Índice

Guía de Estudio — Claude Certified Architect · Professional (CCAR-P)

> **Qué mide este examen: criterio de arquitecto aplicado**. No pregunta "¿qué es X?" sino "dado este caso con estas restricciones, ¿qué decisión?". Casi siempre dos opciones "funcionan", pero una es correcta **por principio**. La habilidad medida es elegir bien bajo trade-offs de costo, latencia, seguridad y mantenibilidad.
>
> **Formato:** 63 ítems · 120 min · corte 720/1000 · single y multi-response (cada ítem dice cuántas marcar). Proctored (Pearson VUE). El reporte da % por dominio, pero el pass lo decide el **score total escalado**.
>
> **Cómo usar esta guía:** cada concepto trae **Qué es → Cómo funciona → Cuándo aplica → Cuándo NO / diferencia → Trampa de examen**. La zona de examen es siempre la **diferencia entre dos opciones parecidas**.

Pesos del blueprint (memorízalos, orientan dónde caen más ítems):

Dominio	Peso
1. Solution Design & Architecture	17%
2. Claude Models, Prompting & Context Engineering	13%
3. Integration	**19%**
4. Evaluation, Testing & Optimization	16%
5. Governance, Safety & Risk Management	14%
6. Stakeholder Communication & Lifecycle Management	14%
7. Developer Productivity & Operational Enablement	7%

Dominio 1 — Solution Design & Architecture (17%)

Traducir problemas de negocio en arquitecturas y elegir el patrón correcto.

1.1 Los cuatro patrones y cuándo cada uno

Qué es: el abanico de arquitecturas, de menos a más autónomas.

- **Augmented LLM:** una llamada + retrieval/tool/memoria. Sin loop. Caso: transformación única con contexto al lado ("resume este ticket usando la KB").
- **Workflow:** varios pasos, **el código decide el orden**. Predecible, auditable. Caso: pasos conocidos e idénticos en cada pedido (clasificar → extraer → validar → aprobar).

- **Agentic:** varios pasos, **el modelo decide el orden y cuándo parar**, en loop planificar-actuar-observar. Caso: la ruta recién aparece durante la ejecución (investigar, debuggear).
- **Multi-agente (orquestador):** coordinador reparte a subagentes especializados y sintetiza. Caso: subtareas con especialización distinta, independientes, paralelizables.

Regla maestra: elegí el patrón **menos complejo** que resuelve. Cada nivel arriba agrega costo, latencia, imprevisibilidad y dificultad de debug/eval. El multi-agente "gana su complejidad" solo cuando las subtareas piden herramientas y contexto **genuinamente distintos** y pueden correr en paralelo.

Trampa: una tarea que suena sofisticada pero tiene flujo fijo → **workflow**, no agente. La sofisticación no justifica autonomía; la imprevisibilidad del camino sí.

1.2 Arquitectura end-to-end y feedback loop

Qué es: el flujo completo entrada → procesamiento → salida → **feedback**.

Cómo funciona: el tramo de feedback devuelve las señales de corrección de producción convertidas en un **conjunto evaluable** que mejora el sistema. Sin ese tramo, la arquitectura está incompleta.

Trampa: un diseño que cubre entrada/proceso/salida pero **omite el feedback**. La opción correcta suele ser la que cierra ese lazo (capturar señales de producción como dataset de evaluación).

1.3 Descomposición de problemas complejos

Qué es: partir un problema grande en piezas manejables.

Cómo funciona: una tarea larga que sale con **ítems salteados y razonamiento superficial** es una falla de descomposición. Secuenciar en pasos da **contexto enfocado** en cada uno y **salidas intermedias verificables**.

Trampa: atribuir a "el modelo es malo" lo que es una descomposición pobre. La firma (cobertura incompleta, superficialidad) apunta a cómo se partió el problema.

1.4 Multi-agente: contexto de subagente

Qué es: qué sabe un subagente.

Cómo funciona: nada se hereda. El subagente arranca con contexto limpio; todo lo que necesita va **explícito en su prompt**.

Trampa: asumir que el subagente ve la conversación del coordinador o sus variables. No las ve.

1.5 Alinear a valor de negocio

Qué es: justificar el diseño en pilares de valor (eficiencia, transformación, productividad, costo, SLA de performance).

Cómo funciona: el **primer proyecto** se elige por **valor medible balanceado con factibilidad y riesgo** → construye respaldo organizacional duradero.

Trampa: elegir el proyecto más vistoso, el más difícil, o el favorito técnico. La correcta balancea valor + factibilidad + riesgo.

Dominio 2 — Claude Models, Prompting & Context Engineering (13%)

2.1 Selección de modelo por trade-offs

Qué es: elegir familia/tamaño según la tarea.

Cómo funciona: - Mecánica / alto volumen / latencia crítica / formato simple → modelo **chico** (rápido, barato). - Juicio complejo / razonamiento multi-paso / síntesis → modelo **grande**. - A gran escala con **precisión comparable** entre modelos, la elección disciplinada es **el más chico que alcanza el umbral** — y la eval confirma que lo sigue alcanzando.

Cuándo NO / diferencia: subir de modelo **no** arregla problemas de auth, scope, retrieval o prompt mal escrito. Solo ayuda cuando el cuello de botella es genuinamente **capacidad de razonamiento** (model mismatch).

Trampa: "usa el modelo más grande" como respuesta universal. Casi nunca lo es fuera de model mismatch.

2.2 Routing por complejidad

Qué es: un triaje que manda cada pedido al modelo adecuado.

Cómo funciona: un modelo chico y barato clasifica la dificultad; lo fácil lo responde él, lo difícil escala al modelo grande. El ~80% fácil se resuelve barato; pagás grande solo por el ~20% difícil.

Ejemplo: chatbot de banco — "¿mi saldo?" → modelo chico; "¿es justo este cobro según mi contrato?" → modelo grande.

Trampa: mandar todo al modelo grande "por las dudas". El routing ahorra sin perder calidad en lo difícil.

2.3 Prompt caching

Qué es: reutilizar un prefijo de tokens idéntico entre requests.

Cómo funciona: el caché coincide **por prefijo**. Poné lo **estable primero** (system + política + few-shot) y lo **variable al final** (mensaje del usuario). Baja time-to-first-token (latencia) y costo por request.

Cuándo NO / diferencia: un **timestamp o ID en la posición 0** hace único cada prefijo → nunca hay acierto de caché. Truncar el documento pierde contexto; bajar de modelo a ciegas arriesga calidad; mover la política a few-shot no crea un prefijo cacheable.

Trampa (caso clásico): "mismo prompt de 8.000 tokens en cada request, importa costo y latencia" → la correcta es **ordenar estático primero + activar caching**. Las otras destruyen contexto o no cachean.

2.4 Técnicas de prompting

Qué es: zero-shot, few-shot, chain-of-thought.

Cómo funciona: - **Zero-shot:** tarea simple. - **Few-shot:** enseñar **formato/patrón**; "mostrar supera a describir" para formato estricto. - **Chain-of-thought:** razonamiento **multi-paso**; en extracción simple solo agrega costo.

Diferencia: few-shot = formato; CoT = razonamiento. Se aplican **por tarea y según beneficio medido**.

2.5 Guardrails en el prompt vs atención posicional

Qué es: dónde poner las reglas críticas.

Cómo funciona: reglas enterradas a mitad de contexto sufren caída de atención posicional. Moverlas al **inicio o final**, separadas del material de referencia, es el fix de mayor palanca. Estructura clara + prioridad explícita ante conflicto + ejemplos concretos de los casos que fallan = elimina ambigüedad.

Ojo — límite: para reglas de **seguridad**, el prompt no basta (ver Dominio 5.1 y 3.4). El prompt guía comportamiento; no es frontera de seguridad.

2.6 Optimizar ventana y tokens

Qué es: manejar contexto grande.

Cómo funciona: recuperar solo las secciones relevantes arregla dos cosas a la vez — esquiva la caída de atención en el medio y recorta tokens por pedido. Extraer hechos duros a un bloque persistente los protege de la compresión.

2.7 Reuse de prompts

Qué es: no repetir trabajo de prompt.

Cómo funciona: caching (prefijos), prompts modulares (componer bloques), y **Skills** (empaquetar instrucciones/workflows reutilizables).

Dominio 3 — Integration (19% · el de más peso)

3.1 Capability bloat en tools/agentes

Qué es: demasiadas herramientas degradan al agente.

Cómo funciona: un agente que creció a 45 tools en 6 dominios con precisión de selección cayendo, latencia subiendo y un tercio de tools nunca invocadas = capability bloat. Fix ordenado: **agentes por dominio detrás de un router**, cada uno con toolset enfocado; sacar tools muertas y solapadas; **progressive disclosure** para mantener chica la superficie por request a medida que el catálogo crece.

Cuándo NO / diferencia: “describí cada tool con más detalle” no arregla un problema de **cantidad**. “Agregá las tools nuevas ahora y reescribí el prompt el próximo trimestre” pospone el problema real.

Trampa: elegir el fix cosmético (más descripción) cuando la causa es volumen. La correcta reduce y segmenta.

3.2 Auth y authz: gaps de seguridad

Qué es: quién puede hacer qué.

Cómo funciona: - **Autenticación (authn):** probar **quién sos** (token, credencial). - **Autorización (authz): qué te dejan hacer** (scope, permisos). - El control de acceso vive en la **capa del sistema** — credencial **por usuario** o autenticación traspasada — para que el modelo **no pueda** recuperar/ejecutar lo que no debe.

El principio central — el prompt no es frontera de seguridad: si “instruimos al modelo a no devolver datos de otro usuario pero a veces lo hace”, la instrucción está haciendo el trabajo

de la autorización, y eso está mal. La autorización tiene que ser estructural (el modelo no recibe acceso), no una petición en texto.

Escalada de privilegios: un agente que corre con permisos del **sistema** actuando para un usuario común → puede hacer cosas que el usuario nunca podría. Los tokens deben tener **scope mínimo**.

Trampa: "mejorá el prompt para que no filtre". La correcta mueve el control a la capa del sistema.

3.3 Least privilege (mínimo privilegio)

Qué es: dar solo los permisos que el rol necesita.

Cómo funciona: si un rol no necesita una capability, **removela**. Jerarquía de controles, de más débil a más fuerte: 1. Loggear (detectivo — te enterás después, el daño ya pasó). 2. Confirmar antes (compensatorio — reduce, no elimina; un injection puede pasar por encima). 3. **Remover la tool (eliminativo — el daño se vuelve imposible). ← siempre gana cuando no se necesita.**

Caso canónico: agente de soporte que puede leer, redactar, **reembolsar y borrar cuentas**, pero soporte solo necesita leer y redactar → **remové reembolso y borrado**. No basta loggear ni confirmar; y un modelo más grande no tiene nada que ver con el scope de autorización.

Frase: no pongas guardias alrededor de una puerta que no debería existir — tapiala.

Trampa: elegir "loggear" o "confirmar" cuando la opción "remover" está disponible y la capability no se necesita.

3.4 Guardrails deterministas (gate en código)

Qué es: hacer cumplir invariantes de seguridad de forma confiable.

Cómo funciona: para una regla dura ("verificá identidad con `get_customer` antes de `process_refund`"), usá un **gate/hook programático** que bloquee la acción hasta que la precondición se cumpla. Determinista: falla 0%.

Cuándo NO / diferencia: una instrucción en el prompt ("siempre verificá primero") falla ~10% — es probabilística. Más ejemplos, ponerlo en bold, o bajar temperatura no lo vuelven confiable.

Trampa: el enunciado dice "instruimos al modelo pero a veces falla" → la correcta es el gate en código, no tocar el prompt.

3.5 Accuracy-latency trade-off

Qué es: balancear precisión contra velocidad/costo.

Cómo funciona: justificás la config según el caso — más pasos/modelo mayor/reranking suben accuracy pero cuestan latencia; menos suben velocidad pero arriesgan calidad. La respuesta correcta ata la decisión al requisito del caso (SLA, criticidad), no a una preferencia genérica.

3.6 RAG: cuándo, y alternativas

Qué es: recuperar contexto relevante de una base grande para el modelo.

Cuándo RAG vs alternativa: - **RAG:** base grande y **cambiante**, necesitás citar fuentes, solo una fracción es relevante por query. Datos que cambian a diario = caso canónico (responde vigente sin reentrenar). - **Long-context (todo en el prompt):** corpus **chico y estable** que cabe holgado. - **Tool/API:** dato **estructurado o en tiempo real** (saldo, stock). - **Fine-tuning:** cambiar **comportamiento/estilo/formato**, no inyectar conocimiento factual.

Diferencia clave: RAG da **conocimiento**; fine-tuning da **comportamiento**. No confundir.

3.7 RAG: chunking

Qué es: cómo cortás los documentos antes de indexarlos.

Cómo funciona: cortá **según la estructura** del documento (cláusulas, secciones) con **metadata de enlace** que preserve el contexto interpretativo. Chunk grande = trae ruido y diluye; chunk chico = pierde contexto.

Cuándo NO / diferencia: pedazos de **tamaño fijo** cortan cláusulas a la mitad y rompen el sentido. Reducir a 100 tokens no "mejora la precisión", empeora el contexto. Traer 50 chunks "por si acaso" mete ruido.

Trampa: la opción de tamaño fijo o de subir k a lo bruto. La correcta corta por estructura con metadata.

3.8 RAG: indexing

Qué es: convertir chunks en embeddings y guardarlos para búsqueda.

Cómo funciona: el modelo de embedding del índice y el de la query **deben coincidir**. Para una KB que se **edita todo el día** y debe reflejarse en minutos → un enfoque de indexación que soporte **updates incrementales** frecuentes.

Diagnóstico: si todo empeora **tras un refresh de documentos** (modelo y latencia iguales) → el problema es el **index/retrieval** (re-index roto, embeddings desalineados, chunks stale), no el modelo.

Trampa: atribuir al modelo un fallo que apareció al tocar los datos.

3.9 RAG: retrieval

Qué es: buscar y traer los chunks relevantes en runtime.

Cómo funciona: semántico (intención, parafraseo), keyword/BM25 (términos exactos, códigos, nombres), **híbrido** (ambos), reranking (precisión en top-k). Matcheá la estrategia al **patrón de query**.

Diferencia entre los tres pasos: - **Chunking** = cómo **cortás** (antes, una vez). - **Indexing** = cómo **guardás/organizás** (antes, una vez). - **Retrieval** = cómo **buscás y traés** (en runtime, cada query).

3.10 Protocolo de conexión: MCP vs API/CLI vs agent-to-agent

Qué es: el mecanismo de integración.

Cómo funciona: - **MCP:** muchos sistemas, muchas apps consumidoras, propiedad **descentralizada**, tools que rotan seguido. Cada equipo mantiene su server una vez, todos reusan. Caso canónico. - **API/CLI:** integración puntual, a medida, un consumidor. - **Agent-to-agent:** delegar a otro agente autónomo con su razonamiento y tools.

Trampa: confundir MCP (protocolo estandarizado reutilizable) con una API directa (plomería a medida).

3.11 Progressive discovery vs monolithic

Qué es: cuánto contexto/tools cargás y cuándo.

Cómo funciona: - **Monolítico:** todo al inicio. Bueno solo con pocas tools y poco contexto. - **Progressive discovery:** cargás solo lo relevante para el paso actual; el resto se revela on-demand. Menos tokens, mejor selección de tool, escala. Es la contraparte del capability bloat.

Trampa: cargar 50 definiciones de tools de golpe. Con catálogo grande → progressive.

3.12 Observabilidad a escala

Qué es: monitorear un sistema agéntico en producción.

Cómo funciona: tracing por request, logs estructurados, métricas por tool/agente. **Qué debe disparar una alerta:** tasa de error de tools sobre umbral **sostenida varios minutos**.

Cuándo NO / diferencia: NO alertar por un solo request más lento que la mediana, ni por el gasto diario subiendo, ni por un rating individual "poco útil". Esas son señales ruidosas, no incidentes.

Trampa: elegir la métrica ruidosa. La correcta es el patrón sostenido y significativo.

Dominio 4 — Evaluation, Testing & Optimization (16%)

4.1 Definir métricas

Qué es: qué medís.

Cómo funciona: accuracy, latency, cost, safety, security. Elegí las relevantes al caso; no todas importan igual siempre.

4.2 Diseñar datasets y frameworks (metodologías mixtas)

Qué es: cómo armás la evaluación pre-producción.

Cómo funciona (caso dominio regulado — los 2 que más importan): - **Golden dataset** etiquetado por **expertos de dominio**, cubriendo casos de **alto riesgo**. - **Casos adversariales** que prueban límites de safety y prompt injection.

Cuándo NO / diferencia (distractores): demo con ejemplos cherry-picked al ejecutivo, benchmark contra competidores, encuesta de satisfacción post-launch. Ninguno es una evaluación rigurosa pre-producción.

Trampa: elegir la demo o la encuesta. La correcta es dataset experto + adversarial.

4.3 Métodos de evaluación

Qué es: cómo puntuás las salidas.

Cómo funciona y cuándo: - **Exact match / assertions programáticas:** salida objetiva verificable (clasificación, extracción, schema). Barato, determinista, escala. - **LLM-as-judge:** calidad subjetiva (resumen, tono, relevancia). Escala, pero **tiene sesgos** (favorece respuestas largas, se puede engañar) → calibralo contra humano. - **Human eval:** casos críticos o para calibrar al juez. Alta calidad, caro, no escala. - **Mixto:** cuando hay componentes objetivos **y** subjetivos → combiná.

Trampa: tratar al LLM-judge como verdad absoluta. Necesita calibración humana.

4.4 A/B testing e iteración

Qué es: comparar cambios con evidencia.

Cómo funciona: cambiá **una variable a la vez** contra un **baseline**, medí, decidí. Iterá.

Trampa: cambiar varias cosas a la vez → no sabés qué causó la diferencia.

4.5 Diagnóstico de fallos (causa → síntoma)

Qué es: ubicar la causa raíz de un problema.

Cómo funciona (tabla mental): - Falló tras **refresh de datos** → retrieval/index. - Falló tras cambiar **el prompt** → prompt failure. - Falla en tarea que un **modelo mayor** sí resuelve → model mismatch. - Inventa datos que no están en la fuente → alucinación (grounding + citas + campos nullable). - Inconsistente entre corridas idénticas → temperatura alta.

Regla: el síntoma aparece justo tras tocar una capa → el problema está en esa capa.

4.6 Optimización costo-latencia-performance

Qué es: hacer el sistema más eficiente sin romper calidad.

Cómo funciona: caching, modelo más chico donde la tarea lo tolera, recorte de tokens, paralelización, streaming (percepción de latencia). Siempre medido contra baseline y confirmado por eval.

4.7 Monitoreo con logging y observabilidad

Ver 3.12. La eval no termina en pre-producción: en producción, alertá por patrones sostenidos y significativos, no por ruido puntual.

Dominio 5 — Governance, Safety & Risk Management (14%)

5.1 Guardrails y controles de seguridad

Qué es: hacer cumplir reglas de seguridad.

Cómo funciona: invariantes duros → **gate en código** (determinista). El prompt guía comportamiento pero **no es frontera de seguridad** (fuga ~10%). Combiná: control de acceso en la capa del sistema + validación de input/output + gates programáticos.

Trampa: confiar la seguridad al prompt. Va en código/sistema.

5.2 Riesgos y failure modes de LLM (diferencias)

Qué es: las maneras en que un LLM falla. Distinguilas por síntoma; cada una tiene mitigación distinta.

- **Alucinación:** inventa info con seguridad por **falta de conocimiento** (rellena). Mitigá: grounding, citas, campos nullable (“no sé” legal).
- **Sicofancia: te da la razón** para complacerte, aunque estés mal (sabe pero te sigue). Mitigá: prompts neutrales que no filtren la respuesta esperada.
- **Prompt injection:** ataque **externo** — datos que el agente procesa contienen instrucciones ocultas que lo secuestran. Mitigá: separar instrucciones de datos, sanitizar input, no dar poder de acción a tools que leen data no confiable.

- **Jailbreak:** ataque del **usuario** para saltar las reglas de seguridad del modelo. Mitigá: guardrails robustos, clasificadores/moderación.
- **Model mismatch:** el modelo **no da la talla** (muy chico para la tarea). Mitigá: subir modelo (acá sí) o routing.
- **Drift:** el desempeño **se degrada con el tiempo** sin cambios (el mundo cambió). Mitigá: monitoreo continuo, re-eval, actualizar datos/prompts.

Diferencias que confunden: - Alucinación (no sabe y rellena) vs sicofancia (sabe pero te complace). - Prompt injection (ataque vía **datos de terceros**) vs jailbreak (ataque del **usuario**). - Model mismatch (falta capacidad) vs los demás (no es cuestión de tamaño).

5.3 Human-in-the-loop

Qué es: meter aprobación humana en el flujo.

Cómo funciona: para acciones **irreversibles o de alto impacto** (transferencias grandes, borrado de datos, decisiones legales/médicas), el humano aprueba **antes** de ejecutar.

Cuándo NO / diferencia: no en cada paso — mata el throughput y anula el valor de automatizar. Es un control **calibrado al riesgo**.

Trampa: "humano en todos los pasos" (mata el sistema) o "sin humano en una acción irreversible crítica" (riesgo inaceptable). La correcta calibra.

5.4 Compliance (regulación ↔ dominio)

Qué es: cumplir marcos regulatorios.

Cómo funciona (asociá): - **GDPR:** datos personales UE — derecho al olvido, minimización, consentimiento, portabilidad. - **HIPAA:** salud protegida (PHI) EE.UU. - **FedRAMP:** cargas de trabajo gov cloud EE.UU. - **SOC 2:** controles de seguridad organizacional.

Trampa: el caso menciona el dominio (hospital → HIPAA, usuarios europeos → GDPR, agencia federal US → FedRAMP). Asociá bien.

5.5 Ética de IA

Qué es: consideraciones de bias, fairness, transparencia.

Cómo funciona: bias (sesgo en datos/salidas), fairness (trato equitativo entre grupos), transparencia (explicabilidad + disclosure de que es IA). Se abordan con datos representativos, auditoría de sesgo, y comunicación clara al usuario.

Dominio 6 — Stakeholder Communication & Lifecycle Management (14%)

6.1 Discovery estructurado

Qué es: levantar requisitos antes de diseñar.

Cómo funciona: entendé el problema de negocio y las restricciones **antes** de proponer arquitectura. Preguntas estructuradas, no asumir.

Trampa: saltar directo a la solución técnica sin discovery. Casi siempre incorrecta.

6.2 Comunicar decisiones y trade-offs

Qué es: explicar la arquitectura al stakeholder.

Cómo funciona: comunicá **trade-offs** (costo vs latencia vs riesgo vs valor), no features técnicos. "Opción A es más barata pero 200ms más lenta" > "usé tal modelo".

Trampa: listar features o jerga técnica. El stakeholder decide por trade-offs de negocio.

6.3 Gestión de expectativas y SLAs

Qué es: alinear lo que el sistema promete.

Cómo funciona: convertí expectativas vagas en **métricas medibles** — p95 de latencia, uptime, accuracy mínima. El SLA es un compromiso cuantificado.

Trampa: aceptar expectativas cualitativas ("que sea rápido"). Cuantificá.

6.4 Documentar y guiar implementación

Qué es: el handoff.

Cómo funciona: documentá decisiones arquitectónicas (el porqué, no solo el qué) y provee guía de implementación para el equipo que construye/mantiene.

6.5 Fases del ciclo de vida

Qué es: el recorrido completo.

Cómo funciona: discovery → design → build → **handoff** → monitoring → iteration. El monitoreo y la iteración no son opcionales en producción; el sistema vive y evoluciona.

Trampa: tratar el proyecto como "termina en el deploy". El ciclo incluye monitoreo e iteración.

Dominio 7 — Developer Productivity & Operational Enablement (7%)

7.1 Configurar Claude Code para equipos

Situación	Correcto
Comando para todo el equipo	`.claude/commands/` en el repo (versionado)
Comando solo mío	`~/.claude/commands/` (home)
Instrucción compartida	`CLAUDE.md` del repo
A un dev no le llegan las instrucciones comunes	Están en su `~/.claude/` personal → mover al repo
Secreto de MCP sin filtrarlo a git	`.mcp.json` versionado + `${ENV_VAR}`
Convención por tipo de archivo disperso	Rule con glob `**/*.ext`
CI se cuelga esperando input	`claude -p "..."` (non-interactive)
Aislar output ruidoso de una tarea	Skill con `context: fork`

7.2 Mejorar workflows con tooling asistido

Qué es: habilitar al equipo con IA.

Cómo funciona: el arquitecto configura y habilita (comandos, skills, MCP compartidos) para que el equipo sea más productivo, no solo lo usa él.

7.3 Debugging y resolución operativa

Qué es: apoyar incidentes con IA.

Cómo funciona: usar Claude Code para investigar logs, reproducir bugs, proponer fixes, acelerar la resolución de incidentes.

Reglas de decisión maestras (el corazón del Professional)

Cuando dos opciones "funcionan", aplicá el principio:

1. **Mínima complejidad que resuelve:** workflow antes que agente si el camino es fijo.
2. **Elimina el riesgo, no lo vigiles:** remover tool > confirmar > loggear.
3. **Seguridad en la capa del sistema, no en el prompt:** gate/auth en código; el prompt no es frontera de seguridad.
4. **Preservá el contexto necesario:** recuperar/extraer, nunca truncar lo que hace falta.
5. **Atacá la causa raíz, no el síntoma.**
6. **Localizá el fallo por la capa que cambió:** datos→retrieval, prompt→prompt, tarea→modelo.
7. **Valor de negocio medible** manda en priorización.
8. **Trade-offs, no features**, al comunicar con stakeholders.
9. **Calibrá al riesgo:** human-in-the-loop donde importa, no en todo.
10. **Chunk por estructura, no por tamaño fijo.**

Distractores que casi nunca son correctos

- "Usa un modelo más grande" (salvo model mismatch real).
- "Sube temperatura" / "ponlo en bold" / "más few-shot" para problemas que no son de formato.
- "Truncá el documento" (pierde contexto).
- "Loggear/confirmar" cuando "remover" está disponible y la capability no se necesita.
- "Mejorá el prompt" para hacer cumplir una regla de seguridad dura.
- "Batch todo para ahorrar" cuando algo es bloqueante/interactivo.
- Alertar por un request lento aislado o un rating individual.
- Demo cherry-picked o encuesta post-launch como "evaluación".

Táctica de examen

- 63 ítems / 120 min ≈ **1.9 min/ítem**. Marcá dudosas y seguí; no te trabes.
- **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)**. Cubrí parejo.
- Ante dos opciones válidas, ganá con el **principio**, no con la intuición.