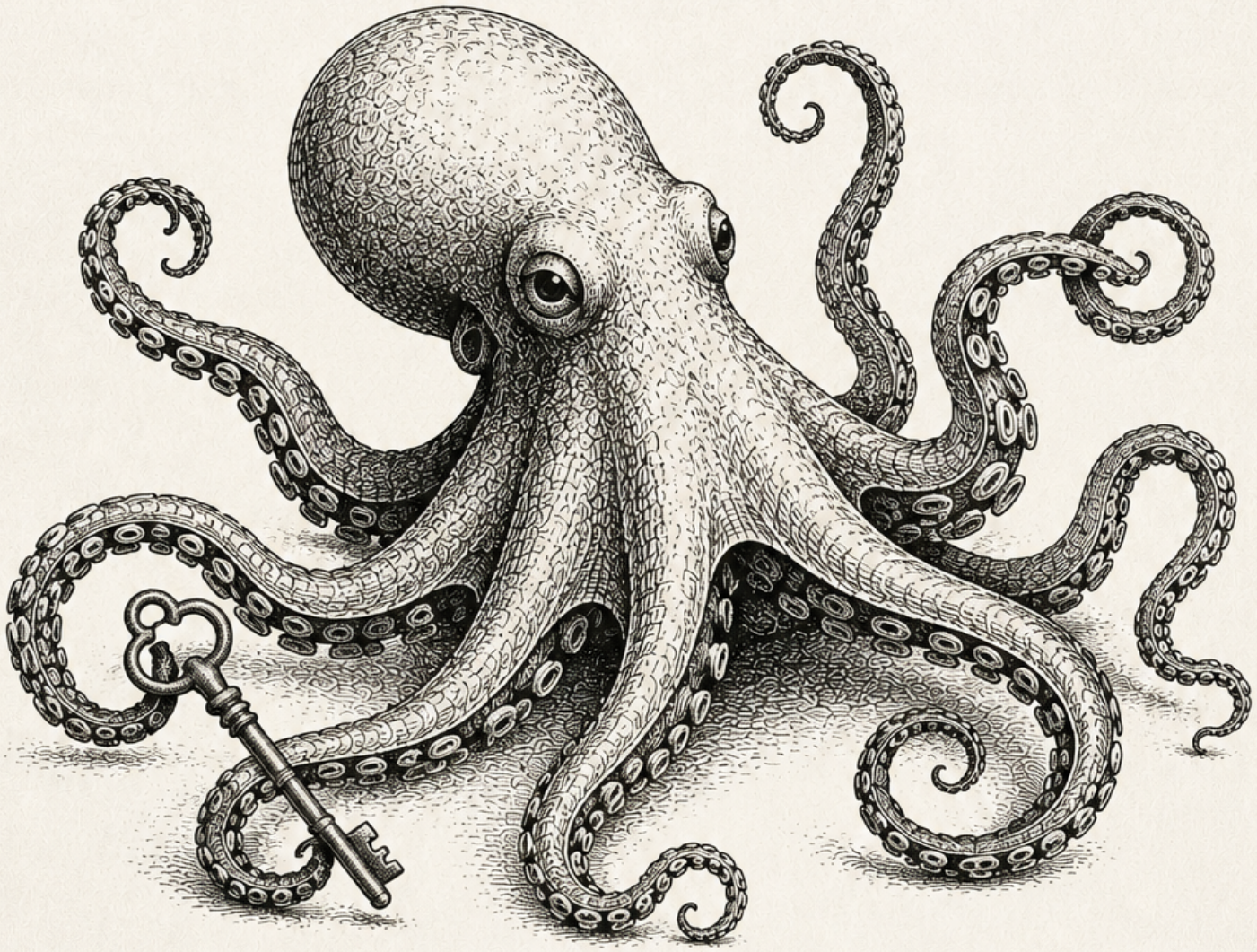
Laravel,
thinking fast



Julian Beaujardin

Table of Contents

Prefacio **10**

De qué trata este libro 10

Para quién es este libro 10

Cómo está estructurado 11

Convenciones 11

Los cuatro principios 12

Qué pasa cuando los ignoras 12

Cómo aprendí esto 13

Qué significa «thinking fast» 14

Capitulo 1: Tu primer endpoint de IA **14**

Empecemos por el sitio equivocado, brevemente 14

La forma que sobrevive 15

Un driver, con todas las rarezas del proveedor 16

El gestor, donde viven las decisiones 18

Falla en voz alta si falta la credencial 19

Las credenciales son del cliente, no de la aplicación 20

Reintenta lo que merece reintento 20

Cómo se usa desde la aplicación 21

Qué has comprado 22

Qué no resuelve todavía 22

Principios 22

Capitulo 2: Obtener una respuesta que puedas usar **23**

Pide JSON, y hazlo en serio 23

El bloque de código que añade igualmente 24

Decodifica o lanza, en la frontera 25
JSON válido que te arruina la tarde 25
Un esquema, leído por todos 26
Las rutas obligatorias son un contrato: manténlas cortas 27
Aleja al modelo de lo que no puede saber 28
Mantén el prompt y el validador en la misma habitación 29
Cuando la validación falla 30
Principios 31

Capitulo 3: Los prompts son codigo 31
Dónde vive un prompt 31
Compón, no bifurques 32
Separa el texto del montaje 33
Ata la instrucción al valor del que habla 34
Escribe reglas que un modelo pueda seguir 35
Enséñale a negarse, y dale forma a la negativa 36
Los prompts que no escribiste 37
Pon delante la parte estable 37
Cambiar un prompt es un despliegue 38
Probar prosa 38
Lo que sostiene este capítulo 39

Capitulo 4: Lo que te cuesta un token 39
La factura tiene más líneas de las que crees 39
El tope es una decisión del servidor, tomada por propósito 40
El tope que también se come el razonamiento 41
Dos límites, no uno 42
Incrementa primero, pregunta después 42
Ata el techo a lo que paga 43
Di que no como es debido 44

Lee el bloque de uso 44
Demuestra que el cacheo funciona de verdad 45
El dial entre coste y latencia 45
Multiplícalo todo por las iteraciones 46
Lo que sostiene este capítulo 46

Capitulo 5: Nunca dejes que el navegador hable con el modelo 47
Lo que estás entregando 47
El fallo que nos lo enseñó 48
Pon la frontera donde están los hechos 48
Las credenciales van en cabeceras, nunca en URLs 49
Una guarda, una respuesta 50
El navegador envía datos, nunca instrucciones 51
Resuelve en el servidor la credencial que paga 51
Lo que el navegador sí puede hacer 52
Lo que sostiene este capítulo 53

Capitulo 6: La lentitud es lo normal 53
La petición web es el sitio equivocado 53
Mueve el turno a una cola 54
Tres tiempos de espera, y su orden importa 55
Publica sobre la marcha 56
Guarda un punto de control, y luego recupera 56
Deja parar al usuario 57
Contarle al usuario qué está pasando 58
Lo que sostiene este capítulo 59

Capitulo 7: Herramientas, darle manos al modelo 59
La descripción es la API 60
Tipa la entrada, y valídala igualmente 61

Los argumentos planos ganan a los anidados 62
Da forma a la carga en la frontera 62
Normaliza la salida, nombra las claves a la defensiva 63
El resultado también es un prompt 64
Ordena el conjunto de herramientas 65
Lo que sostiene este capítulo 66

Capitulo 8: El bucle del agente 66
La forma 66
stop_reason es tu control de flujo 67
pause_turn, y por qué importa el orden 68
El truncamiento no puede mostrarse como una respuesta 69
Negarse no es un error 69
El historial es el estado 70
Lee los bloques por tipo 70
Dos topes, no uno 71
Qué pasa después del bucle 71
Lo que sostiene este capítulo 72

Capitulo 9: Los resultados de las herramientas no son respuestas 73
Ramifica por claves, no por nombres de herramienta 73
Los mensajes de estado no son mensajes de chat 74
Dale identidad a cada tarjeta 74
Una tarjeta que se actualiza, no cinco que se apilan 74
Representar texto del modelo sin abrir una inyección 75
Deja que el resultado mueva la aplicación, no sólo la vista 77
Lo que sostiene este capítulo 78

Capitulo 10: Efectos secundarios que no puedes deshacer 78
La instrucción no es la guarda 79

Persiste primero, y que la base de datos sea la guarda 79
Los tokens de un solo uso deben sobrevivir a un rechazo 80
Nunca mantengas una transacción a través de una llamada de red 81
Revierte antes de reportar 82
Confirma sólo cuando es real 82
La idempotencia es una decisión por herramienta 83
Dile al usuario qué ha pasado 83
Lo que sostiene este capítulo 84

Capitulo 11: El modelo es un llamante en el que no se confia 84
Cuenta los intentos contra la identidad, no contra el intento 85
Dos ventanas, o has moderado el abuso en vez de detenerlo 85
No construyas un oráculo 86
Las vinculaciones hechas al emitir son las autoritativas 87
Da por hecho que un identificador se ha filtrado 87
Algunas herramientas deben existir y no ofrecerse nunca 88
Rechaza de forma útil 88
Lo que sostiene este capítulo 89

Capitulo 12: Probar algo que nunca responde dos veces igual 90
Simula en la frontera que es tuya 90
Prueba el bucle, no el modelo 90
Las pruebas de herramientas son pruebas de integración 91
Prueba las compensaciones, no el camino feliz 91
Las pruebas de contrato ganan a las de salida 92
Registra casos reales como fixtures 92
Qué no probar 93
Lo que sostiene este capítulo 93

Capitulo 13: Cuando el modelo se equivoca 94

El fallo: publicar la posición cero 94
Cobertura, no ingenio 94
El experimento que se borró 95
Los empates van para el proveedor, y sin coincidencias no se cambia nada 96
Determinista significa testable 96
Dónde más aplica esta forma 96
Lo que sostiene este capítulo 97

Capitulo 14: Observabilidad para lo que no puedes reproducir 97
Nombres y tamaños, nunca entradas 97
Una línea estructurada por ida y vuelta 98
Mide el salto sobre el que vas a discutir 99
Registra la pregunta que te van a hacer 100
Lo que sostiene este capítulo 101

Capitulo 15: Fallar es parte del producto 101
Nombra tus dependencias 101
Nombra lo que pierde el usuario 102
Niégate a empezar lo que no puedes terminar 103
Deja de preguntarle a un servicio caído 104
Di qué va mal sin nombrar al proveedor 104
Lo que sostiene este capítulo 105

Capitulo 16: Contenido largo 105
Fragmenta por estructura, no por caracteres 105
Deja sitio para las instrucciones 106
Falla del todo antes que reensamblar una mentira 106
Da nombres, no códigos 107
Cuando el navegador tiene que llamar directamente 108
Limpia lo que queda rancio después 108

Lo que sostiene este capítulo 108

Capitulo 17: Donde encaja la IA **109**

Qué compra un servicio hoja 109

Qué cuesta 110

El salto que nadie midió 110

Alquila el modelo, posee la frontera 110

Prefiere una estrella a una malla 111

Lo que sostiene este capítulo 111

Capitulo 18: Publicarlo **111**

Fija la versión del modelo 112

Prueba el camino, no el modelo 112

Alerta sobre síntomas, no sobre el modelo 112

Guarda una salida humana 113

La prueba de los seis meses 113

Lo que sostiene este capítulo 113

Epilogo: Conocer lo bueno **114**

Plausible es el modo de fallo 114

La habilidad escasa es discriminar, no producir 114

De qué trataban en realidad los dieciocho capítulos 115

La parte que no voy a fingir 115

Lo que pienso de verdad 116

Apendices **116**

Apendice A: Lista de comprobacion previa al lanzamiento 116

Apendice B: Lista de comprobacion para revisar un prompt 117

Apendice C: Lista de comprobacion de coste 118

Apendice D: La tabla de stop_reason 118

Apendice E: Adonde ir despues 118

Prefacio

Toda función de IA queda bien en una demo.

Conectas un modelo a un controlador, escribes una frase y vuelve algo notable. La sala se queda en silencio. Alguien dice que hay que publicarlo ya.

Y funciona de inmediato. Ese es justamente el problema.

Luego llega a producción y descubres que la parte notable nunca fue la parte difícil. La parte difícil es la petición que tardó noventa segundos. La que devolvió prosa cuando pediste JSON. Aquella en la que un usuario pulsó el botón once veces y cada pulsación te costó dinero. Aquella en la que el modelo, cuando se lo pidieron con educación, creó dos veces el mismo recurso de pago.

Nada de eso es un problema de IA. Son los problemas de ingeniería de siempre, los que ya sabes resolver —tiempos de espera, idempotencia, límites de uso, fronteras de confianza, reintentos—, disfrazados.

Ese es todo el argumento de este libro.

El modelo es la parte menos interesante de tu función de IA.

De qué trata este libro

Este libro trata de construir funciones de IA en Laravel que sobrevivan a usuarios reales: funciones que cuestan lo que esperabas, que fallan como tenías previsto y que se pueden entregar a otra persona sin una sesión de espiritismo.

Usa como ejemplo un sistema en producción: una plataforma donde un agente conversacional recoge lo que alguien quiere, lo comprueba, verifica su correo y aprovisiona un sitio web real, publicado y facturable mientras el usuario mira. Ese agente llama a herramientas. Esas herramientas tienen efectos que no se pueden deshacer. Se ha equivocado, ha ido lento y lo han convencido de hacer cosas. Todo lo que hay aquí se aprendió arreglando eso.

No encontrarás folclore sobre *prompts*, ni una comparativa de seis modelos. Encontrarás el código que rodea al modelo: el gestor que aísla al proveedor, el trabajo en cola que sobrevive a un turno demasiado largo, la herramienta que se niega a crear dos veces lo mismo, y el registro que te dice qué pasó cuando no puedes reproducirlo.

Para quién es este libro

Para desarrolladores de Laravel que se manejan con HTTP, colas y pruebas, y que están a punto de poner un modelo delante de sus usuarios —o que ya lo hicieron y ahora conviven con las consecuencias.

No necesitas saber nada de aprendizaje automático. Nada de este libro lo requiere. Si sabes escribir un FormRequest, despachar un job y simular un cliente HTTP en una prueba, tienes todas las habilidades que este libro pide.

Deberías desconfiar de las funciones de IA. Eso es un requisito, no un defecto.

Si has publicado un chat y luego has mirado la factura del proveedor, este libro es para ti. Si te están pidiendo añadir «IA» a un producto y te gustaría que el resultado siguiera siendo mantenible el trimestre que viene, este libro es para ti. Si tu función de IA funciona y no sabes explicar por qué, este libro es muy especialmente para ti.

Cómo está estructurado

El libro avanza hacia fuera desde una sola llamada.

Los primeros capítulos construyen los cimientos aburridos: un endpoint, un proveedor, salida estructurada en la que puedas confiar, *prompts* que viven en el control de versiones y límites sobre lo que todo eso puede costarte. Los capítulos centrales le dan manos al modelo —herramientas, bucles de agente, efectos secundarios y las fronteras de confianza que impiden que un modelo complaciente salga caro. Los últimos capítulos tratan de todo lo que ocurre después de que funcione: probar algo que nunca responde dos veces igual, el rescate determinista cuando el modelo se equivoca, observabilidad, degradación y publicación.

Cada capítulo se apoya en código que hoy está en producción. Cuando un patrón existe porque algo se rompió, el libro dice qué se rompió.

Convenciones

El código sigue las mismas convenciones que el resto de mi trabajo y que el libro anterior: tipado estricto, tipos de retorno explícitos, FormRequests para la validación, *Response Wrappers* para el JSON, DTOs para datos tipados y el patrón Manager-Facade-Driver en los límites de integración. Los ejemplos apuntan a Laravel 13 y PHP 8.4.

Sobre el vocabulario: «proveedor» es quien ejecuta el modelo, la empresa a la que envías una petición y de la que recibes *tokens*. «Driver» es tu clase que habla con un proveedor. «Herramienta» es una función que has expuesto al modelo, y «turno» es todo lo que ocurre entre que un usuario pulsa enter y la función vuelve a quedarse callada.

Algunos términos se quedan en inglés a propósito, porque es como se dicen en el oficio: *prompt*, *token*, *endpoint*, *job*. Traducirlos haría el texto más difícil de leer, no más fácil.

Las APIs de los proveedores cambian más rápido que las de un framework. Cuando este libro muestra la forma de una petición, trata la forma como la lección y los nombres de los campos como una fotografía: consulta la documentación vigente antes de copiarla.

Los cuatro principios

Todo lo que hay aquí se deriva de cuatro ideas. No son opiniones sobre arquitectura. Son lo que queda después de las caídas.

1. Un modelo es una llamada de red poco fiable con disfraz.

Es un endpoint HTTP de terceros, con latencia alta, salida variable, algún que otro 429 y una factura adjunta. Todos tus instintos sobre integrar una pasarela de pago inestable son correctos aquí. Ponle un tiempo de espera. Ponle un tope. Reintenta los fallos que merecen reintento y ninguno más. Da por hecho que falla, y decide ahora qué ve el usuario cuando ocurra.

2. Los *prompts* son código.

Un *prompt* determina el comportamiento. Eso lo convierte en código, y el código va en el control de versiones, en revisión y en un despliegue —no en una fila de base de datos que alguien edita un viernes, ni en un fichero de configuración que nadie revisa. Cuando cambia un *prompt*, cambia el comportamiento, y deberías poder señalar el commit.

3. Pon tope a todo: tokens, reloj, iteraciones, dinero.

Una función de IA sin topes es una factura sin tope. Todo bucle necesita un número máximo de vueltas. Todo job necesita un plazo más corto que su propio tiempo de espera. Todo cliente necesita un techo. Toda respuesta necesita un límite de tokens elegido por ti en el servidor, no insinuado por el cliente. No notarás que falta el tope hasta el mes en que lo notes.

4. El modelo es un llamante en el que no se confía.

Se le puede convencer. Se le convencerá. Cualquier cosa que impongas pidiéndoselo amablemente al modelo no está impuesta. La verificación, la propiedad, las cuotas y los permisos viven en tu base de datos y en tu middleware, detrás de las herramientas —nunca en las instrucciones.

Qué pasa cuando los ignoras

Puedo describir el fallo con precisión, porque he visto cada una de sus partes.

El SDK del proveedor va directo a un controlador, porque es lo que hacía el tutorial. Seis semanas después el proveedor cambia el nombre de un campo y el fallo aparece en cuatro funciones sin relación entre sí, porque no hay un único sitio donde se describa al proveedor.

Nadie pone tope a los tokens de salida, así que una respuesta se corta a media frase. No hay ninguna comprobación de por qué se detuvo, de modo que la media frase se muestra como si fuera una respuesta completa, y un usuario lee algo seguro de sí mismo, equivocado y sin terminar.

La petición se ejecuta en el proceso web, porque en desarrollo funcionaba. En producción tarda noventa segundos, retiene un worker de PHP todo ese rato y muere en el balanceador con un 504 —después de haber pagado el modelo.

El navegador llama directamente al servicio de IA, porque así había un salto menos. Ahora la credencial está en el cliente, el endpoint acepta cualquier cosa y tu gasto con el proveedor sólo está limitado por la paciencia de quien lo encuentre.

Y en algún punto de todo esto, un modelo al que se le ha dicho «crea un solo sitio por usuario» crea dos —porque se lo pidieron dos veces, con educación, y no había nada más que la instrucción para impedirlo.

Ninguno de esos es un fallo exótico. Todos son un martes cualquiera.

Cómo aprendí esto

La primera versión de nuestro agente funcionó a la primera, y ese fue el problema.

Era un chat. Le contabas cuál era tu negocio y te construía una web. En la demo resultaba genuinamente encantador. Hacía buenas preguntas, tomaba decisiones razonables y, más o menos un minuto después, había un sitio real en una URL real. Se lo enseñamos a todo el mundo.

Lo primero que se rompió fue el tiempo. Un turno en el que el modelo usaba varias herramientas pasaba holgadamente del minuto, y todo aquello corría dentro de la petición HTTP. Los usuarios miraban un indicador de carga hasta que algo más arriba se rendía, y entonces no veían nada: ni error, ni respuesta parcial, ni sitio, pese a que ya habíamos pagado cada token. El trabajo estaba hecho. Sencillamente no teníamos dónde ponerlo.

Así que lo movimos a la cola, y se rompió lo segundo: el job tenía un tiempo de espera y un turno largo lo alcanzaba. El job moría a mitad de turno sin haber escrito nada, mientras el widget consultaba educadamente una clave de caché que nunca iba a aparecer. El usuario se quedaba delante de un chat que simplemente había dejado de hablar.

Luego un turno volvió truncado. Habíamos puesto tope a los tokens de salida, con buen criterio, pero nunca mirábamos *por qué* se había detenido el modelo. Una respuesta cortada caía por la misma rama que una terminada y se mostraba como una respuesta. Se leía bien. Estaba mal.

Después un usuario pulsó el botón dos veces.

La herramienta que crea un sitio tenía escrito, en su propia descripción, que sólo debía hacerlo una vez. Y esa instrucción se cumplió a la perfección, hasta que llegaron dos peticiones lo bastante seguidas como para que «una vez» fuera ambiguo. Dos aprovisionamientos para una sola persona. El arreglo no fue una instrucción mejor. El arreglo fue un índice único.

Ninguno de estos era un problema del modelo. El modelo estaba bien. Cada uno de los fallos estaba en el código que lo rodeaba, y cada arreglo fue algo que yo ya sabía hacer —sólo que nunca se me había ocurrido hacerlo *aquí*, porque esta parte venía disfrazada.

Por eso los capítulos de este libro son como son. No van de escribir *prompts*. Van de colas, plazos, índices únicos, fronteras de confianza y registros estructurados, aplicados a un componente que resulta ser no determinista.

Qué significa «thinking fast»

El título no va de latencia, ni de lo rápido que responde el modelo.

Significa lo mismo que significaba «shipping fast» en el libro anterior: tomar una decisión una vez, al nivel correcto, para no tener que volver a tomarla. Dónde vive el proveedor. Dónde viven los *prompts*. Cuánto puede costar un turno. Qué pasa cuando el modelo se equivoca. Decide eso una vez, aplícalo en todas partes, y añadir tu quinta función de IA costará una fracción de lo que costó la primera.

La alternativa es una base de código donde cada función de IA es un caso especial con su propia política de reintentos, su propia idea de lo que es un tiempo de espera, su propio manejo de credenciales y su propio modo de fallar. Eso no es rápido. Eso son cuatro funciones y cuatro caídas.

Lo aburrido escala. Lo ingenioso se descompone. Era cierto antes de que hubiera un modelo en el bucle, y el modelo no lo ha derogado.

Capitulo 1: Tu primer endpoint de IA

Lo más valioso de este capítulo es lo poco llamativo que resulta.

Vamos a construir un endpoint que envía un *prompt* a un proveedor y devuelve lo que vuelve. No tiene ninguna astucia. Lo que tiene, en cambio, es una frontera: un único sitio donde se describe al proveedor, un único contrato con el que habla el resto de la aplicación, y un único conjunto de decisiones sobre credenciales, tiempos de espera y reintentos que toda función de IA futura hereda gratis.

Equivócate aquí y lo pagarás en cada capítulo que sigue. Acierta y la mayor parte de los capítulos restantes se vuelven pequeños.

Empecemos por el sitio equivocado, brevemente

Esto es lo que casi todo el mundo escribe primero, y merece la pena mirarlo con honestidad, porque no es una tontería: simplemente está sin terminar.

```
public function generate(Request $request): JsonResponse
{
    $response = Http::withToken(config('services.openai.token'))
        ->post('https://api.provider.example/v1/chat/completions', [
            'model' => 'some-model',
            'messages' => [['role' => 'user', 'content' =>
$request->input('prompt')]],
        ]);

    return response()->json([
        'content' => $response->json('choices.0.message.content'),
    ]);
}
```

Esto funciona. En un buen día funciona siempre.

Lo que le falta es cada decisión que todavía no has tomado. No hay tiempo de espera, así que un proveedor lento retiene este worker hasta que PHP se rinde. No hay reintento, así que un solo 503 es un fallo visible para el usuario. No se comprueba que la credencial exista, así que una vacía se envía como un *bearer* vacío y vuelve como un error de autenticación que te culpa de algo que no hiciste. La URL del proveedor y la forma de su carga útil están soldadas a un controlador, así que el día que añadas un segundo proveedor —o el día que este cambie— estarás editando controladores.

Y, sobre todo: no hay dónde poner la siguiente decisión. Cuando aprendas algo sobre cómo llamar a este proveedor, esta forma no te ofrece ningún sitio donde anotarlo.

La forma que sobrevive

El patrón es el que Laravel ya usa para cada subsistema intercambiable que posee: caché, colas, sistema de ficheros, correo. Un contrato que dice cuál es la capacidad, un driver por proveedor que la implementa y un gestor que construye drivers a partir de la configuración.

Tres ficheros.

```
app/Services/Ai/AiContract.php            qué puede hacer un proveedor de IA
app/Services/Ai/AiManager.php             construye y configura drivers
app/Http/Integrations/Ai/OpenaiAPI.php    un proveedor
```

El contrato va primero, y debe describir las necesidades *de tu aplicación*, no la superficie de la API del proveedor.

```php
interface AiContract
{
    /**
     * Generate structured site content as a decoded array.
     *
     * @return array<string, mixed>
     */
    public function getSiteContent(string $model, string $prompt): array;

    /**
     * Generate a single block of text, capped at maxTokens.
     */
    public function getContent(string $model, string $prompt, int $maxTokens):
string;
}
```

Fíjate en lo que no está ahí. No hay array `messages`, ni `response_format`, ni `temperature`. Ese es el vocabulario de un proveedor concreto. Si se filtra al contrato, el contrato no es una abstracción: es un cambio de nombre.

La prueba es sencilla: ¿podrías implementar esta interfaz contra un proveedor completamente distinto sin tocar ni un solo llamante? Si la respuesta es sí, la frontera está en el sitio correcto.

Un driver, con todas las rarezas del proveedor

El driver es donde vive la API real del proveedor, y es el único sitio con permiso para conocerla.

```php
final class OpenaiAPI implements AiContract
{
    use SendsRequests;

    public function __construct(
        protected PendingRequest $request
    ) {}

    /**
     * @return array<string, mixed>
     */
    public function getSiteContent(string $model, string $prompt): array
    {
        $data = $this->send('POST', '/chat/completions', [
            'model' => $model,
            'messages' => [
                ['role' => 'user', 'content' => $prompt],
            ],
            'response_format' => ['type' => 'json_object'],
        ]);

        $content = (string) $data->json('choices.0.message.content');
        $result = json_decode($content, true);

        if (! is_array($result) || $result === []) {
            throw new \RuntimeException(
                'AI site content response could not be decoded into a non-empty
array.'
            );
        }

        /** @var array<string, mixed> $result */
        return $result;
    }
}
```

Dos cosas aquí valen más de lo que parecen.

El driver recibe un `PendingRequest` ya configurado, no un token. No sabe de dónde salieron sus credenciales, cuánto dura su tiempo de espera ni cuál es su política de reintentos. Eso es tarea del gestor, y significa que el driver sigue siendo una capa de traducción fina y trivial de simular en una prueba.

La garantía del driver es de transporte, y ahí se detiene. Te promete JSON decodificable y no vacío. Deliberadamente *no* valida que ese JSON tenga las claves que tu función necesita, porque el

driver no tiene ni idea de qué necesita tu función: esa comprobación va donde se conoce el contexto del llamante, y tiene capítulo propio. Mezclar ambas produce un driver que hay que editar cada vez que una función cambia de opinión.

Ese `throw` importa más de lo que parece. La alternativa —devolver `null` o `[]` ante basura— empuja una pregunta indemostrable («¿funcionó?») a cada llamante. Falla aquí, una vez, y en voz alta.

El gestor, donde viven las decisiones

El gestor extiende el `Manager` del propio Laravel, lo que trae gratis la resolución de drivers, su cacheo y los creadores personalizados. Lo que añades es configuración y política.

```
final class AiManager extends Manager
{
    public function getDefaultDriver(): string
    {
        return (string) Config::get('services.ai.default');
    }

    public function createOpenaiDriver(): AiContract
    {
        return $this->buildDriver(
            driverClass: OpenaiAPI::class,
            config: $this->getConfig('openai'),
        );
    }
}
```

Añadir un segundo proveedor es ahora genuinamente aditivo: escribes un driver, añades un método `createXDriver()` y añades un bloque de configuración. Ningún llamante cambia. Ningún controlador cambia.

Y `buildDriver` es donde cada llamada de IA de la aplicación hereda sus modales:

```
protected function buildDriver(string $driverClass, array $config = []):
AiContract
{
    $this->ensureValidDriver($driverClass);

    $token = $config['token'] ?? '';

    if (! is_string($token) || $token === '') {
        throw new InvalidArgumentException(
            message: 'AI driver token is not configured; expected a per-tenant
credential or an environment fallback.',
            code: Response::HTTP_INTERNAL_SERVER_ERROR,
        );
    }

    return new $driverClass(
        request: Http::baseUrl($config['url'] ?? '')
            ->withToken($token)
            ->timeout(Config::integer('services.ai.config.timeout', default:
60))
            ->retry(
                times: max(1, Config::integer('services.ai.config.retry_times',
default: 3)),
                sleepMilliseconds: max(0,
Config::integer('services.ai.config.retry_sleep_ms', default: 500)),
                when: fn (Throwable $e): bool => self::shouldRetry($e),
            )
            ->throw()
    );
}
```

Toda llamada de IA de la aplicación tiene ya un tiempo de espera, una política de reintentos y una credencial validada, y nada de ello se repite en ningún sitio.

Falla en voz alta si falta la credencial

Esa comprobación del token merece su propia sección, porque saltársela cuesta una tarde genuinamente miserable.

Si la credencial falta y no haces nada, Laravel envía alegremente `Authorization: Bearer` —la cabecera existe, su valor está vacío. El proveedor la rechaza con un error de autenticación cuyo mensaje viene a decir: *no has proporcionado una clave de API*.

Lo cual es cierto y completamente engañoso, porque el código que lee la clave se ejecutó bien y el

valor de configuración existe. Te pondrás a buscar una clave revocada, un entorno equivocado, un desfase de reloj, un proxy que quita cabeceras. La respuesta real es que una credencial por cliente resolvió a cadena vacía y nadie lo comprobó.

Falla en la frontera, con un mensaje que nombre los dos sitios de los que podría haber salido el valor. Ese error debería decirle a la siguiente persona exactamente dónde mirar, porque la siguiente persona eres tú dentro de cuatro meses.

Esto se generaliza. **Una mala configuración debe fallar donde se configura, no donde se usa.** Cuanto más se separan esos dos puntos, más caro sale el error.

Las credenciales son del cliente, no de la aplicación

Esta es la decisión que la mayoría de las primeras implementaciones se saltan, y es dificilísima de meter después.

Una única clave de API para toda la aplicación significa que el uso de todos los clientes cae en un mismo saco. No puedes atribuir el gasto. No puedes poner tope a un cliente ruidoso sin ponérselo a todos. No puedes dejar que un cliente traiga su propia clave. Y si esa clave se rota o se limita, todos los clientes fallan a la vez.

Así que la configuración de un driver se resuelve por cliente, con un valor de entorno como respaldo para quienes no han traído el suyo:

```
protected function getConfig(string $driver): array
{
    return (array) Config::get("services.ai.drivers.{$driver}", default: []);
}
```

Lo importante no es el método: es que los valores que hay detrás se resuelven a partir de los ajustes del llamante autenticado antes de construir el driver, y sólo recurren a un valor de entorno compartido cuando no hay nada específico del cliente.

Haz esto el primer día. Meter credenciales por cliente en un sistema que asumía una clave global significa tocar cada punto de llamada, cada prueba y cada clave de caché, normalmente con prisa y normalmente porque ha llegado una factura.

Reintenta lo que merece reintento

Los reintentos son donde las buenas intenciones producen malos sistemas. El instinto por defecto —«reintenta tres veces si falla»— convierte calladamente una petición malformada en tres, y un error de facturación en tres.

La política que sobrevive al contacto con un proveedor real distingue entre fallos que podrían resolverse solos y fallos que desde luego no.

```php
private static function shouldRetry(Throwable $e): bool
{
    if ($e instanceof ConnectionException) {
        return true;
    }

    if ($e instanceof RequestException) {
        $status = $e->response->status();

        return $status === 429 || ($status >= 500 && $status < 600);
    }

    return false;
}
```

Los fallos de conexión y los 5xx del proveedor son transitorios: la misma petición puede perfectamente funcionar un momento después. Un 429 es contrapresión explícita, y es el único caso en el que el proveedor te ha dicho que merece la pena esperar.

Todo lo demás no se reintenta, y el miembro importante de ese conjunto es el 4xx. Una petición malformada seguirá estándolo en el segundo intento. Reintentarla desperdicia latencia, desperdicia límite de uso y —según lo que hayas enviado y cómo cuente el proveedor— puede desperdiciar dinero. Peor aún, esconde el error: un fallo que reintentas tres veces es un fallo que tardas el triple en notar.

La última cláusula importa tanto como las dos primeras. Cualquier cosa que no sea un fallo HTTP —un error al decodificar JSON, un error de tipos, un fallo en tu propio código de mapeo— no debe reintentarse. Esos son problemas tuyos, y volver a ejecutarlos sólo retrasa la traza.

Cómo se usa desde la aplicación

Registra el gestor como singleton y pon una facade delante, para que el código llamante se lea como el resto de Laravel:

```php
final class AiServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        $this->app->singleton(AiManager::class);
    }
}
```

```
$content = AiFacade::getSiteContent(
    model: config('services.ai.model'),
    prompt: $prompt,
);
```

Ahora un llamante no puede ver qué proveedor respondió, no puede construir un cliente sin configurar y no puede saltarse por accidente la política de reintentos. Hay exactamente un camino, y está asfaltado.

Qué has comprado

Haz balance, porque es fácil mirar tres ficheros pequeños y concluir que has escrito un envoltorio por escribirlo.

Ahora tienes un único sitio donde vive la forma de la API del proveedor: un fichero que editar cuando renombren un campo. Un único sitio donde se validan las credenciales, con un error que dice dónde mirar. Un único sitio donde se deciden tiempos de espera y reintentos, aplicados a cada llamada de IA de la aplicación, incluidas las que aún no has escrito. Una interfaz que simular en las pruebas, en una frontera que no se mueve.

Y tienes dónde poner la siguiente lección. Esa es la parte que compone. Todo lo que enseña el resto del libro —techos de gasto, validación de salida estructurada, cancelación, observabilidad— aterriza limpiamente porque hay una frontera donde aterrizar. Sin ella, cada una de esas cosas se convierte en un cambio en cada punto de llamada.

El envoltorio no es el objetivo. El sitio donde poner las decisiones sí lo es.

Qué no resuelve todavía

Ten claro qué falta, porque a estas alturas el endpoint es honesto pero ingenuo.

No tiene ni idea de lo que cuesta una llamada, ni techo sobre cuántas puede hacer un cliente. Confía en que la respuesta tenga la estructura que la función necesita, cosa que todavía no es cierta. Se ejecuta dentro de la petición HTTP, que no sobrevivirá a un turno largo. Y su *prompt* sigue siendo una cadena que alguien montó en algún sitio, que es el tema de los dos capítulos siguientes.

Cada una de esas cosas es un capítulo, y cada una es un cambio *dentro de esta frontera* en lugar de un cambio repartido por tu aplicación. Ese es el retorno de los tres ficheros.

Principios

- **Pon al proveedor detrás de un contrato que describa tus necesidades, no las suyas.** Si la interfaz usa el vocabulario del proveedor, es un cambio de nombre, no una abstracción.
- **El driver es una capa de traducción, y nada más.** Recibe un cliente configurado; no lo