Índice

Guía de Estudio — Claude Certified Architect · Foundations (CCAR-F)

> **Qué mide este examen:** la **mecánica** y los **conceptos**. Foundations pregunta *"¿qué es X y cómo funciona?"*. Domina cómo se comportan agentes, tools, prompts, Claude Code y el contexto. El criterio aplicado de arquitecto es del Professional; acá lo que cuenta es entender bien las piezas.
>
> **Formato:** ~60 ítems · 120 min · corte 720/1000 · single y multi-response (cada ítem dice cuántas marcar).
>
> **Cómo usar esta guía:** cada concepto trae **Qué es → Cómo funciona → Cuándo aplica → Cuándo NO / diferencia → Trampa de examen**. Lee la diferencia con el concepto vecino: ahí es donde te confunden.

Dominio 1 — Agentic Architecture & Orchestration

Este dominio es sobre **cuándo usar un agente y cómo estructurarlo**. La idea central: hay un abanico de patrones de menos a más autónomos, y elegir el correcto es la habilidad medida.

1.1 Augmented LLM (LLM aumentado)

Qué es: una sola llamada al modelo, potenciada con ayudas externas — retrieval (traer datos), tools (ejecutar acciones) o memoria. **No hay loop de decisión.**

Cómo funciona: le das al modelo el contexto necesario (por ejemplo, el fragmento relevante de una base de conocimiento) y una tarea, y devuelve la salida en una pasada. Es la unidad básica; los demás patrones se construyen encima de esto.

Cuándo aplica: - Responder una pregunta usando un documento recuperado ("¿cuál es la política de devoluciones?"). - Extraer campos de un texto a JSON (`{nombre, fecha, monto}`). - Traducir, resumir o reformular un texto.

Cuándo NO / diferencia: si la tarea necesita **varios pasos con decisiones**, ya no es Augmented LLM. Señal para reconocerlo: podés describir la tarea como "entra X → sale Y" **sin ningún "y luego, dependiendo de…"**.

Trampa: te describen algo con retrieval o una tool y te tientan con "es un agente". Si es una sola transformación con contexto al lado, es Augmented LLM, no agente.

1.2 Workflow (flujo orquestado)

Qué es: varios pasos de LLM encadenados donde **el código decide el orden**. El camino es fijo y conocido de antemano.

Cómo funciona: vos, el desarrollador, escribís la secuencia. Sub-patrones: - **Prompt chaining:** la salida de un paso alimenta al siguiente (clasificar → extraer → generar). - **Routing:** clasificás la entrada y la mandás a la rama correcta (¿spam/ofensivo/ok? → rama según resultado). - **Parallelization:** varias llamadas en paralelo y agregás (resumir ventas + soporte + web → fusionar).

Cuándo aplica: - Procesar facturas: clasificar tipo → extraer campos → validar → aprobar/rechazar. - Moderación de contenido con ramas fijas. - Cualquier proceso donde **podés dibujar el diagrama de flujo antes de correr nada**.

Cuándo NO / diferencia con Agentic: en workflow **el código** manda el orden; en agentic **el modelo** lo decide. Si podés enumerar los pasos de antemano, es workflow. Si el orden emerge en runtime, es agente.

Por qué preferirlo: predictibilidad, auditoría paso a paso, debug simple. Brilla cuando los pasos son idénticos en cada pedido y la reproducibilidad importa.

Trampa: una tarea que "suena inteligente" pero tiene flujo fijo → la respuesta correcta es **workflow, no agente**. Meter un agente donde alcanza un workflow agrega costo, latencia e imprevisibilidad sin beneficio.

1.3 Agentic (agente autónomo)

Qué es: el **modelo decide** qué pasos dar, qué tools usar y cuándo parar, en un **loop planificar → actuar → observar**.

Cómo funciona: le das el objetivo y las tools; el agente ejecuta, observa el resultado, decide el siguiente paso, y repite hasta terminar. Puede tomar rutas distintas en cada ejecución.

Cuándo aplica: - "Investiga la competencia y dame un reporte" — no sabés cuántas búsquedas hará ni en qué orden. - Agente de debugging: lee error → hipótesis → cambio → corre test → repite hasta pasar. - "El servidor está lento, averigua por qué" — la ruta depende de lo que va encontrando.

Cuándo NO / diferencia: si podés escribir los pasos de antemano, NO uses agente (es workflow). El agente se justifica solo cuando **la ruta a la respuesta recién aparece durante la ejecución**.

Trampa: confundir "tarea compleja" con "necesita agente". Complejidad ≠ imprevisibilidad. Un cálculo complejo pero de pasos fijos sigue siendo workflow.

1.4 Orquestador / Multi-agente

Qué es: un agente **coordinador** que reparte trabajo a varios **subagentes** especializados y luego **sintetiza** los resultados.

Cómo funciona: el coordinador descompone el problema, lanza subagentes (a menudo en paralelo), cada uno trabaja su parte con su propio contexto, y el coordinador junta todo en una salida.

Cuándo aplica: - Investigación amplia: dividir "IA en industrias creativas" en música, escritura, cine, arte visual → un subagente por subtema en paralelo → sintetizar. - Revisión de PR grande: subagentes de seguridad, performance y estilo, cada uno con su lente. - Migración masiva: un subagente por archivo, en paralelo.

Condición para justificarlo: las subtareas piden **especialización distinta** + son **independientes** + pueden correr **en paralelo**. Si no cumplen esto, multi-agente es complejidad de más → usá workflow.

Concepto crítico — contexto de subagente: los subagentes **NO heredan** la conversación del coordinador, ni su memoria, ni sus variables. **Todo lo que el subagente necesita debe ir explícito en su prompt.** Empiezan con contexto limpio.

Trampa clásica: un reporte salió incompleto (cubrió solo "arte visual" cuando el tema era más amplio). La causa raíz está en la **descomposición del coordinador** (demasiado angosta, omitió dominios enteros), NO en la búsqueda ni en la síntesis. Aprendé a ubicar el fallo en la capa correcta.

1.5 Terminación del loop agéntico

Qué es: cómo sabe un agente que terminó.

Cómo funciona: el criterio correcto es `stop_reason: end_turn` — el modelo dejó de pedir tools y dio su respuesta final. Un **cap de iteraciones** (ej. máximo 10) es una **red de seguridad**, no la condición de terminación.

Cuándo NO / diferencia: buscar una palabra como "done" en la respuesta es frágil. El cap de iteraciones existe para evitar loops infinitos, pero no es la señal de "terminé bien".

Trampa: confundir el cap de iteraciones con la condición de terminación. El cap es el freno de emergencia; `end_turn` es la meta.

1.6 Ejecución paralela de subagentes

Qué es: correr varios subagentes al mismo tiempo para bajar latencia.

Cómo funciona: emitís **varias llamadas a la tool de subagente en UNA sola respuesta** del coordinador. Eso las lanza en paralelo.

Cuándo NO / diferencia: una llamada por turno, en turnos separados, es **secuencial** (lento). Subir `max_tokens` o usar la batch API no las paraleliza.

Trampa: creer que llamar un subagente por turno es "paralelo". Paralelo = múltiples llamadas en la misma respuesta.

Dominio 2 — Tool Design & MCP Integration

Sobre cómo diseñar herramientas que el modelo use bien, manejar errores, e integrar sistemas externos vía MCP.

2.1 Descripciones de tools

Qué es: el texto que le dice al modelo qué hace cada tool y cuándo usarla.

Cómo funciona: una buena descripción incluye formato de input, ejemplos de queries, edge cases y **cuándo usar esta tool vs otra parecida**.

Cuándo aplica el fix: si el agente elige la tool equivocada (llama `get_customer` cuando el usuario pregunta por pedidos, y ambas aceptan IDs similares), el **primer paso** es **expandir las descripciones**, no meter capas nuevas ni fusionar tools.

Cuándo NO / diferencia: agregar few-shot examples, construir un router de keywords, o fusionar en una sola tool son soluciones posteriores o peores. Se ataca primero la causa raíz: descripciones pobres.

Trampa: saltar directo a "fusiona las dos tools" o "agrega ejemplos". La causa raíz suele ser la descripción.

2.2 Número de tools por agente (capability bloat)

Qué es: cuántas herramientas darle a un agente.

Cómo funciona: demasiadas tools **degradan la precisión de selección**. Regla práctica: acotá cada agente a las ~4-5 tools que su rol necesita.

Cuándo aplica: un agente con 18 (o 45) tools que elige mal, con latencia alta y tools nunca usadas → es capability bloat. Fix: sacar tools muertas y solapadas, partir en agentes por dominio detrás de un router, y usar progressive disclosure.

Cuándo NO / diferencia: "más tools = más capacidad" es falso. Más allá de cierto punto, más tools = peor desempeño.

Trampa: creer que el problema es solo de descripciones cuando el catálogo ya es enorme. Con 45 tools el problema es de **cantidad**, no solo de texto.

2.3 Manejo de errores de tools

Qué es: qué devuelve una tool cuando falla.

Cómo funciona: devolvé **metadata estructurada**: `errorCategory` (transient / validation / permission), `isRetryable` (boolean), y una descripción legible. Así el agente decide inteligentemente: reintentar (transient), corregir el input (validation), o escalar (permission).

Cuándo NO / diferencia: - "Operation failed" genérico → el agente no sabe qué hacer. - Devolver éxito silencioso con resultado vacío → el agente cree que funcionó. - Lanzar excepción que mata todo el workflow → pierde la chance de recuperarse.

Trampa: elegir el error genérico o el fallo silencioso. La respuesta correcta siempre da al agente información para **recuperarse**.

2.4 MCP (Model Context Protocol)

Qué es: un estándar para conectar Claude a herramientas, datos y sistemas externos vía un servidor. El "USB-C" de las integraciones.

Cómo funciona: cada equipo mantiene su servidor MCP una vez, y cualquier app/agente lo consume de forma estandarizada y descubrible.

Cuándo aplica: muchos sistemas, muchas apps consumidoras, propiedad descentralizada, herramientas que rotan seguido. Ese es el caso canónico de MCP.

Cuándo NO / diferencia: - **API/CLI directo:** integración puntual, a medida, un solo consumidor. - **Agent-to-agent:** delegás a otro agente autónomo, no a una función.

Trampa: confundir MCP con "llamar una API". MCP es el protocolo estandarizado reutilizable; una API directa es plomería a medida.

2.5 Secretos en configuración MCP

Qué es: cómo manejar tokens/credenciales sin filtrarlos.

Cómo funciona: `.mcp.json` a nivel de proyecto, **versionado**, con **expansión de variables de entorno** (ej. `${GITHUB_TOKEN}`). El archivo referencia la variable; el secreto real vive en el entorno, nunca en git.

Cuándo NO / diferencia: hardcodear el token en `.mcp.json`, pegarlo en `CLAUDE.md`, o guardarlo en texto plano en el config personal = todos filtran el secreto.

Trampa: cualquier opción que ponga el valor del token dentro de un archivo versionado.

Dominio 3 — Claude Code Configuration & Workflows

Sobre configurar Claude Code para equipos y usar los modos correctamente.

3.1 Comandos compartidos (slash commands)

Qué es: comandos reutilizables como `/review`.

Cómo funciona: - **Para todo el equipo:** `.claude/commands/` **dentro del repo**, versionado. Cada dev los recibe al clonar/pull. - **Personales:** `~/.claude/commands/` en el home. Solo ese dev los tiene.

Trampa: "quiero que TODOS lo tengan automáticamente al clonar" → repo `.claude/commands/`, NO el home.

3.2 Instrucciones compartidas (CLAUDE.md)

Qué es: instrucciones persistentes para el proyecto.

Cómo funciona: - **Compartidas:** `CLAUDE.md` en el repo (versionado). - **Personales:** `~/.claude/CLAUDE.md` (user-level, **no se versiona**, solo tuyo).

Diferencia clave: si "a un teammate nuevo no le llegan las instrucciones que todos los demás tienen", es porque están en el `~/.claude/` **personal** de cada uno, no en el repo. Fix: mover al `CLAUDE.md` del repo.

Trampa: confundir user-level con project-level. User-level no viaja con el clone.

3.3 Reglas por patrón de archivo (rules + glob)

Qué es: aplicar convenciones según el tipo de archivo, sin importar la carpeta.

Cómo funciona: un archivo de rule con frontmatter y un glob, ej. `**/*.test.tsx`. La convención se aplica a todos los tests estén donde estén (incluso al lado del componente, como `Button.test.tsx` junto a `Button.tsx`).

Cuándo NO / diferencia: - Poner todo en el `CLAUDE.md` raíz → no se aplica por patrón, se mezcla. - Un `CLAUDE.md` en cada subdirectorio → inmanejable. - Una skill por tipo de código → no es el mecanismo para convenciones automáticas por glob.

Trampa: la respuesta es la rule con glob cuando la convención depende del **tipo de archivo disperso**, no de la carpeta.

3.4 Plan mode vs ejecución directa

Qué es: dos modos de trabajar una tarea.

Cómo funciona: Plan mode explora y diseña antes de tocar código. Úsalo para tareas grandes con decisiones caras de revertir (ej. reestructurar un monolito en microservicios — decenas de archivos, límites de servicio).

Cuándo NO / diferencia: ejecución directa para cambios triviales o cuando ya tenés instrucciones completas y el alcance es chico. "Ejecuta directo y cambia a plan si se complica" es peor que planear desde el inicio cuando ya sabés que es complejo.

Trampa: elegir ejecución directa para una reestructuración arquitectónica. Decisiones caras de revertir → plan primero.

3.5 context: fork en una skill

Qué es: correr una skill en un contexto aislado.

Cómo funciona: la skill corre en un sub-agente separado; **solo vuelve el resultado**, sin ensuciar la conversación principal con el detalle intermedio.

Cuándo NO / diferencia: no crea una rama de git, no forkea el servidor MCP, no duplica el CLAUDE.md. Es aislamiento de **contexto**.

Trampa: interpretaciones literales de "fork" (git branch, fork de proceso). Es aislamiento de contexto conversacional.

3.6 Claude Code en CI (modo no interactivo)

Qué es: correr Claude Code en un pipeline automatizado.

Cómo funciona: `claude -p "..."` (print / non-interactive). Corre y sale, sin esperar input humano.

Cuándo NO / diferencia: si el job "se cuelga esperando input", falta el `-p`. No existe un `--batch` para esto; no basta redirigir stdin.

Trampa: cualquier opción que no sea el modo print para CI que cuelga.

Dominio 4 — Prompt Engineering & Structured Output

Sobre guiar el comportamiento del modelo y garantizar salidas parseables.

4.1 Precisión de un juicio (criterio categórico)

Qué es: cómo bajar los falsos positivos de una tarea de evaluación (ej. code review que marca de más).

Cómo funciona: dale **criterio categórico verificable**: "marcá un comentario solo cuando el comportamiento que declara contradice el código". Un umbral objetivo que el modelo puede aplicar sin adivinar.

Cuándo NO / diferencia: - "Sé conservador" / "solo alta confianza" → subjetivo, el modelo no sabe dónde está el umbral. - Pedirle que autoevalúe su confianza → poco confiable. - Cambiar a un modelo más grande → no arregla un criterio mal definido.

Trampa: las opciones "vagas" (conservador, confianza) tientan. La correcta es la que da un criterio concreto y verificable.

4.2 Técnicas de prompting: zero-shot, few-shot, chain-of-thought

Qué es: las tres técnicas base.

Cómo funciona y cuándo cada una: - **Zero-shot:** tarea simple, sin ejemplos. Default cuando la instrucción basta. - **Few-shot:** das 2-5 ejemplos de entrada→salida. **Para enseñar formato o patrón** cuando describir en prosa no alcanza. "Mostrar supera a describir" en cumplimiento estricto de formato. - **Chain-of-thought (razonamiento paso a paso):** para tareas de **varios pasos** (matemática, lógica, análisis). Gana sus tokens ahí; en extracción simple solo agrega costo y latencia a cambio de nada.

Diferencia clave: few-shot enseña **formato/patrón**; chain-of-thought habilita **razonamiento**. No son intercambiables.

Trampa: aplicar chain-of-thought a una extracción trivial (desperdicio) o creer que few-shot resuelve costo (no; enseña patrón).

4.3 JSON garantizado (tool_use con schema)

Qué es: cómo obtener JSON válido siempre.

Cómo funciona: usá `tool_use` con un **JSON schema**. La validación pasa en la capa de la llamada a tool, y el modelo reintenta si no cumple. No hay errores de sintaxis.

Cuándo NO / diferencia: pedir JSON en el prompt y parsear el texto, bajar temperatura a 0, o post-procesar con regex → ninguno **garantiza** JSON válido.

Trampa: elegir "temp=0" o "pide JSON y parsea". La garantía viene del schema en tool_use.

4.4 Evitar valores inventados (campos opcionales)

Qué es: cómo impedir que el modelo invente cuando falta un dato.

Cómo funciona: hacé **opcionales/nullable** los campos que pueden faltar. Así el modelo devuelve `null` en vez de fabricar.

Cuándo NO / diferencia: hacer todo required fuerza al modelo a inventar; poner un valor default falso contamina; reintentar hasta que lo llene solo produce alucinación.

Trampa: "hazlo required" o "pon un default". La correcta permite `null` como respuesta legal.

4.5 Qué garantiza (y qué NO) un schema estricto

Qué es: los límites de la validación por schema.

Cómo funciona: - **Garantiza:** sintaxis JSON válida y estructura conforme (campos correctos, tipos correctos). - **NO garantiza:** corrección **semántica** — que los números sumen bien, que el campo tenga el valor correcto, que la info sea **factualmente verdadera**.

Diferencia clave: schema = **forma**, no **verdad**. Un JSON perfectamente válido puede tener datos falsos.

Trampa (multi-response típico): te piden qué garantiza y qué no. Garantiza sintaxis; NO garantiza que las cosas sumen ni que sea verdad.

4.6 Retry con feedback: cuándo sirve

Qué es: reintentar una extracción que falló validación, pasándole el error.

Cómo funciona: sirve cuando el fallo es de **formato o estructura** — el modelo puede corregir con el feedback.

Cuándo NO / diferencia: si el dato **no existe en la fuente**, el retry solo produce alucinación. Reintentar no crea información que no está.

Trampa: creer que el retry siempre ayuda. Solo ayuda con errores de forma, no con información ausente.

Dominio 5 — Context Management & Reliability

Sobre manejar la ventana de contexto y hacer el sistema confiable en el tiempo.

5.1 Proteger datos exactos de la compresión

Qué es: evitar perder montos, fechas, IDs en conversaciones largas que se resumen.

Cómo funciona: extraé los hechos transaccionales a un bloque persistente ("case facts") que va en **cada** prompt, **fuera** del historial que se resume. La compresión no puede perder lo que nunca comprimió.

Cuándo NO / diferencia: confiar en que el resumen recuerde los detalles exactos → los pierde. Subir `max_tokens` no evita la pérdida. Reiniciar la sesión cada turno pierde todo.

Diferencia con truncar: truncar **tira** información; extraer **protege** la crítica sacándola del área que se comprime. Son opuestos.

Trampa: "confía en el resumen" o "sube max_tokens". La correcta saca los hechos duros aparte.

5.2 Atención posicional (lost in the middle)

Qué es: el modelo presta menos atención a lo que está enterrado en el medio del contexto.

Cómo funciona: reglas críticas a mitad de un contexto largo se pierden. Moverlas **al inicio o al final**, separadas estructuralmente del material de referencia, es el fix de mayor palanca.

Cuándo NO / diferencia: enterrar la instrucción clave entre documentos largos = mal. Recuperar solo las secciones relevantes también ayuda (menos ruido en el medio).

Trampa: dejar la regla importante sepultada en el medio.

5.3 Batch API vs real-time

Qué es: dos formas de llamar a la API.

Cómo funciona: - **Batch (Message Batches):** ~50% más barato, pero **latencia impredecible** (hasta 24h). Para trabajo offline/nocturno. - **Real-time:** respuesta inmediata, más caro. Para lo interactivo o bloqueante.

Cuándo aplica cada uno: reporte de tech-debt nocturno → batch. Check bloqueante pre-merge que un dev espera → real-time.

Cuándo NO / diferencia: poner en batch algo que bloquea a una persona esperando = mal (la latencia lo mata). El ahorro no compensa bloquear un flujo interactivo.

Trampa: "batch todo para ahorrar 50%". Lo bloqueante/interactivo va real-time.

5.4 Descomponer una revisión grande

Qué es: cómo revisar algo grande (ej. un PR de 14 archivos) sin inconsistencias.

Cómo funciona: un solo pase diluye la atención y no tiene memoria transversal (marca un patrón en un archivo y aprueba el mismo en otro). Fix: **pases por archivo** para lo local + **un pase de integración** para lo cruzado.

Cuándo NO / diferencia: una ventana de contexto más grande sola no arregla la atención diluida. Correr 3 pases completos y quedarte con lo que aparece en ≥2 es más caro y no ataca la causa.

Trampa: "usa una ventana más grande". El problema es de estructura de revisión, no de tamaño de ventana.

Reglas de decisión para el examen (Foundations)

1. **Mínima complejidad que resuelve:** Augmented LLM < Workflow < Agentic < Orquestador. No subas de nivel sin necesidad.
2. **Regla dura → código, no prompt:** invariantes y precondiciones se implementan como gate/hook determinista.
3. **Causa raíz primero:** descripción de tool antes que capas; criterio categórico antes que "sé conservador".
4. **Schema = forma, no verdad:** garantiza sintaxis, no semántica.
5. **Preservá contexto:** extraer/recuperar, nunca truncar lo necesario.
6. **Ubicá el fallo en su capa:** descomposición vs búsqueda vs síntesis; datos vs prompt vs modelo.

Distractores que casi nunca son correctos

- "Usa un modelo más grande" (salvo model mismatch real).
- "Baja la temperatura" para garantizar JSON.
- "Ponlo en bold arriba" / "más few-shot" para un problema que no es de formato.
- "Batch todo para ahorrar".
- "Usa una ventana más grande" para arreglar consistencia de revisión.

Táctica

- ~2 min/ítem. Marcá dudosas y seguí.
- Multi-response: marcá **exacto** el número que pide el ítem.
- El % por dominio sale en el reporte pero **no decide** el pass; cuenta el score total escalado (720).