EDICIÓN #017

Patrones de diseño para Agentes de IA

15 de Enero de 2026

Patrones de diseño para Agentes de IA

Descubre la arquitectura real detrás de los agentes autónomos. Te explico los 7 patrones de diseño clave como ReAct, Swarm y Routing para evitar el desperdicio de tokens.

15 January 2026

2025 terminó con dos noticias que no deberíamos pasar por alto. La primera es **Meta comprando Manus por más de 2.000 millones de dólares**. La segunda, Claude Code alcanzaba una facturación de 1.000 millones anuales. Si algo nos han dejado claro estos movimientos es que la era de los **agentes autónomos** ha dejado de ser una promesa de futuro para convertirse en la realidad del desarrollo de software actual.

Pero aquí surge el problema de siempre. **Todo el mundo quiere construir agentes**, pero muy poca gente se ha parado a pensar en cómo diseñarlos para que no sean una máquina de quemar dinero en forma de Tokens.

La mayoría de los desarrolladores que se lanzan a esto cometen el error de pensar que agente significa simplemente darle a un LLM un bucle infinito, acceso a herramientas y a internet. Y no funciona así. La realidad es que estamos viendo cómo los modelos, por muy avanzados que sean, empiezan a sufrir lo que se conoce como "context rot" o deterioro del contexto. **A medida que las tareas se vuelven más largas y complejas, la capacidad del modelo para mantener la coherencia cae en picado**.

Hoy quiero profundizar en la arquitectura real que hay detrás de estos sistemas. No voy a ponerte ni una sola línea de código aquí, porque para eso he preparado un repositorio externo con todos los ejemplos implementados que te dejaré referenciado. Lo que quiero es que **entiendas los patrones de diseño**, la **lógica que hay detrás** y, sobre todo, **cuándo usar cada uno**. Porque créeme, la mayoría de las veces no necesitas un agente complejo, necesitas un flujo de trabajo bien diseñado. Con los agentes de IA menos es más.

Resumen ejecutivo (TL;DR)

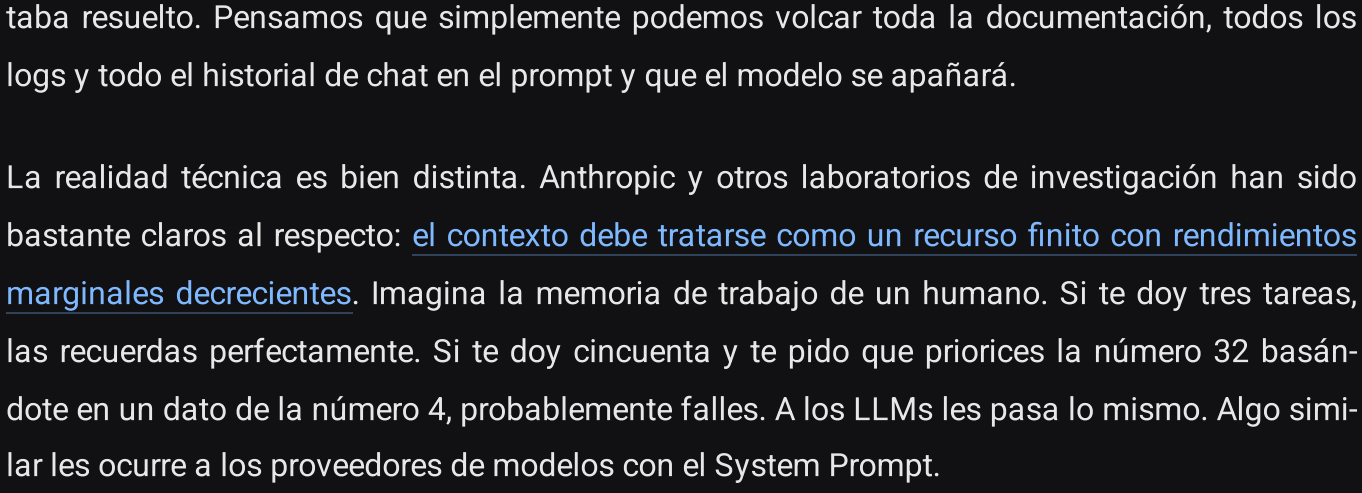
Este artículo es una **guía de arquitectura para diseñar sistemas de IA que funcionen en producción**, evitando la complejidad innecesaria y el desperdicio de tokens.

- No todo necesita ser un agente autónomo. **La mayoría de problemas se resuelven mejor con flujos de trabajo (casi) deterministas**, es decir, workflows, que además son más baratos, rápidos y fáciles.
- El contexto es un recurso finito y caro, y llenarlo de basura provoca **context rot**, haciendo que el modelo pierda precisión exponencialmente a medida que avanza la tarea.
- Existen **patrones de diseño probados** como el Enrutamiento (Routing), la Paralelización y la Reflexión que permiten estructurar el razonamiento de la IA para obtener resultados fiables.
- Para tareas complejas, patrones como el Orquestador-Trabajador (**Orchestrator-Workers**) o la Descomposición Jerárquica permiten dividir problemas grandes en piezas manejables por agentes especializados.
- El futuro de los agentes pasa por darles acceso al sistema operativo (sistema de ficheros y terminal) y gestionar su memoria de forma externa, no intentando meterlo todo en la ventana de contexto del LLM.

Gestión de memoria en LLMs — ¿Qué es el *Context Rot*?

Existe una **idea equivocada bastante extendida** de que con la llegada de modelos con ventanas de contexto de 200.000, un millón o dos millones de tokens, el problema de la memoria estaba resuelto. Pensamos que simplemente podemos volcar toda la documentación, todos los logs y todo el historial de chat en el prompt y que el modelo se apañará.

La realidad técnica es bien distinta. Anthropic y otros laboratorios de investigación han sido bastante claros al respecto: el contexto debe tratarse como un recurso finito con rendimientos marginales decrecientes. Imagina la memoria de trabajo de un humano. Si te doy tres tareas, las recuerdas perfectamente. Si te doy cincuenta y te pido que priorices la número 32 basándote en un dato de la número 4, probablemente falles. A los LLMs les pasa lo mismo. Algo similar les ocurre a los proveedores de modelos con el System Prompt.



En un extremo del espectro, vemos indicaciones frágiles si-demás codificadas, y en el otro extremo vemos indicaciones que son demasiado generales o asumen falsamente contexto compartido.

Fuente: Antrophic

Cada token nuevo que introduces en la ventana de contexto no solo cuesta dinero, sino que **diluye la atención del modelo sobre el resto de tokens**. Esto es lo que nos referimos con *context rot*. Si saturamos la ventana, el modelo empieza a alucinar, a olvidar instrucciones anteriores o a ser incapaz de recuperar información específica (el famoso problema del "needle in a haystack"). Te dejo esta lectura sobre este tema.

Por eso, el diseño de agentes modernos no va de ver quién tiene el prompt más largo, sino de lo que Andrej Karpathy llamó *Context Engineering*. Es el arte delicado de **llenar la ventana de contexto con la información justa y necesaria para el siguiente paso, y nada más**. Si entiendes esto, entenderás por qué los patrones de diseño que vamos a ver a continuación se centran tanto en dividir, filtrar y gestionar la información.

¿Qué diferencia hay entre un Agente de IA y un Workflow agéntico?

Antes de ponernos a dibujar cajas y flechas, hay que hacer una distinción que a menudo se pasa por alto y que es la causa de la mayoría de fracasos en proyectos de IA. **¡¿Necesitas un Agente o necesitas un Workflow?!**

- Un **Workflow** o flujo de trabajo es un sistema donde los **caminos están predefinidos**. Tú, como ingeniero, defines los pasos: primero haces A, luego B, y si pasa X, entonces C. **La orquestación corre a cargo del código, no del modelo**. El **LLM se utiliza solo para ejecutar tareas específicas dentro de esos pasos** (resumir, clasificar, extraer), pero **no decide qué hacer a continuación**.
- Un **Agente**, en cambio, es un sistema donde **el modelo de IA tiene el control del flujo**. El **LLM decide**, basándose en la entrada y en su razonamiento, cuál es el siguiente paso. Puede decidir llamar a una herramienta, puede decidir pedir más información al usuario o puede decidir que ya ha terminado.

La regla de oro aquí es la navaja de Ockham, es decir, **busca siempre la solución más simple**. Si sabes de antemano los pasos necesarios para completar una tarea, usa un Workflow. Son predecibles, fáciles de probar y mucho más baratos. Solo deberías saltar a un diseño de Agente cuando te enfrentes a problemas ambiguos, dinámicos o donde sea imposible prever todas las ramificaciones posibles.

La incertidumbre es cara. **Un agente que decide su propio camino puede entrar en bucles**, no es determinista, puede tomar decisiones erróneas que se propaguen en cadena y es mucho más difícil de depurar. Un workflow es determinista. Dicho esto, vamos a ver los patrones que nos permiten construir ambos sistemas.

Catálogo de patrones de diseño para agentes de IA

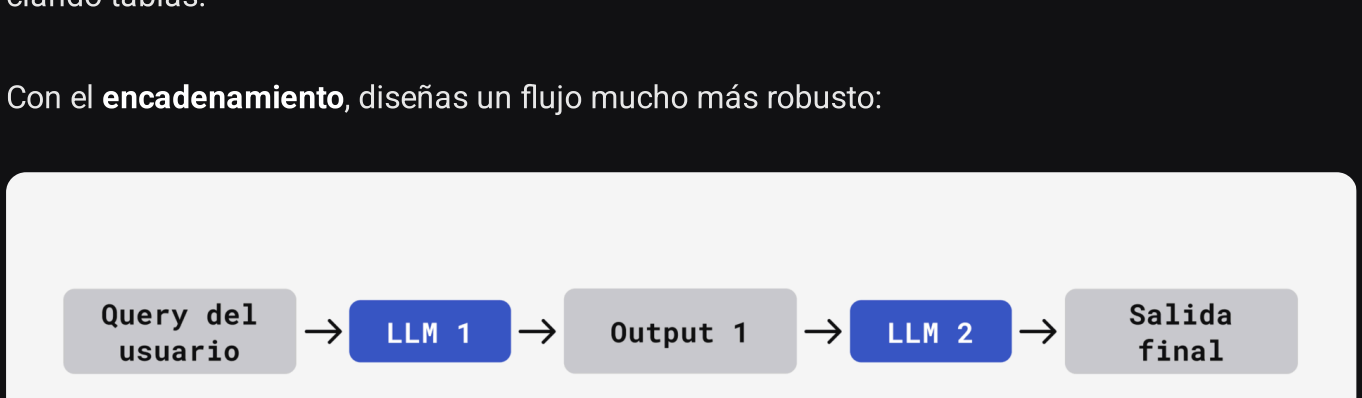
Vamos a desglosar los **patrones de diseño más utilizados hoy en día en la industria**, desde lo más básico hasta arquitecturas multi-agente complejas. Estos patrones no son mutuamente excluyentes, de hecho, los mejores sistemas suelen combinarlos. Al final del post en las referencias técnicas te dejo un enlace a un post del *Google Cloud Architecture Center* con más patrones.

1. Encadenamiento de Prompts — Prompt Chaining

Este es el patrón más básico de todos, pero no por ello menos útil. Consiste en **descomponer una tarea en una secuencia fija de pasos**, donde **la salida de una llamada al LLM se convierte en la entrada de la siguiente**.

Imagina que estás construyendo una herramienta interna para que un equipo de ventas pueda hacer preguntas a una base de datos en lenguaje natural, por ejemplo: ¿Cuántas zapatillas vendimos ayer? Si intentas meter el esquema completo de tu base de datos y la pregunta en un solo prompt gigante, el modelo suele alucinar inventándose columnas que no existen o mezclando tablas.

Con el **encadenamiento**, diseñas un flujo mucho más robusto:



Encadenamiento de prompts — Patrón de diseño para agentes de IA

Si intentas hacer todo de una vez (entender la pregunta, escribir SQL y asegurar que no borre la base de datos), el modelo falla. Con este encadenamiento de dos pasos, el flujo es mucho más robusto:

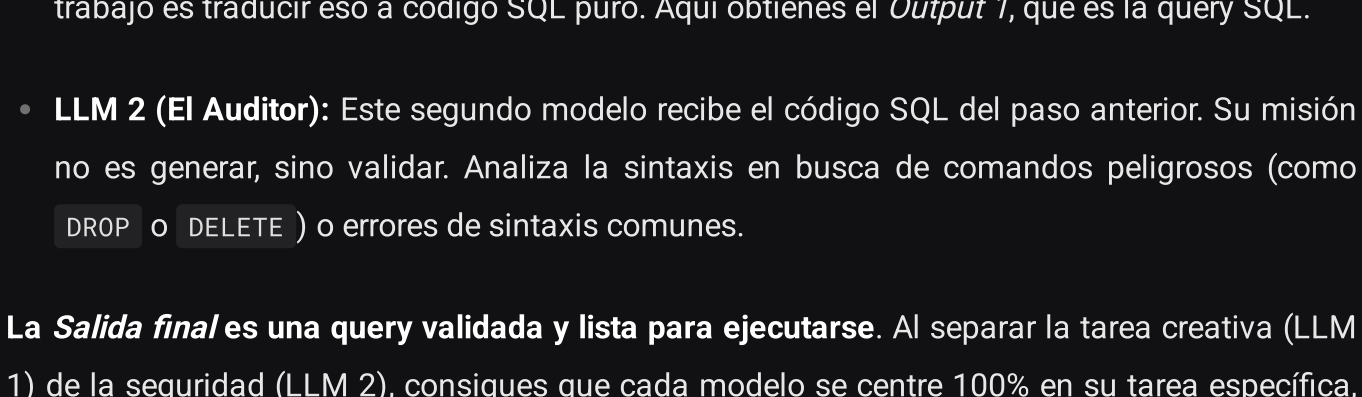
- **LLM 1 (El Generador):** Recibe tu esquema de base de datos y la *Query del usuario*. Su único trabajo es traducir eso a código SQL puro. Aquí obtienes el *Output 1*, que es la query SQL.
- **LLM 2 (El Auditor):** Este segundo modelo recibe el código SQL del paso anterior. Su misión no es generar, sino validar. Analiza la sintaxis en busca de comandos peligrosos (como `DROP` o `DELETE`) o errores de sintaxis comunes.

La *Salida final* es una query validada y lista para ejecutarse. Al separar la tarea creativa (LLM 1) de la seguridad (LLM 2), consigues que cada modelo se centre 100% en su tarea específica, reduciendo drásticamente los errores que tendrías si se lo pidieras todo junto.

2. Enrutamiento — Routing

El patrón de enrutamiento introduce un nivel de decisión, pero mantiene el control bastante atado. Aquí, **una llamada inicial al LLM actúa como un clasificador de tráfico**. Analiza la entrada del usuario y decide cuál es el mejor camino para resolverla.

Piensa en un sistema de atención al cliente. El usuario envía un mensaje. El **Router analiza la intención**.



Enrutamiento — Patrón de diseño para agentes de IA

Si la intención es Devolución, envía el flujo a un proceso especializado en política de devoluciones.

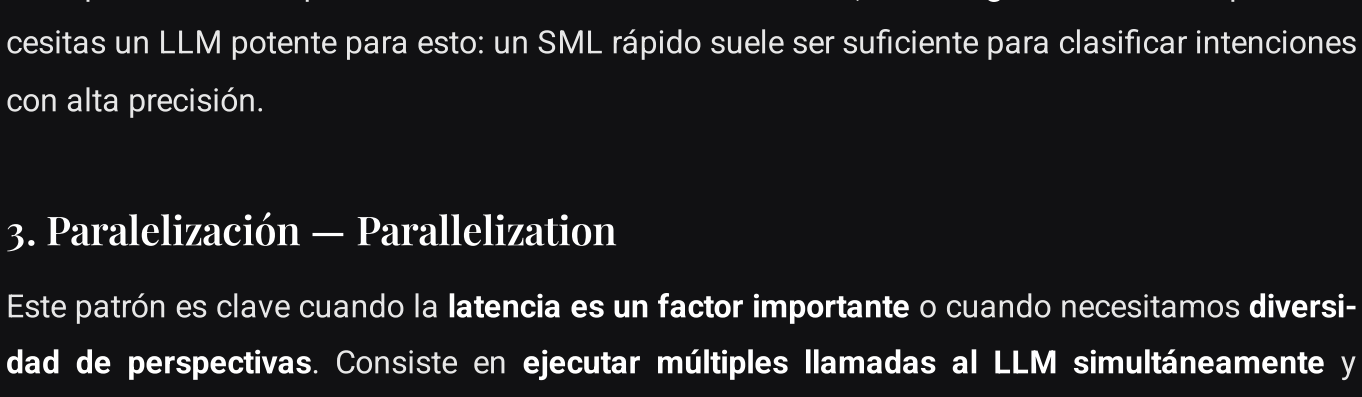
Si la intención es Soporte Técnico, lo envía a un agente que tiene acceso a manuales técnicos.

Lo potente de este patrón es la especialización. Al separar los flujos, puedes tener **prompts (instrucciones) muy específicos** y optimizados para cada caso posterior. El agente de devoluciones no necesita saber nada de soporte técnico, lo que limpia su contexto y reduce las alucinaciones.

Es importante notar que **el Router no suele resolver la duda, solo dirige**. A veces, ni siquiera necesitas un LLM potente para esto: un SML rápido suele ser suficiente para clasificar intenciones con alta precisión.

3. Paralelización — Parallelization

Este patrón es clave cuando la **latencia es un factor importante** o cuando necesitamos **diversidad de perspectivas**. Consiste en **ejecutar múltiples llamadas al LLM simultáneamente** y luego agregar los resultados.



Paralelización — Patrón de diseño para agentes de IA

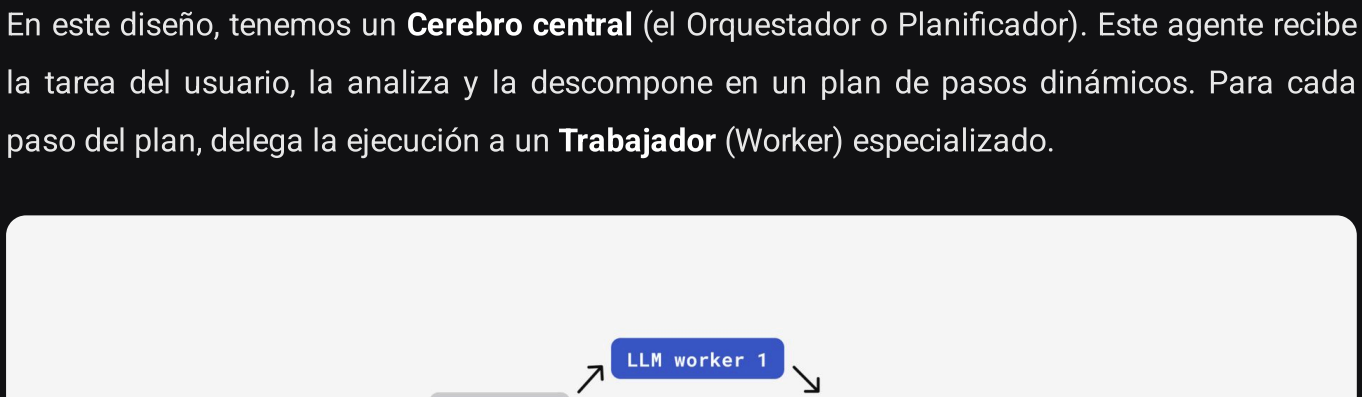
Hay dos variantes principales de este patrón:

- La primera es el **Sectioning**. Imagina que tienes que analizar un documento legal de 500 páginas. En lugar de procesarlo secuencialmente, lo divides en 10 secciones y lanzas 10 procesos en paralelo para que extraigan las cláusulas de riesgo de cada sección. Luego, un último paso **agregador** junta todos los hallazgos. Es básicamente un **Map-Reduce** aplicado a LLMs.
- La segunda variante es el **Voting** o generación de perspectivas. Le pides a tres modelos (o al mismo modelo tres veces) que generen una solución para un problema complejo. Luego, un cuarto modelo actúa como juez, revisa las tres propuestas y elige la mejor o sintetiza una respuesta final combinando lo mejor de cada una. Esto es muy útil para tareas creativas o para reducir el riesgo de errores en tareas lógicas. Andrej Karpathy tiene un proyecto muy interesante que usa este patrón llamado LLM Council. (Cuando digo 3 modelos pueden ser los que quieras depende de la complejidad del problema).

4. Orquestador y Trabajadores — Orchestrator-Workers

Aquí es donde empezamos a entrar en el terreno de los **agentes más complejos y donde se suele aplicar el patrón de planificación**. Este patrón es ideal para tareas que son demasiado complejas para resolverse en un solo paso o con una secuencia fija, y donde **no sabemos a priori cuántos pasos serán necesarios**.

En este diseño, tenemos un **Cerebro central** (el Orquestador o Planificador). Este agente recibe la tarea del usuario, la analiza y la descompone en un plan de pasos dinámicos. Para cada paso del plan, delega la ejecución a un **Trabajador** (Worker) especializado.



Orquestador y trabajadores — Patrón de diseño para agentes de IA

Por ejemplo, si le pides: Crea una web para Coca Cola.

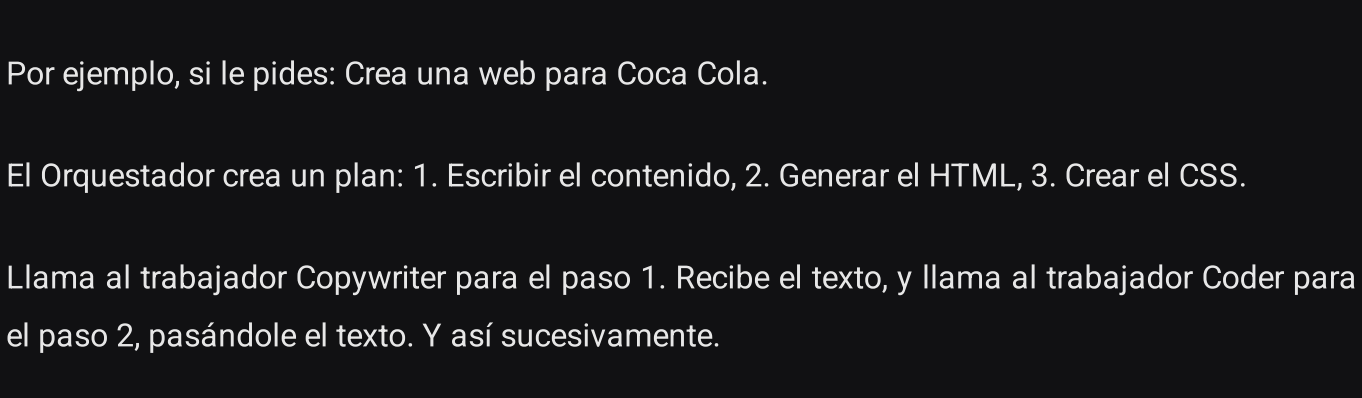
El Orquestador crea un plan: 1. Escribir el contenido, 2. Generar el HTML, 3. Crear el CSS.

Llama al trabajador Copywriter para el paso 1. Recibe el texto, y llama al trabajador Coder para el paso 2, pasándole el texto. Y así sucesivamente.

La clave aquí es que **el Orquestador mantiene el estado global del proyecto y ajusta el plan según los resultados**. Si el Coder devuelve un error, el Orquestador lo ve y puede decidir añadir un paso extra de Debugging antes de seguir. Este patrón centraliza la inteligencia y el control, lo que facilita el seguimiento del progreso, pero convierte al Orquestador en un cuello de botella si la tarea se vuelve extremadamente grande.

5. Reflexión y Crítica — Evaluator-Optimizer

Este es, posiblemente, el patrón que más mejora la calidad del resultado final con una inversión de código relativamente baja. Se basa en imitar el proceso humano de **escribir y revisar**.



Evaluator-Optimizer — Patrón de diseño para agentes de IA

La mayoría de los LLMs funcionan mucho mejor si les das una segunda oportunidad para corregirse. **En el patrón de Reflexión, tenemos un bucle**.

Primero, un **agente Generador** produce una respuesta inicial o un borrador.

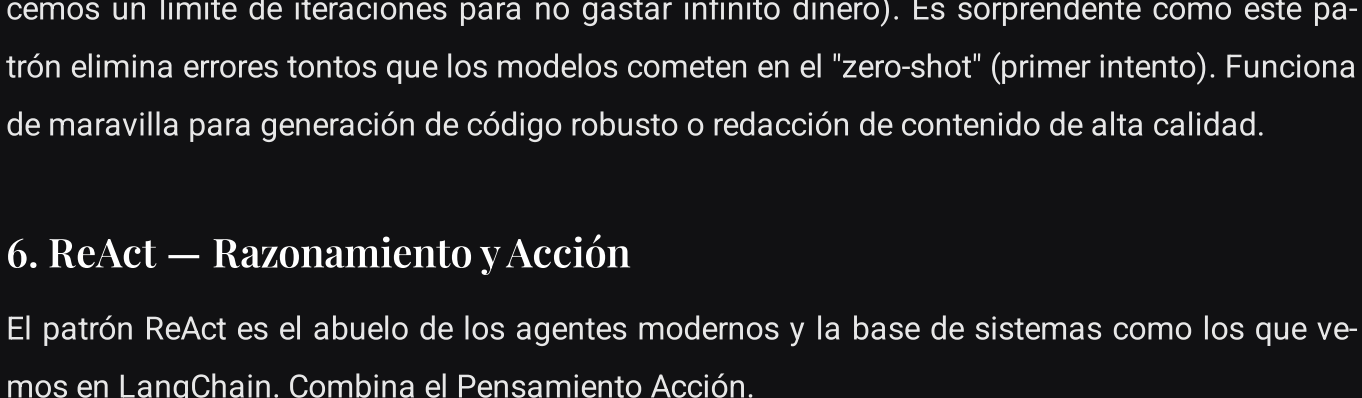
A continuación, un **LLM que actúa de crítico** (que puede ser el mismo modelo con otro prompt) analiza ese borrador. Busca errores lógicos, falta de claridad, problemas de estilo o bugs si es código.

El **Crítico devuelve una lista de mejoras**. El **Generador toma esas críticas y reescribe la respuesta**.

Este ciclo puede repetirse una vez o varias veces hasta que el Crítico dé su visto bueno (o alcancemos un límite de iteraciones para no gastar infinito dinero). Es sorprendente cómo este patrón elimina errores tontos que los modelos cometen en el "zero-shot" (primer intento). Funciona de maravilla para generación de código robusto o redacción de contenido de alta calidad.

6. ReAct — Razonamiento y Acción

El patrón ReAct es el abuelo de los agentes modernos y la base de sistemas como los que vemos en LangChain. Combina el Pensamiento Acción.



Razonamiento y Acción — Patrón de diseño para agentes de IA

En un bucle ReAct, el modelo sigue una secuencia como la siguiente:

1. **Pensamiento**: El modelo verbaliza qué es lo que necesita hacer. "El usuario me ha pedido el tiempo en Madrid. No tengo ese dato, necesito buscarlo".
2. **Acción**: El modelo selecciona una herramienta y genera los parámetros para usarla. "Ejecutar: herramienta_tiempo('Madrid')".
3. **Observación**: El sistema ejecuta la herramienta y le devuelve el resultado al modelo. "Resultado: 25 grados, despejado".
4. **Pensamiento**: El modelo analiza el resultado. "Ya tengo el dato. Puedo responder".
5. **Respuesta final**: "Hace 25 grados en Madrid".

Este patrón es muy potente porque permite al agente interactuar con el mundo exterior (APIs, bases de datos) de forma iterativa. Sin embargo, tiene un problema y es que es muy verboso. **Consume muchos tokens explicando sus pensamientos internos**, lo que puede aumentar la latencia y el coste. Aún así, es fundamental para tareas que requieren exploración y uso de herramientas.

7. Enjambre — Swarm

Swarm es el patrón más experimental y caótico. A diferencia de la jerarquía o el orquestador, **aquí no hay un jefe central** que mande sobre todos. Es una **colaboración todos con todos** o coordinada por un facilitador ligero.

Swarm — Patrón de diseño para agentes de IA

En un enjambre, diferentes agentes especializados (con diferentes roles o herramientas) comparten un espacio de trabajo o un canal de comunicación. **Un agente lanza un problema al grupo**. Los agentes pueden coger la tarea, pasarla a otro agente que crean más capacitado, o colaborar iterativamente.

Es muy útil para procesos creativos, **como un brainstorming donde tienes un agente Creativo, un agente Pragmático y un agente Financiero debatiendo sobre una idea**. Se pasan el turno y van refinando la solución. Sin embargo, controlar esto es muy difícil. Sin una condición de salida clara y una lógica de turnos sólida, pueden entrar en bucles de discusión infinita sin llegar a nada productivo.

Si quieres profundizar más sobre este patrón de diseño de agentes te recomiendo este artículo.

Estrategias Avanzadas de Gestión de Contexto

Ahora que conocemos los patrones, volvamos al problema inicial: la **eficiencia y el contexto**. ¿Cómo aplicamos estos patrones sin arruinarnos en tokens? Aquí es donde entran las estrategias que diferencian una poc de un producto serio.

Dale un ordenador al agente — Computer Use

Una tendencia clarísima para 2026 es dejar de tratar al agente como un chatbot y tratarlo como un operador de ordenador. En lugar de pedirle al agente que simule que escribe código o que simule que lee un archivo, **le damos acceso real a una terminal y a un sistema de archivos**.

Si el agente tiene acceso al sistema de archivos (`filesystem`), no necesitas cargar todo el proyecto en su contexto. El agente puede usar herramientas como `grep` o `ls` para explorar, y `read_file` para leer solo el fichero que necesita en ese momento. Una de las herramientas con más poder en este aspecto es MCP, una solución que debes dominar si quieres empezar a diseñar agentes de IA serios.

Revelación Progresiva — Progressive Disclosure

Relacionado con lo anterior, está la idea de no abrumar al agente con todas las herramientas disponibles desde el principio. Si tienes 50 herramientas o Agent Skills definidas, meter las definiciones de todas en el System Prompt consumirá miles de tokens inútilmente.

La estrategia aquí es la revelación progresiva. **Le das al agente una lista muy breve y resumida de las categorías de herramientas que tiene**. Si el agente decide que necesita una herramienta de Base de Datos por ejemplo, entonces y solo entonces, le cargas las definiciones detalladas de las tablas y las queries SQL disponibles.

Si quieres profundizar sobre la carga dinámica y uso efectivo de las ventanas de contexto puedes echar un ojo al post sobre Agent Skills.

Conclusión: El futuro es híbrido

Si te das cuenta, la tendencia no es hacia un Agente único que lo hace todo con un prompt mágico. **La tendencia es hacia sistemas modulares**, donde combinamos flujos deterministas para lo que sabemos hacer, y **pequeños agentes especializados** para las partes donde necesitamos razonamiento adaptativo.

Para 2026, los sistemas más robustos serán aquellos que sepan **mover la información eficientemente fuera del contexto del modelo** (al disco, a bases de datos vectoriales) y **recuperarla solo cuando sea crítico**.

Como te prometí, no he puesto código aquí para no saturar la lectura, pero si quieres ver implementaciones reales en Python de estos patrones (Routing, Parallelization, Orchestrator, etc.), te recomiendo encarecidamente que revises las referencias técnicas que te dejo abajo. Ahí es donde está la verdad del código.

Referencias técnicas

Aquí tienes los recursos fundamentales para profundizar en la implementación de estos patrones.

1. Building Effective Agents (Anthropic) - La guía oficial de Anthropic sobre patrones de diseño de agentes (Workflows vs Agents), con diagramas y mejores prácticas para gestionar el contexto.
2. Agent Design Patterns (Lance Martin) - Análisis profundo sobre la ingeniería de contexto, el uso de sistemas de archivos ("giving agents a computer") y la revelación progresiva de herramientas.
3. Zero to One: Learning Agentic Patterns (Philipp Schmid) - Una guía práctica excelente que desglosa patrones como Prompt Chaining, Routing, Parallelization y Evaluator-Optimizer con un enfoque de "empezar simple".
4. Google Cloud Architecture Center: Design Patterns for AI Agents - Documentación técnica de Google que clasifica los patrones según los requisitos de carga de trabajo (deterministas vs dinámicos) y ofrece arquitecturas de referencia.
5. Model Context Protocol (MCP) - El estándar para conectar modelos de IA con fuentes de datos y herramientas externas, clave para la estrategia de revelación progresiva y uso de herramientas.
6. Context Engineering (Andrej Karpathy) - Reflexiones fundamentales sobre cómo gestionar la ventana de contexto como un recurso limitado y la necesidad de optimizar qué información se presenta al modelo.