

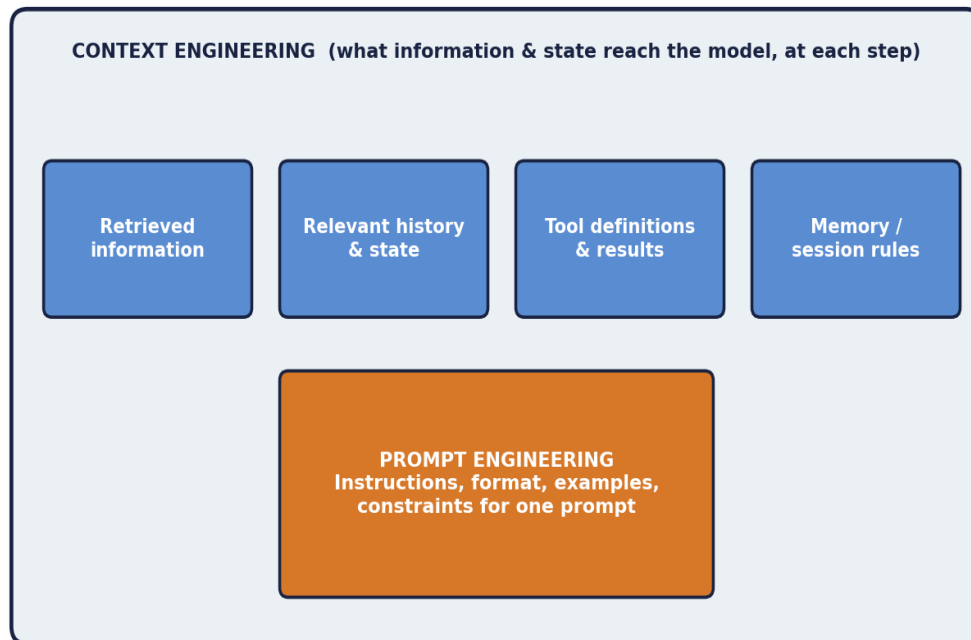
CLAUDE CERTIFIED DEVELOPER

FOUNDATIONS

The Ultimate Study Guide

Every topic, design pattern, and decision rule tested across the practice question banks, organized from fundamentals to production operations.

Prompt Engineering vs. Context Engineering



How to Use This Guide

The developer exam is scenario-driven: nearly every question describes a team with a problem and asks for the most appropriate engineering response. The right answers follow a small set of recurring principles. Read Parts I–VI for the concepts, then use Part VII (answer patterns) and Part VIII (cheat sheet) for final review.

Part	Title	Focus
I	Core Concepts	Context vs. prompt engineering, variability, tokens, model selection, requirements
II	Prompting & Structured Output	Schemas, few-shot, format drift, evals, iteration
III	Agent Architecture & Tools	Orchestrator, manager, graph, context isolation, tool contracts, hooks, SDK
IV	MCP, Skills, Claude Code & Platforms	Transports, Skill vs. MCP, CLAUDE.md, settings.json, CI, Bedrock
V	Reliability & Operations	Validation, retries, rate limits, streaming, debugging, observability, lifecycle
VI	Security & Safety	Threat model, prompt injection, least privilege, enforcement, keys
VII	Software Engineering Practice	Code review, refactoring, testing, upgrades, change management
VIII	Answer Patterns & Cheat Sheet	How to spot the right answer fast; one-page recall

Part I — Core Concepts

1.1 Prompt Engineering vs. Context Engineering

These are not interchangeable terms, and context engineering has not simply "replaced" prompt engineering.

- Prompt engineering: crafting effective instructions for a specific output (wording, format, examples, constraints).
- Context engineering: the broader discipline of managing the information and state supplied to the model across steps — relevant history, retrieved information, tool definitions and results, memory, and what is kept or removed.

Prompt Engineering vs. Context Engineering

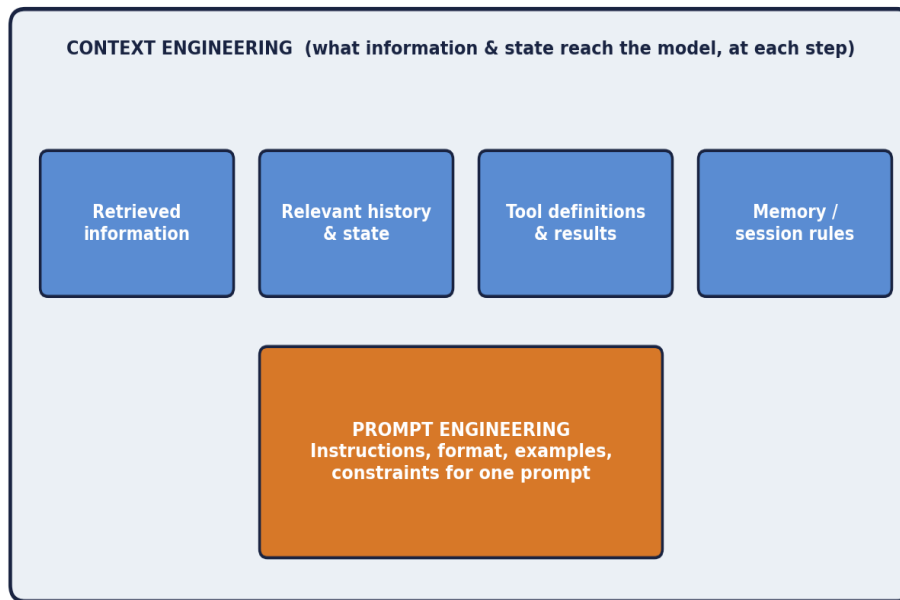


Figure 1.1 — Prompt engineering sits inside the larger scope of context engineering.

Exam takeaway: Prompt engineering = effective instructions. Context engineering = managing what information and state flow to the model across steps.

1.2 LLM Variability (Non-Determinism)

Claude generates output token by token using sampling, so identical prompts can produce slightly different results. This is a normal property of probabilistic generation, not automatically a bug.

- Explain it to stakeholders as manageable: use validation, retries, and parameter settings where consistency matters.
- It is not caused by network latency, and pinning a model snapshot does not eliminate it entirely.
- Unexpected wording alone (e.g., the output sometimes repeats prompt text and sometimes rephrases it) does not prove a request-construction bug. First distinguish normal generation variability from real application-level differences, such as comparing the actual requests sent.

Exam takeaway: Variability is normal and managed — not dismissed as "accepted behavior" and not treated as proof of a bug.

1.3 Tokens, Pricing and Cost Modeling

- Input and output tokens are billed separately; output tokens are typically billed at a higher rate. Track both separately in cost models.
- Output and input rates differing is expected — not a billing error.

1.4 Model Selection

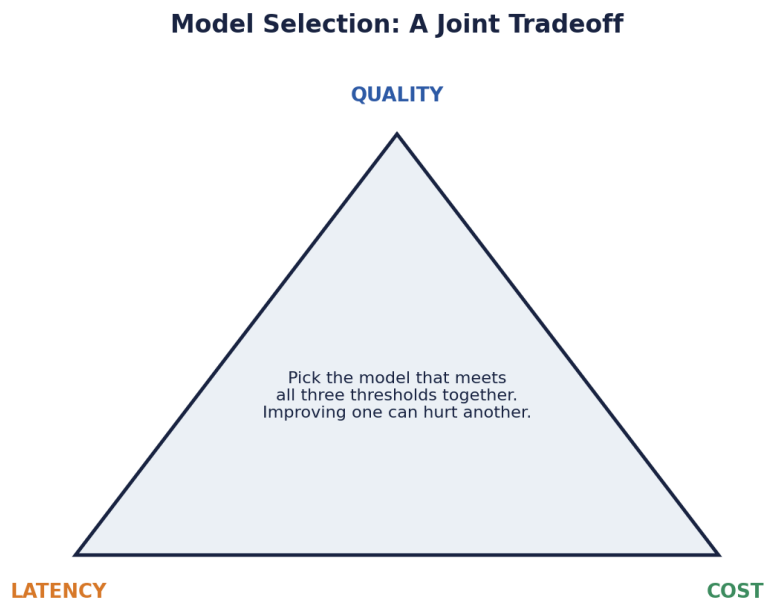


Figure 1.2 — Quality, latency and cost are chosen together.

Select the model that fits the task's quality, latency, and cost requirements together, recognizing that improving one dimension can affect the others. Distractors pick one dimension first and "review the others afterward."

Upgrading to a new model

Example: a new model improves reasoning 8% but yields malformed output on ~3% of requests because the system-prompt format changed. Correct approach: adapt the application's system prompt to the new format, re-evaluate on a representative workload, and upgrade only if malformed output is eliminated while the reasoning gains are preserved. Do not upgrade on the 8% headline alone, rely only on downstream validation, or stay on the old model permanently.

1.5 From Business Goal to Requirements

Given a vague goal such as "agents should help analysts produce client memos faster," the first step is to interpret the functional and infrastructure requirements implied by the business goal. Only then choose models, architecture, or integration points. Benchmarking precedents, picking a model, or asking where to insert Claude all come later.

Part II — Prompting & Structured Output

2.1 Requesting Structured JSON

- State the required schema explicitly in the prompt and instruct Claude to produce only output matching it.
- Also validate the output against the schema in application code before passing it downstream. Prompt plus validation is the complete answer.
- Wrong answers: leave format to Claude, mix JSON and free text, describe the schema loosely, or leave validation entirely to downstream systems.

2.2 Few-Shot (Multi-Shot) Prompting

When you have well-labeled examples of the desired output (typical cases or edge cases), include them in the prompt as few-shot examples. Do not embed them in a database for retrieval at inference, train a custom model, or tell users to avoid edge cases.

2.3 Format Drift Over Multiple Turns

If JSON is correct for the first turns and then loses structure, identify format drift, examine how the conversation context evolves, and address it through context management or output validation. Truncating, switching to a smaller model, or restarting after every turn does not address the cause.

2.4 Eval-Driven Prompt Iteration

1. Identify the underperforming category from eval results or production data.
2. Iterate on the prompt for that category.
3. Validate with evals, including the existing cases, to catch regressions.
4. Keep refining; repeat.

Use a representative evaluation set containing normal cases, edge cases, and important failure modes, and compare against the previous version. Evaluating only the examples used to develop the prompt risks overfitting; deploying because those examples improved is not evidence.

Exam takeaway: Evidence over intuition: eval sets that cover normal + edge + failure cases, with regression comparison, beat warning users, rewriting everything, or splitting apps.

2.5 Unexpected Refusals

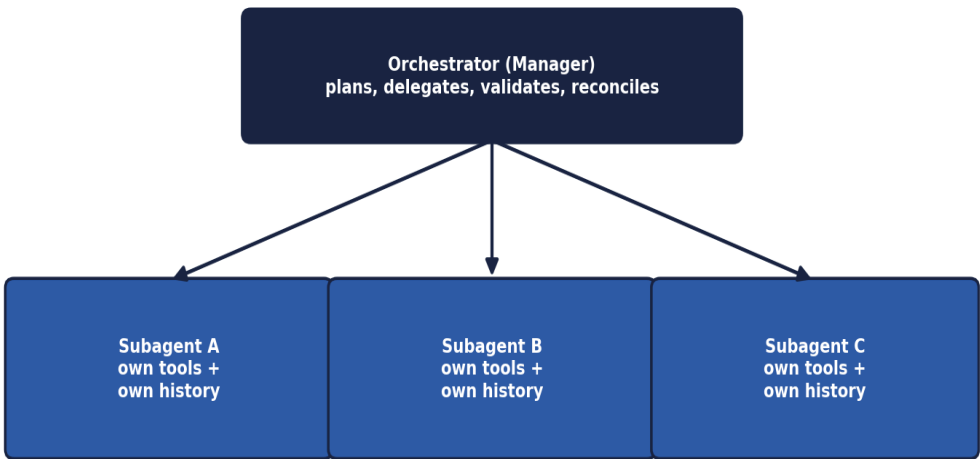
When Claude refuses in-scope questions it previously answered, the first step is a differential comparison: identify the refused questions and compare them with previously answered ones to find what changed. Then use traces and targeted experiments to find the trigger. Do not start removing instructions at random.

Part III — Agent Architecture & Tools

3.1 Choosing an Architecture Pattern

Situation	Pattern
Break a task into pieces, hand each to a focused specialist, consolidate	Orchestrator and subagents
Top-level router sequences specialists; each has own tools and own isolated history (e.g., flights, hotels, ground transport)	Manager/supervisor with isolated context per subagent
Distinct stages with state-dependent transitions and loops (e.g., extraction <-> eligibility, then decision)	Graph-based workflow
Stages with distinct inputs, outputs and success criteria (e.g., triage then response drafting)	Separate Claude calls per stage
One simple tool-driven task	Single tool-use loop

Orchestrator / Manager with Isolated Subagents



Subagents do NOT share history; handoffs go through the orchestrator using explicit contracts (schema-validated outputs, per-agent tool/token budgets).

Figure 3.1 — Orchestrator with isolated subagents.

Graph-Based Workflow (state-driven, with loops)



Each node has its own prompt and success criteria; transitions depend on state.
A linear chain cannot loop back; a single tool loop has no explicit stages.

Figure 3.2 — A graph allows loops between subtasks.

Fixing a fragile multi-agent system

If subagents overrun token budgets and pass contradictory outputs along, strengthen the orchestrator: enforce per-subagent tool/token budgets, validate outputs against a schema before passing them forward, and define explicit handoff contracts. Letting subagents read each other's outputs, collapsing into one giant call, or adding retries alone does not solve it.

3.2 Context Isolation and Pruning

- Context isolation: scope each step (via subagents or multi-step workflows) to only the context it needs. This reduces noise and prevents inappropriate information flow.
- Tool output pruning: remove tool results from active context once later steps no longer need them. Keeping everything indefinitely fills the window; switching to a smaller model or caching accumulated outputs does not fix it.

3.3 Session Lifecycle

Define explicit session expiration rules, state-reset triggers, and rules for starting fresh sessions so stale content does not leak into new conversations. One short global lifecycle, user-only responsibility, or reset-only designs are incomplete.

3.4 Tool Design

- Tool descriptions matter: the model uses them to decide when to call a tool. They convey purpose, usage conditions, inputs and constraints that names alone cannot.
- Keep tool schemas strict. If the agent sends undeclared arguments, validate before dispatch and return a structured error so the agent can retry. Do not loosen the schema or rely on a system-prompt instruction as the main enforcement.
- Built-in vs. custom tool: if a built-in nearly covers the need, evaluate whether the gap can be closed by extending/configuring it; build custom only when justified.

3.5 Hooks

Scope hooks by risk: apply them to high-risk actions (destructive operations, sensitive-data access) and remove them from low-risk actions to avoid needless slowdown. Do not disable all hooks or replace them with prompt instructions.

3.6 Claude Agent SDK

- When the team has chosen the Agent SDK, use its tool interface and let it handle the loop, dispatch, and history. Do not recreate the core loop.
- Direct SDK vs. abstraction layer: with Claude as the only provider and no multi-provider roadmap, integrate directly with the Claude SDK. A third-party abstraction adds complexity without current value.
- Custom SDK overrides (history, retries, error handling): review each customization individually against its original reason and current SDK capabilities. Do not revert all or keep all.

Part IV — MCP, Skills, Claude Code & Platforms

4.1 MCP Transports and Communication

Choosing a Communication Pattern

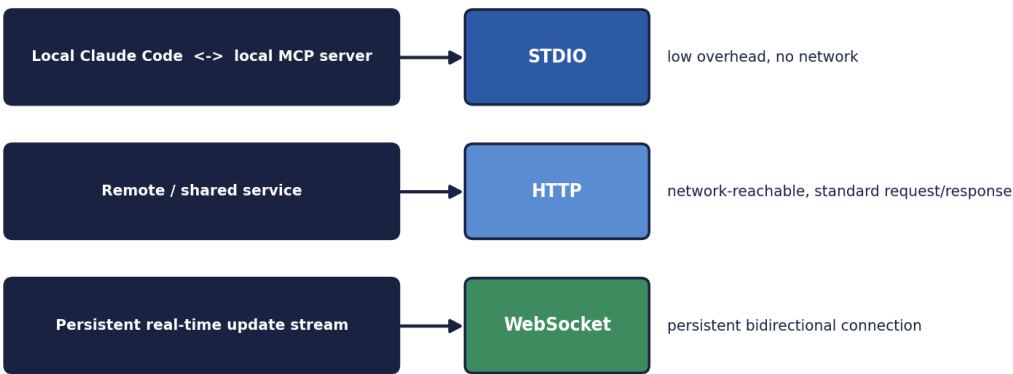


Figure 4.1 — Match the transport to the situation.

- Local Claude Code session talking to a local MCP server: STDIO (low overhead, no unnecessary network).
- Persistent bidirectional real-time updates: WebSocket. Not a held-open HTTP request, filesystem polling, or repeated short HTTP polling.

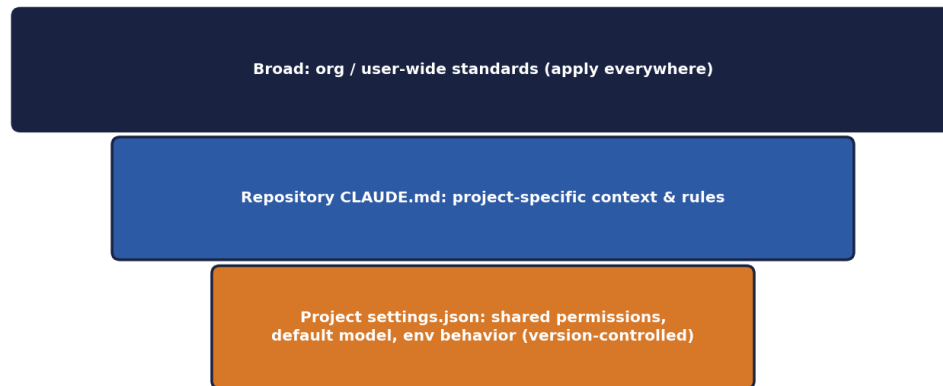
4.2 Skills vs. MCP Servers

Skill	MCP server
Bundles prompts, scripts and data into a package the model uses as a unit	Exposes resources, tools and prompts through a standard client interface (protocol)
Reusable instructions/capabilities	Standardized integration with external systems

They are not equivalent or interchangeable, and neither is simply "older" or "always preferred."

4.3 Claude Code Configuration

CLAUDE.md and Settings: Broad to Specific



Scope general standards broadly; keep repo-specific rules local; avoid duplication.

Figure 4.2 — Scope instructions from broad to specific.

- CLAUDE.md hierarchy: general standards scoped broadly, project-specific rules in each repository's local CLAUDE.md. Avoid a single central file linked everywhere, manual per-developer setup, or an external docs site.
- Project-level settings.json: shared permissions, default model selection, and environment-specific behavior that persists with the repository and is consistent across developers. Not local env vars, spreadsheets, or manually pasted prompts.
- Claude Code in CI: run in headless mode with required permissions configured in settings.json, and capture its output. Do not disable confirmation prompts globally or require a developer to approve each prompt.

4.4 Deploying Through Amazon Bedrock

If a contract requires Claude calls to flow through Amazon Bedrock, build to that from the start: invoke Claude via the Bedrock-compatible API path while keeping application logic provider-agnostic. Do not build direct first and migrate later, build two implementations, or ignore the requirement.

Part V — Reliability & Operations

5.1 Output Validation and Defensive Parsing

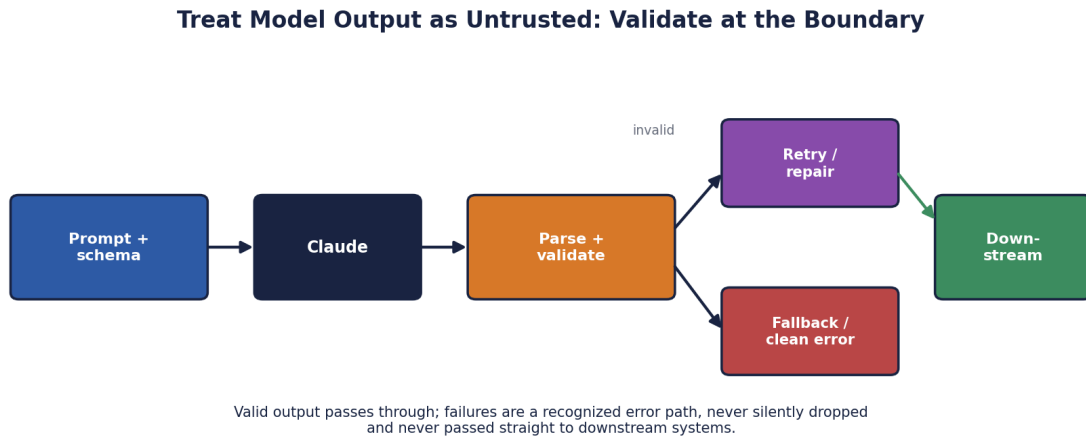


Figure 5.1 — Validation failures are a recoverable error path.

- Treat model output as untrusted input at the application boundary.
- Defensive parsing handles unexpected fields, missing values and type mismatches.
- Validation failures should trigger retry, repair or fallback before failing the request. Do not disable validation, push malformed output downstream, or crash requiring manual recovery. Do not silently drop output either.

5.2 Rate Limits

Identify the applicable rate limits, design the application to stay within them at peak hours, and use exponential backoff when limits are reached. Traffic smoothing helps, but knowing limits plus backoff is the core answer. Streaming does not avoid rate limits.

5.3 Streaming

To show output to users as it is generated, use streaming responses (incremental events for progressive display). Structured output, prompt caching and the batch API do not do this.

5.4 Debugging and Observability

- Inconsistent outputs: capture full traces (input, system prompt, user messages, model output, parameters) and compare consistent vs. inconsistent runs. Do not just retry, lower temperature blindly, or accept it.
- Intermittent multi-step agent failures: capture structured traces covering the request, model configuration, tool calls and results, validation outcomes, errors and timing, while protecting sensitive data. Logging only the final response, disabling logging, or raising temperature does not help.

5.5 Post-Production Lifecycle

Claude Application Lifecycle

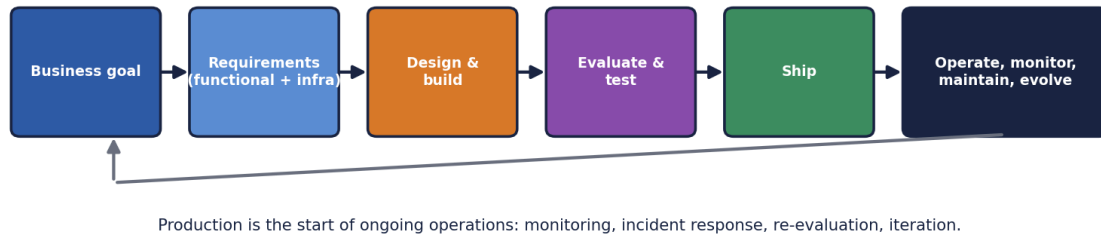


Figure 5.2 — Shipping begins operations.

After shipping, the next phase is to operate and maintain: monitoring, incident response, evaluation and planning evolution. Shipping is not the final stage and not a hand-off with no further ownership.

Part VI — Security & Safety

6.1 AI-Specific Threat Model

Traditional web threats (XSS, SQL injection, DoS) still matter but are not AI-specific. Add these AI-specific categories:

- Prompt injection.
- Data leakage from prompts or context.
- Jailbreak attempts.
- Unsafe model output that bypasses intended controls.

6.2 Prompt Injection Defense

When processing user-provided documents with access to internal tools: separate untrusted content from trusted instructions, constrain tool permissions, and validate sensitive actions before execution. Never treat uploaded text as trusted instructions, and do not give documents priority over system instructions.

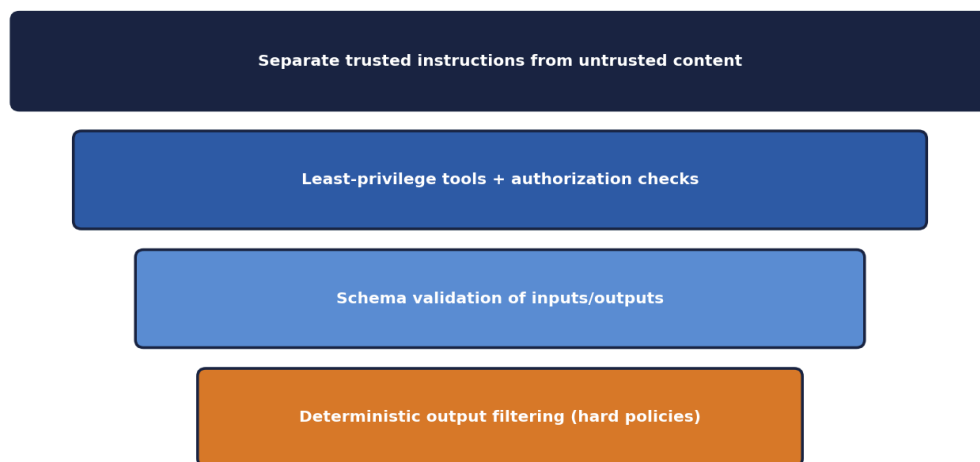
6.3 Least Privilege

Expose only the tools required for the current task and add explicit authorization checks for sensitive operations (e.g., destructive account actions). A prompt warning is not a control, and model confidence is not authorization.

6.4 Enforcing Hard Content Policies

For categories that must never be produced, add deterministic output filtering that checks every response before it reaches the user. System-prompt instructions, model self-review, and user reports are weaker, since they are not guarantees.

Defense in Depth for Claude Applications



Never rely on a system-prompt instruction alone for a hard requirement.

Figure 6.1 — Layered controls.

6.5 API Key Management

Keep keys in a secrets manager and use distinct keys per environment (development, staging, production), so a compromise in one does not affect the others and each can be rotated or revoked independently.

6.6 Session Hygiene

Stale context leaking into new conversations is also a data-leakage risk, which is why explicit session expiration and reset rules matter (see 3.3).

Part VII — Software Engineering Practice

7.1 Code Review of Claude Features

A feature that works on the happy path but hard-codes prompt and model, lacks error handling, and has no integration tests should be sent back. Request all three: move prompt/model to configuration, add Claude API error handling, and add integration tests. Partial fixes or approving with follow-ups are wrong.

7.2 Refactoring

When duplicated logic and tangled control flow already slow feature work, plan a focused refactoring pass first: consolidate duplication, separate concerns, simplify control flow. Postponing for releases, one giant function, or only incidental refactoring are distractors.

7.3 Layered Testing

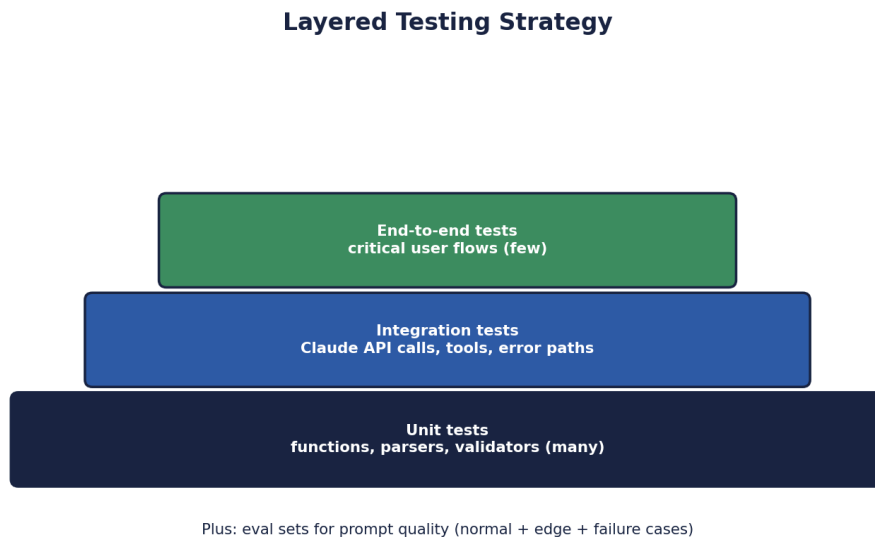


Figure 7.1 — Unit, integration and end-to-end tests, applied consistently.

Formalize unit tests for functions, integration tests for Claude integrations, and end-to-end tests for critical user flows — applied consistently. Not e2e only, not ad hoc plus peer review, not TDD alone.

7.4 Change Management

Before rolling out a configuration or prompt-version change used by multiple downstream systems, assess the impact on each downstream system and coordinate with owners. Passing staging is not enough, since prompt changes can alter output contracts.

7.5 Evaluating Inherited Customizations

Judge each customization on its merits (original requirement vs. current capability). Blanket reverts or blanket keeps are the trap answers.

Part VIII — Answer Patterns & Cheat Sheet

8.1 Patterns That Recur Across the Exam

- Strong validation: validate inputs, tool arguments, and model output in application code, and prompt instructions alone are never the only safeguard.
- Isolation: separate trusted vs. untrusted content, and scope context per step or subagent.
- Least privilege: only the tools needed, with authorization for sensitive actions.
- Evidence-based decisions: traces, evals, representative workloads, and regression comparison before changing prompts, models or configuration.
- Root cause first: fix the cause (e.g., adapt the prompt format) instead of masking symptoms.
- Recoverable errors: retry, repair, fallback, backoff — not crash, ignore, or silently drop.
- Right-size complexity: direct SDK when single-provider; built-in tool before custom; refactor before piling on features.
- Be wary of extremes: options with "always", "never", "disable everything", "revert all", or "keep all" are usually wrong.
- Requirements before solutions: translate the business goal before choosing models or architecture.

8.2 One-Page Cheat Sheet

Topic	Remember
Prompt vs. context engineering	Instructions vs. managing information/state across steps
Variability	Normal sampling behavior; manage with validation, retries, parameters
Pricing	Input and output billed separately; output usually higher
Model choice	Quality + latency + cost together; re-evaluate on upgrade
Structured output	Schema in prompt + application-side validation + retry/repair/fallback
Few-shot	Labeled examples go directly in the prompt
Evals	Representative set: normal + edge + failure; compare to previous version
Orchestrator	Decompose, delegate to specialists, consolidate; enforce budgets and contracts
Manager + isolated context	Subagents keep own tools/history
Graph workflow	State-based transitions with loops, per-node criteria
Separate stages	Distinct inputs/outputs/criteria = separate calls
Context hygiene	Isolate per step; prune stale tool output; explicit session rules
Tools	Detailed descriptions; strict schema; validate before dispatch; structured errors
Hooks	Scope to high-risk actions
Agent SDK	Use its tool interface/loop; direct SDK when single provider

MCP transport	Local = STDIO; persistent real-time = WebSocket
Skill vs. MCP	Packaged capability vs. standard protocol interface
Claude Code	CLAUDE.md hierarchy; settings.json; headless in CI
Bedrock	Honor contract from the start; provider-agnostic logic
Rate limits	Know limits, stay under, exponential backoff
Streaming	Incremental output for progressive display
Debugging	Full traces; differential comparison for refusals
Security	Prompt injection, leakage, jailbreak, unsafe output; isolation; least privilege; deterministic filters
Keys	Distinct per environment in a secrets manager
Engineering	Review requires config + error handling + tests; refactor first; layered tests; assess downstream impact
After launch	Operate, monitor, maintain, evolve

Good luck on the Claude Certified Developer Foundations exam.