thinkia

BROWNFIELD AI · WHITEPAPER · JUNIO 2026

El legacy no se reescribe. Se *descodifica*.

El código heredado se descodifica en specs y se regenera. El spec es el activo; el código, el subproducto. Un método para migrar sistemas críticos con criterio arquitectónico.

RESUMEN EJECUTIVO

Toda organización con historia carga un sistema que sostiene el negocio y que casi nadie se atreve a tocar. La lógica vive en código que pocos entienden, escrito por personas que ya no están. La opción que parece prudente, reescribirlo desde cero, es la más cara y la que más reglas pierde por el camino.

Hay una lectura distinta. El valor de un sistema legacy no es su código: es la intención de negocio atrapada dentro de él. La ingeniería asistida por IA permite extraer esa intención, fijarla en una especificación gobernable y regenerar el código contra una arquitectura objetivo. El spec pasa a ser el activo duradero. El código vuelve a ser lo que siempre debió ser: una consecuencia.

Este documento expone el problema real del legacy, por qué los enfoques clásicos se quedan a medias y cómo se ejecuta una migración brownfield gobernada: el método en cinco fases, los riesgos conocidos, y las cuatro métricas que dicen si está funcionando. No es un catálogo de producto. Es una forma de trabajar.

37–42%

MENOS ESFUERZO

de desarrollo frente al método tradicional, observado en programas Coding 3.0 en producción.[1]

75%

PROFESIONALES

del conocimiento ya usan IA en su trabajo, a menudo sin gobierno de IT.[2]

5

FASES

de diagnóstico a industrialización, con una puerta de calidad en cada una.

CONTENIDOS

Carta del autor

Lo que migra no es el código. Es el *conocimiento*.

Llevo años discutiendo migraciones de sistemas críticos con directores de tecnología, y la conversación empieza casi siempre en el mismo sitio: cuánto código hay que reescribir. Es la pregunta equivocada. El código nunca fue el problema. El problema es que la mitad de las reglas que sostienen el negocio no están escritas en ninguna parte: viven en la cabeza de unas pocas personas y en condicionales que nadie recuerda haber puesto.

Cuando empezamos a trabajar con agentes de IA en serio, esa pregunta se invirtió. Dejó de importarnos cuánto código generaba la máquina y empezó a importarnos qué tan bien recuperaba la intención. Un modelo que reproduce la sintaxis de un sistema legacy no sirve de mucho; uno que recupera la regla que esa sintaxis esconde lo cambia todo.

De ahí sale la tesis de este documento, y la digo sin adornos. El código legacy no se reescribe; se descodifica en specs y se regenera. El spec es el activo. El código es el subproducto. Cuando esa idea se asume de verdad, una migración deja de ser un acto de fe: pasa a ser un proceso con trazabilidad, puertas de calidad y criterio humano.

No es escribir más código. Es otro oficio. El ingeniero deja de teclear y pasa a hacer lo que de verdad sabe hacer: decidir qué reglas importan, qué arquitectura las sostiene y qué prueba demuestra que se han preservado. La máquina genera; la persona juzga.

Quien busque un manifiesto encontrará uno al final. Quien busque un método para mover sistemas reales sin romper lo que tardó años en construirse bien, también. Las dos cosas caben aquí, en este orden.

David Alejano
Chief Strategy Officer · Thinkia · Madrid, junio 2026
linkedin.com/in/dalejano

01

PARTE I

El problema y la *tesis*.

Por qué el código heredado bloquea a las organizaciones, por qué los enfoques clásicos no lo resuelven, y qué cambia cuando el spec se convierte en el activo.

01

El sistema que sostiene el negocio es el que nadie se atreve a *tocar*.

El legacy no es un problema técnico. Es una dependencia. Un núcleo bancario, un sistema de pólizas, un motor de facturación: décadas de reglas acumuladas que funcionan, que mueven dinero real, y que la organización entiende cada vez peor. El riesgo no está en que el sistema falle. Está en que nadie sabe ya por qué funciona.

Cuatro síntomas aparecen casi siempre juntos. El primero es la **deuda técnica**: cada parche urgente dejó una cicatriz, y el coste de mantenimiento crece mientras la capacidad de cambio se hunde. El segundo es la dependencia de **expertos clave**. Hay dos o tres personas que conocen las reglas implícitas; cuando se jubilan o se van, el conocimiento se va con ellas y no queda copia.

El tercero es el miedo a tocar. Un cambio de una línea puede romper algo a tres módulos de distancia, y como nadie tiene el mapa completo, la organización opta por no moverse. La consecuencia es el cuarto síntoma: los ciclos de release se alargan. Lo que debería salir en una semana sale en un trimestre, porque cada despliegue exige una ronda de comprobación manual que sustituye a las pruebas que nunca se escribieron.

En sectores regulados hay un quinto. Cuando un sistema no es trazable, la organización no puede demostrar por qué tomó una decisión: por qué denegó un crédito, por qué liquidó un siniestro de una forma. El riesgo regulatorio deja de ser hipotético y pasa a ser una multa con fecha.

Y mientras tanto, los equipos ya generan código con IA sobre estos sistemas, casi siempre sin contexto de arquitectura y sin gobierno. La IA acelera; pero acelerar encima de una base que nadie entiende solo acumula deuda más rápido.

75 %
de los profesionales del conocimiento ya usan IA en el trabajo, a menudo sin supervisión de IT.[2]

2–3
personas suelen concentrar las reglas de negocio implícitas de un sistema core sin documentar.

×4
de diferencia típica entre un ciclo de release sano y uno frenado por miedo a tocar el legacy.

02

Cada enfoque clásico resuelve la *mitad* del problema.

Las cuatro estrategias habituales para migrar legacy comparten un mismo defecto: tratan el código como el activo y la lógica de negocio como un efecto colateral. Por eso cada una recupera una parte y deja fuera la que más cuesta rehacer.

ENFOQUE	QUÉ PROMETE	POR QUÉ SE QUEDA CORTO
Reescritura desde cero	*Un sistema limpio, moderno, sin deuda*	Reescribe la sintaxis pero pierde las reglas implícitas que nadie documentó. Proyecto largo, caro y de alto riesgo: el sistema nuevo nace sin saber qué hacía el viejo.
Refactor incremental	*Mejora continua, riesgo controlado*	Mantiene la arquitectura original y, con ella, sus límites. Mejora la forma sin recuperar la intención; la deuda se reduce, no se cancela.
Lift-and-shift	*Migrar a cloud rápido y barato*	Cambia dónde corre el sistema, no qué entiende la organización de él. El legacy sigue siendo una caja negra, ahora con factura de cloud.
Encapsulación con APIs	*Modernizar el acceso sin tocar el núcleo*	Expone el sistema sin descodificarlo. Útil como puente, pero aplaza el problema: la lógica crítica sigue encerrada y sin dueño.

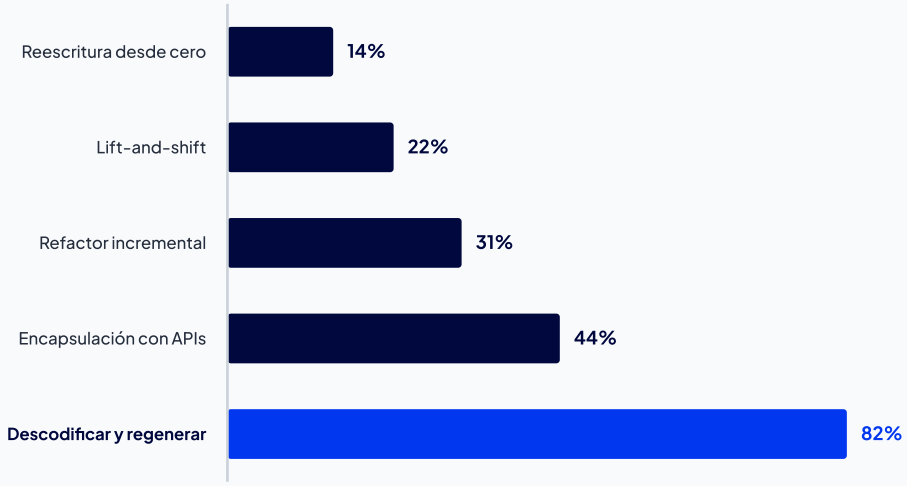
Ninguno es un error: cada uno responde a una pregunta parcial. El problema aparece cuando se eligen como estrategia completa, porque entonces la regla de negocio implícita, el activo más valioso y más frágil, queda sin recuperar.

Si se mide lo que de verdad importa (cuántas reglas de negocio sobreviven a la migración), el orden de los enfoques se invierte respecto a su popularidad. Descodificar y regenerar es el único que recupera el conjunto casi completo, porque es el único diseñado para extraer la intención antes de tocar el código.

EXHIBIT 01

Sólo descodificar y regenerar recupera el conjunto completo de reglas.

Reglas de negocio recuperadas por enfoque, % (modelo ilustrativo)



Fuente: modelo ilustrativo · análisis Thinkia sobre programas reales de migración.

03

El spec es el activo. El código es el *subproducto*.

Durante cincuenta años el código fue la verdad y la documentación, una aproximación que envejecía peor que el propio sistema. La ingeniería asistida por IA invierte esa relación. Cuando una especificación es lo bastante precisa, un modelo puede generar el código que la implementa, en el lenguaje y la arquitectura que haga falta. Lo escaso deja de ser teclear; lo escaso pasa a ser la atención clara.

Un **spec**, en este sentido, no es un documento de requisitos que precede al desarrollo y luego se abandona. Es el contrato que el código implementa, vivo y versionado: qué hace el sistema, qué reglas respeta, qué no debe hacer nunca. El código generado a partir de él es desechable por diseño. Si la arquitectura objetivo cambia, se regenera. Si aparece un lenguaje mejor, se regenera. El spec permanece.

Esto reordena qué es valioso en una migración. El activo que la organización quiere proteger no es el .NET de hace quince años ni el monolito que nadie toca: es la regla de negocio que vive dentro. Descodificarla a un spec la saca de la cabeza de tres personas y la convierte en patrimonio de la empresa, legible para humanos y ejecutable por agentes.

El criterio humano no desaparece; se concentra. En lugar de repartirse entre miles de líneas, se invierte donde decide el resultado: qué reglas son correctas, qué arquitectura las sostiene y qué prueba demuestra que se han preservado. La máquina asume la sintaxis. La persona asume el juicio.

El spec es el *activo*. El código, el *subproducto*.

04

Descodificar es recuperar la intención, no copiar la *sintaxis*.

El reverse engineering asistido por IA no es leer código y traducirlo. Es reconstruir, en tres movimientos, lo que el sistema sabe hacer y por qué. Cada movimiento tiene una salida revisable por una persona antes de pasar al siguiente.

01

Extracción de la lógica de negocio

La IA recorre el sistema legacy y separa la intención de la implementación: qué reglas se aplican, en qué orden, con qué excepciones. Las reglas implícitas (las que viven en condicionales sin comentar o en el conocimiento de un experto) se hacen explícitas y se ponen sobre la mesa para validar.

02

Generación de specs desde el código existente

La lógica recuperada se fija en una especificación gobernable, con su documentación embebida y su trazabilidad hasta el fragmento de origen. El spec deja de depender del sistema viejo: es un artefacto propio, versionado, que cualquier herramienta de AI coding puede consumir para generar el nuevo código.

03

Validación contra la arquitectura objetivo

Antes de generar una sola línea, el spec se contrasta con la arquitectura de destino: qué encaja, qué hay que adaptar, qué regla choca con el nuevo modelo. La migración se valida contra el diseño objetivo, no contra el sistema que se quiere dejar atrás.

02

PARTE II

El método y la *ejecución*.

Cómo se ejecuta una migración brownfield gobernada: las fases, la economía del esfuerzo, los casos, los riesgos y las métricas que deciden si funciona.

05

Cinco fases, una puerta de calidad en *cada una*.

Una migración gobernada avanza por cinco fases. El peso humano es máximo al principio, cuando se decide qué reglas importan y qué arquitectura las sostiene, y la IA asume cada vez más carga a medida que el criterio queda fijado en specs. El diagnóstico precede al compromiso: la primera fase entrega un diagnóstico accionable antes de que la organización se ate a nada.

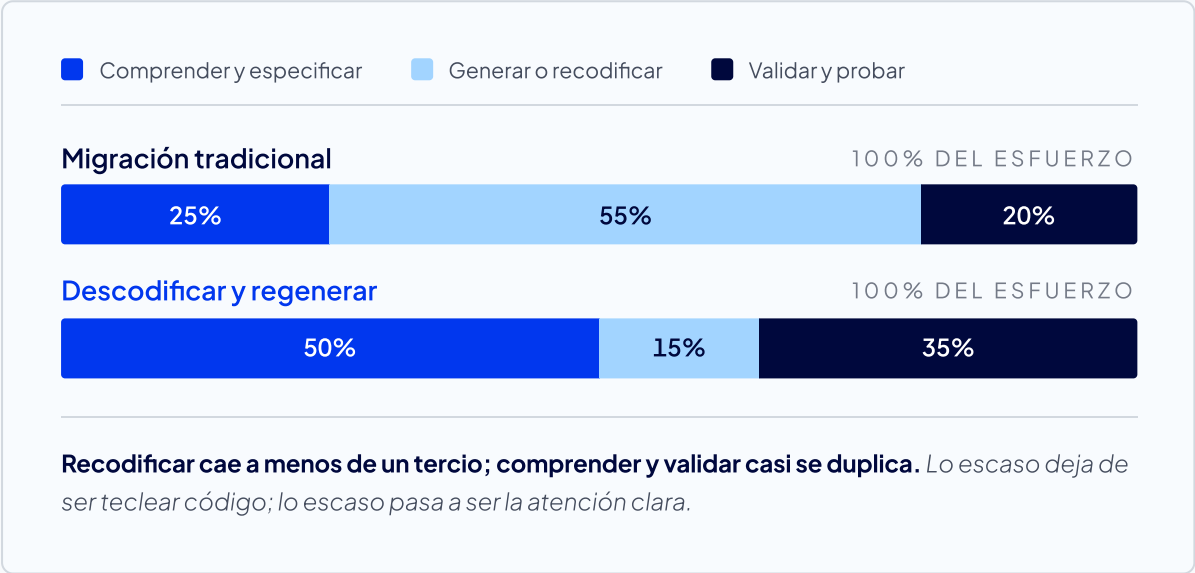
TRACK TRANSVERSAL Arquitectura, reglas de negocio y trazabilidad · activo en todas las fases

F0	F1	F2	F3	F4
Diagnóstico	Cimientos	Casos faro	Escalado	Industrialización
90% humano	60% humano	70% IA	80% IA	Continuo

El orden importa más que la velocidad. **F0** diagnostica y califica el caso; **F1** levanta los cimientos (arquitectura objetivo, reglas críticas, primeras specs); **F2** prueba el método en un caso faro con métrica de negocio; **F3** escala a los dominios contiguos; **F4** industrializa la práctica para que la organización la sostenga sin Thinkia. Ninguna fase pasa la puerta sin validar las reglas de negocio contra el diseño objetivo.

A dónde va el esfuerzo cuando el spec es el activo

El método no reduce el trabajo repartiendo lo mismo entre humano y máquina. Lo redistribuye. En una migración tradicional, la mayor parte del esfuerzo se va en recodificar a mano. Cuando se descodifica y se regenera, ese bloque se desploma y crece el que de verdad protege el negocio: comprender, especificar y validar.



Recodificar cae a menos de un tercio; comprender y validar casi se duplica. *Lo escaso deja de ser teclear código; lo escaso pasa a ser la atención clara.*

El efecto neto, observado en programas Coding 3.0 en producción, es entre un 37 % y un 42 % menos de esfuerzo de desarrollo manteniendo la calidad.[1] No es magia: es dejar de pagar dos veces por el mismo conocimiento, escribiéndolo una sola vez en el spec y regenerando el código cuantas veces haga falta.

06

Dos migraciones, una misma *disciplina*.

Los casos están anonimizados por política de confidencialidad. Importan por lo que enseñan del método, no como prueba de venta: dónde estuvo la fricción y qué se aprendió.

El patrón se repite. La parte cara y arriesgada no fue generar el código nuevo, sino recuperar y validar las reglas que el sistema viejo nunca explicó. Donde el método aportó fue en hacer ese trabajo explícito, trazable y revisable; donde exigió rigor, fue en no dejar pasar ninguna fase sin la firma de una persona.

07

Los riesgos de descodificar con IA son conocidos y *gobernables*.

Descodificar con IA introduce riesgos propios. Negarlos sería irresponsable; ignorarlos, peligroso. Cada uno tiene una mitigación concreta dentro del método.

RIESGO	CÓMO APARECE	MITIGACIÓN
Alucinación sobre el código	*La IA infiere una regla que el sistema nunca aplicó*	Trazabilidad hasta el fragmento de origen y validación humana obligatoria antes de fijar la regla en el spec.
Reglas implícitas perdidas	*Una regla vive solo en la cabeza de un experto*	Sesiones de validación con los expertos clave durante F0–F1, mientras el conocimiento todavía está disponible.
Deriva de arquitectura	*El código generado se aleja del diseño objetivo*	Validación del spec contra la arquitectura de destino en cada puerta de fase, no al final.
Regresiones funcionales	*El sistema nuevo cambia un comportamiento sin querer*	Cobertura de tests sobre el comportamiento legacy, generada desde las reglas recuperadas y ejecutada en cada release.
Dependencia en la transición	*El equipo no sabe operar el método sin ayuda*	F4 industrializa la práctica y transfiere el conocimiento; el objetivo es que la organización lo sostenga sola.

08

Cuatro métricas dicen si la migración *va bien*.

Una migración brownfield no se mide por líneas migradas. Se mide por conocimiento recuperado y riesgo retirado. Cuatro indicadores bastan para saber si va en la dirección correcta.

KPI 1	*Reglas recuperadas*	Porcentaje de reglas de negocio extraídas del legacy y validadas por una persona. Es el indicador maestro: mide el activo, no el código.
KPI 2	*Cobertura de tests*	Porcentaje del comportamiento legacy cubierto por pruebas generadas desde las reglas recuperadas. Sin cobertura, una regla recuperada es una hipótesis sin demostrar.
KPI 3	*Deuda residual*	Qué porción del sistema sigue sin descodificar y sin dueño. Mide lo que falta, no lo que se ha hecho; es la honestidad del proyecto.
KPI 4	*Time-to-first-release*	Tiempo hasta el primer release gobernado del sistema nuevo. Mide si el método entrega valor pronto o solo promete entregarlo tarde.

Conclusión

Migrar legacy deja de ser un acto de *fe*.

Durante demasiado tiempo, modernizar un sistema crítico fue elegir entre dos malas opciones: convivir con una caja negra o reescribirla a ciegas. La ingeniería asistida por IA abre una tercera. Recuperar la intención que el código esconde, fijarla en un spec gobernable y regenerar contra una arquitectura objetivo convierte la migración en un proceso con trazabilidad, puertas de calidad y criterio humano donde decide el resultado.

La consecuencia para una organización con legacy es directa. El conocimiento que hoy vive en tres personas y en condicionales sin comentar puede pasar a ser patrimonio explícito, legible y ejecutable. El sistema viejo deja de ser un rehén y se vuelve una fuente. Y el equipo deja de teclear para volver a hacer ingeniería: decidir qué reglas importan y qué prueba lo demuestra.

La pregunta para cualquier CTO ya no es cuánto código hay que reescribir. Es otra, más incómoda y más útil: ¿están sus reglas de negocio escritas en algún sitio, o solo cree que lo están?

El código legacy no se reescribe. Se descodifica en specs y se regenera. El spec es el *activo*; el código, el *subproducto*.

No hype. No teoría. Solo impacto.

THINKIA · JUNIO 2026

Sobre Thinkia

Una compañía nativa-IA, no una consultora con un *departamento* de IA.

Thinkia es una consultora global nacida para la Era de la Inteligencia. Su ontología y su ADN son IA-nativos: estrategia, diseño e ingeniería trabajan como un solo motor para llevar al cliente de la idea a un resultado que opera. No entregamos tecnología; entregamos resultados medibles, y trabajamos a lo largo de todo el ciclo en lugar de hacer un piloto y marcharnos.

En desarrollo de software, esa convicción toma la forma del paradigma Coding 3.0 y de la metodología AI-SDLC: especificar la intención con rigor, validar contra arquitectura y dejar que el código sea consecuencia del spec. Este documento es una pieza de esa idea aplicada al problema más persistente de cualquier organización con historia: el legacy.

CÓMO CITAR ESTE DOCUMENTO

Alejano, D. (2026). *El legacy no se reescribe: migración brownfield con criterio arquitectónico*. Thinkia Whitepaper. thinkia.com

REFERENCIAS

Fuentes.

Cifras y referencias citadas a lo largo del documento. Los datos de programas reales se presentan anonimizados por política de confidencialidad y RGPD.

01 Análisis interno de Thinkia sobre programas de desarrollo AI-SDLC / Coding 3.0 en producción, 2025. Reducción de esfuerzo de desarrollo del 37–42 % frente a metodología tradicional, manteniendo calidad.

02 Estudios de adopción de IA en el puesto de trabajo, 2025. Aproximadamente el 75 % de los profesionales del conocimiento declara usar IA en su trabajo, con frecuencia sin supervisión de IT (*Shadow AI*).

03 Modelo ilustrativo de recuperación de reglas de negocio por enfoque de migración (Exhibit 01). Análisis Thinkia sobre programas reales; las cifras representan órdenes de magnitud comparativos, no una medición única.

thinkia

thinkia.com