Claude Certified Architect – Foundations Certification Study Notes

Code: CCA-F

Agentic Architecture (27%)

Agentic Loop Fundamentals

The core pattern for building Claude-powered agents

> **Domain Weight**
>
> Agentic Architecture & Orchestration accounts for 27% of the CCA-F exam — the highest-weighted domain.

The Agentic Loop Pattern

- Send request to Claude with tools defined → Inspect stop_reason → Execute tools if needed → Return results → Repeat
- The loop continues while stop_reason === 'tool_use' and exits when stop_reason === 'end_turn'
- Tool results MUST be appended to conversation history so Claude can reason about the next action
- Claude decides which tool to call based on context — NOT pre-configured decision trees

Value	Meaning	Action
tool_use	Claude wants to call a tool	Execute the tool, append result, continue loop
end_turn	Claude is finished	Exit the loop, return final response
max_tokens	Output was truncated	Handle continuation or increase max_tokens
stop_sequence	Custom stop sequence hit	Handle based on application logic
refusal	Model refused the request	Handle gracefully, adjust prompt
pause_turn	Server-side tool pause	Wait for server tool completion

Anti-Patterns to Avoid

- Parsing natural language to determine loop termination — always use stop_reason
- Setting arbitrary iteration caps as the primary stopping mechanism
- Checking for assistant text content as a completion indicator
- Pre-configuring tool call sequences instead of letting Claude reason

Multi-Agent Patterns

- Subagents do NOT inherit the coordinator's conversation history — context must be explicitly passed
- Coordinator responsibilities: task decomposition, delegation, result aggregation, deciding which subagents to invoke
- All communication routed through coordinator for observability and control

Subagent Design Principles

Dynamic Selection

Select subagents based on query complexity — don't always route through the full pipeline.

Scope Partitioning

Assign distinct subtopics or source types to minimize duplication across subagents.

Iterative Refinement

Coordinator evaluates synthesis → re-delegates for gaps → re-invokes synthesis loop.

Task Tool

Mechanism for spawning subagents. allowedTools must include "Task" for the coordinator.

> **Common Pitfall**
>
> Overly narrow task decomposition leads to incomplete coverage. For example, decomposing "AI in creative industries" only into visual arts subtopics misses music, writing, and film.

Context Passing

- Context is EXPLICIT — must be provided in the prompt; no automatic inheritance or shared memory
- AgentDefinition config includes: descriptions, system prompts, and tool restrictions per subagent
- Each subagent has its own isolated conversation; coordinator aggregates results

- Return descriptive tool_result errors that help Claude self-correct on the next iteration
- Use is_error flag in tool results to distinguish errors from successful outputs
- Implement exponential backoff for transient failures (rate limits, network errors)
- Set maximum iteration limits as safety nets — but not as primary control flow

Guardrail Approaches

Approach	Description	Use Case
Input validation	Validate tool inputs before execution	Prevent malformed API calls
Output classification	Use a second Claude call to classify outputs	Detect harmful or off-topic responses
Tool restrictions	Limit available tools per agent/phase	Reduce attack surface in production
Human-in-the-loop	Require approval for high-impact actions	Financial transactions, data deletion
Conversation limits	Cap total iterations or token spend	Prevent runaway agents

Tool Design & MCP (18%)

Domain Weight

> Tool Design & MCP Integration accounts for 18% of the CCA-F exam.

MCP Architecture

- Client-server protocol: MCP client (in Claude/app) connects to MCP servers that expose tools, resources, and prompts
- Transport layers: stdio (local processes) and SSE/HTTP (remote servers)
- Three primitives: Tools (callable functions), Resources (data/context), Prompts (reusable templates)
- JSON-RPC 2.0 message format for communication between client and server

MCP vs Direct Tool Use

Feature	Direct Tool Use	MCP
Integration	Per-application custom code	Standardized protocol
Discovery	Hardcoded tool definitions	Dynamic capability discovery
Reusability	Tied to specific app	Server works with any MCP client
Resources	Not supported	Exposes data alongside tools
Complexity	Simple for 1-2 tools	Better for rich integrations

- Write clear descriptions that explain WHEN to use the tool, not just what it does
- Define input_schema with JSON Schema — include types, descriptions, required fields, and enums
- Keep parameter count small (3-5 preferred) — break complex tools into focused smaller tools
- Include examples in descriptions for ambiguous parameters

Error Handling in Tools

- Return is_error: true with descriptive messages so Claude can self-correct
- Include actionable guidance: 'Customer not found. Try searching by email instead.'
- Never return raw stack traces — transform into Claude-friendly messages
- For validation errors, specify which parameter was invalid and what's expected

Tool Categories

Category	Examples	Key Consideration
Read operations	get_customer, search_orders	Cacheable, low risk
Write operations	update_record, send_email	Require confirmation patterns
Computation	calculate_price, validate_input	Deterministic, fast
External API	fetch_weather, query_database	Handle latency and failures

Server Configuration Files

- Claude Code: Configure in ~/.claude/settings.json or project .mcp.json
- Claude Desktop: Configure in claude_desktop_config.json
- Settings levels: project (.mcp.json) → user (~/.claude/settings.json) → enterprise
- Use allowedTools to restrict which MCP tools are available to specific agents

Security Considerations

Input Validation

Validate all tool inputs server-side. Never trust client/model-provided data without sanitization.

Least Privilege

Grant each MCP server only the permissions it needs. Use allowedTools to restrict tool access.

Secrets Management

Use environment variables for API keys. Never hardcode secrets in MCP server configs.

Transport Security

Use stdio for local servers. For remote servers, use HTTPS/TLS with proper authentication.

Claude Code (20%)

CLAUDE.md — The Project Configuration File

Configuring Claude Code behavior for your project

> **Domain Weight**
>
> Claude Code Configuration & Workflows accounts for 20% of the CCA-F exam.

CLAUDE.md Hierarchy

Level	Location	Scope
User	~/.claude/CLAUDE.md	All projects for this user
Project root	./CLAUDE.md	Anyone working on this project
Subdirectory	./subdir/CLAUDE.md	When working in that directory

Key Contents

- Project conventions: coding style, naming conventions, architecture patterns
- Build/test commands: how to build, test, lint, and deploy the project
- Tool restrictions: which tools Claude Code should or shouldn't use
- File patterns: which files to focus on or ignore
- Context: project-specific knowledge that helps Claude make better decisions

> **Exam Tip**
>
> CLAUDE.md is loaded into context automatically. Keep it concise — overly long files waste context window tokens.

Custom Slash Commands & Hooks

Extending Claude Code with custom workflows

Custom Slash Commands

- Used for repetitive workflows: code review, test generation, documentation

Hooks

- Event-driven scripts that run at specific points in Claude Code's lifecycle
- Hook types: PreToolUse (before tool execution), PostToolUse (after tool execution), Notification
- Configured in .claude/settings.json under the 'hooks' key
- Can modify, block, or augment tool behavior — e.g., auto-format after file writes
- Run as shell commands with environment variables for context

Hook Types

Hook	Timing	Use Case
PreToolUse	Before a tool runs	Validate inputs, require confirmation, block dangerous ops
PostToolUse	After a tool completes	Auto-format files, run linters, update logs
Notification	On specific events	Send alerts, log activity, trigger CI

Claude Code in CI/CD Pipelines

Automating workflows with headless Claude Code

Headless Mode

CI/CD Integration Patterns

PR Code Review

Run headless Claude Code on PR diffs to generate review comments. Use --allowedTools to limit to read-only operations.

Test Generation

Auto-generate tests for new code. Allow Write tool but restrict to test file paths only.

Documentation

Generate or update docs on code changes. Commit results back to the PR branch.

Issue Triage

Analyze new issues and suggest labels, assignees, or related PRs using codebase context.

Built-in Tools

Tool	Purpose	Permission Level
Read	Read file contents	Safe — always allowed
Write	Create or modify files	Requires explicit approval or allowedTools
Bash	Execute shell commands	Highest risk — restrict carefully
Grep	Search file contents	Safe — always allowed
Glob	Find files by pattern	Safe — always allowed
Task	Spawn subagents	Moderate — controls delegation

Controlling Claude's behavior and output format

> **Domain Weight**
>
> Prompt Engineering & Structured Output accounts for 20% of the CCA-F exam.

System Prompt Best Practices

- Define role, task, and constraints clearly at the start
- Use XML tags (<instructions>, <context>, <rules>) to organize sections
- Place examples after instructions, before the user input
- Set explicit output format requirements (JSON, Markdown, etc.)
- Include negative constraints: 'Do NOT include...' to prevent unwanted behavior

Response Prefilling

- Pre-fill the assistant turn to force a specific output format: assistant: '{' for JSON
- Useful for enforcing structured output without tool_use
- Can guide Claude to start with a specific language, format, or section
- Combine with stop_sequences for tight output control

Prompt Structure Pattern

Role Definition

Tell Claude what it is: "You are a senior code reviewer specializing in Python security."

Task Specification

Define exactly what to do: "Review this code for security vulnerabilities and suggest fixes."

Output Format

Specify the format: "Return a JSON

Constraints

Set boundaries: "Only flag high and

Getting reliable structured data from Claude

Approaches to Structured Output

Approach	How	Best For
Tool use	Define a tool with input_schema; Claude returns structured params	Most reliable; schema-validated
Prefilling	Start assistant turn with '{' or '['	Simple JSON without tool overhead
Prompt instruction	Ask for JSON in system prompt	Quick but less reliable
Response format	Use response_format API parameter	API-level enforcement

JSON Schema with Tool Use

- Define a tool with the desired output schema as its input_schema
- Claude will 'call' the tool with properly structured data
- Extract the structured data from the tool_use content block
- Most reliable method because the API enforces schema compliance

Validation Strategies

- Always validate structured output against schema before using it
- Implement retry logic for malformed responses
- Use discriminated unions in schemas for type safety

Leveraging Claude's reasoning capabilities

Chain-of-Thought Prompting

- Ask Claude to 'think step by step' for complex reasoning tasks
- Use XML tags to separate thinking from final answer: <thinking>... </thinking>
- Improves accuracy on multi-step problems, math, and logic puzzles
- Trade-off: uses more tokens but produces more reliable results

Extended Thinking

- API parameter that gives Claude a dedicated thinking block before responding
- Set budget_tokens to control how much thinking is allowed
- Thinking content is returned separately from the response
- Best for complex analysis, planning, and multi-step reasoning tasks
- Cannot be used with: temperature > 1, prefilling, or some streaming modes

When to Use Each

Technique	When to Use	Token Impact
No CoT	Simple, direct tasks (classification, extraction)	Minimal
Prompt CoT	Medium complexity (analysis, comparison)	Moderate
Extended Thinking	Complex reasoning (architecture, debugging)	High but worthwhile

Understanding the Context Window

Managing Claude's input and output token limits

> **Domain Weight**
>
> Context Management & Reliability accounts for 15% of the CCA-F exam.

Context Window Basics

- Context window = total tokens available for input + output combined
- Claude 4 models: up to 200K token context window
- Longer context ≠ better performance — relevance matters more than volume
- System prompt, conversation history, and tool definitions all consume context

Context Optimization Strategies

Summarization

Periodically summarize long conversations to reduce token count while preserving key decisions.

Selective Loading

Only include relevant context. Use tools to fetch data on-demand rather than loading everything upfront.

Compaction

Claude Code's automatic context compaction compresses conversation history when approaching limits.

Context Hierarchy

Place most important context first. Claude attends more strongly to earlier content in long contexts.

Prompt Caching for Performance

Reducing latency and cost with cached prefixes

How Prompt Caching Works

- Minimum 1024 tokens (Claude 4 Sonnet) or 2048 tokens (Claude 4 Opus) for caching to activate
- Cache TTL is 5 minutes — refreshed on each hit

What to Cache

Content Type	Cache Benefit	Example
System prompts	High — static across requests	Role definition, rules, output format
Tool definitions	High — rarely change	20+ tool schemas in agentic systems
Reference documents	High — loaded repeatedly	Code files, documentation, specs
Few-shot examples	Medium — reused per task type	Example input/output pairs
Conversation history	Low — changes each turn	Only cache if multi-user shared context

> **Caching Rule**
>
> Content must be in the same position and identical across requests for cache hits. Any change to cached content invalidates the cache for that block and all subsequent blocks.

- Use automated metrics: accuracy, format compliance, latency, cost per request
- Implement A/B testing for prompt changes — never deploy untested prompt modifications
- Track regression across model versions and prompt updates

Production Reliability

Rate Limiting

Implement client-side rate limiting and retry with exponential backoff for API errors.

Fallback Strategies

Use smaller/faster models as fallbacks. Degrade gracefully when Claude is unavailable.

Monitoring

Log all API calls, tool executions, and errors. Track token usage and response quality metrics.

Idempotency

Design tool executions to be safely retryable. Use idempotency keys for write operations.

Testing Agentic Systems

- Unit test individual tools independently from Claude
- Integration test the full agentic loop with mock tool responses
- End-to-end test with real API calls on representative scenarios
- Stress test with adversarial inputs to find failure modes

Brakto