11-1-2026

AI-First Governed Software Development

A practical methodology for developing with AI under explicit technical governance.

Manuel J. Gonzalez Lopez
CTO – ARQUITECTO DE SISTEMAS – AI GOVERNANCE

Índice

Nota sobre autoría y uso

Este documento recoge un marco de trabajo y una metodología desarrollados a partir de experiencia profesional en diseño, desarrollo y operación de sistemas de software durante más de tres décadas.

Su contenido puede ser leído, utilizado y adaptado libremente en contextos profesionales y organizativos.

Se permite su uso total o parcial siempre que se mantenga la referencia a su autoría original.

No se permite su redistribución como producto propio ni su comercialización como metodología de terceros sin atribución expresa.

Introducción

AI-First en desarrollo: acelerar sin perder el control

Escribir código ya no es el principal cuello de botella. La IA ha reducido drásticamente el esfuerzo de desarrollo y eso es una realidad.

El problema aparece cuando esa reducción de coste lleva a una conclusión equivocada: que el código ya no tiene valor. Lo que ha perdido valor es el esfuerzo mecánico de escribirlo. No el conocimiento ni las decisiones que el código encapsula.

Hoy se está generando software cada vez más rápido sin haber rediseñado el sistema que gobierna cómo se desarrolla. Se acelera la producción, pero no el control. Y cuando eso ocurre, la arquitectura se degrada, la dependencia aumenta y la responsabilidad se diluye.

El IDE ya no es una herramienta pasiva. El modelo propone estructuras, introduce dependencias y condiciona la arquitectura, muchas veces sin que nadie lo haya decidido explícitamente. Sin embargo, seguimos trabajando con roles y procesos pensados para un desarrollo manual que ya no existe.

Este enfoque nace de años diseñando plataformas que generan software (low-code, no-code y multicode)y de haber visto repetirse siempre el mismo patrón: cuando la generación se acelera sin gobierno, el sistema pierde coherencia y control. La IA no ha creado este problema, lo ha amplificado.

AI-First aplicado al desarrollo no significa usar IA en todas partes. Significa diseñar explícitamente el sistema que decide qué se puede generar, bajo qué reglas, cómo se valida, qué se aprende y quién responde cuando algo falla.

Esta metodología existe para una sola cosa: **ganar velocidad sin perder el control del sistema**

A quién va dirigida esta metodología

Esta metodología está pensada para organizaciones y profesionales que **desarrollan software que debe mantenerse y evolucionar en el tiempo**, y que están incorporando la IA de forma estructural en su proceso de desarrollo o en el propio producto.

No está orientada a aprender a usar herramientas de IA, ni a acelerar pruebas aisladas o prototipos sin impacto real.

Está dirigida principalmente a:

- **CTOs y responsables de tecnología**
  que necesitan introducir IA sin perder control sobre arquitectura, calidad y evolución.
- **Arquitectos de software y de soluciones**
  que diseñan marcos técnicos, definen estándares y gobiernan el desarrollo a escala.
- **Responsables de plataformas, ingeniería o desarrollo**
  que deben equilibrar velocidad, coherencia y sostenibilidad del sistema.
- **Organizaciones con software crítico**
  (producto propio, SaaS, ERP, sistemas core, desarrollo a medida) donde los errores tienen impacto real en negocio, clientes o usuarios.

Aplica especialmente en contextos como:

- empresas de producto o software empresarial,
- plataformas SaaS en evolución continua,
- grandes corporaciones con desarrollo interno,
- consultoras y software factories con marcos técnicos propios,
- organizaciones con rotación de equipos y necesidad de continuidad técnica.

No está pensada para:

- aprender a programar,
- experimentar con prompts de forma individual,
- proyectos efímeros sin mantenimiento,
- demos, POCs o pruebas aisladas sin paso a producción,
- equipos que no necesitan gobernar arquitectura ni evolución.

La metodología **no impide** estos usos, pero **no está diseñada para ellos**.

Si una organización, utiliza IA en el desarrollo, quiere ganar velocidad, y no está dispuesta a perder control sobre su sistema, **esta metodología es aplicable**.

Si el objetivo es únicamente experimentar sin gobernar consecuencias, esta metodología **no es necesaria**.

Responsabilidades y perfiles implicados en el desarrollo AI-First

Esta metodología no introduce nuevos roles ni propone reorganizaciones formales. Parte de una realidad simple: **cuando la IA entra en el desarrollo de software, ciertas responsabilidades deben hacerse explícitas** o el sistema pierde control.

La metodología asume que estas responsabilidades pueden recaer:

- en una sola persona,
- en varias,
- repartidas entre perfiles internos y externos,
  dependiendo del tamaño, madurez y estructura de cada organización.

Lo importante no es el cargo, sino **que la responsabilidad exista y esté clara**.

Responsabilidad sobre la arquitectura y el sistema de desarrollo

Debe existir una responsabilidad técnica clara sobre:

- la arquitectura del sistema,
- los patrones de desarrollo,
- los frameworks y versiones soportadas,
- y cómo se integra la IA en el ciclo de vida del software.

Esta responsabilidad garantiza la **coherencia técnica y la sostenibilidad del sistema**, no decide la lógica de negocio.

Perfil habitual
CTO, Arquitecto de software, arquitecto de soluciones o responsable técnico senior.

No se trata de decidir la lógica de negocio, sino de **garantizar coherencia técnica y sostenibilidad**.

Responsabilidad sobre el uso de IA en el desarrollo

Debe existir una responsabilidad explícita sobre:

- dónde se permite usar IA en el desarrollo,
- con qué límites,
- y bajo qué condiciones técnicas.

Incluye decisiones como:

- qué puede generar la IA,
- qué partes del sistema no debe tocar,
- cuando es obligatoria la supervisión humana.

Esta responsabilidad **no debería recaer en perfiles puramente funcionales o administrativos**.

Perfil habitual
Arquitectura, liderazgo técnico o responsables del sistema de desarrollo.

Responsabilidad sobre datasets, contexto y modelos usados en desarrollo

Alguien debe asumir la responsabilidad de **preparar, mantener y poner a disposición**:

- los datasets utilizados como referencia,
- el contexto técnico (código, documentación, ejemplos),
- y los modelos o adaptaciones que usarán los desarrolladores.

No se trata de entrenar modelos genéricos, sino de:

- decidir qué conocimiento técnico es válido,
- normalizarlo y versionarlo,
- y garantizar que la IA trabaja con un contexto coherente con la arquitectura real.

El objetivo no es que cada desarrollador "use la IA a su manera", sino que **la IA opere sobre un conocimiento técnico gobernado**.

Perfil habitual
Arquitectura de software, plataformas internas de desarrollo, equipos técnicos responsables del framework interno.
En organizaciones pequeñas, esta responsabilidad suele recaer en el CTO o arquitecto principal.

Responsabilidad sobre calidad, validación y revisión asistida por IA

En un desarrollo AI-First, la calidad no puede depender únicamente de herramientas clásicas de QA ni de revisiones manuales puntuales.

Debe existir una responsabilidad explícita sobre:

- la validación del código generado o modificado con IA,
- la revisión del cumplimiento de arquitectura y patrones,
- y la detección temprana de desviaciones técnicas o funcionales.

Con el uso de LLMs, esta responsabilidad se amplía de forma significativa.

Es posible crear **agentes de revisión** que analicen el código como si fuera un sistema de conocimiento:

- recorriendo repositorios completos,
- entendiendo relaciones entre componentes,
- y evaluando el código como un conjunto coherente, no como ficheros aislados.

Estos agentes pueden:

- revisar cumplimiento de patrones arquitectónicos,
- detectar inconsistencias entre capas,
- señalar uso incorrecto de frameworks o versiones,
- validar decisiones de diseño,
- e incluso contrastar implementación con especificaciones funcionales.

Esto no sustituye al criterio humano. Lo **refuerza**.

Relación entre QA asistido por IA y herramientas clásicas

Las herramientas tradicionales de calidad (tests, linters, análisis estático, pipelines) **siguen siendo necesarias**, pero dejan de ser suficientes por sí solas.

La IA permite:

- analizar lo que las herramientas no entienden (intención, coherencia, diseño),
- y hacerlo de forma continua, no solo al final del ciclo.

El QA deja de ser una fase reactiva y pasa a ser **una capacidad activa del sistema de desarrollo**.

Perfil habitual que asume esta responsabilidad

Esta responsabilidad puede recaer en:

- responsables de calidad técnica,
- arquitectos,
- liderazgo técnico,
- equipos de plataforma de desarrollo.

No debe recaer exclusivamente en herramientas automáticas ni en desarrolladores individuales actuando de forma aislada.

El QA asistido por IA **no es un reemplazo del equipo**, es una extensión de su capacidad de revisión.

Relación con aprendizaje y evolución

Los resultados del QA asistido por IA son una fuente clave para:

- detectar errores recurrentes,
- identificar malas prácticas,
- ajustar reglas y contexto,
- y mejorar progresivamente el sistema de desarrollo.

Cuando estos resultados no se registran ni se analizan:

- los mismos errores reaparecen,
- y la IA aprende ruido en lugar de criterio.

En un desarrollo AI-First, la calidad ya no es solo comprobar si el código funciona. Es comprobar si **el sistema sigue siendo coherente mientras evoluciona más rápido**.

Responsabilidad sobre excepciones al marco definido

Las excepciones son inevitables en sistemas reales. Lo que no es aceptable es que sean invisibles.

Toda excepción relevante debe:

- tener un responsable técnico explícito,
- quedar registrada,
- y poder revisarse cuando el sistema evoluciona.

No se trata de aprobar cada línea de código, sino de **asumir conscientemente las decisiones fuera del estándar**.

Perfil habitual
Arquitectura o liderazgo técnico, en coordinación con desarrollo.

Responsabilidad sobre la evolución y el aprendizaje del sistema

Cuando el sistema evoluciona (por cambios tecnológicos, decisiones humanas o aprendizaje asistido por IA) alguien debe decidir:

- qué se incorpora como nuevo estándar,
- qué se descarta,
- y qué no debe repetirse.

El desarrollador no desaparece. Cambia su rol: pasa de ejecutar decisiones a **supervisar, validar y aportar criterio técnico**.

Perfil habitual
Arquitectura y responsables técnicos senior, con participación activa de los desarrolladores.

Nota final

Esta metodología no exige comités ni estructuras pesadas. Exige algo más simple y difícil: **que las responsabilidades no queden implícitas**.

En desarrollo, los problemas no aparecen porque falten normas, sino porque **nadie sabe quién debía decidir cuando algo se salió del marco**.

Principios fundacionales de la metodología

Estos principios no pretenden defender el valor del código por nostalgia ni por oficio. Pretenden **ponerlo en su lugar correcto dentro de sistemas que deben funcionar, mantenerse y evolucionar en el tiempo**. La IA no elimina estos principios. Los hace visibles.

Principio 1 La velocidad sin arquitectura no genera valor sostenible

Acelerar el desarrollo siempre ha sido posible. Lo complejo siempre ha sido **mantener y evolucionar lo construido**.

Cuando la velocidad no está gobernada por una arquitectura explícita:

- el conocimiento se fragmenta,
- el control se diluye,
- y la dependencia aumenta.

La IA incrementa la velocidad. Sin arquitectura, incrementa también el riesgo.

Principio 2 El código no es el objetivo, pero sigue siendo un activo crítico

El valor del software no está en escribir líneas de código, pero **el código sigue siendo el soporte donde ese valor se materializa**.

Negar la importancia del código no lo hace desaparecer. Solo hace que se vuelva opaco, frágil y difícil de gobernar.

Cuando el código no se trata como un activo vivo:

- el mantenimiento se encarece,
- la evolución se ralentiza,
- y el sistema pierde capacidad de adaptación.

Principio 3 La estandarización protege el sistema frente a la improvisación

La estandarización no existe para limitar creatividad. Existe para **preservar coherencia a largo plazo**.

Cuando:

- los patrones son claros,
- los componentes están definidos,
- y las reglas son explícitas,

romper el marco cuesta más que seguirlo. Y eso protege al sistema incluso cuando cambian las personas.

Principio 4 El verdadero coste del software está en su ciclo de vida

La construcción inicial representa una parte menor del coste total. La mayor inversión está en **mantener, adaptar y evolucionar el sistema**.

Ignorar esta realidad conduce a soluciones rápidas pero frágiles. La IA acelera la construcción, pero **no elimina la responsabilidad sobre el ciclo de vida**.

Principio 5 Sin gobierno explícito, la IA convierte el desarrollo en una caja negra

La IA puede generar código, pero no entiende el sistema completo. Cuando:

- no hay criterios comunes,
- no hay límites claros,
- y no hay trazabilidad,

el resultado puede funcionar a corto plazo, pero se vuelve incomprensible y difícil de sostener.

La IA sin gobierno no elimina complejidad. **La oculta**.

Principio 6 La arquitectura es el mecanismo real de gobierno del desarrollo

La arquitectura no es documentación ni diagramas. Es el conjunto de decisiones que determinan cómo se construye, se valida y se mantiene el software.

Con IA, la arquitectura no pierde relevancia. Se convierte en el único punto de control efectivo.

Principio 7 La responsabilidad técnica no se automatiza

La IA puede asistir, proponer y ejecutar. La responsabilidad sigue siendo humana.

El desarrollador no desaparece. Su rol evoluciona hacia:

- supervisión,
- validación,
- y garantía de coherencia del sistema.

Eliminar al humano del proceso no es eficiencia. Es renunciar al control.

Decir que "el código ya no tiene valor" es una simplificación peligrosa. El valor no está en escribir código por escribirlo, pero todo sistema crítico sigue dependiendo de cómo ese código se genera, se gobierna y se mantiene.

Esta metodología no defiende el código como fin. Defiende el sistema que lo produce y lo sostiene en el tiempo. **La IA no sustituye ese sistema. Lo exige.**

Qué define esta metodología y qué no

Esta metodología define cómo diseñar y gobernar un sistema de desarrollo AI-First cuando el software es crítico y debe mantenerse en el tiempo.

No es un manual de herramientas ni una guía para elegir modelos concretos. Tampoco pretende estandarizar arquitecturas, proveedores o stacks técnicos.

El motivo es simple: esas decisiones dependen del contexto.

Dos organizaciones pueden aplicar esta metodología de forma correcta y llegar a decisiones técnicas distintas, ambas válidas, porque:

- operan en sectores diferentes,
- tienen niveles de riesgo distintos,
- parten de arquitecturas previas distintas,
- y están sujetas a restricciones legales, técnicas u organizativas diferentes.

La metodología se centra en definir:

- qué decisiones deben tomarse,
- en qué orden,
- con qué criterios,
- y con qué mecanismos de control.

Define qué debe gobernarse antes de escalar el uso de IA en el desarrollo de software.

No prescribe:

- qué modelo utilizar,
- si debe ser comercial u open source,
- si el despliegue es on-premise, cloud o híbrido,
- si la IA se integra mediante IDEs, agentes, servicios o componentes internos,
- ni el coste o la infraestructura concreta necesaria.

No porque estos aspectos no sean importantes, sino porque no pueden decidirse de forma genérica sin analizar el sistema, el negocio y los riesgos reales.

Esas decisiones forman parte del trabajo de implantación, no del marco metodológico.

Esta metodología no pretende decir *cómo implementar IA*, sino cómo evitar perder el control cuando se hace. A partir de aquí, las fases del método describen:

- cómo analizar el contexto real,
- cómo definir el sistema de desarrollo,
- cómo gobernar el código y el conocimiento técnico,
- cómo asegurar observabilidad y trazabilidad,
- y cómo permitir evolución y aprendizaje sin degradar el sistema.

La implementación concreta (modelos, costes, herramientas y arquitectura) es consecuencia de aplicar correctamente estas fases, no su punto de partida.

Si una organización busca una receta rápida o una lista de herramientas, este documento no es suficiente.

Si lo que necesita es un marco sólido para tomar decisiones correctas antes de automatizar, esta metodología aplica.

Cómo usar esta metodología

Esta metodología no está pensada para "implantarse" como un paquete cerrado ni para ejecutarse como una secuencia rígida de pasos. Está pensada para **ordenar el desarrollo asistido por IA desde el sistema completo**, no desde una herramienta concreta ni desde una moda tecnológica.

No exige cambiar de stack, de frameworks ni de proveedores. Exige algo más básico y más difícil: **hacer explícitas decisiones, límites y responsabilidades que normalmente quedan implícitas**.

En la práctica, se utiliza como un marco de análisis y diseño que permite:

- entender dónde la IA puede aportar valor real,
- identificar dónde introduce riesgo si se aplica sin control,
- y definir cómo debe integrarse en el desarrollo sin romper la gobernabilidad del sistema.

La metodología puede recorrerse de forma incremental. No es necesario tener todas las fases "perfectas" para avanzar, pero sí es importante **no saltarse fases sin ser consciente de las consecuencias**.

Fases del método

El recorrido para implantar AI-First en desarrollo

Este método se estructura en **5 fases**. No son principios. No son opcionales. Son **etapas que se recorren**, aunque el ritmo y la profundidad varíen según la organización. No es un "todo o nada". Es un **proceso acumulativo**.

Qué se obtiene en las primeras sesiones de trabajo

Aplicar esta metodología no empieza generando más código ni más agentes. Empieza aclarando **cómo se está desarrollando software cuando la IA entra en el proceso**.

En las primeras sesiones de trabajo, el resultado habitual es:

- Una visión clara de **cómo se toman hoy las decisiones técnicas** cuando se usa IA en el desarrollo:
  - decisiones de diseño,
  - decisiones de arquitectura,
  - decisiones de generación de código,
  - y decisiones de evolución.
- Identificación de **dónde la IA ya está influyendo en el código** sin que exista un marco técnico explícito:
  - versiones no controladas,
  - patrones inconsistentes,
  - decisiones implícitas en prompts o agentes.
- Separación clara entre:
  - código que puede generarse de forma asistida,
  - código que debe seguir patrones estrictos,
  - y partes del sistema que no deberían tocarse sin supervisión.
- Definición de **qué repositorios, ramas, librerías y documentación** forman realmente parte del sistema de desarrollo gobernado.
- Claridad sobre **qué conocimiento técnico puede usarse como contexto o entrenamiento** y cuál no, evitando mezclar legado, excepciones y decisiones obsoletas.
- Un marco común entre arquitectos y desarrolladores para trabajar con IA **sin perder coherencia técnica ni control del sistema**.

El valor de estas primeras fases no está en programar más rápido, sino en **evitar que la IA introduzca deuda técnica acelerada** que luego es muy costosa de revertir.

*En desarrollo, el problema no es que la IA genere código. El problema es **qué decisiones técnicas está tomando mientras lo genera** y si el sistema está preparado para asumirlas.*

Fase 1 Diagnóstico de gobernabilidad y evolución del software

La Fase 1 no analiza herramientas ni modelos.

Analiza si el sistema de software y el sistema de desarrollo pueden soportar la entrada de la IA sin perder control.

Antes de decidir *cómo* usar IA, hay que entender dónde, para qué y con qué impacto. Esta fase es deliberadamente contextual. No existe un diagnóstico único válido para todas las organizaciones.

1. Identificar el tipo de organización y su modelo de desarrollo

Lo primero es entender qué tipo de organización tenemos delante, porque eso condiciona todo lo demás.

Por ejemplo:

- **Software factory**
  Desarrollo para terceros, presión de plazos, rotación alta, múltiples arquitecturas.
- **Empresa de producto / ERP / software propio**
  El código es un activo comercial. Importan versiones, compatibilidad y evolución a largo plazo.
- **Empresa SaaS**
  Cambios continuos, impacto directo en clientes, despliegue frecuente.
- **Gran corporación con desarrollo a medida**
  (banca, seguros, logística, industria)
  Sistemas críticos, legacy, regulación, múltiples equipos y dependencias.

Aquí no se juzga. Se establece el marco real de trabajo.

2. Entender el sector y el impacto del software

No es lo mismo desarrollar software para:

- una administración pública,
- un hospital,
- un laboratorio,
- una entidad financiera,
- una plataforma logística.

En este punto se evalúa:

- nivel de regulación,
- impacto en personas o negocio,
- tolerancia al error,
- necesidad de trazabilidad y auditoría.

La gobernabilidad es proporcional al impacto, no a la tecnología.

3. Clasificar el tipo de software existente y futuro

Este es uno de los puntos más críticos del diagnóstico. Se distingue explícitamente entre:

a) Software existente / legacy

- mantenimiento,
- correctivos,
- pequeños evolutivos,
- sistemas con alta dependencia histórica.

Aquí el objetivo no es acelerar, sino no romper. La IA se usa, si se usa, de forma contenida y controlada.

b) Componentes simples o periféricos

- CRUDs,
- integraciones,
- llamadas a servicios externos,
- procesos BPM,
- lógica determinista.

Aquí la IA puede aportar velocidad con riesgo bajo, y no tiene sentido sobredimensionar el método.

c) Software estructural o de producto

- aplicaciones de gestión,
- ERPs,
- plataformas,
- core systems.

Este software va a llevar IA integrada:

- para decisiones,
- cálculos,
- recomendaciones,
- comportamiento adaptativo.

Aquí la IA no es un añadido, es parte del diseño del sistema. El nivel de gobernabilidad exigido es máximo.

4. Analizar cómo entra hoy la IA en el desarrollo

Solo después de entender el contexto se analiza:

- dónde se usa IA actualmente,
- para qué se usa (copilot, generación, refactorización, soporte),
- con qué modelos,
- con qué controles (si existen),
- y quién la usa realmente.

Aquí suelen aparecer:

- prompts personales,
- decisiones técnicas implícitas,
- dependencia de herramientas,
- ausencia de criterios comunes.

5. Localizar dónde vive hoy el control

Este punto es clave y normalmente incómodo. Se analiza:

- si la arquitectura vive en personas, documentos o herramientas,
- si el conocimiento es explícito o implícito,
- si la coherencia depende solo de revisiones humanas,
- si existen puntos claros donde se puede decir “esto no”.

No se busca culpables. Se mapea la realidad.

6. Evaluar la preparación para software con IA integrada

Dado que el futuro del software incluye IA embebida, se analiza:

- qué decisiones podrían quedar dentro del propio software,
- qué comportamiento será probabilístico,
- qué impacto tendrá eso en negocio y usuarios,
- qué nivel de explicabilidad y trazabilidad será exigible,
- qué partes del sistema no deberían aprender.

Este análisis condiciona todas las fases posteriores.

Qué se obtiene de la Fase 1

La Fase 1 no produce un informe genérico. Produce claridad.

Permite responder con honestidad a preguntas como:

- ¿Dónde podemos acelerar sin riesgo?
- ¿Dónde la velocidad sería peligrosa?
- ¿Qué software no debe tocarse todavía?
- ¿Qué parte del sistema va a integrar IA de forma estructural?
- ¿Dónde perderíamos el control primero?

Artefactos de la Fase 1

Artefacto 1 Mapa de gobernabilidad y evolución del software

Incluye:

- tipo de organización,
- sector e impacto,
- clasificación del software (legacy, simple, estructural),
- uso real de IA,
- puntos de decisión críticos,
- riesgos técnicos y organizativos,
- primeras líneas rojas.

Este mapa **condiciona el resto del método**.

Artefacto 2 — Matriz de riesgo y aceleración

Relaciona:

- tipo de software,
- nivel de automatización posible,
- impacto de error,
- nivel de control requerido.

Sirve para decidir:

- dónde acelerar ya,
- dónde ir despacio,
- y dónde no usar IA todavía.

Checklist

Diagnóstico de gobernabilidad y evolución

Antes de introducir o escalar IA en el desarrollo, debe poder responderse **sí** a lo siguiente:

Contexto y tipo de organización

☐ Está identificado el tipo de organización (producto, SaaS, software factory, corporación).

☐ Está identificado el sector y el impacto del software.

☐ Está claro qué papel juega el software en el negocio.

Tipología de software

☐ El software está clasificado en:

- legacy / mantenimiento,
- componentes simples,
- software estructural o core.

☐ Está definido qué software **no debe tocarse** con IA.

☐ Está definido dónde la IA puede aportar valor sin riesgo.

Uso actual de IA

☐ Se sabe dónde y cómo se está usando IA hoy.

☐ Se identifican dependencias implícitas (prompts, personas, herramientas).

☐ Existen decisiones técnicas que hoy se toman sin criterio común.

Preparación para IA integrada

☐ Está identificado qué software incorporará IA de forma estructural.

☐ Se conocen los riesgos de trazabilidad, impacto y responsabilidad.

☐ Se han definido primeras líneas rojas.

Nota sobre excepciones y responsabilidad

☐ Las excepciones al marco definido están permitidas, pero no implícitas.
☐ Toda excepción relevante tiene un **responsable técnico explícito**.
☐ La excepción queda **registrada** junto con su justificación.
☐ Se define si la excepción es:

- puntual,
- temporal,
- estructural.

☐ Existe un criterio claro para **revisar o retirar la excepción** cuando el sistema evoluciona.

Si no se cumple este checklist, **no se pasara a la Fase 2**.

Si esta fase se hace mal, el resto del método se convierte en genérico. Si se hace bien, ***todo lo demás encaja****.*

La Fase 1 no busca justificar el uso de IA. Busca ***evitar errores estructurales antes de aumentar la velocidad****.*

Fase 2 Definición del sistema de desarrollo asistido por IA

La Fase 2 consiste en diseñar explícitamente cómo debe funcionar el desarrollo asistido por IA en ese contexto concreto, no en abstracto.

Esta fase depende totalmente de la Fase 1. No existe un único sistema de desarrollo válido para todos los tipos de software ni para todas las organizaciones.

Aquí no se decide "usar IA". Aquí se decide dónde, cómo y hasta dónde.

1. Determinar qué software puede y qué software no puede usar IA

El primer paso no es tecnológico, es de descarte. A partir del mapa obtenido en la Fase 1, se distingue claramente:

a) Software existente / legacy

En muchos casos:

- no puede integrar IA de forma estructural,
- no es rentable rediseñarlo,
- el riesgo supera el beneficio.

Aquí pueden darse varios escenarios válidos:

- no usar IA en el desarrollo, y seguir manteniendo manualmente;
- usar IA solo como apoyo en el IDE para pequeñas correcciones;
- limitar la IA a tareas muy concretas y reversibles.

Forzar la IA en este contexto suele ser un error.

b) Software normalizado y bien estructurado

Si el software:

- está bien modularizado,
- tiene arquitectura clara,
- versiones controladas,
- y patrones consistentes,

entonces sí puede beneficiarse del desarrollo asistido por IA, siempre que se haga bajo reglas claras. **Aquí la Fase 2 empieza a tener peso real.**

c) Nuevo software o software estructural

En aplicaciones de gestión, ERPs, plataformas o core systems:

- la IA va a estar integrada en el propio software,
- va a influir en decisiones, procesos y comportamiento,
- y condiciona arquitectura, datos y responsabilidad.

Aquí el sistema de desarrollo tiene que diseñarse desde el inicio para convivir con IA, no para añadirla después.

2. Analizar el contexto organizativo del desarrollo

Especialmente importante en software factories.

Aquí hay que distinguir con claridad:

- desarrollo sobre arquitectura propia,
- desarrollo sobre arquitectura del cliente.

Cuando se trabaja sobre arquitectura del cliente:

- existen restricciones de confidencialidad,
- propiedad intelectual,
- licencias,
- y posibles conflictos legales.

En estos casos:

- no todo el código puede usarse como contexto,
- no todo puede alimentar modelos,
- y no siempre es viable usar herramientas externas.

Definir el sistema de desarrollo sin revisar estos límites es un riesgo legal, no solo técnico.

3. Validar el marco legal y contractual

Antes de definir reglas técnicas, se aclara:

- qué código es reutilizable,
- qué código es confidencial,
- qué acuerdos existen con clientes,
- qué licencias aplican,
- qué datos o conocimiento no pueden salir del perímetro.

Esto condiciona directamente:

- si se pueden usar modelos externos,
- si deben ser modelos locales,
- qué tipo de entrenamiento es posible.

Aquí no se improvisa.

4. Definir el marco de referencia técnico

Solo cuando los límites están claros se define el sistema de desarrollo en sí.

Incluye:

- arquitectura de referencia (qué entendemos por “arquitectura” en este contexto),
- componentes permitidos,
- integraciones oficiales,
- patrones de diseño vigentes,
- versiones exactas de frameworks y librerías,
- reglas de compatibilidad entre versiones.

La arquitectura no es solo código:

- incluye dónde se despliega,
- cómo se ejecuta,
- si es cloud, on-premise o híbrido,
- si hay restricciones de latencia, seguridad o conectividad.

5. Definir cómo entra la IA en el desarrollo

Aquí se concreta cómo se usa la IA, no de forma genérica. **Se define explícitamente:**

- para qué tareas puede usarse,
- en qué tipo de proyectos,
- con qué límites,
- bajo qué supervisión.

No es lo mismo:

- usar IA para generar un CRUD,
- que para modificar un core financiero,
- que para generar código que luego aprenderá.

Cada caso tiene reglas distintas.

6. Anticipar la IA integrada en el propio software

En software que va a llevar IA embebida:

- se anticipa qué tipo de modelos se usarán,
- si serán comerciales, open source o locales,
- dónde se ejecutarán,
- y cómo condicionan la arquitectura.

Esto no es el modelo del IDE, es el modelo que vivirá dentro del sistema. El sistema de desarrollo debe ser compatible con esa realidad futura.

7. Qué se obtiene de la Fase 2

Al finalizar esta fase, la organización sabe:

- qué software puede usar IA y cuál no,
- qué tipo de desarrollo asistido es viable,
- bajo qué límites técnicos y legales,
- qué arquitectura se protege,
- y qué no debe tocarse.

Esto evita errores caros antes de escalar velocidad.

Artefacto clave de la Fase 2

Contrato del sistema de desarrollo asistido por IA Incluye:

- clasificación de aplicaciones,
- reglas de uso de IA por tipo de software,
- límites técnicos y legales,
- arquitectura y versiones de referencia,
- criterios de supervisión,
- exclusiones explícitas.

Este contrato:

- alinea equipos,
- protege conocimiento,
- y evita decisiones implícitas.

Checklist

Definición del sistema de desarrollo asistido por IA Antes de permitir desarrollo asistido a escala, debe existir:

Alcance y límites

☐ Está definido qué software puede usar IA y cuál no.
☐ Está claro si la IA se usa solo como asistente o como parte estructural.
☐ Existen exclusiones explícitas (no ambiguas).
Arquitectura de referencia
☐ Existe una arquitectura de referencia definida.
☐ Están definidos patrones y criterios técnicos obligatorios.
☐ La arquitectura incluye despliegue, ejecución y entorno (cloud, on-prem, híbrido).

Marco legal y contractual

☐ Está claro qué código es propio y reutilizable.
☐ Están definidos límites por confidencialidad y licencias.
☐ Se ha validado el uso de modelos externos o locales.

Uso de IA en desarrollo

☐ Está definido para qué tareas puede usarse IA.
☐ Están definidos los límites por tipo de software.
☐ Existe supervisión explícita en zonas críticas.

Nota sobre excepciones y responsabilidad

☐ Las excepciones al marco definido están permitidas, pero no implícitas.
☐ Toda excepción relevante tiene un **responsable técnico explícito**.
☐ La excepción queda **registrada** junto con su justificación.
☐ Se define si la excepción es:

- puntual,
- temporal,
- estructural.

☐ Existe un criterio claro para **revisar o retirar la excepción** cuando el sistema evoluciona.

Sin este checklist, la IA entra por donde puede, no por donde debe.

La Fase 2 no busca estandarizar por comodidad. Busca diseñar un sistema de desarrollo coherente con la realidad del software y de la organización. ***Sin esta fase: la IA entra por donde puede, no por donde debe. Y eso, a medio plazo, siempre se paga.***

Fase 3 Gobierno del código y del modelo de desarrollo

La Fase 3 establece qué código y qué conocimiento puede usar la IA, bajo qué reglas y con qué garantías de coherencia en el tiempo.

Aquí se asume una realidad básica: el código no vive en un único sitio ni en una única versión, y sin gobierno explícito la IA acabará aprendiendo de lo que no debe.

Esta fase no trata de herramientas. Trata de controlar la fuente del conocimiento técnico.

1. Definir qué repositorios forman parte del sistema gobernado

El primer paso es delimitar qué repositorios entran dentro del perímetro de gobierno.

Normalmente existen:

- repositorios privados,
- múltiples proyectos,
- repositorios compartidos,
- forks históricos,
- y código con distintos niveles de calidad.

Aquí se define explícitamente:

- qué repositorios son referencia válida,
- cuáles quedan fuera,
- y cuáles solo pueden usarse como contexto limitado.

No todo el código de la organización es conocimiento reutilizable.

2. Definir reglas claras de versionado y ramas

Este es uno de los puntos más críticos y más ignorados. En un mismo repositorio puede haber:

- decenas de ramas,
- código experimental,
- hotfixes,
- versiones antiguas,
- refactorizaciones incompletas.

La metodología exige definir explícitamente:

- qué rama es fuente de verdad para:
  - entrenamiento,
  - adaptación,
  - aprendizaje,
  - generación de código.

Ejemplos válidos:

- solo la rama en producción,
- solo main o master,
- una rama específica de referencia,
- una rama consolidada por versión.

Lo importante no es cuál se elija. Lo importante es que no sea ambiguo. La IA no distingue entre estable y provisional si no se le dice.

3. Gobierno del código propio

Aquí se define cómo se trata el código interno. Se establece:

- qué código es válido y actual,
- qué código es legado,
- qué código está obsoleto,
- qué código no debe reutilizarse nunca.

Esto permite:

- evitar que el modelo aprenda decisiones antiguas,
- mantener coherencia arquitectónica,
- y reducir regresiones silenciosas.

El código se convierte explícitamente en dato gobernado.

4. Gobierno del código de frameworks y dependencias

El gobierno no se limita al código propio. Para cada arquitectura se define:

- qué frameworks se usan,
- con qué versiones exactas,
- y qué código fuente de esos frameworks es referencia válida.

Por ejemplo:

- Spring Boot versión concreta,
- NestJS versión concreta,
- Node versión concreta,
- librerías internas homologadas.

No se permite:

- mezclar versiones,
- asumir compatibilidad implícita,
- ni dejar que el modelo "improvise".

El modelo debe aprender solo del ecosistema que realmente existe.

5. Gobierno de la documentación como parte del conocimiento

El código no es suficiente. Para que el desarrollo asistido funcione correctamente, también se gobierna:

- documentación interna,
- guías de uso de componentes,
- normas de seguridad,
- decisiones de diseño,
- documentación oficial de frameworks (versionada).

Toda esa información:

- forma parte del contexto,
- influye en la generación,
- y debe estar alineada con el código.

Código y documentación no pueden evolucionar de forma independiente.

6. Definir qué conocimiento puede usarse para entrenamiento o adaptación

Aquí se toma una decisión crítica:

- qué código y documentación pueden alimentar entrenamiento,
- qué solo pueden usarse como referencia,
- y qué queda excluido.

Esto aplica tanto a:

- adaptación de modelos,
- como a cualquier forma de aprendizaje incremental.

Más contexto no es mejor contexto.

7. Implicaciones sobre el modelo de desarrollo

En esta fase se empieza a tratar el modelo como un componente gobernado, no como una caja negra.

Se define:

- qué modelo o modelos se usan,
- con qué propósito,
- bajo qué contexto,
- y con qué versiones.

Esto no es todavía observabilidad (eso viene después), pero sí es control previo.

Responsabilidades que aparecen en la Fase 3

Aquí sí aparecen responsabilidades nuevas, aunque no necesariamente roles nuevos. Estas responsabilidades suelen recaer en:

- el equipo de arquitectura,
- perfiles con visión transversal del sistema.

Incluyen:

- definir reglas de versionado válidas,
- decidir qué código es referencia,
- proteger coherencia arquitectónica,
- y evitar aprendizaje incorrecto.

En organizaciones grandes puede haber varios responsables. En pymes, una sola persona puede asumirlo. El método no impone estructura, hace explícito lo que debe decidirse.

Qué se obtiene de la Fase 3

Al finalizar esta fase, la organización tiene claro:

- qué código es conocimiento válido,
- qué versiones son referencia,
- qué puede aprender el sistema,
- qué no debe tocarse,
- y cómo evitar degradación progresiva.

Esto reduce drásticamente:

- incoherencias,
- regresiones,
- y dependencia de personas concretas.

Artefactos de la Fase 3

Artefacto 1 — Política de código como dato

- repositorios válidos,
- ramas de referencia,
- clasificación de código,
- exclusiones explícitas.

Artefacto 2 — Marco de versiones y dependencias

- frameworks permitidos,
- versiones exactas,
- código fuente de referencia.

Artefacto 3 — Política de conocimiento reutilizable

- código,
- documentación,
- decisiones de diseño,
- límites de uso.

Checklist

Gobierno del código y del modelo antes de permitir aprendizaje o generación sistemática, debe existir:

Repositorios y versiones

☐ Están definidos los repositorios que forman parte del sistema gobernado.
☐ Está definida la rama de referencia para aprendizaje y generación.
☐ Código experimental y provisional está excluido explícitamente.

Código como dato

☐ El código está clasificado (válido, legado, obsoleto).
☐ Está definido qué código puede reutilizarse como conocimiento.
☐ Está definido qué código no debe usarse nunca como referencia.

Frameworks y dependencias

☐ Están definidas versiones exactas de frameworks y librerías.
☐ Está claro qué código fuente externo es referencia válida.
☐ No se permite mezcla implícita de versiones.

Documentación

☐ La documentación forma parte del contexto gobernado.
☐ Código y documentación están alineados en versiones y criterios.

Nota sobre excepciones y responsabilidad

☐ Las excepciones al marco definido están permitidas, pero no implícitas.
☐ Toda excepción relevante tiene un **responsable técnico explícito**.
☐ La excepción queda **registrada** junto con su justificación.
☐ Se define si la excepción es:

- puntual,
- temporal,
- estructural.

☐ Existe un criterio claro para **revisar o retirar la excepción** cuando el sistema evoluciona.
☐ Está claro quién puede autorizar excepciones al uso de código, librerías o contexto de entrenamiento.

Si este checklist falla, el sistema aprenderá de lo que no debe.

Sin gobierno del código y del modelo, la IA aprende lo que no debe, la arquitectura se degrada y los errores aparecen tarde. Esta fase no ralentiza el desarrollo. Evita que la velocidad rompa el sistema desde dentro.

Fase 4 Observabilidad y control del desarrollo asistido por IA

Una vez que el código se trata como dato (Fase 3), no basta con gobernar qué entra. Es imprescindible observar qué ocurre cuando la IA genera, modifica o propone código.

Esta fase existe para una idea muy concreta. **Si no puedes observar cómo se genera el software, no puedes gobernarlo ni corregirlo.**

1. Qué significa observabilidad en este contexto

Observabilidad no es logging técnico. Aquí significa poder responder, en cualquier momento, a preguntas como:

- ¿Qué decisión tomó la IA?
- ¿Con qué contexto?
- ¿Bajo qué reglas?
- ¿Con qué versión del modelo?
- ¿Qué código generó exactamente?
- ¿Qué acabó entrando en el repositorio?
- ¿Qué se descartó y por qué?

Sin estas respuestas, el sistema puede funcionar, pero no puede auditarse ni evolucionar de forma controlada.

2. Registrar la generación como un evento gobernado

Cada vez que la IA:

- genera código,
- modifica código,
- propone un refactor,
- sugiere una estructura,

eso debe tratarse como un evento del sistema, no como algo efímero del IDE.

Ese evento debe registrar, como mínimo:

- contexto utilizado,
- reglas activas,
- versión del modelo,
- tipo de acción (generar, modificar, sugerir),
- resultado generado.

Esto no obliga a aceptar ese código. Pero sí permite entender qué pasó.

3. Relacionar generación, decisión y resultado final

Uno de los puntos más potentes de esta fase es que permite cerrar el ciclo:

- qué generó la IA,
- qué se aceptó,
- qué se modificó,
- y qué acabó en producción.

Esto permite:

- confrontar decisiones con resultados reales,
- detectar patrones de error,
- identificar desviaciones de arquitectura,
- y entender degradaciones progresivas.

Aquí es donde el sistema empieza a aprender de verdad, no de forma ciega.

4. Auditar y monitorizar el comportamiento del sistema de desarrollo

Al tener trazabilidad completa, se pueden construir mecanismos de:

- auditoría técnica,
- análisis de coherencia,
- detección de anomalías,
- comparación entre versiones del modelo,
- validación de cumplimiento de reglas.

No para fiscalizar personas, sino para evaluar el comportamiento del sistema. La IA y los agentes deben observarse igual que cualquier otro componente crítico.

5. Extraer conocimiento y normas a partir de la observación

La observabilidad no es solo defensiva. Es una fuente de mejora. A partir de los datos observados se puede:

- detectar errores recurrentes,
- identificar reglas mal definidas,
- ajustar contexto y versiones,
- mejorar el sistema de desarrollo,
- definir nuevas restricciones o guías.

Esto permite evolución basada en evidencia, no en sensaciones.

6. Diferenciar claramente observabilidad de aprendizaje

Muy importante: observar no implica aprender automáticamente. En esta fase:

- se recopila,
- se analiza,
- se compara.

El aprendizaje vendrá después, en la Fase 5, y solo cuando esté explícitamente autorizado.

Esta separación evita uno de los errores más comunes: confundir feedback con autoaprendizaje descontrolado.

Qué se obtiene de la Fase 4

Al finalizar esta fase, la organización tiene:

- trazabilidad completa del desarrollo asistido,
- capacidad de auditar decisiones técnicas,
- visibilidad real del comportamiento de la IA,
- evidencia objetiva para corregir desviaciones,
- y control sobre cómo evoluciona el sistema.

Sin esta fase, el sistema puede ser rápido. Pero es opaco.

Artefactos de la Fase 4

Artefacto 1 — Registro de eventos de generación

- decisiones,
- contexto,
- reglas,
- modelo,
- resultados.

Artefacto 2 — Mapa de trazabilidad decisión–código

- qué se generó,
- qué se aceptó,
- qué llegó al repositorio.

Artefacto 3 — Métricas de coherencia y desviación

- patrones incorrectos,
- regresiones,
- incompatibilidades,
- degradación progresiva.

Checklist

Observabilidad y trazabilidad antes de permitir aprendizaje o automatización avanzada, debe existir:

Registro de generación

☐ Cada generación de código se registra como evento.
☐ Se guarda contexto, reglas y versión del modelo.
☐ Se diferencia código generado, aceptado y descartado.

Trazabilidad

☐ Puede reconstruirse por qué se generó un código.
☐ Puede compararse comportamiento entre versiones.
☐ Puede auditarse una decisión técnica a posteriori.

Monitorización

☐ Existen métricas de coherencia arquitectónica.
☐ Se detectan desviaciones y regresiones.
☐ Los errores no dependen solo de revisiones humanas.

Nota sobre excepciones y responsabilidad

☐ Las excepciones al marco definido están permitidas, pero no implícitas.
☐ Toda excepción relevante tiene un **responsable técnico explícito**.
☐ La excepción queda **registrada** junto con su justificación.
☐ Se define si la excepción es:

- puntual,
- temporal,
- estructural.

☐ Existe un criterio claro para **revisar o retirar la excepción** cuando el sistema evoluciona.
☐ Existen mecanismos de revisión del código generado o modificado con IA, más allá de la ejecución de tests técnicos.
☐ Los resultados de estas revisiones (automatizadas o asistidas por IA) pueden relacionarse con el código final y las decisiones tomadas.

Sin observabilidad, no hay control ni aprendizaje fiable.

Tratar el software como dato *implica observar cómo se produce, no solo almacenarlo. La IA no puede ser una caja negra dentro del desarrollo. Debe ser observable, auditable y comparable.*

Esta fase no ralentiza. Permite corregir antes de que el problema sea estructural.

Fase 5 Aprendizaje gobernado y evolución del sistema

La Fase 5 no trata de **"hacer que la IA aprenda más".** Trata de permitir que el sistema evolucione sin perder control.

Esta fase solo tiene sentido cuando:

- el código ya se gobierna como dato,
- el sistema es observable,
- las decisiones son trazables,
- y las responsabilidades están claras.

Por eso es la culminación del método.

1. El objetivo real de esta fase

El desarrollo de software ha cambiado de paradigma. El código ya no es solo:

- algo que se escribe,
- se compila,
- y se mantiene.

Es:

- un activo vivo,
- una fuente de conocimiento,
- y parte del sistema que aprende y evoluciona.

La Fase 5 existe para responder a una necesidad muy concreta: Cómo permitir que el sistema mejore con el tiempo sin que ese aprendizaje se vuelva opaco, incontrolable o peligroso.

2. Aprender es inevitable, hacerlo sin control no

Los frameworks evolucionan, componentes cambian, modelos mejoran y las necesidades de negocio se transforman. Si el sistema no evoluciona:

- se queda obsoleto,
- se vuelve rígido,
- y acaba rompiéndose.

Pero si aprende sin control:

- degrada arquitectura,
- introduce incoherencias,
- y genera dependencia invisible.

El aprendizaje no es opcional. El aprendizaje descontrolado sí lo es.

3. Qué significa "aprendizaje gobernado"

Aprendizaje gobernado significa que nada aprende por accidente. Se define explícitamente:

- qué puede aprender el sistema,
- cuando puede aprender,
- a partir de qué información,
- bajo qué validación humana,
- y con qué capacidad de reversión.

Aprender no es un efecto colateral del uso. Es una decisión de diseño.

4. El papel central del desarrollador (Human-in-the-Loop)

Esta fase deja algo muy claro: el desarrollador no desaparece. Su rol cambia, pero se vuelve más crítico que nunca.

El desarrollador:

- supervisa decisiones,
- valida resultados,
- detecta errores de criterio,
- aporta contexto que la IA no tiene,
- y decide qué aprendizaje es válido.

La IA no sustituye criterio técnico. Lo amplifica cuando está bien diseñado.

El Human-in-the-Loop no es un requisito normativo. Es una condición técnica para que el sistema no se degrade.

5. Fuentes válidas de aprendizaje

El sistema no aprende de todo, aprende de lo que se decide. Fuentes habituales:

- decisiones humanas aceptadas o rechazadas,
- correcciones manuales al código generado,
- feedback explícito de desarrolladores,
- incidencias detectadas en producción,
- cambios de versión deliberados,
- ajustes arquitectónicos validados.

Todo aprendizaje:

- parte de eventos observados (Fase 4),
- y pasa por validación humana.

6. Evolución frente a obsolescencia

Esta fase permite que el sistema:

- incorpore nuevas versiones de frameworks,
- adapte patrones cuando cambia el contexto,
- mejore su comportamiento con el tiempo,
- sin arrastrar decisiones antiguas.

La evolución no es automática. Es progresiva y reversible. Si algo empeora:

- se detecta,
- se analiza,
- y se revierte.

7. Cambios de roles y responsabilidades (sin ruptura)

Aquí se consolida el cambio de paradigma:

- menos foco en escribir código líneo a línea,
- más foco en diseño, validación y supervisión,
- más responsabilidad en definir criterios,
- menos dependencia de memoria individual.

No se elimina al desarrollador. Se elimina la fragilidad del sistema.

Qué se obtiene al completar la Fase 5

Cuando esta fase está bien implantada:

- el sistema mejora con el tiempo,
- la velocidad no degrada la calidad,
- la IA deja de ser una caja negra,
- los cambios son explicables,
- y el conocimiento no se pierde con la rotación.

La organización aprende sin perder control.

Artefactos de la Fase 5

Artefacto 1 Política de aprendizaje gobernado

- qué aprende,
- cuando,
- quién valida,
- cómo se versiona,
- cómo se revierte.

Artefacto 2 Ciclo de feedback humano

- decisiones aceptadas/rechazadas,
- correcciones,
- aportaciones del equipo.

Artefacto 3 Registro de evolución del sistema

- cambios de criterio,
- mejoras,
- regresiones,
- causas identificadas.

Checklist

Aprendizaje gobernado y evolución, antes de permitir que el sistema evolucione automáticamente, debe existir:

Decisión explícita de aprendizaje

☐ Está definido qué puede aprender el sistema.
☐ Está definido cuándo puede aprender.
☐ Está definido qué eventos alimentan el aprendizaje.

Validación humana

☐ Existe validación humana antes de incorporar aprendizaje.
☐ El desarrollador actúa como supervisor (human-in-the-loop).
☐ El feedback humano se registra como señal válida.

Evolución y reversión

☐ El aprendizaje está versionado.
☐ Puede revertirse un cambio si degrada el sistema.
☐ La evolución no rompe compatibilidad ni arquitectura.

Nota sobre excepciones y responsabilidad

☐ Las excepciones al marco definido están permitidas, pero no implícitas.
☐ Toda excepción relevante tiene un **responsable técnico explícito**.
☐ La excepción queda **registrada** junto con su justificación.
☐ Se define si la excepción es:

- puntual,
- temporal,
- estructural.

☐ Existe un criterio claro para **revisar o retirar la excepción** cuando el sistema evoluciona.
☐ Está claro quién valida qué decisiones humanas pueden incorporarse como aprendizaje del sistema.
☐ Los resultados de revisiones de calidad y validación técnica se utilizan como entrada para ajustar reglas, contexto o aprendizaje del sistema.

Si este checklist no se cumple, el sistema se degrada aunque "aprenda".

El objetivo final del AI-First aplicado al desarrollo no es automatizar personas. *Es diseñar sistemas que evolucionen sin perder criterio humano. La IA puede generar, proponer y ejecutar. Pero el juicio sigue siendo humano. Sin esta fase, el sistema se congela o se degrada. Con ella, el sistema se adapta sin romperse.*

Una visión práctica del desarrollo AI-First

La IA está cambiando el desarrollo de software. Eso no es discutible.

Está cambiando la forma de escribir código, la velocidad a la que se construyen soluciones y la manera en que se organizan los equipos. También está introduciendo nuevos perfiles, nuevas dinámicas y posibilidades técnicas que hace pocos años no existían.

Pero **hay una confusión peligrosa en el discurso actual**: *confundir automatización de tareas con desaparición de responsabilidades*.

El hecho de que hoy podamos generar código con agentes, auditarlo con otros agentes y acelerar procesos completos no elimina la necesidad de entender el sistema, de gobernarlo y de responder por él cuando está en producción. Al contrario: a mayor potencia, mayor necesidad de control.

Los modelos actuales (por muy avanzados que sean) siguen siendo modelos entrenados mayoritariamente sobre repositorios públicos, con todo lo que eso implica: código de calidad desigual, patrones contradictorios, decisiones heredadas y soluciones válidas fuera de contexto. En escenarios simples, esto puede ser suficiente. En sistemas complejos o críticos, no lo es.

La IA aplicada al desarrollo aporta una capacidad extraordinaria. Pero la potencia sin control no acelera el progreso: acelera los errores.

Por eso esta metodología no intenta frenar el uso de IA, ni idealizar formas de trabajo pasadas. Lo que hace es ordenar algo que, en realidad, siempre ha existido: la necesidad de arquitectura, de normalización, de gobierno y de responsabilidad cuando el software es importante.

Muchas de las prácticas que aquí se describen no son nuevas. Son sentido común aplicado durante años en sistemas que debían mantenerse, evolucionar y escalar. La diferencia es que ahora el código deja de ser solo un resultado y pasa a tratarse como dato: se reutiliza, se analiza, se audita y se convierte en parte activa del conocimiento del sistema. Y eso exige gobierno.

También es importante aclarar algo: **la adopción de IA no implica necesariamente una reducción de equipos**. Implica una reorganización del trabajo. En equipos sobredimensionados puede haber ajustes. En equipos bien dimensionados, lo que cambia es el foco: menos esfuerzo en tareas mecánicas y más responsabilidad en supervisión, validación, diseño y control.

El desarrollador no desaparece.
El arquitecto no desaparece.
El factor humano no desaparece.

Lo que desaparece es la excusa de improvisar sin consecuencias.

El verdadero valor de la IA no está en usar modelos cada vez más grandes que saben un poco de todo. Está en la especialización, en la adaptación al contexto real de una organización, en el uso de modelos entrenados o ajustados a una arquitectura, unos patrones y unas decisiones claras. Y eso, lejos de ser inasumible, es perfectamente viable cuando se aborda con criterio y método.

Decir que "el código ya no tiene valor" es una simplificación que ignora la realidad de los sistemas en producción. El valor no está en escribir líneas por escribirlas, pero el valor del código aumenta cuando se convierte en un activo gobernado, verificable y reutilizable dentro de un sistema bien diseñado.

Esta metodología no pretende imponer una forma única de trabajar ni ofrecer recetas universales. Pretende algo más modesto y difícil: ayudar a que la adopción de IA en el desarrollo no rompa aquello que luego hay que mantener.

Porque cuando la IA deja de ser novedad y pasa a ser infraestructura, lo que marca la diferencia no es quién genera más rápido, sino quién mantiene el control.

Y eso, hoy más que nunca, sigue siendo una responsabilidad humana.

Nota final

Este documento nace de experiencia práctica en diseño, desarrollo y operación de sistemas de software durante más de 25 años, aplicando low-code, generación de código, arquitectura de plataformas y, en los últimos años, IA integrada en el ciclo de desarrollo.

La metodología está pensada para ser leída, cuestionada y adaptada a distintos contextos.

Para conversaciones técnicas, contraste de enfoques o acompañamiento en su aplicación:

Manuel J. González (CTO- Codeflowx AIS)
LinkedIn: www.linkedin.com/in/manueljgonzalezlopez
Email: mgonzalez@codeflowx.com