Facsímil 04

La caja de herramientas

Capítulo 02

APIs de modelos: mensajes, streaming y salidas estructuradas

686f6c61 · 2026

Capítulo 02: APIs de modelos: mensajes, streaming y salidas estructuradas

El modelo no vive solo en una caja de texto

Cuando usamos un chat en el navegador, parece que el sistema funciona así: escribes una pregunta y aparece una respuesta. Para aprender a construir productos con IA, esa imagen se queda corta. Entre tu aplicación y el modelo hay un contrato: qué modelo llamas, qué mensajes envías, qué herramientas están disponibles, qué formato esperas, si quieres streaming, qué harás si la salida no valida y cómo guardarás la traza.

Este capítulo baja un escalón desde el criterio del capítulo 01. Allí decidimos cuándo tiene sentido usar prompt, schema, RAG, tool o ajuste. Aquí aprendemos a convertir esa decisión en una petición concreta a una API de modelos.

La idea central es sencilla: **una API de modelos no es un buzón de texto; es una frontera entre software probabilístico y software verificable**.

Estado del arte con fecha de corte

Fecha de corte: 10 de junio de 2026.
Fuentes consultadas ese día: documentación oficial de OpenAI sobre Responses API, generación de texto, archivos, visión, salidas estructuradas, function calling, streaming, SDKs y Agents SDK; documentación de Anthropic sobre Messages API, visión, PDF, streaming y structured outputs; documentación de Google sobre Gemini API, documentos, visión, structured outputs, ADK, MCP y A2A; documentación oficial del protocolo A2A; JSON Schema; y la especificación WHATWG de Server-Sent Events.[1]

La parte estable es el patrón: enviar una petición con modelo, instrucciones, entrada, herramientas opcionales, formato esperado y opciones de ejecución. La parte cambiante son los nombres exactos de endpoints, SDKs, modelos, campos y límites. Por eso conviene aprender la arquitectura mental antes que memorizar una firma de cliente.

OpenAI documenta la generación de texto y el uso de salidas estructuradas para pedir respuestas que cumplan un schema.[2][3] También documenta function calling para que el modelo solicite llamadas a funciones definidas por la aplicación.[4] Anthropic organiza su API alrededor de mensajes y documenta streaming y salidas estructuradas como patrones de construcción.[5][6][7] Google documenta Gemini como una API de generación de contenido con entrada textual y multimodal, además de structured outputs basadas en schema.[8][9]

La revisión del 10 de junio no cambia la tesis del capítulo, pero sí refuerza una decisión de ingeniería: **no acoples tu dominio al JSON exacto de un proveedor**. OpenAI mantiene como piezas centrales Responses, structured outputs, function calling y streaming; Anthropic conserva Messages API, streaming y patrones de salida estructurada; Google añade además superficies recientes alrededor de Interactions API, ADK y protocolos de agentes.[101112] La conclusión práctica es aburrida y muy importante: escribe un adaptador por proveedor, valida la salida en tu código, guarda trazas comparables y deja que tu aplicación hable en objetos propios, no en campos de moda.

Qué no es una API de modelos

Una API de modelos no es “el prompt por HTTP”. Si la tratas así, acabarás pegando instrucciones, datos, historial, formato esperado y lógica de negocio en una sola cadena. Eso funciona en una demo; se vuelve frágil cuando hay usuarios reales.

Tampoco es una promesa de verdad. La API puede devolver texto muy convincente, pero tu aplicación sigue necesitando validación, métricas, trazas y reglas de producto. Si pides JSON y no validas JSON, no tienes contrato: tienes esperanza.

Y no es una interfaz idéntica entre proveedores. Muchos conceptos se parecen, pero los detalles cambian: roles disponibles, nombre del campo de entrada, eventos de streaming, formato de tools, límites, modelos, errores y objetos de respuesta. Por eso el diseño de tu aplicación debería tener una capa propia que traduzca “lo que necesita mi producto” a “lo que espera este proveedor”.

Qué sí es: un contrato entre capas

Una API de modelos es un contrato entre tres mundos. El primero es tu aplicación: usuarios, permisos, pantallas, flujos y datos. El segundo es el proveedor del modelo: endpoint, modelo, tokens, eventos, herramientas y respuesta. El tercero es tu código de integración: validadores, retries, logs, evals y transformación a objetos de dominio.

Formalmente, una petición no es una cadena: es un valor de un **tipo producto**, lo que en programación llamamos un registro o `struct`. Tiene un campo por cada decisión, y cada campo se puede tipar y validar por separado, igual que un modelo de Pydantic o una interfaz de TypeScript. En notación de tipos:

$$R = M \times I \times C \times F \times T \times S$$

Una petición concreta $r \in R$ fija un valor en cada campo:

CAMPO	SIGNIFICADO	EJEMPLO
M	Modelo elegido.	Un modelo rápido para clasificación.
I	Instrucciones y mensajes.	Rol del sistema, contexto y pregunta del usuario.
C	Contexto externo.	Fragmentos RAG, datos de producto o historial relevante.
F	Formato de salida.	JSON con `categoria`, `prioridad` y `siguiente_paso`.
T	Tools disponibles.	`consultar_expediente(id_alumno)`.
S	Opciones de servicio.	Streaming, límite de salida, metadata o timeout.

Verlo como registro tipado, y no como una cadena opaca, es lo que permite localizar el fallo en un campo concreto. Si falla F, quizá necesitas schema. Si falla C, quizá necesitas retrieval. Si falta T, el modelo no debería inventar estado externo. Si falla S, puede que el problema sea latencia, no inteligencia.

Qué APIs se usan hoy y quién las usa

A 10 de junio de 2026, cuando una empresa dice “vamos a llamar a un LLM desde nuestra app”, normalmente está hablando de una de estas familias. No son las únicas, pero sí sirven para entender el mercado: APIs directas de laboratorios, APIs cloud que empaquetan varios modelos, gateways que unifican proveedores y runtimes locales que imitan una interfaz remota.

OpenAI organiza gran parte de su construcción moderna alrededor de la Responses API: una llamada que puede recibir entrada multimodal, instrucciones, tools, formato de salida, streaming y opciones de ejecución.[13] La usan equipos que quieren construir asistentes, clasificadores, flujos con herramientas, extracción estructurada, análisis de documentos, experiencias con visión o productos donde una respuesta textual debe convertirse en objeto de software.

Anthropic expone Claude principalmente mediante Messages API para conversaciones y turnos sin estado: envías una lista estructurada de mensajes y el modelo genera el siguiente mensaje.[14] Es habitual en productos que necesitan lectura y escritura largas, análisis documental, asistentes internos, ayuda a programación, tutoría o flujos donde interesa controlar muy bien el historial enviado.

Google ofrece la Gemini API alrededor de `generateContent`, con entradas que pueden combinar texto, imágenes, vídeo y audio, además de configuración de generación y salidas estructuradas.[15] Encaja especialmente en productos multimodales, prototipos integrados con

el ecosistema Google, procesamiento de documentos y aplicaciones que quieren aprovechar modelos con ventanas largas o capacidades visuales.

Además existen plataformas como Amazon Bedrock, Vertex AI, Azure AI Foundry, Mistral, Cohere, Groq, Together, Fireworks, OpenRouter o Vercel AI SDK. La lección no es aprenderlas todas de memoria. La lección es que casi todas terminan pidiendo lo mismo con nombres distintos: modelo, entrada, instrucciones, parámetros de generación, herramientas, schema, streaming, límites y metadatos.

Los parámetros: no memorizar nombres, entender familias

Cuando un alumno ve por primera vez una referencia de API, se pierde porque parece una lista interminable de campos. La forma útil de estudiarla es agrupar parámetros por intención, es decir, agrupar los campos del registro R en seis familias: **qué modelo** usamos, **qué entra**, **cuánto y cómo responde**, **qué herramientas puede pedir**, **qué forma debe tener la salida** y cómo se **opera** el sistema. Memorizar nombres concretos es inútil; reconocer a qué familia pertenece cada campo sirve en cualquier proveedor.

La tabla siguiente no pretende congelar una API que cambia. Sirve para reconocer esas equivalencias entre familias:

FAMILIA	OPENAI RESPONSES API	ANTHROPIC MESSAGES API	GEMINI API
Modelo	`model`	`model`	modelo en `models.generateContent` o ruta REST.
Entrada	`input` con mensajes y bloques de contenido.	`messages` con contenido por turnos.	`contents` con `parts`.
Instrucciones estables	`instructions` o mensajes equivalentes según SDK.	`system` como campo superior.	`systemInstruction` o configuración equivalente.
Límite de salida	`max_output_tokens` según modelo y endpoint.	`max_tokens`.	`maxOutputTokens` dentro de configuración.
Aleatoriedad	`temperature`, `top_p` cuando el modelo lo permita.	`temperature`, `top_p`, `top_k` según modelo.	`temperature`, `topP`, `topK`.
Parada	secuencias de parada si están disponibles.	`stop_sequences`.	`stopSequences`.
Salida estructurada	formato de texto/schema o Structured Outputs.	`output_config.format` o tool estricta, según caso.	`responseMimeType` y `responseJsonSchema`.
Tools	`tools` y `tool_choice`.	`tools` y `tool_choice`.	`tools` y function calling.
Streaming	`stream` y eventos.	`stream` con eventos SSE.	métodos de streaming del SDK o REST.
Metadatos y operación	`metadata`, trazas, `user` o campos de servicio cuando existan.	`metadata`, uso de tokens y estado de parada.	`usageMetadata`, configuración y metadatos del entorno.

El parámetro más peligroso no es siempre el más técnico. Muchas integraciones fallan por no fijar bien `max_tokens` o `max_output_tokens`, por mezclar instrucciones con datos del usuario, por no versionar el schema o por asumir que `temperature=0` convierte una salida probabilística en una función matemática.

Entradas y salidas: qué ocurre en cada caso

Una API de modelos no devuelve siempre “texto”. Puede devolver texto, JSON, una petición de tool, eventos parciales o una combinación de bloques. Por eso conviene diseñar el flujo antes de escribir el cliente.

CASO	ENTRA	SALE	QUÉ DEBE HACER TU APLICACIÓN
Texto a texto	Pregunta, instrucciones y contexto breve.	Respuesta natural.	Mostrar, registrar y quizá evaluar calidad.
Texto a JSON	Texto y schema.	Objeto estructurado.	Parsear, validar y transformar a objeto de dominio.
Documento a resumen	Archivo, páginas o fragmentos.	Resumen, citas o extracción.	Conservar referencia a página, versión y documento original.
Imagen a explicación	Imagen más pregunta.	Descripción, clasificación o lectura visual.	Revisar límites visuales y pedir evidencia cuando importe.
Texto a tool call	Petición y definición de función.	Nombre de tool y argumentos.	Validar argumentos, comprobar permisos y ejecutar código.
Tool result a respuesta	Resultado externo.	Explicación final.	Separar dato consultado de redacción generada.
Streaming	Petición normal con `stream`.	Eventos parciales.	Acumular, cancelar, manejar errores y cerrar estado.

La salida estructurada y las tool calls se parecen porque usan schemas, pero no cumplen la misma misión. Una salida estructurada es el resultado final con forma de dato. Una tool call es una solicitud intermedia para que el sistema consulte, calcule o actúe. Si confundes ambas cosas, tu app termina ejecutando texto como si fuera una decisión cerrada.

Documentos e imágenes: multimodal no significa comprensión automática

Enviar una imagen o un PDF a un modelo multimodal no equivale a “el modelo lo sabe todo sobre ese archivo”. Equivale a darle una entrada rica que el modelo puede procesar dentro de sus límites. OpenAI documenta bloques como `input_file` para archivos y `input_image` para imágenes dentro de la entrada.[16][17] Anthropic documenta visión y soporte de PDF para Claude.[18][19] Gemini permite pasar imágenes como datos inline o mediante Files API, y documenta límites específicos para PDF.[20][21]

Para una sola imagen, suele bastar con enviar la imagen y una pregunta clara: “lee esta factura y extrae fecha, proveedor e importe”. Para muchas imágenes o archivos repetidos, conviene subirlos una vez y referenciarlos. Para un PDF largo, hay que decidir si se manda

entero, si se divide por páginas, si se indexa en un sistema RAG o si se usa una combinación: retrieval para localizar fragmentos y modelo multimodal para interpretar tablas, figuras o páginas concretas.

Hay cuatro cosas que no deberíamos olvidar:

CUIDADO	POR QUÉ IMPORTA	BUENA PRÁCTICA
Tamaño y coste	Imágenes y documentos consumen contexto y pueden aumentar latencia.	Medir tokens, páginas, resolución y tiempo de respuesta.
Referencias	Una respuesta sin página o fragmento es difícil de revisar.	Pedir `pagina`, `fragmento` o `evidencia` en el schema.
Privacidad	Los documentos pueden contener datos personales o internos.	Minimizar, redactar cuando sea posible y revisar política del proveedor.
Lectura visual	Un modelo puede interpretar mal tablas, sellos o capturas pequeñas.	Comprobar con OCR, reglas o revisión humana cuando importe.

La regla práctica: si el documento es fuente de verdad, no lo trates como “contexto decorativo”. Guárdalo con identificador, versión, fecha de carga y forma de recuperación. Si mañana alguien pregunta por qué la app contestó eso, debes poder reconstruir qué archivo vio, qué páginas entraron y qué schema validó la salida.

Mensajes: separar instrucciones, contexto y petición

La mayoría de APIs modernas no reciben solo una cadena. Reciben mensajes o una estructura equivalente. El objetivo no es teatralizar una conversación, sino separar responsabilidades: qué reglas gobiernan la tarea, qué dijo la persona usuaria, qué contestó antes el modelo y qué resultados devolvieron herramientas.

PIEZA	QUÉ REPRESENTA	RIESGO SI SE MEZCLA
Instrucciones de sistema o desarrollador	Comportamiento estable que quieres mantener.	El usuario puede pisar reglas importantes con texto accidental.
Mensaje de usuario	La petición concreta de esta interacción.	El sistema no distingue objetivo de contexto.
Contexto recuperado	Evidencia externa añadida por la aplicación.	El modelo no sabe qué parte citar o priorizar.
Mensaje del asistente	Respuesta anterior o salida actual.	Se pierde trazabilidad en conversaciones largas.
Resultado de tool	Dato externo obtenido por código.	Se confunde texto generado con dato comprobado.

Un patrón robusto consiste en construir la petición desde piezas separadas y solo al final traducirlas al formato del proveedor. Así puedes cambiar de modelo sin reescribir la lógica de negocio.

Para entenderlo: si una app universitaria pregunta “¿puede Ana matricularse de Sistemas Inteligentes?”, no deberías mandar solo esa frase. Deberías separar la política académica vigente, el identificador de Ana, las reglas de formato de salida y, si hace falta, la tool que consulta expediente.

Salidas estructuradas: cuando el texto debe convertirse en dato

Pedir “responde en JSON” es una intención. Usar un schema es un contrato. JSON Schema define vocabulario para describir tipos, propiedades, campos requeridos y reglas de validación sobre documentos JSON.[22] Las APIs de modelos aprovechan esa idea para reducir la distancia entre respuesta natural y objeto que tu software puede consumir.

La métrica mínima de una salida estructurada es la tasa de conformidad:

$$\text{validez} = \frac{N_{\text{válidas}}}{N_{\text{total}}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$N_{\text{válidas}}$	Respuestas que cumplen schema.	97 de 100 respuestas.
N_{total}	Respuestas evaluadas.	100 casos de prueba.
validez	Proporción de salidas estructuralmente correctas.	$0{,}97$.

Como toda proporción medida sobre una muestra, la validez tiene incertidumbre: con $N_{\text{total}} = 100$ y $\text{validez} = 0{,}97$, el intervalo de confianza al 95 % es $0{,}97 \pm 1{,}96 \quad 0{,}97 \cdot 0{,}03/100 \approx \pm 0{,}034$. La lectura práctica es la contraria a la de una eval de calidad: en validez de schema, un solo fallo de cada 100 no es ruido, es un patrón roto que tu validador debe rechazar siempre, no un porcentaje aceptable.

Pero cuidado: una respuesta puede cumplir schema y seguir siendo mala. Si el schema pide `prioridad`, el valor puede ser válido como texto y estar mal como decisión. Por eso necesitamos dos validaciones:

VALIDACIÓN	PREGUNTA	EJEMPLO
Estructural	¿Cumple tipos, campos y restricciones?	`prioridad` existe y vale `alta`, `media` o `baja`.
Semántica	¿El contenido es correcto para el caso?	El mensaje realmente requiere prioridad alta.

Un detalle de ingeniería que la práctica de este capítulo aprovecha: declarar `additionalProperties: false` junto con el modo estricto convierte el schema en un contrato de **mundo cerrado**, en el que el modelo no puede añadir campos que no estén declarados. Sin eso, una respuesta puede validar y aun así arrastrar claves inventadas que tu backend no espera. Un buen contrato no solo dice qué campos deben estar; dice también qué campos no pueden estar.

La salida estructurada arregla el contrato con el software. No reemplaza la evaluación del criterio.

Streaming: que la respuesta llegue por partes

Streaming significa que la aplicación no espera a tener toda la respuesta para empezar a recibir fragmentos. En la web moderna suele implementarse con eventos o flujos parecidos a Server-Sent Events, donde el servidor envía datos progresivamente al cliente.[23] OpenAI y

Anthropic documentan streaming para respuestas de modelos.[24][25]

La razón no es solo estética. El streaming cambia la experiencia percibida.

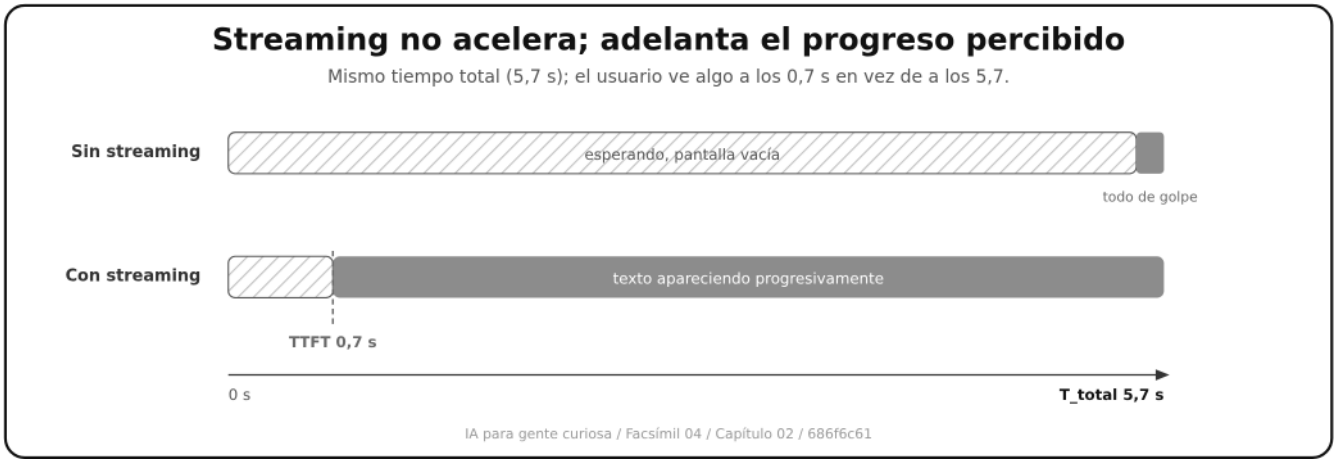
La clave está en separar el tiempo total de la latencia percibida. El tiempo total hasta terminar una respuesta se descompone, según el modelo de dos fases (*prefill* y *decode*) de la inferencia autoregresiva, en el tiempo hasta el primer token y el tiempo de generar el resto:[26]

$$T_{\text{total}} = \text{TTFT} + \frac{n_{\text{salida}}}{v}$$

Donde v es el ritmo de generación en tokens por segundo, de modo que n_{salida}/v es la fase de decode y equivale a $(n_{\text{salida}})\cdot\text{TPOT}$, el tiempo por token de salida.

SÍMBOLO	SIGNIFICADO	EJEMPLO
TTFT	Tiempo hasta el primer token (*time to first token*).	0,7 s.
n_{salida}	Tokens que hay que generar.	300 tokens.
v	Velocidad de generación (tokens por segundo).	60 tokens/s.
T_{total}	Tiempo total de la respuesta.	$0{,}7 + 300/60 = 5{,}7$ s.

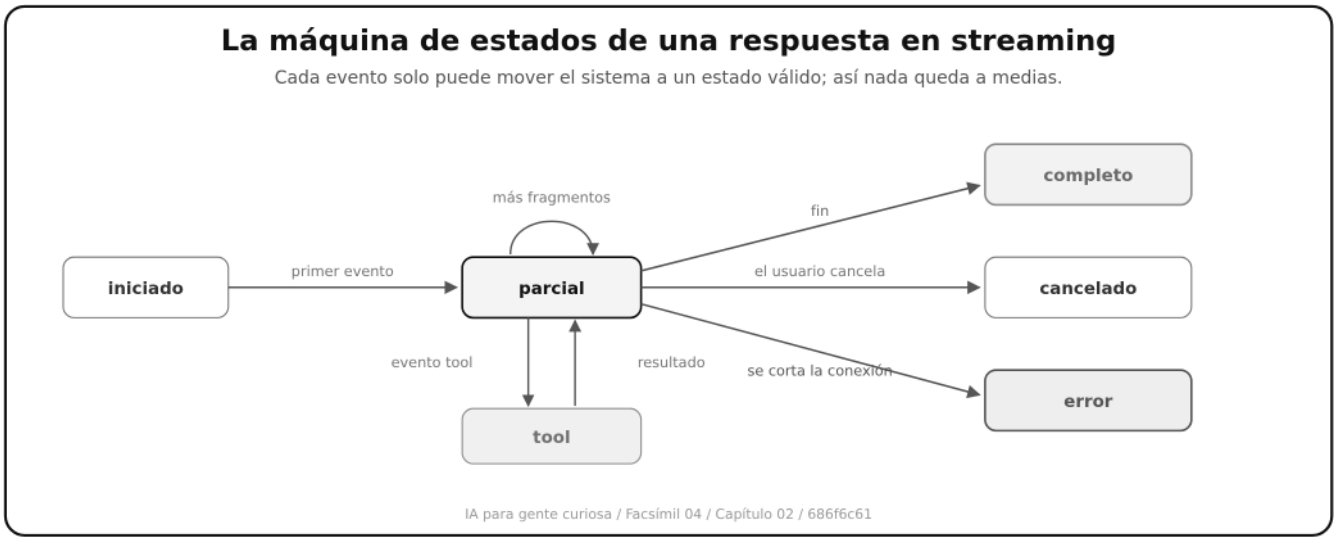
Streaming no cambia T_{total}: el modelo tarda lo mismo en terminar. Lo que cambia es **cuándo el usuario percibe progreso**. Sin streaming, no ve nada hasta los 5,7 segundos; con streaming, ve el primer fragmento a los 0,7 (la TTFT) y el resto va apareciendo. La latencia percibida cae de T_{total} a TTFT, aunque el coste total no se mueva.



Streaming no hace que el modelo “piense mejor”. Hace que el producto pueda mostrar progreso, cancelar, actualizar UI y registrar eventos. También complica: debes ensamblar fragmentos, manejar cortes, distinguir eventos de texto y eventos de tool, y decidir cuándo

una salida estructurada está lista para validarse.

Por eso una respuesta en streaming se modela mejor como una **máquina de estados finita**: un conjunto de estados y las transiciones permitidas entre ellos. Tratarla así evita el error más común, dejar la interfaz o la traza a medias cuando la conexión se corta, porque cada evento que llega solo puede mover el sistema a un estado válido.

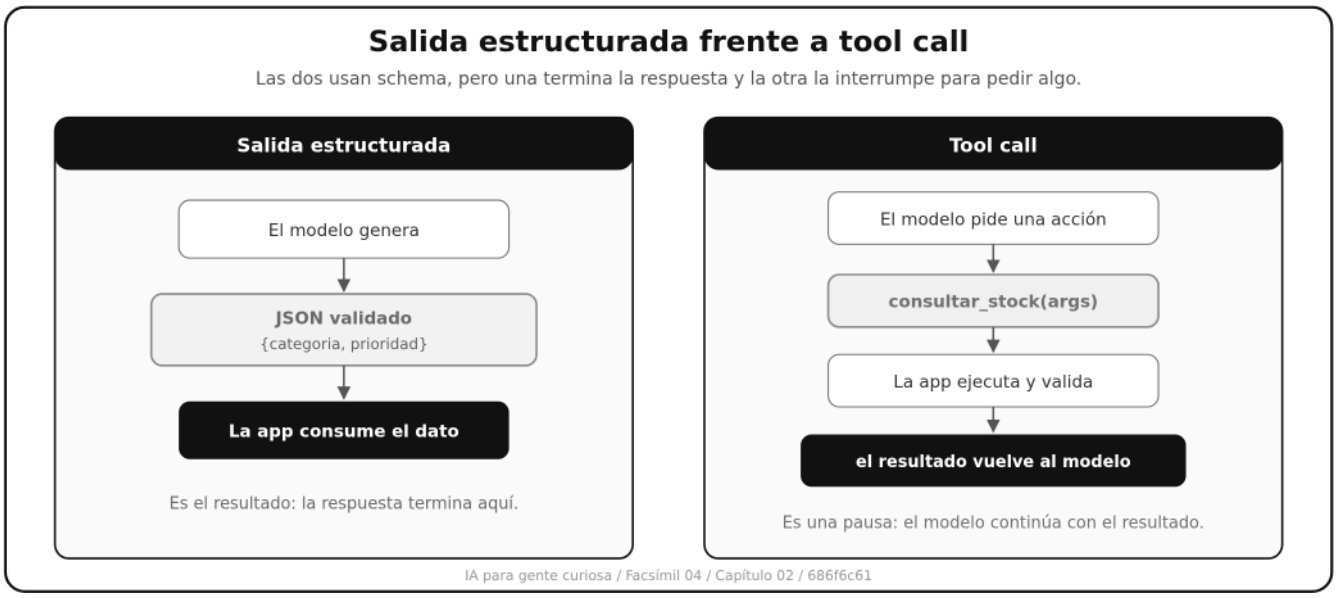


Tool calls: cuando responder no basta

Una salida estructurada devuelve datos. Una tool call pide que tu aplicación ejecute algo. Esa diferencia parece pequeña y es enorme.

NECESIDAD	SALIDA ESTRUCTURADA	TOOL CALL
Clasificar un mensaje	Devuelve `{categoria, prioridad}`.	Normalmente no hace falta.
Consultar stock	Puede devolver intención de consulta.	Llama `consultar_stock(producto, talla)`.
Calcular una cuota	Puede proponer fórmula.	Llama `calcular_cuota(importe, plazo)`.
Abrir un ticket	Puede redactar el contenido.	Llama `crear_ticket(...)` si el usuario confirma.

La regla práctica: **si el dato existe fuera del modelo, no lo conviertas en adivinanza**. Define una tool pequeña, valida sus argumentos, ejecuta el código y devuelve el resultado como contexto para que el modelo lo explique.



Para entenderlo antes de tocar código

Pensemos en cuatro productos que parecen parecidos porque todos “usan IA”, pero que piden contratos distintos.

PRODUCTO	QUÉ ENVÍA A LA API	QUÉ ESPERA RECIBIR	PIEZA CRÍTICA
Clasificador de correos internos	Texto del correo e instrucciones.	JSON con cola, prioridad y motivo breve.	Schema y validador.
Tutor universitario	Pregunta, nivel del curso y rúbrica.	Explicación paso a paso.	Mensajes bien separados.
Asistente de matrícula	Pregunta y contexto normativo recuperado.	Respuesta con cita y quizá tool de expediente.	RAG, tool y trazabilidad.
Redactor con respuesta larga	Brief, tono y ejemplos.	Texto progresivo en pantalla.	Streaming y cancelación.

El mismo modelo puede participar en los cuatro, pero la API no se usa igual. En uno importa más el schema; en otro, el streaming; en otro, tools; en otro, trazabilidad del contexto.

Buenas prácticas de integración

Una integración madura no empieza llamando al SDK desde cualquier pantalla. Empieza con un contrato propio de la aplicación. Ese contrato dice: “para clasificar una solicitud necesito estos campos, esta política, este schema, estas tools permitidas y esta forma de guardar trazas”. Después un adaptador traduce ese contrato al proveedor elegido.

El adaptador evita que el resto del producto sepa si por debajo hay OpenAI, Claude, Gemini, un modelo local o un gateway. También te obliga a decidir lo importante en un sitio: versionar prompts, schemas y modelos; convertir errores de proveedor en errores propios; medir coste; registrar identificadores; y probar la misma tarea con casos de evaluación.

PRÁCTICA	QUÉ RESUELVE	CÓMO SE VE EN CÓDIGO O PRODUCTO
Adaptador por proveedor	Evita acoplar pantallas a nombres de campos externos.	`crearRespuestaTutor(...)` traduce a OpenAI, Claude o Gemini.
Schema versionado	Permite cambiar formato sin romper consumidores.	`respuesta_matricula.v2.json`.
Validador propio	No delega toda la corrección al proveedor.	Pydantic, Zod, JSON Schema o validación del backend.
Trazas mínimas	Permite reproducir fallos sin guardar más de lo necesario.	`trace_id`, modelo, versión de prompt, schema y resumen de entrada.
Timeouts y reintentos	Evita dejar al usuario esperando sin cierre.	Reintento solo en operaciones idempotentes.
Tools pequeñas	Reduce ambigüedad y facilita permisos.	`consultar_expediente(id)` en vez de `hacer_cosas(datos)`.
Evals antes de publicar	Mide si la integración mejora o empeora.	Conjunto de casos fijos con salida esperada.
Streaming con máquina de estados	Evita UI a medias.	`iniciado`, `parcial`, `tool`, `completo`, `cancelado`, `error`.

Los reintentos tienen una lógica que conviene cuantificar. Si una llamada falla de forma transitoria (un `429`, un `503`, un timeout) con probabilidad $1 - p$ en cada intento, y los fallos son independientes, la probabilidad de éxito tras k intentos es:

$$p_{\text{eff}} = 1 - (1 - p)^k$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
p	Probabilidad de éxito de un intento.	0,9.
k	Número de intentos permitidos.	3.
p_{eff}	Probabilidad de éxito tras los reintentos.	$1-(1-0{,}9)^3 = 0{,}999$.

Con $p = 0{,}9$, pasar de un intento a tres sube la fiabilidad del 90 % al 99,9 %. La trampa es la palabra **independientes**: la fórmula solo vale para fallos transitorios. Si la petición está mal formada o el schema es inválido, fallará igual las tres veces, y reintentar solo multiplica el coste. Por eso los reintentos van sobre operaciones idempotentes y sobre códigos de error transitorios (`429` , `500` , `502` , `503` , `504`), nunca sobre un `400` de validación.

La integración más limpia tiene forma de **pipeline cerrado**: la petición sale del producto, atraviesa cada capa y vuelve al producto ya convertida en algo validado, no en texto suelto.

```
producto -> contrato propio -> adaptador -> proveedor -> validador -> producto
```

PIEZA	PREGUNTA QUE RESPONDE
Producto	¿Qué necesita conseguir la persona usuaria?
Contrato propio	¿Qué datos, formato, tools y reglas exige nuestro flujo?
Adaptador	¿Cómo se expresa eso en la API concreta?
Proveedor	¿Qué modelo genera, pide tool o devuelve eventos?
Validador	¿La salida cumple estructura y criterio mínimo?
Producto	¿Mostramos, pedimos confirmación, guardamos o repetimos?

Cómo sería una API perfecta para integrar

Una API perfecta no sería “la que siempre acierta”. Eso no existe. Sería la que hace fácil construir software fiable alrededor de una capacidad probabilística. Si diseñáramos una interfaz ideal para una app profesional, tendría estas propiedades:

RASGO	POR QUÉ IMPORTA
Entrada multimodal tipada	Texto, imágenes y documentos no llegan como una cadena opaca.
Instrucciones separadas	Las reglas estables no se mezclan con lo que escribe la persona usuaria.
Salida schema-first	El contrato de datos se declara antes de generar.
Tools tipadas y pequeñas	La API distingue pedir una acción de ejecutar una acción.
Eventos normalizados	Streaming, tool calls y errores siguen una secuencia predecible.
Uso y coste visibles	La respuesta trae tokens, latencia y modelo usado.
Versionado explícito	Prompt, schema, modelo y toolset se pueden congelar y comparar.
Errores tipados	La app sabe si hubo límite, timeout, validación fallida o contenido incompleto.
Idempotencia	Reintentar no duplica acciones sensibles ni crea registros repetidos.
Privacidad configurable	Puedes decidir qué se guarda, qué se omite y durante cuánto tiempo.
Evals integradas	El mismo contrato puede probarse con casos antes de publicarse.

En pseudocódigo, una llamada ideal se parecería menos a “envía este texto” y más a esto:

```
respuesta = modelo.generar({
  tarea: "clasificar_solicitud_matricula",
  contrato: "solicitud_matricula.v2",
  entrada: {texto, documentos, usuario_contexto},
  salida: schema_respuesta,
  tools: [consultar_expediente],
  ejecucion: {stream: true, timeout_ms: 12000, trace_id},
  politica: {confirmar_antes_de_crear_ticket: true}
})
```

La clave no es que todos los proveedores adopten exactamente ese formato. La clave es que tu aplicación sí tenga esa claridad interna. Si tu dominio está bien modelado, cambiar de proveedor es una migración. Si tu dominio vive pegado al prompt, cambiar de proveedor es cirugía.

El contrato de OpenAI Responses, campo a campo

Antes de armar la práctica, conviene ver una petición real a la Responses API de OpenAI con sus campos principales, agrupados por las seis familias del registro *R*. Tómalo como un ejemplo de contrato: no para memorizarlo, sino para saber qué decide cada pieza. La referencia exacta y siempre actualizada vive en la documentación de OpenAI; aquí mostramos los campos estables y explicamos cada trozo.

```
peticion = {
    # Familia 1 (M): qué modelo
    "model": "modelo-vigente",

    # Familia 2 (I, C): qué entra
    "instructions": "Reglas estables del sistema, separadas del usuario.",
    "input": [
        {
            "role": "user",
            "content": [
                {"type": "input_text", "text": "Texto de la persona usuaria."},
                {"type": "input_file", "file_id": "file_normativa_2026"},
                {"type": "input_image", "image_url": "https://example.edu/captura.png"},
            ],
        },
    ],
    "previous_response_id": None,   # encadenar con una respuesta anterior, si la hubo

    # Familia 3 (g): cuánto y cómo responde
    "max_output_tokens": 900,
    "temperature": 0.2,
    "top_p": 0.9,
    "reasoning": {"effort": "medium"},   # solo en modelos de razonamiento
    "truncation": "auto",                # qué hacer si el contexto se desborda

    # Familia 4 (T): qué herramientas puede pedir
    "tools": [TOOL_CONSULTAR_EXPEDIENTE],
    "tool_choice": "auto",               # auto | none | required | una tool concreta
    "parallel_tool_calls": False,

    # Familia 5 (F): qué forma tiene la salida
    "text": {
        "format": {
            "type": "json_schema",
            "name": "clasificacion",
            "strict": True,
            "schema": RESPUESTA_SCHEMA,
        }
    },

    # Familia 6 (S): operación
    "stream": True,
    "store": False,                      # guardar o no la respuesta en el proveedor
    "include": [],                       # datos extra a incluir en la respuesta
    "metadata": {"trace_id": "trc_matricula_00042"},
}
```

Ahora cada trozo, por familia:

Qué modelo. `model` fija el modelo y su versión. No uses un alias que pueda cambiar bajo tus pies en producción: usa la versión exacta y guárdala en la traza.

Qué entra. `instructions` lleva las reglas estables del sistema, separadas del `input`, donde va la petición de la persona usuaria. El `input` no es una cadena: es una lista de mensajes con bloques de contenido tipados, `input_text` para texto, `input_file` para documentos e `input_image` para imágenes, que es como entra lo multimodal. `previous_response_id` permite encadenar con una respuesta anterior reutilizando estado del servidor, en vez de reenviar todo el historial.

Cuánto y cómo responde. `max_output_tokens` acota la longitud de salida (y el coste). `temperature` y `top_p` controlan la aleatoriedad; para una tarea estructurada se bajan. `reasoning` con su `effort` ajusta cuánto piensa un modelo de razonamiento antes de responder. `truncation` decide qué pasa si la entrada se pasa de contexto: recortar de forma automática o fallar.

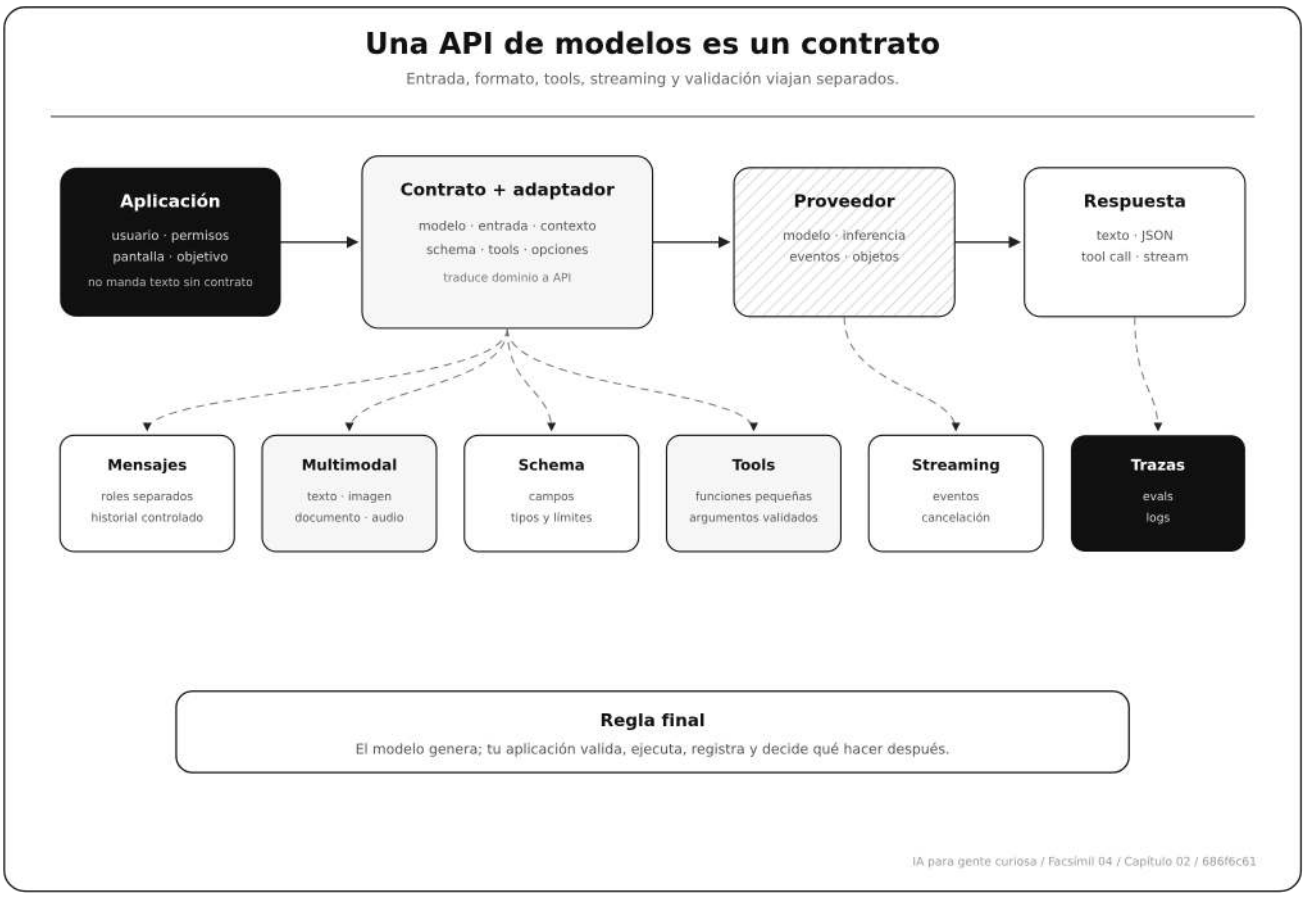
Qué herramientas. `tools` declara las funciones tipadas que el modelo puede pedir; `tool_choice` decide si puede elegir (`auto`), no debe usar ninguna (`none`), está obligado (`required`) o debe usar una concreta; `parallel_tool_calls` permite o no varias llamadas a la vez, que conviene desactivar cuando el orden importa.

Qué forma tiene la salida. El bloque `text.format` con `type: "json_schema"` es lo que convierte el texto en dato: `name` identifica el schema, `strict: True` activa el modo estricto y `schema` declara los campos. Combinado con `additionalProperties: false`, fija un contrato de mundo cerrado.

Operación. `stream` activa la entrega progresiva; `store` decide si el proveedor guarda la respuesta; `include` pide datos extra en la respuesta; `metadata` lleva tu `trace_id` y versiones para poder reproducir el caso. Y un matiz importante: `timeout`, la política de reintentos y la `idempotency_key` no viajan en este JSON, son parte de tu cliente HTTP o SDK, pero forman parte igual del contrato de operación.

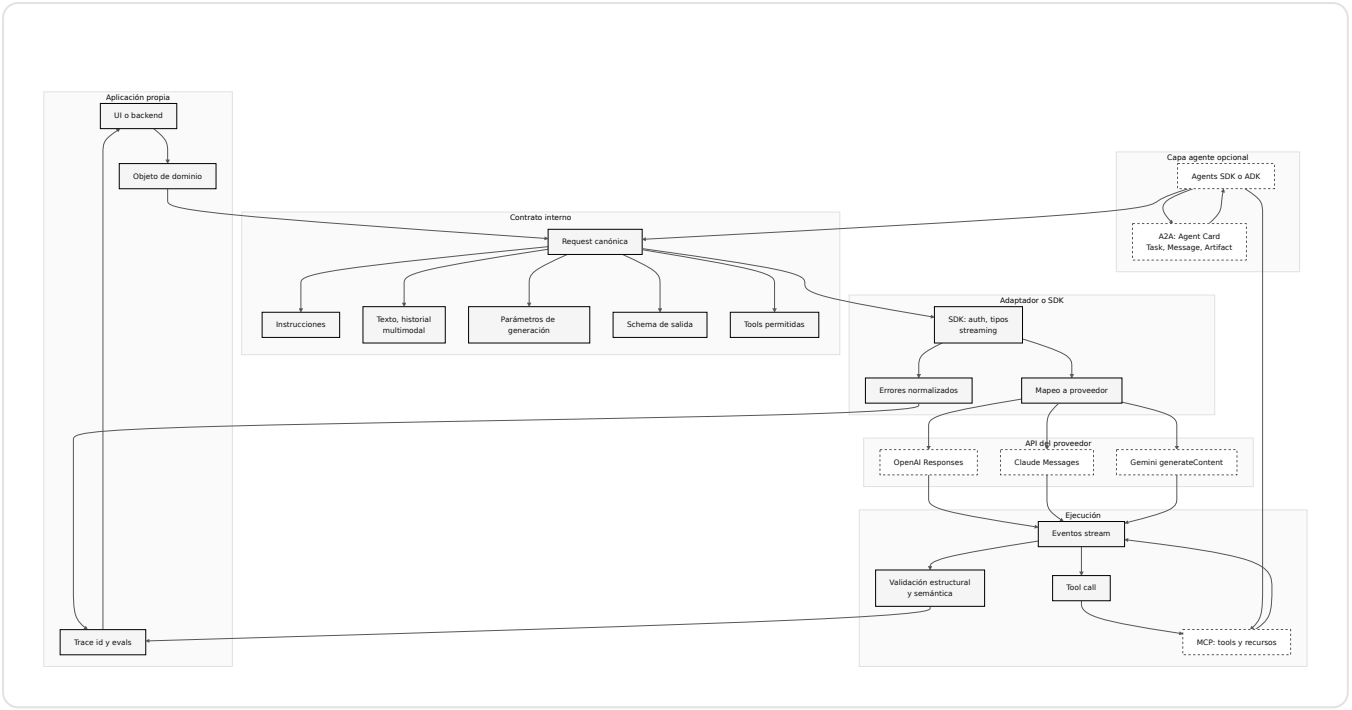
Mapa visual de una petición robusta

El diagrama resume la idea práctica del capítulo: la aplicación no debería hablar con el modelo como quien manda una frase suelta, sino como quien prepara un contrato que luego valida.



Mapa Mermaid: todo lo que viaja en una API

El SVG anterior da la intuición editorial. Ahora conviene ver la llamada como arquitectura técnica: qué construye tu aplicación, qué transforma el SDK o adaptador, qué recibe el proveedor y qué vuelve a tu sistema.



SDKs, ADK, MCP y A2A: cada cosa en su capa

Aquí suele nacer mucha confusión porque todo parece “la API”. No lo es. Una API es el contrato de red y datos. Un SDK es una biblioteca en tu lenguaje que envuelve esa API. Un framework de agentes es una capa que orquesta pasos, estado, tools y decisiones. Y un protocolo de interoperabilidad define cómo se comunican sistemas que quizá ni comparten proveedor ni framework.

OpenAI documenta SDKs oficiales para lenguajes como JavaScript/TypeScript y Python, pensados para llamar a la API desde código de aplicación.[27] El SDK no cambia la semántica: si el endpoint espera `input`, `tools`, `stream` o un schema, el SDK lo expresa con tipos, métodos, helpers de streaming, subida de archivos y manejo de errores. Es cómodo, pero no sustituye al diseño del contrato.

OpenAI también documenta Agents SDK para casos donde tu servidor posee la orquestación, la ejecución de tools, el estado y las aprobaciones del flujo.[28] Google ADK va en esa misma familia de herramientas de construcción de agentes: su documentación organiza piezas como agentes, equipos de agentes, workflows, ejecución, observabilidad, evaluación, tools, sesiones, memoria, artefactos, MCP y A2A.[29] La propia documentación de ADK incluye guías para exponer agentes a otros sistemas y consumir agentes remotos mediante A2A.[30]

MCP y A2A resuelven problemas distintos. ADK describe MCP como un estándar para que LLMs y agentes se comuniquen con aplicaciones externas, fuentes de datos y herramientas mediante recursos, prompts y tools.[31] A2A, en cambio, es para comunicación entre agentes: la documentación oficial lo presenta como un estándar abierto para interoperabilidad entre agentes construidos con distintos frameworks o proveedores.[32]

Técnicamente, A2A introduce piezas que no aparecen en una simple llamada a un modelo: `AgentCard`, `Message`, `Part`, `Task`, `TaskStatus`, `Artifact`, métodos para enviar mensajes, consultar tareas, cancelar, suscribirse a eventos y entregar resultados por streaming o notificaciones.[33] La `AgentCard` dice qué agente hay al otro lado, qué capacidades ofrece y cómo se accede. Un `Message` lleva partes de contenido; una `Task` representa trabajo con estado; un `Artifact` es una salida producida por el agente remoto.

CAPA	OBJETO PRINCIPAL	QUIÉN CONTROLA EL ESTADO	PARA QUÉ SIRVE
API de modelos	Request y response.	Tu aplicación y el proveedor.	Generar, estructurar, llamar tools o recibir eventos.
SDK	Cliente tipado del lenguaje.	Tu aplicación.	Autenticación, tipos, helpers, streaming y errores.
Agents SDK / ADK	Run, sesión, agente, workflow, tool context.	Tu servidor o runtime de agentes.	Orquestar pasos, tools, memoria, evaluación y observabilidad.
MCP	Tool, recurso, prompt.	Host de IA y servidor MCP.	Conectar agentes o apps a herramientas y datos externos.
A2A	Agent Card, Message, Task, Part, Artifact.	Agente cliente y agente remoto.	Delegar tareas entre agentes independientes y recibir progreso o resultados.

Para entenderlo con una situación concreta: si una app de la universidad pregunta a un modelo “clasifica esta solicitud”, quizá basta la API y un SDK. Si necesita consultar expediente, el modelo puede pedir una tool; esa tool puede venir de tu backend o de un servidor MCP. Si además hay un agente remoto especializado en normativa académica, tu agente podría descubrirlo mediante una `AgentCard`, enviarle un `Message`, recibir una `Task` en estado `working`, escuchar eventos y recoger un `Artifact` final. Ahí ya no estás “llamando a un modelo”: estás coordinando sistemas.

En el día a día

En un proyecto real, este capítulo aparece cuando alguien dice: “ya tenemos el prompt, vamos a integrarlo”. Ahí empieza el trabajo serio. Hay que decidir qué parte será configuración, qué parte será código, qué parte será schema y qué parte quedará en logs para poder depurar.

Si una respuesta alimenta otra pantalla, no basta con que “se lea bien”. Debe llegar como objeto fiable. Si una respuesta se muestra mientras se genera, debes pensar en streaming y cancelación. Si el modelo pide una tool, debes decidir quién ejecuta, con qué permisos, cómo

se registra y qué ocurre si faltan argumentos.

La integración buena suele tener una capa intermedia: tu aplicación habla en términos de dominio, y esa capa traduce a la API concreta. Así evitas que cada pantalla dependa de detalles de un proveedor.

Por qué debería importarte

Porque una mala integración convierte una capacidad potente en un sistema difícil de mantener. Si guardas texto sin estructura, mañana no podrás medir. Si no validas schema, el backend se rompe tarde. Si no separas mensajes, no sabrás qué instrucción produjo qué comportamiento. Si no registras eventos de streaming y tools, no podrás explicar por qué una respuesta salió como salió.

La buena noticia: una API bien tratada como contrato permite cambiar modelos, añadir RAG, introducir tools y medir calidad sin rehacer todo el producto.

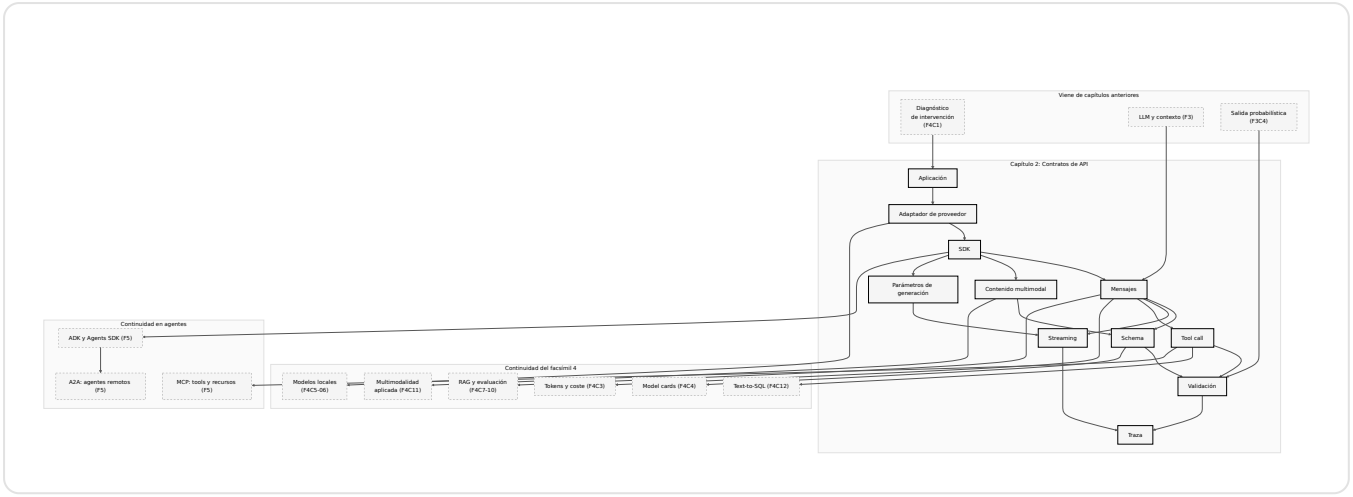
Dónde solía tropezar yo

Estos tropiezos aparecen cuando uno pasa de probar prompts a construir una aplicación que tiene que vivir.

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Meter todo en un único prompt	Instrucción, contexto, formato y datos se vuelven inseparables.	Construir mensajes y contexto por capas.
Validar solo que haya JSON	Un objeto puede estar bien formado y contener una decisión mala.	Separar validación estructural y semántica.
Confundir tool call con ejecución	Que el modelo pida una función no significa que deba ejecutarse sin más.	Validar argumentos y aplicar reglas de negocio antes de ejecutar.
Usar streaming sin estado claro	Si se corta el flujo, puedes dejar la UI o la traza a medias.	Diseñar estados: iniciado, parcial, completo, cancelado y error.
Mandar documentos sin referencia	La respuesta queda desligada del archivo, página o versión que la produjo.	Guardar identificador, páginas usadas y schema de extracción.
Acoplarse al proveedor demasiado pronto	Cada pantalla acaba hablando el dialecto de una API concreta.	Usar un contrato propio y adaptadores por proveedor.
Confundir SDK con arquitectura	El SDK facilita la llamada, pero no decide schemas, permisos, trazas ni evaluación.	Diseñar primero contrato y flujo; elegir SDK después.
Mezclar MCP y A2A	MCP conecta tools y recursos; A2A conecta agentes completos con tareas y artefactos.	Dibujar qué sistema habla con qué sistema antes de integrar.
No guardar trazas mínimas	Sin request, respuesta, schema y versión de modelo no puedes reproducir fallos.	Registrar lo necesario para depurar sin guardar datos innecesarios.

Cómo encaja todo

Este mapa sitúa el capítulo dentro del facsímil. El capítulo anterior decide qué intervención toca; este convierte esa decisión en contrato de API.



Vocabulario aprendido

Estos términos nos permiten hablar de integración sin mezclarlo todo en “el prompt”.

TÉRMINO	DEFINICIÓN
API de modelos	Interfaz para enviar entradas a un modelo y recibir salidas bajo un contrato técnico.
Mensaje	Pieza de conversación con rol y contenido.
Rol	Etiqueta que indica qué función cumple un mensaje dentro de la petición.
Streaming	Entrega progresiva de la respuesta por eventos o fragmentos.
Salida estructurada	Respuesta obligada a cumplir un schema.
Schema	Contrato que define campos, tipos y restricciones de una salida.
Tool call	Petición del modelo para que el sistema ejecute una función externa.
Validador	Código que comprueba si una salida cumple el contrato esperado.
Contenido multimodal	Entrada formada por texto, imágenes, documentos, audio o vídeo según lo permita el modelo.
Adaptador de proveedor	Capa que traduce el contrato propio de la aplicación al formato de una API concreta.
Tool choice	Opción que limita o fuerza qué herramienta puede pedir el modelo durante una llamada.
Idempotencia	Propiedad de repetir una operación sin duplicar efectos cuando hay reintentos.
SDK	Biblioteca cliente que envuelve una API desde un lenguaje concreto.
ADK	Framework para construir agentes con tools, sesiones, memoria, workflows y observabilidad.
MCP	Protocolo para conectar aplicaciones de IA con herramientas, recursos y contexto externos.
A2A	Protocolo para que agentes independientes se descubran, se envíen mensajes y coordinen tareas.
Agent Card	Documento de metadatos que publica identidad, endpoint, capacidades y requisitos de acceso de un agente.
Traza	Registro mínimo de lo enviado, recibido y validado para poder depurar.

Antes de pasar página

- [ ] ¿Puedo explicar por qué una API de modelos no es solo un prompt por HTTP? (Si no, vuelve a «Qué no es una API de modelos».)
- [ ] ¿Sé comparar OpenAI Responses API, Claude Messages API y Gemini API sin memorizar cada SDK? (Si no, vuelve a «Qué APIs se usan hoy y quién las usa».)
- [ ] ¿Reconozco las familias de parámetros: modelo, entrada, generación, tools, schema, streaming y operación? (Si no, vuelve a «Los parámetros: no memorizar nombres, entender familias».)
- [ ] ¿Sé separar instrucciones, mensaje de usuario, contexto recuperado, tool y salida? (Si no, vuelve a «Mensajes: separar instrucciones, contexto y petición».)
- [ ] ¿Entiendo qué cambia cuando la entrada incluye documentos o imágenes? (Si no, vuelve a «Documentos e imágenes: multimodal no significa comprensión automática».)
- [ ] ¿Puedo distinguir salida estructurada de tool call? (Si no, vuelve a «Tool calls: cuando responder no basta».)
- [ ] ¿Entiendo por qué JSON válido no significa decisión correcta? (Si no, vuelve a «Salidas estructuradas: cuando el texto debe convertirse en dato».)
- [ ] ¿Sé explicar cuándo streaming mejora experiencia y qué complica? (Si no, vuelve a «Streaming: que la respuesta llegue por partes».)
- [ ] ¿Puedo describir una capa de adaptador que proteja mi producto de cambios de proveedor? (Si no, vuelve a «Buenas prácticas de integración».)
- [ ] ¿Puedo distinguir API, SDK, Agents SDK, ADK, MCP y A2A sin meterlos en el mismo saco? (Si no, vuelve a «SDKs, ADK, MCP y A2A: cada cosa en su capa».)
- [ ] ¿Sé explicar qué son `AgentCard`, `Task`, `Message`, `Part` y `Artifact` dentro de A2A? (Si no, vuelve a «SDKs, ADK, MCP y A2A: cada cosa en su capa».)
- [ ] ¿Puedo diseñar un schema mínimo para una respuesta que usará otro software? (Si no, vuelve a «Salidas estructuradas: cuando el texto debe convertirse en dato».)
- [ ] ¿Sé diseñar un caso que el validador de salida estructurada deba rechazar y por qué? (Si no, repasa «Salidas estructuradas: cuando el texto debe convertirse en dato».)

En resumen

Una buena integración trata la API como contrato. El modelo produce una salida; tu aplicación decide cómo construir la petición, validar la respuesta y registrar lo necesario.

IDEA FUERZA	DETALLE
La API no es una caja de texto.	Es una frontera entre software probabilístico y software verificable.
Los proveedores cambian de dialecto, no de problema.	OpenAI, Claude y Gemini piden piezas parecidas con nombres y límites distintos.
Los mensajes separan responsabilidades.	Instrucciones, usuario, contexto y tools no deberían vivir en una sola cadena.
La multimodalidad exige trazabilidad.	Si entran documentos o imágenes, guarda fuente, versión, página y criterio de validación.
El schema convierte texto en dato.	Pero aún necesitas validar si el contenido tiene sentido.
Streaming mejora la experiencia percibida.	También obliga a manejar estados parciales y cancelación.
Una tool call no es ejecución automática.	La aplicación valida argumentos y decide qué hacer.
El adaptador protege tu producto.	Tu dominio debería hablar su propio contrato y traducirlo al proveedor.
SDK y ADK no son lo mismo.	El SDK llama APIs; ADK o Agents SDK orquestan agentes, tools, sesiones y workflows.
MCP y A2A resuelven fronteras distintas.	MCP conecta herramientas y recursos; A2A conecta agentes completos mediante tareas y artefactos.
Sin trazas no hay depuración seria.	Guardar contrato, versión y resultado ayuda a reproducir problemas.

Para saber más

Anthropic. (2026). *Messages API*. https://platform.claude.com/docs/en/api/messages

Anthropic. (2026). *PDF support*. https://platform.claude.com/docs/en/build-with-claude/pdf-support

Anthropic. (2026). *Streaming messages*. https://platform.claude.com/docs/en/build-with-claude/streaming

Anthropic. (2026). *Structured outputs*. https://platform.claude.com/docs/en/build-with-claude/structured-outputs

Anthropic. (2026). *Vision*. https://platform.claude.com/docs/en/build-with-claude/vision

A2A Protocol. (2026). *Agent2Agent (A2A) Protocol*. https://a2a-protocol.org/latest/

A2A Protocol. (2026). *Overview specification*. https://a2a-protocol.org/latest/specification/

Google. (2026). *ADK with Agent2Agent (A2A) Protocol*. https://adk.dev/a2a/

Google. (2026). *Agent Development Kit*. https://adk.dev/

Google. (2026). *Document understanding*. https://ai.google.dev/gemini-api/docs/document-processing

Google. (2026). *Image understanding*. https://ai.google.dev/gemini-api/docs/image-understanding

Google. (2026). *Model Context Protocol (MCP)*. https://adk.dev/mcp/

Google. (2026). *Structured outputs*. https://ai.google.dev/gemini-api/docs/structured-output

Google. (2026). *Text generation*. https://ai.google.dev/gemini-api/docs/text-generation

JSON Schema. (2020). *JSON Schema Validation: A Vocabulary for Structural Validation of JSON*. https://json-schema.org/draft/2020-12/json-schema-validation

OpenAI. (2026). *Create a model response*. https://developers.openai.com/api/docs/api-reference/responses/create

OpenAI. (2026). *File inputs*. https://developers.openai.com/api/docs/guides/file-inputs

OpenAI. (2026). *Function calling*. https://developers.openai.com/api/docs/guides/function-calling

OpenAI. (2026). *Images and vision*. https://developers.openai.com/api/docs/guides/images-vision

OpenAI. (2026). *Agents SDK*. https://developers.openai.com/api/docs/guides/agents

OpenAI. (2026). *SDKs and CLI*. https://developers.openai.com/api/docs/libraries

OpenAI. (2026). *Streaming API responses*. https://developers.openai.com/api/docs/guides/streaming-responses

OpenAI. (2026). *Structured model outputs*. https://developers.openai.com/api/docs/guides/structured-outputs

OpenAI. (2026). *Text generation*. https://developers.openai.com/api/docs/guides/text

WHATWG. (2026). *Server-sent events*. https://html.spec.whatwg.org/multipage/server-sent-events.html

Notas

1. Lista de documentos consultados el 10 de junio de 2026. OpenAI: *Create a model response* (https://developers.openai.com/api/docs/api-reference/responses/create), *Text generation* (https://developers.openai.com/api/docs/guides/text), *File inputs* (https://developers.openai.com/api/docs/guides/file-inputs), *Images and vision* (https://developers.openai.com/api/docs/guides/images-vision), *Structured model outputs* (https://developers.openai.com/api/docs/guides/structured-outputs), *Function calling* (https://developers.openai.com/api/docs/guides/function-calling), *Streaming API responses* (https://developers.openai.com/api/docs/guides/streaming-responses), *SDKs and CLI* (https://developers.openai.com/api/docs/libraries), *Agents SDK* (https://developers.openai.com/api/docs/guides/agents). Anthropic: *Messages API* (https://platform.claude.com/docs/en/api/messages), *Vision* (https://platform.claude.com/docs/en/build-with-claude/vision), *PDF support* (https://platform.claude.com/docs/en/build-with-claude/pdf-support), *Streaming messages* (https://platform.claude.com/docs/en/build-with-claude/streaming), *Structured outputs* (https://platform.claude.com/docs/en/build-with-claude/structured-outputs). Google: *Text generation* (https://ai.google.dev/gemini-api/docs/text-generation), *Document understanding* (https://ai.google.dev/gemini-api/docs/document-processing), *Image understanding* (https://ai.google.dev/gemini-api/docs/image-understanding), *Structured outputs* (https://ai.google.dev/gemini-api/docs/structured-output), *Agent Development Kit* (https://adk.dev/), *Model Context Protocol* (https://adk.dev/mcp/), *ADK with A2A* (https://adk.dev/a2a/). A2A Protocol: especificación oficial (https://a2a-protocol.org/latest/specification/). JSON Schema 2020-12: *Validation* (https://json-schema.org/draft/2020-12/json-schema-validation). WHATWG: *Server-sent events* (https://html.spec.whatwg.org/multipage/server-sent-events.html). ↩

2. OpenAI. (2026). *Text generation*. https://developers.openai.com/api/docs/guides/text. Consultado el 10 de junio de 2026. ↩

3. OpenAI. (2026). *Structured model outputs*. https://developers.openai.com/api/docs/guides/structured-outputs. Consultado el 10 de junio de 2026. ↩

4. OpenAI. (2026). *Function calling*. https://developers.openai.com/api/docs/guides/function-calling. Consultado el 10 de junio de 2026. ↩

5. Anthropic. (2026). *Messages API*. https://platform.claude.com/docs/en/api/messages. Consultado el 10 de junio de 2026. ↩

6. Anthropic. (2026). *Streaming messages*. https://platform.claude.com/docs/en/build-with-claude/streaming. Consultado el 10 de junio de 2026. ↩

7. Anthropic. (2026). *Structured outputs*. https://platform.claude.com/docs/en/build-with-claude/structured-outputs. Consultado el 10 de junio de 2026. ↩

8. Google. (2026). *Text generation*. https://ai.google.dev/gemini-api/docs/text-generation. Consultado el 10 de junio de 2026. ↩

9. Google. (2026). *Structured outputs*. https://ai.google.dev/gemini-api/docs/structured-output. Consultado el 10 de junio de 2026. ↩

.0. Google. (2026). *Interactions API overview*. https://ai.google.dev/gemini-api/docs/interactions/interactions-overview. Consultado el 10 de junio de 2026. ↩

.1. Google. (2026). *Agent Development Kit*. https://adk.dev/. Consultado el 10 de junio de 2026. ↩

.2. A2A Protocol. (2026). *Agent2Agent Protocol*. https://a2a-protocol.org/latest/. Consultado el 10 de junio de 2026. ↩

.3. OpenAI. (2026). *Create a model response*. https://developers.openai.com/api/docs/api-reference/responses/create. Consultado el 10 de junio de 2026. ↩

.4. Anthropic. (2026). *Messages API*. https://platform.claude.com/docs/en/api/messages. Consultado el 10 de junio de 2026. ↩

.5. Google. (2026). *Text generation*. https://ai.google.dev/gemini-api/docs/text-generation. Consultado el 10 de junio de 2026. ↩

.6. OpenAI. (2026). *File inputs*. https://developers.openai.com/api/docs/guides/file-inputs. Consultado el 10 de junio de 2026. ↩

.7. OpenAI. (2026). *Images and vision*. https://developers.openai.com/api/docs/guides/images-vision. Consultado el 10 de junio de 2026. ↩

.8. Anthropic. (2026). *Vision*. https://platform.claude.com/docs/en/build-with-claude/vision. Consultado el 10 de junio de 2026. ↩

.9. Anthropic. (2026). *PDF support*. https://platform.claude.com/docs/en/build-with-claude/pdf-support. Consultado el 10 de junio de 2026. ↩

!0. Google. (2026). *Image understanding*. https://ai.google.dev/gemini-api/docs/image-understanding. Consultado el 10 de junio de 2026. ↩

!1. Google. (2026). *Document understanding*. https://ai.google.dev/gemini-api/docs/document-processing. Consultado el 10 de junio de 2026. ↩

!2. JSON Schema. (2020). *JSON Schema Validation: A Vocabulary for Structural Validation of JSON*. https://json-schema.org/draft/2020-12/json-schema-validation. ↩

!3. WHATWG. (2026). *Server-sent events*. https://html.spec.whatwg.org/multipage/server-sent-events.html. Consultado el 10 de junio de 2026. ↩

24. OpenAI. (2026). *Streaming API responses*. https://developers.openai.com/api/docs/guides/streaming-responses. Consultado el 10 de junio de 2026. ↩

25. Anthropic. (2026). *Streaming messages*. https://platform.claude.com/docs/en/build-with-claude/streaming. Consultado el 10 de junio de 2026. ↩

26. Woosuk Kwon y otros (2023). *Efficient Memory Management for Large Language Model Serving with PagedAttention*. SOSP. https://arxiv.org/abs/2309.06180. La métrica TTFT y el ritmo de decodificación se formalizan en la literatura de serving, p. ej. Yinmin Zhong y otros (2024). *DistServe*. OSDI 2024. https://arxiv.org/abs/2401.09670. ↩

27. OpenAI. (2026). *SDKs and CLI*. https://developers.openai.com/api/docs/libraries. Consultado el 10 de junio de 2026. ↩

28. OpenAI. (2026). *Agents SDK*. https://developers.openai.com/api/docs/guides/agents. Consultado el 10 de junio de 2026. ↩

29. Google. (2026). *Agent Development Kit*. https://adk.dev/. Consultado el 10 de junio de 2026. ↩

30. Google. (2026). *ADK with Agent2Agent (A2A) Protocol*. https://adk.dev/a2a/. Consultado el 10 de junio de 2026. ↩

31. Google. (2026). *Model Context Protocol (MCP)*. https://adk.dev/mcp/. Consultado el 10 de junio de 2026. ↩

32. A2A Protocol. (2026). *Agent2Agent (A2A) Protocol*. https://a2a-protocol.org/latest/. Consultado el 10 de junio de 2026. ↩

33. A2A Protocol. (2026). *Overview specification*. https://a2a-protocol.org/latest/specification/. Consultado el 10 de junio de 2026. ↩

Cuadernos para practicar

Escanea el código con el móvil para abrir el cuaderno en Google Colab y ejecutarlo sin instalar nada.

Salidas estructuradas: que el modelo rellene tu formulario

https://colab.research.google.com/github/686f6c61/iaparagentecuriosa-notebooks/blob/main/Facs%C3%ADmil%2004/05-salidas-estructuradas.ipynb