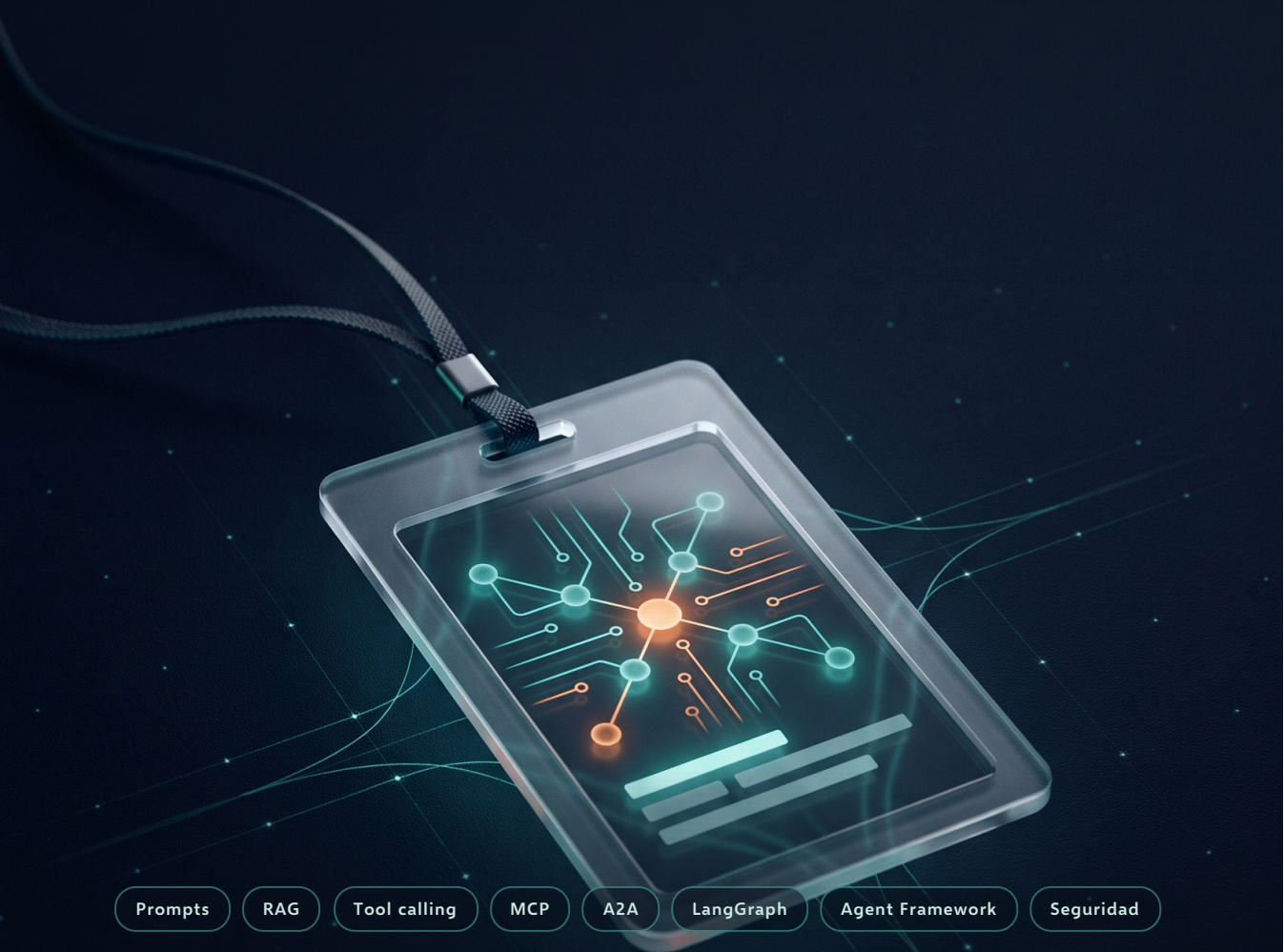
ROGER CARIAS

Contratar
a una máquina

Cómo diseñar, construir y gobernar
agentes de IA y sistemas multiagente



Prompts · RAG · Tool calling · MCP · A2A · LangGraph · Agent Framework · Seguridad

16	62	109	2026
CAPÍTULOS	DIAGRAMAS	PRUEBAS DE CÓDIGO	EDICIÓN

Contratar a una máquina

ROGER CARIAS

Contratar a una máquina

Cómo diseñar, construir y gobernar agentes de IA y sistemas multiagente

Fundador y CTO de Bitsideas · Microsoft Certified Trainer

Contratar a una máquina
Cómo diseñar, construir y gobernar agentes de IA y sistemas multiagente

© 2026 Roger Carias. Todos los derechos reservados. Primera edición: octubre de 2026.

Ninguna parte de este libro puede reproducirse, almacenarse o transmitirse por ningún medio sin autorización escrita del autor, salvo citas breves con mención de la fuente. Los fragmentos de código pueden reutilizarse en proyectos propios.

Versiones de referencia verificadas en octubre de 2026: LangChain 1.4, LangGraph 1.2, Microsoft Agent Framework 1.20, SDK de MCP para Python 1.30 (API v1), a2a-sdk 1.2, azure-ai-projects 2.7. Las bibliotecas de agentes cambian rápido; el repositorio acompañante fija las versiones con las que se ejecutó cada ejemplo.

Microsoft, Azure, Microsoft 365, Copilot, Copilot Studio, Foundry, Entra, Purview y Power Platform son marcas de Microsoft Corporation. OpenAI, Anthropic, Claude, LangChain, LangGraph, LangSmith y las demás marcas citadas pertenecen a sus respectivos titulares. Su mención es informativa y no implica patrocinio.

Este libro describe técnicas de ingeniería. Los ejemplos de empresas son casos compuestos e ilustrativos salvo que se indique lo contrario. Antes de poner un agente en producción, evalúalo con tus datos, tus riesgos y tu marco regulatorio.

Diseño editorial, diagramas y maquetación: Bitsideas.
bitsideas.com

PARA EDITH, RONALD, PILAR, LUCIANA Y THOMAS

Escribí un libro entero sobre qué delegar en una máquina.

Lo que ustedes me dan no se delega.

CONTENIDO

Índice

Cómo leer este libro 8

PARTE I · FUNDAMENTOS

01 Del chatbot al agente 12

02 ¿De verdad necesitas un agente? 30

03 El motor: modelos, tool calling y salidas estructuradas 50

04 Prompts para agentes: de la ingeniería de prompts a la ingeniería de contexto 68

PARTE II · CONSTRUIR UN AGENTE QUE FUNCIONE

05 Tu primer agente desde cero, sin framework 92

06 RAG para agentes: el manual del empleado 113

07 Que no alucine 133

08 Memoria y estado: la libreta del agente 152

09 Herramientas e integraciones: los permisos del empleado 172

PARTE III · FRAMEWORKS Y SISTEMAS MULTIAGENTE

10 LangChain y LangGraph a fondo 195

11 El stack Microsoft: Agent Framework, Foundry y Copilot Studio 218

12 Patrones multiagente 239

13 Protocolos y ecosistema: MCP, A2A y los demás frameworks 259

PARTE IV · PRODUCCIÓN

14 Evaluar y observar agentes 281

15 Seguridad y gobierno de agentes 301

16 Casos de uso de punta a punta y paso a producción 322

APÉNDICES

A Recetario: prompts, esquemas y plantillas 345

B Glosario, comparativa de frameworks y repositorio 378

Sobre el autor 400

Cómo leer este libro

La primera vez que puse un agente frente a clientes reales no falló el modelo. Falló todo lo demás: una herramienta que devolvía errores que el modelo no entendía, un documento escaneado que el índice convirtió en ruido, un bucle que nadie había limitado y un prompt que en las pruebas funcionaba porque las pruebas eran tres. El modelo hizo exactamente lo que le pedimos. El problema es que le pedimos mal.

Este libro nace de esa lección. Trata a un agente de IA como lo que en la práctica es: un empleado digital que hay que contratar bien. Tiene una descripción del cargo (sus instrucciones), un perfil (el modelo), permisos (las herramientas), un manual de la empresa (el conocimiento que recupera), una libreta (la memoria), un jefe (el orquestador), un periodo de prueba (la evaluación) y un reglamento interno (los guardarraíles). Cuando una de esas piezas está mal diseñada, ningún modelo, por grande que sea, la compensa.

Para quién es

Para desarrolladores, arquitectos, ingenieros de datos y líderes técnicos que ya programan y quieren construir agentes que aguanten producción. Necesitas Python de nivel intermedio, saber qué es una API REST y leer JSON sin esfuerzo. No necesitas experiencia previa con modelos de lenguaje: el capítulo 3 explica desde cero cómo funciona el tool calling por dentro.

Cómo está organizado

La **Parte I** fija los fundamentos y la pregunta incómoda que casi nadie hace: si tu problema necesita un agente o le basta un flujo. La **Parte II** construye un agente pieza por pieza, primero sin framework para que entiendas el bucle y después con las herramientas que usarás en el trabajo: RAG, defensas contra la alucinación, memoria e integraciones con MCP. La **Parte III** recorre los frameworks (LangChain y LangGraph, el stack de Microsoft con Agent Framework, Foundry y Copilot Studio) y los patrones multiagente con sus protocolos. La **Parte IV** es el paso a producción: evaluación, observabilidad, seguridad, gobierno y cuatro casos completos de punta a punta. Los apéndices son un recetario de prompts y esquemas listos para copiar, un glosario y la guía del repositorio de código.

Puedes leerlo en orden o saltar al capítulo que necesites; cada uno remite a los demás cuando hace falta. Si vas con prisa, lee el 2, el 7 y el 15 antes de poner nada en producción.

El código

Todo el código del libro se ejecutó antes de imprimirse, con las versiones fijadas que verás en el apéndice B. Para que puedas correrlo sin pagar un solo token, cada ejemplo detecta si tienes credenciales: si las tienes, usa el modelo real; si no, usa un modelo simulado con un guion de respuestas, y el resto (herramientas, grafos, validadores, middleware, servidores MCP y A2A) se ejecuta de verdad. Los bloques con nombre de archivo en la cabecera corresponden a archivos del repositorio acompañante, donde también está la salida real de cada ejecución.

Los nombres de modelos no aparecen fijos en el código: salen de una variable de entorno. Los modelos cambian cada pocos meses y no quiero que el libro envejezca por una cadena de texto.

Las piezas recurrentes

MODELO PROPIO
Una herramienta de criterio con nombre para decidir y explicar decisiones. Hay uno por capítulo.

EN LA PRÁCTICA
La lista de verificación que cierra cada capítulo: lo que harías el lunes.

DESDE LA TRINCHERA
Experiencia de proyectos reales, contada sin nombres de clientes.

CUIDADO
Trampas técnicas, de seguridad o de costo que salen caras si las descubres en producción.

EN MICROSOFT
El servicio concreto en Azure, Foundry, Copilot Studio o Microsoft 365.

CÓDIGO EJECUTABLE
Bloques con nombre de archivo: están en el repositorio y se ejecutaron antes de imprimir.

DIAGRAMAS
Arquitecturas, secuencias y matrices para ver lo que el texto solo describiría.

PROMPTS COMPLETOS
Instrucciones de sistema listas para copiar, en su propio formato.

Una advertencia honesta

Escribo en octubre de 2026 y este campo se mueve rápido. Verifiqué versiones, nombres de clases, estados de disponibilidad y cifras contra fuentes primarias, y cuando algo está en vista previa lo digo. Aun así, algunas firmas de API cambiarán antes de que termines de leer. Por eso el libro pone el énfasis en lo que dura: el criterio para decidir qué contratar, cómo supervisarlo y cuándo no darle las llaves.

Los modelos propios del libro

CAP.	MODELO	CAPÍTULO
01	**Anatomía en siete piezas**	Del chatbot al agente
02	**La prueba del becario**	¿De verdad necesitas un agente?
03	**El contrato de herramienta**	El motor: modelos, tool calling y salidas estructuradas
04	**Prompt en capas**	Prompts para agentes: de la ingeniería de prompts a la ingeniería de contexto
05	**El bucle mínimo viable**	Tu primer agente desde cero, sin framework
06	**RAG en cuatro anillos**	RAG para agentes: el manual del empleado
07	**Las seis defensas**	Que no alucine
08	**Memoria en tres tiempos**	Memoria y estado: la libreta del agente
09	**Permiso mínimo, efecto confirmado**	Herramientas e integraciones: los permisos del empleado
10	**Grafo antes que prompt**	LangChain y LangGraph a fondo
11	**Escalera low-code → pro-code**	El stack Microsoft: Agent Framework, Foundry y Copilot Studio
12	**El organigrama mínimo**	Patrones multiagente
13	**Mapa de interoperabilidad**	Protocolos y ecosistema: MCP, A2A y los demás frameworks
14	**Pirámide de evaluación de agentes**	Evaluar y observar agentes
15	**El perímetro del agente**	Seguridad y gobierno de agentes
16	**Del piloto a la nómina en 90 días**	Casos de uso de punta a punta y paso a producción

Roger Carias

Bogotá, octubre de 2026

PARTE I

1

Fundamentos

Qué estás contratando

01 Del chatbot al agente

02 ¿De verdad necesitas un agente?

03 El motor: modelos, tool calling y salidas estructuradas

04 Prompts para agentes: de la ingeniería de prompts a la ingeniería de contexto

01

Del chatbot al agente

«Agente» se volvió la palabra que todo el mundo usa y casi nadie define. Este capítulo la define con precisión de ingeniero, separa al agente de sus parientes cercanos y resuelve el mismo problema tres veces para que veas, con código ejecutado, qué cambia cuando le das al modelo el control del siguiente paso.

Un lunes a las ocho de la mañana llega un ticket al equipo de soporte: un cliente asegura que el asistente de la tienda le confirmó que su pedido «llegaba en dos o tres días» y que podía cambiar la dirección «desde la sección Mis pedidos». Ninguna de las dos cosas era verdad. El pedido seguía en bodega y esa sección no existe. El asistente no tenía acceso a ningún sistema; era un modelo de lenguaje con un prompt amable, y respondió lo que un texto de atención al cliente suele decir. Nadie le había dado los medios para averiguarlo.

Ese ticket resume la distancia entre un chatbot y un agente. El chatbot genera texto plausible. El agente consulta, decide, actúa y comprueba lo que pasó. En medio hay un espectro entero de diseños, y elegir mal el punto del espectro es la causa más común de proyectos que se ven brillantes en la demo y se caen en la segunda semana de producción. La tesis de este libro es que un agente se parece más a un empleado que contratas que a un software que instalas: necesita una descripción de cargo, permisos, un manual, una libreta, un jefe y un reglamento. Antes de contratar a alguien, conviene saber exactamente qué puesto estás llenando.

Qué es un agente (y qué no)

La definición operativa que uso en todo el libro es corta: **un agente es un sistema en el que un modelo de lenguaje decide, en un bucle, qué acción tomar a continuación, ejecuta esa acción mediante herramientas y usa el resultado para decidir la siguiente, hasta que la**

tarea termina o se alcanza un límite. La documentación de LangChain lo dice de forma todavía más compacta: un agente es un modelo que llama herramientas en un bucle hasta completar la tarea. Anthropic, en su artículo «Building effective agents», lo describe como modelos que usan herramientas a partir de la retroalimentación del entorno, en un bucle.

Las tres definiciones coinciden en dos elementos, y los dos son obligatorios. El primero es el **bucle**: una sola llamada al modelo, por buena que sea, no es un agente. El segundo es la **decisión del modelo sobre el flujo de control**: si tu código decide que primero se consulta el pedido y después se redacta la respuesta, tienes un programa que usa un modelo, no un agente. La guía práctica de OpenAI para construir agentes traza la misma frontera desde el otro lado y excluye de la categoría a los chatbots, a las aplicaciones de una sola llamada y a los clasificadores que integran un modelo sin dejarle gobernar la ejecución.

Que algo no sea un agente no es un insulto. Muchas veces es una virtud, y el capítulo 2 está dedicado entero a decidir cuándo no hace falta uno.

El bucle percibir, razonar, actuar, observar

Mira la figura 1.1. El ciclo tiene cuatro tiempos y cada uno corresponde a código concreto que vas a escribir o que un framework escribe por ti.

1. **Percibir.** El runtime arma el contexto que verá el modelo: instrucciones, historial de la conversación, resultados de herramientas anteriores y, si aplica, fragmentos recuperados de una base de conocimiento. Lo que no entra en esa ventana no existe para el modelo.
2. **Razonar.** El modelo lee el contexto y produce una de dos cosas: texto final para el usuario o una o varias solicitudes de herramienta (*tool calls*), que son objetos JSON con un nombre de función y unos argumentos.
3. **Actuar.** Tu código, no el modelo, valida esa solicitud y ejecuta la función. Aquí viven los permisos, las reglas de negocio y los efectos sobre el mundo real.
4. **Observar.** El resultado de la herramienta, incluido un error, vuelve al contexto como un mensaje más. Con eso empieza la siguiente vuelta.

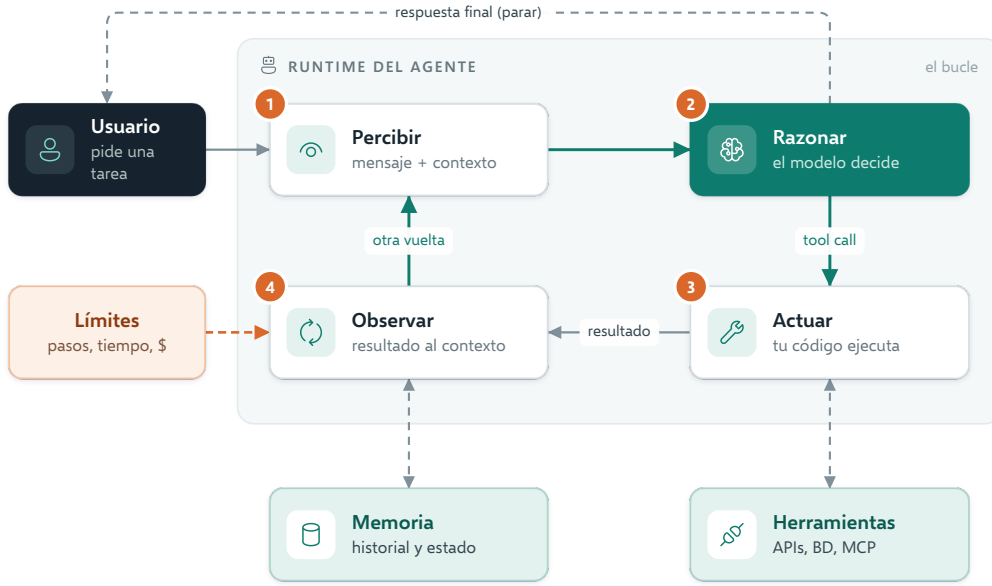


Figura 1.1. El bucle del agente. El modelo solo participa en el paso 2; percibir, actuar y observar son código del runtime. La flecha punteada superior es la salida normal (el modelo responde sin pedir herramientas) y los límites de pasos, tiempo y dinero son la salida de emergencia.

Dos detalles de la figura importan más de lo que parece. El primero es que el bucle tiene dos salidas. La normal ocurre cuando el modelo responde con texto y no pide ninguna herramienta. La de emergencia la pones tú: un número máximo de pasos, un tiempo máximo o un presupuesto en dinero. Anthropic recomienda explícitamente incluir condiciones de parada como un máximo de iteraciones para mantener el control, y no es un consejo decorativo. Un agente sin límite de pasos que recibe un error ambiguo de una API puede reintentar la misma llamada toda la noche; la historia del bucle que gastó cuarenta dólares mientras el equipo dormía se repite en casi todas las empresas que conozco.

El segundo detalle es la frontera entre razonar y actuar. El modelo nunca ejecuta nada. Propone. Esa separación es la que te permite poner controles: validar argumentos, pedir aprobación humana antes de un pago, rechazar una herramienta que no está en la lista. El capítulo 3 abre esa caja y muestra el JSON exacto que viaja en cada formato de API, y el capítulo 5 construye este mismo bucle en unas 160 líneas de Python sin framework.

Cuatro diseños para un mismo problema

Para hablar con precisión del espectro de autonomía conviene separar cuatro diseños que en las presentaciones comerciales suelen aparecer mezclados bajo la etiqueta «IA conversacional».

El LLM solo. Una llamada: entra un prompt, sale texto. Sirve para redactar, resumir, clasificar o traducir cuando todo lo necesario está en el prompt. No sabe nada que no le hayas dado y no puede tocar ningún sistema.

El asistente con RAG. Antes de llamar al modelo, tu código busca fragmentos relevantes en una base documental y los pega en el contexto (*retrieval-augmented generation*, generación aumentada por recuperación). Responde con datos de tu empresa, pero sigue siendo una tubería fija: buscar y luego responder, siempre en ese orden y una sola vez. Es el caso más común de «chatbot corporativo» y lo tratamos a fondo en el capítulo 6.

El flujo (workflow). Varios pasos, algunos con modelo y otros sin él, encadenados por código que tú escribiste. Anthropic lo define como un sistema en el que modelos y herramientas se orquestan mediante rutas de código predefinidas. El modelo puede clasificar, extraer o redactar dentro de un paso, pero el orden de los pasos y las bifurcaciones los decide tu programa.

El agente. El mismo artículo lo define por contraste: sistemas en los que el modelo dirige de forma dinámica su propio proceso y el uso de herramientas. El número de pasos no se conoce de antemano; lo descubre el modelo sobre la marcha.

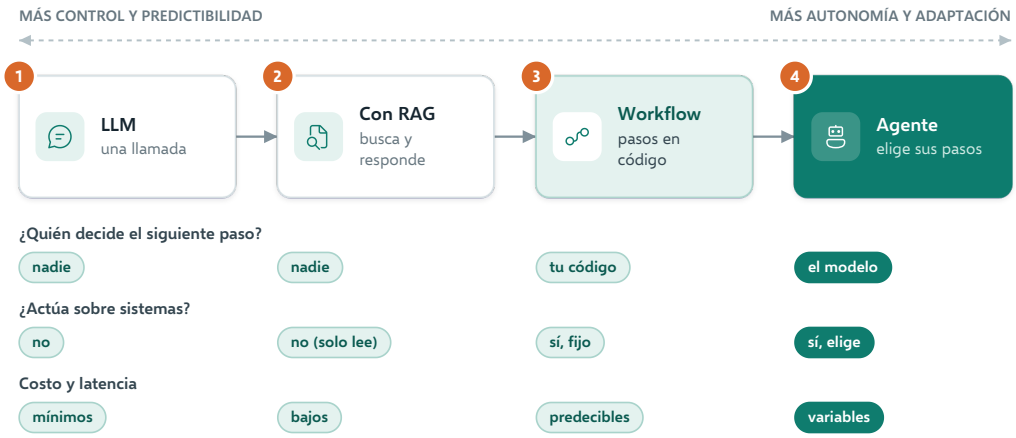


Figura 1.2. El espectro de autonomía. Moverse a la derecha compra adaptación a cambio de predictibilidad, costo y latencia. La pregunta que separa al flujo del agente es quién decide el siguiente paso.

La figura 1.2 ordena los cuatro diseños en un eje. Fíjate en que el salto conceptual grande no está entre el LLM y el RAG, ni entre el RAG y el flujo, aunque esos saltos añaden datos y acciones. Está entre el flujo y el agente, porque ahí cambia quién tiene el control. Anthropic insiste en que la mayoría de los casos se resuelven bien con los primeros escalones y en que los sistemas agénticos suelen cambiar latencia y costo por mejor desempeño en la tarea, un intercambio que hay que justificar. Microsoft llega a la misma conclusión en la documentación de Agent Framework con una regla todavía más tajante: si puedes escribir una función que resuelva la tarea, hazlo en lugar de usar un agente.

Tabla 1.1. Los cuatro diseños frente a frente.

DISEÑO	QUIÉN DECIDE EL PASO	FUERTE EN	FALLA TÍPICA	ÚSALO CUANDO
LLM solo	Nadie (un paso)	Redactar, resumir	Inventa datos que no tiene	Todo cabe en el prompt
Asistente con RAG	Nadie (tubería fija)	Preguntas sobre documentos	Recupera mal y responde seguro	Solo hay que leer, no actuar
Flujo	Tu código	Procesos conocidos, auditables	Se rompe con casos no previstos	Puedes dibujar el proceso
Agente	El modelo	Tareas abiertas, pasos variables	Bucles, acciones equivocadas, costo	Los pasos dependen de lo que encuentre

El mismo problema, resuelto tres veces

Las definiciones se entienden mejor con código. Tomemos la pregunta del ticket del lunes: «¿Dónde está mi pedido A-102? ¿Puedo cambiar la dirección a Calle 45 # 12-30?». Es una pregunta pequeña con una trampa de negocio: la dirección solo se puede cambiar si el pedido todavía no salió de bodega. El repositorio del capítulo (`code/ch01/`) trae un backend de juguete con dos pedidos: A-102, en preparación, y A-215, ya despachado. Esa regla vive en una función de Python, `cambiar()`, y las tres versiones la comparten para que la comparación sea justa.

Todo el código del capítulo se ejecutó con el modelo simulado del libro (`code/common/fake_llm.py`), que funciona sin claves de API. Conviene ser honesto sobre lo que eso significa: el texto que «escribe» el modelo sale de un guion, pero el resto es real. La extracción del número de pedido, las herramientas, la regla de negocio, el bucle de `create_agent` y los cambios en la base de datos se ejecutan de verdad. Si defines `OPENAI_API_KEY` y `MODEL` (o las variables de Azure OpenAI), el mismo código usa un modelo real y el guion se ignora.

Versión 1: un prompt

La primera versión es el asistente del ticket: un system prompt de atención al cliente y la pregunta del usuario. Una llamada, ningún dato.

PYTHON ch01/v1_prompt.py

```python
def responder(pregunta: str) -> dict:
    # Guion del modelo simulado: lo que suele hacer un modelo sin datos,
    # contestar con seguridad algo que no puede saber.
    modelo = get_chat_model(guion(
        "Tu pedido está en camino y debería llegar en 2 o 3 días. "
        "Sí, puedes cambiar la dirección desde la sección «Mis pedidos»."
    ))
    respuesta = modelo.invoke([
        {"role": "system", "content": SISTEMA},
        {"role": "user", "content": pregunta},
    ])
    return {"respuesta": respuesta.content, "llamadas_modelo": 1,
            "herramientas": []}
```

El guion reproduce lo que un modelo suele hacer en esta situación, y lo hace a propósito: contesta con seguridad algo que no puede saber. Un modelo real bien entrenado a veces dirá «no tengo acceso a tu pedido», y eso ya es una mejora, pero ni siquiera en el mejor caso resuelve la pregunta. El problema no se arregla con un prompt mejor. Al modelo le faltan datos y le faltan manos.

Versión 2: un flujo con un paso de modelo

La segunda versión invierte el reparto de responsabilidades. El código hace todo lo que se puede hacer de forma determinista: extrae el número de pedido con una expresión regular, consulta el sistema, detecta si el cliente pidió un cambio y, si es así, llama a `cambiar()`, que aplica la regla. Al modelo solo le queda un trabajo: redactar una respuesta amable a partir de un JSON con datos verificados.

PYTHON ch01/v2_flujo.py

```python
def responder(pregunta: str, guion_llm: str) -> dict:
    pasos: list[str] = []
    # Paso 1 (código): extraer el número de pedido. Sin él, no seguimos.
    pedido_id = extraer_id(pregunta)
    if pedido_id is None:
        return {"respuesta": "¿Me compartes el número de pedido (A-123)?",
                "llamadas_modelo": 0, "herramientas": []}
    # Paso 2 (código): consultar el sistema de pedidos.
    datos = consultar(pedido_id)
    pasos.append("consultar")
    # Paso 3 (código): si pide cambio y trae dirección, aplicar la regla.
    m = PATRON_DIRECCION.search(pregunta)
    if "cambiar" in pregunta.lower() and m:
        datos["cambio"] = cambiar(pedido_id, m.group(1).strip())
        pasos.append("cambiar")
    # Paso 4 (LLM): redactar con los datos, sin decidir nada.
    modelo = get_chat_model(guion(guion_llm))
    texto = modelo.invoke([
        {"role": "system", "content": "Redacta una respuesta breve al "
         "cliente usando SOLO estos datos JSON. No inventes nada."},
        {"role": "user", "content": json.dumps(datos, ensure_ascii=False)},
    ]).content
    return {"respuesta": texto, "llamadas_modelo": 1, "herramientas": pasos}
```

Este diseño es aburrido en el mejor sentido. Hace una sola llamada al modelo, su costo es predecible y el orden de los pasos se puede auditar línea por línea. Si mañana el área legal pregunta por qué se cambió una dirección, la respuesta está en el código y no en la interpretación de un prompt. Su debilidad también es evidente: la extracción de la dirección depende de una expresión regular que espera «a Calle…» o «por Cra….». Un cliente que escriba «me mudé, mándalo a donde mi mamá en Chapinero» queda fuera del camino feliz, y cada variante nueva exige una rama nueva.

Versión 3: un agente con dos herramientas

La tercera versión le entrega el control al modelo. Le damos instrucciones, dos herramientas (`consultar_pedido` y `cambiar_direccion`) y dejamos que `create_agent`, la función de LangChain 1.x que sustituye al antiguo `create_react_agent`, monte el bucle.

PYTHON ch01/v3_agente.py

```python
INSTRUCCIONES = """Eres el agente de soporte de pedidos de una tienda.
- Antes de afirmar algo sobre un pedido, consúltalo con consultar_pedido.
- Solo cambia la dirección si el cliente da la nueva dirección completa.
- Si una herramienta devuelve un error, explícalo; no lo maquilles.
- Si no tienes el número de pedido, pídelo. No lo adivines."""


@tool
def consultar_pedido(pedido_id: str) -> dict:
    """Consulta estado, transportadora, fecha estimada y dirección
    de un pedido. pedido_id tiene el formato A-123."""
    return consultar(pedido_id)


@tool
def cambiar_direccion(pedido_id: str, nueva_direccion: str) -> dict:
    """Cambia la dirección de entrega. Solo funciona si el pedido
    todavía no fue despachado; si no, devuelve ok=False y el motivo."""
    return cambiar(pedido_id, nueva_direccion)


def construir_agente(turnos_simulados):
    modelo = get_chat_model(guion(*turnos_simulados))
    return create_agent(modelo, tools=[consultar_pedido, cambiar_direccion],
                        system_prompt=INSTRUCCIONES)
```

Hay tres decisiones de diseño en ese fragmento que se repetirán en todo el libro. Las instrucciones dicen cuándo usar cada herramienta y qué hacer ante un error o un dato faltante; son la descripción del cargo, y el capítulo 4 enseña a escribirlas por capas. Los docstrings de las herramientas no son comentarios para humanos: LangChain los convierte en la descripción que el modelo lee para decidir si llama la función, así que son parte del prompt. Y la regla de negocio no está en las instrucciones, está dentro de `cambiar()`. Le pedimos al modelo que se comporte, pero no dependemos de que lo haga.

La figura 1.3 muestra la traza real de la ejecución para el pedido A-102. El modelo pide consultar el pedido, ve que está en preparación, pide cambiar la dirección, recibe `ok: true` y solo entonces responde con texto, lo que cierra el bucle. Son tres llamadas al modelo frente a una del flujo, y ese multiplicador es el precio de la flexibilidad.

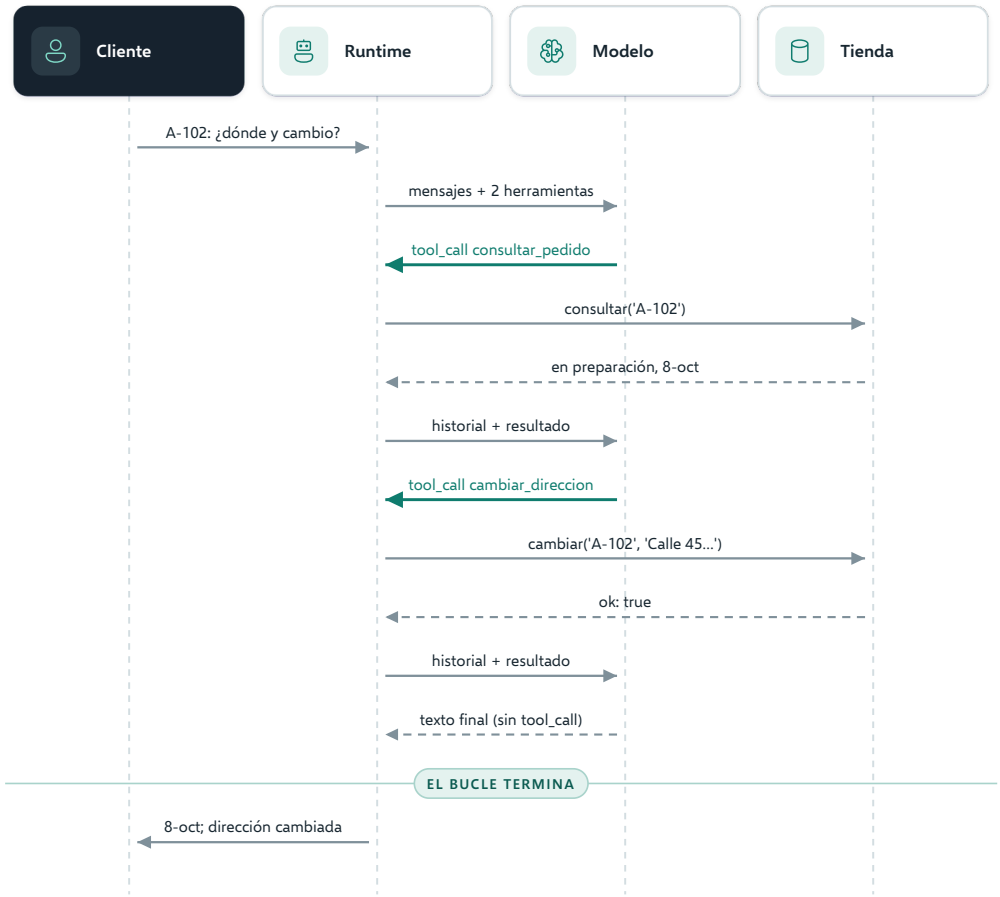


Figura 1.3. Traza de la versión agente para el pedido A-102. Cada flecha verde es una decisión del modelo; cada llamada a la tienda la ejecuta el runtime. El bucle termina cuando el modelo responde sin pedir herramientas.

Qué cambió en las salidas

El script `comparar.py` corre las tres versiones sobre tres casos: el pedido editable, el despachado y un cliente que no da número de pedido. Este es un extracto de la salida guardada en `RESULTADOS.txt`:

PROMPT

```
=== A-215 (ya despachado) ===
[1 prompt] llamadas al modelo=1 herramientas=[]
    → Tu pedido está en camino y debería llegar en 2 o 3 días.
      Sí, puedes cambiar la dirección desde la sección «Mis pedidos».
[2 flujo ] llamadas al modelo=1 herramientas=['consultar', 'cambiar']
    → Tu pedido A-215 ya salió con Envíos Andes (llega el 5 de
      octubre) y por eso no se puede cambiar la dirección.
[3 agente] llamadas al modelo=2 herramientas=['consultar_pedido']
    → Tu pedido A-215 ya fue despachado con Envíos Andes y llega el
      5 de octubre. Como ya salió, no puedo cambiar la dirección;
      ¿quieres que te indique cómo coordinar con la transportadora?

=== Sin número de pedido ===
[1 prompt] llamadas al modelo=1 herramientas=[]
    → Tu pedido está en camino y debería llegar en 2 o 3 días. (…)
[2 flujo ] llamadas al modelo=0 herramientas=[]
    → ¿Me compartes el número de pedido (A-123)?
[3 agente] llamadas al modelo=1 herramientas=[]
    → Claro. ¿Me compartes el número de pedido (formato A-123) y la
      nueva dirección completa?
```

Leída con calma, la tabla de resultados dice cosas más finas que «el agente gana». El prompt único falla en los tres casos con la misma frase, porque no tiene nada distinto que decir. El flujo acierta en los tres y es el más barato: en el caso sin número ni siquiera llama al modelo. El agente también acierta, con un matiz: ante el pedido despachado decide no intentar el cambio después de ver el estado, y ofrece una alternativa que ninguna rama del código contempla. En la ejecución simulada ese comportamiento sale del guion; con un modelo real es lo que buscan las instrucciones, pero nada lo garantiza. Esa iniciativa, con su incertidumbre, es lo que pagas con llamadas extra. En este problema, con dos herramientas y una regla, yo pondría en producción el flujo y guardaría el agente para cuando la conversación real demuestre que las variantes no caben en ramas.

Hay una prueba más en `test_ch01.py` que vale la pena mirar. Simula un modelo «terco» que intenta cambiar la dirección del A-215 sin consultar antes. La herramienta devuelve `ok: false` y la base de datos no cambia. Esa prueba no evalúa al modelo; evalúa tu diseño, y es el tipo de prueba que puedes correr en integración continua sin gastar un centavo en tokens.

CUIDADO

El agente de este capítulo no tiene un límite de llamadas propio (el `recursion_limit` que LangGraph trae por defecto es de miles de pasos), y en producción eso no es aceptable. En LangChain 1.x, `ModelCallLimitMiddleware` y `ToolCallLimitMiddleware` cortan el bucle por número de llamadas; en el capítulo 10 los configuramos junto con reintentos y modelos de respaldo. Pon el límite desde el primer día: un agente que no termina no falla con un error visible, falla con una factura.

Anatomía de un agente

Las tres versiones anteriores difieren en cuánto control tiene el modelo, pero también en cuántas piezas tienen. El prompt único tiene dos: un modelo y unas instrucciones. El agente ya tiene cinco o seis, aunque algunas sean mínimas. Cuando un agente falla en producción, el primer diagnóstico útil es preguntar cuál de esas piezas falló, porque casi nunca es el modelo a secas.

OpenAI resume los componentes centrales en tres: modelo, herramientas e instrucciones. Anthropic parte del «LLM aumentado», un modelo con recuperación, herramientas y memoria. Ambas descripciones son correctas y ambas se quedan cortas para producción, porque dejan fuera lo que controla al agente y lo que permite auditarlo. De ahí sale el modelo de este capítulo.

MODELO PROPIO

Anatomía en siete piezas

Un agente de producción tiene siete piezas y una capa transversal. Cada una corresponde a algo que le darías a una persona nueva en su primer día de trabajo. Si una pieza falta o está mal hecha, el síntoma casi siempre se reporta como «el modelo se equivocó».

1. **Modelo: el perfil.** La capacidad de razonar que contratas. Se elige por tarea, costo, latencia y residencia de datos, y es la pieza que más rápido cambia: diseña para poder reemplazarla (capítulo 3).
2. **Instrucciones: la descripción del cargo.** Rol, objetivo, cuándo usar cada herramienta, qué hacer cuando no sabe y cuándo parar (capítulo 4).
3. **Herramientas: los permisos.** Lo que el agente puede leer y lo que puede cambiar. Cada herramienta es un permiso; las de escritura llevan confirmación (capítulos 3 y 9).
4. **Conocimiento: el manual de la empresa.** Documentos y datos que el agente consulta en vez de inventar, con citas verificables (capítulos 6 y 7).
5. **Memoria: la libreta.** Lo que recuerda dentro de la tarea, entre sesiones y a largo plazo, y lo que debe olvidar (capítulo 8).
6. **Orquestación: el jefe y el proceso.** El bucle, el grafo o el supervisor que decide quién hace qué, con límites de pasos y puntos de aprobación humana (capítulos 10 a 12).
7. **Guardarraíles: el reglamento.** Validaciones de entrada y salida, mínimo privilegio, detección de inyección de prompts y límites de gasto (capítulo 15).

La capa transversal es la **observabilidad**, la hoja de vida verificable: trazas de qué vio el agente, qué decidió, qué ejecutó y cuánto costó cada paso. Sin ella no hay periodo de prueba posible (capítulo 14).



Figura 1.4. Las siete piezas agrupadas por función: quién es el agente, con qué trabaja y quién lo controla. La observabilidad atraviesa todas, porque cada pieza debe dejar rastro.

La figura 1.4 agrupa las piezas en tres columnas, y la agrupación es intencional. Las dos primeras piezas definen *quién* es el agente; las tres del centro, *con qué* trabaja; las dos últimas, *quién lo controla*. En los proyectos que fracasan suele pasar que el equipo invierte casi todo en la primera columna (el modelo más caro, el prompt más largo) y casi nada en la tercera. Es el equivalente a contratar al candidato con la mejor hoja de vida y no darle jefe ni reglamento.

Aplica el modelo a la versión 3. El modelo viene de una variable de entorno. Las instrucciones son cuatro viñetas. Hay dos herramientas, una de lectura y una de escritura. No hay conocimiento documental, porque la pregunta no lo necesita. La memoria es la lista de mensajes de una sola invocación. La orquestación es el bucle de `create_agent`, sin límite de pasos. Y el único guardarraíl es la regla dentro de `cambiar()`. Ya sabes, pieza por pieza, qué le falta para salir a producción: límites, confirmación antes de escribir, memoria entre turnos y trazas.

DESDE LA TRINCHERA

Cuando reviso un agente que «alucina», empiezo por las herramientas, no por el modelo. En la mayoría de los casos la causa es una herramienta que devuelve un error críptico o una lista vacía sin explicación, y el modelo rellena el hueco con lo más probable. Cambiar `None` por «No existe el pedido A-999» suele arreglar más que cambiar de modelo.

> El modelo propone; tu código dispone. Todo control que no esté en el código es una petición, no una garantía.

Por qué ahora: el punto de inflexión de 2025–2026

La idea del agente no es nueva. ReAct, el trabajo de Yao y colegas publicado a finales de 2022, ya proponía que el modelo alternara razonamiento y acciones, y los experimentos de 2023 con agentes autónomos llenaron las redes de demos espectaculares que casi nunca terminaban la tarea. Lo que cambió entre 2024 y 2026 no fue la idea, sino cuatro condiciones técnicas que la volvieron viable para software serio.

Tool calling fiable y salidas estructuradas estrictas. Los proveedores principales ofrecen hoy modos en los que la llamada a herramienta o la respuesta se ajusta a un esquema JSON declarado. OpenAI lo hace con `strict: true` en sus salidas estructuradas y Anthropic tiene la salida estructurada en GA, con herramientas marcadas como estrictas. Para un agente, la diferencia entre «casi siempre JSON válido» y «JSON que cumple el esquema» es la diferencia entre un bucle que se rompe una vez por hora y uno que puedes monitorear. El capítulo 3 explica los requisitos de cada API.

Un protocolo común para herramientas: MCP. Anthropic publicó el Model Context Protocol el 25 de noviembre de 2024 como un estándar abierto para conectar asistentes con los sistemas donde viven los datos, en lugar de un conector distinto por cada combinación de aplicación y fuente. En diciembre de 2025 lo donó a la Agentic AI Foundation, un fondo de la Linux Foundation cofundado con Block y OpenAI, y la especificación vigente al escribir este libro es la del 28 de julio de 2026. Que la herramienta escrita para un cliente funcione en otro cambia la economía de construir agentes. El capítulo 9 lo trata a fondo.

Un protocolo para que los agentes hablen entre sí: A2A. Google anunció Agent2Agent el 9 de abril de 2025 con más de 50 socios tecnológicos y lo presentó como complemento de MCP: MCP conecta un agente con herramientas y contexto; A2A conecta agentes entre sí como pares opacos. La especificación alcanzó su versión 1.0 bajo la Linux Foundation y, en su primer aniversario, más de 150 organizaciones la respaldaban. Microsoft Foundry y Copilot Studio la soportan. Lo vemos en el capítulo 13.

Modelos de razonamiento y ventanas largas. Con o1, en septiembre de 2024, OpenAI popularizó los modelos que dedican cómputo a una cadena de pensamiento antes de responder, entrenados con aprendizaje por refuerzo para descomponer problemas y corregir errores. Hoy todas las familias principales tienen variantes de razonamiento, y las ventanas de contexto de los modelos frontera rondan el millón de tokens. Para un agente, razonar mejor

significa elegir mejor la siguiente herramienta y darse cuenta antes de que va por mal camino; una ventana larga significa poder cargar más historial y más resultados sin perder el hilo, aunque con un costo, como veremos en el capítulo 4 al hablar de ingeniería de contexto.

Hay un indicador que resume el efecto combinado. METR, un laboratorio de evaluación independiente, propuso medir el «horizonte temporal» de un modelo: la duración, medida en tiempo humano, de las tareas que el modelo completa con un 50 % de éxito. En su estudio de 2025 ese horizonte venía duplicándose aproximadamente cada siete meses desde 2019, y Claude 3.7 Sonnet rondaba los 50 minutos en su conjunto de tareas de software. Las tareas de una hora eran, hasta hace poco, territorio exclusivo de humanos.

EN MICROSOFT

Microsoft Agent Framework, GA desde el 3 de abril de 2026 y sucesor de Semantic Kernel y AutoGen, separa en su API las dos mitades del espectro: `Agent` para cuando el modelo decide el siguiente paso, y workflows en grafo (`WorkflowBuilder`) para cuando el proceso está definido de antemano. Su documentación recomienda usar workflows cuando necesitas control explícito del orden de ejecución, y probar antes patrones más simples. Microsoft Foundry Agent Service ofrece la versión gestionada con prompt agents y hosted agents. El capítulo 11 los recorre con código.

Lo que sigue fallando

Sería deshonesto cerrar la sección anterior sin la otra mitad de la historia. Que los agentes sean viables no significa que sean fiables, y los números públicos lo dejan claro.

La fiabilidad cae con los intentos repetidos. τ-bench, publicado por Sierra en 2024, pone a un agente a atender clientes simulados en dominios de comercio y aerolínea, con herramientas y políticas de negocio. Los autores encontraron que incluso agentes con tool calling de primer nivel, como GPT-4o en ese momento, resolvían menos del 50 % de las tareas, y que en el dominio de comercio la métrica pass^8, que exige acertar la misma tarea en ocho intentos, caía por debajo del 25 %. Las cifras de los modelos actuales son otras, pero la lección de la métrica sigue vigente: a un cliente no le importa tu tasa de acierto promedio, le importa que su caso salga bien siempre.

Las tareas sencillas para humanos siguen siendo difíciles. GAIA, de 2023, reunió 466 preguntas conceptualmente simples para una persona pero que exigen buscar, leer y combinar información. Los humanos acertaban el 92 % y GPT-4 con plugins, el 15 %. La distancia se ha reducido mucho desde entonces, y ese es el punto: los benchmarks de agentes envejecen rápido. OpenAI dejó de reportar SWE-bench Verified en febrero de 2026 tras auditar que al

menos el 59,4 % de los problemas revisados tenían pruebas defectuosas. Desconfía de cualquier cifra de benchmark que no venga con su fecha y su metodología, y construye tus propias evaluaciones (capítulo 14).

Muchos fallos son de diseño, no del modelo. El estudio más útil para un arquitecto es «Why Do Multi-Agent LLM Systems Fail?», de un equipo de UC Berkeley. Analizaron más de 1.600 trazas anotadas de siete frameworks multiagente y construyeron MAST, una taxonomía de 14 modos de fallo agrupados en tres categorías: problemas de especificación y diseño del sistema, desalineación entre agentes y verificación deficiente de la tarea. El acuerdo entre anotadores fue alto ($\kappa = 0{,}88$). Según el estudio, la categoría de diseño y especificación es la más grande (cerca del 44 % de los fallos en la versión revisada del artículo), y entre los modos más frecuentes están repetir pasos, desobedecer la especificación de la tarea y no reconocer cuándo hay que parar. Ninguno de esos tres se arregla cambiando de modelo; se arreglan con instrucciones claras, límites, criterios de parada y verificación. Los capítulos 7, 12 y 14 convierten esa lista en controles.

El costo se multiplica. Anthropic reportó en 2025, al describir su sistema multiagente de investigación, que sus agentes consumen del orden de cuatro veces los tokens de una conversación de chat, y los sistemas multiagente unas quince veces. La versión 3 de este capítulo ya mostró la versión doméstica del fenómeno: tres llamadas donde el flujo hacía una. El capítulo 2 enseña a calcular el costo por tarea antes de decidir.

Hay un patrón en estos fallos. Ninguno es exótico. Son los mismos que tendrías con una persona nueva sin supervisión: repite trabajo, se sale del procedimiento, no sabe cuándo dar por terminada una tarea y cuesta más de lo presupuestado. Por eso el libro trata el diseño de agentes como un problema de contratación y gestión, y no solo de elegir el modelo correcto.

> Un agente no falla como un programa, con una excepción que puedes leer. Falla como un empleado: con seguridad, con buena redacción y en silencio.

Cómo leer el resto del libro

Con las definiciones de este capítulo tienes el vocabulario para todo lo que sigue. La Parte I termina de construir los fundamentos: el capítulo 2 decide si de verdad necesitas un agente, el 3 abre el motor (tool calling y salidas estructuradas) y el 4 escribe las instrucciones. La Parte II construye un agente pieza por pieza siguiendo la anatomía de la figura 1.4: el bucle desde cero, conocimiento, defensa contra alucinaciones, memoria y herramientas. La Parte III sube a frameworks, a sistemas con varios agentes y a protocolos, y la Parte IV se ocupa del periodo de prueba y del reglamento: evaluación, observabilidad, seguridad y paso a producción.

Si solo te llevas una idea de estas páginas, que sea la pregunta de la figura 1.2: quién decide el siguiente paso. Cada vez que respondas «el modelo», estás pagando con costo, latencia y riesgo una flexibilidad que tiene que valer la pena. Y cada vez que la respuesta pueda ser «mi código», probablemente debería serlo.

EN LA PRÁCTICA

- [ ] Antes de diseñar, escribe en una frase quién decide el siguiente paso en tu sistema. Si es tu código, construye un flujo.
- [ ] Resuelve primero tu caso con un flujo y un paso de modelo, como en `v2_flujo.py`; pasa a un agente solo cuando tengas ejemplos reales que el flujo no cubra.
- [ ] Pon las reglas de negocio dentro de las herramientas, nunca solo en el prompt, y escribe una prueba con un modelo simulado que intente violarlas.
- [ ] Configura un límite de llamadas al modelo y a herramientas desde el primer prototipo.
- [ ] Haz que cada herramienta devuelva errores legibles («el pedido ya fue despachado»), no excepciones ni valores vacíos.
- [ ] Repasa las siete piezas de la figura 1.4 y anota cuáles tiene tu agente y cuáles faltan antes de mostrarlo a un usuario.
- [ ] Guarda la traza completa de cada ejecución (mensajes, herramientas, argumentos, costo) desde el día uno.
- [ ] Cuando leas una cifra de benchmark, busca su fecha, su metodología y si mide fiabilidad repetida o acierto promedio.

REFERENCIAS

1. Schluntz, E. y Zhang, B. (Anthropic). *Building effective agents*. 2024. https://www.anthropic.com/engineering/building-effective-agents
2. LangChain. *Agents (documentación de LangChain para Python)*. 2026. https://docs.langchain.com/oss/python/langchain/agents
3. OpenAI. *A practical guide to building agents*. 2025. https://cdn.openai.com/business-guides-and-resources/a-practical-guide-to-building-agents.pdf
4. Microsoft. *Microsoft Agent Framework overview: when to use agents vs workflows*. 2026. https://learn.microsoft.com/agent-framework/overview/
5. Microsoft. *Workflows (Agent Framework journey)*. 2026. https://learn.microsoft.com/agent-framework/journey/workflows
6. Yao, S. et al. *ReAct: Synergizing Reasoning and Acting in Language Models*. arXiv:2210.03629. 2022. https://arxiv.org/abs/2210.03629
7. Anthropic. *Introducing the Model Context Protocol*. 2024. https://www.anthropic.com/news/model-context-protocol
8. Model Context Protocol. *MCP joins the Agentic AI Foundation*. 2025. https://blog.modelcontextprotocol.io/posts/2025-12-09-mcp-joins-agentic-ai-foundation/

9. Google. *Announcing the Agent2Agent Protocol (A2A)*. 2025.
https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/

10. Linux Foundation. *A2A protocol surpasses 150 organizations*. 2026.
https://linuxfoundation.org/press/a2a-protocol-surpasses-150-organizations-lands-in-major-cloud-platforms-and-sees-enterprise-production-use-in-first-year

11. OpenAI. *Learning to reason with LLMs*. 2024.
https://openai.com/index/learning-to-reason-with-llms/

12. Kwa, T. et al. (METR). *Measuring AI Ability to Complete Long Software Tasks*. arXiv:2503.14499. 2025.
https://arxiv.org/abs/2503.14499

13. Yao, S., Shinn, N., Razavi, P. y Narasimhan, K. *τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains*. arXiv:2406.12045. 2024.
https://arxiv.org/abs/2406.12045

14. Mialon, G. et al. *GAIA: a benchmark for General AI Assistants*. arXiv:2311.12983. 2023.
https://arxiv.org/abs/2311.12983

15. OpenAI. *Why we no longer evaluate SWE-bench Verified*. 2026.
https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/

16. Cemri, M. et al. *Why Do Multi-Agent LLM Systems Fail?* arXiv:2503.13657. 2025.
https://arxiv.org/abs/2503.13657

17. Anthropic. *How we built our multi-agent research system*. 2025.
https://www.anthropic.com/engineering/multi-agent-research-system

18. Microsoft. *Microsoft Agent Framework version 1.0*. 2026.
https://devblogs.microsoft.com/agent-framework/microsoft-agent-framework-version-1-0/