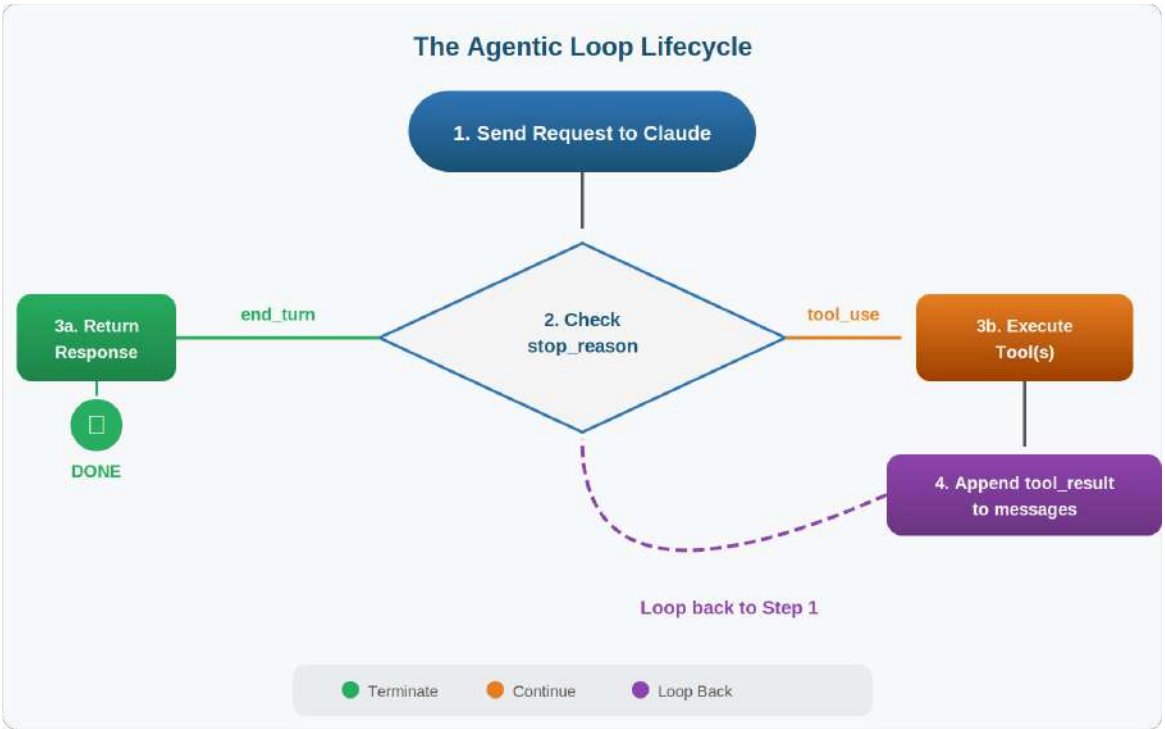
Chapter 4: The Agentic Loop Lifecycle

4.1 Understanding the Core Loop

An agentic loop is the fundamental execution pattern for autonomous Claude-based systems. At its core, the pattern is: send a request to Claude, inspect the stop_reason, and if it's "tool_use", execute the requested tools, append the results to conversation history, and send another request. When stop_reason is "end_turn", the agent has completed its task and you present the final response.

This loop is deceptively simple but forms the backbone of every agentic system. The critical insight is that **Claude drives the decision-making**. The model decides which tool to call next based on the full conversation context, including previous tool results. Your code manages the execution—calling tools and returning results—but the intelligence lives in Claude's reasoning.

```
// Pseudocode for the agentic loop
while (true) {
  response = await claude.messages.create({ messages, tools })
  if (response.stop_reason === "end_turn") {
    return response.content  // Final answer
  }
  if (response.stop_reason === "tool_use") {
    for (const block of response.content) {
      if (block.type === "tool_use") {
        result = await executeTool(block.name, block.input)
        messages.push({ role: "assistant", content: response.content })
        messages.push({ role: "user", content: [{ type: "tool_result",
tool_use_id: block.id, content: result }] })
      }
    }
  }
}
```



4.2 Stop Reason Handling

The stop_reason field is how Claude communicates its intent to your orchestration code. There are two primary values you must handle:

- **"tool_use"**: Claude wants to call one or more tools. Extract the tool calls from the response content, execute them, and feed the results back. The loop continues.
- **"end_turn"**: Claude has finished its task. The response content contains the final answer. The loop terminates.

A critical anti-pattern is using anything other than stop_reason to determine loop behavior. Do not parse the assistant's text content to check for phrases like "I'm done" or "here is the final answer." Do not set arbitrary iteration caps as the primary stopping mechanism. These approaches are fragile and interfere with Claude's natural reasoning flow.

> **ANTI-PATTERN ALERT**
>
> Never check for text content like "Final answer:" or count iterations to determine loop termination. Always use stop_reason. Iteration caps should be safety nets, not primary control flow.

4.3 Tool Results in Conversation History

When Claude calls a tool, the result must be appended to the conversation history before the next API call. This is essential because Claude needs the tool's output to reason about what to do next. Without the result in context, Claude would be "blind" to what the tool returned.

The correct pattern is: first append the assistant's response (which contains the tool_use blocks) as an assistant message, then append a user message containing the tool_result blocks. This maintains the required alternating user/assistant message structure that the API expects.

Each tool_result must reference the corresponding tool_use block via the tool_use_id field. If Claude called multiple tools in a single response, you must return all results in the same user message.

4.4 Model-Driven vs. Pre-Configured Decision Making

A fundamental design principle is the distinction between model-driven decision-making and pre-configured decision trees. In a model-driven approach, Claude reasons about which tool to call next based on the full context—the user's request, previous tool results, and its own analysis. This is flexible and adaptive.

In contrast, pre-configured decision trees hardcode the sequence of tool calls. For example: "Always call get_customer first, then lookup_order, then decide on process_refund." This is rigid and cannot adapt to novel situations.

The exam tests your ability to recognize when each approach is appropriate. Model-driven is generally preferred for complex, variable tasks. Pre-configured sequences are appropriate when strict compliance is required (e.g., identity verification before financial operations)—but even then, the enforcement should be programmatic (via hooks or prerequisite gates), not just prompt instructions.

EXAM INSIGHT

Questions often present scenarios where an agent skips required steps. The correct answer usually involves programmatic enforcement (hooks, prerequisite gates) rather than stronger prompt instructions or few-shot examples. Prompt-based approaches are probabilistic; they reduce but don't eliminate failures.

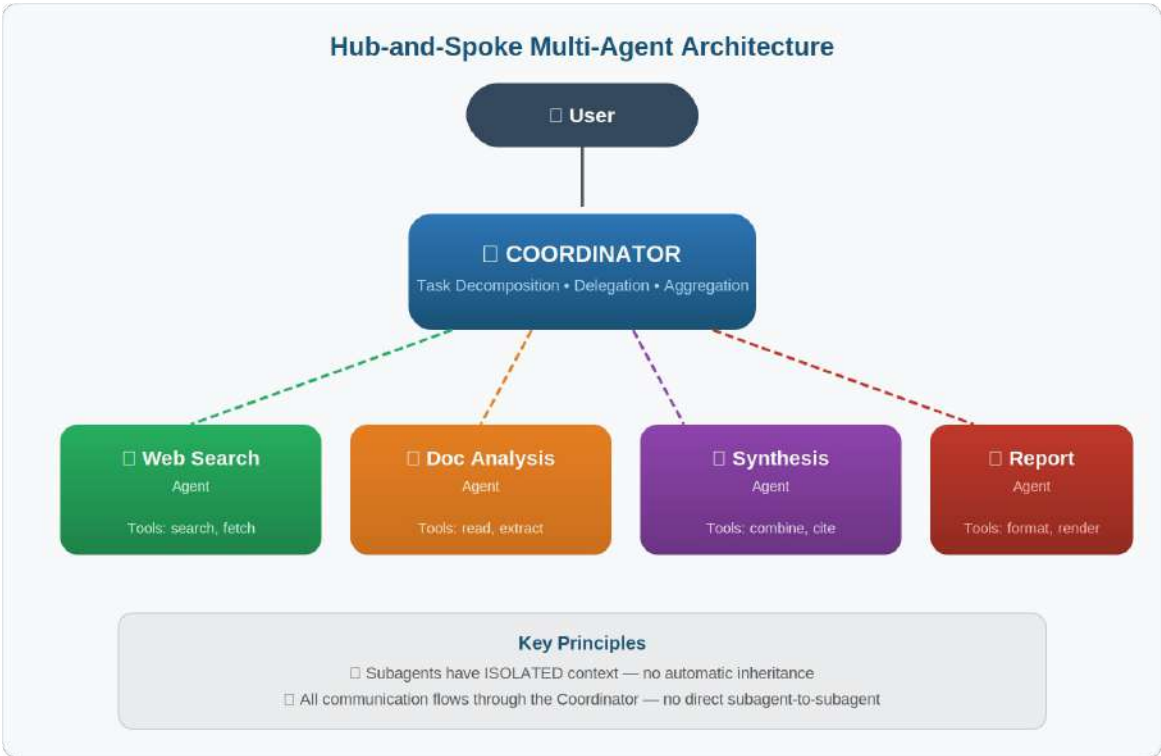
Chapter 5: Multi-Agent Orchestration

5.1 Hub-and-Spoke Architecture

Multi-agent systems in the Claude Agent SDK follow a hub-and-spoke pattern. A coordinator agent sits at the center and manages all communication between specialized subagents. The coordinator is responsible for task decomposition, delegation, result aggregation, and deciding which subagents to invoke.

This architecture has several key properties. First, **subagents do not communicate directly with each other**. All information flows through the coordinator. Second, **subagents operate with isolated context**—they do not automatically inherit the coordinator's conversation history. Third, the coordinator maintains the global view of the task and is responsible for ensuring completeness and coherence.

The coordinator pattern provides observability (all inter-agent communication is visible to the coordinator), consistent error handling (the coordinator can implement unified recovery strategies), and controlled information flow (the coordinator decides what context each subagent receives).



5.2 Subagent Context Isolation

This is one of the most important concepts for the exam: **subagents do not automatically inherit parent context or share memory between invocations**. When the coordinator spawns a subagent, that subagent starts with only what the coordinator explicitly provides in its prompt.

This means if the coordinator wants the synthesis subagent to have the findings from the web search and document analysis subagents, it must explicitly include those findings in the synthesis subagent's prompt. There is no shared memory, no global state, no automatic context inheritance.

The practical implication is that context passing must be intentional and structured. The coordinator should pass:

- The specific research question or task
- All relevant findings from prior agents, in a structured format
- Metadata about sources (URLs, document names, page numbers)
- Any constraints or quality criteria

5.3 Task Decomposition by the Coordinator

The coordinator's first job is to decompose the user's request into subtasks that can be delegated to subagents. This is where many system-level failures originate. If the coordinator decomposes a broad topic too narrowly, the final output will have coverage gaps even if every subagent executes perfectly.

For example, if asked about "the impact of AI on creative industries," a coordinator that decomposes this into "AI in digital art," "AI in graphic design," and "AI in photography" has missed music, writing, and film—a narrow decomposition that covers only visual arts.

A well-designed coordinator should be instructed to generate a broad decomposition first, then validate completeness before delegating. The exam frequently tests whether you can identify the correct level at which a problem originates—coordinator decomposition vs. subagent execution vs. synthesis quality.

> **COMMON EXAM TRAP**
>
> When a multi-agent system produces incomplete results, the root cause is often at the coordinator level (narrow task decomposition), not at the subagent level. Each subagent may be working correctly within its assigned scope. Always trace the problem back to where the scope was defined.

5.4 Dynamic vs. Fixed Routing

The coordinator should dynamically select which subagents to invoke based on the query's requirements, rather than always routing through the full pipeline. For a simple factual question, invoking the full search-analyze-synthesize-report pipeline is wasteful. The coordinator should recognize that a single search subagent might suffice.

This is analogous to the principle of proportionate response: match the complexity of the solution to the complexity of the problem. The exam tests this through scenarios where a simpler routing would be more efficient than invoking all subagents.

5.5 Iterative Refinement Loops

Advanced multi-agent systems implement iterative refinement: the coordinator evaluates the synthesis output, identifies gaps, re-delegates to search and analysis subagents with targeted queries, and re-invokes synthesis until coverage is sufficient.

This pattern transforms a single-pass pipeline into a quality-driven loop. The coordinator acts as a quality gate, checking whether the output meets predefined criteria before presenting it to the user.

Implementation involves: the synthesis agent produces output with coverage annotations indicating which areas are well-supported and which have gaps. The coordinator inspects these annotations and, if gaps exist, generates targeted follow-up queries for the appropriate subagents.

Chapter 6: Subagent Invocation, Context Passing & Spawning

6.1 The Task Tool

In the Claude Agent SDK, subagents are spawned using the **Task** tool. For a coordinator to invoke subagents, its **allowedTools** configuration must include "Task". Without this, the coordinator cannot spawn subagents regardless of its system prompt instructions.

The Task tool allows the coordinator to specify a prompt (the instructions and context for the subagent), the subagent's definition (system prompt, allowed tools), and any constraints. This is the only mechanism for spawning subagents—there is no direct API for inter-agent communication.

6.2 AgentDefinition Configuration

Each subagent type is configured via an **AgentDefinition** that specifies:

- **description**: A brief description of the subagent's role, used by the coordinator to decide when to invoke it
- **System prompt**: Detailed instructions for the subagent's behavior, including output format requirements
- **allowedTools**: The specific tools this subagent can access (scope tools tightly to the subagent's role)

The AgentDefinition is essentially the subagent's blueprint. Getting the tool restrictions right is critical—a synthesis agent should not have access to web search tools, and a search agent should not have access to file-writing tools.

6.3 Explicit Context Passing

Because subagents have isolated context, the coordinator must explicitly pass all necessary information in the subagent's prompt. This is the most common source of bugs in multi-agent systems: forgetting to pass context that the subagent needs.

Best practices for context passing include:

- Include complete findings from prior agents directly in the subagent's prompt
- Use structured data formats to separate content from metadata (source URLs, document names, page numbers)
- Specify research goals and quality criteria rather than step-by-step procedural instructions
- Include the original user question so the subagent understands the end goal

6.4 Parallel Subagent Execution

The coordinator can spawn multiple subagents in parallel by emitting multiple Task tool calls in a single response. This is significantly faster than sequential execution because the subagents run concurrently.

For example, if the coordinator needs web search results and document analysis performed simultaneously, it can emit both Task calls in one turn. The orchestration layer executes both subagents in parallel and returns both results together.

This is a key optimization for reducing latency in multi-agent systems. The exam may test whether you recognize opportunities for parallel execution vs. cases where sequential ordering is required (e.g., synthesis must wait for search results).

6.5 Fork-Based Session Management

The **fork_session** mechanism creates independent branches from a shared analysis baseline. This is useful when you want to explore divergent approaches without contaminating the main session. For example, after a shared codebase analysis, you might fork the session to compare two different refactoring strategies in parallel.

Each fork maintains its own conversation history from the point of forking. Changes in one fork do not affect other forks or the original session.

Chapter 7: Workflow Enforcement & Handoff Patterns

7.1 Programmatic vs. Prompt-Based Enforcement

This is a critical distinction that the exam tests repeatedly. **Programmatic enforcement** uses hooks, prerequisite gates, or code-level checks to guarantee that certain conditions are met before proceeding. **Prompt-based guidance** relies on system prompt instructions to encourage the model to follow a particular workflow.

The key insight: **prompt instructions have a non-zero failure rate**. No matter how well-crafted your system prompt is, there is always a probability that the model will deviate. For low-stakes workflows, this is acceptable. For high-stakes operations like financial transactions, identity verification, or medical decisions, prompt-based enforcement is insufficient.

When the exam presents a scenario where an agent occasionally skips a required step (e.g., 12% of cases skip identity verification before processing refunds), the correct solution is programmatic enforcement, not stronger prompt instructions or more few-shot examples.

Approach	Guarantee Level	Use When
Programmatic enforcement	Deterministic (100%)	Critical business rules, financial operations, identity verification, compliance requirements
Prompt instructions + few-shot	Probabilistic (90–99%)	Preferred workflows, quality guidelines, tone/style requirements
Prompt instructions alone	Probabilistic (80–95%)	Soft preferences, formatting suggestions, general behavioral guidance

7.2 Implementing Programmatic Prerequisites

Programmatic prerequisites block downstream tool calls until prerequisite steps have completed. For example, blocking process_refund and lookup_order until get_customer has returned a verified customer ID.

Implementation typically involves a state machine or flag in the orchestration code. Before executing a tool call, the orchestration layer checks whether the prerequisite condition has been met. If not, it either blocks the call and instructs the model to complete the prerequisite first, or it automatically executes the prerequisite step.

7.3 Structured Handoff Protocols

When an agent escalates to a human, the human agent typically does not have access to the full conversation transcript. Therefore, the AI agent must compile a structured handoff summary that includes all relevant information: customer ID, order details, root cause analysis, actions already taken, refund amount (if applicable), and recommended next action.

The exam tests your ability to design these handoff summaries. A good handoff summary should enable the human agent to continue from where the AI left off without requiring the customer to repeat information.

Chapter 8: Agent SDK Hooks

8.1 PostToolUse Hooks

PostToolUse hooks intercept tool results after a tool executes but before the model processes the result. This is useful for data normalization: converting heterogeneous formats from different MCP tools into a consistent format before the agent sees them.

For example, different backend systems might return dates as Unix timestamps, ISO 8601 strings, or human-readable formats. A PostToolUse hook can normalize all dates to ISO 8601 before the agent processes them, reducing confusion and improving reliability.

8.2 Tool Call Interception Hooks

Tool call interception hooks run before a tool executes. They can inspect the tool call parameters and either allow the call to proceed, modify it, or block it entirely. This is the mechanism for enforcing compliance rules.

For example, a hook can intercept process_refund calls and check whether the refund amount exceeds $500. If it does, the hook blocks the call and redirects to a human escalation workflow. This provides a deterministic guarantee that no refund over $500 is processed without human approval, regardless of what the model's prompt says.

8.3 Hooks vs. Prompts: When to Choose Which

Use hooks when:

- Business rules require guaranteed compliance (financial limits, identity verification)
- Data normalization is needed (format conversion, unit standardization)
- Policy enforcement must be deterministic (never allow X, always require Y)

Use prompt instructions when:

- Guidelines are soft preferences (preferred tone, formatting style)
- Behavior requires judgment (when to escalate, how much detail to provide)
- The cost of occasional non-compliance is low

Chapter 9: Task Decomposition Strategies

9.1 Prompt Chaining (Fixed Sequential Pipelines)

Prompt chaining breaks a complex task into a fixed sequence of steps, where each step's output feeds into the next. This is appropriate for predictable, well-understood workflows where the steps are known in advance.

Example: A code review pipeline might chain: (1) analyze each file individually for local issues, (2) run a cross-file integration pass for data flow problems, (3) generate a summary report. Each step is well-defined and the sequence doesn't vary.

9.2 Dynamic Adaptive Decomposition

Dynamic decomposition generates subtasks based on what is discovered at each step. This is appropriate for open-ended investigation tasks where you don't know the structure of the problem in advance.

Example: "Add comprehensive tests to a legacy codebase" requires first mapping the codebase structure, then identifying high-impact areas, then creating a prioritized plan that adapts as dependencies are discovered. You can't predetermine the steps because they depend on what the codebase contains.

9.3 Choosing the Right Pattern

Pattern	Use When	Example
Prompt chaining	Steps are predictable, well-defined, and don't vary	Multi-aspect code review with known categories
Dynamic decomposition	Steps depend on intermediate findings	Open-ended investigation, legacy codebase analysis
Hybrid	Initial steps are known, later steps depend on results	Plan a migration (known phases), adapt per-service (variable)

9.4 Splitting Large Code Reviews

A key exam topic is splitting large code reviews to avoid attention dilution. When reviewing many files at once, models give inconsistent depth: detailed feedback for some files but superficial comments for others, and sometimes contradictory feedback (flagging a pattern as problematic in one file while approving identical code elsewhere).

The solution is to split into focused passes: analyze each file individually for local issues (per-file passes), then run a separate integration-focused pass examining cross-file data flow. This ensures consistent depth and catches both local and integration issues.

Chapter 10: Session State, Resumption & Forking

10.1 Named Session Resumption

Claude Code supports named sessions via **--resume <session-name>**. This continues a specific prior conversation, preserving the full conversation history. Named sessions are useful for long-running investigations that span multiple work sessions.

However, there's an important caveat: if files have been modified since the last session, the agent's prior tool results may be stale. When resuming, you should inform the agent about specific changes to previously analyzed files so it can perform targeted re-analysis rather than requiring full re-exploration.

10.2 When to Resume vs. Start Fresh

The decision between resuming a session and starting fresh depends on how much the context has changed:

- **Resume** when prior context is mostly valid and only targeted updates are needed
- **Start fresh with injected summaries** when prior tool results are stale, significant code changes have been made, or the session's context is cluttered with verbose exploration output

Starting fresh with a structured summary of prior findings is often more reliable than resuming with stale context. The summary provides a clean foundation without the noise of outdated tool results.

10.3 Fork Session

fork_session creates independent branches from a shared analysis baseline. Use cases include:

- Comparing two testing strategies from a shared codebase analysis
- Exploring alternative refactoring approaches in parallel
- Trying different implementation approaches for a feature

Each fork is independent—changes in one fork do not affect others.

Chapter 11: Domain 1 Practice Questions

Test your understanding of agentic architecture with these practice questions. Each question includes a detailed explanation.

Question 1: Your customer support agent occasionally processes refunds for the wrong customer because it skips the get_customer verification step and directly calls lookup_order with the customer's stated name. Production data shows this happens in approximately 12% of cases. What is the most effective solution?

A) Strengthen the system prompt to emphasize that get_customer is mandatory before any order operations

B) Add few-shot examples showing the agent always calling get_customer first

C) Implement a programmatic prerequisite that blocks lookup_order and process_refund until get_customer has returned a verified customer ID

D) Implement a routing classifier that analyzes each request and pre-selects the appropriate tool sequence

Correct Answer: C

When a specific tool sequence is required for critical business logic (identity verification before financial operations), programmatic enforcement provides deterministic guarantees. Options A and B rely on probabilistic LLM compliance, which is insufficient when errors have financial consequences. Option D addresses tool availability rather than ordering, which is not the actual problem.

Question 2: You're building a multi-agent research system where a coordinator delegates to search, analysis, and synthesis subagents. The synthesis subagent needs the findings from both the search and analysis subagents. How should the coordinator pass this information?

A) Set up a shared memory store that all subagents can read from and write to

B) Include the complete findings from the search and analysis subagents directly in the synthesis subagent's prompt

C) Configure the subagents to automatically inherit the coordinator's full conversation history

D) Have each subagent write its findings to a file, then tell the synthesis subagent where to find them

Correct Answer: B

Subagents operate with isolated context—they do not automatically inherit the coordinator's conversation history. The coordinator must explicitly include all relevant findings in the subagent's prompt. Option A describes a shared memory mechanism that doesn't exist in the Agent SDK. Option C is incorrect because automatic context inheritance is not how subagents work. Option D adds unnecessary complexity when direct prompt injection is simpler and more reliable.

Question 3: Your agentic loop implementation checks for the text "Final Answer:" in the assistant's response to determine when to stop iterating. What is the primary problem with this approach?

A) The text "Final Answer:" might appear in tool results, causing premature termination

B) The stop_reason field provides a reliable, structured signal; parsing text content is fragile and unreliable

C) This approach doesn't work with streaming responses

D) The text pattern is language-dependent and won't work in multilingual contexts

Correct Answer: B

The agentic loop should always use stop_reason to determine control flow. stop_reason "tool_use" means continue; "end_turn" means stop. Parsing natural language signals from the assistant's text content is an anti-pattern because it's fragile, unreliable, and bypasses the structured communication mechanism that the API provides. While A is a valid secondary concern, B captures the fundamental design principle.

Part III: Domain 2 — Tool Design & MCP Integration (18%)

This domain covers how to design tool interfaces that Claude can use effectively, implement structured error handling for MCP tools, distribute tools across agents appropriately, integrate MCP servers, and select the right built-in tools for code exploration tasks.

Chapter 12: Designing Effective Tool Interfaces

12.1 Tool Descriptions as the Primary Selection Mechanism

Tool descriptions are the primary mechanism LLMs use for tool selection. When Claude receives a request and has access to multiple tools, it reads each tool's description to decide which one to call. Minimal descriptions lead to unreliable selection, especially when multiple tools have similar functionality.

A good tool description should include: the tool's purpose, expected input format, what it returns, example queries it can handle, edge cases, and boundaries explaining when to use it versus similar alternatives.

For example, compare these two descriptions for a customer lookup tool:

```
// BAD: Minimal description
"Retrieves customer information"
```

```
// GOOD: Detailed description with boundaries
"Looks up customer records by email, phone, or customer ID.
 Returns: name, email, phone, account status, last 5 orders.
 Use when: User needs customer account details or identity verification.
 Do NOT use for: Order-specific lookups (use lookup_order instead),
 billing history (use get_billing_history instead).
 Example inputs: 'john@example.com', '+1-555-0123', 'CUST-12345'"
```

12.2 Eliminating Functional Overlap

When two tools have overlapping descriptions, Claude may misroute requests. For example, if you have both **analyze_content** and **analyze_document** with near-identical descriptions, Claude will inconsistently choose between them.

The solution is to either rename the tools to make their purposes distinct or consolidate them. For example, renaming analyze_content to **extract_web_results** with a web-specific description immediately clarifies when it should be used versus analyze_document.

12.3 Splitting Generic Tools

A single generic tool that handles many different operations forces Claude to figure out which sub-operation to invoke, increasing error rates. Splitting a generic **analyze_document** into purpose-specific tools like **extract_data_points**, **summarize_content**, and **verify_claim_against_source** gives Claude clear, unambiguous choices.

Each tool should have a well-defined input/output contract. This makes tool selection more reliable and makes it easier to debug when things go wrong.

12.4 System Prompt Interference

Keyword-sensitive instructions in the system prompt can create unintended tool associations. For example, if the system prompt says “Always use the analyze tool for any request containing the word ‘analyze’,” this can override well-written tool descriptions and cause misrouting.

When debugging tool selection issues, always review the system prompt for instructions that might be overriding tool descriptions. Tool descriptions and system prompts should work together, not conflict.

Chapter 13: Structured Error Responses for MCP Tools

13.1 The MCP isError Flag

MCP defines an **isError** flag for tool responses that signals to the agent that the tool call failed. This is the standard mechanism for communicating failures. However, just setting isError is not enough—the error response should include structured metadata to help the agent make intelligent recovery decisions.

13.2 Error Categories

Not all errors are the same, and the agent needs to know what kind of error occurred to respond appropriately:

- **Transient errors**: Timeouts, service unavailability. These are retryable—the same request may succeed on retry.
- **Validation errors**: Invalid input format, missing required fields. These require the agent to fix the input before retrying.
- **Business errors**: Policy violations (e.g., refund exceeds limit, item not eligible for return). These are not retryable and require alternative workflows.
- **Permission errors**: Insufficient access rights. These typically require escalation.

13.3 Structured Error Metadata

A well-designed error response includes:

```
{
  "isError": true,
  "errorCategory": "business",  // transient | validation | permission | business
  "isRetryable": false,
  "description": "Refund amount ($750) exceeds automated processing limit ($500)",
  "customerFriendlyMessage": "This refund requires manager approval",
  "suggestedAction": "escalate_to_human"
}
```

This structured metadata enables the agent to: retry transient errors automatically, fix validation errors and retry, explain business errors to the customer in friendly language, and escalate permission errors appropriately.

13.4 Anti-Patterns in Error Handling

Two critical anti-patterns to avoid:

- **Generic error messages**: Returning “Operation failed” without category, retryability, or context prevents the agent from making informed recovery decisions.

- **Silent error suppression**: Catching errors and returning empty results marked as successful hides failures and leads to incomplete or incorrect outputs.

A third anti-pattern is terminating entire workflows on a single failure. In a multi-agent system, one subagent's failure shouldn't kill the whole pipeline. Instead, partial results should be preserved, and the coordinator should decide how to proceed.

13.5 Access Failures vs. Valid Empty Results

A critical distinction: an empty result set from a query that timed out (access failure) is fundamentally different from an empty result set from a query that ran successfully but found no matches (valid empty result). The agent's response should be different in each case.

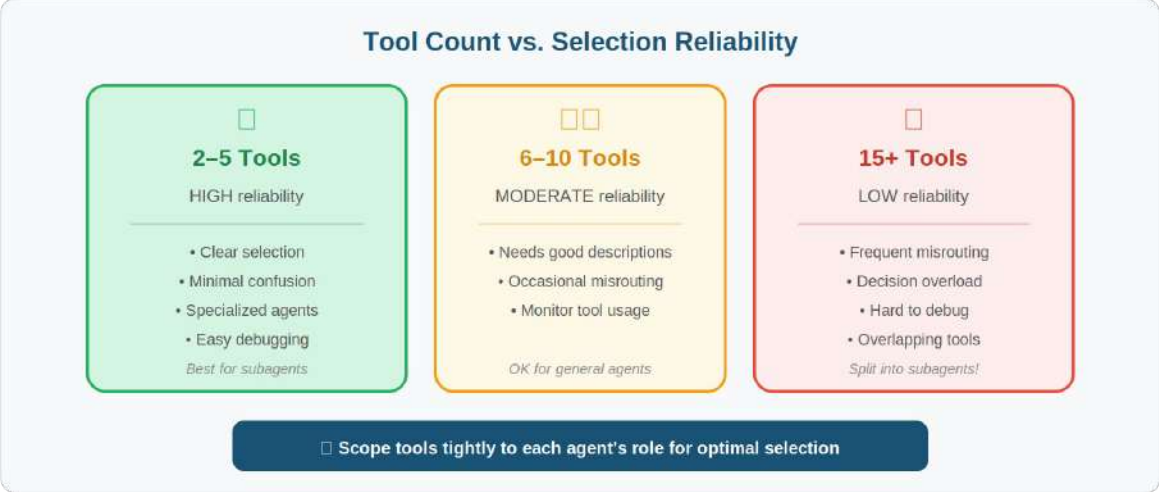
For access failures, the agent might retry or inform the user that data is temporarily unavailable. For valid empty results, the agent should communicate that no matching records were found. Confusing these two cases leads to poor user experience.

Chapter 14: Tool Distribution Across Agents

14.1 The Tool Overload Problem

Giving an agent access to too many tools degrades tool selection reliability. Research and practice show that agents with 4–5 well-scoped tools make significantly better decisions than agents with 15–20 tools. The decision complexity increases combinatorially with the number of available tools.

The principle is **scope tools to the agent's role**. A synthesis agent needs tools for combining and formatting results, not tools for web searching or file writing. A search agent needs tools for querying and fetching, not tools for processing refunds.



14.2 Scoped Cross-Role Tools

Sometimes a subagent has a high-frequency need that would normally require routing through the coordinator. For example, a synthesis agent that frequently needs to verify simple facts would benefit from having a scoped **verify_fact** tool—a limited version of the search agent's capabilities designed for simple lookups only.

This is a tradeoff: adding a scoped cross-role tool reduces latency (no round-trip through the coordinator) but slightly increases the agent's tool count. The exam tests your judgment about when this tradeoff is worthwhile (high-frequency simple lookups) versus when it's not (complex queries that should go through the coordinator).

14.3 tool_choice Configuration

The **tool_choice** parameter controls how Claude selects tools:

- **"auto"** (default): Claude may call a tool or return text. Good for general-purpose agents where tool calls are sometimes but not always needed.

- **"any"**: Claude must call a tool but can choose which one. Use when you need guaranteed structured output but have multiple extraction schemas.
- **Forced selection**: {"type": "tool", "name": "specific_tool"}. Claude must call the specified tool. Use when a specific step must happen (e.g., forcing extract_metadata before enrichment).

EXAM TIP

tool_choice: "any" guarantees a tool call but allows the model to choose which tool. Forced selection guarantees a specific tool. tool_choice: "auto" allows the model to skip tool calls entirely. Know when each is appropriate.

Chapter 15: MCP Server Integration

15.1 MCP Server Scoping

MCP servers can be configured at two levels:

- **Project-level** (.mcp.json): Shared team tooling, committed to version control. All team members get the same MCP server configuration.
- **User-level** (~/.claude.json): Personal or experimental servers. Not shared via version control.

The scoping decision is about audience: if the tool is useful for the whole team, put it in .mcp.json. If it's experimental or personal, put it in ~/.claude.json.

15.2 Environment Variable Expansion

The .mcp.json file supports environment variable expansion for credentials. For example, ${GITHUB_TOKEN} in the configuration file is replaced with the actual token from the environment. This allows teams to commit .mcp.json to version control without exposing secrets.

15.3 MCP Resources

MCP resources expose content catalogs—like issue summaries, documentation hierarchies, or database schemas—that give agents visibility into available data without requiring exploratory tool calls. Instead of the agent making multiple tool calls to discover what data is available, it can inspect the resources catalog and make targeted requests.

Resources reduce latency and token usage by eliminating the discovery phase. The agent can look at the schema and know exactly what fields are available, rather than calling a tool and parsing the response to learn the structure.

15.4 Community vs. Custom MCP Servers

For standard integrations (Jira, GitHub, Slack), use existing community MCP servers rather than building custom ones. Custom servers should be reserved for team-specific workflows that aren't served by available community options. This saves development time and benefits from community maintenance and bug fixes.

Chapter 16: Built-in Tools

16.1 Tool Selection Guide

Claude Code provides several built-in tools. Selecting the right one for each task is a key exam skill:

Tool	Purpose	Use When
Grep	Content search	Searching file contents for patterns (function names, error messages, import statements)
Glob	File path matching	Finding files by name or extension pattern (e.g., **/*.test.tsx)
Read	Full file operations	Loading complete file contents for analysis
Write	Full file creation	Creating new files or full file replacement
Edit	Targeted modifications	Modifying specific text within a file using unique text matching
Bash	Shell commands	Running build commands, tests, complex file operations

16.2 Incremental Codebase Understanding

A best practice for exploring unfamiliar codebases is to build understanding incrementally rather than trying to read everything upfront:

- Start with **Grep** to find entry points (main functions, API endpoints, route handlers)
- Use **Read** to follow imports and trace execution flows from the entry points
- Use **Glob** to find related files (test files, configuration, documentation)
- Build a mental model incrementally, expanding outward from the entry points

This approach is more efficient than reading all files upfront because it focuses on the most relevant code first and avoids overwhelming the context window with irrelevant content.

16.3 When Edit Fails

The Edit tool relies on finding unique text matches to know where to make modifications. When the text isn't unique (e.g., a common pattern appears in multiple places), Edit fails. The fallback is to use Read to load the full file contents, then Write to replace the entire file with the modified version.

Chapter 17: Domain 2 Practice Questions

Question 4: Your 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. What's the most effective first step?

A) Add 5–8 few-shot examples showing order queries routing to lookup_order

B) Expand each tool's description to include input formats, example queries, edge cases, and boundaries

C) Implement a routing layer that pre-selects tools based on keyword detection

D) Consolidate both tools into a single lookup_entity tool

Correct Answer: B

Tool descriptions are the primary mechanism for tool selection. Expanding descriptions is a low-effort, high-leverage fix that addresses the root cause. Few-shot examples (A) add token overhead without fixing the underlying issue. A routing layer (C) is over-engineered. Consolidation (D) is valid but more effort than warranted as a "first step."

Question 5: A web search subagent times out during a research task. Which error propagation approach best enables intelligent coordinator recovery?

A) Return structured error context including failure type, attempted query, partial results, and alternative approaches

B) Implement automatic retry with exponential backoff, returning generic "search unavailable" after retries exhaust

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

D) Propagate the timeout exception to terminate the entire workflow

Correct Answer: A

Structured error context enables the coordinator to make informed recovery decisions: retry with modified queries, try alternatives, or proceed with partial results. Option B hides context behind a generic message. Option C silently suppresses the error. Option D kills the whole pipeline unnecessarily.

Part IV: Domain 3 — Claude Code Configuration & Workflows (20%)

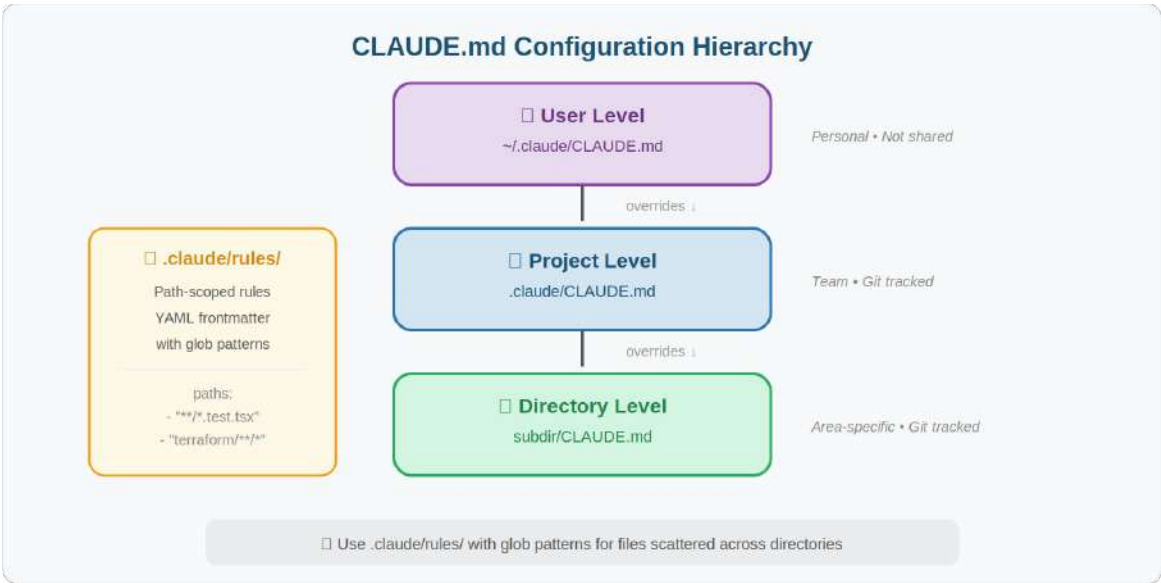
Chapter 18: CLAUDE.md Configuration Hierarchy

18.1 The Three Levels

CLAUDE.md files form a hierarchy that controls Claude Code's behavior. Understanding this hierarchy is essential for the exam:

- **User-level** (~/.claude/CLAUDE.md): Personal preferences that apply only to you. Not shared with teammates via version control.
- **Project-level** (.claude/CLAUDE.md or root CLAUDE.md): Shared team standards committed to the repository. Applied to all team members.
- **Directory-level** (subdirectory CLAUDE.md files): Context-specific instructions that apply when working in that directory.

A common exam scenario: a new team member isn't receiving certain instructions. The diagnosis is that the instructions are in the user-level configuration (~/.claude/CLAUDE.md) rather than the project-level configuration, so they aren't shared via version control.



18.2 @import Syntax

The **@import** syntax allows CLAUDE.md files to reference external files, keeping configurations modular. For example, each package in a monorepo can import only the standards files relevant to its domain:

```
// In packages/api/CLAUDE.md
@import ../../standards/api-conventions.md
```

```
@import ../../standards/error-handling.md

// In packages/frontend/CLAUDE.md
@import ../../standards/react-conventions.md
@import ../../standards/accessibility.md
```

18.3 .claude/rules/ Directory

As an alternative to a monolithic CLAUDE.md, you can organize instructions into topic-specific files in the **.claude/rules/** directory. For example: testing.md, api-conventions.md, deployment.md. This is easier to maintain and allows different team members to own different rule files.

18.4 The /memory Command

Use **/memory** to verify which memory files are loaded and diagnose inconsistent behavior across sessions. If Claude is behaving differently for different team members, /memory can reveal whether the configuration files are loading correctly.

Chapter 19: Custom Slash Commands & Skills

19.1 Slash Commands

Custom slash commands are defined as files in the .claude/commands/ directory. They provide team-wide shortcuts for common workflows.

- **Project-scoped** (.claude/commands/): Shared via version control, available to all team members
- **User-scoped** (~/.claude/commands/): Personal commands, not shared

19.2 Skills

Skills are defined in **.claude/skills/** with **SKILL.md** files that support frontmatter configuration. Skills are more powerful than simple commands because they support several configuration options:

- **context: fork**: Runs the skill in an isolated sub-agent context, preventing verbose output from polluting the main conversation
- **allowed-tools**: Restricts which tools the skill can use during execution
- **argument-hint**: Prompts developers for required parameters when invoking the skill without arguments

19.3 Skills vs. CLAUDE.md

Skills are invoked on demand for task-specific workflows. CLAUDE.md rules are always loaded as universal standards. Choose skills when you need a specific workflow triggered by the developer; choose CLAUDE.md when the instructions should always apply.

19.4 Personal Skill Customization

To customize a skill without affecting teammates, create a personal variant in ~/.claude/skills/ with a different name. This preserves the team's shared skill while giving you a personalized version.

Chapter 20: Path-Specific Rules

20.1 YAML Frontmatter with Glob Patterns

Files in .claude/rules/ can include YAML frontmatter with a **paths** field containing glob patterns. When this is present, the rule file loads only when editing files that match the pattern.

```
---
paths:
  - "terraform/**/*"
---
# Terraform Conventions
Always use modules for reusable infrastructure...
```

20.2 Advantages Over Directory-Level CLAUDE.md

Path-specific rules have a significant advantage over directory-level CLAUDE.md files: they can apply to files spread across multiple directories. For example, test files are often co-located with source files (Button.test.tsx next to Button.tsx). A directory-level CLAUDE.md can’t apply testing conventions to all test files because they’re spread throughout the codebase. A path-specific rule with the glob pattern **/*.test.tsx can.

Path-scoped rules also reduce irrelevant context and token usage because they load only when editing matching files.

Chapter 21: Plan Mode vs. Direct Execution

21.1 When to Use Plan Mode

Plan mode is designed for complex tasks that involve:

- Large-scale changes affecting many files
- Multiple valid approaches with different tradeoffs
- Architectural decisions that affect the whole system
- Multi-file modifications with dependencies

Examples: microservice restructuring, library migrations affecting 45+ files, choosing between integration approaches with different infrastructure requirements.

21.2 When to Use Direct Execution

Direct execution is appropriate for simple, well-scoped changes:

- Single-file bug fixes with clear stack traces
- Adding a validation check to one function
- Updating a configuration value
- Any change where the scope and approach are obvious

21.3 The Explore Subagent

The **Explore** subagent isolates verbose discovery output, returning only summaries to the main conversation. This preserves the main conversation's context window during multi-phase tasks. Without Explore, discovery output can fill the context window, leaving insufficient room for the actual implementation.

21.4 Combining Approaches

A powerful pattern is to combine plan mode for investigation with direct execution for implementation. First, use plan mode to explore the codebase, understand dependencies, and design an approach. Then switch to direct execution to implement the planned changes.

Chapter 22: Iterative Refinement Techniques

22.1 Input/Output Examples

When prose descriptions produce inconsistent results, 2–3 concrete input/output examples are the most effective way to communicate expected transformations. Show the exact input and the exact expected output.

22.2 Test-Driven Iteration

Write test suites covering expected behavior, edge cases, and performance requirements before implementation. Then iterate by sharing test failures with Claude to guide progressive improvement. This creates a clear, objective feedback loop.

22.3 The Interview Pattern

Have Claude ask questions to surface considerations you may not have anticipated before implementing. This is especially valuable in unfamiliar domains where you don’t know what you don’t know. For example, before implementing a caching system, Claude might ask about cache invalidation strategies, TTL requirements, memory limits, and failure modes.

22.4 Batched vs. Sequential Issue Resolution

When fixing multiple issues, consider whether they interact. Interacting problems (where fixing one affects another) should be addressed in a single message. Independent problems can be fixed sequentially, one at a time.

Chapter 23: CI/CD Integration

23.1 Non-Interactive Mode

The **-p** (or **--print**) flag runs Claude Code in non-interactive mode, which is required for CI/CD pipelines. Without this flag, Claude Code waits for interactive input, causing the pipeline to hang indefinitely.

```
claude -p "Analyze this pull request for security issues"
```

23.2 Structured Output for CI

For CI contexts, use **--output-format json** and **--json-schema** to produce machine-parseable structured findings. This enables automated posting as inline PR comments.

23.3 Session Context Isolation

The same Claude session that generated code is less effective at reviewing its own changes. The model retains reasoning context from generation, making it less likely to question its own decisions. Use an independent review instance without the generator's reasoning context for better reviews.

23.4 Avoiding Duplicate Comments

When re-running reviews after new commits, include prior review findings in context and instruct Claude to report only new or still-unaddressed issues. This prevents duplicate comments that frustrate developers.

23.5 Leveraging CLAUDE.md for CI

Document testing standards, valuable test criteria, and available fixtures in CLAUDE.md. CI-invoked Claude Code reads the same CLAUDE.md, so these instructions improve test generation quality and reduce low-value output.

Chapter 24: Domain 3 Practice Questions

Question 6: You want to create a /review slash command available to every developer when they clone the repository. Where should you create this command file?

A) .claude/commands/ in the project repository

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

C) CLAUDE.md at the project root

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

Correct Answer: A

Project-scoped commands in .claude/commands/ are version-controlled and automatically available to all developers. Option B is for personal commands. Option C is for project instructions, not command definitions. Option D describes a mechanism that doesn't exist.

Question 7: Your codebase has test files spread throughout directories (Button.test.tsx next to Button.tsx). You want all tests to follow the same conventions regardless of location. What's the most maintainable approach?

A) Create .claude/rules/ files with YAML frontmatter glob patterns like **/*.test.tsx

B) Consolidate all conventions in the root CLAUDE.md

C) Create skills in .claude/skills/ for each code type

D) Place a CLAUDE.md in each subdirectory

Correct Answer: A

Path-specific rules with glob patterns apply conventions based on file paths regardless of directory location—ideal for scattered test files. Option B relies on inference. Option C requires manual invocation. Option D can't handle files across many directories.

Part V: Domain 4 — Prompt Engineering & Structured Output (20%)

Chapter 25: Designing Precise Prompts

25.1 Explicit Criteria Over Vague Instructions

The most common prompt engineering mistake is using vague instructions like “be conservative” or “only report high-confidence findings.” These fail to improve precision because they don’t define what “conservative” or “high-confidence” means in concrete terms.

Instead, define explicit categorical criteria. For a code review, specify: “Flag comments only when claimed behavior contradicts actual code behavior” rather than “check that comments are accurate.” The first gives a clear, testable criterion; the second is ambiguous.

25.2 Impact of False Positives

High false positive rates destroy developer trust. If your code review tool frequently flags correct code as problematic, developers will start ignoring all findings—including legitimate bugs. The exam tests your understanding that a few accurate categories with low false positives are more valuable than many categories with high false positive rates.

Strategy: temporarily disable high false-positive categories to restore trust while improving prompts for those categories separately.

25.3 Severity Classification

Define explicit severity criteria with concrete code examples for each level. Don’t rely on the model’s judgment about what constitutes “high” vs. “medium” severity—provide examples that demonstrate the boundary.

Chapter 26: Few-Shot Prompting

26.1 Why Few-Shot Examples Work

Few-shot examples are the most effective technique for achieving consistent output when detailed instructions alone produce inconsistent results. They work because they:

- Demonstrate the exact format and structure expected
- Show how to handle ambiguous cases
- Enable the model to generalize judgment to novel patterns
- Reduce hallucination in extraction tasks

26.2 Designing Effective Few-Shot Examples

Best practices for few-shot examples:

- Use 2–4 targeted examples (not 10–20—too many examples waste tokens without improving quality)
- Focus examples on ambiguous scenarios that show reasoning for why one action was chosen over alternatives
- Include examples of the desired output format (location, issue, severity, suggested fix)
- Show examples distinguishing acceptable code patterns from genuine issues
- Demonstrate correct handling of varied document structures

26.3 Few-Shot for Extraction Tasks

In extraction tasks, few-shot examples are especially effective for handling varied document structures. Show examples of extracting from documents with inline citations vs. bibliographies, methodology sections vs. embedded details, and informal measurements vs. standardized units. This teaches the model to handle structural variety rather than just matching one expected format.

Chapter 27: Enforcing Structured Output

27.1 tool_use with JSON Schemas

Using **tool_use** with JSON schemas is the most reliable approach for guaranteed schema-compliant structured output. When you define a tool with a JSON schema, Claude's output is constrained to match that schema exactly, eliminating JSON syntax errors.

However, strict schemas eliminate syntax errors but do NOT prevent semantic errors. For example, the schema might require a "total" field, and Claude might fill it with a value that doesn't match the sum of line items. Schema compliance guarantees format, not correctness.

27.2 Schema Design Best Practices

- Make fields **optional (nullable)** when the source document may not contain the information. This prevents the model from fabricating values to satisfy required fields.
- Use **enum fields** with an "other" option plus a detail string for extensible categories. This captures known categories while allowing for unexpected ones.
- Add an **"unclear"** enum value for ambiguous cases, giving the model an honest option instead of forcing a guess.
- Include **format normalization rules** in prompts alongside schemas to handle inconsistent source formatting.

27.3 tool_choice for Structured Output

When you need guaranteed structured output:

- **tool_choice: "any"**: Model must call a tool but can choose which one. Use when you have multiple extraction schemas and the document type is unknown.
- **Forced tool selection**: {"type": "tool", "name": "extract_metadata"}. Use when a specific extraction must happen. The model cannot skip the tool call or choose a different tool.

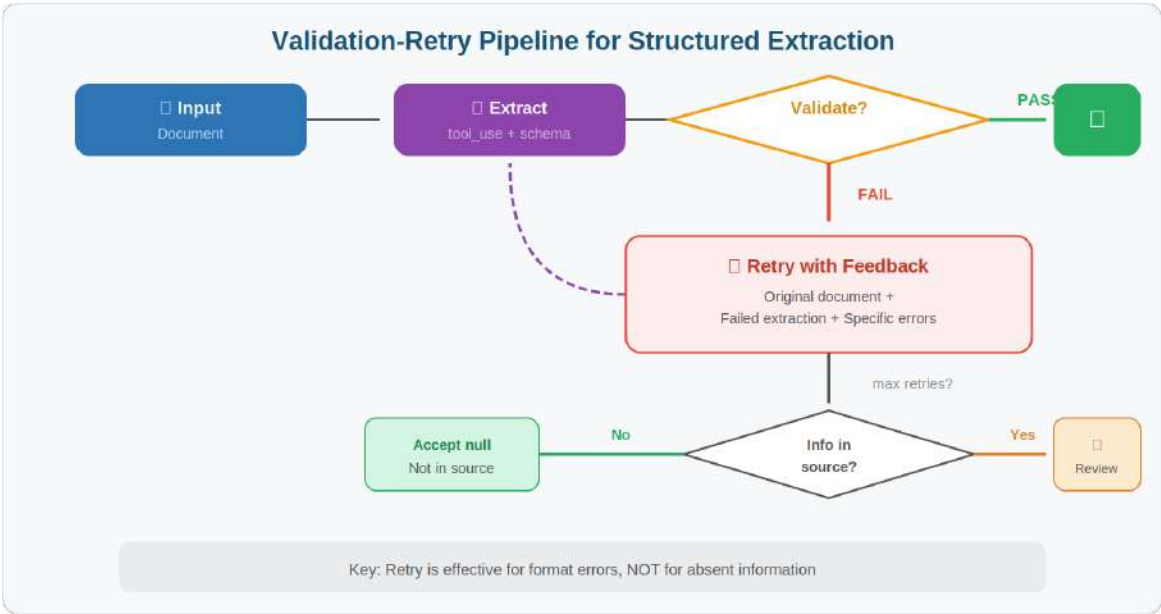
COMMON MISTAKE

tool_choice: "auto" does NOT guarantee a tool call. The model may return text instead. If you need guaranteed structured output, use "any" or forced selection.

Chapter 28: Validation, Retry & Feedback Loops

28.1 Retry-with-Error-Feedback

When extraction fails validation, the most effective retry strategy is to include the specific validation errors in the retry prompt. Send the original document, the failed extraction, and the exact validation errors. This guides the model toward correction by telling it precisely what went wrong.



28.2 Limits of Retry

Retries are effective for format mismatches and structural output errors. They are NOT effective when the required information is simply absent from the source document. If the document doesn't contain a phone number, no amount of retrying will produce a correct phone number.

The exam tests your ability to distinguish between retryable errors (the information exists but was extracted incorrectly) and non-retryable errors (the information doesn't exist in the source).

28.3 Feedback Loop Design

For ongoing improvement, add a **detected_pattern** field to structured findings. When developers dismiss a finding, you can analyze which code patterns trigger false positives and improve your prompts for those specific patterns.

28.4 Self-Correction Validation

Design extraction flows that facilitate self-correction. For example, extract both **calculated_total** (computed from line items) and **stated_total** (from the document). If they differ, flag the discrepancy. Add **conflict_detected** booleans for inconsistent source data.

Chapter 29: Batch Processing

29.1 Message Batches API

The Message Batches API offers 50% cost savings compared to real-time API calls. The tradeoff: processing can take up to 24 hours with no guaranteed latency SLA.

29.2 When to Use Batch Processing

Appropriate	Not Appropriate
Overnight technical debt reports	Pre-merge checks (developers waiting)
Weekly code audits	Real-time customer interactions
Nightly test generation	Interactive coding sessions
Batch document extraction	Blocking CI/CD pipeline steps

29.3 Batch API Limitations

The batch API does **not** support multi-turn tool calling within a single request. You cannot execute tools mid-request and return results. Each batch request is a single-turn interaction.

Use **custom_id** fields to correlate batch request/response pairs. When batch failures occur, resubmit only failed documents (identified by custom_id) with appropriate modifications.

29.4 Prompt Refinement Before Batch

Before batch-processing large volumes, refine your prompts on a sample set. This maximizes first-pass success rates and reduces costly iterative resubmissions.

Chapter 30: Multi-Instance Review Architectures

30.1 Self-Review Limitations

A model retains reasoning context from generation, making it less likely to question its own decisions. Self-review instructions or extended thinking do not fully overcome this limitation. An independent review instance (without the generator's reasoning context) is more effective at catching subtle issues.

30.2 Multi-Pass Review

For large code reviews, split into focused passes:

- **Per-file local analysis**: Analyze each file individually for bugs, style issues, and local logic errors
- **Cross-file integration pass**: Examine data flow between files, API contract consistency, and integration issues

This avoids attention dilution and eliminates contradictory findings (flagging a pattern in one file while approving it elsewhere).

Chapter 31: Domain 4 Practice Questions

Question 8: Your CI pipeline produces code review comments, but developers are ignoring them because 40% are false positives, mostly in the "comment accuracy" category. Legitimate findings in other categories (bugs, security) are also being ignored. What should you do?

A) Add "be conservative" to the system prompt to reduce false positives

B) Temporarily disable the "comment accuracy" category while improving its prompts separately

C) Increase the confidence threshold for all categories

D) Switch to a more capable model

Correct Answer: B

High false positive rates in one category undermine trust in all categories. Disabling the problematic category restores trust in legitimate findings while you improve the prompts. Option A is vague and ineffective. Option C would suppress real findings. Option D doesn't address the prompt quality issue.

Part VI: Domain 5 — Context Management & Reliability (15%)

Chapter 32: Managing Conversation Context

32.1 Progressive Summarization Risks

When conversations grow long, a common approach is to summarize earlier parts. But naive summarization loses critical details: numerical values get rounded or dropped, specific dates become “recently,” and customer-stated expectations become vague paraphrases.

The solution is to extract transactional facts (amounts, dates, order numbers, statuses) into a persistent “case facts” block that is included in every prompt, separate from the summarized conversation history. This ensures critical details survive summarization.

32.2 The “Lost in the Middle” Effect

Models reliably process information at the beginning and end of long inputs but may miss findings from middle sections. This is a well-documented attention pattern. To mitigate it:

- Place key findings summaries at the beginning of aggregated inputs
- Organize detailed results with explicit section headers
- Put critical information in prominent positions rather than burying it in the middle

32.3 Trimming Verbose Tool Outputs

Tool results accumulate in context and consume tokens disproportionately to their relevance. For example, an order lookup might return 40+ fields when only 5 are relevant to the current task. Trim verbose tool outputs to only relevant fields before they accumulate in context.

32.4 Maintaining Conversation History

In subsequent API requests, pass the complete conversation history to maintain coherence. Without history, Claude loses track of what was discussed, leading to repetitive questions and contradictory responses.

Chapter 33: Escalation & Ambiguity Resolution

33.1 Appropriate Escalation Triggers

The exam tests your judgment about when to escalate. Appropriate triggers include:

- Customer explicitly requests a human agent (honor immediately)
- Policy exceptions or gaps (the agent's policy doesn't cover the customer's specific situation)
- Inability to make meaningful progress after reasonable attempts

33.2 When NOT to Escalate

Do not escalate solely based on:

- Sentiment analysis (frustrated customers may have simple issues)
- Self-reported confidence scores (poorly calibrated—agents are often incorrectly confident on hard cases)
- Case complexity alone (straightforward cases with clear solutions should be resolved, even if they involve multiple steps)

33.3 Handling Customer Preferences

When a customer explicitly asks for a human, honor that request immediately without first attempting investigation. However, if the customer is simply frustrated but the issue is within the agent's capability, acknowledge the frustration while offering resolution. Only escalate if the customer reiterates their preference for a human.

33.4 Policy Gaps

Escalate when policy is ambiguous or silent on the customer's specific request. For example, if the customer asks for competitor price matching and your policy only addresses own-site price adjustments, this is a policy gap that requires human judgment.

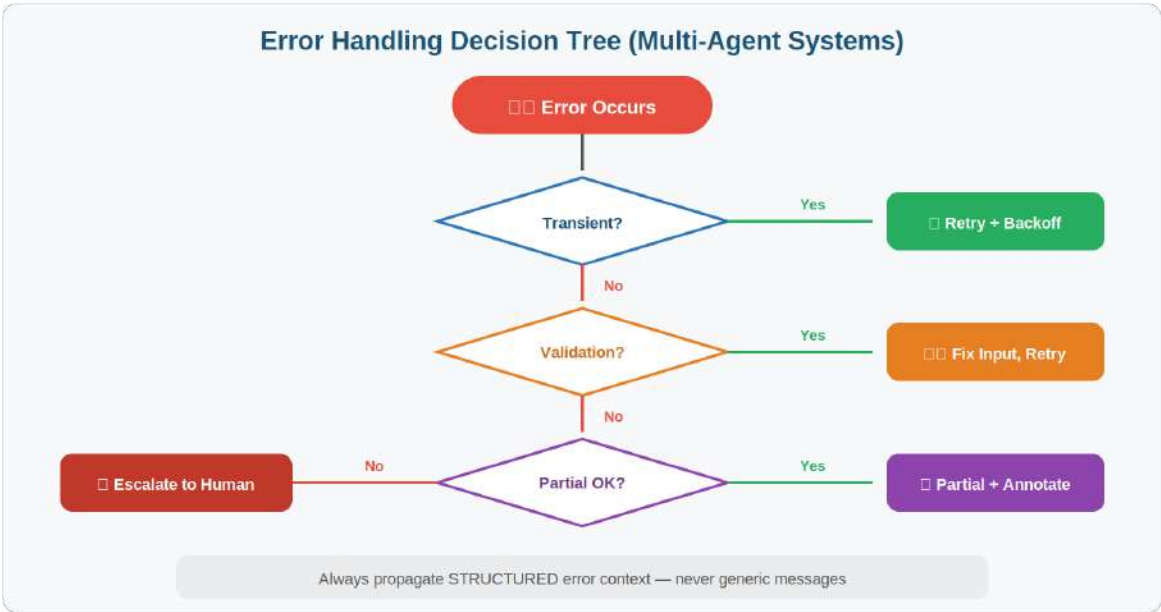
33.5 Multiple Customer Matches

When a customer lookup returns multiple matches, ask for additional identifiers (email, phone, order number) rather than selecting based on heuristics. Heuristic selection risks acting on the wrong account.

Chapter 34: Error Propagation in Multi-Agent Systems

34.1 Structured Error Context

When a subagent fails, it should return structured error context to the coordinator: failure type, what was attempted, partial results (if any), and potential alternative approaches. This enables the coordinator to make intelligent recovery decisions.



34.2 Anti-Patterns

- **Generic errors**: “Search unavailable” hides valuable context
- **Silent suppression**: Returning empty results as success prevents any recovery
- **Workflow termination**: Killing the entire pipeline on a single subagent failure is disproportionate

34.3 Local Recovery

Subagents should implement local recovery for transient failures (retry logic, backoff). Only propagate errors that cannot be resolved locally, along with what was attempted and any partial results.

34.4 Coverage Annotations

Synthesis output should include coverage annotations indicating which findings are well-supported versus which topic areas have gaps due to unavailable sources. This transparency helps the coordinator and end user assess the completeness of the research.

Chapter 35: Context Management in Large Codebases

35.1 Context Degradation

In extended sessions, models start giving inconsistent answers and referencing “typical patterns” rather than specific classes discovered earlier. This is context degradation—the model is losing track of specific details as the context window fills with exploration output.

35.2 Scratchpad Files

Have agents maintain scratchpad files recording key findings. When asking follow-up questions, the agent can reference the scratchpad rather than relying on degraded context. This counteracts context degradation by persisting key information outside the conversation.

35.3 Subagent Delegation for Exploration

Spawn subagents to investigate specific questions (“find all test files,” “trace refund flow dependencies”) while the main agent preserves high-level coordination. The subagent handles the verbose exploration, returning only a concise summary to the main agent.

35.4 /compact

Use **/compact** to reduce context usage during extended exploration sessions when the context fills with verbose discovery output. This summarizes the conversation to reclaim token budget.

35.5 Crash Recovery

Design crash recovery using structured agent state exports (manifests). Each agent exports its state to a known location, and the coordinator loads the manifest on resume, injecting the saved state into agent prompts.

Chapter 36: Human Review & Confidence Calibration

36.1 Aggregate Accuracy Metrics Can Be Misleading

A system reporting 97% overall accuracy may have 99% accuracy on common document types but 70% accuracy on rare ones. Always analyze accuracy by document type and field before automating high-confidence extractions.

36.2 Stratified Random Sampling

Implement stratified random sampling of high-confidence extractions. Even when the system reports high confidence, random samples should be reviewed to measure actual error rates and detect novel error patterns that the system hasn't encountered before.

36.3 Field-Level Confidence Scores

Have models output confidence scores for each extracted field, not just the overall extraction. Calibrate review thresholds using labeled validation sets. Route low-confidence extractions to human review, prioritizing limited reviewer capacity on the cases most likely to contain errors.

Chapter 37: Information Provenance

37.1 Preserving Source Attribution

Source attribution is commonly lost during summarization steps. When findings are compressed, the connection between claims and their original sources often disappears. The solution is structured claim-source mappings that the synthesis agent must preserve.

37.2 Handling Conflicting Sources

When credible sources report different statistics, annotate the conflict with source attribution rather than arbitrarily selecting one value. For example: “Source A reports 25% growth while Source B reports 18% growth.” Let the reader assess the discrepancy.

37.3 Temporal Data

Require publication or data collection dates in structured outputs. Without dates, temporal differences can be misinterpreted as contradictions. A statistic from 2020 and one from 2024 aren’t contradictory—they reflect different time periods.

37.4 Content-Appropriate Rendering

Different content types should be rendered appropriately: financial data as tables, news as prose, technical findings as structured lists. Don’t force everything into a uniform format.

Chapter 38: Domain 5 Practice Questions

Question 9: Your customer support agent achieves 55% first-contact resolution (target: 80%). Logs show it escalates straightforward cases while attempting complex policy-exception cases autonomously. What’s the most effective improvement?

A) Add explicit escalation criteria with few-shot examples to the system prompt

B) Have the agent self-report confidence scores and auto-escalate below a threshold

C) Deploy a separate classifier trained on historical tickets

D) Implement sentiment analysis to detect frustration and auto-escalate

Correct Answer: A

The root cause is unclear decision boundaries. Explicit criteria with examples directly address this. Option B fails because self-reported confidence is poorly calibrated. Option C is over-engineered. Option D confuses sentiment with complexity.

Question 10: Your extraction system reports 97% overall accuracy, but a client complains about frequent errors on handwritten invoices. What should you investigate?

A) Increase the model’s temperature to explore more extraction possibilities

B) Analyze accuracy by document type and field to find segment-specific issues

C) Add more training examples of handwritten invoices to the few-shot prompt

D) Switch to a vision-specialized model for all documents

Correct Answer: B

Aggregate accuracy masks segment-specific problems. Analyzing by document type and field will reveal whether handwritten invoices (and possibly other types) have significantly lower accuracy. Option C might help but doesn’t diagnose the scope of the problem first. Options A and D are not proportionate responses.

Part VII: Comprehensive Review & Exam Strategies

Chapter 39: Cross-Domain Decision Framework

Many exam questions require applying concepts from multiple domains simultaneously. This chapter provides decision frameworks that span domains.

39.1 The Enforcement Spectrum

When choosing how to enforce a behavior, consider the consequences of non-compliance:

Consequence Level	Enforcement	Example
Critical (financial, legal, safety)	Programmatic hooks/gates	Identity verification before refunds
Important (quality, consistency)	Prompt + few-shot examples	Code review format, escalation criteria
Preferred (style, tone)	Prompt instructions	Response tone, formatting preferences

39.2 The Tool Count Spectrum

The optimal number of tools per agent depends on the agent's role:

- **Specialized subagent**: 2–4 tightly scoped tools for its specific role
- **General-purpose agent**: 4–6 tools covering its domain
- **Coordinator**: Task tool plus any tools needed for its own analysis

More tools = more decision complexity = lower tool selection reliability. When an agent has 15+ tools, misrouting becomes common.

39.3 Error Handling Decision Tree

When an error occurs in a multi-agent system:

- 1. Is it transient? → Retry locally with backoff
- 2. Is it a validation error? → Fix input and retry locally
- 3. Is it a business/permission error? → Cannot retry; propagate structured context to coordinator
- 4. Does the coordinator have alternative approaches? → Try them
- 5. Is the task partially completable? → Proceed with partial results + coverage annotations
- 6. None of the above? → Escalate to human with full context

Chapter 40: Technologies Quick Reference

40.1 Claude Agent SDK

Concept	Key Details
Agent definitions	AgentDefinition with description, system prompt, allowedTools
Agentic loops	Check stop_reason: "tool_use" = continue, "end_turn" = stop
Hooks	PostToolUse (normalize results), tool call interception (enforce rules)
Subagent spawning	Via Task tool; allowedTools must include "Task"
Context	Subagents have isolated context; must explicitly pass data in prompt

40.2 Model Context Protocol (MCP)

Concept	Key Details
MCP servers	Project-level (.mcp.json) vs user-level (~/.claude.json)
MCP tools	Actions the agent can perform; need detailed descriptions
MCP resources	Content catalogs reducing exploratory tool calls
isError flag	Signals tool failure; include structured metadata
Environment vars	${GITHUB_TOKEN} in .mcp.json for secure credential management

40.3 Claude Code

Concept	Key Details
CLAUDE.md hierarchy	User (~/.claude/CLAUDE.md) > Project (.claude/CLAUDE.md) > Directory
.claude/rules/	Topic-specific rules with YAML frontmatter path-scoping
.claude/commands/	Project-scoped slash commands (version controlled)
.claude/skills/	SKILL.md with context: fork, allowed-tools, argument-hint
Plan mode	Complex tasks, architectural decisions, multi-file changes
Direct execution	Simple, well-scoped changes with clear scope
-p flag	Non-interactive mode for CI/CD pipelines
--output-format json	Machine-parseable structured output for CI
/compact	Reduce context during long sessions
--resume	Continue named sessions; fork_session for branches

40.4 Claude API

Concept	Key Details
tool_use	JSON schemas for guaranteed structured output (eliminates syntax errors)
tool_choice: "auto"	Model may call tool OR return text
tool_choice: "any"	Model MUST call a tool (can choose which)
Forced tool selection	{"type": "tool", "name": "..."} — must call specific tool
stop_reason	"tool_use" = continue loop; "end_turn" = stop loop
Message Batches API	50% cost savings, up to 24-hour processing, no latency SLA
custom_id	Correlate batch request/response pairs

Chapter 41: Common Exam Traps & How to Avoid Them

41.1 The “Stronger Prompt” Trap

When the exam describes a reliability problem (agent skipping steps, inconsistent tool selection), a tempting but wrong answer is “strengthen the system prompt.” Stronger prompts are appropriate for soft preferences, but for critical business logic, always prefer programmatic enforcement.

41.2 The “Bigger Model” Trap

Switching to a more capable model or larger context window rarely solves architectural problems. Attention dilution in large inputs, tool misrouting due to poor descriptions, and context degradation in long sessions are structural issues that require structural solutions.

41.3 The “Over-Engineered Solution” Trap

When the question asks for the “most effective first step,” prefer simple, high-leverage changes (better tool descriptions, explicit criteria) over complex infrastructure (ML classifiers, routing layers). Save complex solutions for when simpler approaches have been tried and failed.

41.4 The “Wrong Level” Trap

Multi-agent system problems often originate at a different level than where symptoms appear. Coverage gaps in synthesis output might trace back to narrow coordinator decomposition. Misrouted tool calls might trace back to poor tool descriptions rather than agent logic. Always identify the root cause level.

41.5 The “Sentiment = Complexity” Trap

Customer frustration does not correlate with case complexity. A frustrated customer may have a simple issue that’s easy to resolve. Sentiment-based escalation confuses emotional state with case difficulty.

41.6 The “Confidence Score” Trap

Self-reported confidence scores from LLMs are poorly calibrated. An agent that’s incorrectly confident on hard cases won’t be helped by a confidence-based routing system. Use explicit criteria and few-shot examples instead.

Chapter 42: Final Exam Preparation Checklist

Use this checklist to verify your readiness before taking the exam:

- Can you implement an agentic loop that correctly handles stop_reason “tool_use” and “end_turn”?
- Can you explain why subagents have isolated context and how to pass information between agents?
- Can you design a coordinator that dynamically selects subagents based on query complexity?
- Do you understand when to use programmatic enforcement vs. prompt-based guidance?
- Can you implement PostToolUse hooks for data normalization and tool call interception for compliance?
- Can you write tool descriptions that clearly differentiate similar tools?
- Do you understand the MCP isError flag and structured error metadata?
- Can you explain the difference between .mcp.json (project) and ~/.claude.json (user)?
- Can you configure CLAUDE.md at user, project, and directory levels?
- Do you know when to use .claude/rules/ with glob patterns vs. subdirectory CLAUDE.md?
- Can you create slash commands and skills with appropriate frontmatter?
- Can you explain when plan mode vs. direct execution is appropriate?
- Do you understand the -p flag for CI/CD and --output-format json for structured output?
- Can you design JSON schemas with optional/nullable fields for extraction tools?
- Do you understand tool_choice options: auto, any, and forced selection?
- Can you implement retry-with-error-feedback loops?
- Do you know when batch processing is appropriate vs. real-time API calls?
- Can you explain the “lost in the middle” effect and how to mitigate it?
- Do you understand escalation triggers: customer request, policy gaps, inability to progress?
- Can you design human review workflows with confidence calibration and stratified sampling?
- Do you understand claim-source mappings and how to handle conflicting sources?

FINAL TIP

The exam tests practical judgment, not memorization. For each question, ask yourself: What is the root cause? What is the simplest solution that addresses it? What are the tradeoffs? When in doubt, prefer

> the solution that provides deterministic guarantees for critical operations and proportionate complexity for the problem at hand.

Good luck on your exam! With thorough preparation using this handbook and hands-on practice, you are well-equipped to demonstrate your expertise in building production-grade applications with Anthropic’s Claude platform.

Part VIII: Detailed Scenario Walkthroughs

This section walks through each of the six exam scenarios in depth, exploring the architecture decisions, common pitfalls, and exam question patterns you should expect for each.

Chapter 43: Scenario Deep-Dive — Customer Support Resolution Agent

43.1 Architecture Overview

The customer support agent is built with the Claude Agent SDK and connects to four MCP tools: get_customer (identity verification), lookup_order (order details), process_refund (financial operations), and escalate_to_human (human handoff). The architecture is a single-agent system with tool access, not a multi-agent system.

The core challenge is balancing autonomous resolution (80%+ target) with appropriate escalation. The agent must handle high-ambiguity requests where multiple interpretations are possible, multi-concern requests where the customer has several issues in one message, and edge cases where policy is unclear.

43.2 Identity Verification Flow

The most tested pattern in this scenario is identity verification before financial operations. The exam repeatedly tests whether you understand that this requires programmatic enforcement, not just prompt instructions.

The flow should be: (1) Customer provides some identifying information, (2) get_customer is called to verify identity, (3) Only after verified customer ID is obtained can lookup_order and process_refund be called. A programmatic prerequisite gate ensures steps 2 and 3 cannot be reordered.

Why this matters: In production, if the agent skips verification, it might process a refund for the wrong customer. Even a 5% failure rate means real financial losses. Programmatic gates reduce this to 0%.

43.3 Multi-Concern Request Handling

Customers often combine multiple issues in one message: “I want to return order #1234, and also I was charged twice for order #5678, and can you update my shipping address?” The agent must decompose this into distinct items, investigate each one (potentially in parallel using shared customer context), and synthesize a unified resolution.

The exam tests whether you understand the decomposition pattern: identify each concern, investigate them using the appropriate tools, and respond comprehensively. A common mistake is the agent addressing only the first concern and ignoring the rest.

43.4 Structured Handoff Summaries

When the agent escalates, the human agent typically cannot see the conversation transcript. The AI must compile a structured summary:

```
{
  "customer_id": "CUST-12345",
  "customer_name": "Jane Smith",
  "issue_summary": "Requesting refund for damaged item",
  "root_cause": "Product arrived with visible damage, photos provided",
  "actions_taken": ["Verified identity", "Confirmed order #ORD-789", "Reviewed
damage photos"],
  "refund_amount": "$150.00",
  "recommended_action": "Approve full refund - clear damage evidence",
  "escalation_reason": "Refund exceeds automated limit of $100"
}
```

43.5 Common Exam Questions for This Scenario

Expect questions about: (1) How to enforce tool call ordering (programmatic gates), (2) When to escalate vs. resolve (explicit criteria + few-shot examples), (3) How to handle multiple customer matches (ask for additional identifiers), (4) Tool description improvements for reducing misrouting, (5) PostToolUse hooks for data normalization.

Chapter 44: Scenario Deep-Dive — Code Generation with Claude Code

44.1 Team Configuration Architecture

This scenario focuses on configuring Claude Code for a development team. The key architectural decisions involve: where to put shared vs. personal configurations, how to organize conventions for different code areas, and when to use plan mode vs. direct execution.

A well-configured team setup includes: project-level CLAUDE.md with universal coding standards, .claude/rules/ files with path-scoping for different code areas (API handlers, React components, database models, test files), project-scoped slash commands for common workflows (/review, /test, /deploy), and .mcp.json for shared tool integrations.

44.2 Configuration Hierarchy Deep-Dive

The three-level hierarchy serves different purposes. User-level settings (~/.claude/CLAUDE.md) are for personal preferences like preferred editor style or personal shortcuts. These are NOT shared with the team. Project-level settings (.claude/CLAUDE.md) are for team-wide standards. Directory-level settings (subdirectory CLAUDE.md files) are for area-specific context.

A critical diagnostic scenario: a new team member joins and doesn't get the same behavior. The cause is usually that instructions are in user-level rather than project-level configuration. The fix is to move shared instructions to project-level configuration.

44.3 Plan Mode Decision Framework

The exam presents tasks of varying complexity and asks you to choose between plan mode and direct execution. Here's a detailed framework:

Task	Approach	Rationale
Fix typo in error message	Direct execution	Single file, clear scope, no ambiguity
Add input validation to one function	Direct execution	Well-scoped, single change point
Migrate from REST to GraphQL	Plan mode	Multi-file, architectural decisions, multiple approaches
Restructure monolith to microservices	Plan mode	Dozens of files, dependency analysis, service boundaries
Add date validation to a form field	Direct execution	One component, clear requirement
Implement new authentication system	Plan mode	Security implications, multiple integration points
Update a dependency version	Direct execution (usually)	Unless it has breaking changes affecting many files
Legacy codebase test coverage	Plan mode	Map structure first, prioritize, then implement

		iteratively

44.4 Skills Configuration Example

A skill for automated code review might be configured as:

```
---
context: fork
allowed-tools:
  - Read
  - Grep
  - Glob
argument-hint: "Path to file or directory to review"
---
# Code Review Skill
Review the specified code for bugs, security issues, and style violations...
```

The **context: fork** option is critical here because the review produces verbose analysis output. Running it in a fork prevents this output from polluting the main conversation context. The **allowed-tools** restriction prevents the review skill from accidentally modifying files (no Write or Edit tool).

Chapter 45: Scenario Deep-Dive — Multi-Agent Research System

45.1 System Architecture

The multi-agent research system has four specialized subagents coordinated by a central coordinator. The coordinator receives the research topic, decomposes it into subtasks, delegates to appropriate subagents, collects results, evaluates quality, and may iterate if coverage is insufficient.

Understanding the information flow is essential:

- 1. User submits research topic to the coordinator
- 2. Coordinator decomposes topic into research subtasks
- 3. Coordinator delegates subtasks to web search and document analysis subagents (potentially in parallel)
- 4. Search/analysis subagents return structured findings with source metadata
- 5. Coordinator passes combined findings to synthesis subagent
- 6. Synthesis subagent produces a draft report
- 7. Coordinator evaluates the report for coverage gaps
- 8. If gaps exist, coordinator generates targeted follow-up queries and returns to step 3
- 9. When coverage is sufficient, the report subagent formats the final output

45.2 Context Passing Between Agents

Each subagent starts fresh with no memory of previous interactions. The coordinator must explicitly pass all relevant context. For the synthesis subagent, this means including:

- The original research question
- All findings from the web search subagent, with source URLs and dates
- All findings from the document analysis subagent, with document names and page numbers
- Any quality criteria (minimum source count, recency requirements)

The format should separate content from metadata. A structured format like JSON or XML preserves the boundary between claims and their sources, enabling accurate attribution in the final report.

45.3 Parallel Execution Optimization

When the coordinator emits multiple Task tool calls in a single response, the subagents execute in parallel. This is a significant latency optimization. In the research system, the web search subagent and document analysis subagent can often run simultaneously because they access different data sources.

However, the synthesis subagent cannot run in parallel with the search/analysis subagents because it depends on their output. Understanding which agents can run in parallel (independent data sources) vs. which must run sequentially (output dependencies) is a key exam skill.

45.4 Handling Conflicting Sources

When two credible sources report different statistics (e.g., one says “25% growth” and another says “18% growth”), the synthesis subagent should NOT arbitrarily pick one. Instead, it should:

- Annotate the conflict: “Source A (2024 report) states 25% while Source B (2023 study) reports 18%”
- Include publication dates to enable temporal interpretation
- Preserve both values with full source attribution
- Let the reader assess the discrepancy

45.5 Coverage Gap Detection

The coordinator evaluates synthesis output by checking coverage annotations. The synthesis subagent should produce output that indicates which topics are well-covered, which have limited sources, and which have no coverage due to unavailable data. This enables the coordinator to make targeted re-delegation decisions.

Chapter 46: Scenario Deep-Dive — Developer Productivity

46.1 Codebase Exploration Strategy

This scenario tests your understanding of built-in tools and incremental codebase understanding. The key principle is: build understanding from entry points outward, rather than trying to read everything.

Step-by-step exploration strategy:

- 1. **Grep** to find entry points: main functions, API routes, event handlers
- 2. **Read** the entry point files to understand the top-level structure
- 3. **Grep** for imports from entry point files to trace dependencies
- 4. **Glob** to find related files: tests, configs, documentation
- 5. Build a mental model, then dive deeper into specific areas of interest

46.2 MCP Server Integration for Developer Tools

Developer productivity tools often integrate with external services like Jira, GitHub, or internal documentation systems via MCP servers. The exam tests your understanding of:

- When to use community MCP servers vs. custom implementations
- How to configure project-level vs. user-level MCP servers
- How to enhance tool descriptions so the agent prefers MCP tools over built-in tools when appropriate

46.3 Tracing Function Usage

A common development task is tracing how a function is used across a codebase, including through wrapper modules. The approach is: (1) Use Grep to find the function definition, (2) Identify all exported names from the module containing the function, (3) Search for each exported name across the codebase to find all usage sites.

This multi-step approach is necessary because wrapper modules may re-export functions under different names, and direct search for the original function name would miss indirect usages.

Chapter 47: Scenario Deep-Dive — CI/CD Integration

47.1 Pipeline Architecture

A Claude Code CI/CD integration typically includes: (1) A pre-merge check that runs on every PR, analyzing code for bugs and security issues, (2) A test generation step that suggests new tests for changed code, (3) A post-merge analysis for technical debt tracking.

Critical architectural decision: the pre-merge check must use real-time API calls (blocking, needs fast results), while the technical debt analysis can use the Message Batches API (non-blocking, 50% cost savings, results needed by next morning).

47.2 Reducing False Positives in CI Reviews

False positives are the number one adoption blocker for CI code review tools. Every false positive costs developer attention and erodes trust. Strategies for reducing false positives:

- Define specific review criteria: “Flag bugs and security issues; skip minor style and local patterns”
- Provide few-shot examples showing acceptable patterns vs. genuine issues
- Temporarily disable high false-positive categories
- Include detected_pattern in findings to enable systematic false positive analysis
- Use concrete severity criteria with code examples, not vague confidence scores

47.3 Test Generation Best Practices

When generating tests in CI:

- Provide existing test files in context so Claude doesn’t suggest duplicate scenarios
- Document testing standards and available fixtures in CLAUDE.md
- Focus on valuable tests: edge cases, error handling, integration points
- Avoid low-value tests: trivial getter/setter tests, obvious happy-path tests that add no insight

47.4 Batch Processing for Overnight Analysis

The Message Batches API is ideal for overnight analysis tasks. Design considerations:

- Submit batches with enough lead time (account for up to 24-hour processing)
- Use custom_id to correlate results with source files
- Handle failures gracefully: resubmit only failed items with modifications
- Refine prompts on a sample set before submitting large batches

Chapter 48: Scenario Deep-Dive — Structured Data Extraction

48.1 Schema Design Patterns

The extraction schema is the foundation of the entire pipeline. Design decisions here ripple through validation, retry, and human review. Key patterns:

Required vs. Optional Fields: Make fields required only when the information is guaranteed to exist in every document. For fields that might be absent, use nullable/optional types. This prevents the model from fabricating values to satisfy required field constraints.

Enum + Other Pattern: For categorical fields, list known categories as enum values and include an "other" option with a detail string. This captures known categories cleanly while handling unexpected ones:

```
{
  "document_type": {
    "type": "string",
    "enum": ["invoice", "receipt", "contract", "letter", "other"]
  },
  "document_type_detail": {
    "type": "string",
    "description": "If document_type is 'other', describe the type here"
  }
}
```

Unclear Values: Include an "unclear" enum value for genuinely ambiguous cases. This gives the model an honest option rather than forcing a guess that might be wrong.

48.2 Validation-Retry Pipeline

The validation-retry pipeline follows this flow:

- 1. Extract data using tool_use with JSON schema
- 2. Validate the extraction (schema compliance + semantic checks)
- 3. If validation fails, append specific errors to the prompt and retry
- 4. If retry fails again, check: is the information present in the source?
- 5. If absent from source, accept null/unclear values
- 6. If present but consistently misextracted, flag for human review

48.3 Semantic Validation

Schema compliance (guaranteed by tool_use) eliminates syntax errors but not semantic errors. Implement additional validation:

- **Cross-field consistency**: Line items sum to stated total
- **Date logic**: Start date before end date, dates within reasonable range
- **Unit consistency**: Amounts in expected currency, quantities non-negative
- **Reference validation**: Referenced document IDs match expected patterns

48.4 Human Review Routing

Not all extractions need human review. Implement a confidence-based routing strategy:

- High confidence + all validations pass → Auto-approve (with random sampling)
- Medium confidence or minor validation issues → Quick review queue
- Low confidence or conflicting source data → Detailed review queue
- Ambiguous/contradictory sources → Expert review queue

Calibrate these thresholds using labeled validation sets. Field-level confidence is more useful than document-level confidence because different fields have different error rates.

Part IX: Extended Practice Questions

This section contains 30 additional practice questions covering all five domains. Each question includes a detailed explanation of why the correct answer is right and why each distractor is wrong.

Questions 11–15: Agentic Architecture

Question 11: Your coordinator agent always invokes all four subagents (search, analyze, synthesize, report) regardless of query complexity. For simple factual questions, this adds unnecessary latency and cost. How should you modify the coordinator's behavior?

A) Add conditional logic in the orchestration code that routes simple queries to a single search subagent

B) Design the coordinator's prompt to analyze query requirements and dynamically select which subagents to invoke

C) Create a separate "fast path" agent for simple queries and a routing classifier to direct queries

D) Reduce the number of subagents by combining search and analyze into one

Correct Answer: B

The coordinator should dynamically select subagents based on query complexity. This keeps the model-driven architecture intact while allowing proportionate resource usage. Option A moves intelligence out of the coordinator into hardcoded logic. Option C adds infrastructure complexity. Option D loses specialization benefits for complex queries.

Question 12: In your multi-agent system, the coordinator spawns the search subagent, waits for results, then spawns the analysis subagent, waits, then spawns synthesis. The total latency is the sum of all three subagent execution times. How can you reduce latency?

A) Use a faster model for each subagent

B) Spawn the search and analysis subagents in parallel by emitting multiple Task tool calls in a single coordinator response

C) Combine all subagents into a single agent with access to all tools

D) Pre-cache search results to eliminate the search step

Correct Answer: B

Parallel execution via multiple Task tool calls in one response reduces latency from the sum to the maximum of the parallel steps. Search and analysis can run simultaneously if they access independent data sources. Option A may help marginally but doesn't address the architectural bottleneck. Option C loses specialization. Option D isn't feasible for dynamic queries.

Question 13: Your agent uses a PostToolUse hook that normalizes all dates to ISO 8601 format. However, you notice the agent is still confused by dates because different MCP tools return timestamps in Unix epoch seconds vs. ISO 8601 strings. What's the most likely issue?

A) The hook is not applied to all MCP tools

B) The hook runs after the model processes the data, so the model sees the raw format first

C) The agent's system prompt doesn't mention date handling

D) ISO 8601 format is ambiguous about timezones

Correct Answer: A

PostToolUse hooks intercept tool results before the model processes them (not after, eliminating B). If the hook isn't applied to all tools, some will still return non-normalized formats. The most likely issue is that the hook's tool filter doesn't include all relevant MCP tools. Options C and D are secondary concerns that wouldn't cause the described behavior.

Question 14: You're designing a workflow where identity verification must happen before any financial operation. Which implementation provides the strongest guarantee?

A) System prompt: "You MUST call get_customer and verify identity before calling process_refund or lookup_order"

B) System prompt with 3 few-shot examples showing the correct verification flow

C) A tool call interception hook that checks whether get_customer has been called and returned a verified ID before allowing process_refund

D) Setting tool_choice to force get_customer on the first turn

Correct Answer: C

A tool call interception hook provides a deterministic guarantee that process_refund cannot execute without prior verification. Options A and B are probabilistic. Option D only forces get_customer on the first turn but doesn't prevent the agent from calling process_refund without verification in subsequent turns.

Question 15: Your research coordinator decomposes “impact of climate change on agriculture” into subtasks about crop yields, water availability, and soil quality. The final report is comprehensive on these topics but completely omits livestock, supply chains, and food security. What is the root cause?

A) The search subagent’s queries are too narrow

B) The synthesis subagent is filtering out relevant findings

C) The coordinator’s task decomposition is too narrow, missing major aspects of the topic

D) The document analysis subagent is not processing enough sources

Correct Answer: C

The coordinator decomposed the topic too narrowly (only direct agricultural impacts), missing livestock, supply chains, and food security dimensions. The subagents worked correctly within their assigned scope. The problem is at the coordinator level.

Questions 16–20: Tool Design & MCP

Question 16: Your agent has access to 18 tools and frequently selects the wrong one, especially for queries that could plausibly match 3-4 different tools. What is the most effective solution?

A) Add more detailed examples to the system prompt showing correct tool selection

B) Reduce the number of tools available to the agent to 4-5 well-scoped tools per agent role

C) Implement a pre-processing layer that selects the tool before Claude sees the query

D) Switch to a more capable model that handles large tool sets better

Correct Answer: B

Giving an agent too many tools (18 vs 4-5) degrades tool selection reliability by increasing decision complexity. Reducing to well-scoped tools per role addresses the root cause. Option A adds token overhead without fixing the fundamental problem. Options C and D add complexity without addressing the design issue.

Question 17: Your MCP tool returns this error when a customer's refund request exceeds the automated limit: {"error": "Operation failed"}. The agent responds to the customer with "I'm sorry, an error occurred." How should you improve the error response?

A) Add retry logic with exponential backoff

B) Return structured error metadata including errorCategory, isRetryable: false, and a customer-friendly explanation

C) Add error handling in the system prompt telling the agent how to interpret generic errors

D) Log the error and silently proceed with the next step

Correct Answer: B

Structured error metadata enables the agent to understand the error type (business rule violation, not transient), know it's not retryable, and communicate the specific issue to the customer. Option A is wrong because business errors aren't retryable. Option C can't fix what the prompt can't see. Option D suppresses the error.

Question 18: You're configuring MCP servers for your team. A GitHub integration should be available to all developers, and you're experimenting with a custom analytics tool personally. Where should each be configured?

A) Both in .mcp.json for consistency

B) GitHub in .mcp.json, analytics in ~/.claude.json

C) Both in ~/.claude.json to avoid repository changes

D) GitHub in CLAUDE.md, analytics in ~/.claude.json

Correct Answer: B

Shared team tools go in project-level .mcp.json (version controlled). Personal/experimental tools go in user-level ~/.claude.json (not shared). Option A exposes experimental tools to the whole team. Option C doesn't share the team tool. Option D uses the wrong file type for MCP configuration.

Question 19: Your document analysis subagent returns an empty result set. The coordinator can't tell whether the analysis found no relevant content (valid empty result) or whether the document service was down (access failure). How should you fix this?

A) Always retry empty results once in case of transient failure

B) Add a health check before each tool call

C) Distinguish between access failures and valid empty results in the tool's response format

D) Set a minimum result count and flag violations

Correct Answer: C

The tool should explicitly indicate whether it connected successfully but found nothing ("status: success, results: []") or whether it failed to connect ("isError: true, errorCategory: transient"). This enables the coordinator to make the right recovery decision. Option A wastes resources on valid empties. Options B and D don't solve the ambiguity.

Question 20: You have a search tool with the description "Searches for information" and an analysis tool with the description "Analyzes information." The agent frequently routes search queries to the analysis tool. What's the first fix?

A) Add a routing layer to intercept queries

B) Rename the tools and expand descriptions to clearly differentiate their purpose, inputs, outputs, and boundaries

C) Reduce the system prompt size to give the model more attention for tool descriptions

D) Add tool_choice: "auto" to let the model figure it out

Correct Answer: B

Vague, overlapping descriptions are the root cause. Expanding them to include purpose, expected inputs, outputs, example queries, and boundaries is the highest-leverage fix. tool_choice: "auto" is already the default and doesn’t fix the underlying ambiguity.

Questions 21–25: Claude Code & Prompt Engineering

Question 21: A developer creates a personal slash command in ~/.claude/commands/ and expects their teammate to have access to it after pulling the latest changes. Why doesn't the teammate see the command?

A) Slash commands require a Claude Code restart to take effect

B) User-scoped commands in ~/.claude/commands/ are not shared via version control

C) The teammate needs to explicitly import the command

D) Only skills, not slash commands, can be shared across team members

Correct Answer: B

Commands in ~/.claude/commands/ are personal and not version-controlled. To share with the team, move the command to .claude/commands/ in the project repository.

Question 22: You're running Claude Code in a CI pipeline. The pipeline hangs indefinitely with no output. The command is: claude "Review this PR for security issues." What's wrong?

A) The prompt is too short for meaningful analysis

B) The -p flag is missing, causing Claude Code to wait for interactive input

C) Claude Code needs a --ci flag for pipeline mode

D) The system needs the CLAUDE_HEADLESS=true environment variable

Correct Answer: B

The -p (--print) flag runs Claude Code in non-interactive mode. Without it, Claude Code waits for interactive input, causing the pipeline to hang. Options C and D reference non-existent features.

Question 23: Your code review prompt says "be conservative and only flag high-confidence issues." The review still produces too many false positives. What's the most effective improvement?

A) Change "be conservative" to "be very conservative"

B) Add a confidence threshold: "only flag issues with confidence above 90%"

C) Replace vague instructions with specific categorical criteria defining which issue types to flag vs. skip

D) Reduce the number of files reviewed per pass

Correct Answer: C

Vague instructions like “be conservative” and “high confidence” don’t improve precision because they’re undefined. Specific criteria (“Flag bugs and security issues; skip minor style, local patterns”) give the model clear, actionable boundaries. Self-reported confidence thresholds (B) are poorly calibrated.

Question 24: You need to extract structured data from invoices. Some invoices have a “purchase order number” field and some don’t. Your extraction schema has purchase_order_number as a required string field. What problem does this create?

A) The schema is too strict and will reject valid invoices

B) The model may fabricate a purchase order number to satisfy the required field constraint

C) The extraction will timeout on invoices without the field

D) The JSON output will have syntax errors

Correct Answer: B

When a field is required but the information isn’t in the source document, the model may fabricate a value (hallucinate) to satisfy the schema constraint. Making the field nullable/optional allows the model to honestly return null when the data isn’t present.

Question 25: You’re using the Message Batches API for a weekly security audit. Your manager wants to also use it for the pre-merge code review (currently using real-time API calls) to save costs. Should you agree?

A) Yes, the batch API works the same way with 50% savings

B) No, the batch API has no latency SLA and developers would have to wait up to 24 hours to merge

C) Yes, but only if you add polling to check for completion

D) No, the batch API doesn’t support code review tasks

Correct Answer: B

The batch API has up to 24-hour processing with no guaranteed latency SLA. This is unacceptable for blocking pre-merge checks. Keep real-time calls for blocking workflows; use batch API for non-blocking overnight/weekly tasks.

Questions 26–30: Context Management & Reliability

Question 26: During a long customer support conversation, the agent suddenly “forgets” the customer’s order number that was mentioned 15 messages ago and asks for it again. What’s the most effective solution?

A) Increase the model’s context window

B) Extract key transactional facts (order numbers, amounts, dates) into a persistent “case facts” block included in each prompt

C) Summarize the conversation more frequently

D) Tell the customer to repeat their order number

Correct Answer: B

A persistent “case facts” block ensures critical details survive conversation summarization and context window pressure. Regular summarization (C) risks losing the specific details. Increasing context (A) delays but doesn’t prevent the problem.

Question 27: Your multi-agent research system produces a report on healthcare costs. The report states “healthcare spending increased by 8%” without any source attribution. When you check, two sources reported different numbers: 6.5% and 9.2%. What went wrong?

A) The synthesis subagent averaged the two numbers

B) Source attribution was lost during summarization, and the synthesis subagent picked a middle ground

C) The search subagent didn’t find enough sources

D) The model hallucinated the 8% figure

Correct Answer: B

Source attribution is commonly lost during summarization steps. Without structured claim-source mappings, the synthesis agent has no way to preserve which number came from which source. The solution is to require subagents to output structured claim-source mappings that the synthesis agent must preserve. The 8% likely represents the model’s attempt to reconcile conflicting data without proper source tracking.

Question 28: Your extraction system has 97% overall accuracy. A new client sends mostly handwritten invoices and reports 30% error rates. Your automated confidence scoring shows high confidence for these extractions. What should you conclude?

A) The model is working correctly; handwritten invoices are inherently unreliable

B) The confidence scoring is miscalibrated for this document type; stratified accuracy analysis by document type is needed

C) The 97% metric is wrong and should be recalculated

D) The new client's documents are outside the system's supported format

Correct Answer: B

Aggregate accuracy (97%) masks segment-specific problems. The confidence scoring is calibrated on the typical document mix, not specifically on handwritten invoices. Stratified analysis would reveal the performance gap. This is a key exam concept: always verify accuracy by document type and field before trusting aggregate metrics.

Question 29: During an extended codebase exploration session, Claude starts giving vague answers like "this module likely follows typical patterns for..." instead of referencing specific classes it analyzed earlier. What is happening?

A) The model is running out of compute budget

B) Context degradation: the conversation has accumulated so much exploration output that earlier specific findings are being lost

C) The codebase has too many files for Claude to track

D) The model's temperature is too high

Correct Answer: B

Context degradation in extended sessions causes models to lose track of specific details and fall back to generic patterns. Solutions include using scratchpad files to persist key findings, spawning subagents for exploration (keeping the main conversation clean), and using /compact to reclaim context budget.

Question 30: A customer asks your support agent to match a competitor's price. Your company's policy covers price adjustments for own-site pricing changes but says nothing about competitor matching. What should the agent do?

A) Deny the request because the policy doesn't explicitly allow it

B) Approve it because it's in the spirit of price matching

C) Escalate to a human because the policy is silent on this specific request

D) Ask the customer for proof of the competitor’s price before deciding

Correct Answer: C

When policy is ambiguous or silent on the customer’s specific request, the agent should escalate. The policy covers own-site adjustments but says nothing about competitor matching—this is a policy gap that requires human judgment. The agent shouldn’t make policy decisions that aren’t explicitly authorized.

Additional Practice Questions: Mixed Domain (Q31–80)

These 50 questions are designed to simulate exam difficulty. They span all five domains and frequently combine concepts across domains, just like the real exam.

Q31 — Domain 1: Agentic Architecture

Your agentic loop sets MAX_ITERATIONS = 5 as the primary stopping mechanism. On a complex customer issue requiring 7 tool calls, the loop terminates early and gives an incomplete answer. What is the fundamental design flaw?

A) MAX_ITERATIONS is too low; increase it to 20

B) The loop should use stop_reason as the primary control; iteration caps should only be safety nets

C) Add a “continue” flag that the model sets when it needs more iterations

D) Split the task into smaller sub-tasks that each require fewer iterations

Correct Answer: B

Iteration caps as the primary stopping mechanism interfere with Claude’s natural reasoning flow. The correct design uses stop_reason ("tool_use" to continue, "end_turn" to stop) as primary control. Iteration caps are safety nets for runaway loops, not flow control. Option A treats the symptom. Options C and D add unnecessary complexity.

Q32 — Domain 1: Multi-Agent

Your coordinator spawns a search subagent, then passes its findings to a synthesis subagent. The synthesis output contains claims with no source attribution. Where is the most likely breakdown?

A) The search subagent didn’t include source URLs in its output

B) The coordinator didn’t pass source metadata when delegating to the synthesis subagent

C) The synthesis subagent’s prompt doesn’t require source attribution

D) All of the above could contribute, but B is the most common root cause

Correct Answer: D

While all three can contribute, B is the most common root cause in practice. Even if the search subagent returns source metadata, the coordinator may strip it when constructing the synthesis

prompt. The coordinator is responsible for explicitly passing all necessary context, including metadata.

Q33 — Domain 2: Tool Design

You have a tool called "manage_account" that handles password resets, email changes, subscription upgrades, and profile updates. The agent frequently calls it for the wrong operation. What's the best fix?

A) Add more detailed examples of each operation to the tool's description

B) Split it into four purpose-specific tools: reset_password, update_email, upgrade_subscription, update_profile

C) Add a required "operation" parameter to disambiguate

D) Create a routing layer that maps user intent to the correct operation

Correct Answer: B

A single generic tool that handles many operations forces Claude to figure out the sub-operation, increasing error rates. Splitting into purpose-specific tools with clear descriptions gives Claude unambiguous choices. Option C still requires the model to correctly select the operation value. Options A and D add complexity without fixing the root issue.

Q34 — Domain 2: MCP Error Handling

Your MCP tool for order lookup sometimes times out (transient error) and sometimes returns "order not found" (valid result). Both currently return {"error": true, "message": "..."}. The agent retries valid "not found" results and tells customers to wait for timeouts. How do you fix this?

A) Add a 2-second delay before all order lookups to prevent timeouts

B) Distinguish error categories: isRetryable: true for timeouts, isRetryable: false with a valid-empty flag for "not found"

C) Have the agent parse the error message text to determine the type

D) Always retry twice regardless of error type

Correct Answer: B

Structured error metadata (errorCategory, isRetryable) enables the agent to make correct recovery decisions. Timeouts should be retried; “not found” should not. Parsing error message text (C) is fragile. Always retrying (D) wastes resources on valid results.

Q35 — Domain 3: Configuration

Your team’s CLAUDE.md says “Use Prettier for formatting.” A new developer’s personal ~/.claude/CLAUDE.md says “Use ESLint with Airbnb style.” When the new developer uses Claude Code, which formatting standard applies?

A) The project-level CLAUDE.md always takes precedence

B) The user-level CLAUDE.md takes precedence over project-level

C) Both are loaded; the model sees potentially conflicting instructions

D) Only the project-level CLAUDE.md is loaded when inside a project

Correct Answer: C

Both user-level and project-level CLAUDE.md files are loaded simultaneously. When they conflict, the model sees both instructions and may behave inconsistently. The solution is to move team standards to project-level and keep user-level for non-conflicting personal preferences.

Q36 — Domain 3: Skills

A developer creates a code analysis skill that reads dozens of files and produces verbose output. After running it, the main conversation’s context is nearly full, leaving no room for follow-up work. What SKILL.md option would prevent this?

A) allowed-tools: [Read, Grep, Glob]

B) context: fork

C) argument-hint: “Directory to analyze”

D) max-tokens: 2000

Correct Answer: B

context: fork runs the skill in an isolated sub-agent context, returning only the summary to the main conversation. This prevents verbose exploration output from consuming the main conversation’s context budget. Options A and C are useful but don’t address context pollution. Option D doesn’t exist.

Q37 — Domain 4: Prompt Engineering

Your code review prompt produces findings with 4 severity levels (critical, high, medium, info). Developers complain that severity assignments are inconsistent: similar issues get different severities. What's the most effective fix?

A) Reduce to 2 severity levels (critical, non-critical)

B) Define explicit criteria for each severity level with concrete code examples showing boundary cases

C) Add a calibration step where the model re-evaluates all findings and normalizes severity

D) Let developers override severity levels in their feedback

Correct Answer: B

Inconsistent severity comes from vague criteria. Concrete examples showing "this is critical because..." vs. "this is medium because..." give the model clear calibration points. Option A loses useful granularity. Options C and D add complexity without fixing the root cause.

Q38 — Domain 4: Structured Output

You're extracting data from medical documents. Some documents list medications with dosages; others mention medications without dosages. Your schema has dosage as a required string field. What problem will you see?

A) The extraction will fail validation on documents without dosages

B) The model will fabricate dosage values to satisfy the required field

C) The model will skip medications that lack dosages entirely

D) The extraction speed will decrease due to increased reasoning

Correct Answer: B

When a required field's data isn't in the source, the model may hallucinate a value rather than violate the schema constraint. Making dosage nullable lets the model honestly return null. This is a critical schema design principle tested frequently on the exam.

Q39 — Domain 5: Context Management

Your customer support agent is in a long conversation (30+ messages). The customer mentions their order number in message #3. By message #25, the agent asks for the order number again. Summarization is active. What's the best prevention?

A) Disable summarization entirely

B) Increase the context window size

C) Extract transactional facts into a persistent "case facts" block maintained across summarization cycles

D) Store the conversation in a database and retrieve it on demand

Correct Answer: C

A persistent case facts block (order numbers, amounts, customer IDs, dates) survives summarization because it's maintained separately from the conversation history. Option A causes the context to overflow. Option B delays but doesn't prevent the problem. Option D adds unnecessary infrastructure.

Q40 — Domain 5: Escalation

A customer writes: "This is ridiculous! I've been waiting 3 weeks for my refund! I want to speak to a manager!" Your agent has verified the refund is processing normally and should arrive in 2 days. What should the agent do?

A) Immediately escalate because the customer explicitly requested a manager

B) Inform the customer about the refund status and offer to escalate if they still want a manager

C) Ignore the escalation request and just provide the status update

D) Escalate with a note that the customer is frustrated

Correct Answer: A

When a customer explicitly requests a human agent or manager, honor that request immediately. Don't attempt to resolve first. The customer's explicit preference takes priority. Include a structured handoff summary with the refund status so the human agent has full context.

Q41 — Domain 1: Hooks

Your PostToolUse hook normalizes currency values from different MCP tools (some return “$100”, others “100.00 USD”, others “10000” in cents). After adding the hook, the agent’s financial calculations become more reliable. What property of hooks makes this effective?

A) Hooks run before the model processes the data, so the model sees consistent formats

B) Hooks modify the model’s system prompt to include currency formatting rules

C) Hooks filter out tools that return non-standard formats

D) Hooks cache the results for faster retrieval

Correct Answer: A

PostToolUse hooks intercept tool results after execution but before the model processes them. By normalizing the data before the model sees it, the model receives consistent formats without needing to handle multiple format variations. This is deterministic normalization, not probabilistic.

Q42 — Domain 2: Built-in Tools

A developer asks Claude Code to “find all files where the calculateTax function is used.” Which built-in tool is most appropriate?

A) Glob — to find files matching a name pattern

B) Read — to read each file and check for the function

C) Grep — to search file contents for the function name

D) Bash — to run grep from the command line

Correct Answer: C

Grep searches file contents for patterns, making it ideal for finding where a function is used. Glob matches file paths/names, not content. Read requires knowing which files to check. Bash could work but Grep is the purpose-built tool for content search.

Q43 — Domain 3: CI/CD

Your CI pipeline runs Claude Code to review PRs. The same Claude session that generates suggested fixes also reviews the changes for correctness. Code quality has not improved. Why?

A) The prompt isn’t detailed enough for effective review

B) The model retains reasoning context from generation, making it less likely to question its own decisions

C) CI pipelines can't run Claude Code effectively

D) The model needs more context about the codebase

Correct Answer: B

Self-review is an anti-pattern. The model retains reasoning context from generation, creating blind spots. Use an independent review instance without the generator's reasoning context for more effective reviews.

Q44 — Domain 4: Few-Shot Examples

Your extraction prompt has 15 few-shot examples covering various document types. Token usage is high and extraction quality hasn't improved over the version with 3 examples. What should you do?

A) Add even more examples to cover edge cases

B) Reduce to 2–4 targeted examples focusing on ambiguous scenarios that require judgment

C) Remove all examples and rely on detailed instructions

D) Randomize which examples are included per request

Correct Answer: B

Research shows 2–4 targeted examples achieve the best balance of quality and efficiency. Too many examples waste tokens without improving quality. Focus examples on genuinely ambiguous cases where reasoning must be demonstrated, not straightforward extractions.

Q45 — Domain 5: Provenance

Your research system produces a report stating "healthcare spending grew 8%" but two sources cite different figures: Source A says 6.5% and Source B says 9.2%. The 8% appears to be an average. What went wrong?

A) The synthesis agent averaged the conflicting numbers instead of preserving both with attribution

B) The search agent found unreliable sources

C) The model hallucinated a different number

D) The coordinator didn’t specify which source to prioritize

Correct Answer: A

When sources conflict, the correct behavior is to annotate the conflict with full attribution: “Source A reports 6.5%; Source B reports 9.2%.” Averaging or selecting a middle ground hides the discrepancy and reduces the report’s trustworthiness.

Q46 — Domain 1: Task Decomposition

You’re asked to review a PR touching 45 files across 8 packages. A single-pass review produces inconsistent depth: detailed feedback on some files, superficial on others, and contradictory feedback on identical patterns. What’s the solution?

A) Switch to a model with a larger context window

B) Split into per-file local analysis passes, then a cross-file integration pass

C) Ask the developer to submit smaller PRs

D) Only review the most critical files and skip the rest

Correct Answer: B

Attention dilution causes inconsistent depth in large reviews. Splitting into focused per-file passes ensures consistent analysis, then a cross-file pass catches integration issues. This is the prompt chaining pattern applied to code review.

Q47 — Domain 2: tool_choice

You need to extract metadata from a document but the model sometimes returns a text response instead of using the extraction tool. How do you guarantee structured output?

A) Add “You MUST use the extraction tool” to the system prompt

B) Set tool_choice to "auto" and hope the model complies

C) Set tool_choice to {"type": "tool", "name": "extract_metadata"} to force the specific tool

D) Set tool_choice to "any" and include only the extraction tool

Correct Answer: C

Forced tool selection guarantees the model calls the specified tool. tool_choice: "auto" allows text responses. tool_choice: "any" forces a tool call but allows choosing between multiple tools. Option

D would also work if there's only one tool, but C is more explicit and correct regardless of tool count.

Q48 — Domain 3: Path-Specific Rules

Your monorepo has infrastructure code in /infra/terraform/, /infra/ansible/, and /deploy/kubernetes/. You want Claude to follow different conventions for each. What's the most maintainable approach?

A) One root CLAUDE.md with sections for each infrastructure type

B) Three .claude/rules/ files, each with a paths: field matching the relevant directories

C) CLAUDE.md files in each subdirectory

D) Skills with different allowed-tools for each infrastructure type

Correct Answer: B

Path-specific rules in .claude/rules/ load only when editing matching files, providing clean separation of concerns. Root CLAUDE.md (A) loads everything always, wasting context. Directory CLAUDE.md (C) works but is harder to maintain. Skills (D) require manual invocation.

Q49 — Domain 4: Batch Processing

Your manager wants to run overnight security audits on 500 files using the Message Batches API. Each audit request includes the file content and a JSON schema for structured findings. Some requests fail because files exceed the model's context limit. How should you handle failures?

A) Retry all 500 files in a new batch

B) Use custom_id to identify failed files, split oversized files into chunks, and resubmit only those

C) Switch to real-time API for the failed files

D) Exclude oversized files from future audits

Correct Answer: B

custom_id correlates batch requests with responses, enabling targeted resubmission of only failed items. Splitting oversized files into chunks addresses the specific failure cause. Retrying all 500 wastes resources. Excluding files creates coverage gaps.

Q50 — Domain 5: Human Review

Your extraction pipeline reports 95% field-level accuracy. You implement a rule: auto-approve any extraction with model confidence > 90%. After deployment, you discover that 15% of auto-approved extractions have errors. What went wrong?

A) The confidence threshold is too low; increase to 95%

B) Model confidence scores are poorly calibrated; high confidence doesn't guarantee accuracy. Implement stratified sampling to validate

C) The pipeline has a bug that miscalculates confidence

D) The 95% accuracy metric was measured on a different dataset

Correct Answer: B

LLM confidence scores are often poorly calibrated—the model can be confidently wrong. Stratified random sampling of high-confidence extractions is needed to measure actual error rates and calibrate thresholds using labeled validation sets.

Q51 — Domain 1: Session Management

A developer resumes a Claude Code session from yesterday to continue investigating a bug. The relevant source files have been refactored since then. The agent references class names that no longer exist. What's the best approach?

A) Use /compact to refresh the context

B) Start a fresh session with a structured summary of prior findings plus notes about which files changed

C) Delete the session and start over without any context

D) Continue the session but tell the agent to re-read all files

Correct Answer: B

When significant changes have occurred since the last session, prior tool results are stale. Starting fresh with a structured summary provides a clean foundation without outdated details. Simply telling the agent to re-read (D) may not overcome stale reasoning in context.

Q52 — Domain 2: MCP Resources

Your agent makes 5–7 exploratory tool calls at the start of every conversation to discover what data is available (listing database tables, API endpoints, document categories). Each call adds latency and consumes tokens. What MCP feature would reduce this overhead?

A) MCP prompts that pre-load common queries

B) MCP resources that expose content catalogs (schema listings, available categories) without requiring tool calls

C) Caching previous tool call results in the system prompt

D) Pre-executing all tools at session start

Correct Answer: B

MCP resources expose content catalogs that give agents visibility into available data without exploratory tool calls. This eliminates the discovery phase, reducing both latency and token usage.

Q53 — Domain 3: /memory and Diagnostics

Two developers report that Claude Code behaves differently for the same task in the same repository. Developer A gets correct test patterns; Developer B doesn't. Both pull the latest code. What's the most likely cause and how to diagnose?

A) The model is non-deterministic; this is expected behavior

B) Developer A has additional instructions in their ~/.claude/CLAUDE.md; use /memory to verify loaded configuration

C) Developer B has a corrupted Claude Code installation

D) The repository's .claude/CLAUDE.md has a syntax error

Correct Answer: B

Different behavior for the same project usually means different configuration. Developer A likely has user-level CLAUDE.md with testing instructions that Developer B lacks. The /memory command reveals which configuration files are loaded, enabling quick diagnosis.

Q54 — Domain 4: Validation-Retry

Your invoice extractor fails validation because the calculated total ($487.50) doesn't match the stated total ($487.50) due to floating-point precision ($487.4999999). What's the correct fix?

A) Retry the extraction with a prompt emphasizing precision

B) Use a tolerance-based comparison (e.g., difference < $0.01) in the validation logic

C) Round all values before comparison

D) Switch to integer arithmetic (store amounts in cents)

Correct Answer: B

This is a validation logic issue, not an extraction issue. The extraction is correct but the comparison is too strict. Tolerance-based validation accounts for floating-point precision while still catching genuine mismatches. Retrying (A) won't fix a validation logic problem.

Q55 — Domain 5: Lost in the Middle

Your multi-agent system aggregates findings from 8 subagents into a single synthesis prompt. The synthesis consistently misses findings from subagents 4 and 5 while correctly incorporating findings from subagents 1–3 and 6–8. What's happening?

A) Subagents 4 and 5 are returning lower-quality findings

B) The "lost in the middle" attention pattern: findings from subagents 4–5 are in the middle of the aggregated input

C) The synthesis subagent's prompt doesn't mention subagents 4 and 5

D) Token limits are truncating the middle of the input

Correct Answer: B

The "lost in the middle" effect causes models to miss content in the middle of long inputs while reliably processing the beginning and end. Mitigation: place key findings summaries at the beginning, use explicit section headers, and ensure critical content isn't buried in the middle.

Q56 — Cross-Domain: Architecture Decision

You're designing a system that extracts data from contracts, validates against business rules, and flags issues for legal review. Which architecture pattern best fits this workflow?

A) Single agent with access to extract, validate, and flag tools

B) Multi-agent: extraction subagent → validation subagent → review routing subagent, coordinated by a coordinator

C) Prompt chaining: extract → validate → route, with each step's output feeding the next

D) Batch processing: run all contracts through extraction, then all through validation, then all through routing

Correct Answer: C

This is a predictable, fixed-sequence workflow (extract → validate → route). Prompt chaining is the ideal pattern because the steps are known in advance and don't vary. A multi-agent system (B) adds unnecessary coordination overhead. A single agent (A) risks attention dilution. Batch processing (D) prevents per-document error handling between steps.

Q57 — Cross-Domain: Tool + Prompt

Your extraction tool's schema includes a field "payment_terms" with enum values ["net_30", "net_60", "net_90", "upon_receipt", "other"]. Documents frequently contain terms like "payment due within 45 days" which the model maps to "net_30". How do you improve accuracy?

A) Add "net_45" to the enum

B) Add an "other" option already exists; add a detail field and few-shot examples showing when to use "other" vs. the closest match

C) Change from enum to free text

D) Add a validation rule that rejects "net_30" when the document says "45 days"

Correct Answer: B

The "other" enum + detail field pattern captures non-standard values accurately. Few-shot examples showing "45 days → other, detail: net_45" vs. "30 days → net_30" teach the model to use "other" appropriately rather than forcing the closest match. Free text (C) loses categorization benefits.

Q58 — Cross-Domain: CI + Reliability

Your CI code review comments duplicate feedback that was already given in a previous review run. Developers are frustrated by repeated comments on issues they've already acknowledged. What's the fix?

A) Only review new files, not previously reviewed ones

B) Include prior review findings in context and instruct Claude to report only new or still-unaddressed issues

C) Limit reviews to one per PR

D) Have developers mark comments as “resolved” and filter those

Correct Answer: B

Including prior findings in context gives Claude awareness of what’s already been flagged. Instructions to report only new or unaddressed issues prevent duplicates. Option A misses regressions in previously reviewed files. Option D requires additional infrastructure.

Q59 — Domain 1: Parallel Execution

Your coordinator needs results from both a web search subagent and a database query subagent before it can synthesize a response. Currently it runs them sequentially (6 seconds total: 3s each). How do you reduce latency?

A) Use a faster model for each subagent

B) Cache previous results to avoid redundant queries

C) Emit both Task tool calls in a single coordinator response to trigger parallel execution

D) Merge both subagents into one that handles both tasks

Correct Answer: C

When the coordinator emits multiple Task tool calls in one response, the subagents execute in parallel. This reduces latency from 6s (sequential) to approximately 3s (maximum of the two). This works because the subagents access independent data sources and don’t depend on each other’s output.

Q60 — Domain 2: Community vs. Custom MCP

Your team needs to integrate Claude Code with GitHub for issue tracking and PR management. A developer proposes building a custom MCP server. What’s the recommended approach?

A) Build a custom server for maximum control

B) Use the existing community MCP server for GitHub; build custom only for team-specific workflows not served by community options

C) Use a webhook-based integration instead of MCP

D) Access GitHub directly through Bash commands instead of MCP

Correct Answer: B

Community MCP servers for standard integrations (GitHub, Jira, Slack) are well-maintained and tested. Custom servers should be reserved for team-specific workflows that aren't served by available community options. Building custom adds unnecessary development and maintenance burden.

Q61 — Domain 3: Plan Mode

A developer is asked to "add a date validation to the signup form's birthday field." They activate plan mode, which takes 3 minutes to explore the codebase before making a simple one-line change. What went wrong?

A) Plan mode is always the safest approach

B) This is a simple, well-scoped change that should use direct execution instead of plan mode

C) The developer should have used a skill instead

D) Plan mode needs better configuration to be faster

Correct Answer: B

Plan mode is designed for complex, multi-file tasks with multiple valid approaches. A single-field validation is a well-scoped change with an obvious approach. Direct execution is more efficient for simple, clear-scope tasks.

Q62 — Domain 4: Self-Correction

Your extraction pipeline extracts both a calculated_total (sum of line items) and a stated_total (from the document's total line). They frequently differ by small amounts ($0.01–$0.50). What does this enable?

A) Nothing useful; the differences are just rounding errors

B) Automatic self-correction: when they differ, the system flags it for review and includes both values with a conflict_detected boolean

C) Proof that the extraction is working correctly

D) A way to calculate tax rates

Correct Answer: B

Extracting both calculated and stated totals enables self-correction validation. Small differences may be rounding; large differences indicate extraction errors. The conflict_detected boolean lets downstream systems handle discrepancies appropriately. This is a key schema design pattern for reliability.

Q63 — Domain 5: Scratchpad Files

During a long codebase exploration session, Claude starts referring to “typical patterns” instead of the specific classes it discovered 20 messages ago. You implement scratchpad files where the agent records key findings. How does this help?

A) It frees up context space by removing the exploration output

B) It persists specific findings outside the conversation, so the agent can reference the scratchpad instead of relying on degraded context memory

C) It creates a backup of the conversation

D) It allows other agents to access the findings

Correct Answer: B

Scratchpad files persist key findings outside the conversation context. When the agent needs specific details, it reads the scratchpad file rather than relying on context memory, which degrades over long sessions. This is a practical mitigation for context degradation.

Q64 — Cross-Domain: Full Scenario

Your customer support agent incorrectly processes a $750 refund (exceeding the $500 automated limit) for a customer whose identity was verified. The prompt says “do not process refunds over $500” but the agent did it anyway. What combination of fixes is most effective?

A) Stronger prompt language + more few-shot examples of the limit

B) A PreToolUse hook that blocks process_refund when amount > $500 + structured error response redirecting to escalation

C) A PostToolUse hook that reverses refunds over $500 after processing

D) Remove the agent’s access to process_refund entirely

Correct Answer: B

A PreToolUse hook provides a deterministic guarantee that the tool call is blocked before execution. The structured error response guides the agent to escalate instead. Option A is probabilistic. Option C is dangerous (the refund already processed). Option D removes useful functionality.

Q65 — Domain 1: Dynamic Routing

A user asks your multi-agent research system: “What year was the Eiffel Tower built?” The system invokes all 4 subagents (search, analyze, synthesize, report), taking 30 seconds. The answer is a simple fact. What should change?

A) Add a cache for common factual questions

B) Train the coordinator to dynamically select which subagents to invoke based on query complexity

C) Create a separate FAQ bot for simple questions

D) Reduce the number of subagents to 2

Correct Answer: B

Dynamic routing allows the coordinator to match solution complexity to query complexity. A simple factual question needs only the search subagent, not the full pipeline. This is the proportionate response principle.

Q66 — Domain 2: Scoped Cross-Role Tools

Your synthesis subagent frequently needs to verify simple facts during report generation, requiring round-trips through the coordinator to the search subagent. This adds significant latency. What’s the recommended optimization?

A) Give the synthesis subagent full access to the search agent’s tools

B) Add a scoped verify_fact tool to the synthesis subagent for simple lookups only

C) Pre-fetch all potentially needed facts before starting synthesis

D) Skip fact verification to reduce latency

Correct Answer: B

A scoped cross-role tool reduces latency for high-frequency simple operations without giving the subagent excessive tool access. It’s a targeted optimization: simple fact lookups happen locally while complex queries still route through the coordinator.

Q67 — Domain 3: @import

Your monorepo has shared coding standards documented in /standards/api-conventions.md and /standards/security-checklist.md. The API package’s CLAUDE.md needs both, but the frontend package only needs the security checklist. What’s the cleanest approach?

A) Duplicate the content in each package’s CLAUDE.md

B) Put everything in the root CLAUDE.md

C) Use @import in each package’s CLAUDE.md to reference only the relevant standards files

D) Create skills for each standard

Correct Answer: C

@import allows selective inclusion of external files, keeping configurations modular. Each package imports only what it needs. Duplication (A) creates maintenance burden. Root-level everything (B) loads irrelevant context. Skills (D) require manual invocation.

Q68 — Domain 4: tool_choice “any” vs forced

You have multiple extraction schemas: one for invoices, one for receipts, one for contracts. The document type is unknown at extraction time. Which tool_choice setting is appropriate?

A) tool_choice: "auto" — let the model decide whether to extract

B) tool_choice: "any" — force a tool call but let the model choose which extraction schema

C) Forced selection of the invoice schema as a default

D) Run all three schemas and pick the best result

Correct Answer: B

tool_choice: "any" guarantees a tool call (structured output) while allowing the model to select the appropriate schema based on the document content. "auto" might return text instead. Forced selection (C) uses the wrong schema for non-invoices. Running all three (D) triples cost.

Q69 — Domain 5: Temporal Data

Your research report says “unemployment is 3.5%” based on Source A and “unemployment is 7.2%” based on Source B. The reader sees a contradiction. Investigation reveals Source A is from 2023 and Source B from 2020. What’s the fix?

A) Discard the older source

B) Average the two figures

C) Require publication dates in all structured outputs so temporal differences aren’t misinterpreted as contradictions

D) Only use the most recent source for any statistic

Correct Answer: C

Without dates, temporal differences appear as contradictions. Including publication dates in structured outputs (e.g., “3.5% (2023), 7.2% (2020)”) provides context that reveals these are different time periods, not conflicting claims. Simply discarding old data (A, D) may lose valuable trend information.

Q70 — Cross-Domain: Error + Architecture

In your multi-agent system, the web search subagent fails with a timeout. The coordinator receives a generic “search failed” message. It abandons the research entirely and returns “I couldn’t find information on that topic.” What are the TWO problems?

A) The timeout indicates the search service is permanently down

B) The error message lacks structured context (errorCategory, isRetryable); the coordinator terminates the entire workflow on a single subagent failure

C) The coordinator should have used a different search engine

D) The search subagent should have retried internally

Correct Answer: B

Two problems: (1) Generic error messages prevent intelligent recovery, and (2) workflow termination on a single failure is disproportionate. The error should include isRetryable: true for timeouts. The coordinator should retry with a modified query, try alternative approaches, or proceed with partial results from other subagents.

Q71–75: Rapid-Fire Scenario Questions

Q71: An agent's system prompt says "always verify identity first." In testing, identity is skipped 8% of the time. Fix?

Answer: Programmatic prerequisite gate (not stronger prompt). Critical business rule needs deterministic enforcement.

Q72: Your extraction schema has an enum field ["approved", "rejected", "pending"] but documents sometimes say "conditionally approved." The model forces "approved." Fix?

Answer: Add "other" to the enum plus a detail string field. Optionally add "unclear" for genuinely ambiguous cases.

Q73: A customer lookup returns 3 matches for "John Smith." The agent picks the one with the most recent order. Correct?

Answer: No. Ask for additional identifiers (email, phone, order number). Heuristic selection risks acting on the wrong account.

Q74: Your batch API job for 200 contracts fails for 15 due to oversized content. Resubmit strategy?

Answer: Use custom_id to identify the 15 failed contracts. Split oversized documents into sections. Resubmit only the 15 with modifications.

Q75: You create a /deploy command file in ~/.claude/commands/deploy.md. Your teammate can't find it. Why?

Answer: User-scoped commands (~/.claude/commands/) are personal and not version-controlled. Move to .claude/commands/ for team-wide access.

Q76–80: Advanced Integration Questions

Q76: Design a review system where generated code is reviewed by an independent instance. What's the key architectural principle?

Answer: Session context isolation. The reviewer must NOT have the generator's reasoning context. Use a separate Claude instance that receives only the code and review criteria, not the generation prompt or reasoning.

Q77: Your multi-agent system needs crash recovery. After a crash mid-pipeline, you restart and the coordinator doesn't know what was completed. Design the recovery mechanism.

Answer: Structured agent state exports (manifests). Each subagent exports its completion state to a known location. On restart, the coordinator loads the manifest to determine what's done and what needs re-execution. Include input hashes to detect stale results.

Q78: An extraction tool uses tool_use with a JSON schema. The output is always schema-compliant (no syntax errors) but sometimes contains fabricated values. How is this possible?

Answer: JSON schemas enforce structure (format compliance) but NOT semantic correctness. The model can produce syntactically valid JSON that contains hallucinated or incorrect values. Additional semantic validation (cross-field checks, range validation) is needed beyond schema compliance.

Q79: Your coordinator should use plan mode for complex research but direct execution for simple factual lookups. How do you implement this without hardcoding query classification?

Answer: Design the coordinator's prompt to analyze query complexity and dynamically select the appropriate approach. Include few-shot examples showing: simple query → single search subagent, medium query → search + synthesis, complex query → full pipeline with iterative refinement. The coordinator makes the judgment call.

Q80: Your system handles both English and Japanese invoices. English extractions are 98% accurate; Japanese extractions are 72% accurate. Both report high confidence. How do you improve the system?

Answer: (1) Stratified accuracy analysis confirms the language-specific gap. (2) Add language-specific few-shot examples for Japanese documents. (3) Calibrate confidence thresholds separately per language. (4) Route low-confidence Japanese extractions to bilingual human reviewers. (5) Include language as a field in the extraction schema to enable per-language quality tracking.

Part X: Appendices

Appendix A: Complete API Reference for Exam Topics

A.1 Messages API — stop_reason Values

Value	Meaning	Action
"end_turn"	Model finished responding	Present response to user; exit loop
"tool_use"	Model wants to call tool(s)	Execute tools, append results, continue loop
"max_tokens"	Response hit token limit	Handle truncation; consider increasing max_tokens
"stop_sequence"	Custom stop sequence matched	Process based on application logic

A.2 tool_choice Options

Setting	Behavior	When to Use
"auto" (default)	Model may call a tool OR return text	General-purpose agents; tool use is optional
"any"	Model MUST call a tool; can choose which	Guaranteed structured output with multiple schemas
{"type": "tool", "name": "X"}	Model MUST call tool X specifically	Enforcing specific extraction steps; first-turn tool use

A.3 Message Batches API

Feature	Detail
Cost savings	50% compared to real-time API
Processing time	Up to 24 hours; no latency SLA
Correlation	custom_id field links request to response
Multi-turn	NOT supported (single-turn only)
Tool use	Supported but no mid-request tool execution
Failure handling	Resubmit failed items by custom_id
Best for	Overnight reports, weekly audits, batch extraction
Not suitable for	Blocking workflows, real-time interactions, pre-merge checks

A.4 MCP Error Response Pattern

```
// Structured error response for MCP tools
{
  "isError": true,
  "content": [{
    "type": "text",
    "text": JSON.stringify({
      "errorCategory": "transient | validation | business | permission",
      "isRetryable": true | false,
      "description": "Human-readable error description",
      "customerFriendlyMessage": "Message safe to show end users",
      "suggestedAction": "retry | fix_input | escalate | alternative_workflow"
    })
  }]
}
```

Appendix B: Configuration File Reference

B.1 .mcp.json (Project-Level MCP Configuration)

```
{
  "mcpServers": {
    "github": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-github"],
      "env": {
        "GITHUB_TOKEN": "${GITHUB_TOKEN}"
      }
    }
  }
}
```

B.2 CLAUDE.md Hierarchy Summary

Level	Location	Scope	Version Controlled
User	~/.claude/CLAUDE.md	Personal preferences	No
Project	.claude/CLAUDE.md or root CLAUDE.md	Team standards	Yes
Directory	subdirectory/CLAUDE.md	Area-specific context	Yes

B.3 SKILL.md Frontmatter Options

Option	Values	Purpose
context	fork | (default: inline)	Isolate skill output from main conversation
allowed-tools	List of tool names	Restrict tool access during skill execution
argument-hint	String	Prompt for required parameters on invocation

B.4 Claude Code CLI Flags for CI/CD

Flag	Purpose	Example
-p / --print	Non-interactive mode	claude -p "Review this PR"
--output-format json	JSON output	claude -p --output-format json "..."
--json-schema	Enforce output schema	claude -p --json-schema schema.json

		"..."
--resume <name>	Continue named session	claude --resume review-session

Appendix C: Glossary of Key Terms

Term	Definition
Agentic loop	The cycle of sending requests to Claude, executing tool calls, and continuing until end_turn
AgentDefinition	Configuration for a subagent including description, system prompt, and allowed tools
allowedTools	List of tools a specific agent can access; controls tool scope
context: fork	SKILL.md option that runs a skill in an isolated sub-agent context
custom_id	Identifier in batch API for correlating requests with responses
end_turn	stop_reason value indicating Claude has finished its response
fork_session	Creates an independent branch from a shared analysis baseline
hub-and-spoke	Multi-agent pattern where coordinator manages all inter-agent communication
isError	MCP flag signaling a tool call failure
isRetryable	Metadata indicating whether a failed operation can be retried
Lost in the middle	Attention pattern where models miss information in middle sections of long inputs
MCP	Model Context Protocol — standard for tool and resource integration
PostToolUse	Hook that intercepts tool results for transformation before model processing
Prompt chaining	Breaking complex tasks into fixed sequential steps
Scratchpad file	Persistent file where agents record key findings to counteract context degradation
stop_reason	API field indicating why Claude stopped generating (tool_use, end_turn, etc.)
Stratified sampling	Random sampling within categories to ensure segment-level accuracy measurement
Task tool	Mechanism for spawning subagents in the Claude Agent SDK
tool_choice	API parameter controlling how Claude selects tools (auto, any, forced)
tool_use	stop_reason value indicating Claude wants to call one or more tools

Appendix D: Decision Trees for Common Exam Patterns

D.1 Agent Skips Required Steps

Symptom: Agent occasionally bypasses a required step (e.g., identity verification).

- Is compliance critical (financial, legal, safety)? → Programmatic enforcement (hooks, gates)
- Is it a soft preference? → Strengthen prompt + add few-shot examples
- Is it a tool selection issue? → Improve tool descriptions

D.2 Multi-Agent Output Has Gaps

Symptom: Final report is missing expected coverage areas.

- Check coordinator's task decomposition first → Is the decomposition too narrow?
- If decomposition is broad, check subagent execution → Did subagents find relevant content?
- If subagents have content, check synthesis → Is the synthesis dropping findings?
- If sources are conflicting, check provenance handling → Are conflicts annotated?

D.3 Tool Selection is Unreliable

Symptom: Agent frequently calls the wrong tool.

- Are tool descriptions minimal? → Expand descriptions with examples and boundaries
- Do multiple tools have overlapping descriptions? → Rename and differentiate
- Does the agent have too many tools? → Reduce to 4-5 scoped tools per role
- Does the system prompt override descriptions? → Remove keyword-sensitive instructions

D.4 Choosing Plan Mode vs Direct Execution

Symptom: Need to decide the right approach for a task.

- Is the scope clear and limited (1-2 files)? → Direct execution
- Are there multiple valid approaches? → Plan mode
- Does it require architectural decisions? → Plan mode
- Is it a simple bug fix with a clear stack trace? → Direct execution
- Does it affect dozens of files? → Plan mode

D.5 Error in Multi-Agent System

Symptom: A subagent encounters an error.

- Is it transient (timeout, service down)? → Retry locally with backoff

- Is it a validation error (bad input)? → Fix input, retry locally
- Is it a business/permission error? → Cannot retry; propagate structured context to coordinator
- Can the task continue with partial results? → Proceed with coverage annotations
- Is recovery impossible? → Escalate with full context

Appendix E: Exam Day Strategies

E.1 Time Management

With multiple-choice questions across 4 scenarios, pace yourself. Don't spend too long on any single question. If unsure, mark your best guess and move on—there's no penalty for guessing.

E.2 Reading Questions Carefully

Pay attention to qualifiers like "most effective," "first step," and "most likely root cause." These narrow the correct answer:

- **"Most effective"** = the solution that best addresses the root cause
- **"First step"** = prefer simple, high-leverage changes over complex infrastructure
- **"Most likely root cause"** = trace the problem to the correct architectural level

E.3 Eliminating Distractors

Common distractor patterns to recognize:

- **Over-engineered solutions**: ML classifiers, routing layers, when prompt improvements haven't been tried
- **Wrong level solutions**: Fixing subagents when the coordinator is the problem
- **Probabilistic solutions for deterministic needs**: Prompts/few-shot for critical business rules
- **Non-existent features**: Environment variables, CLI flags, or config files that don't actually exist

E.4 Scenario Context

Each question is framed within a scenario. Before answering, make sure you understand which scenario you're in and what the primary domains are. The scenario context helps you apply the right mental models.

E.5 Final Review

If time permits, review flagged questions. Often, reading the question a second time with fresh eyes reveals the correct answer. Trust the decision frameworks you've practiced:

- Critical business logic → Programmatic enforcement
- Root cause first → Trace to the correct level
- Simplest effective solution → Don't over-engineer
- Structured information → Enable intelligent downstream decisions

Appendix F: Anti-Pattern Catalog

This appendix catalogs every anti-pattern discussed in the handbook with a summary of why it fails and the correct alternative.

Anti-Pattern	Why It Fails	Correct Approach
Parsing text for loop termination	Fragile; bypasses structured API	Use stop_reason field
Arbitrary iteration caps as primary stop	Interrupts valid reasoning; may terminate too early	Use stop_reason; caps as safety nets only
Prompt-only enforcement for critical rules	Non-zero failure rate; financial/safety risk	Programmatic hooks or prerequisite gates
Giving agent 15+ tools	Decision complexity degrades selection reliability	4-5 scoped tools per role
Generic error messages ("Operation failed")	Agent cannot make informed recovery decisions	Structured error metadata with categories
Silent error suppression (empty as success)	Hides failures; leads to incomplete results	Explicit isError flag with context
Terminating workflow on single failure	Disproportionate; loses partial progress	Local recovery; propagate only unresolvable errors
Self-review (same session)	Model retains reasoning context; blind to own errors	Independent review instance
"Be conservative" as precision fix	Vague; model has no concrete definition	Specific categorical criteria
Confidence-based escalation	Self-reported confidence is poorly calibrated	Explicit escalation criteria with few-shot examples
Sentiment-based escalation	Frustration doesn’t correlate with complexity	Criteria based on case type and policy coverage
Reading all files upfront	Overwhelms context with irrelevant content	Incremental exploration from entry points
Required fields for optional data	Model fabricates values to satisfy schema	Nullable/optional fields
Batch API for blocking workflows	Up to 24-hour latency; no SLA	Real-time API for blocking; batch for overnight
Shared memory between subagents	Doesn’t exist in Agent SDK	Explicit context in subagent prompts
Directory CLAUDE.md for scattered files	Can’t reach files across many directories	Path-specific rules with glob patterns
User-level config for team standards	Not shared via version control	Project-level config in repository

Appendix G: Comparison Tables

G.1 Programmatic vs. Prompt-Based Enforcement

Aspect	Programmatic (Hooks/Gates)	Prompt-Based (Instructions/Few-Shot)
Guarantee level	Deterministic (100%)	Probabilistic (80-99%)
Implementation effort	Higher (code changes)	Lower (prompt edits)
Flexibility	Rigid; follows exact rules	Flexible; can adapt to novel situations
Best for	Financial, legal, safety-critical rules	Quality guidelines, behavioral preferences
Failure mode	Blocks unauthorized actions	Occasionally allows unauthorized actions
Maintenance	Code updates for rule changes	Prompt updates for rule changes
Debugging	Clear audit trail	Probabilistic; hard to predict failures

G.2 Plan Mode vs. Direct Execution

Aspect	Plan Mode	Direct Execution
Task complexity	High (multi-file, architectural)	Low (single-file, clear scope)
Approach clarity	Multiple valid approaches	Obvious approach
Risk of rework	Reduced (explore before commit)	Acceptable (scope is small)
Context usage	Higher (exploration output)	Lower (focused changes)
Speed	Slower (planning phase)	Faster (immediate action)
When files affected	Dozens or more	1-3 files
When dependencies	Complex, unknown	Clear, minimal

G.3 Real-Time API vs. Message Batches API

Aspect	Real-Time API	Message Batches API
Cost	Standard pricing	50% discount
Latency	Seconds to minutes	Up to 24 hours
Latency SLA	Yes	No
Multi-turn tool use	Supported	Not supported
Blocking workflows	Appropriate	Not appropriate

Use case examples	Pre-merge checks, real-time chat	Overnight reports, weekly audits
Correlation	Request-response direct	Via custom_id field
Failure handling	Immediate retry	Resubmit failed items by custom_id

G.4 Project-Level vs. User-Level Configuration

Aspect	.claude/ (Project)	~/.claude/ (User)
Version controlled	Yes	No
Shared with team	Yes (via git)	No
CLAUDE.md	.claude/CLAUDE.md or root CLAUDE.md	~/.claude/CLAUDE.md
Commands	.claude/commands/	~/.claude/commands/
Skills	.claude/skills/	~/.claude/skills/
MCP servers	.mcp.json	~/.claude.json
Best for	Team standards, shared workflows	Personal preferences, experiments

G.5 Slash Commands vs. Skills vs. CLAUDE.md Rules

Aspect	Slash Commands	Skills	CLAUDE.md / Rules
Invocation	Manual (/command)	Manual or referenced	Automatic (always loaded)
Isolation	No (main context)	Optional (context: fork)	No (main context)
Tool restrictions	No	Yes (allowed-tools)	No
Path scoping	No	No	Yes (glob patterns in rules)
Arguments	Positional	argument-hint prompt	N/A
Best for	Quick team workflows	Complex isolated tasks	Universal standards

Appendix H: Detailed Code Examples

H.1 Complete Agentic Loop Implementation

This example shows a production-grade agentic loop with proper stop_reason handling, tool result management, and safety limits:

```
async function agenticLoop(systemPrompt, userMessage, tools) {
  const messages = [{ role: "user", content: userMessage }];
  const MAX_ITERATIONS = 50;  // Safety limit, not primary control

  for (let i = 0; i < MAX_ITERATIONS; i++) {
    const response = await anthropic.messages.create({
      model: "claude-sonnet-4-20250514",
      max_tokens: 4096,
      system: systemPrompt,
      messages,
      tools,
    });

    // Primary control: use stop_reason
    if (response.stop_reason === "end_turn") {
      return response.content;  // Agent is done
    }

    if (response.stop_reason === "tool_use") {
      // Append assistant response to history
      messages.push({ role: "assistant", content: response.content });

      // Execute all tool calls and collect results
      const toolResults = [];
      for (const block of response.content) {
        if (block.type === "tool_use") {
          const result = await executeToolWithErrorHandling(block);
          toolResults.push({
            type: "tool_result",
            tool_use_id: block.id,
            content: JSON.stringify(result)
          });
        }
      }

      // Append all tool results in a single user message
      messages.push({ role: "user", content: toolResults });
```

```
    }
  }
  throw new Error("Agent exceeded maximum iterations");
}
```

H.2 PostToolUse Hook for Data Normalization

```
// Hook that normalizes dates from various formats to ISO 8601
const dateNormalizationHook = {
  type: "PostToolUse",
  handler: async (toolResult) => {
    const data = JSON.parse(toolResult.content);
    // Normalize Unix timestamps
    if (data.created_at && typeof data.created_at === "number") {
      data.created_at = new Date(data.created_at * 1000).toISOString();
    }
    // Normalize "MM/DD/YYYY" format
    if (data.order_date && /\d{2}\/\d{2}\/\d{4}/.test(data.order_date)) {
      const [m, d, y] = data.order_date.split("/");
      data.order_date = `${y}-${m}-${d}T00:00:00Z`;
    }
    return { ...toolResult, content: JSON.stringify(data) };
  }
};
```

H.3 Tool Call Interception for Compliance

```
// Hook that blocks refunds above $500 and redirects to escalation
const refundLimitHook = {
  type: "PreToolUse",
  toolName: "process_refund",
  handler: async (toolCall) => {
    const amount = toolCall.input.refund_amount;
    if (amount > 500) {
      return {
        blocked: true,
        alternativeAction: "escalate_to_human",
        reason: `Refund amount ($${amount}) exceeds $500 automated limit`,
        toolResult: {
          isError: true,
          errorCategory: "business",
          isRetryable: false,
          description: "Refund requires manager approval"
```

```
        }
      };
    }
    return { blocked: false };  // Allow the call to proceed
  }
};
```

H.4 Structured Output with tool_use

```
// Define extraction tool with JSON schema
const extractionTool = {
  name: "extract_invoice_data",
  description: "Extract structured data from an invoice document",
  input_schema: {
    type: "object",
    properties: {
      vendor_name: { type: "string" },
      invoice_number: { type: "string" },
      invoice_date: { type: ["string", "null"] },  // nullable
      total_amount: { type: "number" },
      currency: { type: "string", enum: ["USD", "EUR", "GBP", "other"] },
      currency_detail: { type: ["string", "null"] },
      purchase_order: { type: ["string", "null"] },  // nullable
      confidence: { type: "string", enum: ["high", "medium", "low", "unclear"]
}
    },
    required: ["vendor_name", "total_amount", "currency", "confidence"]
  }
};

// Force the model to use this specific tool
const response = await anthropic.messages.create({
  model: "claude-sonnet-4-20250514",
  tools: [extractionTool],
  tool_choice: { type: "tool", name: "extract_invoice_data" },
  messages: [{ role: "user", content: documentContent }]
});
```

H.5 Path-Specific Rules Example

Example .claude/rules/ files for a monorepo with different conventions per area:

File: .claude/rules/testing.md

```
---
paths:
  - "**/*.test.ts"
  - "**/*.test.tsx"
  - "**/*.spec.ts"
---
# Testing Conventions
- Use describe/it blocks with descriptive names
- Test edge cases: null inputs, empty arrays, boundary values
- Use factory functions from test/factories/ for test data
- Mock external services; never make real API calls in tests
- Assert specific values, not just truthiness
```

File: .claude/rules/api-handlers.md

```
---
paths:
  - "src/api/**/*"
  - "src/routes/**/*"
---
# API Handler Conventions
- Use async/await (never raw promises)
- Validate all input with zod schemas
- Return consistent error format: { error: string, code: number }
- Log errors with request ID for traceability
- Use middleware for auth, rate limiting, CORS
```

File: .claude/rules/terraform.md

```
---
paths:
  - "terraform/**/*"
  - "infra/**/*.tf"
---
# Terraform Conventions
- Use modules for reusable infrastructure
- Tag all resources with environment and team
- Use remote state with S3 backend
- Variables must have descriptions and type constraints
```

Appendix I: Quick-Review Flashcard Summary

Use these key facts for last-minute review before the exam:

- **stop_reason "tool_use"** = continue the loop; **"end_turn"** = stop the loop
- **Subagents have isolated context** — no automatic inheritance from coordinator
- **Task tool** spawns subagents; coordinator's allowedTools must include "Task"
- **Parallel subagents** = multiple Task calls in one coordinator response
- **Programmatic enforcement** for critical business rules (deterministic)
- **Prompt + few-shot** for quality guidelines (probabilistic)
- **PostToolUse** hooks normalize data; **PreToolUse** hooks enforce compliance
- **Tool descriptions** are the primary mechanism for tool selection
- **4-5 tools** per agent role; more degrades selection reliability
- **MCP resources** reduce exploratory tool calls by exposing content catalogs
- **.mcp.json** = project-level (shared); **~/.claude.json** = user-level (personal)
- **${ENV_VAR}** expansion in .mcp.json for credential management
- **CLAUDE.md hierarchy**: user > project > directory
- **.claude/rules/** with YAML paths field for glob-scoped rules
- **.claude/commands/** = project slash commands; **~/.claude/commands/** = personal
- **context: fork** in SKILL.md isolates verbose skill output
- **Plan mode** for complex, multi-file, architectural tasks
- **Direct execution** for simple, well-scoped, single-file changes
- **-p flag** for non-interactive CI/CD mode
- **--output-format json** + **--json-schema** for structured CI output
- **tool_use** with JSON schemas = guaranteed schema compliance (no syntax errors)
- **tool_choice: "any"** = must call a tool; **"auto"** = may return text instead
- **Nullable fields** prevent model from fabricating values for absent data
- **Retry-with-error-feedback** = include failed extraction + specific errors in retry prompt
- **Message Batches API** = 50% savings, up to 24-hour processing, no latency SLA
- **custom_id** correlates batch request/response pairs
- **Lost in the middle** — place key info at beginning and end of long inputs
- **Trim verbose tool outputs** to relevant fields before context accumulation
- **Case facts block** persists critical details across summarization

- **Escalate on**: customer request, policy gaps, inability to progress
- **Don’t escalate on**: sentiment alone, self-reported confidence, complexity alone
- **Scratchpad files** persist findings across context boundaries
- **/compact** reclaims context during long exploration sessions
- **Stratified sampling** detects segment-specific accuracy problems
- **Field-level confidence** is more useful than document-level confidence
- **Claim-source mappings** preserve attribution through synthesis
- **Conflicting sources** — annotate with attribution; don’t arbitrarily pick one
- **Publication dates** in outputs prevent temporal misinterpretation

Appendix J: Recommended Study Schedule

Week 1: Foundation (Domains 1 & 2)

Day 1–2: Read Chapters 4–6 (Agentic loops, multi-agent orchestration, context passing). Practice implementing a basic agentic loop with tool calling.

Day 3–4: Read Chapters 7–10 (Enforcement patterns, hooks, task decomposition, session management). Build a multi-tool agent with escalation logic (Exercise 1 from exam guide).

Day 5–6: Read Chapters 12–16 (Tool design, error handling, MCP integration, built-in tools). Design MCP tools with structured error responses (Exercise 3 from exam guide).

Day 7: Complete Domain 1 and Domain 2 practice questions. Review any weak areas.

Week 2: Configuration & Prompting (Domains 3 & 4)

Day 8–9: Read Chapters 18–23 (CLAUDE.md hierarchy, commands, skills, rules, plan mode, CI/CD). Configure Claude Code for a real project (Exercise 2).

Day 10–11: Read Chapters 25–30 (Precise prompts, few-shot examples, structured output, validation, batch processing, multi-pass review). Build an extraction pipeline (Exercise 3).

Day 12–13: Complete Domain 3 and Domain 4 practice questions. Focus on distinguishing when to use each configuration mechanism.

Day 14: Review and consolidate. Revisit any concepts that felt unclear.

Week 3: Reliability & Integration (Domain 5 + Review)

Day 15–16: Read Chapters 32–38 (Context management, escalation, error propagation, codebase exploration, human review, provenance). Build a multi-agent research pipeline (Exercise 4).

Day 17–18: Complete all remaining practice questions (Chapters 11, 17, 24, 31, 38, and the extended set in Part IX). Analyze any patterns in questions you get wrong.

Day 19–20: Read the scenario walkthroughs (Part VIII). Take the official practice exam. Review explanations for every question, including ones you got right.

Day 21: Light review of decision trees (Appendix D), glossary (Appendix C), and exam day strategies (Appendix E). Rest and prepare mentally.

STUDY TIP

The most effective study technique is building actual systems. Each exercise in the exam guide maps to specific domains. Complete at least Exercises 1, 2, and 3 for hands-on experience with the most heavily tested topics.

— End of Handbook —

Bonus: Detailed Worked Examples

This bonus section provides end-to-end worked examples that tie together concepts from multiple domains. These are the types of integrated problems that appear on the exam.

Worked Example 1: Building a Complete Customer Support System

Requirements

Build a customer support agent that handles returns, billing disputes, and account modifications. The system must verify customer identity before any financial operation, handle multi-concern requests, compile structured handoff summaries, and achieve 80%+ first-contact resolution.

Step 1: Define MCP Tools with Clear Descriptions

Following Domain 2 best practices, we define four tools with detailed descriptions that eliminate ambiguity:

```
const tools = [
  {
    name: "get_customer",
    description: "Looks up a customer record by email, phone, or customer ID.
" +
      "Returns: name, email, phone, account status, verification level. " +
      "Use FIRST before any order lookups or financial operations. " +
      "Do NOT use for order-specific queries (use lookup_order instead).",
    input_schema: { /* ... */ }
  },
  {
    name: "lookup_order",
    description: "Retrieves order details by order ID or customer ID. " +
      "Returns: items, amounts, shipping status, return eligibility. " +
      "Requires verified customer_id from get_customer. " +
      "Do NOT use for customer account info (use get_customer instead).",
    input_schema: { /* ... */ }
  },
  {
    name: "process_refund",
    description: "Processes a refund for a verified order. " +
      "Requires verified customer_id AND order_id. " +
      "Maximum automated refund: $500. " +
      "For amounts over $500, use escalate_to_human instead.",
    input_schema: { /* ... */ }
  },
```

```
  {
    name: "escalate_to_human",
    description: "Transfers the case to a human agent with a structured
summary. " +
      "Use when: customer requests human, policy gap, refund > $500, " +
      "unable to resolve after reasonable attempts.",
    input_schema: { /* ... */ }
  }
];
```

Step 2: Implement Programmatic Prerequisites

Following Domain 1 Task Statement 1.4, we implement a prerequisite gate that blocks financial operations until identity is verified:

```
// State tracking for prerequisite enforcement
let verifiedCustomerId = null;

const prerequisiteHook = {
  type: "PreToolUse",
  handler: async (toolCall) => {
    // Block lookup_order and process_refund without verification
    if (["lookup_order", "process_refund"].includes(toolCall.name)) {
      if (!verifiedCustomerId) {
        return {
          blocked: true,
          toolResult: {
            isError: true,
            errorCategory: "validation",
            description: "Customer identity must be verified first. " +
              "Call get_customer to verify the customer."
          }
        };
      }
    }
    return { blocked: false };
  }
};
```

Step 3: Add Refund Limit Enforcement

Following Domain 1 Task Statement 1.5, we add a hook that enforces the $500 refund limit:

```
const refundLimitHook = {
```

```
  type: "PreToolUse",
  toolName: "process_refund",
  handler: async (toolCall) => {
    if (toolCall.input.amount > 500) {
      return {
        blocked: true,
        redirectTo: "escalate_to_human",
        toolResult: {
          isError: true,
          errorCategory: "business",
          isRetryable: false,
          description: "Refund of $" + toolCall.input.amount +
            " exceeds $500 automated limit. Escalating to human agent."
        }
      };
    }
    return { blocked: false };
  }
};
```

Step 4: Design Escalation Criteria

Following Domain 5 Task Statement 5.2, we include explicit escalation criteria with few-shot examples in the system prompt:

```
const systemPrompt = `You are a customer support agent. Your goal is 80%+
first-contact resolution while knowing when to escalate.

ESCALATION CRITERIA:
- ALWAYS escalate: customer explicitly asks for a human agent
- ALWAYS escalate: policy doesn't cover the customer's specific request
- ALWAYS escalate: refund amount exceeds $500 (system will block
automatically)
- ALWAYS escalate: unable to make progress after 2 investigation attempts
- DO NOT escalate: standard returns with clear eligibility
- DO NOT escalate: billing disputes with clear transaction records

EXAMPLE 1 - Resolve (standard return):
Customer: "My order arrived damaged, I want a refund"
Action: Verify identity -> Look up order -> Process refund (if under $500)

EXAMPLE 2 - Escalate (policy gap):
Customer: "I found this cheaper on Amazon, can you match the price?"
Action: Our policy covers own-site adjustments but not competitor matching.
```

```
This is a policy gap. Escalate with structured summary.

EXAMPLE 3 - Acknowledge then resolve:
Customer: "This is ridiculous! The same item broke twice!"
Action: Acknowledge frustration, then verify and process the return.
Frustration alone is not an escalation trigger.`;
```

Step 5: Context Management

Following Domain 5 Task Statement 5.1, we implement a case facts block that persists critical details:

```
function buildCaseFactsBlock(facts) {
  return `\n=== CASE FACTS (always reference these) ===
Customer ID: ${facts.customerId || 'Not yet verified'}
Customer Name: ${facts.customerName || 'Unknown'}
Active Orders: ${facts.orders?.join(', ') || 'None looked up'}
Issues Identified: ${facts.issues?.join('; ') || 'None yet'}
Actions Taken: ${facts.actions?.join('; ') || 'None yet'}
Refund Amount: ${facts.refundAmount || 'N/A'}
===\n`;
}
```

This worked example demonstrates how concepts from Domains 1, 2, 4, and 5 come together in a production system. The exam tests your ability to make these integrated design decisions.

Worked Example 2: Configuring Claude Code for a Monorepo

Requirements

Configure Claude Code for a monorepo containing: a React frontend, a Node.js API backend, Terraform infrastructure code, and shared utilities. Test files are co-located with source files throughout. The team has 8 developers.

Step 1: Project-Level CLAUDE.md

Create .claude/CLAUDE.md with universal standards that apply to all code areas:

```
# Project Standards

## Universal Rules
- Use TypeScript for all new code
- All functions must have JSDoc comments
- No console.log in production code (use structured logger)
- All API endpoints require input validation
- Commit messages follow Conventional Commits format

## Testing
- All new features require tests
- Use factories from test/factories/ for test data
- Integration tests use the test database (see test/setup.ts)
```

Step 2: Path-Specific Rules

Create .claude/rules/ files for each code area with glob patterns:

Following Domain 3 Task Statement 3.3, these rules load only when editing matching files, reducing irrelevant context and token usage.

```
# .claude/rules/react-components.md
---
paths:
  - "packages/frontend/src/**/*.tsx"
  - "packages/frontend/src/**/*.ts"
---
# React Component Conventions
- Use functional components with hooks (no class components)
- Use React.memo for expensive renders
- Props interfaces must be exported for reuse
- Use CSS Modules for styling (no inline styles)
```

```
# .claude/rules/testing.md
---
paths:
  - "**/*.test.ts"
  - "**/*.test.tsx"
  - "**/*.spec.ts"
---
# Test File Conventions
- Use describe/it blocks with behavior-driven names
- Each test file must import from test/factories/ for test data
- Assert specific values, not truthy/falsy
- Mock external services with msw (Mock Service Worker)
```

```
# .claude/rules/terraform.md
---
paths:
  - "infra/**/*.tf"
  - "infra/**/*.tfvars"
---
# Terraform Conventions
- All resources must have environment and team tags
- Use modules for reusable infrastructure
- Variables must have descriptions and type constraints
- Use remote state with S3 backend and DynamoDB locking
```

Step 3: Team Slash Commands

Create shared commands in .claude/commands/ for common workflows:

```
# .claude/commands/review.md
Review the current file for bugs, security issues, and convention violations.
Focus on: null pointer risks, unhandled errors, missing input validation.
Skip: minor style issues, local naming preferences.
Output format: location, issue, severity (critical/warning/info), suggested
fix.
```

```
# .claude/commands/test.md
Generate tests for the current file. Include:
- Happy path for each public function
- Edge cases: null inputs, empty arrays, boundary values
- Error handling: invalid inputs, service failures
Use factories from test/factories/ for test data.
Check existing tests first to avoid duplicates.
```

Step 4: MCP Server Configuration

Configure shared MCP servers in .mcp.json:

```
{
  "mcpServers": {
    "github": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-github"],
      "env": { "GITHUB_TOKEN": "${GITHUB_TOKEN}" }
    },
    "jira": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-jira"],
      "env": {
        "JIRA_URL": "${JIRA_URL}",
        "JIRA_TOKEN": "${JIRA_TOKEN}"
      }
    }
  }
}
```

This configuration uses community MCP servers (following the recommendation to avoid custom servers for standard integrations) with environment variable expansion for secrets.

Step 5: Skill with Fork and Tool Restrictions

Create a project skill for codebase analysis that runs in isolation:

```
# .claude/skills/analyze-codebase/SKILL.md
---
context: fork
allowed-tools:
  - Read
  - Grep
  - Glob
argument-hint: "Directory or module to analyze"
---
# Codebase Analysis Skill
Analyze the specified directory or module. Report:
1. Module structure and key files
2. Public API surface (exported functions/classes)
3. Dependencies (internal and external)
4. Test coverage gaps
5. Potential architectural concerns
```

The **context: fork** option is essential here because codebase analysis produces verbose exploration output that would pollute the main conversation. The **allowed-tools** restriction limits the skill to read-only operations, preventing accidental modifications during analysis.

This complete configuration demonstrates how Domains 2 and 3 concepts work together to create a maintainable, team-friendly Claude Code setup.

Worked Example 3: Designing a Validation-Retry Pipeline

Requirements

Build a structured data extraction pipeline for invoices that: extracts key fields using JSON schemas, validates extracted data with semantic checks, implements retry-with-error-feedback for recoverable errors, and routes ambiguous cases to human review.

Step 1: Schema Design with Nullable Fields

Following Domain 4 Task Statement 4.3, design the extraction schema with nullable fields for data that might be absent:

```
const invoiceSchema = {
  type: "object",
  properties: {
    vendor_name: { type: "string", description: "Company or person that issued
the invoice" },
    invoice_number: { type: ["string", "null"], description: "Invoice ID if
present" },
    invoice_date: { type: ["string", "null"], description: "Date in ISO 8601
format" },
    due_date: { type: ["string", "null"], description: "Payment due date if
stated" },
    line_items: {
      type: "array",
      items: {
        type: "object",
        properties: {
          description: { type: "string" },
          quantity: { type: ["number", "null"] },
          unit_price: { type: ["number", "null"] },
          total: { type: "number" }
        }
      }
    },
    stated_total: { type: "number", description: "Total as stated on the
invoice" },
    calculated_total: { type: "number", description: "Sum of line item totals"
},
    currency: { type: "string", enum: ["USD", "EUR", "GBP", "JPY", "other"] },
    currency_detail: { type: ["string", "null"] },
    purchase_order: { type: ["string", "null"] },
    payment_terms: { type: ["string", "null"] },
    confidence: { type: "string", enum: ["high", "medium", "low",
"unclear"] },
```

```
    conflict_detected: { type: "boolean", description: "True if source data
has inconsistencies" }
  },
  required: ["vendor_name", "stated_total", "calculated_total", "currency",
"confidence", "conflict_detected"]
};
```

Key design decisions: **stated_total** and **calculated_total** are both required to enable cross-validation. **conflict_detected** flags inconsistencies for human review. **purchase_order** is nullable because not all invoices have PO numbers. The **currency** enum includes “other” with a detail field for uncommon currencies.

Step 2: Semantic Validation

```
function validateExtraction(data) {
  const errors = [];

  // Cross-field validation: totals must match
  const tolerance = 0.01;
  if (Math.abs(data.stated_total - data.calculated_total) > tolerance) {
    errors.push(`Total mismatch: stated=$${data.stated_total}, ` +
      `calculated=$${data.calculated_total}`);
  }

  // Date validation
  if (data.invoice_date && data.due_date) {
    if (new Date(data.due_date) < new Date(data.invoice_date)) {
      errors.push("Due date is before invoice date");
    }
  }

  // Quantity and price validation
  for (const item of data.line_items || []) {
    if (item.quantity !== null && item.quantity < 0) {
      errors.push(`Negative quantity for: ${item.description}`);
    }
  }

  return { valid: errors.length === 0, errors };
}
```

Step 3: Retry-with-Error-Feedback

When validation fails, retry with the specific errors in context:

```
async function extractWithRetry(document, maxRetries = 2) {
  let extraction = await extractInvoice(document);
  let validation = validateExtraction(extraction);

  for (let i = 0; i < maxRetries && !validation.valid; i++) {
    // Key: include the failed extraction AND specific errors
    extraction = await anthropic.messages.create({
      tools: [extractionTool],
      tool_choice: { type: "tool", name: "extract_invoice" },
      messages: [{
        role: "user",
        content: `Re-extract this invoice. Previous attempt had errors:\n` +
          validation.errors.join("\n") +
          `\n\nPrevious extraction:\n${JSON.stringify(extraction)}` +
          `\n\nDocument:\n${document}`
      }]
    });
    validation = validateExtraction(extraction);
  }

  return { extraction, validation, retriesUsed: /* ... */ };
}
```

This pipeline demonstrates integrated concepts from Domains 4 and 5: JSON schema design with nullable fields (preventing hallucination), tool_choice forced selection (guaranteeing structured output), semantic validation beyond schema compliance, retry-with-error-feedback for recoverable errors, and confidence scoring for human review routing.