MockCert

Study guide | Anthropic | CCAR-P

Claude Certified Architect – Professional

Notes for every domain of the Claude Certified Architect – Professional exam, organized by the official exam guide's domains and weights.

By MockCert about 127 min read Updated November 24, 2026

CONTENTS

DOMAIN 1 · 18%
Solution Design and Architecture 2

DOMAIN 2 · 13%
Claude Models, Prompting, and Context Engineering 16

DOMAIN 3 · 21%
Integration 29

DOMAIN 4 · 17%
Evaluation, Testing, and Optimization 47

DOMAIN 5 · 16%
Governance, Safety, and Risk Management 62

DOMAIN 6 · 7%
Stakeholder Communication and Lifecycle Management 75

DOMAIN 7 · 8%
Developer Productivity and Operational Enablement 82

MockCert is an independent certification preparation platform. It is not affiliated with, endorsed by or operated by Anthropic. Exam domains and weights follow the official exam guide. These notes were last updated on November 24, 2026; the notes at mockcert.com are kept up to date as the exam changes.

Domain 1 | 18% of the exam | 20 min read

Solution Design and Architecture

Overview

This domain is about 18% of the exam, roughly 12 of its 70 questions.

Every point below comes from Anthropic's official documentation; each section lists its sources.

Task Delegation Between Claude and Humans

Start here

THINK FIRST

When you map a business process onto Claude, which steps can run alone, which need a person first, and why?

Why it matters: Questions describe a business process and ask which steps Claude, a human, or both should own, and why.

Reversibility and stakes decide where a human gate belongs. Steps that change nothing, or that can be undone and audited, can run on their own. Steps with an immediate effect the agent cannot undo need human approval before they run.

Key terms

Term	Meaning
Reversibility	Whether a step's effect can be undone afterward, such as rewinding a file edit.
Pre-execution gate	A pause that waits for human approval before an action runs, not after.
Deterministic control	A rule enforced the same way every time, rather than left to the model's own judgment.
Permission mode	A Claude Code setting that decides which actions Claude takes without asking you first.

Sort steps by what they change

Side-effect-free and reversible steps can run without a gate. Reading a request or looking up an account changes nothing. An update to contact details can be undone and is logged. So Claude can own these steps alone, and no reviewer needs to slow them down.

Gate the irreversible step up front

An immediate charge the agent cannot undo needs approval first. A billing plan change creates a financial effect right away. Because the agent cannot reverse it, a person must approve it before it runs. The gate sits only on that step, so low-risk paths keep moving.

EXAM TRAP

Reviewing an audit log afterward feels safe, but it comes too late: the charge has already happened and cannot be prevented.

Why the other gates fail

Only a deterministic gate on the risky step holds up. Each tempting alternative breaks for a different reason. Compare them by when they act and what goes wrong.

Where to place a human check for an irreversible billing step

Control	When it acts	Problem
Review of the audit log	After execution	Cannot stop a charge that is already made
Approval of the update plan	Before every request	Delays all requests; execution can drift from the plan
Model's self-reported confidence	When the model decides	Not reliable; the gate is left to model judgment
Approval gate on the billing step	Before that step only	None: deterministic and placed where stakes are high

EXAM TRAP

A confidence threshold sounds like a smart filter, yet an irreversible step needs a rule the model cannot talk its way past.

Undo works locally, not remotely

File edits can be rewound; remote actions cannot. Claude Code snapshots a file before it edits it, and pressing **Esc** twice rewinds. Databases, APIs and deployments get no checkpoint. Control those actions with permission modes and rules instead.

EXAM TRAP

Checkpoints do not make every action safe to delegate. A deployment or database write stays done even after you rewind your files.

Choose how much Claude asks

Permission modes set the level of human involvement. Pick the mode that fits the stakes of the work in front of you.

Claude Code permission modes and who decides

Mode	What Claude does	Human role
Manual	Asks before file edits and shell commands	Approves each edit and command
Accept edits	Edits files and runs mkdir or mv freely	Approves other commands
Auto	A classifier blocks risky actions	Rarely asked; risky actions are blocked
Plan	Explores and proposes without editing files	Reviews the plan before any change

Collaboration and ownership

Humans can steer Claude at any point while it works. You can interrupt, add context or ask for another approach mid-task. On the Claude Platform, permission policies decide whether tools run alone, wait for you, or are checked on each call. Teams still need a control that keeps humans owning AI-generated code.

Key takeaways

- Reversibility and stakes decide where a human approval gate belongs.
- Gate an irreversible step before it runs; a review afterward cannot undo it.
- Use a deterministic gate, never the model's own confidence, for irreversible actions.
- Local file edits can be rewound, but remote systems need permission rules.

Sources: Claude Certification Program Exam guide Claude Certified Architect – Professional Exam Guide (opens in a new tab) · How Claude Code works - Claude Code Docs (opens in a new tab) · Features overview - Claude Platform Docs (opens in a new tab)

End-to-End Architecture Evaluation

Start here

THINK FIRST

Looking at a proposed Claude architecture end to end, how do you spot the component that breaks first under load or a weaker trust boundary?

Why it matters: Questions describe an architecture plus a load or trust condition and ask which component fails first or which requirement is unmet.

Walk the path from input to output and test each component against the stated condition. Under load, look for hard limits and time windows. Under trust pressure, look for credentials that reach too far or endpoints with no authentication.

Key terms

Term	Meaning
Trust level	How far a tenant or workload is trusted; one deployment should serve only one.
Environment key	A credential able to read events and post tool results across all sessions in its environment.
Workload Identity Federation	Exchanging a JWT from your identity provider for a short-lived Anthropic token.
custom_id	Your own label on each batch request, used to match results back to requests.

Batch processing limits

Batches have hard caps on size and time. One batch holds up to **100,000** requests or **256 MB**, whichever comes first. Most finish within **1** hour, but a batch expires if processing is not done within **24** hours.

EXAM TRAP

Batch results do not come back in request order. Without meaningful `custom_id` values, the output stage cannot match results to requests.

Cost and latency surprises

Some components behave differently at scale or on first use. High-throughput batches may slightly exceed the workspace spend limit. A new structured outputs schema adds latency on first use while its grammar compiles; compiled grammars stay cached **24** hours from last use.

Token refresh under failure

A WIF token outage surfaces as errors near expiry. The SDK tries an advisory refresh at expiry minus **120** seconds and keeps the cached token if that fails. A failed mandatory refresh at expiry minus **30** seconds raises an error.

Token lifetime and reuse

Retry loops are the usual cause of JWT reuse. A JWT with a `jti` claim is single-use by default, so reuse fails with `jti_reused`. Token lifetime is the lesser of **3,600** seconds by default and twice the JWT's remaining life, never under **60** seconds.

EXAM TRAP

Federation does not make the system secure by itself. It is only as strong as the identity provider that signs the JWT.

Trust boundaries in self-hosting

One deployment serves one tenant at one trust level. Environments in a deployment do not separate trust levels, and the server API key reaches every environment. Mixed-trust tenants in one deployment fail the access-control requirement.

Credentials inside sandboxes

Roles only limit requests made with the caller's own token. Agent code can call the server with any credential its sandbox holds. Give every session its own sandbox and keep the environment key out of it, or one session can act in all.

Exposed endpoints

Two endpoints rely on network placement for protection. Compare the self-hosted server's risky endpoints and what protects them.

Self-hosted server endpoints that need network protection

Endpoint	Weakness	Protection
Prometheus metrics	Asks for no credential	Control where it listens
Egress gateway listener	Plain HTTP; replies carry stored credential values	Let only the gateway reach it

Maintainability failure points

Defaults and proxies can break the whole system. With no database, the default, agents and sessions live in memory and a restart erases them. If your own proxy rejects `metadata.user_id` in the body, every model call fails.

EXAM TRAP

A self-hosted server does not persist sessions by default. Without a configured database, a restart loses all agents and session history.

Key takeaways

- A batch caps at **100,000** requests or **256 MB** and expires after **24** hours unfinished.
- Never mix tenants or trust levels in one self-hosted deployment.
- Keep the environment key out of each session's sandbox.
- Mint a fresh JWT per WIF exchange; reused `jti` claims fail.
- Configure a database, or a server restart erases agents and sessions.

Sources: Batch processing - Claude Platform Docs (opens in a new tab) · Data and access in self-hosted Claude Managed Agents - Claude Platform Docs (opens in a new tab) · Workload Identity Federation - Claude Platform Docs (opens in a new tab) · Structured outputs - Claude Platform Docs (opens in a new tab)

Architectural Pattern Selection

Start here

THINK FIRST

Given a use case's reuse, governance and risk needs, when should you package guidance as a Skill rather than a prompt, tool or custom loop?

Why it matters: Questions describe reuse, governance and risk constraints and ask which architecture fits and what it delegates to Claude.

Each pattern draws a line between what Claude decides and what your code or Anthropic runs. Tools let Claude choose actions your application executes. Skills package reusable guidance that loads on demand, and they need trusted sources and independent review.

Key terms

Term	Meaning
Delegation boundary	The line between what Claude decides and what your code, or Anthropic, runs and controls.
Client tool	A tool Claude chooses to call but your application runs.
Skill	Packaged instructions and guidance that Claude loads on demand when the task calls for them.
Version pinning	Naming a specific Skill version so new uploads cannot change what production runs.

Tools extend the model

With tools, Claude decides when to call; someone else runs the call. Claude picks a tool from the request and the tool's description, then returns a structured call. Your application runs client tools, and Anthropic runs server tools. Execution stays outside the model.

Who runs the agent loop

Build paths differ in how much of the loop you own. Choose by how much control you need versus how much you want to offload.

Three ways to build a Claude agent compared

Feature	Messages API	Agent SDK	Managed Agents
Agent loop	You write it	Provided, in your process	Hosted by Anthropic
Tool execution	Your own tools and infrastructure	Provided, in your process	Hosted by Anthropic

Feature	Messages API	Agent SDK	Managed Agents
Control	Most	In between	Least; you offload the most

EXAM TRAP

Managed Agents is not the choice for maximum control. The Messages API gives the most control, because you write the loop yourself.

When reuse points to Skills

Skills suit guidance repeated across many conversations. A prompt is a conversation-level instruction for a one-off task. A Skill loads on demand, so the same guidance is not repeated each time. Only its ~**100**-token metadata is always loaded; instructions under **5k** tokens load when triggered.

EXAM TRAP

Skills are not loaded in full at startup. Only the short metadata is always present; instructions load when the Skill triggers.

Scripts move work out of the model

Bundled scripts give deterministic steps without filling context. Claude runs a Skill's scripts through bash. Only the output enters the context, not the code. That makes scripts a good boundary for exact, repeatable operations.

Skills in API deployments

API Skills run in a locked-down environment. They have no network access and cannot install packages at runtime. A request can carry at most **20** Skills. A role needing more should consolidate Skills or route tasks to different Skill sets.

Governance for Skills

Skills are code, so treat them as supply-chain risk. Use Skills only from yourself or Anthropic, since a malicious one can misuse tools. Bundled scripts run with full environment access. Authors should not review their own Skills.

Release and isolation controls

Pin versions and isolate tenants in production. Without a pinned version, any new upload in the workspace changes production at once. Skills API uploads are not scanned, so rely on review and pinning. On multitenant platforms, give each tenant its own workspace.

EXAM TRAP

Content scanning does not protect Skills uploaded through the Skills API. API deployments must rely on the review checklist and version pinning.

Growing a Skill library

Start narrow, then bundle by role. Begin with narrow, workflow-specific Skills rather than broad ones. As patterns repeat across the organization, consolidate related Skills into role-based bundles.

Key takeaways

- With tool use, Claude chooses the call, but your app or Anthropic executes it.
- The Messages API gives the most control; Managed Agents offloads the most to Anthropic.
- Pick Skills for reusable guidance that should load on demand across conversations.
- Pin production Skills to versions so new uploads cannot change what agents run.
- Custom Skills are isolated by workspace, so give each tenant its own workspace.

Sources: Guides to common use cases - Claude Platform Docs (opens in a new tab) · Agent Skills - Claude Platform Docs (opens in a new tab) · Skills for enterprise - Claude Platform Docs (opens in a new tab) · Using Agent Skills with the API - Claude Platform Docs (opens in a new tab)

Orchestrator and Subagent Decomposition

Start here

THINK FIRST

When you review a proposed orchestrator and subagent design, where will its task boundaries, control flow or result synthesis break?

Why it matters: Questions present a multi-agent design with observability and error-recovery needs and ask where it fails and what fixes it.

Key terms

Term	Meaning
Orchestrator	The parent agent that delegates tasks to subagents through a prompt string and receives only their final messages.
Prompt string	The Agent tool input that carries all the context a non-fork subagent receives.
Workflow tool	Moves orchestration into a script the runtime runs outside the conversation context.
Thread	One agent's conversation within a session; a subagent's thread stays open for follow-ups until archived.

Depth and concurrency limits

Know the limits before you design a deep or wide tree. A design that spawns past these limits will hit them at run time. The limits differ between the Agent SDK, Claude Code and Managed Agents.

Delegation limits by environment

Limit	Default	What happens
Agent SDK depth	**3** layers below the main agent	Set with `CLAUDE_CODE_MAX_SUBAGENT_SPAWN_DEPTH`
Claude Code concurrency	**20** subagents at once	Spawn refused with "Concurrent subagent limit reached"
Managed Agents depth	Not a default: only one level deep	Listing an agent with multiagent set returns a **400** error

Scale needs a different tool

Subagents suit a few delegated tasks per turn. Coordinating dozens or hundreds of agents? Reach for the Workflow tool instead. It runs orchestration as a script outside the conversation context. Note the trade-off: you cannot send follow-ups to a workflow run's threads.

Plan for failed subagents

Tell the orchestrator what to do when an agent fails. A system prompt can say one failure must not fail the run. Failed items get listed as not covered, and the final report names them. That makes gaps in the synthesis visible.

> **EXAM TRAP**
> An API error such as a rate limit that ends a subagent early never arrives as its result.

Spot unfinished results

A subagent stopped at `maxTurns` returns a partial result. Claude Code marks that output as partial, so the orchestrator knows the work is unfinished.

Make delegation observable

Subagent spans nest under the parent's `claude_code.tool` span. So the whole delegation chain shows as one trace. In the Console session viewer, the timeline minimap gives each thread its own lane. Set `CLAUDE_CODE_OTEL_DIAG_STDERR=1` too, because export errors fail silently by default.

Key takeaways

- Pass every path, error and decision a subagent needs in its prompt string.
- Only a subagent's final message returns to the parent.
- Use the Workflow tool, not subagents, for dozens to hundreds of agents.
- Design the run so one failed agent is reported as a gap, not a failed run.

Sources: Subagents in the SDK - Claude Code Docs (opens in a new tab) · Multiagent orchestration - Claude Platform Docs (opens in a new tab) · Observability with OpenTelemetry - Claude Code Docs (opens in a new tab) · Inspect sessions and track usage - Claude Platform Docs (opens in a new tab)

Value-Based Architectural Decisions

Start here

THINK FIRST

Under a fixed cost or SLA ceiling, how do you choose the Claude architecture that delivers the most measurable business value?

Why it matters: Questions give a cost or SLA ceiling and competing proposals, and ask which option best serves the stated business value pillars.

Value is measured per completed task, not per token. Compare options on your hardest tenth of tasks, where failures drive the bill. Then turn on prompt caching before any other cost lever.

Key terms

Term	Meaning
Value pillars	Stated business goals: efficiency, transformation, productivity, solution cost and performance SLAs.
Cost per completed task	Spend divided by tasks actually finished, the fair basis for comparing options.
Effort	Soft guidance on how much output Claude spends on thinking; not a guaranteed token count.
max_tokens	A hard cap on one request's total output, thinking and response combined.

Measure value correctly

Compare options on cost per completed task. You pay for finished work, so cost per token misleads. Test on the hardest **10%** of tasks: each failure bills its tokens, a retry and any downstream damage.

EXAM TRAP

The model with the lowest token price is not automatically the cheapest. Its failures on hard tasks add retries and downstream costs.

Pick the model tier

Start with Opus 5.5 at its default medium effort. Match the tier to the workload: high-volume checkable work, everyday coding, or complex multi-step reasoning.

Model choices matched to workload

Option	Best fit	Cost note
Opus **5.5**, default medium effort	Most agent workloads	Recommended default

Option	Best fit	Cost note
Haiku **5.5**	High-volume, latency-sensitive work with checkable outputs	Outputs must be checkable
Sonnet in Claude Code	Most coding tasks	Costs less than Opus
Opus in Claude Code	Complex architecture or multi-step reasoning	Reserve for the hardest reasoning

Free cost wins first

Turn on prompt caching before any other lever. By default each agentic turn resends the whole growing conversation. That includes the system prompt, tool definitions and every prior turn.

Trade latency for cost

The Batch API halves cost if results can wait. It takes **50%** off every token, cached ones too, with results due within **24** hours. It does not combine with fast mode, message threads or Managed Agents sessions.

EXAM TRAP

Message threads do not lower cost. They lower time to first token, but the model still reads the whole conversation.

Effort and caps

Low effort with high-effort retries can cut cost about 40%. When outputs can be checked, start every task at low effort and retry only the failures at high, keeping the same pass rate. Remember `max_tokens` caps only one request, so it does not bound a whole tool-use turn.

When delegation pays

Single-chain work favors one model over delegation. For one dependent chain, or work fitting in a single context, the orchestrator's model alone at lower effort won every measured case. Added tools cost too: the advisor tool adds about **1,000** prompt tokens per request.

EXAM TRAP

The advisor tool is not free when unused. Its definition adds about **1,000** prompt tokens to every request.

Track outcomes against the ceiling

Stream per-user cost metrics to your own stack. OpenTelemetry export runs on every setup. It is the only option that sends each user's token and cost figures to your own monitoring in near real time. You can then watch the ceiling continuously.

Key takeaways

- Compare options on cost per completed task, using your hardest tasks.
- Start most agent workloads on Opus **5.5** at its default medium effort.
- Turn on prompt caching before any other cost lever.
- With checkable outputs, low effort plus high-effort retries saves about **40%**.
- Use the Batch API for **50%** off when results can wait **24** hours.

Sources: Optimizing for cost and intelligence - Claude Platform Docs (opens in a new tab) · Claude Certification Program Exam guide Claude Certified Architect – Professional Exam Guide (opens in a new tab) · Steering thinking - Claude Platform Docs (opens in a new tab) · Manage costs effectively - Claude Code Docs (opens in a new tab)

Deployment Surface and Agent Runtime Selection

Start here

THINK FIRST

How do residency, compliance, feature needs and who runs the infrastructure steer your choice of platform and agent runtime?

Why it matters: Questions state residency, compliance, feature or ownership constraints and ask which deployment surface and agent runtime fit.

Each cloud surface keeps data and features within its own limits, so constraints narrow the platform first. The runtime choice then decides who hosts the agent loop, the sessions and the sandboxes. Every step toward self-hosting adds custody and adds operational work.

Key terms

Term	Meaning
Regional endpoint	An endpoint that keeps inference in one region for data residency, usually at a price premium.
Self-hosted sandboxes	You host only the tool sandboxes; Anthropic still runs the service and stores transcripts.
Self-hosted Managed Agents	You host the service too: the API, the agent loop and session storage.
Zero data retention	A ZDR arrangement; hosted Managed Agents falls outside it because transcripts are stored.

Residency options per surface

Regional endpoints send requests only to the region you name. On Bedrock and Google Cloud (Google managed), regional endpoints cost **10%** more than global ones. Other surfaces keep inference local with a per-request setting or a Data Zone scope.

How each surface keeps inference in one place

Surface	Residency option	Trade-off
Amazon Bedrock	Regional endpoint in one AWS region	**10%** above global pricing
Google Cloud (Google managed)	Regional endpoint for strict mandates	**10%** above global; needed for provisioned throughput
Google Cloud (Anthropic managed)	Set `inference_geo` to `us` per request	Request location alone does not fix inference
Microsoft Foundry on Azure	Data Zone scope keeps inference in the US	Only for deployments hosted on Azure

EXAM TRAP

Sending requests to a US location on Google Cloud (Anthropic managed) does not by itself keep inference in the US. Each request must set `inference_geo`.

Feature gaps on cloud surfaces

Not every Claude API feature exists on every cloud. Bedrock lacks Agent Skills, the MCP connector, programmatic tool calling, batches and Managed Agents. On Foundry hosted on Azure, code execution, Skills, programmatic tool calling and the Files API return a **400** error. So check required features before picking a surface.

EXAM TRAP

A design that depends on Agent Skills or programmatic tool calling cannot simply move to Bedrock or Azure-hosted Foundry. Those calls are unsupported there.

Custody and access per cloud

Each cloud draws its own line on data and access. Bedrock runs on AWS infrastructure that Anthropic staff cannot access. On Azure-hosted Foundry, prompts and completions stay on Azure, but operational data sits on Anthropic systems. Google Cloud (Anthropic managed) needs Google credentials, not API keys. On Google Cloud (Google managed), Google sets the model lifecycle dates.

Three levels of agent hosting

Self-hosting moves custody and work to you, step by step. If only the tools must run on your infrastructure, use self-hosted sandboxes.

Hosted, self-hosted sandboxes and self-hosted Managed Agents

Feature	Hosted	Self-hosted sandboxes	Self-hosted Managed Agents
Who runs the service	Anthropic	Anthropic	You
Where tools run	Anthropic	Your sandboxes	Your sandboxes

Feature	Hosted	Self-hosted sandboxes	Self-hosted Managed Agents
Transcript storage	Anthropic, until you delete	Anthropic	Your database
Your operating work	None	Sandboxes only	Server, database, sandboxes, upgrades

EXAM TRAP

Self-hosted sandboxes do not move transcripts to you. Only self-hosted Managed Agents does, so pick it when transcript custody is the constraint.

Running your own server

You run and secure the server, its database and your sandboxes. It serves only `self_hosted` environments; asking for cloud returns **400**. Without a database, a restart erases agents and sessions. Sessions never expire by age. Moving over copies nothing, not even history. Each deployment should serve one tenant at one trust level.

Network and inference controls

Model calls can go to a cloud or your own gateway. MCP hosts must appear in `mcp.allowed_hostnames`, which starts empty and allows no wildcards. Anthropic gets a connection only when the Claude API is the provider. Then calls carry ID headers that cannot be switched off.

Key takeaways

- Bedrock does not support Agent Skills, the MCP connector or Managed Agents.
- Choose self-hosted sandboxes when only tools must run on your infrastructure.
- Only self-hosted Managed Agents moves transcript storage to you, along with all operations.

Sources: Claude in Amazon Bedrock (bedrock-mantle) - Claude Platform Docs (opens in a new tab) · Self-hosted Claude Managed Agents - Claude Platform Docs (opens in a new tab) · Claude Models on Google Cloud (Google managed) - Claude Platform Docs (opens in a new tab) · Data and access in self-hosted Claude Managed Agents - Claude Platform Docs (opens in a new tab) · Claude Models on Google Cloud (Anthropic managed) - Claude Platform Docs (opens in a new tab) · Claude in Microsoft Foundry - Claude Platform Docs (opens in a new tab) · Differences from the hosted service - Claude Platform Docs (opens in a new tab)

Domain 2 | 13% of the exam | 19 min read

Claude Models, Prompting, and Context Engineering

Overview

This domain is about 13% of the exam, roughly 9 of its 70 questions.

Every point below comes from Anthropic's official documentation; each section lists its sources.

Model Tier Selection

Start here

THINK FIRST

Given a workload's latency, cost and quality limits, how do you pick the Claude model tier and explain why the others fail?

Why it matters: Questions give a workload profile and ask which model tier meets every binding constraint, and why each rejected tier does not.

The tiers run from Haiku through Sonnet and Opus to Fable, rising in cost and capability. Most workloads start on Opus. Tune effort before switching, and judge cost per completed task, not per token.

Key terms

Term	Meaning
Model tier	A model's place on the cost and capability ladder: Haiku, Sonnet, Opus or Fable.
Effort	A parameter that gives up some intelligence for lower latency and cost, within one model.
Cost per completed task	What a finished task really costs, counting failures and retries, rather than the token price.
Frontier model	The most capable and most costly current model, Claude Fable **5.2**.

The tier ladder

Claude Opus 5.5 is the usual starting tier. From there, each tier trades cost against capability, so each one has a typical fit.

Current Claude tiers, from lowest to highest cost and capability

Tier	Choose it for	Example workloads
Claude Haiku **5.5**	The lowest latency and price	Real-time apps, high-volume processing, subagent tasks
Claude Sonnet **5.5**	Speed and capability for everyday work	Code generation, data analysis, agentic tool use
Claude Opus **5.5**	The usual starting point	Most workloads
Claude Fable **5.2**	The highest available capability	Hours-long agent sessions, multistep deep research

Tune effort before switching

Effort is often a better lever than changing models. It trades intelligence for latency and cost inside one model. On the Messages API, most models default to high effort; Opus **5.5** and Haiku **5.5** default to medium.

EXAM TRAP

Opus **5.5** tokens cost twice as much as Sonnet **5.5** tokens. Yet at low effort, Opus solved about as many SWE-bench Pro subset tasks for only about **7%** more per solved task.

Compare cost per completed task

You pay for completed tasks, not tokens. A failed task is still billed for its tokens. Then you pay for the retry and whatever the failure costs downstream. So a cheaper model is only cheaper if it rarely fails.

When Haiku is enough

Haiku fits high-volume, latency-sensitive work with checkable outputs. On GPQA Diamond, Haiku **5.5** scored **85%** against **91%** for Opus and Sonnet. Its cost per correct answer was about a twentieth of Opus's.

When to move up to Fable

Move to Fable when Opus at top effort still falls short. If evals at xhigh or max effort still miss on demanding reasoning or long-horizon agent work, switch to Claude Fable **5.2**. Claude Mythos **5.2** offers the same capabilities, but only to verified organizations.

EXAM TRAP

The top tier does not always pay off. On DeepResearch Bench II, Fable **5.2** cost **4.3** times as much as Sonnet **5.5** with no measurable score gain.

Key takeaways

- Tiers rise in cost and capability: Haiku, Sonnet, Opus, then Fable.
- Start most workloads on Opus **5.5** and tune effort before changing tiers.
- Judge models on cost per completed task, since failures bill tokens, retries and downstream costs.
- Escalate to Fable when Opus at xhigh or max effort still falls short.

Sources: Optimizing for cost and intelligence - Claude Platform Docs (opens in a new tab) · Choosing the right model - Claude Platform Docs (opens in a new tab)

System Prompt Diagnosis and Revision

Start here

THINK FIRST

When a system prompt gives vague or inconsistent output, how do you find what it leaves out and fix it without over-constraining Claude?

Why it matters: Questions show a system prompt and its weak output, then ask which revision fixes scope, format or behavior best.

Do not rely on Claude to infer extra work from a vague prompt; ask for it explicitly. Say why a behavior matters to get more targeted responses. Avoid aggressive wording, which more responsive models might overtrigger on.

Key terms

Term	Meaning
Underspecification	A prompt so vague that Claude must infer what you want instead of being told.
Overtriggering	Claude uses a tool or behavior too often because the prompt pushes it too hard.
Positive instruction	Telling Claude what to do, such as writing flowing prose, instead of what to avoid.
Preset with append	An Agent SDK option that keeps the `claude_code` preset and adds your instructions after it.

Run the colleague test

If a new colleague would be confused, so will Claude. Show the prompt to someone with minimal context on the task. Their confusion is a preview of Claude's.

State scope and motivation

Ask explicitly for anything beyond the bare task. Claude will not reliably infer extra work from a vague prompt. Saying why a behavior matters gives Claude your goal, so responses become more targeted. A one-sentence role also focuses behavior and tone.

EXAM TRAP

Expecting Claude to go beyond the task without being asked is the classic gap. Request the extra behavior explicitly.

Structure the prompt

Separate content types and order matters. Wrap instructions, context and input in their own XML tags to avoid misreading. Use numbered steps when order or completeness matters. For inputs of **20k+** tokens, put long documents at the top.

Steering output format

Say what format you want, not what to avoid. Ask for smoothly flowing prose paragraphs rather than banning markdown. The prompt's own style also carries over, so a prompt without markdown tends to get less markdown back.

EXAM TRAP

A long anti-markdown block is not always helpful. Claude Fable **5.1** already formats less, so such a block can suppress structure the content needs.

Avoid over-constraining

Aggressive or blanket rules cause overtriggering. Revisions that fix one problem can create another. Pick the one that targets the need without forcing the behavior everywhere.

Over-constrained wording and better revisions

Problem	Over-constrained wording	Better revision
Tool overuse	CRITICAL: You MUST use this tool when...	Use this tool when...
Blanket default	If in doubt, use [tool]	Use [tool] when it helps you understand the problem

EXAM TRAP

Aggressive wording like CRITICAL or MUST can backfire. Models that are more responsive to the system prompt might overtrigger on it.

Agent SDK system prompt choices

Leaving systemPrompt unset drops the preset's safety rules. The SDK then uses a minimal prompt that covers tool calling only, unlike `claude -p`. The `claude_code` preset with append adds your rules and removes nothing, the lowest-risk option. A custom string sends only what you write.

EXAM TRAP

An unset Agent SDK system prompt is not the full Claude Code prompt. It omits the preset's security and safety instructions.

Key takeaways

- If a colleague with little context would be confused by the prompt, Claude will be too.
- Ask explicitly for anything beyond the task, and say why it matters.
- Tell Claude what format to use instead of what to avoid.
- Replace aggressive or blanket tool rules with normal, targeted wording.
- Use the `claude_code` preset with append for the lowest-risk SDK customization.

Sources: Prompting best practices - Claude Platform Docs (opens in a new tab) · Modifying system prompts - Claude Code Docs (opens in a new tab)

Prompting Technique Selection

Start here

THINK FIRST

Given a failure pattern and a task's complexity, which prompting technique fixes it without paying for tokens you do not need?

Why it matters: Questions show a failing output and ask which technique (examples, reasoning, thinking or structured output) fixes it at the least token cost.

Match the technique to the failure. Examples fix format and tone, thinking or prompted reasoning fixes weak logic, and structured outputs fix schema errors. Then size the reasoning to the task, since extra thinking costs tokens and latency.

Key terms

Term	Meaning
Few-shot prompting	Giving Claude a few well-crafted examples in the prompt; also called multishot prompting.
Manual chain-of-thought	Asking Claude to reason step by step before answering when thinking is turned off.
Effort	Soft guidance on how much of the output Claude spends on thinking; it sets no exact token count.
Structured outputs	A mode that uses constrained decoding so every response matches a JSON schema.

Pick the technique by failure

Each technique targets a different kind of failure. Start from what went wrong, not from a favorite technique. A wrong format calls for examples or a schema. Weak reasoning calls for thinking or a reasoning prompt.

Observed failure and the technique that addresses it

Failure	Technique	Why it works
Inconsistent format, tone or structure	Few-shot examples	Examples steer format, tone and structure reliably
Invalid JSON or missing required fields	Structured outputs	Constrained decoding guarantees schema-compliant output
Weak reasoning with thinking off	Manual chain-of-thought	Claude reasons before answering, final answer in tags
Weak reasoning with thinking on	General instruction such as "think thoroughly"	Often beats a hand-written step-by-step plan

Write examples that generalize

Few-shot examples must be diverse. Cover edge cases and vary the examples enough that Claude does not copy unintended patterns. Examples also work with thinking: worked examples guide how Claude tackles similar problems inside its thinking blocks.

EXAM TRAP

Examples are not only for runs without thinking. With thinking on, worked examples still shape how Claude reasons.

Prompted reasoning versus thinking

Prefer general instructions over prescriptive steps. Claude's own reasoning often goes beyond what a person would script. When thinking is off, ask it to work through the problem first. Have it wrap the final answer in answer tags so code can pull it out.

EXAM TRAP

A detailed hand-written plan is not the best reasoning aid. A general prompt like "think thoroughly" often produces better reasoning.

Size reasoning to the task

Give simple tasks a budget near 1,024 tokens. With manual extended thinking, complex tasks start at **16,000** tokens or more. Adjust from there for latency and quality. Want Claude to think less often? Turn the effort level down before you try prompt-based steering.

Effort versus max_tokens

`max_tokens` **is a hard cap; effort is soft guidance.** The cap covers thinking and response text together, and Claude never goes past it. Effort only shapes how much output goes to thinking. Most models default to high effort. Claude Opus **5.5** defaults to medium, and so does Claude Haiku **5.5**.

EXAM TRAP

Truncated at `max_tokens`? Raising the cap is right only when those answers needed the reasoning. If Claude over-thought them, reduce effort instead.

Structured output costs and caveats

Structured outputs add a small token cost. Claude receives an extra system prompt, so input tokens rise slightly. Changing `output_config.format` invalidates the prompt cache for that thread. Enum capitalization is not guaranteed, so compare enum values case-insensitively.

EXAM TRAP

Do not add a schema property that asks for step-by-step reasoning. It can cause a `reasoning_extraction` refusal, so request a brief explanation instead.

Trim instructions that waste tokens

On Claude Opus 5, remove old verification instructions. This model checks its own work well without being told. Verification instructions carried over from prompts for earlier models cause over-verification, which adds tokens and latency. Delete them rather than rewording them.

Key takeaways

- Use a few diverse, well-crafted examples to steer format, tone and structure.
- With thinking off, fall back to manual chain-of-thought and extract the answer from tags.
- Structured outputs remove schema-violation retries but add a slightly larger input prompt.
- Start simple tasks near the **1,024**-token thinking minimum and complex ones at **16,000** or more.

Sources: Prompting best practices - Claude Platform Docs (opens in a new tab) · Structured outputs - Claude Platform Docs (opens in a new tab) · Extended thinking - Claude Platform Docs (opens in a new tab) · Steering thinking - Claude Platform Docs (opens in a new tab)

Context Composition and Token Economics

Start here

THINK FIRST

Given a token budget and a call volume, where can you cut token use without losing information the task needs?

Why it matters: Questions give a context layout, a call volume and a budget and ask where token use or cost can safely drop.

Every part of a request fills the context window, and a fuller window can hurt accuracy. Caching cuts the price of a stable prefix but not its size. So you cut size by keeping unneeded material out, and cut cost by caching what repeats.

Key terms

Term	Meaning
Cache prefix	Tools, then system, then messages, up to the block marked with `cache_control`.
Cache breakpoint	The block marked with `cache_control`; it should end the static part of the prompt.
Cache lifetime	How long a cached prefix lasts: **5** minutes by default, refreshed free on each use.
Context rot	Accuracy and recall degrading as the token count grows.

Know what fills the window

Every part of the request counts toward context. That includes the system prompt, tool definitions and every message. Tool results, images and documents count as well. Extra tokens are not free in quality either, because recall degrades as context grows.

Size cuts versus cost cuts

Some techniques shrink context; caching only lowers the bill. Decide whether the budget limits tokens or spend, then pick accordingly.

What each technique reduces

Technique	Reduces tokens in context	Other effect
Programmatic tool calling	Yes: intermediate results stay out of history	One script replaces five tool roundtrips
Tool search	Yes: much lower baseline usage	Costs one extra turn of latency per lookup
Prompt caching	No: cached prefixes still count	Lowers what you pay for cached tokens

EXAM TRAP

Caching a long prompt does not help a context window that is too full. Cached prefixes still occupy the window.

Place the breakpoint well

Put the cache breakpoint at the end of the static prefix. The cache covers tools, system and messages in that order, up to the marked block. A varying suffix, such as a timestamp, must sit after the breakpoint. So mark the last block that stays identical across the requests meant to share a cache.

EXAM TRAP

Editing a tool's name, description or parameters is not a small change. It invalidates the whole cache, including system and messages.

Match lifetime to call volume

The default cache lasts 5 minutes and refreshes free on use. So a prompt sent more often than every **5** minutes should stay on the default lifetime. Each call keeps the cache warm at no extra charge.

Short prompts are not cached

Below the minimum length, caching silently does nothing. The request still succeeds, and no error is returned. Check the cache fields in the usage to confirm hits rather than assuming them.

Read the usage numbers

Total input is cache reads plus cache writes plus uncached input. Sum `cache_read_input_tokens`, `cache_creation_input_tokens` and `input_tokens` for the full count. Token counting gives an estimate without caching logic. Caching happens only when a message is actually created.

Key takeaways

- Prompt caching lowers cost, but cached tokens still count toward the context window.
- End the cached prefix before any varying content such as timestamps.
- Changing any tool definition invalidates the entire prompt cache.
- Programmatic tool calling keeps intermediate tool results out of the history.

Sources: Context windows - Claude Platform Docs (opens in a new tab) · Prompt caching - Claude Platform Docs (opens in a new tab) · Manage tool context - Claude Platform Docs (opens in a new tab) · Token counting - Claude Platform Docs (opens in a new tab)

Cache Breakpoints and Reusable Packaging

Start here

THINK FIRST

How should you order and package a repeated prompt so the most content is reused from cache across calls?

Why it matters: Questions describe a repeated prompt sequence and call pattern and ask where breakpoints and packaging maximize cache reuse.

The cache matches an exact prefix, built from tools, then system, then messages. So stable content goes first and anything that varies goes last, after the breakpoint. Skills and deferred tools keep extra material out of that prefix until it is needed.

Key terms

Term	Meaning
Cache breakpoint	A block marked with `cache_control`; the cached prefix ends there.
Cache prefix order	Tools first, then system, then messages, each level building on the one before.
Lookback	The backward search, up to **20** positions from a breakpoint, for an earlier cache entry.
Progressive disclosure	Skills load only their name and description until triggered, so they add little context.
Automatic caching	Caching that places its own cache breakpoint; it can be combined with explicit breakpoints.

Stable first, dynamic last

Put static content at the start and mark where it ends. Tool definitions, system instructions, context and examples belong first. Then mark the end of that reusable part with `cache_control`. A hit needs every segment up to that block to be identical.

Never mark the varying block

The breakpoint goes at the end of the static prefix. A timestamp or incoming message changes on every call. Cache writes happen only at the breakpoint. So a breakpoint on a changing block never writes a reusable entry, and the lookback cannot save the stable part behind it.

> **EXAM TRAP**
> The lookback does not rescue a breakpoint placed on the incoming message. Writes happen only at the breakpoint, so nothing stable gets cached.

Breakpoint budget and lookback

One request can set up to 4 cache breakpoints. Use them for sections that change at different rates. Automatic caching sets a breakpoint of its own. Combined with explicit breakpoints, that automatic breakpoint takes one of the **4** slots. Each breakpoint looks back at most **20** positions for an earlier entry.

Caching tool definitions

Mark the last tool to cache the whole tools prefix. Keep tool definitions stable across calls.

How tool changes affect the cache

Action	Effect on cache	Advice
`cache_control` on the last tool	Caches all tool definitions	Covers the first tool through the marked one

Action	Effect on cache	Advice
Edit a tool name or description	Invalidates the entire cache	Avoid edits between calls
Discover a deferred tool	Cache preserved; appended inline	None needed; the prefix stays untouched

Package reusable know-how as Skills

A Skill costs only its name and description until triggered. So many Skills can be installed without a context penalty.

Timing and silent misses

A cache entry exists only once the first response starts. Parallel requests should wait for that first response to get hits. The cache lasts **5** minutes by default and refreshes free when used. A prompt below the minimum length is not cached, and no error says so.

EXAM TRAP

Firing ten identical requests at once does not give nine cache hits. The entry only appears after the first response begins.

Key takeaways

- Order the prompt from stable to dynamic and end the cached prefix before varying content.
- Cache hits need prompt segments that are exactly identical up to the breakpoint.
- A request can hold up to **4** breakpoints, and automatic caching uses one.
- Any change to a tool definition invalidates the whole cache.

Sources: Prompt caching - Claude Platform Docs (opens in a new tab) · Agent Skills - Claude Platform Docs (opens in a new tab) · Tool use with prompt caching - Claude Platform Docs (opens in a new tab)

Reasoning Effort and Context Lifecycle

Start here

THINK FIRST

For an agent that runs for hours, how do you set reasoning effort and manage its context so it stays accurate without wasting tokens?

Why it matters: Questions describe a long-running agent with latency and accuracy needs and ask which effort level and context strategy fit it.

Effort applies to every output token, so it shapes cost with or without thinking; on Claude Opus **5.5** it is the primary control. Over a long session, compaction and the memory tool keep the working context small while saving what must persist.

Key terms

Term	Meaning
Effort	A setting that scales every output token: text, tool calls and thinking.
Context editing	Clearing old tool uses and results from the prompt once it passes a set trigger.
Compaction	Summarizing the conversation so the active context stays small, with no bookkeeping in your client.
Memory tool	A client-side tool: Claude requests memory operations and your app runs them on storage you control.

Effort shapes every token

Effort applies to all output, with or without thinking. It covers text, tool calls and thinking alike. Turning effort down also gives you fewer, shorter tool calls. Most models default to high; Claude Opus **5.5** and Claude Haiku **5.5** default to medium.

Choosing an effort level

Match the level to the task's length and depth. Low suits simple, fast work such as subagents. The xhigh level targets long agentic and coding tasks. On Claude Opus **5.5**, effort can change mid-conversation through a per-message `output_config` that keeps the prompt cache.

Effort levels for agent workloads

Level	Fits	Trade-off
low	Simpler tasks needing speed and low cost, such as subagents	Significant token savings, some capability loss
xhigh	Agentic and coding tasks over **30** minutes	Token budgets in the millions; set a large `max_tokens`

EXAM TRAP

On Claude Opus **5.5**, you cannot save tokens by disabling thinking. Adaptive thinking is always on, and a disabled-thinking request returns a **400** error.

Leave room at high effort

Give xhigh and max effort a generous `max_tokens`. `max_tokens` is a hard limit on thinking plus response text together. A small cap can cut off a long, deep answer before it finishes.

Clearing old tool results

Context editing clears tool results past a trigger. The `clear_tool_uses_20250919` strategy triggers at **100,000** input tokens by default. It then keeps the **3** most recent tool use and result pairs, dropping the oldest first. Use `exclude_tools` for tools whose results must never be cleared.

EXAM TRAP

Clearing is not free: it invalidates cached prompt prefixes. Set `clear_at_least` so each clearing removes enough tokens to justify the cache loss.

Pairing editing with memory

Claude gets a warning before context is cleared. When context editing runs with the memory tool, the warning arrives near the clearing threshold. Claude can then save key facts to memory files. Your application performs each memory operation against its own storage.

Compaction plus memory

Long-running agents can use compaction and memory together. Compaction keeps the active context small. Memory holds the information that must survive summarization. Where on-demand compaction is available, prefer it over threshold compaction.

Key takeaways

- Effort affects every output token, so it changes cost even when thinking is off.
- Use low effort for subagents and xhigh for agent tasks longer than **30** minutes.
- Context editing triggers at **100,000** input tokens and keeps **3** tool pairs by default.
- Combine compaction with the memory tool so key facts survive summarization.

Sources: Effort - Claude Platform Docs (opens in a new tab) · Memory tool - Claude Platform Docs (opens in a new tab) · Context editing - Claude Platform Docs (opens in a new tab) · Compaction overview - Claude Platform Docs (opens in a new tab)

Domain 3 | 21% of the exam | 26 min read

Integration

Overview

This domain is about 21% of the exam, roughly 15 of its 70 questions.

Every point below comes from Anthropic's official documentation; each section lists its sources.

Least-Privilege Tool Configuration

Start here

THINK FIRST

When you set up an agent or subagent, how do you decide which tools it truly needs and which ones to take away?

Why it matters: Questions show an agent or subagent tool setup and ask which tools to remove, scope down or load in a leaner way.

Give each agent only the tools its job needs. Extra tools add risk, fill the context window and make tool choice less accurate. Remove or restrict the rest, and keep bulky intermediate results out of the conversation.

Key terms

Term	Meaning
Tool allowlist	Tools start disabled by default and only the ones you name are enabled.
Tool denylist	All tools stay enabled by default and you explicitly disable the unwanted ones.
`disallowedTools`	A subagent field that removes named tools, or whole MCP servers by pattern, from its tool set.
Programmatic tool calling	Claude runs one sandboxed script that calls several tools, so intermediate results stay out of history.

Why fewer tools is better

Every loaded tool costs context and accuracy. About **50** tools can use **10-20K** tokens of context, leaving less room for real work. Tool selection accuracy also drops once more than **30-50** tools load at once. Fewer tools also mean less risk of unintended actions.

Scoping a subagent's tools

List tools and the subagent gets only those. Match the list to the role, as the table shows. Turning on memory also adds `Read`, `Write` and `Edit`, so check it before calling a subagent read-only.

Subagent roles and the tools that fit them

Role	Tools	What it can do
Code reviewer	`Read`, `Grep`, `Glob`	Read and search, but not edit files
Doc reviewer	`Read`, `Grep`	Analyze files but never modify them
Code modification	`Read`, `Edit`, `Write`, `Grep`, `Glob`	Full read/write access, no command execution

EXAM TRAP

Leaving out the `tools` field does not give a subagent zero tools. It gets every tool available to subagents.

Removing tools and servers

`disallowedTools` **strips tools from a subagent's set.** It accepts tool names and MCP server patterns. `mcp__server__*` removes every tool from that one server, and `mcp__*` removes every MCP tool from any server. A tool listed in `tools` is allowed, but listing it does not pre-approve it.

EXAM TRAP

Adding a tool to an `AgentDefinition`'s `tools` array is not the same as approving it. Listing a tool does not pre-approve it.

MCP toolsets: allow or deny

Every MCP toolset tool is enabled by default. For an allowlist, set `enabled: false` in `default_config`, then enable chosen tools in `configs`. For a denylist, keep the default and disable unwanted tools. Denylisting write or destructive tools suits read-only assistants or a human check before state changes.

Example: An MCP toolset allowlist: every tool off by default, one tool turned back on.

```
{
  "default_config": { "enabled": false },
  "configs": {
    "my_tool": { "enabled": true }
  }
}
```

- `"default_config": { "enabled": false }`: the set-level default turns every tool off
- `"my_tool": { "enabled": true }`: a tool-specific setting overrides the set-level default

Keeping context lean

For a small tool set, loading everything upfront is typically faster. With fewer than about **10** tools that fit comfortably, loading them all upfront is typically faster than tool search. Beyond that, trim what reaches the context with the patterns below.

Patterns that keep tools and results out of the context

Pattern	What it does	What stays out
`defer_loading: true`	Holds back the tool's description at first; used with the tool search tool	Tool definitions
Programmatic tool calling	One sandboxed script calls several tools in one run	Intermediate results
Context editing	Removes old `tool_result` blocks once they have served their purpose	Stale tool results

EXAM TRAP

Tool search is not always the faster choice. With fewer than about **10** tools that fit comfortably, loading them all upfront is typically faster.

Key takeaways

- Omitting `tools` hands a subagent every available tool, so list only what its job needs.
- Turning on memory adds `Read`, `Write` and `Edit` to a subagent's tools.
- Use `disallowedTools` with `mcp__server__*` or `mcp__*` to strip whole MCP servers.
- Tool selection accuracy degrades past **30-50** loaded tools, so trim large toolsets.

Sources: Subagents in the SDK - Claude Code Docs (opens in a new tab) · Scale to many tools with tool search - Claude Code Docs (opens in a new tab) · Manage tool context - Claude Platform Docs (opens in a new tab) · MCP connector - Claude Platform Docs (opens in a new tab)

Authentication and Authorization Posture

Start here

THINK FIRST

Looking at a Claude integration design, where can a credential leak or be misused, and which control closes each gap?

Why it matters: Questions describe an integration and ask which credential exposure or privilege risk exists and which mitigation addresses it.

Assume any credential can leak, then limit who owns it, where it works, how long it lasts and what it can do. No single control replaces the others, so layer them.

Key terms

Term	Meaning
Service account	An organization identity for a workload, so its key does not depend on one person.
Identity-backed key	A personal or service account key that stops working when its owner leaves the organization.
IP allowlisting	Limiting the addresses from which an API key can be used.
App Attest access token	A token scoped to your workspace that expires after **1** hour and authorizes only Messages API calls.

Keep keys out of browsers

Call the Claude API from your back end, never from page code. Any visitor to a webpage can read a key embedded in its code. A browser-side call design is credential exposure by default.

EXAM TRAP

Hiding a key in front-end code is not protection. Any visitor can read it, so the call must move to your back end.

Who owns the key

Use identity-backed keys, and service accounts for automation. Personal and service account keys stop working when their identity is removed, so they never outlive their owner. A shared personal key stands in for one person and fails once they leave.

Limit where and how long

Expiration limits time; IP allowlisting limits place. In the Console, choose expiration at creation: **3** hours, **1**, **7** or **30** days, custom or Never. It cannot be changed later. Store keys in a secrets manager and rotate them.

Responding to a leak

Disable is reversible; delete is permanent. Disable a suspect key to stop it at once; re-enabling returns it to active. Delete archives the key for good. Either way, act on any key you suspect has leaked.

Scoped Claude tokens

Each token has a fixed lifetime and a narrow scope. An App Attest access token lasts **1** hour; a `claude setup-token` token lasts **1** year. The setup-token is never saved, so you set it as `CLAUDE_CODE_OAUTH_TOKEN` yourself.

Scoped Claude tokens compared

Feature	App Attest access token	claude setup-token OAuth token
Lifetime	**1** hour	**1** year
What it authorizes	Only Messages API calls	Only model requests
Scope	Your workspace	Your Claude subscription on a Pro, Max, Ultra, Team or Enterprise plan

Federation trust chain

Workload Identity Federation is only as strong as its chain. A long-lived secret earlier in the chain, such as a static cloud credential that can mint identity tokens, can still undermine it. Review the identity provider's configuration too.

EXAM TRAP

Adopting federation does not by itself guarantee end-to-end security. A static credential upstream can still mint tokens.

Agents and sandboxed code

Code tricked into running can misuse any credential it reaches. Use vault credentials, limited networking and approvals for risky actions. MCP credentials never enter the sandbox, and environment variable credentials show only a placeholder. Restrict what each key may do where it was issued.

EXAM TRAP

Restricting allowed hosts does not stop an agent using a credential on a host you allowed. These settings do not limit a key's permissions either, so limit those at the service that issued it.

Claude Code credential storage

Know where local logins live and what is not enforced. Linux stores credentials in `~/.claude/.credentials.json` with mode `0600`; macOS uses the Keychain. Claude Code does not check a saved Console credential's organization at startup, so `forceLoginOrgUUID` is no guarantee.

Key takeaways

- Never put an API key in browser code; call Claude from your back end.
- Give CI and production services a service account, not a shared personal key.
- Set expiration and IP allowlisting to limit how long and where a leaked key works.
- Agent network limits do not limit key permissions; set those at the issuing service.
- Console key expiration is fixed at creation and cannot be changed later.

Sources: Authentication - Claude Platform Docs (opens in a new tab) · Security risks and sandbox boundaries - Claude Platform Docs (opens in a new tab) · Authentication - Claude Code Docs (opens in a new tab)

Relieving the Binding Performance Constraint

Start here

THINK FIRST

When a Claude system is too slow or not accurate enough, which configuration change fixes it while costing least elsewhere?

Why it matters: Questions name the binding constraint, accuracy, latency or cost, and ask which change relieves it with the least harm to the others.

Turn on prompt caching before any other lever, since every agentic turn resends the whole conversation. Then tune effort before changing the model tier: lower it when cost is too high, restore it when quality falls short.

Key terms

Term	Meaning
Cache pre-warming	Loading the system prompt or tool definitions into the cache before a user sends a real request.
Effort	A speed and cost control, high by default on most models; on Haiku **5.5**, low is the cheapest and fastest level.
Prompt caching	Letting the API resume from a stored prompt prefix to save time and money.
TTFT	Time to first token: from sending the prompt until the first response token appears.

Start with caching

Turn on prompt caching before any other lever. In an agentic task, every turn resends the whole growing conversation, including the system prompt and tools. Caching that prefix cuts processing time and cost.

EXAM TRAP

Caching is not only a cost lever. Resuming from a cached prefix also reduces processing time, so it helps latency too.

Cache pricing and lifetime

Writes cost a little more; reads cost far less. With the **5**-minute lifetime, a cache write costs **25%** more than base input, while a read costs **10%** of it. Each use refreshes the lifetime at no extra cost, so repeated prefixes pay off quickly.

Levers and their trade-offs

Know what each lever helps and what it costs. Read each row from the gain to the catch. Note that caching helps two things at once: it cuts both processing time and cost.

Configuration levers compared on what they help and what they cost

Lever	Relieves	Costs
Lower effort	Cost and latency	Restore effort if quality falls short
Next model tier up	Accuracy	Compare it on cost per completed task
Batch API	Cost, with **50%** off every token	Results may take up to **24** hours
Prompt caching	Latency and cost	Cache writes cost **25%** more than base input with the **5**-minute TTL

When accuracy binds

Restore effort first, then move up a tier. If quality falls short and you lowered effort, raise it back. Otherwise try the next model tier at its default effort. Effort defaults to high on most models but medium on Opus **5.5** and Haiku **5.5**.

EXAM TRAP

Jumping straight to a bigger model is not the first move if you lowered effort earlier. Restore effort first, then try the next tier up.

When cost binds

Lower effort on the current model before switching models. If quality is fine but cost is high, try lower effort on the model you use now. Compare options on cost per completed task, not per token, because you pay for completed tasks.

EXAM TRAP

The cheapest price per token is not the cheapest system. You pay for completed tasks, so compare cost per completed task.

When latency binds

Fewer tokens processed and generated means a faster response. Haiku **5.5** gives the fastest responses while staying highly capable, and low effort is its fastest level. Shorter prompts and outputs help on any model.

Cutting time to first token

Pre-warm the cache so the first request does not miss. Loading the system prompt or tools into the cache before a user arrives can reduce the first-request miss penalty and lower **TTFT**. Streaming then shows output early, improving perceived speed.

When latency does not matter

The Batch API halves cost if results can wait. It takes **50%** off every token, cached ones included, but results can arrive any time within **24** hours. So it suits work whose results can wait that long.

Key takeaways

- Turn on prompt caching first; every agentic turn resends the whole conversation.
- Cost too high but quality fine: lower effort on the current model.
- Quality too low: restore effort if lowered, else try the next tier up.
- For speed, Haiku **5.5** at low effort is the fastest option.
- Compare models on cost per completed task, not cost per token.

Sources: Optimizing for cost and intelligence - Claude Platform Docs (opens in a new tab) · Prompt caching - Claude Platform Docs (opens in a new tab) · Reducing latency - Claude Platform Docs (opens in a new tab)

Instrumentation and Monitoring Gaps

Start here

THINK FIRST

When a scaled agent system fails quietly, which missing signal let it go unnoticed, and what monitoring would have caught it?

Why it matters: Questions describe a failure nobody saw and ask which instrumentation gap hid it and which monitoring strategy would have surfaced it.

Monitoring only sees what you switch on and route somewhere. Telemetry is often off or redacted by default, and event signals such as webhooks can be missed. Close gaps by enabling the right signals and adding a scheduled check behind them.

Key terms

Term	Meaning
Span	A trace record sent to the traces exporter; a subagent's API and tool spans nest under its parent's tool span.
Exporter selector	A variable such as `OTEL_TRACES_EXPORTER` that decides where a telemetry signal is sent.
Offline reviewer	A separate job that has Claude read each session's events after they are written.
Managed settings	Administrator-controlled settings that can lock telemetry choices for every developer.

Three telemetry signals

Claude Code exports metrics, events and optional traces. Metrics arrive as time series data. Events use the logs/events protocol. Distributed traces through OpenTelemetry are optional, so a team can have dashboards yet no trace of what a session did.

Tracing is off by default

Traces need two flags plus an exporter. Set both `CLAUDE_CODE_ENABLE_TELEMETRY=1` and `CLAUDE_CODE_ENHANCED_TELEMETRY_BETA=1`. Then `OTEL_TRACES_EXPORTER` picks where spans go. Miss any of these and no spans arrive.

EXAM TRAP

Turning on telemetry alone does not produce traces. The enhanced telemetry beta flag must also be set to **1**.

Redaction hides the detail

Spans redact prompts and tool details by default. Prompt text, tool inputs and tool content all stay hidden. Spans can show that a tool ran but not what it received.

Variables that add redacted content back to spans

Content	Hidden by default	Variable to include it
User prompt text	Yes	`OTEL_LOG_USER_PROMPTS=1`
Tool input details	Yes	`OTEL_LOG_TOOL_DETAILS=1`
Tool content	Yes	`OTEL_LOG_TOOL_CONTENT=1`

Where settings are ignored

Repository settings cannot turn telemetry on. Claude Code ignores exporter variables placed in a repository's `.claude/settings.json` and `.claude/settings.local.json`. Use managed settings, the shell or `~/.claude/settings.json` instead. A developer can still switch a signal off, so lock the selectors in managed settings too.

EXAM TRAP

Claude Code does not pass `OTEL_*` variables to the processes it spawns. Hooks, MCP servers, language servers and the Bash tool do not inherit them.

Tracing subagent work

Subagent spans nest under the parent's tool span. When the Agent tool spawns a subagent, its API and tool spans sit under the parent's `claude_code.tool` span. On cost data, the `query_source` attribute names the subsystem behind a request, such as `main` or `subagent`.

Check the setup works

Look for `claude_code.session.count` in the backend. Claude Code emits this metric when a session starts, so its absence points to a broken export. Remember the timing too: metrics export every **60000** ms by default, logs every **5000** ms.

Back up webhooks with polling

Webhooks can be missed, so poll sessions on a schedule. Managed Agents sends a webhook for each event in the table, but a delivery can go missing. List sessions on a timer and look at any that sit idle. For behavior beyond the agent's grant, run an offline reviewer over session events.

Managed Agents failures and the webhook that reports each

Failure	Webhook	Backup check
Session reaches a budget	`session.budget_reached`	Scheduled session listing
Session terminates	`session.status_terminated`	Scheduled session listing
MCP OAuth credential cannot refresh	`vault_credential.refresh_failed`	Scheduled session listing

Key takeaways

- Tracing is off by default and needs two enable flags plus a traces exporter.
- Spans hide prompts, tool inputs and tool content unless the matching variables are set to **1**.
- Telemetry variables in a repository's `.claude` settings files are ignored.
- Never rely on webhooks alone: list sessions on a timer and look at idle ones.

Sources: Monitoring - Claude Code Docs (opens in a new tab) · Review and monitor agent sessions - Claude Platform Docs (opens in a new tab)

Retrieval Architecture Selection

Start here

THINK FIRST

Given a corpus, its queries and a budget, when should Claude use a RAG pipeline, agentic search or the whole corpus in context?

Why it matters: Questions describe a corpus, its queries and latency or cost limits, and ask which retrieval architecture and configuration fit best.

Putting the whole corpus in context is bounded by the model's window and by context rot. Retrieval supplies search results from a fixed pipeline or from tools Claude calls. Citations tie answers to your documents, and dynamic filtering keeps irrelevant search results out of context.

Key terms

Term	Meaning
Context rot	The drop in accuracy and recall as the token count in context grows.
Search result block	A content block that lets Claude cite your own documents, with the origin and title you supply.
Dynamic filtering	Claude writes and runs code to filter web search results before they enter the context window.
Dynamic RAG	Retrieval where your custom tools return search results during the conversation.

Limits of full-corpus context

More context is not automatically better. As the token count grows, accuracy and recall degrade. Window size is a ceiling too: the deprecated Claude Sonnet **4.5**, for example, has a **200k**-token window. Prompt caching changes what you pay for a repeated corpus, but cached prefixes still fill the window.

EXAM TRAP

Prompt caching does not free up context space. It changes what you pay for cached tokens, not whether they count toward the window.

Pipeline or agentic retrieval

Search results can be fetched ahead or by tools. Pre-fetched or cached content goes straight into user messages. Custom tools that return search results support dynamic RAG. To make retrieval run on every request, force the tool call with `tool_choice`.

Ways to bring retrieved content to Claude

Approach	How content arrives	Note
Pre-fetched results	Placed directly in user messages	Suits pre-fetched or cached content
Custom retrieval tool	Your tool returns search results	Force it with `tool_choice` to run every time
Web search tool	Claude searches the web as needed	Cap searches per turn with `max_uses`

Controlling web search cost

Each 1,000 web searches cost 10 USD, plus tokens. Its results count as input tokens. Dynamic filtering in `web_search_20260209` or later lets Claude filter results with code first, so less irrelevant content reaches context. Claude also skips searching when a request draws on stable knowledge.

EXAM TRAP

Basic web search does not filter what it loads. Every result enters the context window, and much of it can be irrelevant.

Configuring citations

Citations on search result blocks are off by default. Enable them explicitly, and give every search result in a request the same setting. Claude cites whole text blocks, so split content into smaller blocks for finer citations. These blocks hold text only, not images.

Media in the corpus

A request can carry up to 600 images or PDF pages. Models with a **200k**-token window allow only **100**. So check the page count against the model's window size before loading a document set whole.

Key takeaways

- Context rot means a bigger context can lower accuracy and recall.
- Cached prefixes still count toward the context window.
- Use custom tools for dynamic RAG and user messages for pre-fetched content.
- Turn on citations explicitly and split content into small blocks for precise citations.

Sources: Context windows - Claude Platform Docs (opens in a new tab) · Search results - Claude Platform Docs (opens in a new tab) · Web search tool - Claude Platform Docs (opens in a new tab)

Connection Mechanism Selection

Start here

THINK FIRST

Given a task's limits on latency, security and state, how do you choose between an MCP server, a direct API call or a subagent handoff?

Why it matters: Questions give latency, security or state constraints and ask which connection mechanism or MCP transport fits them.

Pick the connection that the constraints favor. HTTP suits remote request-response servers, WebSocket suits servers that push events, and stdio suits local system access. A subagent handoff keeps noisy side work in its own context and returns only a summary.

Key terms

Term	Meaning
MCP connector	A Messages API feature that reaches remote MCP servers directly, with no separate MCP client.

Term	Meaning
Stdio server	An MCP server that runs as a local process on your own machine.
Subagent	A helper with its own context window, system prompt, tool access and permissions.
Vault	Where a Managed Agents session gets MCP credentials, so secrets stay out of the agent definition.

Match the transport to the server

HTTP is the recommended transport for remote MCP servers. It is the most widely supported transport for cloud services. It also supports OAuth and the `--transport` flag. The older SSE transport is deprecated, so use HTTP where you can.

Three MCP server transports compared

Feature	HTTP	WebSocket	Stdio
Where it runs	Remote server	Remote server	Local process on your machine
Best fit	Servers that only answer requests	Servers that push events unprompted	Direct system access or custom scripts
Default idle window	**5** minutes	**5** minutes	**30** minutes

EXAM TRAP

WebSocket is not the default pick for remote servers. Use it only when the server pushes events, because it lacks OAuth and the `--transport` flag.

Connecting from the Messages API

The MCP connector calls remote servers with no MCP client. Only tool calls from the MCP specification are supported. The server must be public over HTTP, so local stdio servers cannot connect directly. You also handle the OAuth flow and token refresh yourself.

EXAM TRAP

A local stdio server cannot be attached to the MCP connector. The connector needs a server that is publicly exposed over HTTP.

Security and data retention

MCP connector data falls outside Zero Data Retention. Tool definitions and results exchanged with MCP servers follow Anthropic's standard retention policy. Servers that fetch external content add prompt injection risk. So verify you trust each server before connecting it.

EXAM TRAP

A ZDR arrangement does not cover data sent through the MCP connector. That data is kept under the standard retention policy.

Handing off to a subagent

A subagent keeps noisy side work out of the main context. Each subagent has its own context window, system prompt, tool access and permissions. It suits a side task that would flood the conversation with logs or file contents. Only its summary comes back.

Scoping MCP servers and secrets

Define a server inline in a subagent to save main context. A server listed in the subagent's `mcpServers` field stays out of the main conversation. Its tool descriptions then use no context there. In Managed Agents, each session takes MCP credentials from a vault, not the agent definition.

Key takeaways

- HTTP is the recommended transport for remote MCP servers, and SSE is deprecated.
- Use WebSocket only for servers that push events, and stdio for local system access.
- The MCP connector supports only tool calls and needs a public HTTP server.
- Data exchanged through the MCP connector is not covered by ZDR.
- Hand noisy side tasks to a subagent, which returns only a summary.

Sources: Connect Claude Code to tools via MCP - Claude Code Docs (opens in a new tab) · Create custom subagents - Claude Code Docs (opens in a new tab) · MCP connector - Claude Platform Docs (opens in a new tab)

Progressive Discovery vs. Monolithic Context Loading

Start here

THINK FIRST

When should an agent discover tools and context on demand instead of loading everything into the window up front?

Why it matters: Questions describe a workload's context demands and latency tolerance and ask which loading strategy fits its economics.

Everything in the request counts toward the context window, and more tokens can lower accuracy. Tool search keeps tool definitions out until Claude asks for them, at the price of one extra lookup turn. Prompt caching suits a large but fixed tool list, and it lowers only what you pay.

Key terms

Term	Meaning
Context rot	The drop in accuracy and recall as the number of tokens in context grows.
Tool search	A way for Claude to find deferred tool definitions on demand instead of loading them all.
Prompt caching	Reuse of a stored prompt prefix that cuts cost on later requests, not token count.
Context editing	Removing old tool results from history mid-conversation once they have done their job.

Everything in the window counts

Tool definitions, the system prompt and all messages use context. Tool results count too, along with images and documents in messages. More is not automatically better. As tokens pile up, accuracy and recall degrade, which is called context rot.

EXAM TRAP

Loading every document and tool "just in case" is not the safe choice. Extra tokens can make the model less accurate, not more.

Progressive discovery with tool search

Deferred tools stay out of context until Claude asks. Mark tools with `defer_loading: true` and add a tool search tool. Claude then looks up a definition only when it needs one. The cost is one extra turn of latency; the gain is a much smaller baseline context.

When tool search pays off

Use it at 10 or more tools, or when baseline context grows. It suits a large toolset where most tools are not needed on a given turn. If the workload cannot spare an extra lookup turn, weigh that latency against the context saved.

Caching versus discovery

Caching cuts cost; discovery cuts tokens. The two solve different problems, so pick by what the workload needs to shrink.

Prompt caching and tool search compared

Feature	Prompt caching	Tool search
Best fit	A large but fixed list of tools	Many tools, few used each turn
Tokens in context	Unchanged; cached prefixes still count	Reduced; schemas load on demand

Feature	Prompt caching	Tool search
Main cost	Writes cost **25%** above base input price	An extra lookup turn when a tool is needed

EXAM TRAP

Caching a huge prompt does not free up the context window. The cached tokens still count; only the price of reusing them drops.

Keep intermediate results out

Programmatic tool calling runs a chain of calls as one script. Its intermediate results never enter the conversation history. Context editing removes old tool results once they have served their purpose. It does this without restarting the conversation.

Clearing and caching interact

Clearing tool results breaks cached prompt prefixes. When content is cleared, cached prompt prefixes are invalidated. So clear enough tokens to make that invalidation worthwhile. The `clear_at_least` parameter sets a minimum number of tokens cleared each time.

Key takeaways

- Tool definitions, the system prompt and tool results all use context window space.
- Tool search trades one extra lookup turn for a much smaller baseline context.
- Consider tool search at **10** or more tools, most unused on any turn.
- Prompt caching lowers cost but never lowers the tokens counted in context.

Sources: Context windows - Claude Platform Docs (opens in a new tab) · Manage tool context - Claude Platform Docs (opens in a new tab) · Context editing - Claude Platform Docs (opens in a new tab)

Production Reliability Controls

Start here

THINK FIRST

Reading error traces from a production integration, how do you tell which reliability control fits without causing duplicate side effects?

Why it matters: Questions show error traces or throughput data and ask which retry, rate-limit, idempotency or fallback control fixes the failure safely.

Each failure has its own signature and its own fix. Claude Code retries transient errors with backoff, but it does not re-run a request that fails after a tool call completed. Retry refusals on a fallback model sized for the load.

Key terms

Term	Meaning
Exponential backoff	Claude Code's retry pattern for transient failures, up to **10** retries before an error.
Stop reason	Part of a successful response that says why generation stopped normally.
Server-side fallback	The API retries a request on a fallback model, but only after a safety classifier decline.
Sticky routing	After a fallback, later turns of that conversation go straight to the fallback model.

Read the failure signature

Stop reasons mean success; `4xx` and `5xx` codes mean failure. An error means request processing failed. A stop reason comes inside a valid response and explains why generation ended. For example, `pause_turn` means a server-tool loop hit its limit, which is **10** iterations per request by default.

Which failures to retry

Claude Code retries transient failures before it reports them. It makes up to **10** retries with exponential backoff, then shows an error. Some failures are never retried, as the table shows.

How Claude Code treats common failures

Failure	Retried?	Why
Temporary **429** throttle	Yes	It is a transient throttle, retried with backoff
Gateway spend-limit **429**	No	It is not a throttle
TLS certificate validation failure	No, reported on the first attempt	The certificate setup needs fixing
TLS handshake timeout	Yes	It is a transient TLS condition

EXAM TRAP

Not every **429** is a throttle. A gateway's spend-limit **429** is not retried, since it is not a throttle.

Avoid duplicate tool effects

Claude Code does not re-run a request after a tool call completed. This covers a server error, dropped connection or stalled stream after Claude finished a text block or tool call. A re-run might run those same tool calls a second time. It keeps the completed work and continues the turn from there.

Size fallback for load

Every fallback attempt counts against its own model's limits. That includes attempts that declined. If the fallback model is rate limited or overloaded, no attempt is made and the refusal comes back. Give it rate limits big enough for your expected volume of refusals.

Sticky routing after fallback

Fallen-back conversations stick to the fallback model. The routing decision is kept for about **1** hour, within your organization only. It is best-effort, so your code must still handle the requested model being tried again.

Monitor what errors miss

Refusals arrive as HTTP 200, so error dashboards miss them. Track refusals as their own signal and alert on the gap between refusals and fallback-served responses. Read rate limits from the Rate Limits API at startup and on a schedule. Each limit is listed by type, such as `requests_per_minute`.

Key takeaways

- Retry transient failures with backoff, up to **10** times in Claude Code.
- Claude Code does not re-run a request that fails after a tool call completed, since the tool could run twice.
- Server-side fallback reacts only to safety classifier declines, not to rate limits or overloads.
- Instrument refusals separately, because they return HTTP **200**.

Sources: Error reference - Claude Code Docs (opens in a new tab) · Refusals and fallback - Claude Platform Docs (opens in a new tab) · Stop reasons and fallback - Claude Platform Docs (opens in a new tab) · Rate Limits API - Claude Platform Docs (opens in a new tab)

Domain 4 | 17% of the exam | 22 min read

Evaluation, Testing, and Optimization

Overview

This domain is about 17% of the exam, roughly 12 of its 70 questions.

Every point below comes from Anthropic's official documentation; each section lists its sources.

Evaluation Coverage Gaps

Start here

THINK FIRST

After a failure slips into production, how do you tell which metric your evaluation suite was missing?

Why it matters: Questions describe a production failure and an existing eval suite and ask which missing metric would have caught it.

An evaluation suite covers six dimensions: accuracy, groundedness, latency, cost, safety and security. A failure that reached production points to a dimension, or a grading target, the suite never measured. The fix is the specific metric that would have flagged it before release.

Key terms

Term	Meaning
Metric dimension	One of six areas to measure: accuracy, groundedness, latency, cost, safety or security.
Time to first token	Time from sending the prompt until the first response token, key when streaming.
Baseline latency	Total time to handle the prompt and produce a reply, leaving token throughput aside.
Noise bar	The run-to-run spread found by repeating the baseline, used to judge a real change.

Name the missing dimension

Match the failure to one of six metric dimensions. A suite can score accuracy well and still miss latency, cost, safety or security. Ask which dimension the failure belongs to, then which metric would have shown it pre-release.

Pick the right latency metric

Streaming users feel time to first token most. Each latency metric answers a different question about speed.

Two latency metrics compared

Metric	What it measures	When it matters
Time to first token	Prompt sent until the first token appears	Streaming responses
Baseline latency	Prompt processing plus response generation	Overall response time, ignoring token speed

EXAM TRAP

A good total response time does not prove a streaming interface feels fast. The missing metric there is time to first token.

Cost metrics that undercount

Several usage fields miss part of the real spend. A cost gap often hides in how usage is read.

Agent SDK usage pitfalls and the fix

Pitfall	Why it misleads	Fix
Result `usage` field	Counts only the top-level loop	Use `modelUsage` for the whole tree
Parallel tool calls in one turn	Messages share one ID	Deduplicate usage by message ID
Per-step `output_tokens`	It is a placeholder	Read output tokens from the result message
Failed conversations	They still consume tokens	Read every result, including error subtypes

EXAM TRAP

`total_cost_usd` and `costUSD` are not billing data. They are client-side estimates, so do not treat them as authoritative.

Build tasks from real traffic

Pull evaluation tasks from production logs, weighted like real use. Write an outcome check for each task, such as tests passing, a ticket closed or a correct row count. Record cost per task beside the score.

Gate changes against noise

Repeat the baseline before the first change. Identical runs of one configuration can differ by several tasks. That spread is the noise bar a change must clear. A multimodel setup must beat the single model's whole

score-versus-spend curve.

Keep watching after release

Shadow the winner on a traffic slice before cutover. Keep the evaluation suite running after the switch, too. Re-running failures at higher effort needs a real failure signal. If the grader approves bad output, those failures are never retried.

EXAM TRAP

Retrying failures at higher effort fixes nothing if the grader passes bad output. Without a failure signal, nothing triggers the retry.

Key takeaways

- Check all six dimensions: accuracy, groundedness, latency, cost, safety and security.
- Use time to first token when users see streamed responses.
- Read whole-tree usage from `modelUsage`, not the top-level `usage` field.
- Repeat the baseline first so a change has a noise bar to beat.

Sources: Claude Certification Program Exam guide Claude Certified Architect – Professional Exam Guide (opens in a new tab) · Optimizing for cost and intelligence - Claude Platform Docs (opens in a new tab) · Reducing latency - Claude Platform Docs (opens in a new tab) · Track cost and usage - Claude Code Docs (opens in a new tab)

Deriving Evaluations from Business Processes

Start here

THINK FIRST

How do you turn a support team's business goals into evaluations that can gate a prompt change, a model change or a release?

Why it matters: Questions describe a business process and ask which part makes a valid evaluation, which success criterion fits, and which test mix covers it.

Break the ideal customer interaction into every task, so each one can be prompted and evaluated. Then define success criteria with measurable benchmarks and goals. Judge readiness with a systematic mix of quantitative and qualitative methods.

Key terms

Term	Meaning
Success criterion	A measurable target tied to a business outcome, such as an escalation accuracy goal.
Deflection rate	The share of customer inquiries the chatbot resolves with no human help.

Term	Meaning
Escalation accuracy	Correct escalations counted against conversations that needed escalation but did not get it.
Rubric	A list of explicit, gradeable criteria that a grader scores one at a time.
CSAT	Customer satisfaction score, usually collected through a survey after the interaction.

Start from the tasks

Break the ideal interaction into every task first. Do this before you build anything. Each task then gets its own prompt and its own evaluation. The task list also shows how wide a range of interactions your test cases must cover.

Tie each criterion to an outcome

Define success criteria with the team that owns the process. Work with the support team to write detailed evaluations with measurable benchmarks and goals. Each example below pairs a way to measure with a numeric target.

Example success criteria for a support chatbot

Metric	How it is measured	Target
Query comprehension	Review a sample of conversations for correct intent	**95%** or higher
Response relevance	LLM-based grading over a set of conversations	**90%** or above
Escalation accuracy	Correct escalations against missed ones	**95%** or higher
Deflection rate	Inquiries handled with no human intervention	**70–80%**, by inquiry complexity
Sentiment	Sentiment analysis at the start and end of each conversation	Kept or improved in **90%**
CSAT	Post-interaction surveys	**4** out of **5** or higher
Company and product facts	Correctness against the information given in context	**100%** accuracy

EXAM TRAP

Targets differ by metric. Company and product facts aim for **100%** accuracy, while deflection aims for only **70–80%**.

Match the fix to the failure

Not every failing eval needs prompt engineering. For example, you can sometimes improve latency and cost more easily by choosing a different model than by rewriting the prompt.

EXAM TRAP

Rewriting the prompt is not always the answer to a latency or cost miss. Choosing another model can fix it more easily.

Mix numbers with human review

Readiness comes from a systematic mix of methods. Combine quantitative metrics with qualitative review before you call a chatbot ready. Run the prompt in a test production setting too. Without that step, you cannot really tell how well it works.

Write gradeable rubrics

Each rubric criterion must be explicit and gradeable. The grader scores every criterion on its own, so vague wording gives noisy results. "The CSV has a numeric price column" beats "the data looks good." No rubric yet? Have Claude analyze a known-good example, then build the rubric from that analysis.

Run evaluations at scale

Use the Message Batches API for large evaluation runs. Batches handle thousands of test cases asynchronously. That makes it practical to test the full spread of tasks from the first step, not just a handful of samples.

Key takeaways

- Break the process into every task before building, so each task can be prompted and evaluated.
- Set measurable targets with the process owners, such as **95%** escalation accuracy.
- Latency and cost can sometimes be improved more easily by changing the model than the prompt.
- Judge readiness with both quantitative and qualitative methods.

Sources: Customer support agent - Claude Platform Docs (opens in a new tab) · Define outcomes - Claude Platform Docs (opens in a new tab) · Prompt engineering overview - Claude Platform Docs (opens in a new tab) · Batch processing - Claude Platform Docs (opens in a new tab)

LLM-as-Judge Design and Bias

Start here

THINK FIRST

When an LLM judge's scores disagree with human reviewers, how do you work out which bias is responsible and which correction fixes it?

Why it matters: Questions show judge results that disagree with human review and ask which bias caused it

and which correction to apply.

Human review is the reference an LLM judge is calibrated against. When the two disagree, name the responsible bias from the evidence, then choose the correction aimed at that bias, not a generic fix.

Key terms

Term	Meaning
LLM-as-judge	An evaluator named beside rubric-based evaluation, calibrated against human review.
Calibration	Checking and adjusting an LLM judge against human review.
Evaluation-design bias	Bias that comes from how the evaluation is built, as opposed to the data or prompt framing.
Metric dimension	What a suite measures: accuracy, groundedness, latency, cost, safety or security.

Start from the disagreement

The symptom is judge scores that disagree with human review. Scenarios hand you artifacts such as evaluation results and expect you to reason from them. Read the results closely before choosing a fix.

The four judge biases

Position, verbosity, self-preference and judge drift. These are the **4** biases a correction may need to target. The task is to pick the bias the evidence points to, then the correction that addresses it.

EXAM TRAP

A correction aimed at the wrong bias will not fix the disagreement. Match the correction to the responsible bias.

Calibrate against humans

Human review is the yardstick for a judge. Calibrating an LLM judge means comparing its verdicts with human review. A judge whose verdicts disagree with that review is what this objective asks you to diagnose.

Find where the bias comes from

Data, prompt framing and evaluation design are distinct origins. Before choosing a mitigation, decide where the bias comes from. Each origin calls for its own mitigation, so identifying it comes first.

EXAM TRAP

Not every bias is a data problem. It may come from prompt framing or evaluation design, each needing a different mitigation.

Biased judge or missing metric

A missing dimension is a different failure from a biased judge. If a production failure slipped past the suite, ask which dimension or grading target was missing: accuracy, groundedness, latency, cost, safety or security. That is a coverage gap, not judge bias.

Where this sits on the exam

Judge design falls under Domain 4, worth 17%. Domain **4**, Evaluation, Testing, and Optimization, carries **17%** of the Professional exam. Judge design and bias is one of its task statements.

Key takeaways

- An LLM judge is calibrated against human review; disagreement is the symptom.
- Know the four judge biases: position, verbosity, self-preference and judge drift.
- Choose the correction for the responsible bias, not a generic fix.
- Tell apart bias in the data, in prompt framing and in the evaluation's design.

Sources: Claude Certification Program Exam guide Claude Certified Architect – Professional Exam Guide (opens in a new tab)

Experimental Design for Configuration Comparison

Start here

THINK FIRST

With a limited budget, how do you design a comparison of two Claude configurations so the result reflects the change, not random variation?

Why it matters: Questions give a sample and cost budget and ask which experimental design gives an interpretable comparison of two configurations.

Identical runs of one configuration already differ, so a single run per configuration proves little. Repeat runs to measure that noise, test on realistic tasks with outcome checks, and confirm the winner on data the search never saw.

Key terms

Term	Meaning
Run-to-run noise	The spread in scores between identical runs of the same configuration.
Noise bar	The spread from repeated baseline runs, used to judge whether a change is real.
Held-out split	Tasks the configuration search never reads, kept to confirm the final choice.
Cost per completed task	Total spend divided by tasks actually finished, the fair way to compare models.

Start with success criteria

Prompt engineering assumes success criteria and a way to test them. The prompt-engineering guide assumes you already have clear success criteria for your use case. It also assumes some empirical way to test against them, and a first draft prompt to improve.

Build the task set

Sample real tasks, weighted like real traffic. Pull a few tasks from production logs and write an outcome check for each, such as tests passing or a row count matching. Record cost per task beside the score.

EXAM TRAP

Comparing on typical tasks hides the expensive ones. Price the tail by comparing models on your hardest **10%** of tasks, not the median.

Repeat runs to measure noise

Run the baseline more than once before any change. Identical runs of one configuration differed by several tasks. The repeats give a noise bar, so a change is adopted only when it beats that spread. Anthropic used **3** runs per configuration on **200** problems.

EXAM TRAP

One run per configuration cannot separate the configuration effect from noise. Repeat the baseline to know how large chance differences are.

Interpreting differences

A gap inside the noise is not a real difference. These examples show how results read against run-to-run noise.

Example comparisons read against run-to-run noise

Comparison	Score result	Conclusion
Three answer formats, **3** runs each	All within noise	Choose on cost, not correctness
Cache breakpoints added	Pass rate fell less than the spread	Cost per task cut without real quality loss
Opus **5.5** vs Fable **5.1**, default effort	Matched, inside noise	Roughly one fifth of the cost per solved task

Compare across the cost curve

Test effort levels, not just the default. Baseline model tiers at several effort levels and plot score against spend. A multimodel setup must beat the single model's whole curve. Judge models by cost per completed task rather than per token.

Guard against overfitting

Confirm on a split the search never read. A change can fit only the tasks the search saw. Judging adoption on a held-out split catches that. Before cutover, run the winner in shadow on a slice of traffic, then keep the suite running.

Stay within budget

Some savings cut spend without touching quality. Batch processing costs **50%** less for work that can wait up to **24** hours. Prompt edits are not always the best fix: another model can sometimes improve latency and cost more easily. Published results are directional, so measure your own workload.

Key takeaways

- Repeat the baseline first; its spread is the noise bar a change must beat.
- Use production tasks weighted like real traffic, with outcome checks and cost per task.
- Confirm the winner on a held-out split the search never read.
- Judge models by cost per completed task rather than cost per token.
- Shadow the winner on a traffic slice before cutover.

Sources: Optimizing for cost and intelligence - Claude Platform Docs (opens in a new tab) · Prompt engineering overview - Claude Platform Docs (opens in a new tab)

Root-Cause Isolation from Logs and Traces

Start here

THINK FIRST

Given a failing Claude system's logs, how do you tell a prompt problem from a retrieval, tool, hallucination or model problem?

Why it matters: Questions show a log, trace or symptom and ask which failure class caused it and what the fix should be.

Key terms

Term	Meaning
Prompt failure	The model did what the wording literally asked, not what you meant.
Model mismatch	A request uses a setting the chosen model no longer accepts, such as `budget_tokens` on Claude **4.7**.
Overtriggering	The model calls tools or skills too often because the prompt pushes it too hard.
Prompt chaining	Splitting a task into steps, each a separate API call you can log and inspect.

Read the failure signature

Some failures return a 400, while others raise no error at all. Setting `budget_tokens` on Claude **4.7** or later returns a **400**, and so does forced tool use with adaptive thinking on models such as Opus **5.5**. Suggestions where edits were meant come back with no error.

Common symptoms and the failure class they point to

Symptom in the trace	Likely class	Corrective action
400 error after setting `budget_tokens`	Model mismatch	Remove it and set the effort parameter
Claude suggests edits but never makes them	Prompt failure	Rewrite as a direct instruction to change the code
Answer cites code Claude never opened	Hallucination	Require reading referenced files before answering
Many subagents spawned for a simple search	Subagent overuse	Steer toward a direct `grep` call

EXAM TRAP

Do not answer a **400** from `budget_tokens` on Claude **4.7** or later by rewording the prompt. Remove the setting and use the effort parameter to control thinking depth.

Model and API mismatch

Settings from older models stop working on newer ones. On Claude **4.7** and later, `budget_tokens` is rejected; use the effort parameter. From the **4.6** models, prefilling the last assistant turn is no longer supported. A system migrated without updating its requests fails here.

EXAM TRAP

Switching to a bigger model is not the first fix for latency or cost trade-offs. Tuning effort on the same model is often the better lever.

Tool and subagent errors

Forcing a tool while thinking is on can return a 400 error. With manual extended thinking, `tool_choice` must stay at `auto` or `none`. With adaptive thinking on models such as Opus **5.5**, forced tool use also errors. Trace logs showing a forced `tool_choice` explain the failure.

Overtriggering and overuse

Too many tool calls can trace back to an aggressive prompt. Opus **4.5** and **4.6** follow the system prompt closely, so aggressive tool instructions can cause overtriggering; soften that wording. Opus **4.6** also tends to spawn subagents where a direct `grep` call would do.

Prompt failures

Claude sometimes suggests changes instead of making them. Asking whether Claude can suggest changes may produce only suggestions. If the log shows advice where edits were expected, the fix is an explicit instruction, such as telling it to change the function.

Retrieval and long context

Where long inputs sit in the prompt affects answer quality. For inputs of **20k+** tokens, put the documents near the top, above the query and instructions. Asking Claude to quote the relevant parts first helps it focus and ignore the rest.

EXAM TRAP

Weak answers over long documents are not always a retrieval bug. Check whether the documents sit below the query, and whether quoting first was requested.

Hallucination and lost reasoning

Speculating about unopened code invites hallucinations. Tell Claude never to speculate about code it has not opened. Also echo assistant messages exactly: dropping thinking blocks loses their reasoning with no error. Editing earlier turns invalidates later thinking blocks on newer models.

Make failures traceable

Separate API calls per step make root causes visible. In a chained flow such as draft, review and refine, each step is its own call. Each call can be logged, scored or branched on its own, so the failing step stands out.

Key takeaways

- Setting `budget_tokens` on Claude **4.7** or later returns a **400**; use the effort parameter instead.
- Forced tool use fails with thinking on; keep `tool_choice` at `auto` or `none`.
- Suggestions instead of edits is a prompt failure; give an explicit instruction to act.
- Chain steps as separate calls so each one can be logged and checked.

Sources: Prompting best practices - Claude Platform Docs (opens in a new tab) · Thinking in tool and multi-turn workflows - Claude Platform Docs (opens in a new tab) · Choosing the right model - Claude Platform Docs (opens in a new tab)

Cost and Latency Optimization

Start here

THINK FIRST

Reading cost and latency telemetry, how do you find where spend or delay concentrates and fix it without dropping below the quality floor?

Why it matters: Questions give usage, cost or cache telemetry and ask where spend or latency comes from and which fix keeps quality acceptable.

Usage telemetry shows where tokens go, split by type over time. When the cache misses, diagnostics report the earliest divergence, so fix that one first. Then apply one optimization at a time and check that quality stays above the threshold.

Key terms

Term	Meaning
Cache read tokens	Input tokens served from the prompt cache, reported as `usage.cache_read_input_tokens`.
Cache diagnostics	A request option that reports where a request first diverged from the previous one.
TTFT	Time from sending the prompt to the first response token; key for streaming apps.
Quality floor	The output quality threshold an optimization must not drop below.

Where usage and cost data live

The Usage and Cost APIs report at different granularity. Use the Usage API to see token types over time, and the Cost API for daily spend.

Usage API and Cost API compared

Feature	Usage API	Cost API
Time buckets	**1m**, **1h** or **1d**	Daily (**1d**) only
Measures	Tokens split into uncached input, cached input, cache creation and output	Costs in USD, including web search and code execution

EXAM TRAP

In OpenTelemetry spans, `gen_ai.usage.input_tokens` already includes cache reads and writes. Adding the cache attributes on top double counts.

Spotting a cache problem

Cache read tokens falling to zero is the warning sign. Without diagnostics, that drop is the only signal, and it does not say what changed. Turn on diagnostics by passing `previous_message_id: null` on the first turn, then the previous response's id.

Reading diagnostics results

Each divergence reason has its own fix. Fix the earliest divergence first, since only that one is reported and later ones may hide behind it.

Cache miss reasons and their fixes

Reason	Typical cause	Fix
`system_changed`	A timestamp or request ID in the system prompt	Keep the system prompt byte-stable; move the value to the user message
`tools_changed`	Tools added, removed, reordered or serialized differently	Same tools each turn, fixed order, deterministic schemas
`messages_changed`	History truncated, edited or re-serialized	Append only; echo assistant content and tool results verbatim

When requests match but miss

Null diagnostics with low cache reads means no entry existed. The requests were identical, but the cache entry had expired, or the prompt was shorter than the minimum cacheable length.

EXAM TRAP

Null diagnostics with zero cache reads does not mean something changed in the prompt. The entry expired or the prompt was too short to cache.

Cutting cost

Batch processing gives a 50% discount on batch requests. Each request sent as a batch request costs half the usual price. That makes batch processing a direct lever on spend.

Cutting latency

Pick a faster model or make output appear sooner. Haiku **4.5** gives the fastest responses, with effort as its main speed and cost control. Streaming sends output before it is complete, so users see results sooner and **TTFT** matters most.

Protect the quality floor

Apply one optimization at a time and re-check quality. Build a latency eval, change a single technique, and confirm output quality stays above the threshold before applying the next one.

Key takeaways

- A drop of `cache_read_input_tokens` to zero signals a cache problem; diagnostics tell you why.
- Fix the earliest reported divergence first; later ones may be hidden.
- Keep the system prompt byte-stable and put per-request values in the user message.
- Don't add cache attributes to `gen_ai.usage.input_tokens`; it already includes them.
- Test one optimization at a time against a quality threshold.

Sources: Usage and Cost API - Claude Platform Docs (opens in a new tab) · Cache diagnostics - Claude Platform Docs (opens in a new tab) · Reducing latency - Claude Platform Docs (opens in a new tab) · OpenTelemetry tracing for the Claude API - Claude Platform Docs (opens in a new tab)

Model Version Migration

Start here

THINK FIRST

A model you rely on has been deprecated: how do you decide whether to stay pinned, re-tune prompts, re-test or move now?

Why it matters: Questions give a deprecation timeline and test results for a new model and ask which migration step fits.

Key terms

Term	Meaning
Deprecated	Still works but no longer recommended; it has a named replacement and a retirement date.
Retired	No longer available for use; requests to the model fail.
Pinned model ID	An ID whose underlying model stays constant for the whole lifetime of that ID.
Alias	A convenience pointer that resolves to the most recent dated snapshot for a minor version.

Pinned IDs and aliases

A model ID pins one version for its whole lifetime. That guarantee does not cover aliases. An alias such as `claude-sonnet-4-5` resolves to the latest dated snapshot for its minor version. From the **4.6** generation on, the dateless ID is itself a fixed snapshot, and updates ship under new IDs.

EXAM TRAP

Switching to a dateless alias does not dodge a retirement. The Claude Sonnet **4.5** retirement also covers `claude-sonnet-4-5`.

Drift on a stable ID

Weights for an ID are fixed, but serving infrastructure can change. The request router, safety classifiers and sampling logic can be updated. Anthropic never changes the weights or configuration of an existing ID. So unexpected behavior changes on a previously stable ID most likely come from an infrastructure update.

EXAM TRAP

A behavior shift on an unchanged model ID does not mean the weights changed. An infrastructure update is the most likely cause.

Reading a deprecation timeline

Find where deprecated models run, then test early. Export the Usage page in Claude Console for a CSV broken down by API key and model. Then test your applications with the replacement well before the retirement date. The table shows one real timeline.

The Claude Sonnet 4.5 retirement on the Claude API

Date	Event	Model
September 30, 2026	Developers notified of the retirement	`claude-sonnet-4-5-20250929`
November 30, 2026	Retirement of the deprecated model	`claude-sonnet-4-5-20250929`; replacement `claude-sonnet-5-5`

Parameters that stop working

Some request parameters fail on newer API versions. In API version `2026-10-21`, `temperature`, `top_p` and `top_k` were removed and return a **400** error. Omit them and guide model behavior through prompting instead.

Migrating in Claude Code

Update Claude Code first, then change every model setting. A newer model can need a newer Claude Code version. Check which model you use, even with no model set or an alias such as `sonnet`. Set the recommended replacement for a deprecated model.

EXAM TRAP

Changing only your main model setting is not enough. Subagent definitions and any model your administrator set must change too.

Key takeaways

- Deprecated models still work until their retirement date; retired models fail every request.
- Publicly released models get at least **60** days' notice before retirement.
- A model ID stays constant for its lifetime; an alias can point to a newer snapshot.
- Drift on an unchanged model ID most likely comes from an infrastructure update.
- Test the replacement well before retirement, and update every place a model is set.

Sources: Model deprecations - Claude Platform Docs (opens in a new tab) · Model IDs and versioning - Claude Platform Docs (opens in a new tab)

Domain 5 | 16% of the exam | 18 min read

Governance, Safety, and Risk Management

Overview

This domain is about 16% of the exam, roughly 11 of its 70 questions.

Every point below comes from Anthropic's official documentation; each section lists its sources.

Guardrail Placement

Start here

THINK FIRST

Given a described misuse path, where would you place a guardrail so it closes the path without breaking legitimate use?

Why it matters: Questions describe a misuse path and a legitimate use case and ask which control, placed where, closes the path.

Each misuse path enters at a specific point: user input, third-party content, or Claude's output. Put the control at that entry point. Multiple risk levels then let you adjust how aggressive that control is.

Key terms

Term	Meaning
Indirect prompt injection	Hostile instructions hidden in third-party content Claude processes, while the user is trusted.
Harmlessness screen	A lightweight model that checks user input before it reaches the main conversation.
Least privilege	Withholding unneeded secrets, sandboxing tools and scoping permissions to limit injection damage.
Refusal	A response whose `stop_reason` is `refusal`, delivered as a successful HTTP **200**.

Match the control to the entry point

Place each guardrail where the misuse path enters. Different paths need different controls at different points.

Guardrails by where the misuse enters

Entry point	Control	Effect
User input	Lightweight model pre-screens the input	Harmful requests stop before the main conversation
Tool outputs	Small classifier checks each raw output	Only clean output returns as a `tool_result`
Third-party content	Deliver it only inside `tool_result` blocks	Never mixed with system or user instructions
Claude's output	Filter for leak keywords with regex	Reduces prompt leaks after generation

Contain indirect injection

In indirect injection, the attack rides in on content. Webpages, emails, documents and tool results can all carry it. JSON-encode untrusted strings to give clear delimiters that an attacker cannot break out of. If a screen flags `injection_suspected`, return an error or a stripped summary, not the raw content.

EXAM TRAP

Pasting third-party content into the system prompt or a plain user block is the wrong place. It belongs inside a `tool_result` block.

Keep your own instructions separate

Never put developer instructions inside tool results. Claude may ignore them or flag them as injection. Put them in a user turn after the `tool_result` block. Least privilege then limits the damage if an injection still succeeds.

Grade risk instead of blocking all

Multiple risk levels let you adjust how aggressive moderation is. High-risk queries can be blocked automatically. Users who send many medium-risk queries get flagged for human review. For repeat offenders, adjust responses and consider throttling or banning them.

EXAM TRAP

Binary moderation is not the gentlest option. It cannot flag a pattern of medium-risk queries for review while still serving normal ones.

Handle refusals correctly

A refusal arrives as a success, not an error. Monitoring built only on error rates will miss it. Treat a refusal from a screening classifier as a harmful verdict. Then reset the context by removing or rephrasing the turn, or refusals continue.

EXAM TRAP

Retrying the same conversation after a refusal will not help. The triggering turn must be removed, rephrased or the history cleared.

Do not overbuild leak defenses

Use leak-resistant prompting only when absolutely necessary. Leak-proofing makes the overall task more complex, which might degrade performance in other parts of the task. Post-processing is a separate technique: filter Claude's outputs for keywords that might indicate a leak.

Key takeaways

- Deliver third-party content inside `tool_result` blocks, never in system or user text.
- Screen user input with a lightweight model before the main conversation sees it.
- Use multiple risk levels to block high risk and route medium risk to people.
- Refusals return HTTP **200**, so error-rate monitoring alone will not catch them.

Sources: Mitigate jailbreaks and prompt injections - Claude Platform Docs (opens in a new tab) · Reduce prompt leak - Claude Platform Docs (opens in a new tab) · Content moderation - Claude Platform Docs (opens in a new tab) · Handle streaming refusals - Claude Platform Docs (opens in a new tab)

Prompt Injection Surface and Controls

Start here

THINK FIRST

Where can untrusted text enter Claude's context in your architecture, and what stops it from acting as an instruction?

Why it matters: Questions describe an agent's context inputs and ask where injection can enter and which control blocks it.

Anything that lands in the model's context can carry instructions, not just the user's prompt. Retrieved documents, tool results, MCP responses and third-party Skills all widen that surface. Controls limit what each input can reach and pin what was reviewed.

Key terms

Term	Meaning
Injection surface	Every input in the model's context, such as tool results or Skills, that could carry hostile instructions.
Version pinning	Naming an exact Skill version so a new upload cannot silently change production behavior.
Tool denylist	Blocking write or destructive MCP tools so an assistant stays read-only.

Term	Meaning
Separation of duties	A governance rule that a Skill's author must not also be its reviewer.

Map every input to the context

Four input types can carry injected instructions. Retrieved documents, tool results, MCP server responses and third-party Skills can all sit in the model's context. For each one, specify the controls that stop untrusted content from being treated as instruction.

Shrink what MCP servers expose

Give each MCP server only the tools and data it needs. Denylist write or destructive tools for a read-only assistant, or when a person should confirm state changes. You can also pin a server's tool list. Then a mid-conversation change on the server does not change what Claude receives.

EXAM TRAP

Trusting an MCP server once at setup is not enough. Without a pinned tool list, the server can change the tools Claude sees partway through a conversation.

Red flags in a third-party Skill

Some Skill contents are high-concern on sight. Review each Skill for these signs before it is deployed.

Skill risk indicators and why they matter

Indicator	Why it is risky	Concern
Directives to ignore safety rules or hide actions	Can bypass security controls	High
Bundled scripts in the Skill folder	They run with full environment access	High
Instructions that reference MCP tools	Extends access beyond the Skill	High
File-read and network tools together	Reviewers must consider their combined risk	Weigh together

Exfiltration without a network call

A Skill can leak data through Claude's replies. Blocking network tools does not close every path out. Reviewers should look for instructions that send data out through responses, or that change behavior on specific inputs.

EXAM TRAP

Removing network access does not make a hostile Skill harmless. Data can still leave in Claude's conversational answers.

Scanning and its blind spot

Scanning blocks a Skill that fails or has not finished its scan. One that passes with a warning stays usable, shown with a caution notice. The gap: Skills uploaded through the Skills API are not scanned at all. So API deployments depend on the review checklist and pinned versions instead.

Lock in what was reviewed

An unpinned request always runs the latest Skill version. Anyone in the workspace who uploads a new version changes production at once. Pin versions, take checksums when Skills are reviewed, and check them again when deploying. Signed commits prove provenance, and authors never review their own Skills.

EXAM TRAP

Reviewing a Skill once does not protect production if requests omit the version. The next upload replaces what you reviewed.

Key takeaways

- Documents, tool results, MCP responses and Skills all belong to the injection surface.
- Expose only needed MCP tools and denylist destructive ones for read-only use.
- Bundled Skill scripts are high concern because they run with full environment access.
- Pin Skill versions; Skills API uploads are not scanned.

Sources: Claude Certification Program Exam guide Claude Certified Architect – Professional Exam Guide (opens in a new tab) · MCP tunnels security - Claude Platform Docs (opens in a new tab) · Skills for enterprise - Claude Platform Docs (opens in a new tab) · MCP connector - Claude Platform Docs (opens in a new tab)

Risk Assessment and Ranking

Start here

THINK FIRST

Given a proposed agent deployment, which risks matter most, and how do you rank attack surface, failure, quota and supply-chain exposure?

Why it matters: Questions describe an agent deployment and ask which risk category is most exposed and which control reduces it.

Assume a prompt injection can succeed, because model behavior is not a security control. Deployment settings must limit what a successful attempt can do. No single limit replaces the others.

Key terms

Term	Meaning
Prompt injection	Hidden instructions in content an agent reads that steer what the agent does.

Term	Meaning
Untrusted input	A repository, package or archive that someone outside your team can write.
Session budget	A Managed Agents spend cap at list prices that pauses the session when reached.
Domain fronting	A trick that reaches a blocked host through an allowed hostname when TLS is not inspected.

Credential exposure

Never give the agent a Claude API key. Even from a vault, sandbox code could call the Claude API directly, start or delete sessions, and bypass the agent's tool settings, approvals and budget. Mitigate other credentials with vaults, limited networking and approvals.

EXAM TRAP

Putting a Claude API key in a vault does not make it safe to give the agent. Sandbox code can still use it directly.

Network attack surface

Set networking explicitly in every environment. A create request that omits networking might get unrestricted access. The sandbox proxy does not inspect TLS by default, so domain fronting could reach hosts outside the allowlist.

EXAM TRAP

Limiting allowed hosts does not stop credential misuse. The agent can still use a credential on any host you allowed.

Supply-chain risk

Content outsiders can write is untrusted input. Treat external repositories, packages and archives as untrusted. `allow_package_managers` opens every repository on the github.com, gitlab.com and bitbucket.org hosts. Anyone can publish there, so keep it false unless packages are listed.

Isolation and escape

Choose isolation strength to match the threat model. Sandboxed processes share the host kernel, so a kernel flaw could allow escape. Use gVisor or a VM when kernel-level isolation is required.

Isolation technologies compared

Technology	Isolation strength	Performance overhead
Sandbox runtime	Good, with secure defaults	Very low
Docker containers	Depends on setup	Low
gVisor	Excellent, if set up correctly	Medium to high

Technology	Isolation strength	Performance overhead
VMs such as Firecracker	Excellent, if set up correctly	High

Token and quota exhaustion

A session budget pauses spend but can be exceeded. At the cap, the session pauses, yet in-flight requests finish and their tool calls run, so spend can pass it. In containers, `--memory 2g` prevents resource exhaustion and `--pids-limit 600` stops fork bombs.

EXAM TRAP

A session budget is not a hard stop. Requests already running complete, so total spend can go over the cap.

Failure modes and blind spots

Platform settings leave some components unprotected. Deployment settings do not secure custom tools, MCP servers or self-hosted infrastructure, nor limit key permissions. Limit permissions where each key was issued. Webhooks can be missed, so list sessions on a schedule and check idle ones.

Key takeaways

- Model resilience to prompt injection is not a security control; settings must limit the damage.
- Never give an agent a Claude API key, even in a vault.
- Set networking in every environment create request to avoid unrestricted access.
- Treat anything outsiders can write, such as packages and repositories, as untrusted input.
- Session budgets can be exceeded by in-flight requests and their tool calls.

Sources: Security risks and sandbox boundaries - Claude Platform Docs (opens in a new tab) · Secure your Managed Agents deployment - Claude Platform Docs (opens in a new tab) · Securely deploying AI agents - Claude Code Docs (opens in a new tab) · Review and monitor agent sessions - Claude Platform Docs (opens in a new tab)

Human-in-the-Loop Checkpoints

Start here

THINK FIRST

In an agentic workflow, which steps truly need a human to approve them, and how do you gate those without slowing everything else?

Why it matters: Questions describe a workflow with risky steps and ask where the human approval gate belongs and how to enforce it.

Reversibility and stakes decide where a gate sits. Put a deterministic pre-execution gate only on irreversible, high-stakes actions, and let safe steps run freely so the workflow does not stall.

Key terms

Term	Meaning
Pre-execution approval	A human approves an action before it runs; unlike review after execution, it can stop an irreversible step.
Deterministic gate	A gate on an irreversible step that does not rest on the model's own judgment.
canUseTool	The Agent SDK callback asked to decide tool calls that need approval and no rule covers.
PreToolUse hook	A hook that runs before a tool call and can reject it so it never executes.

Where the gate must sit

Gate the irreversible, high-stakes action before it runs. A billing change has an immediate financial effect the agent cannot undo, so it needs pre-execution approval. Reviewing after execution cannot prevent an irreversible charge.

EXAM TRAP

Post-execution review sounds safe but is wrong for irreversible steps. By the time a human looks, the charge has already happened.

Gate placement options

After-the-fact review and plan gating both fall short. Compare the common designs on whether they stop the harmful action and whether they slow safe requests.

Human-in-the-loop designs compared

Design	Stops irreversible harm?	Main drawback
Review actions after execution	No	The charge has already happened
Approve the whole plan up front	Not reliably	Delays every request; execution may diverge
Model's own confidence threshold	Not reliably	Not deterministic, so not a reliable control

EXAM TRAP

Gating the whole plan is not the safest choice. It delays every request, and the run may still diverge from the plan a human approved.

Make the gate deterministic

Never leave an irreversible step to the model's judgment. A self-reported confidence threshold is not a reliable control. Claude resists prompt injection well, but model behavior is not a security control, so settings must limit harm if an attack works.

Agent SDK permission controls

Rules and modes decide which calls run, ask or are denied. `allowedTools` auto-approves listed tools, and `disallowedTools` blocks listed tools regardless of other settings. A rule such as `Bash(npm *)` approves only matching commands, so the rest still reach the gate.

Who answers the prompt

Unapproved calls go to canUseTool, or are denied. In `default` mode, a call needing approval that no allow rule covers triggers `canUseTool`; with no callback it is denied. `dontAsk` never prompts and denies such calls, which suits headless agents.

Hooks as hard stops

A PreToolUse hook can reject a call before it executes. The hook short-circuits the loop. The tool never runs, and Claude receives the rejection message instead.

Managed Agents pitfalls

Confirmation handling can silently remove the gate. A client branch that allows every call needing confirmation turns `always_ask` into always allow. A confirmation that arrives too late returns **200**, which does not mean the tool ran. No approval step precedes a run, so ask the agent to describe it and wait.

> **EXAM TRAP**
>
> A **200** reply to a tool confirmation does not prove the tool ran. A late confirmation also returns **200**.

Key takeaways

- Reversibility and stakes decide where a human gate must sit.
- Post-execution review cannot prevent an irreversible action.
- Gates on irreversible steps must be deterministic, never a model confidence score.
- Leave side-effect-free, reversible and deterministically constrained steps ungated.
- Allowing every confirmation in client code turns `always_ask` into always allow.

Sources: Claude Certification Program Exam guide Claude Certified Architect – Professional Exam Guide (opens in a new tab) · How the agent loop works - Claude Code Docs (opens in a new tab) · Security risks and sandbox boundaries - Claude Platform Docs (opens in a new tab) · Workflow runs - Claude Platform Docs (opens in a new tab)

Architectural Controls for Compliance Obligations

Start here

THINK FIRST

Given a compliance obligation, which mix of residency, retention and audit controls actually provides evidence that you meet it?

Why it matters: Questions state a compliance obligation and ask which residency, retention or audit controls evidence it.

Each obligation maps to a control with its own scope and gaps. Geo settings decide where data is processed and stored, while retention arrangements cover eligible features only. Audit tools differ in how far back they reach and whether they act in real time.

Key terms

Term	Meaning
Inference geo	Where inference runs: `global` by default, or `us` for US-based infrastructure only.
Workspace geo	Where data rests and endpoint processing runs; fixed when the workspace is created.
Zero data retention	A ZDR arrangement, enabled per organization, that covers eligible features only.
HIPAA readiness	A configuration for protected health information that blocks ineligible features.

Two separate geo controls

Inference geo and workspace geo are set independently. Inference geo defaults to `global`, which may run anywhere; `us` keeps inference on US infrastructure. Workspace geo covers data at rest and processing such as code execution. It is fixed at creation and cannot change later.

EXAM TRAP

Setting inference geo to `us` does not move stored data or code execution. Workspace geo controls those, and it cannot be changed after creation.

Enforce residency per workspace

A request outside the allowed geos is rejected. If a request names an `inference_geo` missing from `allowed_inference_geos`, the API returns a **400** error. That turns a residency policy into an enforced rule, not just a convention.

ZDR versus HIPAA readiness

Pick the arrangement that matches the data you hold. They handle ineligible features in opposite ways.

Zero data retention and HIPAA readiness compared

Feature	Zero data retention	HIPAA readiness
Organization holds PHI	Not needed on top of HIPAA readiness	The arrangement to use
Ineligible feature used	Not blocked; that data leaves ZDR	Blocked with a **400** error
Scope	Per organization; not inherited	Not on Claude Platform on AWS

EXAM TRAP

An organization handling health data does not need ZDR on top of HIPAA readiness. HIPAA readiness is the right control on its own.

Retention gaps to plan for

Some data stays retained even under ZDR. Managed Agents keeps session transcripts until you delete them, so ZDR does not cover it. A Claude Code session flagged for a policy violation may be kept up to **2** years. Never put PHI in JSON schemas, which lack PHI protections.

Audit: real time or after the fact

Choose the audit tool by when it must act. Each tool gives different evidence for an auditor.

Audit and governance tools compared

Tool	When it acts	What it offers
Inference hooks	Inline, before the prompt runs	Your security server can deny each prompt
Compliance API	After the fact	Activity Feed kept **6** years
claude.ai audit log export	After the fact	CSV only, capped lookback, no content

Key takeaways

- Inference geo `us` limits where inference runs; workspace geo governs stored data.
- HIPAA readiness blocks ineligible features, while ZDR lets them run outside the arrangement.
- ZDR is set per organization and does not cover Managed Agents transcripts.
- Use inference hooks to block prompts in real time; the Compliance API reviews later.

Sources: Data residency - Claude Platform Docs (opens in a new tab) · API and data retention - Claude Platform Docs (opens in a new tab) · Compliance API - Claude Platform Docs (opens in a new tab) · Zero data retention - Claude Code Docs (opens in a new tab)

Bias Source Identification and Mitigation

Start here

THINK FIRST

When results differ across groups, how do you tell whether the bias comes from the data, the prompt or the evaluation itself?

Why it matters: Questions show disparate results across groups and ask where the bias comes from and which mitigation fits that source.

Disparate results are a symptom, not a diagnosis. The bias can sit in the data, in how the prompt frames the task, or in how the evaluation is designed. Find the source first, because each one needs a different fix.

Key terms

Term	Meaning
Data bias	Bias that comes from the dataset itself, one of three sources to tell apart.
Prompt-framing bias	Bias that comes from how the prompt frames the task rather than from the data.
Evaluation-design bias	Bias built into how results are judged, such as an LLM judge that disagrees with human review.
LLM-as-judge	An evaluation design whose results can clash with human review because of biases such as position or verbosity.

Find the source first

Weigh the evidence before you choose a mitigation. When outputs or a dataset show different results across groups, ask where the bias comes from. It may be data bias, prompt-framing bias or evaluation-design bias. The mitigation must match that source.

EXAM TRAP

Prompt engineering is not the answer to every failing eval. Some problems are solved more easily another way, such as choosing a different model.

Bias in the evaluation

Judge results that clash with human review point to the evaluation. In an LLM-as-judge or rubric-based evaluation, look for four named biases: position, verbosity, self-preference and judge drift. Then pick the correction that targets the one responsible.

Bias in the prompt's framing

How you write the prompt shapes how Claude answers. A prompt's formatting style can influence the style of the response. Explaining why an instruction matters gives Claude more context about your goals, which leads to more targeted answers.

Examples that skew results

Few-shot examples must be diverse and relevant. Cover edge cases and vary the examples, so Claude does not learn a pattern you never meant. Keep them close to your actual use case.

EXAM TRAP

A set of near-identical examples is not neutral. Without variety and edge cases, Claude can pick up unintended patterns from them.

Test the mitigation

Define success criteria and empirical tests first. Set both up before any prompt engineering begins. A technique measured on one model is not proven for another, so re-check it against your own evals first.

Key takeaways

- Separate data bias from prompt-framing and evaluation-design bias before choosing a fix.
- When a judge disagrees with human review, check position, verbosity, self-preference and judge drift.
- Keep few-shot examples diverse and close to the real use case.
- Re-check any model-specific technique against your own evals.

Sources: Claude Certification Program Exam guide Claude Certified Architect – Professional Exam Guide (opens in a new tab) · Prompting best practices - Claude Platform Docs (opens in a new tab) · Prompt engineering overview - Claude Platform Docs (opens in a new tab)

Domain 6 | 7% of the exam | 10 min read

Stakeholder Communication and Lifecycle Management

Overview

This domain is about 7% of the exam, roughly 5 of its 70 questions.

Every point below comes from Anthropic's official documentation; each section lists its sources.

Unsupportable Commitments in Requirements and SLAs

Start here

THINK FIRST

Reading a draft SLA or discovery notes, how do you spot a promise the proposed Claude architecture cannot keep, and what change fixes it?

Why it matters: Questions quote discovery notes or a draft SLA and ask which commitment the design cannot meet and what clarification or change resolves it.

Many commitments clash with how Claude features actually combine. Fast mode, batch discounts, data residency and encryption each come with conditions. Check every promise on speed, cost, location and security against those conditions before you sign.

Key terms

Term	Meaning
Fast mode	The same model on a faster inference setup, giving up to **2.5x** output tokens per second at premium pricing.
Batch API	Asynchronous processing of large request volumes at a **50 percent** token discount.
Data residency	Settings for the region that runs inference (inference geo) and the region that stores data at rest (workspace geo).
CMEK	A customer-managed encryption key that protects data written after it takes effect.

Latency promises

Fast mode speeds output, not time to first token. A target on first-token latency cannot rest on fast mode. It raises output tokens per second by up to **2.5x**, with the same intelligence and capabilities.

Combinations that rule out fast mode

Commitment	Conflict	Resolution
Fast output and batch discount	Fast mode is not available with the Batch API	Split the workloads or drop one promise
Fast output under Priority Tier	Fast mode is not available with a Priority Tier commitment	Choose one of the two
Fast mode on Claude Opus **4.8**	Deprecated; removed on November **30**, **2026**	Move to a supported model

EXAM TRAP

Fast mode does not share standard rate limits. Exceeding its own limit returns a **429** with a `retry-after` header; to avoid waiting, retry without fast speed.

Cost promises

Batch discounts suit only work that can wait. The Batch API is meant for non-time-sensitive tasks. Managed Agents sessions get no batch discount, since sessions are stateful and interactive. A session budget sets a hard dollar limit on one session at list prices.

EXAM TRAP

A cost estimate carried over from older models can be too low. Claude **4.7** and later use a newer tokenizer that turns the same text into about **30 percent** more tokens.

Data location promises

US-only inference needs Claude 4.6 or later. On older models, a request with `inference_geo` returns a **400** error. On newer ones, US-only inference adds a **1.1x** multiplier to every token price. Residency multipliers also stack on top of fast mode pricing.

Platform and network promises

Managed Agents does not run on partner clouds. It is unavailable on partner-operated cloud platforms. The IP allowlist covers calls to the Managed Agents endpoints, not the agents' own networking. It also does not apply to Amazon Bedrock, Google Cloud or Microsoft Foundry.

Encryption promises

A customer-managed key is a one-way decision. Once attached, it cannot be removed or replaced, and it protects only data written afterward. A Claude Platform workspace that uses such a key has to retain data. If the key is deleted, Anthropic cannot recover the protected data.

Access duration promises

An allowlist limits where a key works, not how long. Add a short expiration to each key. Or switch to Workload Identity Federation and retire static keys altogether.

Key takeaways

- Fast mode cannot be combined with the Batch API or a Priority Tier commitment.
- Fast mode improves output speed, not time to first token.
- US-only inference needs Claude **4.6** or later and adds a **1.1x** price multiplier.
- A customer-managed key cannot be removed or replaced once attached.

Sources: Pricing - Claude Platform Docs (opens in a new tab) · Fast mode - Claude Platform Docs (opens in a new tab) · Customer-managed encryption keys - Claude Platform Docs (opens in a new tab) · Features overview - Claude Platform Docs (opens in a new tab) · IP allowlisting - Claude Platform Docs (opens in a new tab)

Architecture Decision Records

Start here

THINK FIRST

What makes an architecture decision record honest about its trade-offs and the risk that remains after the decision?

Why it matters: Questions show a draft ADR that misses a stakeholder constraint and ask which revision states the trade-off and residual risk.

A good ADR names the constraint, the trade-off chosen and the risk left over. It treats vendor figures as directional and controls as risk reducers, not guarantees. It also never treats model behavior as a security control.

Key terms

Term	Meaning
Residual risk	The risk a control lowers but does not remove; a good ADR revision makes it explicit.
Pinned model ID	A model ID whose underlying model stays the same for the ID's whole lifetime.
Deprecated model	A model that still works but is no longer recommended and has a retirement date.
Effort parameter	A setting that lowers latency and cost at some cost in intelligence, without changing model.

What the revision must do

Address the missed constraint and state the residual risk. The better revision makes the trade-off explicit. It also avoids claiming more certainty about model behavior than anyone has. A draft that sounds confident but skips a stakeholder's constraint is the weaker one.

Overstated versus honest wording

Replace guarantees with conditions and measured claims. Each common overstatement below has a more accurate revision that names its condition.

Overstated ADR claims and honest revisions

Topic	Overstated claim	Honest revision
Prompt injection	The model resists injection, so no control is needed	Model behavior is not a security control
Cost per task	Published figures set our budget	Figures are directional; measure on our workload
Sandbox settings	Settings remove data-leak risk	Settings lower risk; keep must-stay data out
Host limits	Limiting hosts stops credential misuse	An allowed host can still receive a credential

EXAM TRAP

Claude's resilience to prompt injection does not justify skipping controls. Anthropic states model behavior is not a security control.

Plan for model change

A pinned model ID keeps the same model for its lifetime. Even so, infrastructure updates can cause small behavior shifts. The router, safety classifiers and sampling logic can change around a fixed model. Each model ID also has its own deprecation and retirement schedule.

EXAM TRAP

A fixed model ID does not mean identical behavior forever. Serving infrastructure around it can still change outputs slightly.

Lifecycle and platform risk

Retired models fail; deprecated ones still work for now. Anthropic gives at least **60** days' notice before retiring a public model with active deployments. Partner platforms such as Bedrock set their own retirement dates and regional pricing. So an ADR should note which schedule applies.

Evaluate before you switch

A use-case-specific evaluation set comes first. Before changing models, measure on your own workload. Tuning effort within one model is often a better lever than switching. Organization settings can still block a server tool the API lists as supported.

Key takeaways

- State the residual risk: Managed Agents sandbox settings lower risk but do not remove it.
- Model behavior is not a security control, however resilient Claude is.
- Treat published cost and intelligence figures as directional, and measure your own workload.
- Build a use-case-specific evaluation set before deciding to change models.

Sources: Claude Certification Program Exam guide Claude Certified Architect – Professional Exam Guide (opens in a new tab) · Optimizing for cost and intelligence - Claude Platform Docs (opens in a new tab) · Security risks and sandbox boundaries - Claude Platform Docs (opens in a new tab) · Model IDs and versioning - Claude Platform Docs (opens in a new tab) · Model deprecations - Claude Platform Docs (opens in a new tab) · Choosing the right model - Claude Platform Docs (opens in a new tab) · Pricing - Claude Platform Docs (opens in a new tab) · Models overview - Claude Platform Docs (opens in a new tab)

Operational Handoff Completeness

Start here

THINK FIRST

When a Claude system is handed to an operations team, which missing assumption, credential or runbook step would stop them on day one?

Why it matters: Questions describe a handoff package and ask which missing assumption, dependency or runbook element blocks the receiving team, and what to write down to close it.

A handoff breaks on what the builders knew but did not write down. Record who owns each credential and when it expires, and which limits apply. Then add the recovery steps for a key or token that stops working.

Key terms

Term	Meaning
Service account	An admin-created identity for shared or automated workloads, so they do not depend on one person.
Admin API key	A key with the `sk-ant-admin` prefix, passed in the `x-api-key` header; only admins can provision one.
Workload Identity Federation	A way to mint an org:admin token for CI and other non-interactive work without an interactive login.
Content mode	The setting that decides whether telemetry spans carry only metadata or full prompts and responses.

Credential ownership

Personal keys die with their owner's membership. When a personal key is shared, it still acts as one person, and it breaks once that person leaves. Production services and CI need a service account, created by an organization admin, so they have their own identity.

What keeps each kind of API key working

Key type	Stops working or is archived when	Note
Personal key	Its user is removed from the organization	Unsafe to share for CI or production
Service account key	Its service account is archived	Keeps working if its creator is removed
Claude Code workspace key	The member who created it is removed	Bound to its creator
Any key created for a workspace	Its workspace is archived	The workspace is deactivated too

Expiry is a runbook item

Key expiration is fixed at creation and cannot change. An expired key returns a **401** `authentication_error` and cannot be reactivated, so the runbook must say how to create a new one. For a key with a lifetime of **14** days or more, Anthropic emails its creator **7** days ahead.

EXAM TRAP

The Admin API cannot create API keys. Keys are created in the Claude Console; the Admin API only reads, renames and changes key status.

Admin access dependencies

Some Admin API endpoints need an org:admin OAuth token. An Admin API key covers most endpoints. The service-account, federation-issuer and federation-rule endpoints take only the OAuth token. Interactive tokens are short-lived, so document how to refresh the token when requests return **401**.

EXAM TRAP

Federation is not automatically secure. A long-lived secret earlier in the chain, such as a static cloud credential able to mint identity provider (IdP) tokens, can still weaken it.

Workspace assumptions

Write down the workspace each key needs. A key not scoped to a workspace must send `anthropic-workspace-id` on every request. Organization users and developers must be added to each workspace explicitly. Workspace limits can hold usage below the organization's limits but never raise them.

Limits that surprise operators

Some caps are organization-wide or not configurable. The cap of **10** active MCP tunnels counts across the whole organization. Invites expire after **21** days, with no setting to change it. A Default Workspace spend

limit might not count or stop some usage there.

Telemetry the team will see

The default content mode is metadata only. Prompts and responses appear only in Full content mode. With a HIPAA, FERPA or K-12 restriction, destinations export only metadata whatever the mode. API tracing also needs the Admin permission in the Console.

Key takeaways

- Run production and CI on service accounts, not shared personal keys.
- Record every key's expiry, since it cannot be changed and expired keys cannot be revived.
- Note which Admin API endpoints need an org:admin OAuth token instead of an Admin API key.
- Document workspace IDs and limits, because workspace limits cannot exceed the organization's.

Sources: Authentication - Claude Platform Docs (opens in a new tab) · Workspaces - Claude Platform Docs (opens in a new tab) · Admin API - Claude Platform Docs (opens in a new tab) · OpenTelemetry tracing for the Claude API - Claude Platform Docs (opens in a new tab)

Domain 7 | 8% of the exam | 12 min read

Developer Productivity and Operational Enablement

Overview

This domain is about 8% of the exam, roughly 6 of its 70 questions.

Every point below comes from Anthropic's official documentation; each section lists its sources.

Packaging and Distributing Claude Code Workflows

Start here

THINK FIRST

How do you roll out a Claude Code workflow that one team has proven to every team, safely, even when it runs unattended?

Why it matters: Questions describe a workflow ready to spread and ask which plugin part, distribution path or managed control fits.

Package the workflow as a plugin, list it in a marketplace, and install it on every machine through managed settings. Managed settings outrank every other scope, so the controls that must always hold go there.

Key terms

Term	Meaning
Plugin marketplace	A directory or repository whose `.claude-plugin/marketplace.json` lists plugins and where to fetch each one.
Managed settings	The policy file or server-delivered policy that every machine in the organization reads.
Plugin hook	A hook shipped in the plugin's `hooks/hooks.json`; use it for a rule that must hold every time.
Side-loading	Loading plugins or agents from startup flags such as `--plugin-dir` instead of from a marketplace.

What goes in a plugin

Hooks and MCP servers each have a set file at the plugin root. Hooks and MCP servers are added at the plugin level. A plugin's agent files cannot add them on their own.

Where plugin parts live and what to watch for

Part	Where it goes	Watch for
Hooks	`hooks/hooks.json`, under a top-level `hooks` key	Same shape as the hooks object in `settings.json`
MCP servers	`.mcp.json` at the plugin root	Same shape as a project `.mcp.json`
Subagent files	The plugin's agent files	`permissionMode`, `hooks`, `mcpServers` and `initialPrompt` are ignored
`CLAUDE.md`	The plugin root	Not loaded as project context

EXAM TRAP

A `CLAUDE.md` at the plugin root is not loaded. A rule that must hold every time belongs in a plugin hook, not a skill.

Publishing through a marketplace

A marketplace lists plugins and where to fetch each one. Its `marketplace.json` needs a `name`, an `owner` and a `plugins` array. Each plugin entry needs a `name` and a `source`. The install id is the entry's name, an `@`, and the marketplace name. Pin archive sources with `sha256` so a changed download is refused.

EXAM TRAP

No `marketplace.json` field turns on background auto-update. It is off by default, and a user or an admin must enable it.

Rolling out to the fleet

Set `extraKnownMarketplaces` and `enabledPlugins` together. In managed settings, the first registers the marketplace on each machine, and the second names the plugins to install and enable. A plugin force-enabled in managed settings keeps loading even if a user disables it. For one repository's contributors only, set both keys in that repository's `.claude/settings.json`.

Unattended and CI runs

Plugins install in the background in `-p` and CI runs. So a plugin can be missing from the first turn. Set `CLAUDE_CODE_SYNC_PLUGIN_INSTALL=1` to make the run wait for the install. Runners that cannot clone at runtime can use a plugins directory built in, named by `CLAUDE_CODE_PLUGIN_SEED_DIR`.

EXAM TRAP

A CI run does not wait for plugins by default. Without the sync variable, the first query can run before the plugin is installed.

Locking permission rules

Managed permission rules can override every local rule. Setting `allowManagedPermissionRulesOnly` to `true` makes Claude Code ignore permission rules from user, project and local files and from `--allowedTools`. Without it, users' allow rules merge with yours.

Example: Managed settings that deny two file reads, turn off bypass mode and keep only managed permission rules.

```
{
  "permissions": {
    "deny": ["Read(./.env)", "Read(./secrets/**)"],
    "disableBypassPermissionsMode": "disable"
  },
  "allowManagedPermissionRulesOnly": true
}
```

- `"allowManagedPermissionRulesOnly": true`: ignores permission rules from user, project and local files
- `"disableBypassPermissionsMode": "disable"`: turns off bypass mode

EXAM TRAP

Managed allow rules do not replace users' allow rules on their own. The two merge unless `allowManagedPermissionRulesOnly` is set.

Controlling plugin and MCP sources

By default, anyone can connect any MCP server they choose. Managed keys narrow which sources teams may use. `deniedMcpServers` also filters managed servers, so a matching one will not load.

Managed controls over MCP servers and plugin sources

Setting	What it controls	Watch for
`managed-mcp.json`	The only MCP servers users can use	Blocks plugin servers and `--mcp-config` servers
`managedMcpServers`	MCP servers delivered through managed settings	No exclusive control
`strictKnownMarketplaces`	Allowlist of marketplace sources	`[]` blocks every source, even the official one
`disableSideloadFlags`	Side-loading at startup	Rejects `--plugin-dir`, `--plugin-url` and `--agents`

EXAM TRAP

`managed-mcp.json` cannot be delivered through server-managed settings. It is a standalone file, so use `managedMcpServers` to deliver servers that way.

Endpoint versus server-managed

Endpoint-managed settings are harder for users to change. On MDM-enrolled devices, the OS can protect the settings file from user changes. But they do not reach cloud sessions, so add server-managed settings too. Those need Claude for Teams or Enterprise, and cached copies expire **24** hours after the last fetch.

EXAM TRAP

Group configurations do not merge with each other or the organization-wide one. Each must hold every setting that group needs.

Failing closed without policy

Fail-closed startup refuses to run without fresh policy. The CLI blocks at startup until server-managed settings are freshly fetched, and exits if the fetch fails. Hooks delivered this way also show interactive users a security approval dialog, because hooks run shell commands.

Key takeaways

- Ship the workflow as a plugin in a marketplace, then install it with `extraKnownMarketplaces` and `enabledPlugins`.
- Managed settings take precedence over every other scope, so central controls belong there.
- Put a rule that must always hold in a plugin hook, not a skill or a plugin `CLAUDE.md`.
- In CI, set `CLAUDE_CODE_SYNC_PLUGIN_INSTALL=1` so plugins are ready before the first query.
- `allowManagedPermissionRulesOnly` stops user, project and local permission rules from applying.

Sources: Create a marketplace - Claude Code Docs (opens in a new tab) · Manage Claude Code plugins for your organization - Claude Code Docs (opens in a new tab) · Add components to a plugin - Claude Code Docs (opens in a new tab) · Host and maintain a marketplace - Claude Code Docs (opens in a new tab) · Deploy managed settings - Claude Code Docs (opens in a new tab) · Configure server-managed settings - Claude Code Docs (opens in a new tab) · Control MCP server access for your organization - Claude Code Docs (opens in a new tab)

Agent SDK and Claude Code Configuration

Start here

THINK FIRST

How would you set up Claude Code or the Agent SDK so a team's automation stays fast but inside its safety guardrails?

Why it matters: Questions describe a team workflow and its guardrails and ask which permission mode, hook, subagent or CI setting fits.

Permission modes decide which tool calls run without a prompt, from reads and pre-approved tools up to everything. Deny rules block in every mode, and a PreToolUse hook can allow, deny, ask or defer a call. Setting sources and flags such as `--bare` decide which settings and hooks a session loads.

Key terms

Term	Meaning
Permission mode	The session-wide baseline, such as `acceptEdits` or `dontAsk`, for which tool calls run without a prompt.
Deny rule	A permission rule that blocks a tool in every mode, even `bypassPermissions`.
PreToolUse hook	A hook that runs before a tool call and can allow, deny, ask or defer it.
`canUseTool` callback	An Agent SDK callback that confirms calls no mode or rule has already approved.
`settingSources`	The Agent SDK option that lists which settings files a session loads.

Pick the permission mode

Each mode trades prompts for automation differently. Match the mode to where the session runs and how much risk it may take.

Claude Code permission modes compared

Mode	Runs without asking	Note
`acceptEdits`	Reads, file edits, `mkdir`, `touch`, `mv`, `cp`	Calls it approves skip `canUseTool`
`plan`	No edits until you approve a plan	Exploring a codebase before changes
`dontAsk`	Reads in working directories and pre-approved tools	Locked-down CI and scripts
`bypassPermissions`	Everything	Isolated containers and VMs only

EXAM TRAP

`dontAsk` does not approve everything quietly. It denies any call that would need a prompt, so only pre-approved tools and reads run.

allowedTools pre-approves, not restricts

`allowedTools` **skips prompts for listed tools only.** Claude can still call unlisted tools, and the permission mode decides those calls. Calls approved by a mode or an allow rule never reach `canUseTool`, so checks placed there are skipped. A matching ask rule, though, still sends the call to `canUseTool`, even in bypass mode.

EXAM TRAP

Putting a safety check in `canUseTool` is not enough on its own. Any tool already approved by `acceptEdits` or an allow rule bypasses it silently.

Hooks and matchers

A hook without a matcher fires for every event of its type. A matcher such as `Write|Edit` is tested against the tool name. A PreToolUse hook can answer allow, deny, ask or defer; defer ends the turn with `tool_deferred` for a later resume. Set `onFailure` to block so a failing hook stops the action. Filesystem hooks also fire inside subagents.

Which settings a session loads

Project rules count only when project settings are enabled. An explicit `settingSources` must include project for `.claude/settings.json` rules to count. An empty list loads no user, project or local files. For most single values, the SDK settings option beats those files; only managed policy ranks higher.

Running headless in CI

`--bare` **skips auto-discovery for a faster, cleaner start.** With `--bare`, `claude -p` skips hooks, skills, subagents, plugins, MCP servers, memory and CLAUDE.md discovery. Without it, project hooks and `.mcp.json` servers load even in an untrusted folder, with no trust dialog. Container filesystem state is lost on restart or scale-down.

> **EXAM TRAP**
> A headless run does not show a workspace trust dialog. Without `--bare`, a cloned repo's hooks and MCP servers run automatically.

Caps, fallbacks and environment

`maxTurns` **of 0 means unlimited;** `maxBudgetUsd` **of 0 fails.** The CLI rejects a zero budget at startup, so the session never runs. Hitting a cap returns `error_max_turns` or `error_max_budget_usd`. A `fallbackModel` steps in when the primary is overloaded. In TypeScript, `env` replaces the environment; in Python it merges.

Key takeaways

- Deny rules block a tool in every permission mode, including `bypassPermissions`.
- `allowedTools` pre-approves listed tools but never restricts Claude to them.
- Auto-approved calls never reach `canUseTool`, so do not rely on it alone.
- Use `dontAsk` for locked-down CI and keep `bypassPermissions` for isolated containers.
- A non-fork subagent sees only its prompt string, so pass every needed detail.

Sources: Choose a permission mode - Claude Code Docs (opens in a new tab) · Intercept and control agent behavior with hooks - Claude Code Docs (opens in a new tab) · Configure permissions - Claude Code Docs (opens in a new tab) · Run Claude Code programmatically - Claude Code Docs (opens in a new tab) · Configure your agent - Claude Code Docs (opens in a new tab) · Use Claude Code features in the SDK - Claude Code Docs (opens in a new tab) · Hosting the Agent SDK - Claude Code Docs (opens in a new tab) · Subagents in the SDK - Claude Code Docs (opens in a new tab)

Accountability for AI-Generated Code

Start here

THINK FIRST

When a team lets Claude write and review code, which controls keep a person accountable for what gets merged?

Why it matters: Questions describe a team's AI-assisted review process and ask which control keeps human ownership of the code.

Claude Code Review informs the merge decision but never makes it. It cannot approve, block or push, and a clean review is no guarantee. Branch protection can require each finding's thread to be resolved before merging, and only an Owner turns the service on or changes its scope.

Key terms

Term	Meaning
Finding severity	Important should be fixed before merging, Nit is minor and non-blocking, Pre-existing was not caused by the pull request.
REVIEW.md	A file of review instructions read from the base or default branch, not the pull request.
Branch protection	Repository rules, such as required conversation resolution, that gate what can merge.
Ultrareview	A review that runs only when a user starts it with `/code-review ultra`; every finding is reproduced and verified.

Review informs; humans decide

Code Review never approves or blocks a pull request. Its check run never ends as a failure, so it cannot fail a required check. Inside a review, repository access is read-only and no commits are pushed. The merge decision stays with the team.

Make findings count

Findings can be ignored unless branch protection says otherwise. A pull request can merge with open findings unless conversation resolution is required. Replying to a finding does not dismiss it. Dismissing one without a code change means resolving the thread on GitHub.

EXAM TRAP

Answering a finding with a comment does not close it. The thread must be resolved, and only required resolution forces anyone to act.

Protect the review rules

REVIEW.md is read from the base branch, never the pull request. So an author cannot weaken the rules in their own change. Rules that lower severity, exclude paths or cap nits apply only from a protected default branch. A ruleset counts as protection when it restricts updates or requires a fresh approval.

Who controls the service

Only an Owner turns on Code Review and sets its scope. Other developers must ask an Owner to adjust the level, when reviews run, or which repositories are covered. A comment command starts a review only from someone with write access.

Managed review versus local commands

The managed service and local commands behave differently. Know which one a team relies on before trusting its output.

Managed Code Review and local review commands compared

Feature	Managed Code Review	Local `/code-review`
Instructions read	CLAUDE.md and REVIEW.md	CLAUDE.md only
Where it runs	Anthropic infrastructure; no workflow files	A Claude Code session on the user's own machine
Edits	None: read-only, no commits	Through the user's permission mode

What review covers and skips

By default it hunts correctness bugs, not style. Formatting and missing tests are out of scope by default. Breaking a CLAUDE.md rule becomes a finding. Automatic reviews skip draft and fork pull requests. Organizations with Zero Data Retention or HIPAA settings cannot use it.

Key takeaways

- Code Review never approves or blocks a pull request; the team decides the merge.
- Require conversation resolution in branch protection so findings cannot be ignored.
- REVIEW.md comes from the base branch, so a pull request cannot rewrite its own rules.
- A review with no findings does not prove the code is correct or secure.

Sources: Code Review - Claude Code Docs (opens in a new tab) · Read and request reviews on a pull request - Claude Code Docs (opens in a new tab) · Set up Code Review - Claude Code Docs (opens in a new tab) · Customize Code Review - Claude Code Docs (opens in a new tab) · Find bugs with ultrareview - Claude Code Docs (opens in a new tab) · Review locally with /code-review - Claude Code Docs (opens in a new tab)

MockCert

Take a Claude Certified Architect – Professional mock exam the way the real one feels

5 timed mock exams · 70 questions in 120 minutes each · one is free

- Free to start: study notes, explained quizzes, flashcards and the first mock exam
- Timed mock exams built to the official exam guide
- A verifiable Certificate of Assessment when you pass a premium mock

Explore Claude Certified Architect – Professional on MockCert

Open it on your phone

mockcert.com/ccar-p/

Scan the code with your phone's camera.