

Ingeniería de agentes de IA

Diseñar, construir, evaluar y operar
agentes que puedas verificar



Enrique Gutiérrez

Ingeniería de agentes de IA

Diseñar, construir, evaluar
y operar agentes que puedas verificar

Enrique Gutiérrez

Ingeniería de agentes de IA

Diseñar, construir, evaluar y operar agentes que puedas verificar

Traducción de *AI Agents, Engineered: Designing, Building, Evaluating, and Operating Agents You Can Verify* (primera edición, 2026). Traducción del autor.

Copyright © 2026 Enrique Gutiérrez. Todos los derechos reservados.

Ninguna parte de este libro puede reproducirse ni transmitirse en forma alguna sin el permiso previo y por escrito del autor, salvo citas breves en reseñas y trabajos académicos. El material citado sigue siendo propiedad de sus respectivos autores y se cita en las notas.

Los nombres de productos y empresas mencionados en este libro se usan solo como ejemplos y pueden ser marcas registradas de sus propietarios. El autor ha puesto todo su empeño en asegurar la exactitud de la información aquí contenida; no obstante, esta se ofrece sin garantía, y el autor no será responsable de ningún daño derivado de su uso.

Publicado en Leanpub. Primera edición, 2026.

Índice general

Prefacio	ix
I Fundamentos	1
1 ¿Qué es un agente?	2
1.1 La idea de agente	2
1.2 Agentes, flujos de trabajo y chatbots	5
1.3 ¿De verdad necesitas un agente?	8
2 El motor: cómo funcionan los modelos de lenguaje	12
2.1 Los LLM como predictores del siguiente token	12
2.2 Los tokens y la ventana de contexto	16
2.3 Cómo se genera el texto	21
2.4 Fundamentos de la escritura de prompts	25
2.5 Salida estructurada y llamada a funciones	29
2.6 Limitaciones y modos de fallo	32
II Construcción de agentes	38
3 El bucle del agente	39
3.1 El bucle	39
3.2 ReAct: intercalar razonamiento y acción	43
3.3 Un agente mínimo desde cero	47
3.4 Condiciones de parada, presupuestos y frameworks	51
4 Planificación, razonamiento y autocorrección	55
4.1 Descomposición de tareas y planificación	55
4.2 Técnicas de razonamiento	62
4.3 Reflexión y autocorrección	66

5	Las herramientas y el espacio de acciones	73
5.1	Elegir el conjunto y la granularidad de las herramientas . . .	74
5.2	La interfaz de la herramienta	79
5.3	Adaptar programas hechos para personas	82
5.4	La ejecución de código como acción universal	85
5.5	Uso de la computadora y automatización del navegador . . .	89
6	Habilidades, protocolos e interoperabilidad	94
6.1	Qué es una habilidad	95
6.2	Revelación progresiva	99
6.3	Habilidades frente a herramientas frente a protocolos	103
6.4	Protocolos e interoperabilidad	106
6.5	Patrones de diseño de habilidades y composición	110
III	Ingeniería de contexto	116
7	Gestión de la ventana de contexto	117
7.1	Ingeniería de contexto frente a ingeniería de prompts	118
7.2	Un recurso escaso y limitado por la atención	121
7.3	Cómo falla el contexto	124
7.4	Las cuatro operaciones: escribir, seleccionar, comprimir, aislar	130
8	Recuperación y conocimiento	135
8.1	Generación aumentada por recuperación	136
8.2	Más allá de la recuperación ingenua	140
8.3	Recuperación agéntica	144
8.4	RAG frente a contexto largo frente a ajuste fino	147
9	Memoria: el estado de trabajo en ejecuciones largas	151
9.1	La taxonomía de la memoria	152
9.2	Arquitecturas de memoria	155
9.3	Mantener sana una ejecución larga	160
9.4	Aislamiento y cuarentena	164
IV	Patrones y decisiones	168
10	Flujos de trabajo y patrones de composición	169
10.1	El LLM aumentado como unidad básica	170
10.2	Encadenamiento de prompts, enrutamiento y paralelización .	172
10.3	El bucle evaluador–optimizador	179

11 Sistemas multiagente	186
11.1 El patrón orquestador–trabajadores	187
11.2 Un contexto enfocado y autocontenido para cada trabajador .	191
11.3 Modos de fallo de la coordinación	195
11.4 Arquetipos de subagente con nombre	198
11.5 Agentes en paralelo orquestados por una persona	200
11.6 El muro de fusión y el veredicto: un agente frente a varios . .	203
12 Supervisión y autonomía	210
12.1 Puntos de aprobación y escalamiento	211
12.2 Revisar la intención y los resultados, no las líneas	216
12.3 Interrumpibilidad, redirección y el dial de autonomía	221
13 Escribir el bucle externo	226
13.1 Breve historia del bucle	227
13.2 La función objetivo	230
13.3 Ingeniería de bucles	233
13.4 Cuando los bucles se componen	237
13.5 Mantener seguro el bucle	241
14 Elegir el enfoque: cuándo <i>no</i> construir un agente	246
14.1 De vuelta a la escalera	247
14.2 Indicios de que lo necesitas, indicios de que no	252
V Agentes fiables	256
15 Observabilidad y depuración	257
15.1 Trazas y spans	258
15.2 Monitoreo en producción	264
15.3 Reproducir fallos no deterministas	268
15.4 Una taxonomía de los bugs habituales de los agentes	273
16 Evaluación de agentes	279
16.1 Por qué es difícil evaluar	280
16.2 Construir un conjunto de evaluación	284
16.3 El LLM como juez	287
16.4 Desarrollo guiado por evaluaciones y barreras de regresión . .	292
17 Seguridad, protección y salvaguardas	296
17.1 Inyección de prompts	297
17.2 La trifecta letal	301
17.3 Las salvaguardas como capa separada y determinista	305
17.4 Mínimo privilegio, sandbox y aprobación humana	310
17.5 La cadena de suministro	314

18	Fiabilidad, estado y el harness	320
18.1	Devolver los errores al modelo	321
18.2	Reintentos, tiempos límite y presupuestos	324
18.3	Estado y recuperación tras un fallo	326
18.4	Alternativas, degradación y fallo parcial	330
18.5	Ingeniería del harness	332
18.6	Cuánto harness construir	336
VI	Puesta en producción y operación	340
19	Costo, latencia y rendimiento	341
19.1	Adónde van los tokens (y el dinero)	342
19.2	Palancas de costo	346
19.3	Latencia	350
19.4	El triángulo exactitud–costo–latencia	354
20	Despliegue y escalado	358
20.1	Formas de servicio	359
20.2	Ejecución duradera y estado a escala	363
20.3	Ejecución en sandbox a escala	366
20.4	Límites de tasa, colas y contrapresión	368
20.5	La escalera de despliegue	370
VII	Aplicaciones y el camino por delante	375
21	Agentes de programación	376
21.1	Por qué la programación fue la primera	377
21.2	El bucle del agente de programación y el abanico de tareas	381
21.3	Programación aumentada frente a vibe coding	384
21.4	Hacer el código legible para agentes	388
21.5	La lección general	392
22	El flujo de trabajo de programación en la práctica	396
22.1	El bucle canónico: lluvia de ideas, especificación, plan, ejecución	397
22.2	Desarrollo guiado por especificaciones	402
22.3	Desarrollo guiado por pruebas con agentes	405
22.4	Git como red de seguridad	408
22.5	QA y revisión agénticos	412
22.6	Entender código que no escribiste	414

23 Agentes de investigación y de negocio	420
23.1 Agentes de investigación y de información	421
23.2 La brecha de verificación	424
23.3 Agentes de negocio y operativos	429
23.4 Plantear el retorno de la inversión frente al riesgo	434
24 UX de agentes y confianza humana	438
24.1 Calibración de la confianza	439
24.2 Transparencia y retroalimentación sobre el progreso	443
24.3 Capacidad de redirección y fallo elegante	446
24.4 De hacer a dirigir	450
25 Recetas transversales: grandes patrones que atraviesan sectores	454
25.1 El control deslizante de autonomía	455
25.2 Autoresearch	457
25.3 La base de conocimiento personal que se mantiene sola	460
25.4 El analista de investigación profunda	463
25.5 El vigía ambiental	464
25.6 De lo no estructurado a lo estructurado, a escala	466
25.7 El triaje y enrutador de colas	468
25.8 El compañero de trabajo digital	470
25.9 El premortem y el simulador de equipo rojo	471
26 Agentes como clasificadores y puntuadores	476
26.1 El clasificador sin conjunto de entrenamiento	477
26.2 Los ejemplos como libro de texto, no como base de datos	480
26.3 Primero razonar, después explicar	482
26.4 De las etiquetas a las puntuaciones	485
26.5 Desarrollarlo sin engañarte	486
26.6 Empaquetarlo, y cuándo recurrir a él	489
27 La frontera y cómo seguir aprendiendo	494
27.1 Emparejar la generatividad con verificadores deterministas	495
27.2 El ecosistema como capas duraderas	501
27.3 Construir frente a adoptar	505
27.4 Tendencias duraderas, y cómo seguir aprendiendo	508
VIII Apéndices	514
A Un agente mínimo, comentado	515
A.1 El programa completo	516
A.2 El cliente del modelo	518
A.3 El registro de herramientas	519

A.4	Ejecutar una solicitud	520
A.5	El bucle	521
A.6	Las instrucciones permanentes	522
A.7	Una ejecución, de principio a fin	523
A.8	Lo que falta a propósito	525
B	Glosario	526
C	Lecturas adicionales comentadas y mapa de fuentes	541
C.1	Cómo está organizada la biblioteca	542
C.2	El mapa capítulo a capítulo	542
C.3	Referencias primarias duraderas	546
C.4	Un segundo estante: fundamentos que se dan por sabidos y debates en curso	549

Prefacio

Este libro existe por un momento concreto, y sospecho que ya lo has vivido. Le encargas a un modelo de lenguaje una tarea que a una persona le llevaría media hora: averiguar por qué se rompió el build, por ejemplo, o conciliar una hoja de cálculo caótica con la base de datos. En lugar de una respuesta, lo ves *trabajar*. Busca, lee, prueba algo, tropieza con un error, se corrige y vuelve unos minutos después con la tarea terminada. La primera sensación es de fascinación. La segunda llega un instante después y dura más: *No pude ver todos los pasos. ¿Hasta qué punto puedo confiar en esto?*

Esa segunda sensación es el tema de este libro.

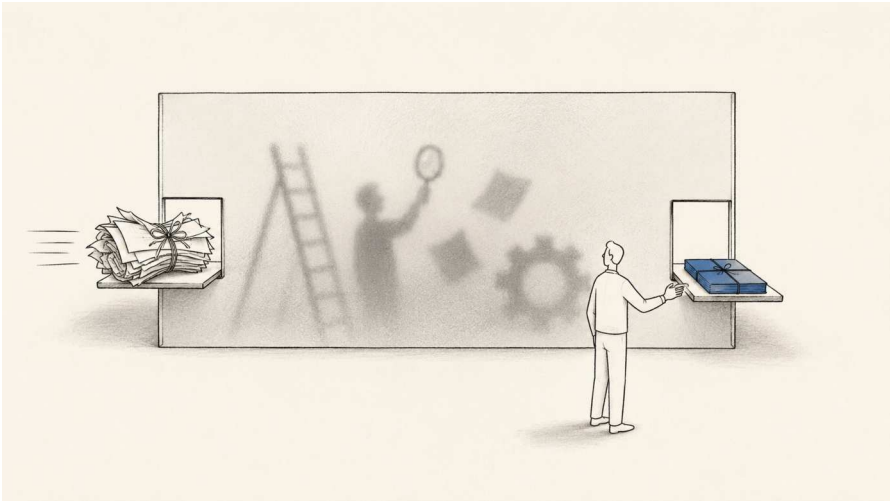


Figura 1: El trabajo desordenado entra por un lado de una pared de vidrio esmerilado y el trabajo terminado sale por el otro; de los pasos intermedios solo se ven siluetas. La fascinación es inmediata. La duda dura más, y es el tema de este libro.

Lo escribí para quien ejerce la ingeniería —alguien que se maneja con soltura con el software, pero no necesariamente con el aprendizaje automá-

tico—, está evaluando agentes o acaba de empezar a construir con ellos, y quiere dejar atrás tanto la euforia como el temor para llegar a la ingeniería que hay debajo. Encontrarás definiciones construidas desde primeros principios, patrones que se repiten en todos los ámbitos donde los agentes hacen un trabajo útil, ejemplos en pseudocódigo sencillo y un interés persistente por lo que cuestan las cosas y por cómo fallan. El único requisito previo son los conocimientos habituales de ingeniería; cada vez que aparece un término técnico, lo defino en el momento, con palabras llanas.

Dos promesas dan forma a cada capítulo, y quiero hacerlas explícitas porque, juntas, explican lo que vas a encontrar aquí y lo que no.

La primera promesa es que el libro es *atemporal*. Este campo avanza a un ritmo que deja en evidencia a cualquiera que ponga algo por escrito; mientras reunía material para estos capítulos, notas que había tomado apenas unos meses antes ya se estaban quedando anticuadas, con los productos rebautizados y las cifras de sus benchmarks superadas. Por eso me he negado a anclar nada a la temporada en que se escribió. Ninguna afirmación de este libro depende de qué modelo es el mejor ahora mismo, de cuánto cuesta un token este trimestre ni del tamaño de las ventanas de contexto de este año. Donde aparece una cifra, está marcada como ilustración. El objetivo es un libro que se lea tan bien dentro de unos años como hoy.

La segunda promesa es que es *agnóstico en cuanto a soluciones*. El conocimiento duradero de este campo está en los conceptos y los patrones —el bucle, el contrato de las herramientas, el presupuesto de contexto, la batería de evaluación—, y esos se trasladan de un proveedor a otro y de un framework a otro. La sintaxis de un producto concreto no se traslada, y envejece en cuestión de meses. Así que este libro enseña el patrón y nombra herramientas concretas solo de pasada, como ejemplos etiquetados de una categoría. Cuando lo termines, deberías poder sentarte ante el *harness* (literalmente, «arnés»: la capa de software dentro de la cual se ejecuta un agente) que haya adoptado tu equipo, o ante ninguno, y saber qué construir.

Hay una idea que se repite tanto en estas páginas que equivale a una tesis, y conviene que la conozcas ya: *un agente solo es tan digno de confianza como la señal con la que puedes verificarlo*. Un modelo de lenguaje produce texto fluido y seguro de sí mismo tanto si acierta como si se equivoca; su tono no te dice nada en lo que puedas apoyarte. Por eso la confianza que depositas en un agente tiene que venir de algún lugar externo al modelo: una batería de pruebas que pasa, un esquema que valida, una fuente que puedes abrir y comprobar, una persona que aprueba el paso irreversible. Donde existe una señal así, puedes conceder autonomía con generosidad; esa es, como veremos, buena parte de la razón por la que la programación fue el primer ámbito en el que los agentes se ganaron el sueldo, porque el código llega con su propio veredicto: compílalo, ejecuta las pruebas. Donde no existe ninguna señal, la autonomía es un acto de fe, y este libro lo dirá una y otra vez. Casi todas las preguntas de diseño de los capítulos que siguen se resuelven con el mismo

primer movimiento: preguntar qué señal te dice que funcionó.

Unas palabras sobre el vocabulario. «Agente» quizá sea el término más sobrecargado del software moderno; se aplica con la misma seguridad a un bot con guion que recita un menú telefónico y a un sistema al que se le confía trabajar solo toda una tarde. La jerga solo es útil cuando quien habla y quien escucha comparten una definición; de lo contrario, dos personas pueden discutir apasionadamente sobre cosas completamente distintas¹. Así que el primer capítulo hace el trabajo poco vistoso de fijar los términos, y el resto del libro se atiene a ellos.

En cuanto a cómo leerlo: el libro se divide en siete partes, ordenadas como un curso. La parte I establece qué es un agente y cómo funciona en realidad el motor que lleva debajo, el modelo de lenguaje. Las partes II y III construyen uno: el bucle, las herramientas, las habilidades reutilizables y la gestión cuidadosa de la memoria de trabajo del modelo. La parte IV trata de arquitectura: flujos de trabajo, sistemas multiagente, el dial entre supervisión y autonomía, el oficio de escribir el bucle externo del harness y la disciplina de preguntarse si lo más simple que funcione necesita siquiera un agente. Las partes V y VI cubren lo que resulta ser la mayor parte de la ingeniería de agentes real: evaluación, observabilidad, seguridad, fiabilidad, costo y despliegue. La parte VII pasa a las aplicaciones y a lo que parece duradero en el camino que queda por delante. Los apéndices contienen un agente mínimo en pseudocódigo comentado, un glosario y un mapa de lecturas adicionales. Lee las partes en orden para hacerte con una base o, cuando ya te hayas orientado, saquéalas como obra de consulta; los capítulos se remiten unos a otros y procuran sostenerse por sí solos.

Debo ser honesto sobre lo que un libro como este puede prometer. Algunos detalles quedarán superados, probablemente antes de lo que me gustaría; he procurado escribir solo las partes que sobreviven a sus ejemplos y señalar mi incertidumbre allí donde la tengo. La apuesta de este libro es que los principios sobreviven a los productos. Si dentro de unos años los ejemplos resultan pintorescos pero las preguntas —*¿qué señal verifica esto?*, *¿qué es lo más simple que funciona?*— siguen siendo las que te haces, el libro habrá cumplido su cometido.

¹Simon Willison, «I think “agent” may finally have a widely enough agreed upon definition to be useful jargon now», simonwillison.net (septiembre de 2025). La observación sobre la jerga compartida y la definición de trabajo que este libro adopta en el capítulo 1 son suyas.

Parte I

Fundamentos

Capítulo 1

¿Qué es un agente?

Este capítulo cumple una promesa hecha en el prefacio: antes de construir nada, fijamos qué *es* un agente. La necesidad es práctica e inmediata. Los proveedores etiquetan como «agentes» a bots con guion; los investigadores reservan la palabra para sistemas que funcionan de forma desatendida durante horas; y en algún punto intermedio tú tienes que tomar decisiones reales con presupuestos reales. Si el término sigue siendo borroso, recurrirás a un agente autónomo donde un pipeline de tres pasos habría sido más barato y más fácil de depurar, o montarás a mano un pipeline rígido para un problema cuyos pasos de verdad no puedes predecir. El capítulo hace tres cosas. Enuncia la idea misma de agente, explica por qué se volvió práctica cuando lo hizo —deliberadamente sin fechas— y presenta la brújula con la que se orientará el libro. Después traza la distinción práctica entre chatbots, flujos de trabajo y agentes en la que se apoyan todos los capítulos posteriores. Y termina con la pregunta que muy poca gente se hace en voz alta: si necesitas siquiera un agente.

1.1 La idea de agente

Quita las demos y el marketing, y la idea es lo bastante pequeña como para caber en una mano. Un *agente* es un modelo de lenguaje colocado en un bucle. Recibe un objetivo y un conjunto de *herramientas*, funciones que puede pedirle a tu programa que ejecute: busca esto, lee aquello, ejecuta este comando. Observa el estado de la tarea, elige una acción, ve el resultado y vuelve a empezar, hasta que juzga que el objetivo se ha cumplido o una regla de parada pone fin a la ejecución. Tras años de diálogo de sordos, la comunidad profesional ha convergido en gran medida, en el momento de escribir estas líneas, en una versión de una sola frase: un agente *ejecuta herramientas en*

un bucle para alcanzar un objetivo¹. En esa frase, cada palabra soporta carga. «Herramientas» es lo que conecta al modelo con el mundo; «bucle» es lo que le permite reaccionar a lo que encuentra; «un objetivo» es lo que pone fin al bucle: un agente tiene una condición de parada, y eso es lo que lo distingue de un proceso desbocado. El capítulo 3 desmenuza el bucle en código.

Lo genuinamente nuevo aquí es dónde vive el flujo de control. En todos los programas que has escrito hasta ahora, la secuencia de operaciones la decidías tú, de antemano, en el código. En un agente, la secuencia la decide el modelo, en tiempo de ejecución, en respuesta a lo que observa. Ese único cambio es lo que hace que los agentes merezcan un libro, y también lo que los hace caros, lentos y, de vez en cuando, alarmantes. El resto del libro trata de las dos mitades de esa frase.

Es razonable que te preguntes por qué esto se volvió práctico cuando lo hizo. Los modelos de lenguaje escribían con fluidez mucho antes de que nadie le confiara un bucle a uno de ellos; la fluidez nunca fue la pieza que faltaba. Tres cosas tenían que madurar. Cada una volverá más adelante en el libro, primero como maquinaria y después como modo de fallo cuando escasea.

La primera: los modelos aprendieron a aceptar instrucciones. Un modelo recién salido de su entrenamiento inicial es un puro continuador de texto: hazle una pregunta y puede que responda con más preguntas, porque esa es una forma plausible de que el texto continúe (el capítulo 2 muestra por qué). Una etapa posterior del entrenamiento, que el campo llama posentrenamiento (*post-training*), convierte a ese continuador en algo a lo que puedes encargarle un trabajo: sigue indicaciones, trabaja hacia el objetivo planteado y se detiene cuando termina. Un agente es una delegación, y delegar exige un trabajador que acepte el encargo.

La segunda: el texto adquirió un puente hacia la acción. La *llamada a funciones* (*function calling*) —el modelo emite una solicitud legible por máquina, «llama a esta función con estos argumentos», que tu código ejecuta antes de devolverle el resultado— dio al modelo un canal disciplinado hacia el mundo. El modelo propone; tu código dispone. El capítulo 2 cubre el mecanismo en detalle. Aquí lo que importa es la consecuencia: un sistema que solo podía escribir *sobre* el mundo pasó a ser uno que puede pedir, de una forma que un programa puede obedecer con seguridad, que en él *ocurran* cosas.

La tercera: la memoria de trabajo del modelo creció. Todo lo que un agente sabe en un momento dado tiene que caber en la *ventana de contexto* del modelo (el tramo acotado de texto que un modelo puede considerar en una sola llamada). Cuando en las ventanas cabían unas pocas páginas, un bucle se moría de hambre: las instrucciones, el historial y el último resultado de una

¹Simon Willison, «I think “agent” may finally have a widely enough agreed upon definition to be useful jargon now», simonwillison.net (septiembre de 2025). La definición citada —«An LLM agent runs tools in a loop to achieve a goal»— es suya, y allí la defiende frase por frase. *LLM* es la abreviatura que usa el campo para *large language model* (gran modelo de lenguaje), que este libro llama casi siempre modelo de lenguaje, o simplemente modelo.

herramienta no cabían juntos. Las ventanas crecieron hasta dar cabida a algo más parecido a una sesión de trabajo, y los bucles se volvieron sostenibles.

Cada una de ellas maduró como una pendiente que cruza un umbral sin hacer ruido, lo cual nos conviene a los dos: la explicación de «por qué ahora» no caduca. También te dice qué comprobar cuando se anuncie alguna capacidad nueva: pregunta cuál de los tres ingredientes acaba de mejorar.

Antes de la taxonomía y la maquinaria, quiero entregarte la brújula con la que se orienta este libro: cuatro rumbos que reaparecerán en casi todos los capítulos. Los enuncio aquí sin su defensa completa; la defensa es el libro.

La *verificabilidad* va primero. El prefacio enunció la tesis y vale la pena repetirla: un agente solo es tan digno de confianza como la señal con la que puedes verificarlo. La seguridad que muestra el propio modelo no aporta ninguna información sobre si tiene razón, así que la confianza debe venir de fuera: una prueba que pasa, un esquema que valida, una fuente que puedes abrir, una persona que aprueba. La primera pregunta que hay que hacerle a cualquier diseño de agente es la que este libro hará una y otra vez: *¿qué señal te dice que funcionó?*

Después viene *la escasez de contexto*. La ventana de contexto es finita, y lo que sorprende a la gente es que la atención del modelo se degrada antes de que se agote el espacio: un dato enterrado en mitad de un montón largo queda, en la práctica, medio olvidado. Trata el contexto como un presupuesto que se gasta de forma deliberada. La parte III de este libro trata de gastarlo bien.

El *error compuesto* es el tercero. Un agente encadena muchos pasos, y la fiabilidad se multiplica a lo largo de ellos. Supón, solo a modo de ilustración, que cada paso sale bien el 95% de las veces; una ejecución de veinte pasos encadenados sale bien entonces alrededor del $0.95^{20} \approx 36\%$ de las veces. Las cifras concretas varían; la aritmética no, y es implacable. Los bucles largos solo sobreviven con recuperación de errores, puntos de control y verificación por el camino. Este libro dedica capítulos enteros a esa maquinaria.

Y el último rumbo es *lo más simple que funcione*. El consejo de ingeniería que mejor ha envejecido en este joven campo es tajante: busca la solución más simple posible y añade complejidad solo cuando esté demostrado que compensa, lo que puede significar no construir un agente en absoluto². A veces lo más simple es un único prompt. A veces es una expresión regular. La última sección de este capítulo se toma este rumbo lo bastante en serio como para dedicarle una discusión propia. La figura 1.1 coloca los cuatro rumbos sobre una misma rosa de los vientos, con la aguja apoyada en el primero.

Una brújula es un instrumento modesto; no va a diseñar tu sistema por ti.

²Erik Schluntz y Barry Zhang, «Building Effective Agents», blog de ingeniería de Anthropic (2024): «recomendamos buscar la solución más simple posible y aumentar la complejidad solo cuando haga falta. Eso puede significar no construir sistemas agénticos en absoluto». La distinción entre flujo de trabajo y agente que hace ese ensayo vertebraba también la sección siguiente.

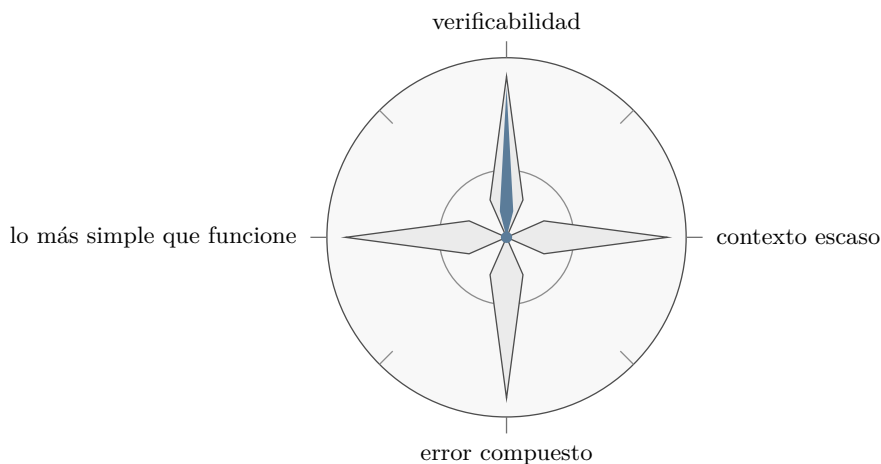


Figura 1.1: La brújula del libro. Cuatro rumbos deciden las bifurcaciones de diseño en casi todos los capítulos posteriores, y la aguja se apoya en el primero de ellos, la verificabilidad, porque un agente solo es tan digno de confianza como la señal con la que puedes comprobarlo. Los otros tres —la escasez de contexto, el error compuesto y lo más simple que funcione— se leen junto a él, no en su lugar.

Pero cada vez que el libro llegue a una bifurcación —más autonomía o menos, otro agente o un simple script, un contexto más rico o más austero— verás que son los mismos cuatro rumbos los que la deciden, y espero que al final estén decidiendo también tus bifurcaciones.

1.2 Agentes, flujos de trabajo y chatbots

Déjame ponerte delante tres sistemas. Decide cuál de ellos es el agente antes de seguir leyendo.

El primero es un asistente de preguntas y respuestas. Pegas una función y preguntas qué hace, y te lo explica; pegas un diff y pides un borrador de mensaje de commit, y lo escribe; después espera tu siguiente mensaje. El segundo procesa de la misma manera cada ticket de soporte que entra: una llamada al modelo clasifica el ticket, código ordinario busca el documento de política correspondiente, una segunda llamada al modelo redacta una respuesta, una tercera revisa el borrador con una rúbrica, y el resultado va a parar a una cola de revisión humana; cuatro pasos, en ese orden, siempre. Al tercero se le dice «las pruebas del checkout están fallando: arréglalas» y, sin supervisión, busca en la base de código, lee tres archivos, edita uno, ejecuta las pruebas, lee el nuevo fallo, vuelve a editar y se detiene cuando toda la batería está en

verde.

El instinto de casi todo el mundo los clasifica al instante: el tercero parece de otra naturaleza, y el instinto acierta. Lo útil es decir con precisión *por qué*. Los tres pueden estar contruidos exactamente sobre el mismo modelo. Lo que cambia es la pregunta que la sección anterior señaló como lo genuinamente nuevo: quién decide qué pasa a continuación. En el primer sistema decides tú, mensaje a mensaje. En el segundo, tu código; el modelo solo rellena contenido a lo largo de una ruta que quedó fijada antes de que llegara el primer ticket. En el tercero decide el *modelo*: qué archivo leer, si editar o volver a ejecutar, cuándo parar. Esa pregunta —quién es el dueño del flujo de control— es toda la distinción, y nos da los tres términos de trabajo que este libro usa en todas partes. La figura 1.2 dibuja los tres uno junto a otro; solo se diferencian en quién decide el siguiente paso.

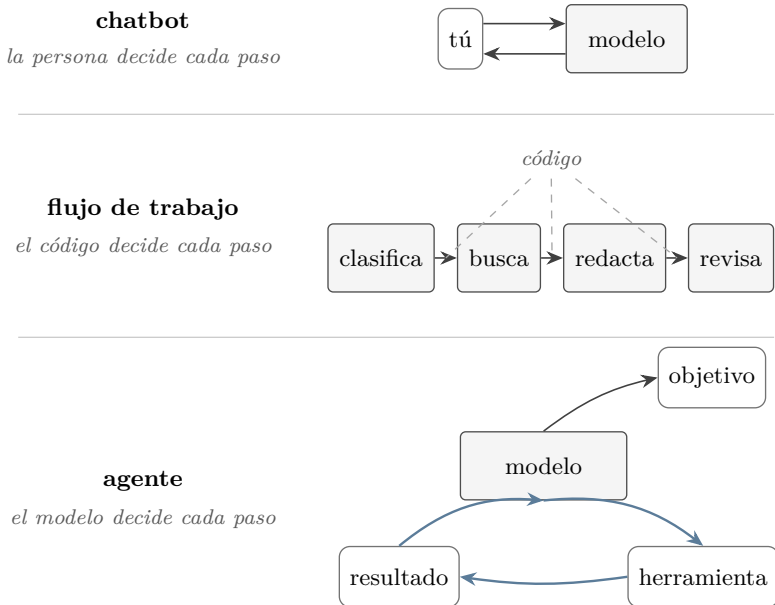


Figura 1.2: Chatbot, flujo de trabajo y agente se diferencian en una sola cosa: quién decide el siguiente paso. En un chatbot la persona dirige cada turno; en un flujo de trabajo el camino está fijado en el código y el modelo se limita a rellenar cada paso; en un agente el modelo es dueño del bucle, y elige herramientas y lee resultados hasta alcanzar el objetivo. El bucle en manos del modelo (en el color de acento) es todo el tema de este libro.

Un *chatbot* es una conversación turno a turno. La persona dirige cada paso; el modelo genera texto y espera. No hay bucle y no se ejecuta nada. Los chatbots son útiles de verdad y a menudo son la herramienta adecuada,

y su superficie de fallo es pequeña de un modo que quiero dejar por escrito: lo peor que puede producir un chatbot es una respuesta equivocada, porque actuar sobre el mundo queda fuera de su alcance.

Un *flujo de trabajo* (*workflow*) son varias llamadas al modelo (y llamadas a herramientas) conectadas a lo largo de rutas que tu código define de antemano. El modelo rellena el contenido en cada paso —clasifica esto, redacta aquello—, pero es tu código el que decide qué paso viene después. Las formas recurrentes tienen nombres que volverás a encontrar en el capítulo 10: *encadenamiento* (la salida de una llamada alimenta la siguiente), *enrutamiento* (clasificar y después bifurcar) y *paralelización* (lanzar varias llamadas a la vez y después agregar sus resultados). El pipeline de tickets de antes es un flujo de trabajo, y podrías haber dibujado su diagrama de flujo antes de ponerlo en marcha.

Un *agente* le entrega al modelo el flujo de control. Un ensayo de ingeniería muy citado traza la línea exactamente aquí: los flujos de trabajo son «sistemas en los que los LLM y las herramientas se orquestan mediante rutas de código predefinidas», mientras que los agentes son sistemas en los que los modelos «dirigen dinámicamente sus propios procesos y su uso de herramientas»³. El número de pasos, y qué paso viene en qué momento, no figura en ninguna parte de tu código; el modelo va eligiendo sobre la marcha, reaccionando a lo que acaba de observar. Por eso el sistema que arregla las pruebas parece otra cosa: su transcripción se lee como el registro de la tarde de un colega, y solo de lejos como la ejecución de un programa.

Debajo de los dos últimos está el mismo átomo, que merece un nombre porque con él se ensamblan los patrones que encontrarás en el capítulo 10. El *LLM aumentado* (*augmented LLM*) es una llamada al modelo reforzada con tres cosas: *recuperación* (traer información relevante en el momento de la solicitud), *herramientas* (funciones que puede solicitar) y *memoria* (alguna forma de llevar consigo lo que importa). Un flujo de trabajo son varios de estos átomos conectados por tu código a lo largo de una ruta fija. Un agente es uno de estos átomos colocado dentro de un bucle, donde decide por sí mismo qué hacer en cada vuelta. (Un chatbot, en esta imagen, es la llamada al modelo a secas, con las ampliaciones casi sin usar.) El mismo átomo, dos disposiciones: quédate con esa imagen y las taxonomías dejan de parecer misteriosas.

Esta es la prueba decisiva que usaré durante el resto del libro. Si puedes dibujar con aplomo el diagrama del flujo de control antes de que llegue la solicitud, tienes delante un flujo de trabajo. Si el diagrama solo puede dibujarse a posteriori, una vez que el modelo ha reaccionado a lo que encontró, tienes delante un agente.

Es tentador tratar chatbot, flujo de trabajo y agente como tres cajas. En

³Erik Schluntz y Barry Zhang, «Building Effective Agents», blog de ingeniería de Anthropic (2024); ambas frases están citadas del ensayo, que también da nombre al bloque básico del *LLM aumentado* descrito en el párrafo siguiente. Un estudio panorámico anterior —Lilian Weng, «LLM Powered Autonomous Agents», lilianweng.github.io (2023)— describe la misma anatomía desde dentro: un modelo que hace de «cerebro», apoyado en la planificación, la memoria y el uso de herramientas.

realidad es un dial (figura 1.3). En un extremo, el modelo genera texto y una persona lo decide todo. Cerca de ese mismo extremo está el pipeline fijo (el sistema de tickets de hace un momento), en el que el código es dueño de cada paso y el modelo solo rellena contenido. Un pequeño paso más allá, un flujo de trabajo le concede al modelo una decisión: un enrutador elige qué rama fijada en el código atiende la solicitud. Más adelante, un agente itera libremente sobre sus herramientas pero se detiene a pedir aprobación humana antes de cualquier cosa con consecuencias —enviar el correo, gastar el dinero, borrar los datos—; buena parte de lo que hay en producción está aquí. En el extremo opuesto, el modelo es dueño del camino de principio a fin durante muchos pasos, sin confirmación en cada uno: máxima capacidad, máximo *radio de impacto* (*blast radius*) —el daño que una acción equivocada podría causar antes de que algo la detenga—, un término que el capítulo 17 precisará. Lo que se mueve a lo largo de este dial es una sola magnitud, cuánta parte del flujo de control está en manos del modelo, y los sistemas maduros eligen una posición deliberadamente en lugar de heredarla de una demo.

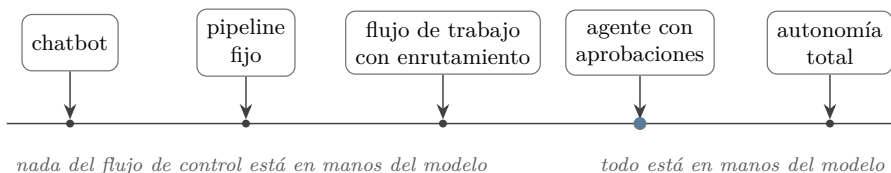


Figura 1.3: La autonomía es un dial, no un interruptor. La magnitud que varía a lo largo de él es cuánta parte del flujo de control está en manos del modelo. Las etiquetas son paradas a las que vale la pena poner nombre; los sistemas reales suelen quedar entre ellas.

Una observación más antes de dejar la taxonomía, porque anticipa adónde llega este libro. Casi nada de lo que sale a producción es un flujo de trabajo puro o un agente puro. La forma dominante en producción es una carcasa de flujo de trabajo con uno o dos pasos genuinamente agénticos dentro: estructura predecible allí donde puedes predecir, y autonomía en manos del modelo solo en los puntos que no puedes enumerar. Estás ajustando un dial, paso a paso, y la siguiente sección trata de cuánto girarlo.

1.3 ¿De verdad necesitas un agente?

Ahora que sabes distinguir un agente de un flujo de trabajo, te enfrentas a la pregunta que esta sección existe para hacer respetable: ¿necesitas uno siquiera? La planteo tan pronto a propósito. El error contra el que este campo más previene a los recién llegados es recurrir a un agente autónomo cuando un único prompt, un pipeline fijo o cincuenta líneas de código ordinario habrían

sido más baratos, más rápidos y más fáciles de depurar. Una demo de un agente es algo seductor. Ves a un modelo planificar, actuar, recuperarse y terminar, y la conclusión natural es que ese poder tiene sitio en todas partes. Resiste la conclusión el tiempo suficiente para ponerle precio.

Frente a una sola llamada al modelo, un agente añade varios costos recurrentes, y llegan en paquete. Primero, el dinero: un agente hace muchas llamadas al modelo por tarea en lugar de una, y cada llamada reenvía la conversación, que no deja de crecer, de modo que el costo escala por partida doble: una vez por el número de llamadas y otra por el tamaño de cada una. Un bucle de diez pasos puede costar un orden de magnitud más que la única llamada a la que sustituyó (el multiplicador es ilustrativo; lo que importa es la dirección). Segundo, la latencia: las llamadas son secuenciales, cada una a la espera del último resultado de una herramienta, así que una respuesta de menos de un segundo se convierte en muchos segundos o en minutos; si hay una persona esperando al otro lado, esto por sí solo puede descartar el diseño. Tercero, el no determinismo: la misma entrada puede tomar un camino distinto y producir una respuesta distinta en cada ejecución, lo que rompe sin hacer ruido tus hábitos de prueba, depuración y auditoría⁴. Y la superficie de fallo cambia de naturaleza, que es el costo que más quiero que sientas: el fallo acotado que la sección anterior le reconoció al chatbot es exactamente aquello a lo que un agente renuncia. Su error es una *acción* equivocada, seguida de diez acciones más construidas sobre ella, y el error se va componiendo bucle abajo exactamente como predice la aritmética de la brújula.

Hay otros dos costos que es fácil pasar por alto cuando la demo deslumbra. Dar herramientas y autonomía a un modelo amplía tu superficie de ataque; la comunidad de seguridad tiene un nombre para cuando se le da de más: *agencia excesiva* (*excessive agency*), es decir, más permisos, funcionalidad o libertad de los que la tarea requiere⁵. El peligro, además, va más allá de lo que concedes deliberadamente: el texto que un agente simplemente *lee* puede dirigir lo que hace a continuación, una familia de ataques llamada *inyección de prompts* (*prompt injection*) que el capítulo 17 examina. Por último, la carga operativa: un chatbot necesita un prompt, mientras que un agente necesita además el cableado de las herramientas, gestión del contexto, recuperación de errores, condiciones de parada, puntos de aprobación, trazas y evaluaciones. Esa maquinaria que lo rodea se llama *harness*, y construirla es la mayor parte del trabajo; un modelo brillante en un harness endeble falla en producción.

Así que la opción honesta por defecto es una escalera, que es el cuarto rumbo de la brújula llevado a la práctica, y la disciplina consiste en quedarte en el peldaño más bajo que resuelva tu problema. El peldaño inferior es

⁴Un análisis publicado sobre la fiabilidad de los agentes recoge tasas de éxito en una sola ejecución de alrededor del 60% que caen a aproximadamente el 25% cuando la misma tarea debe salir bien en ocho ejecuciones consecutivas; son cifras que ilustran la brecha de consistencia, no hechos permanentes. Fiddler AI, «AI Agent Failure Rate: Why 70–95% Fail in Production», blog de fiddler.ai (2026).

⁵OWASP, «Top 10 for LLM Applications»: la entrada sobre Excessive Agency.

código corriente —una expresión regular, una consulta SQL, una sentencia `if`—, gratis, instantáneo, determinista y auditable; si la lógica puede especificarse por completo, no hace falta ningún modelo. El siguiente peldaño es una única llamada al modelo bien trabajada, quizá con recuperación y unos cuantos ejemplos en el prompt; una fracción sorprendente de los «necesitamos un agente» se disuelve en «necesitábamos un prompt mejor». El peldaño de encima es un flujo de trabajo: varias llamadas por rutas que dibujaste de antemano. El peldaño más alto es un agente. Cada peldaño que subes compra adaptabilidad y la paga en costo, latencia e imprevisibilidad. Para un problema cuyos pasos puedes dibujar de antemano, un peldaño inferior es la respuesta correcta de forma permanente; quedarse ahí es la disciplina. La figura 1.4 dibuja la escalera: la adaptabilidad aumenta a medida que subes, y el costo aumenta con ella.

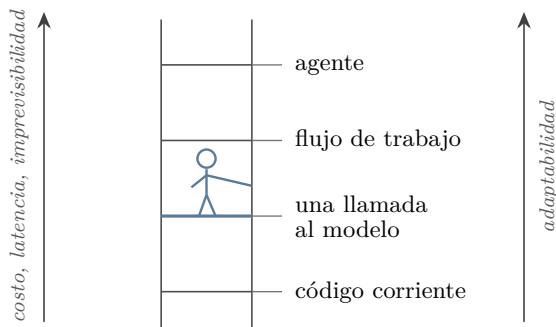


Figura 1.4: La escalera de complejidad. Cada peldaño que subes —de código corriente a una única llamada al modelo, a un flujo de trabajo, a un agente— compra adaptabilidad y la paga en costo, latencia e imprevisibilidad. La disciplina consiste en quedarte en el peldaño más bajo que resuelva el problema (en el color de acento) y subir solo cuando un caso real demuestre que el peldaño de abajo no alcanza.

La prueba práctica es la de la sección anterior, usada ahora como ayuda para decidir: ¿puedes dibujar el diagrama de flujo antes de que llegue la solicitud? Si es así, detente en un flujo de trabajo o más abajo: los pasos conocidos, los requisitos de corrección auditables, los presupuestos ajustados de latencia o de costo y las tareas sencillas de consultar, clasificar y transformar apuntan todos escalera abajo. Lo que apunta escalera arriba es el objetivo genuinamente abierto: los pasos dependen de lo que el modelo descubra, y hace falta criterio en bifurcaciones que no puedes enumerar. Incluso entonces, exige el primer rumbo de la brújula en forma concreta: un criterio de éxito claro, una señal de retroalimentación en cada paso y un lugar sensato donde una persona pueda mirar. Un par concreto para calibrar: «enruta cada correo entrante a facturación, soporte técnico o ventas y redacta una primera respuesta» tiene

pasos que se pueden conocer y un volumen alto, y pide un flujo de trabajo; «haz que pase esta prueba que falla en una base de código que nunca has visto» tiene pasos imposibles de conocer y un verificador incorporado, y es un auténtico problema de agente.

Una trampa merece párrafo propio, porque defiende con mucha elocuencia el diseño equivocado: «quizá algún día tenga que adaptarse». Eso es una hipótesis, y sale barato ponerla a prueba con honestidad: construye primero la versión más simple, mídela con casos reales y sube por la escalera solo cuando puedas señalar una clase de entradas en la que el sistema simple falla de forma demostrable y confirmar que un agente de verdad lo hace mejor. La trampa inversa es el *agent-washing*: como la palabra vende, pipelines con guion se reetiquetan como agentes. La pregunta que desactiva ambas es la prueba decisiva que ya tienes, aplicada al sistema que tienes delante, diga lo que diga su etiqueta: ¿decide el modelo el siguiente paso, o lo decide el código? El capítulo 14 vuelve a esta decisión con la ventaja de todo lo que hay en medio; el capítulo 16 muestra cómo hacer la medición.

Un agente, entonces, es un costo que pagas por una adaptabilidad a la que puedes poner nombre. Si no puedes ponérselo, guárdate el dinero. Y si puedes —si de verdad los pasos no se pueden dibujar de antemano y existe una señal que lo verifique—, entonces un agente es la herramienta adecuada, y los próximos capítulos tratan de construir uno bien. Antes, sin embargo, necesitas entender el motor que vamos a poner a dar vueltas: el capítulo 2 trata de lo que es en realidad un modelo de lenguaje.

Capítulo 2

El motor: cómo funcionan los modelos de lenguaje

Todos los sistemas de este libro, desde el chatbot más sencillo hasta una flota de agentes que cooperan, están contruidos alrededor del mismo componente: un gran modelo de lenguaje. Puedes usar uno con provecho sin saber nada de sus entrañas, igual que puedes conducir un automóvil sin abrir jamás el capó. Construir un agente es una relación distinta con la máquina. Vas a meter este componente en un bucle, entregarle herramientas y dejar que tome decisiones mientras tú estás en otra parte; cuando se comporte de forma extraña, «es magia» no te ayudará a acotar el bug. Este capítulo construye el modelo mental mínimo que necesita quien construye agentes, con la intuición por delante y sin más matemáticas que la aritmética. Empezamos por lo que el modelo es en el fondo (un predictor del siguiente token); después vemos su memoria de trabajo (los tokens y la ventana de contexto), su manera de escribir (el muestreo, y por qué dos ejecuciones difieren), la manera de dirigirlo (la escritura de prompts), el puente que lleva de su texto a la acción real (la salida estructurada y la llamada a funciones) y, por último, las formas en que falla con regularidad. Los capítulos posteriores se apoyan en cada una de estas ideas; ninguna exige que entrenes nada.

2.1 Los LLM como predictores del siguiente token

Termina esta frase: «La capital de Francia es —». La palabra estaba en tu mente antes de que decidieras buscarla. No consultaste un atlas mental, ni sopesaste ciudades candidatas, ni razonaste sobre geografía; toda una vida de lecturas hizo que *París* simplemente llegara. Conserva ese reflejo, porque es la intuición más exacta que hay sobre lo que hace un gran modelo de lenguaje.

El modelo completa texto como acabas de hacerlo tú; a diferencia de ti, no tiene otro modo.

Mecánicamente, un LLM es una función: recibe un tramo de texto y produce, para cada posible *token* siguiente, la probabilidad de que sea el que viene a continuación. Un token es un trozo de texto de un vocabulario fijo —a menudo una palabra, a veces un pedazo de una o un signo de puntuación—; en promedio, piensa en tres cuartos de una palabra inglesa. Para escribir cualquier cosa, el sistema que lo rodea elige un token de esa lista ordenada, lo añade al texto y vuelve a ejecutar el modelo sobre el texto, ahora más largo; así, token a token, una y otra vez, hasta que salta una condición de parada. El procedimiento se llama *generación autorregresiva* (cada token nuevo está condicionado por todo lo anterior, prompt y salida por igual), y una de sus propiedades nos acompañará durante todo el libro: el modelo nunca redacta su respuesta por adelantado. Todo su método es una conjetura local, repetida; no guarda en reserva ningún plan ni ningún esbozo (figura 2.1). La siguiente sección examina a fondo los tokens y la memoria de trabajo del modelo; la que viene después mira cómo se elige un token y por qué la elección varía.

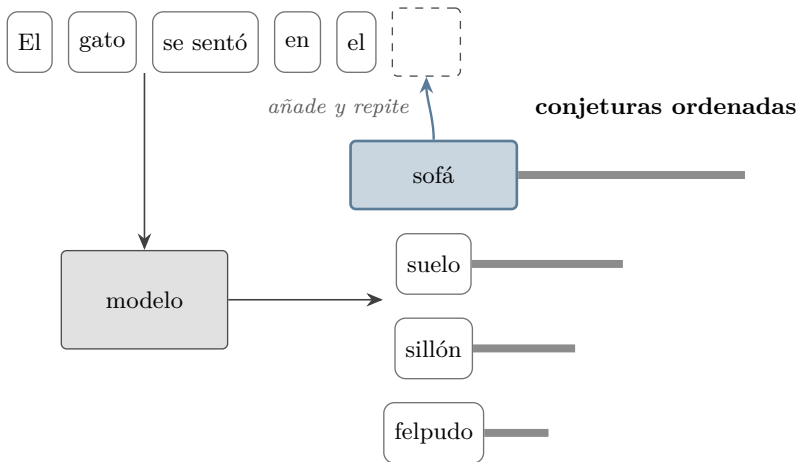


Figura 2.1: La generación autorregresiva, todo el método del motor. El texto que hay hasta el momento entra en el modelo, que devuelve una lista ordenada de conjeturas para el siguiente token; se elige una, se añade al texto, y el modelo vuelve a ejecutarse sobre la entrada, ahora más larga. El token elegido y su camino de vuelta están en el color de acento: no hay ningún plan guardado en reserva, solo esta conjetura local repetida.

¿Dónde vive esa capacidad de conjeturar? En los *parámetros* del modelo, también llamados *pesos*: miles de millones de números, fijados durante el entrenamiento, que determinan cómo fluye el texto de entrada por la red neuronal hasta producir esas probabilidades. La imagen que me resulta más

útil es esta: los pesos son un destilado comprimido y con pérdida de los patrones de todo lo que el modelo leyó durante el entrenamiento: gramática, hechos, modismos, estilo de código, la forma de los argumentos. «Con pérdida» es la expresión clave. El modelo absorbió la forma de sus lecturas y no conservó ningún archivo del texto en sí, y por eso pedirle que cite una fuente al pie de la letra, o que recuerde una cifra exacta, es una apuesta; volveremos a las consecuencias al final de este capítulo.

Hay dos actividades que conviene mantener bien separadas, porque confundirlas produce algunos de los malentendidos más comunes sobre estos sistemas. El *entrenamiento* es el proceso, único y enormemente caro, de encontrar buenos valores para los pesos: al modelo se le muestra texto, se le pide que prediga el siguiente token y se le corrige ligeramente cada vez que se equivoca, a lo largo de más texto del que una persona podría leer en mil vidas. Lo llevan a cabo los creadores del modelo, en hardware especializado, antes de que tú llegues a verlo. La *inferencia* es todo uso posterior: los pesos están congelados, y el modelo se limita a pasar tu entrada a través de ellos. Todo lo que haces con un modelo —cada chat, cada paso de un agente— es inferencia. De ahí se siguen dos consecuencias, y diseñarás en torno a ambas constantemente. La primera: el modelo no aprende de tus conversaciones; nada de lo que tecleas cambia un solo peso, y lo que parece memoria dentro de una sesión es solo la transcripción, cada vez más larga, que se relee en cada llamada. La segunda: el conocimiento incorporado del modelo se detiene en una *fecha de corte del entrenamiento* (el punto a partir del cual no vio más datos). Los acontecimientos posteriores al corte sencillamente no están, y no puedes contar con que el modelo avise del hueco; responde desde un conocimiento caducado exactamente con la misma seguridad con la que responde desde el reciente. Todo lo que dependa de la actualidad tiene que llegar en el prompt, o traerlo una herramienta.

El propio entrenamiento ocurre por etapas, y las etapas explican un enigma con el que, de otro modo, tropezarías. La primera etapa, y la más costosa, el *preentrenamiento*, es pura predicción del siguiente token sobre un corpus vasto y variado: páginas web, libros, código. Su producto es un *modelo base*: fluido, con amplios conocimientos y extrañamente inútil como asistente. El capítulo 1 mencionó, de pasada, que un modelo así puede responder a una pregunta con más preguntas, y prometió que este capítulo mostraría por qué. Ya tienes todas las piezas de la explicación; ensámblala antes de seguir leyendo. Un modelo base solo continúa texto, así que cuando le entregas «¿Cuáles son tres formas de hacer más rápida esta consulta?», busca una continuación plausible, y en el texto del mundo real que estudió, a una pregunta la siguen a menudo otras parecidas. Una lista de más preguntas es una respuesta perfectamente probable. El modelo base está haciendo su trabajo a la perfección; el trabajo es la continuación.

El *posentrenamiento* (*post-training*) es la etapa posterior que convierte a ese continuador en bruto en el asistente que de hecho conoces, y es a lo que

se refería el «los modelos aprendieron a aceptar instrucciones» del capítulo 1. Dos movimientos importan para lo que nos ocupa. El *ajuste por instrucciones* (*instruction tuning*) entrena al modelo con pares curados de instrucción y buena respuesta, y le enseña que una pregunta merece una respuesta y no más preguntas. El *ajuste por preferencias* (*preference tuning*) —la forma más conocida se llama *RLHF*, aprendizaje por refuerzo con retroalimentación humana— hace que unas personas (o unos modelos entrenados para imitar su criterio) ordenen respuestas rivales, y ajusta los pesos hacia el tipo de respuestas que la gente prefiere: más útiles, más honestas, de mejores modales. Del posentrenamiento sale el conocido personaje de asistente, y es una mano de pintura comparativamente barata sobre el enorme núcleo preentrenado.

Ten igual de claro lo que el posentrenamiento no compra. No añade conocimiento nuevo, y no elimina la tendencia a inventar. Una formulación memorable, tomada de una conferencia muy vista sobre el tema: el posentrenamiento no impide que el modelo sueñe; da a los sueños la forma de sueños de asistente servicial¹. Bajo los buenos modales, el motor no ha cambiado: un predictor del siguiente token, que optimiza la continuación plausible. Lo plausible y lo verdadero están correlacionados. La correlación existe porque el texto de entrenamiento lo escribieron sobre todo personas que describían un mundo real; pero los dos se separan exactamente allí donde al modelo le falta el dato, y en ese momento la maquinaria produce la cadena de aspecto más plausible en lugar de detenerse. Primero fluido; correcto, después.

Para quien construye agentes, el corolario que más quiero que conserves es que el conocimiento del modelo vive en dos lugares de fiabilidad desigual. El conocimiento *paramétrico* está en los pesos: congelado en la fecha de corte, aproximado, imposible de citar. El conocimiento *de contexto* es el texto que el modelo tiene delante ahora mismo: reciente, inspeccionable, comprobable. El modelo es notablemente más fiable cuando los datos que necesita están en su contexto que cuando tiene que rescatarlos de sus pesos, y de esa asimetría nace la mitad de la ingeniería de este libro. La recuperación, las herramientas y toda la disciplina de la ingeniería de contexto de la parte III son, en el fondo, maneras de sacar del primer lugar los datos que soportan carga y llevarlos al segundo.

Así que este es el motor: soberbio en el trabajo con forma de lenguaje —redactar, reescribir, resumir, traducir, explicar, transformar código— y poco fiable como base de datos o como calculadora, con un dial de seguridad en sí mismo atascado para siempre en el máximo. Un agente le pedirá a este motor que tome decisiones, muchas seguidas, sobre un mundo que solo puede ver por el ojo de la cerradura de su contexto. Hacia ese ojo de la cerradura vamos ahora.

¹La imagen del «sueño» es de Andrej Karpathy, de sus charlas y comentarios, muy difundidos, sobre los grandes modelos de lenguaje (p. ej., «Intro to Large Language Models»); aquí se ofrece como paráfrasis, no como cita literal.

2.2 Los tokens y la ventana de contexto

Todo lo que el modelo sabe en un momento dado llega por ese ojo de la cerradura, así que más nos vale medirlo. Al capítulo 1 le bastó con poco más que una definición de la *ventana de contexto*: el tramo acotado de texto que un modelo puede considerar en una sola llamada. Esta sección completa la imagen, y hay que empezar por la unidad en la que se mide la ventana, porque casi todos los límites con los que chocarás al construir —el techo de lo que el modelo puede ver, la factura a fin de mes, la pausa antes de que empiece una respuesta— se expresan en tokens. La sección anterior ofreció una versión de servilleta: un token es un trozo de texto, en promedio unos tres cuartos de una palabra inglesa². La versión de servilleta esconde un problema de diseño real, y la solución a ese problema explica varios comportamientos que de otro modo parecerían pura estupidez.

El problema de diseño es este. Un modelo necesita un vocabulario fijo: un menú finito de trozos con los que se ensamblará toda entrada y toda salida posibles. Supón que el menú fueran palabras enteras. El vocabulario se dispara —love, loves, loved, lovingly necesitan cada una su propia entrada— y la primera errata, el primer nombre inventado o la primera palabra acuñada después del entrenamiento no tienen entrada alguna. Supón, en cambio, que el menú fueran caracteres sueltos. Ahora nada es irrepresentable y el vocabulario es diminuto, pero cada texto se convierte en una secuencia larguísima de unidades casi sin significado. Los modelos en producción se quedan con el término medio entre ambos, llamado *tokenización en subpalabras*: un vocabulario fijo de trozos, mucho mayor que un alfabeto y mucho menor que un diccionario con todas las formas de cada palabra, en el que las palabras comunes viajan como un solo token mientras que las más raras se ensamblan a partir de piezas, de modo que **annoyingly** podría convertirse en **annoying** + **ly**, y **unhappiness** en **un** + **happi** + **ness**³. Las cadenas frecuentes se representan de forma barata; cualquier cadena, sea la que sea, puede representarse recurriendo a piezas más pequeñas.

La máquina que hace el troceado, el *tokenizador*, merece un párrafo breve, porque es mucho más simple de lo que la gente supone. Es una tabla de consulta fija y determinista, construida una sola vez junto con el modelo y congelada durante toda la vida de este. A la entrada, convierte tu texto en una lista de identificadores enteros; a la salida, convierte de nuevo en texto los

²La regla de los tres cuartos de palabra (o, lo que es lo mismo, unos cuatro caracteres por token en inglés) procede de la nota del centro de ayuda de OpenAI «What are tokens and how to count them?»; es un promedio aproximado para la prosa en inglés, no una constante.

³La documentación de tokenizadores de Hugging Face («Summary of the tokenizers») describe los algoritmos estándar de subpalabras —BPE, WordPiece, Unigram— y los ilustra con divisiones como estas. Para verlo a fondo y de forma práctica, la clase de Andrej Karpathy «Let's Build the GPT Tokenizer» construye uno desde cero y recorre los comportamientos extraños que provoca.

identificadores que el modelo eligió. Fíjate en lo que esto implica: la red nunca ve tus caracteres. Desde donde está el modelo, el mundo es una secuencia de trozos numerados, y lo que ocurra *dentro* de un trozo se borró antes de que el modelo llegara a intervenir.

Ese borrado se puede poner a prueba, y te recomiendo que hagas la prueba por tu cuenta. Pregúntale a un modelo cuántas veces aparece la letra *r* en alguna palabra corriente, o pídele que deletree una palabra al revés. Los modelos llevan años tropezando justo con preguntas así, a veces de forma cómica, en la misma conversación en la que resuelven un problema difícil de verdad. El patrón deja de parecer absurdo en cuanto tienes presente el tokenizador: la palabra llegó como uno o dos trozos opacos, y las letras que hay dentro de un trozo son precisamente la información que la tabla de consulta descartó. Para contarlas, el modelo tiene que reconstruir la ortografía a partir de lo que haya absorbido sobre esa palabra durante el entrenamiento: una tarea de memoria, y poco fiable. (Para cuando leas esto, puede que el modelo que pruebes la supere; esas lagunas se van parcheando con entrenamiento, o el modelo recurre discretamente a una herramienta. El mecanismo que el fallo deja al descubierto sigue estando debajo.) Los dígitos cuentan la misma historia: en al menos un tokenizador muy usado, 520 es un solo token mientras que 521 se divide en $5 + 21^4$, de modo que el modelo hace aritmética sobre trozos que ignoran el sistema de numeración: una de las razones por las que su cálculo mental es más inseguro de lo que su fluidez sugiere. La lección se generaliza: el mundo del modelo tiene un grano, y el grano es el token.

Tres consecuencias de la tokenización le importan a quien construye más que su funcionamiento interno. La primera: la proporción de tokens por palabra varía con el contenido; el código, los números y los datos estructurados se tokenizan con menos eficiencia que la prosa, así que una página de JSON cuesta más tokens que una página de ensayo. La segunda: la mayoría de los tokenizadores se entrenaron sobre todo con inglés, de modo que el mismo significado puede costar varias veces más tokens en otro idioma —un análisis encontró que el birmano necesitaba aproximadamente ocho veces el número de tokens del inglés equivalente, una cifra que cito como ilustración de la disparidad, no como una constante⁵—, lo que tiene consecuencias evidentes de costo y de equidad si tus usuarios escriben en muchos idiomas. La tercera: distinciones en las que nunca pensarías son reales para el tokenizador; la misma

⁴Qué números concretos se dividen es un hecho arbitrario del entrenamiento de cada tokenizador. En el tokenizador de pares de bytes de la generación GPT-2/GPT-3, por ejemplo, 520 es un solo token mientras que 521 se convierte en $5 + 21$; los tokenizadores posteriores tienden a conservar enteros los números cortos, pero siguen troceando los largos de forma irregular. Lo importante es que algunos números se dividen, y que el modelo no puede ver dentro de un trozo.

⁵Aleksandar Petrov et al., «Language Model Tokenizers Introduce Unfairness Between Languages», NeurIPS 2023 (arXiv:2305.15425), que mide disparidades de hasta aproximadamente quince veces en una serie de tokenizadores; la cifra de aproximadamente $8\times$ para el birmano es una ilustración conservadora de lo amplia que puede llegar a ser la brecha en los idiomas poco representados en el entrenamiento de los tokenizadores.

palabra, con espacio inicial o sin él, con mayúscula o sin ella, puede corresponder a tokens completamente distintos. El hábito práctico que se desprende de las tres: cuando un recuento importe, mide tu texto real con el tokenizador del propio proveedor o con su endpoint de recuento. Cada familia de modelos trocea el texto de forma distinta, así que un recuento tomado de una es solo una estimación para otra. (Observo que casi todos los números de esta sección llegan envueltos en un «aproximadamente». Con los tokenizadores, lo aproximado es el tono honesto.)

Ahora, la ventana en sí. La imagen que usaré para ella, aquí y durante el resto del libro, es una mesa de trabajo: la superficie que queda al otro lado del ojo de la cerradura, el lado del modelo, y en la que tiene que caber a la vez todo lo que el modelo consulta durante una sola llamada. Repartidos por esa mesa, en una llamada típica de un agente, están tus instrucciones permanentes, toda la conversación hasta el momento, las definiciones de todas las herramientas que el agente podría usar, el texto completo de todos los resultados de herramientas que han vuelto, los documentos que se hayan traído para la tarea y, además, la respuesta que el modelo está escribiendo, que va a parar a la misma mesa a medida que crece. Un único presupuesto de tokens lo cubre todo. No hay cajón, ni estante, ni segunda mesa. Si un dato está sobre la mesa durante esta llamada, el modelo puede usarlo; si está en cualquier otro lugar del universo, mientras dure esta llamada no existe.

La mesa, además, queda despejada entre una llamada y la siguiente, y aquí la metáfora necesita su frontera honesta: tu memoria de trabajo llega hasta la tarde, mientras que la del modelo no sobrevive hasta la siguiente solicitud. Una API de LLM funciona *sin estado* (cada llamada parte de cero), lo que da su forma operativa a lo que la sección anterior decía sobre la memoria: para continuar una conversación, el software que rodea al modelo vuelve a poner sobre la mesa la transcripción entera en cada turno, igual que tú releerías todo un hilo de correo antes de responder a su mensaje más reciente. El hábito es inofensivo con tres mensajes y ruinoso con trescientos. Un esbozo, con números inventados pero realistas: instrucciones permanentes, 500 tokens; conversación hasta el momento, 8000; dos documentos traídos, 40 000; el último resultado de una herramienta, 3000. El modelo lee más de cincuenta mil tokens antes de escribir una palabra, tú pagas cada uno de ellos, y en el turno siguiente el montón es todavía mayor, porque este turno ya forma parte del historial. (Los proveedores suavizan el costo de releer un montón que no ha cambiado mediante la caché, una de las principales palancas del capítulo 19.)

¿Qué pasa cuando el montón ya no cabe? Nada elegante. Según la API y el software que haya en medio, la solicitud puede rechazarse sin más con un error; o la entrada puede truncarse —recortarse hasta que quepa, normalmente eliminando los turnos más antiguos, a menudo sin avisar a nadie—; o un framework intermedio puede resumir el historial antes de enviarlo. El error sin más es el desenlace amable. El truncamiento silencioso es el traicionero: la

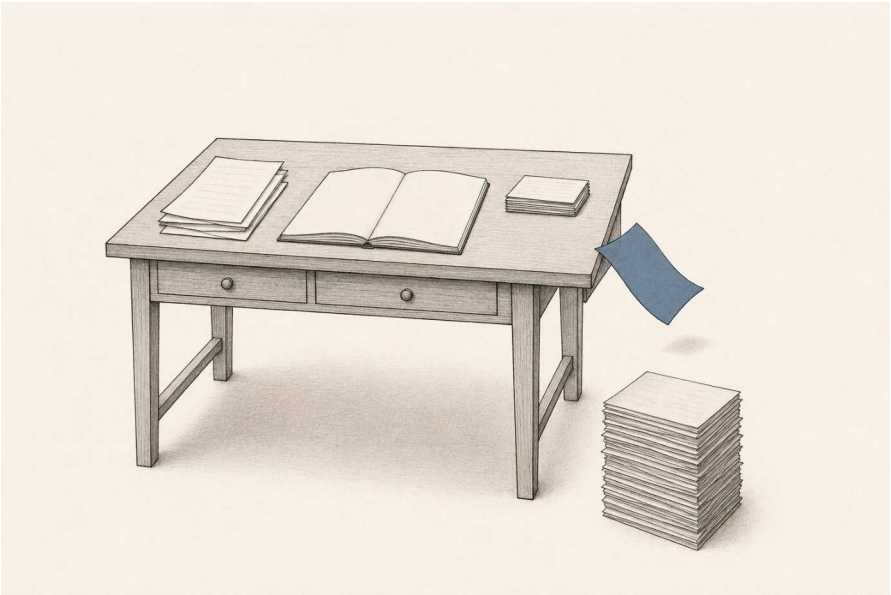


Figura 2.2: La ventana de contexto como una única mesa de trabajo finita: todo lo que el modelo puede usar en este momento tiene que caber físicamente en su superficie.

llamada sale bien, la respuesta vuelve fluida, y el modelo ha estado razonando sobre una transcripción amputada de la que tus instrucciones originales pueden haberse caído sin hacer ruido. Desde fuera, esto parece un modelo que se volvió más tonto de la noche a la mañana, y se lleva por delante las horas de depuración correspondientes. Cuando una sesión larga empiece a «olvidar», comprueba si hay desbordamiento antes de culpar al modelo. (Un primo de esta sorpresa: la entrada y la salida comparten la ventana, pero muchos modelos limitan aparte cada respuesta a una cifra mucho menor. Una respuesta que se detiene a mitad de frase normalmente chocó con el tope de salida.)

Una propiedad más de la mesa, que al capítulo 1 ya se le escapó: incluso lo que cabe se usa de forma desigual. En entradas largas, los modelos recuerdan el material cercano al principio y al final notablemente mejor que el material enterrado en el medio; la brújula llamó medio olvidado a un dato así, y la literatura de investigación llama al efecto, con acierto, «perdido en el medio» (*lost in the middle*)⁶. Por eso una mesa más grande resuelve menos de lo que parece: los bordes siguen nítidos mientras el medio crece. Esta es la base mecánica del segundo rumbo del libro, la escasez de contexto, y el oficio al que obliga —decidir qué merece siquiera estar en la mesa— ocupa la parte III, que empieza en el capítulo 7.

Incluso cuando todo cabe y está donde el recuerdo es bueno, el montón nunca es gratis. Los tokens son la unidad de tu factura en los dos sentidos, entrada y salida. Son también la unidad de tu espera: antes de poder emitir el primer token de su respuesta, el modelo tiene que leer el montón entero —una fase llamada *prefill*— y, en la arquitectura estándar, el costo de esa lectura crece muy deprisa con la longitud, aproximadamente con su cuadrado. Un contexto hinchado hace que el agente tarde más en empezar a responder y que cada llamada sea más cara, ayude o no el material adicional. Así que la ventana es un techo, no una meta; la pregunta útil es cuál es la mesa más pequeña en la que de verdad cabe tu tarea.

Para un chatbot, estos son datos de fondo. Para un agente, son el tiempo que hace cada día: el bucle añade cosas a la mesa en cada paso —razonamiento, llamada a herramienta, resultado de la herramienta, una y otra vez—, y nadie cura el montón si tú no construyes la curación. A los veinte pasos, la transcripción pesa por los listados de directorios caducados y los volcados de páginas relevantes solo a medias, la instrucción que de verdad importa está a mitad del montón, donde el recuerdo flaquea, y cada paso adicional lee, y factura, toda la acumulación. La mesa ya está medida. De la mecánica queda la escritura en sí: cómo elige el modelo un token entre sus conjeturas ordenadas, y por qué dos ejecuciones idénticas de tu agente no se comportarán de forma idéntica. Eso viene a continuación.

⁶Nelson F. Liu et al., «Lost in the Middle: How Language Models Use Long Contexts», *Transactions of the Association for Computational Linguistics* (2024). La exactitud en tareas de contexto largo era máxima cuando el pasaje relevante estaba al principio o al final de la entrada y decaía cuando estaba en el medio.

2.3 Cómo se genera el texto

Entrégale al modelo el fragmento «El gato se sentó en el» y, como describió el arranque de este capítulo, vuelve una lista ordenada: una probabilidad para cada token del vocabulario. Digamos que la parte alta de esa lista dice **sofá** con 0.42, **suelo** con 0.18, **sillón** con 0.13 y **felpudo** con 0.09, con los últimos restos de probabilidad repartidos entre miles de comparsas. (Los números son inventados; la forma es típica.) Ahora, la pregunta sobre la que gira esta sección, y me gustaría que te comprometieras con una respuesta antes de seguir leyendo: ¿qué palabra escribe el modelo? La respuesta que parece obvia es **sofá**: toma la mejor conjetura; ¿por qué iba una máquina a hacer otra cosa? Sin embargo, para la mayoría de los sistemas a los que llamarás en tu vida, esa respuesta es incorrecta. Por defecto, la maquinaria tira los dados.

El paso en cuestión —convertir una lista ordenada en un único token elegido— se llama *decodificación*, y la estrategia obvia tiene nombre: *decodificación voraz*, tomar siempre el token más probable. La voraz es determinista, barata y suena a opción responsable. También produce texto con una dolencia bien documentada: una prosa plana, extrañamente repetitiva, que puede quedar atrapada en bucles, con la misma frase asomando una y otra vez⁷. La razón es lo bastante contraintuitiva como para explicarla despacio. La escritura humana no deja de correr pequeños riesgos —un adjetivo inesperado, un giro nuevo—, mientras que un decodificador que nunca apuesta va derivando hacia el rincón más anodino del idioma y después da vueltas en él. La palabra localmente más segura, elegida todas las veces, da como resultado un pasaje globalmente mediocre.

La alternativa, y la opción por defecto en casi todas partes, es el *muestreo*: sortear el siguiente token al azar, ponderado por las probabilidades del modelo. **sofá** gana el sorteo 42 veces de cada cien; a **felpudo** le llega el turno unas nueve veces de cada cien. La imagen que conviene conservar es una *lotería ponderada*, que se celebra una vez por token: el modelo imprime las probabilidades y el decodificador saca un boleto. Todas las respuestas fluidas y de sonido natural que has admirado se ensamblaron así, sorteo a sorteo. Y aquí se está fijando una propiedad de tus futuros agentes, antes de que exista agente alguno: la variabilidad forma parte del diseño. Dos solicitudes idénticas pueden sacar boletos distintos, y en el momento en que un sorteo difiere, los dos textos se separan para siempre, porque cada token posterior está condicionado por los anteriores.

Entre lo voraz y lo desbocado hay mandos, y todos hacen una sola cosa: remodelar o recortar la lista ordenada antes del sorteo. Todos actúan después de la predicción: la lista ordenada está fijada, y los mandos solo gobiernan

⁷Ari Holtzman et al., «The Curious Case of Neural Text Degeneration», ICLR 2020. El artículo documentó cómo la decodificación por maximización degenera en repetición y presentó el muestreo de núcleo (top- p) como remedio.

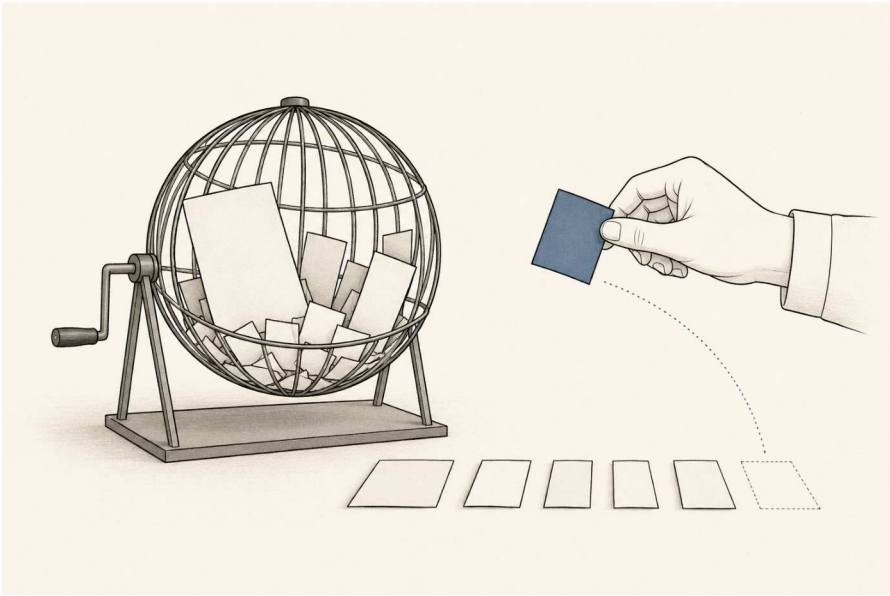


Figura 2.3: El muestreo como una lotería ponderada que se celebra una vez por token. El modelo imprime las probabilidades y el decodificador saca un boleto; el favorito suele ganar, pero no siempre, y cada sorteo ocupa su lugar en la frase que hay hasta el momento.

cómo eliges de ella. La *temperatura* es el dial de la nitidez. Al bajarla, ensancha la distancia entre los favoritos y los improbables hasta que, en cero, el token más probable gana todos los sorteos y vuelves a la decodificación voraz; al subirla, aplanas el campo, das opciones reales a los improbables y acaba por disolver el texto en ruido. Fíjate en lo que te da el dial: la temperatura baja te da consistencia, la alta, variedad, y la corrección no está en ningún extremo; una respuesta equivocada puede salir del sorteo con todo aplomo en cualquier ajuste.

Los otros dos mandos habituales recortan el campo en lugar de remodelarlo. *Top-k* conserva solo los k tokens de mayor probabilidad y descarta el resto: simple, pero k es un tamaño de grupo fijo, y el número de continuaciones genuinamente plausibles varía enormemente de un momento a otro. *Top-p* (también llamado *muestreo de núcleo*; en inglés, *nucleus sampling*) lo arregla conservando el conjunto más pequeño de los tokens más probables cuyas probabilidades suman al menos p : después de «La capital de Francia es», el grupo puede contener un único token; después de «Mi comida favorita es», puede contener decenas. El grupo se dimensiona según la propia seguridad del modelo, y por eso *top-p* es un valor por defecto habitual. La orientación práctica es breve. Elige el ajuste según la tarea —temperatura baja para todo lo que vaya a analizar un programa o comprobar una prueba, más alta para la lluvia de ideas y los borradores— y mueve un mando cada vez, ya que varios de ellos aprietan la misma distribución.

La generación también tiene que terminar, y termina de una de tres maneras. El modelo puede emitir un token especial de fin de secuencia —el entrenamiento le enseñó a decir «he terminado» en forma de vocabulario—, que es el camino feliz. Un tope estricto de longitud máxima puede cortar la salida a mitad de una idea; la sección anterior se lo encontró como la respuesta que se detiene a mitad de frase. O una *secuencia de parada* que aporta quien llama (una cadena que designas de antemano, a menudo un delimitador) pone fin a la generación en el momento en que se produce. Un hábito que conviene adquirir ya: las API suelen informar de cuál de estas cosas puso fin a la respuesta. Lee ese campo. Una respuesta truncada por el tope de longitud tiene exactamente el aspecto de un fallo del modelo, y en realidad es un presupuesto que fijaste tú.

Ahora, la pregunta que la sección anterior dejó en el aire: por qué dos ejecuciones idénticas de tu agente diferirán. La primera fuente de variación está a la vista. El muestreo es una lotería, así que cualquier temperatura por encima de cero significa que el mismo prompt puede sacar boletos distintos, bifurcarse y no volver a juntarse nunca. Esta fuente es intencionada y está por completo bajo tu control: pon la temperatura en cero y el sorteo se vuelve determinista. En teoría.

Aquí va un ejercicio de predicción para la intuición que ya tienes. Supón que se envía un mismo prompt mil veces a un modelo alojado, con temperatura cero (decodificación voraz, sin lotería). ¿Cuántas respuestas distintas

deberían volver? La aritmética de tu cabeza dice que una. Cuando un equipo de investigación hizo de verdad este experimento, obtuvo ochenta. Las mil respuestas fueron idénticas durante los primeros 102 tokens; en el token 103 —la frase había llegado al lugar de nacimiento del biografiado— 992 de ellas lo expresaron de una manera y 8 de otra, y a partir de esa bifurcación las transcripciones se dispersaron⁸. La explicación popular culpa al error de redondeo en hardware paralelo, y es incompleta. La explicación más completa tiene que ver con una propiedad llamada *invariancia al lote* (*batch invariance*), o más bien con su ausencia. Un endpoint alojado atiende a muchos clientes a la vez y agrupa sus solicitudes en lotes por eficiencia; el tamaño del lote sube y baja con el tráfico; y las rutinas numéricas de bajo nivel que hay dentro del modelo pueden devolver resultados minúsculamente distintos con tamaños de lote distintos. Con lo minúsculo basta. Cuando dos tokens están casi empatados en lo alto de la lista ordenada, un temblor en, digamos, el sexto decimal decide el ganador, la decodificación voraz lo elige fielmente, y el texto se bifurca. Desde tu lado de la API, la entrada decisiva es cuántos desconocidos estaban consultando por casualidad los mismos servidores en el mismo instante: una entrada que no puedes ver, controlar ni repetir. Existen versiones de esas rutinas que son invariantes al lote y que lo arreglan, a costa de velocidad de servicio; en el momento de escribir estas líneas son la excepción. Así que trata la temperatura cero como una preferencia fuerte, respetada casi siempre, y recuerda que ese «casi siempre», con una tirada de dados en cada token de cada paso, deja sitio para las sorpresas.

¿Qué debe hacer con todo esto quien construye? Tres hábitos. Primero, renuncia a la reproducibilidad de cadenas exactas como cimienta: escribe pruebas que comprueben propiedades de la salida —se puede analizar, nombra al cliente correcto, supera la comprobación—, porque una prueba que exige igualdad byte a byte fallará de forma intermitente y te hará perder semanas. Segundo, cuenta con que el fallo que viste una vez se resista a reproducirse a voluntad; eso es un dato sobre la infraestructura, y el capítulo 15 construye las herramientas de grabación que, aun así, hacen depurables esos fallos. Tercero, siente la consecuencia con forma de agente: un agente celebra esta lotería en cada token de cada paso, y la salida de cada paso se convierte en la entrada del siguiente, así que la variación entre ejecuciones se compone a lo largo del bucle: la misma tarea, un camino distinto, ambos plausibles. La respuesta a esto que mejor ha envejecido en el campo es hacer comprobable el resultado en lugar de idéntica la salida: el rumbo de la verificabilidad del capítulo 1, que llega ahora con su justificación mecánica.

Así que la máquina escribe por lotería, y la lotería explica la variedad.

⁸Horace He et al., «Defeating Nondeterminism in LLM Inference», Thinking Machines Lab (2025). El experimento: 1000 respuestas con temperatura cero a «Tell me about Richard Feynman» produjeron 80 salidas únicas, idénticas hasta el token 102 y que divergían por primera vez en la frase sobre el lugar de nacimiento («Queens, New York» frente a «New York City»). Las cifras ilustran el fenómeno, no una constante de la naturaleza.

La siguiente pregunta es cómo dirigirla. Si los pesos están congelados y a los dados les da igual, la única palanca que queda es el texto que pones sobre la mesa: qué decir, en qué orden y qué mostrar con ejemplos. Esa palanca es la siguiente sección.

2.4 Fundamentos de la escritura de prompts

Cuenta las palancas que de verdad tienes sobre esta máquina. Los pesos están congelados; nada de lo que hagas en tiempo de inferencia los cambiará. Los mandos de muestreo de la sección anterior ajustan cómo se sortea un token de la lista ordenada, nunca lo que el modelo predijo de entrada. Todo lo demás se reduce a un único instrumento: lo que decides escribir. Ese texto es el *prompt*, y el oficio de escribirlo bien —la *ingeniería de prompts*, un nombre grandilocuente para una práctica que es en parte escritura y en parte experimento— es la destreza básica de dirección sobre la que descansa todo lo agéntico. Sus fundamentos caben en esta única sección, y merecen la relectura aunque lleves años escribiendo prompts: un agente ensambla un prompt nuevo en cada paso de su bucle, así que una debilidad aquí es una debilidad que se repite en todo lo que construyas después.

Empieza por la anatomía. Los modelos de chat modernos reciben una lista de *mensajes*, cada uno etiquetado con un *rol*, y hay tres roles casi universales (sus nombres exactos y sus formatos de transmisión varían según el proveedor; trata cualquier sintaxis concreta como un detalle del proveedor). El mensaje *system* (de sistema) lleva las instrucciones permanentes: quién se supone que es el modelo, qué puede y qué no puede hacer, y qué forma deben tener sus respuestas. Se escribe una vez y persiste en todos los turnos (el software que rodea al modelo lo reenvía en cada llamada; el modelo en sí, recuerda, no retiene nada). El mensaje *user* (de usuario) lleva la solicitud del momento: esta pregunta, este ticket, este archivo. Y los mensajes *assistant* (de asistente) son las respuestas del propio modelo, que se acumulan como historial de la conversación; algo útil: también puedes escribir tú los mensajes de asistente, poniendo palabras en boca del modelo como forma de mostrar, en lugar de describir, cómo es una buena respuesta. Mi regla para repartir: si una instrucción tendría que repetirse en todos y cada uno de los turnos, va en el mensaje de sistema; si es específica de esta solicitud concreta, va en el mensaje de usuario. Confundirlos te pasa factura más adelante: una regla repetida en cada mensaje de usuario sobrevive solo mientras todos los que llaman se acuerden de repetirla. La figura 2.4 dispone los tres roles y la falta de estado que obliga a reenviar toda la pila en cada llamada.

La mayor palanca, con diferencia, es la claridad, y tras dos secciones de mecánica ya puedes ver por qué funciona. El modelo continúa texto; una instrucción vaga deja amplia la lista ordenada, y la lotería se pasea a sus anchas por todo el espacio que dejes abierto. Una instrucción precisa estrecha

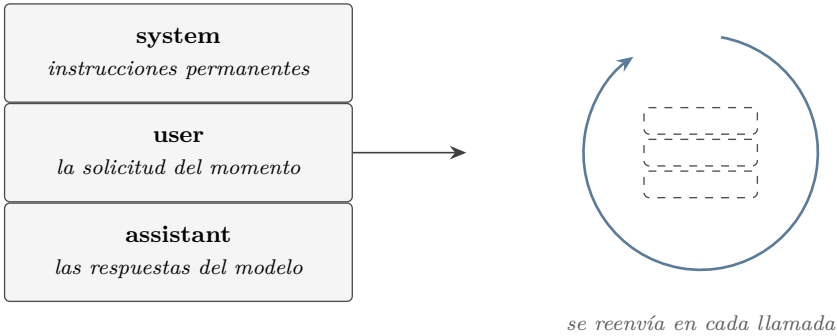


Figura 2.4: La anatomía de un prompt de chat. Las instrucciones permanentes van en el mensaje *system*, la solicitud del momento en el mensaje *user*, y las respuestas del propio modelo se acumulan como mensajes *assistant*. Como el modelo no retiene nada entre llamadas, el software que lo rodea reenvía la pila entera en cada turno (el bucle en el color de acento): lo que parece memoria es solo la transcripción, que se relea.

el campo antes de que rueden los dados. En la práctica esto significa preferir las restricciones que se pueden exigir a los adjetivos: «de tres a cinco frases» vence a «que sea corto», y «como máximo cinco viñetas, cada una de menos de doce palabras» vence a «sé conciso», porque lo primero de cada par se puede comprobar y lo segundo es un estado de ánimo. Significa decir lo que *no* quieres, ya que la servicialidad con la que se entrenó al modelo tiende a pasarse de la raya: «no inventes precios»; «si el cliente pide un reembolso, no se lo prometas». Y significa dar una alternativa para el caso en que la tarea no pueda hacerse: «si la respuesta no está en el texto proporcionado, di que no tienes información suficiente». Sin esa última línea, un modelo al que le falta la respuesta hará aquello para lo que fue entrenado, que es responder.

Aquí está la anatomía ensamblada, lo bastante pequeña como para leerla de un vistazo:

```
Eres un asistente de triaje de tickets de soporte.
Clasifica el ticket siguiente en exactamente una de:
    billing, bug, feature-request, other.
Ticket: "La app se cierra cada vez que abro
        los ajustes."
Responde solo con la palabra de la categoría
y nada más.
```

Siete líneas, cuatro funciones: un rol que fija el alcance, una instrucción con un conjunto cerrado de respuestas permitidas, los datos de entrada y un contrato de salida. Las frases que hacen el trabajo de verdad son las estrictas: «exactamente una de», la lista fija, «nada más». Ejecútalo y un modelo capaz devuelve **bug** casi siempre.

Casi. Supón que una ejecución de cada veinte vuelve como «Bug.», con mayúscula y punto, o como «Esto parece un bug en la pantalla de ajustes». Tu analizador se atraganta, y por más que mires la instrucción no le encuentras ningún defecto. El siguiente paso habitual es añadir *ejemplos*:

```
Ticket: "Me cobraron dos veces este mes."    -> billing
Ticket: "¿Pueden añadir un modo oscuro?"     -> feature-request
Ticket: "La app se cierra cada vez
       que abro los ajustes."                 ->
```

A escribir prompts solo con instrucciones se le llama *zero-shot* («sin ejemplos»); añadir demostraciones resueltas como estas es *few-shot* («con unos pocos ejemplos»); y la facilidad del modelo para captar el patrón tiene nombre propio, *aprendizaje en contexto* (*in-context learning*): el modelo se adapta a tu tarea a partir de ejemplos dentro del prompt, sin reentrenamiento de ningún tipo. Lo que enseñan los ejemplos es lo que el fallo de arriba mostró que faltaba (formato y tono), más una tercera cosa que las instrucciones describen mal: el tratamiento de los casos límite. Hay aquí un resultado de investigación instructivo: buena parte del beneficio de las demostraciones viene de su *estructura* —un formato consistente, un conjunto visible de etiquetas— y sobrevive incluso cuando algunas de las etiquetas individuales son erróneas⁹. Los ejemplos funcionan como un molde para la forma de la respuesta. Dos o tres bien elegidos captan la mayor parte de la ganancia disponible; cada ejemplo adicional cuesta tokens en cada llamada, y en el caso de un agente en cada paso, con rendimientos decrecientes.

Conviene saber también lo que las demostraciones no compran: una tarea sobre la que el modelo razona mal tiende a seguir saliendo mal por muchas respuestas terminadas que la precedan. Demostrar el *razonamiento* mismo es un recurso distinto, y recibe tratamiento propio en el capítulo 4. La progresión que sale de todo esto es agradablemente simple: empieza en *zero-shot*; añade dos o tres ejemplos cuando la forma de la salida se tambalee; busca en otra parte cuando el problema sea el contenido.

El formato de salida merece su propia instrucción explícita siempre que algo posterior dependa de él. Para la prosa destinada a personas, nombra las secciones, su orden y los límites de longitud. Para una salida que va a analizar un programa, di la forma con claridad, y entiende que una instrucción es una solicitud, no una garantía. El modelo puede respetar «responde solo con la palabra de la categoría y nada más» mil veces y después, en la ejecución mil uno, envolver la palabra en una frase cortés. Donde el análisis importa, la solicitud puede ascender a contrato, y ese ascenso es el tema de la siguiente sección.

⁹Sewon Min et al., «Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?», EMNLP 2022. Las demostraciones con etiquetas sustituidas al azar conservaron buena parte del beneficio de las correctamente etiquetadas; el formato y el espacio de etiquetas se llevaban una gran parte del efecto.

Unas palabras sobre los *personajes* (*personas*), el rincón más folclórico de la escritura de prompts. Decirle al modelo quién debe ser —«eres un revisor de seguridad sénior», «escribes como una guía de campo»— moldea de forma fiable la voz, el tono y el alcance, y para la escritura abierta esa es a menudo justo la palanca que quieres. El folclore va más lejos y dice que un personaje también hace las respuestas más *exactas*: dile que sea un experto y se convierte en uno. El panorama de la investigación es mucho menos halagüeño: en tareas factuales y analíticas, los personajes simples muestran efectos dispares y a menudo insignificantes en modelos capaces¹⁰. Usa los personajes para fijar la voz y los límites; usa instrucciones, ejemplos y un apoyo genuino al razonamiento para la corrección.

Dos hábitos de colocación completan la caja de herramientas. Primero, cuando una tarea tiene varias partes, detállalas como pasos ordenados —«primero lista los nombres de las empresas, después las personas, después los temas recurrentes»— en lugar de señalar vagamente el conjunto («extrae las entidades»). Los pasos explícitos mejoran la salida y, lo que es igual de valioso, te dicen *qué* paso falló cuando alguno falla; volverás a encontrar este instinto, ya crecido, cuando los agentes descompongan proyectos enteros en el capítulo 4. Segundo, la posición importa. Cuando un prompt empieza con páginas de material —un documento, una transcripción, un montón de resultados de herramientas—, pon la instrucción *después* del montón y repite cerca del final la única regla que soporta carga. Las dos mitades de la razón ya son tuyas: el modelo es un continuador, así que un prompt que termina a mitad de documento invita a continuar el documento en lugar de la tarea; y los bordes de la mesa siguen nítidos mientras el medio flaquea, así que las últimas líneas están donde el recuerdo es más fuerte. Lo último que lea el modelo debe ser lo que más necesitas que haga.

No voy a fingir que nada de esto equivale a una ciencia. Los prompts son objetos empíricos: dos redacciones que consideras idénticas pueden comportarse de forma distinta, y un prompt que hoy funciona puede desviarse cuando se actualiza el modelo que hay detrás. Trata los prompts como tratas el código —versionados, probados, vueltos a probar tras cada cambio de modelo—; el capítulo 16 construye el aparato. Y quédate con el cambio de perspectiva hacia el que esta sección ha estado llevándote. En un agente, nadie teclea el prompt final. El mensaje de sistema se escribe una vez; las definiciones de las herramientas las inyecta la maquinaria; el historial se acumula turno a turno; los documentos recuperados y los resultados de herramientas los coloca el código en tiempo de ejecución. Todos los principios anteriores siguen valiendo —alguien sigue eligiendo lo que lee el modelo—, solo que ese alguien pasa a ser

¹⁰Mingqian Zheng et al., «When “A Helpful Assistant” Is Not Really Helpful: Personas in System Prompts Do Not Improve Performances of Large Language Models», Findings of EMNLP 2024; el repaso de PromptHub «Role-Prompting: Does Adding Personas to Your Prompts Really Make a Difference?» reúne este y otros estudios relacionados. Los personajes elaborados y ajustados al ámbito han mostrado mejoras en algunos contextos; un escueto «eres un experto», en general, no.

un programa que escribes tú. Decidir, paso tras paso, qué se gana un lugar en la mesa es un problema lo bastante grande como para llevar nombre propio, *ingeniería de contexto*, y la parte III de este libro está dedicada a él. Pero la parte III queda lejos, y el hueco más cercano está a mano: una instrucción solo puede *solicitar* un formato. La siguiente sección muestra cómo garantizarlo.

2.5 Salida estructurada y llamada a funciones

El capítulo 1 describió la llamada a funciones como el puente del texto a la acción y prometió que este capítulo aportaría el mecanismo. Esta sección cumple la promesa, y hay que empezar por la pregunta de quién lee la salida del modelo. Mientras la lea una persona, la redacción puede vagar libremente; las personas perdonan. En el momento en que la lee un *programa*, ese vagar rompe cosas, y todo lo que hace un agente depende de programas que leen la salida del modelo.

Siente el problema antes de resolverlo. Supón que tu código le pide al modelo que extraiga dos datos del correo de una clienta —su nombre y el total del pedido— para escribirlos en una base de datos. Tres ejecuciones, tres respuestas, todas correctas como prosa:

```
Nombre: Ada, Total: $42
La clienta es Ada Lovelace y gastó cuarenta y dos dólares.
¡Con gusto! El pedido de Ada suma $42.00 (factura #1041).
```

Antes de seguir leyendo, esboza el código de análisis que maneje las tres. Lo que hayas escrito para la primera se rompe con la segunda; lo que maneje la segunda tropieza con el paréntesis de la tercera; y mañana llega una cuarta redacción, porque la lotería de hace dos secciones no te debe ninguna consistencia. Ese es el problema de raíz: el lenguaje natural tiene una variabilidad superficial enorme, y nada obliga al modelo a expresar el mismo dato de la misma manera dos veces. Un programa no puede consumir posibilidades. Necesita un *contrato*: una forma fija a la que se garantiza que la salida se ajusta.

El contrato estándar es *JSON* (un formato de texto simple y universal para datos anidados de tipo clave-valor), con su forma fijada por un *esquema*: una declaración de qué campos existen, de qué tipo es cada uno, cuáles son obligatorios y qué valores se permiten. Para la tarea de extracción, en pseudocódigo (la sintaxis exacta de cada proveedor difiere):

```
{ name:      string,                required
  total:    number,                required
  currency: one of "USD" | "EUR" | "GBP",
  no other fields permitted }
```

Una respuesta que se ajusta a este contrato se deserializa directamente en una estructura de datos que el resto de tu programa ya entiende: los campos

están donde el código los espera, el total es un número, y la frase cortés sobre la factura #1041 nunca llega a producirse.

Cómo se consigue que el modelo cumpla importa más de lo que parece a primera vista, porque hay dos grados de garantía y suenan parecido. El grado débil es pedir: instrucciones de formato en el prompt, a menudo respaldadas por una opción de la API (llamada habitualmente «modo JSON») que garantiza que la salida al menos *se puede analizar* como JSON. Que se pueda analizar es un listón bajo. El modelo todavía puede omitir un campo obligatorio, inventarse uno de más o poner una cadena donde va el número: la sintaxis está garantizada; la forma, tan solo deseada. El grado fuerte es la *decodificación restringida*, y ya tienes la imagen que necesita. Cada token, recuerda, se elige de una lista ordenada mediante un sorteo. La decodificación restringida compila tu esquema en un filtro que actúa en todos y cada uno de los pasos y tacha de la lista, antes de que se haga el sorteo, cada token que violaría el esquema. Si el esquema dice que a continuación va un número, ninguna letra puede ganar; si la lista de campos es cerrada, ningún campo inventado puede empezar. El modelo elige libremente entre las continuaciones válidas y no puede emitir una inválida: la conformidad se impone en la maquinaria en lugar de solicitarse en el prompt. Los proveedores lo ofrecen con nombres como «structured outputs» o modos «strict» para las herramientas; el precio es un poco de sobrecarga en la decodificación y algunos límites a lo elaborado que puede ser el esquema.

Una frontera que quiero dejar marcada con tinta: una garantía sobre la forma no dice nada sobre la verdad. La decodificación restringida promete que el formulario vuelve relleno; que esté relleno *con la verdad* es una cuestión completamente distinta. Si el modelo no conoce el total del pedido, el esquema no le hará confesar su ignorancia: le hará producir un número bien tipado. Un esquema mal usado hace que la invención sea más fácil de consumir, no más rara. Así que la validación se queda en tu código en cualquier caso: rangos, reglas entre campos, «¿existe de verdad este identificador de cliente?». El esquema vigila el formato; tú sigues siendo quien audita el contenido.

Ahora apunta la misma idea hacia las acciones. La *llamada a funciones* (*function calling*) —o, indistintamente, *llamada a herramienta*— empieza cuando describes, junto al prompt, un conjunto de funciones que el modelo puede solicitar: cada una con un nombre, una descripción en lenguaje natural y un esquema para sus argumentos. El modelo nunca ejecuta nada. No tiene manos; solo escribe. Lo que sí puede hacer, cuando juzga que una acción es útil, es emitir una solicitud estructurada en lugar de prosa —un pequeño objeto que nombra una función y aporta argumentos que se ajustan al esquema— y después detenerse y esperar. El intercambio va así (de forma esquemática; los formatos de transmisión varían según el proveedor):

1. tú → modelo: la pregunta del usuario
+ la lista de definiciones de herramientas
2. modelo → tú : { call: "get_weather",

```

arguments: { city: "Paris" } }
3. tu código      : ejecuta la función real get_weather("Paris")
4. tú      -> modelo: la conversación hasta ahora
                  + { result: { temp_c: 18 } }
5. modelo -> tú    : "Ahora mismo hay unos 18 grados en París."
                   (u otra llamada a herramienta, y el ciclo se repite)

```

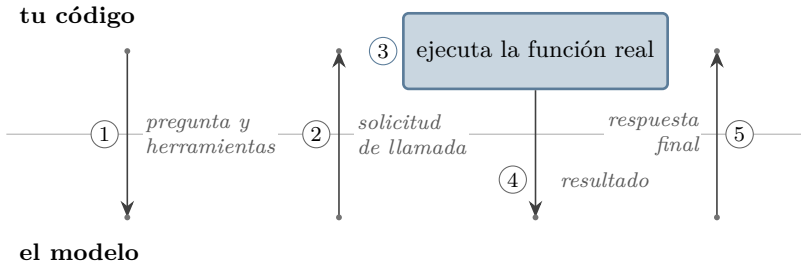


Figura 2.5: La llamada a funciones, paso a paso. Tu código envía la pregunta y la lista de herramientas; el modelo responde, en lugar de con prosa, con una solicitud estructurada para llamar a una de ellas; tu código ejecuta la función real (en el color de acento, el paso 3, el único lugar donde ocurre un efecto secundario); el resultado de la herramienta vuelve al modelo, que entonces escribe la respuesta final. El modelo propone; tu código dispone.

Dos propiedades de este intercambio —que la figura 2.5 dispone en dos carriles— merecen subrayarse. La primera es el reparto de autoridad: *el modelo propone; tu código dispone*. El capítulo 1 usó esa frase de pasada, y ahora puedes ver que es literal. Todo efecto secundario del sistema ocurre en el paso 3, dentro de código ordinario que escribiste tú, donde puedes registrarlo, probarlo, restringirlo o rechazarlo; toda la aportación del modelo es una estructura de datos que expresa un deseo. La segunda es que el intercambio tiene *dos* llamadas al modelo como mínimo, y en la segunda es donde tropiezan los principiantes: el primer bug clásico es ejecutar la herramienta y entregarle su resultado en bruto al usuario, cuando el resultado debe volver al *modelo* (paso 4), porque solo el modelo sabe por qué lo pidió y qué hacer a continuación. (Y cuando compongas ese mensaje del paso 4, devuelve los pocos campos que el modelo necesita en lugar de todo el bloque que entrega la API; cada byte del resultado va a parar a la mesa.)

Una definición de herramienta, para concretar el otro lado (de nuevo en pseudocódigo):

```

name:      create_ticket
description: Abre un ticket de soporte. Úsala cuando el usuario
            informe de un problema que necesite seguimiento
            humano.
arguments: { title:      string,                                required

```

```
severity: one of "low" | "med" | "high", required
body:      string }
```

Esta definición —nombre, descripción, esquema de argumentos— es todo lo que el modelo sabrá jamás sobre tu herramienta, y el campo **description** es la pieza que más quiero que mires con atención, porque es la que los recién llegados subestiman. La decisión de si llamar a **create_ticket**, y cómo, se toma solo a partir de esta definición: no hay código fuente que consultar, ni wiki, ni colega a quien preguntar. Escríbela como documentación de API para una persona capaz recién contratada, en su primera mañana: qué hace la herramienta, cuándo recurrir a ella, cuándo su vecina es la mejor opción. Cuando un modelo usa mal una herramienta, o ignora una que habría ayudado, la descripción es la primera sospechosa, por delante del modelo. Diseñar las herramientas mismas, su número, su granularidad y sus mensajes de error, es un oficio que tiene capítulo propio (capítulo 5).

La realidad pondrá a prueba los caminos infelices, así que planifícalos. El modelo puede emitir argumentos que no superan la validación, nombrar una herramienta que no existe o hacer varias llamadas en un mismo turno: tu manejador debe esperar cero, una o muchas. La recuperación que funciona sorprendentemente bien es devolver el fallo al modelo como datos: devuelve un mensaje de error estructurado como resultado de la herramienta («no existe la herramienta **create_tickets**; las herramientas disponibles son ...») y deja que el modelo lo lea y rectifique en su siguiente turno. Limita esos reintentos con un presupuesto estricto, tras el cual la ejecución falla ruidosamente; nunca debe permitirse que una confusión obstinada dé vueltas a tu costa. El capítulo 18 convierte este instinto en una disciplina.

Una advertencia de cierre, la que esta sección debe dejar a la vista antes de entregarte el puente: «bien formado» es una afirmación sobre la sintaxis, y «seguro» es una afirmación sobre las consecuencias. Una llamada a **delete_account** impecablemente conforme al esquema sigue borrando la cuenta. Si *debe* ejecutarse —autorización, confirmación, límites— es una decisión que le toca tomar a tu entorno de ejecución en el paso 3, y los capítulos 12 y 17 tratan de tomarla bien. Con esa advertencia a la vista, el puente está tendido. Todo lo que este libro llame «herramienta» de aquí en adelante es exactamente este mecanismo, y un agente —como dijo el capítulo 1 y como por fin construirá el capítulo 3— es este intercambio de solicitar, ejecutar y devolver, puesto a dar vueltas en un bucle. Queda una pieza de preparación: un mapa honesto de las formas en que falla el motor.

2.6 Limitaciones y modos de fallo

Quien hace ingeniería con cuidado lleva dos listas sobre cualquier componente del que depende: lo que hace bien y cómo falla. Este capítulo ha sido sobre todo la primera lista. Esta sección es la segunda, e importa más de lo

que su extensión sugiere, porque cada salvaguarda (*guardrail*), cada evaluación y cada patrón de verificación de las partes posteriores del libro existe a causa de alguna entrada de aquí. Dos notas de encuadre antes de la lista. Estas son propiedades de la generación actual de modelos: algunas se suavizarán a medida que mejoren los métodos de entrenamiento, y ninguna, en el momento de escribir estas líneas, está resuelta; en qué punto están tus modelos es algo que hay que verificar, y este libro no dejará de enseñarte cómo. Y la mayoría de las entradas se siguen directamente de lo que el motor *es*: una máquina de plausibilidad moldeada por las preferencias humanas tiene maneras características de equivocarse, y esas maneras responden a la ingeniería, rara vez a una redacción más severa del prompt.

La primera entrada ya la conoces en principio: la *alucinación*, el término del campo para una salida fluida, concreta, segura de sí misma y errónea, como una cita inventada, una función que no existe o una fecha plausible en la que no pasó nada. El mecanismo quedó expuesto al comienzo de este capítulo: cuando al modelo le falta un dato, la maquinaria produce la cadena de aspecto más plausible allí donde debería ir el dato, porque producir cadenas plausibles es todo lo que hace. Lo que vuelve operativamente peligrosa la alucinación es la entrega: la respuesta inventada llega exactamente con el mismo tono seguro que la correcta. El tono no lleva ninguna información sobre la verdad: tú no puedes oír la diferencia y, en cualquier sentido útil, el modelo tampoco. Un análisis afila la cuestión de forma incómoda: a fuerza de calificarlos, enseñamos a estos sistemas a adivinar, porque tanto el entrenamiento como los benchmarks premian un intento hecho con aplomo por encima de una admisión de incertidumbre, exactamente igual que un examen de opción múltiple premia al estudiante que nunca deja una respuesta en blanco¹¹. Peor aún, las invenciones reclutan refuerzos: una vez que una afirmación falsa queda en la transcripción, el modelo —condicionado, como siempre, por todo lo anterior— tiende a seguir construyendo sobre ella de forma coherente, un patrón conocido como *efecto bola de nieve de las alucinaciones* (*hallucination snowballing*)¹².

La segunda entrada derrota la intuición que más te gustaría conservar: que la capacidad es uniforme, que un sistema tan bueno en las cosas difíciles tiene que ser al menos igual de bueno en las fáciles. Viste el contraejemplo en miniatura antes en este capítulo: un motor que resuelve de un tirón un problema difícil de verdad y, acto seguido, no puede contar de forma fiable las

¹¹Adam Tauman Kalai et al., «Why Language Models Hallucinate» (2025), arXiv:2509.04664. El artículo sostiene que estas invenciones «se originan sencillamente como errores de clasificación binaria» durante el entrenamiento y persisten porque «los modelos de lenguaje están optimizados para ser buenos haciendo exámenes, y adivinar cuando no se está seguro mejora el resultado del examen».

¹²Muru Zhang et al., «How Language Model Hallucinations Can Snowball» (2023), arXiv:2305.13534. Los modelos a los que se había llevado a comprometerse con una afirmación errónea pasaron a generar detalles de apoyo falsos que, cuando se les preguntaba por separado, sabían identificar correctamente como falsos.

letras de una palabra. El patrón se generaliza, y se ha ganado un nombre memorable: la *frontera irregular* (*jagged frontier*). En un experimento de campo prerregistrado (uno cuyos análisis se fijaron de antemano, para que los hallazgos no pudieran seleccionarse a conveniencia), varios cientos de consultores trabajaron en tareas realistas con y sin la ayuda de un modelo. En las tareas situadas dentro de la frontera de competencia del modelo, el grupo asistido completó apreciablemente más tareas, más rápido y con una calidad mejor valorada. En una tarea elegida deliberadamente para quedar justo fuera de la frontera, el grupo asistido tuvo diecinueve puntos porcentuales *menos* de probabilidad de llegar a la respuesta correcta¹³. De ese resultado salen dos lecciones. El borde de la frontera es invisible desde donde estás: la tarea fallida no parecía más difícil; los investigadores la eligieron precisamente porque parecía comparable. Y la competencia en la puerta de al lado no demuestra nada sobre la competencia aquí: la única manera fiable de saber si tu modelo resuelve tu tarea es probar tu modelo en tu tarea. Esa frase vuelve en el capítulo 16, donde se convierte en una disciplina.

Tercera, y breve: el modelo es sensible a la redacción de maneras que ninguna especificación toleraría. Cambia el orden de dos cláusulas, sustituye un sinónimo, da otro formato a una lista, y la respuesta puede cambiar, porque dos prompts que tú consideras equivalentes son, para la máquina, sencillamente dos secuencias de tokens distintas. Esta es la razón mecánica por la que la sección anterior te dijo que trataras los prompts como objetos empíricos: probados como el código, y vueltos a probar cuando cualquier cosa aguas arriba cambia las palabras que les llegan.

La cuarta es la *adulación* (*sycophancy*), el sesgo del modelo a decirte lo que parece que quieres oír. Haz el experimento por tu cuenta. Hazle a un modelo una pregunta factual que responda bien; después contesta: «No creo que eso sea correcto». Con una regularidad inquietante, el modelo se disculpa y abandona la respuesta correcta, sin que se haya ofrecido ninguna evidencia nueva ni se haya pedido ninguna. El rastro lleva, al menos en parte, al posentrenamiento: los modelos se ajustan hacia las respuestas que prefieren los evaluadores humanos, los evaluadores prefieren con bastante frecuencia el acuerdo y el halago, y el modelo aprendió bien la lección¹⁴. La adulación muerde más fuerte justo donde más te gustaría apoyarte en el modelo: en la comprobación del trabajo. «¿Estás seguro?» es una verificación débil, porque

¹³Fabrizio Dell'Acqua et al., «Navigating the Jagged Technological Frontier», Harvard Business School Working Paper 24-013 (2023). El experimento prerregistrado contó con 758 consultores; las ganancias dentro de la frontera fueron un 12.2% más de tareas completadas, una finalización un 25.1% más rápida y una calidad valorada más de un 40% superior, frente a un déficit de exactitud de 19 puntos porcentuales en la tarea diseñada para quedar fuera de la frontera. El artículo acuñó el término.

¹⁴Mrinank Sharma et al., «Towards Understanding Sycophancy in Language Models», Anthropic (2023; publicado en ICLR 2024). El estudio documenta el comportamiento adulator en varios asistentes de frontera y lo atribuye en parte a juicios de preferencia humanos que favorecen las respuestas complacientes.

el modelo tiende a ratificar el marco que la pregunta le entrega: presiona con escepticismo y cede; pregunta con aprobación y da su visto bueno. Cuando este libro insista más adelante en que un verificador real vence al juicio del modelo sobre sí mismo (capítulo 4) y en que un modelo usado como juez debe validarse a su vez (capítulo 16), la adulación será buena parte de la razón.

Quinta: el cálculo exacto. Un predictor del siguiente token no contiene ninguna calculadora, y antes en este capítulo viste cómo hasta el sistema de numeración se disolvía en trozos arbitrarios. La aritmética de varias cifras, el conteo preciso, las cadenas largas de deducción exacta: todo eso se tambalea, y cuando se tambalea a mitad de cadena el modelo no se detiene; continúa con fluidez a partir del valor intermedio equivocado. El remedio es arquitectónico y barato: envía el trabajo exacto a algo exacto —una calculadora, una base de datos, código ejecutado— y deja que el modelo orqueste en lugar de calcular. El capítulo 5 hace de esto un principio de diseño.

Sexta, una entrada de la que solo daré un adelanto porque el capítulo 17 la trata a fondo: el modelo lee instrucciones y datos en un único flujo indiferenciado, y no puede distinguirlos de forma fiable. Un texto que simplemente pasa por el contexto —una página web traída, un correo, el resultado de una herramienta— puede dirigir el comportamiento del modelo si tiene forma de instrucción. La familia de ataques es la que el capítulo 1 nombró de pasada, la inyección de prompts, y se gana su lugar en esta lista por una razón: es una propiedad del motor, presente antes de que hayas escrito una sola línea de código de agente. (El no determinismo y el conocimiento caducado, los demás sospechosos habituales, tuvieron su tratamiento antes en este capítulo.)

Tomadas de una en una, son molestias que tienen remedio. La razón por la que merecen residencia permanente en tu cabeza es lo que ocurre cuando las encadenas. La brújula del capítulo 1 hizo esta aritmética una vez, como afirmación; aquí es donde se defiende, y adonde los capítulos posteriores (los capítulos 16 y 18 entre ellos) remitirán en lugar de volver a derivarla. Un agente es una cadena de pasos dependientes: cada llamada al modelo razona sobre los resultados de las llamadas anteriores. Supón que cada paso sale bien con probabilidad p , de forma independiente, y que una tarea necesita n pasos; la ejecución completa sale bien entonces con una probabilidad de aproximadamente p^n . Ahora pon números honestos. Una fiabilidad por paso del 95% suena excelente, y en una demo lo es. A lo largo de una tarea de veinte pasos: $0.95^{20} \approx 36\%$. A lo largo de treinta pasos con un 92%, algo peor: $0.92^{30} \approx 8\%$. La cifra por paso se habría ganado una bonificación; la cifra de extremo a extremo es una moneda por la que nunca apostarías. Las exponenciales no negocian. La figura 2.6 dibuja la caída.

Y la multiplicación limpia es la versión optimista, porque supone que los pasos fallan de forma independiente. No es así. Los errores de un agente van a parar a su propia transcripción, donde —por la lógica de todo este capítulo— se convierten en contexto que condiciona todo lo que viene después. El efecto bola de nieve de las alucinaciones era este patrón aplicado a los

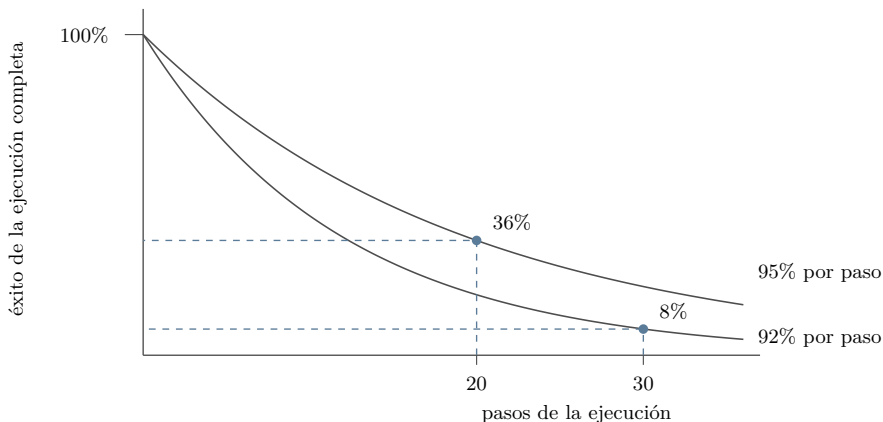


Figura 2.6: El error compuesto, dibujado. El éxito de la ejecución completa es la fiabilidad por paso elevada al número de pasos encadenados, así que decae exponencialmente: un 95% por paso se queda en torno al 36% tras veinte pasos, y un 92% por paso en torno al 8% tras treinta (los puntos en el color de acento). Una cifra por paso excelente se convierte en una moneda por la que nunca apostarías; las exponenciales no negocian.

hechos; vale también para las acciones. Una ejecución que ha empezado a torcerse razona, de ahí en adelante, sobre una mesa parcialmente equivocada, y su tasa de error por paso va subiendo a medida que la ejecución se alarga; quienes miden tareas largas describen exactamente esta firma: una fiabilidad que decae con la longitud de la tarea más deprisa de lo que predeciría la independencia¹⁵. Un agente confundido sigue confundido y, si se le deja solo, lo está cada vez más.

Si la aritmética es implacable, también es esclarecedora: te dice exactamente dónde están las palancas, y son tres. Reduce *n*: fusiona pasos, calcula de antemano lo que pueda calcularse de antemano y lleva el grueso del trabajo mecánico a código ordinario, que no tira dados (el capítulo 5 muestra lo lejos que llega esto). Sube el *p efectivo*: verifica en cada paso y recupérate del fallo, para que un error cueste un reintento en lugar de la ejecución entera; esa maquinaria ocupa la parte V. Y abarata el precio de morir: crea puntos de control, para que cuando una ejecución falle se reanude desde el último estado bueno en lugar de desde cero (capítulo 18). Lo que no puedes hacer es ignorar el exponente y cruzar los dedos. La exactitud por paso es un techo, no un pronóstico.

¹⁵El efecto de autocondicionamiento —la exactitud por paso disminuye a medida que los errores anteriores del propio modelo se acumulan en su contexto— se describe en trabajos sobre ejecución de horizonte largo, p. ej., Sinha et al., «The Illusion of Diminishing Returns: Measuring Long Horizon Execution in LLMs» (2025), arXiv:2509.09677.

El motor ya está cartografiado, que es todo lo que un mapa puede prometer. En un capítulo has reunido un predictor que continúa texto y no guarda ningún plan en reserva; una mesa que es finita, cara, que queda despejada entre llamadas y es más nítida en los bordes; una lotería que escribe token a token y nunca promete el mismo texto dos veces; una disciplina de dirección que trabaja enteramente con texto; un puente que convierte el texto en solicitudes comprobadas y ejecutables; y un catálogo de fallos cuya entrada principal es el exponente. Nada de ello exigió más matemáticas que la aritmética, y todo ello empieza a soportar carga de inmediato: la parte II toma este motor —brillante, olvidadizo, demasiado seguro de sí mismo, movido por los dados— y lo mete en un bucle.

Fin de la muestra

Has leído el prefacio y la Parte I: Fundamentos.

El libro completo continúa con seis partes más: construcción de agentes, ingeniería de contexto, patrones, fiabilidad, operación y aplicaciones, además de tres apéndices.

leanpub.com/ingenieria-de-agentes-de-ia