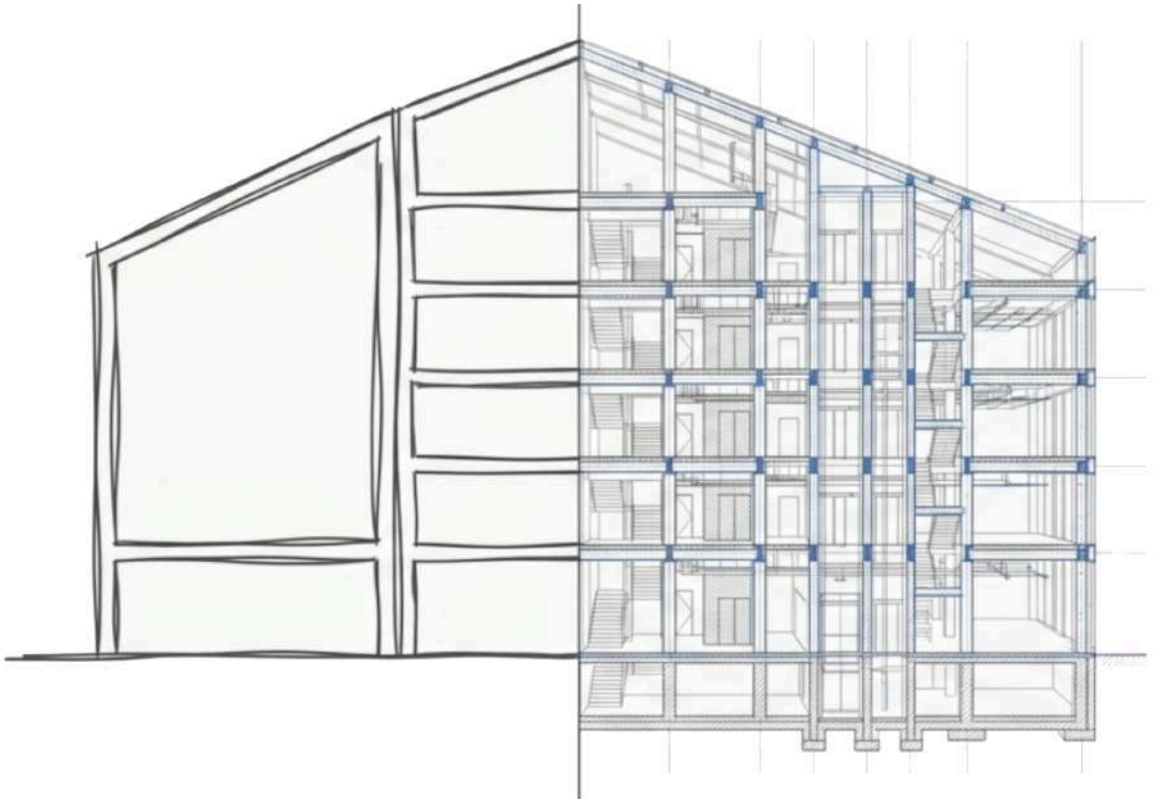
# SDD – Spec Driven Development

## Cuando el código es la consecuencia

## Desarrollo dirigido por SDD en equipos ágiles con IA



## Guía Didáctica

Versión 1.0 — Abril 2026

# SDD Spec Driven Development

*Cuando el código es la consecuencia*
*Desarrollo dirigido por SDD en equipos ágiles con IA*

Versión: 1.0 – abril 2026

© Autor: Juan Palacio Asistente de IA: Claude Opus 4.6 (Anthropic)

Edita: Scrum Manager (scrummanager.com)



*Los contenidos de esta guía didáctica están sujetos a revisión y actualización.*

*Consulta siempre la versión más reciente en scrummanager.com.*

# Tabla de contenidos

# Nota al lector

A lo largo de esta guía se usan algunos términos del oficio en su forma inglesa original cuando su traducción forzaría la lectura o no está asentada en la profesión: spec, prompt, commit, pull request, vibe coding, brownfield, greenfield, framework, waterfall, stakeholder, pipeline. Cuando un término tiene equivalente natural en español (informe, admitir, normalmente, kit de herramientas, traspaso, heredado), se usa la forma castellana.

El término *spec* se trata en femenino, por analogía con *la especificación*: «una spec», «la spec», «las specs».

# Formación y acreditación

Los contenidos de esta guía están disponibles en Skill Arena, la plataforma de autoformación y entrenamiento de Scrum Manager. Allí puedes realizar ejercicios de práctica, refuerzo y evaluación, y también obtener un diploma acreditativo del conocimiento adquirido. Como la aplicación de SDD con IA evoluciona rápidamente, Skill Arena mantiene el contenido actualizado, por lo que es la referencia recomendada para consultar la versión más reciente.

# PARTE I — El cambio de paradigma

## 1. Del vibe coding al desarrollo dirigido por especificación

### El punto de partida: la programación conversacional

El 2 de febrero de 2025, Andrej Karpathy, cofundador de OpenAI y exdirector de IA en Tesla, publicó un mensaje en X que bautizó un fenómeno que ya estaba en marcha. Lo llamó **vibe coding**, y lo definió como el acto de «entregarse completamente a las vibes, abrazar las exponenciales y olvidar que el código existe». En la práctica describe una forma de trabajar con asistentes de IA: tú le dices lo que quieres en lenguaje natural, el modelo genera el código, tú pruebas a ver si funciona, y si algo falla le pegas el error al modelo hasta que salga del paso.

El tuit viralizó un modo de programar que ya describían los datos. En la encuesta de GitHub de 2023, el 92 % de los desarrolladores profesionales en Estados Unidos declaraba usar herramientas de IA en su trabajo diario. En marzo de 2025, Garry Tan, CEO de Y Combinator, confirmó en una entrevista con CNBC que en aproximadamente el 25 % de las startups de su cohorte de invierno de 2025 el 95 % del código estaba escrito por IA. El fenómeno no era marginal.

El vibe coding no es solo una forma de trabajar más rápido. Es una propuesta metodológica: el desarrollador describe lo que quiere, la IA genera el código, y el resultado se acepta si funciona. No hay diseño previo, no hay especificación formal, no hay separación entre lo que se quiere y cómo se construye. El prompt es la intención, el código es la respuesta, y la iteración conversacional es el método.

Y funciona. Para prototipos, para explorar ideas, para herramientas internas rápidas, para todo lo que es más importante llegar pronto que llegar bien, el vibe coding es extraordinariamente productivo. No hay que quitarle ese mérito.

### Donde las vibes dejan de funcionar

El problema no es el vibe coding en sí. El problema es que tiene un techo, y ese techo aparece antes de lo que la mayoría espera.

En julio de 2025, la organización de investigación METR publicó un ensayo controlado aleatorizado con 16 desarrolladores experimentados trabajando sobre sus propios repositorios open source, algunos con más de un millón de líneas de código. El resultado fue contraintuitivo: cuando usaban herramientas de IA, los participantes tardaban un 19 % más en completar sus tareas. Pero creían haber ido un 20 % más rápido. La percepción de velocidad no coincidía con la realidad medida. Ni siquiera después de experimentar el frenazo los desarrolladores eran capaces de verlo.

Lo que ocurre es fácil de reconstruir. Pides a un asistente que genere 500 líneas repartidas en 12 ficheros. Treinta segundos después tienes el código delante y te sientes productivo: apareció instantáneamente. Pero entonces empieza la revisión. Un primer error. Luego otro. ¿Cuántos más habrá enterrados? ¿Merece la pena arreglarlo o tirar y empezar de nuevo? Dos horas después, la cabeza está agotada y todavía no tienes certeza de que la implementación sea correcta.

Este es el problema de la carga cognitiva del vibe coding. Cuando escribes código tú mismo, tu memoria de trabajo retiene el contexto de lo que acabas de escribir. Es fresco y es tuyo. Cuando revisas código generado por otro —sea una persona o una IA—, la carga de comprensión es fundamentalmente distinta. Y esa carga se acumula con cada iteración.

Los datos sobre calidad del código generado han ido apareciendo. El *2025 GenAI Code Security Report* de Veracode, que analizó 80 tareas de codificación en más de 100 modelos de lenguaje, encontró que en el 45 % de los casos el modelo introducía alguna vulnerabilidad del OWASP Top 10 cuando podía elegir entre una solución segura y una insegura. El informe *AI Copilot Code Quality Research* de GitClear, sobre 211 millones de líneas cambiadas entre 2020 y 2024, mostró que el código copiado y pegado pasó del 8,3 % al 12,3 % del total de cambios, mientras que el código refactorizado (líneas movidas) cayó del 25 % a menos del 10 %. En 2024, por primera vez, el código duplicado superó al código refactorizado.

Daniel Stenberg, mantenedor del proyecto cURL, llegó en enero de 2026 a cerrar su programa de *bug bounty* en HackerOne, incapaz de sostener la marea de informes de seguridad fabricados por IA —convincentes en apariencia, vacíos al examinarlos— que estaban ahogando a su pequeño equipo de mantenedores voluntarios.

## La respuesta: *Spec-Driven Development*

Spec-Driven Development, desarrollo dirigido por especificación, en adelante SDD, es la respuesta metodológica a estos problemas. Su premisa es sencilla: en lugar de saltar directamente al código, se produce una serie de documentos de especificación que definen qué se va a construir, cómo se va a construir y qué pasos se van a seguir. Solo después de que esas specs se han revisado y aprobado se pasa a la implementación.

SDD no es una herramienta. No es un framework. No es un plugin que se instala. Es una metodología —una forma de organizar el trabajo— que puede implementarse con diferentes herramientas y que se adapta a diferentes contextos. Esto conviene entenderlo desde el principio, porque el ecosistema de herramientas alrededor de SDD crece deprisa y es fácil confundir el método con el instrumento.

La definición más clara la ofreció Birgitta Böckeler, *Distinguished Engineer* en Thoughtworks, en el artículo *Understanding Spec-Driven Development: Kiro, spec-kit, and Tessl* publicado en el sitio de Martin Fowler (octubre de 2025): «Spec-driven

development significa escribir una spec antes de escribir código con IA». Bajo esa definición aparentemente simple hay un cambio profundo en la relación entre el profesional y la herramienta. Donde el vibe coding trata al agente de IA como un interlocutor conversacional al que se le van dando indicaciones sobre la marcha, SDD trata al agente como un ejecutor que recibe un contrato claro de lo que debe producir.

Böckeler propone una taxonomía de tres niveles que ayuda a entender las diferentes intensidades de adopción:

- **Spec-first**: se escribe la spec antes de codificar, se usa para la tarea en curso y se descarta al terminar. Es el nivel más básico y el punto de entrada natural.
- **Spec-anchored**: la spec se mantiene después de completar la tarea y se usa para la evolución y el mantenimiento de la funcionalidad. Las specs pasan a ser documentación viva del sistema.
- **Spec-as-source**: la spec es el artefacto principal y permanente. Solo se edita la spec; nunca se toca el código directamente. El código se regenera desde la spec cada vez.

La mayoría de equipos y herramientas actuales operan en el nivel spec-first. Algunos aspiran a spec-anchored. Solo unos pocos experimentan con spec-as-source. Saber en qué nivel se trabaja ayuda a calibrar las expectativas y a elegir herramientas.

## Lo que SDD no es

Antes de avanzar, conviene despejar malentendidos frecuentes.

SDD no es documentar después de construir. El orden importa: la spec precede a la implementación. Un documento que describe código ya existente puede ser útil como documentación, pero no es SDD.

SDD no es escribir requisitos exhaustivos al estilo de los pliegos de condiciones clásicos. No hablamos de documentos de 200 páginas que intentan prever cada detalle antes de escribir una sola línea. Como veremos en la Parte III, una buena spec para un agente de IA es inteligente, no larga.

SDD no requiere abandonar la agilidad. De hecho —y esta es la tesis central de esta guía—, SDD es lo que ocurre cuando aplicas principios ágiles a un equipo donde parte de los constructores son agentes de IA. Pero este argumento merece su propio capítulo.

## Un ejemplo que acompañará la guía

Para hacer concretos los conceptos que siguen, imaginemos un mismo caso que iremos retomando a lo largo de la guía: una plataforma de gestión documental para despachos profesionales quiere añadir un **sistema de notificaciones multicanal** que avise a los usuarios cuando cambia un documento que tienen asignado. El sistema debe poder

enviar la notificación por correo electrónico, por notificación push en la aplicación móvil y por mensaje dentro de la propia plataforma, respetando las preferencias de cada usuario. Volveremos a este ejemplo en las fases de requisitos, diseño, tareas y revisión.

## 2. La spec como artefacto primario del desarrollo

### Un cambio de identidad profesional

Hay una frase de Addy Osmani —ingeniero de Google, actualmente director en Google Cloud AI y autor de la referencia más citada sobre specs para agentes— que captura la esencia del cambio: «La IA no es el cuello de botella. Tu spec lo es».

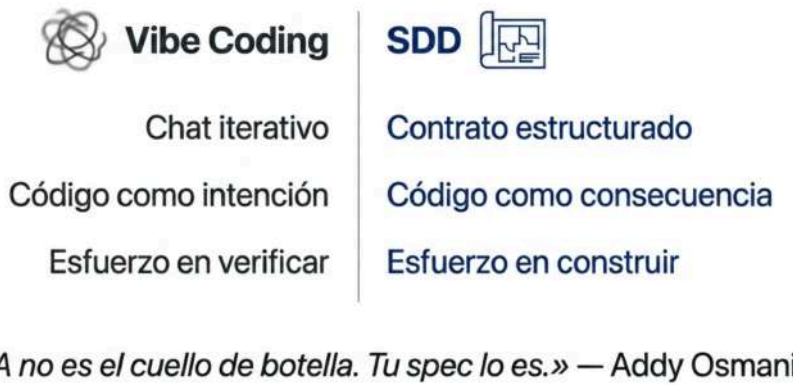



***Imagen 1.*** *La spec como artefacto primario del desarrollo*

Esta frase merece detenerse. Durante décadas, la competencia central de un profesional del software ha sido escribir código. Todo el ecosistema profesional —formación, certificaciones, entrevistas de trabajo, herramientas— se ha construido alrededor de esa competencia. Saber programar, conocer un lenguaje, dominar un framework: eso era lo que te hacía valioso.

SDD propone que esa competencia se está desplazando. No desaparece —entender código sigue siendo necesario para revisar, evaluar y tomar decisiones de arquitectura—, pero deja de ser el acto central del trabajo diario. Lo que ahora te hace productivo es tu capacidad de especificar con claridad qué quieres construir, por qué, bajo qué restricciones y con qué criterios de éxito.

Para un product owner o un product architect, esto puede sonar natural: «eso es lo que yo hago, definir qué hay que construir». Y en parte es cierto. Pero SDD eleva la precisión requerida de esa definición a un nivel que la mayoría de product owners no suele emplear. No basta con decir «como usuario, quiero ver mi actividad reciente». Un agente de IA con esa instrucción tomará docenas de decisiones implícitas: qué

tecnología de base de datos usar, qué periodo de tiempo considerar «reciente», cómo paginar los resultados, qué hacer si no hay actividad. Cada decisión no especificada es una decisión que el agente toma por ti, y cada decisión que el agente toma por ti es un punto potencial de divergencia entre lo que querías y lo que obtienes.


***Imagen 2.*** *Cambio de identidad profesional*

## El código como consecuencia

Para entender el cambio que propone SDD, ayuda una analogía con la arquitectura de edificios. Un arquitecto no coloca ladrillos. Diseña planos que otros ejecutan. La calidad del edificio depende enormemente de la calidad de esos planos: si son ambiguos, el albañil improvisa; si son precisos, el resultado es predecible. El plano es el artefacto primario; el edificio es la consecuencia.

En el desarrollo con IA, la spec es el plano y el código es el edificio. El profesional que diseña la spec está haciendo el trabajo de alto valor: decidir qué se construye, cómo se estructura, qué límites tiene, qué significa que esté terminado. El agente de IA que genera el código está ejecutando ese diseño.

Esto no significa que el código no importe. Un edificio puede tener planos excelentes y una ejecución desastrosa. El código hay que revisarlo, probarlo, validarlo. Pero la dirección de la cadena causal cambia: la spec produce el código, no al revés. Y cuando algo falla, la primera pregunta no es «¿qué bug tiene el código?» sino «¿qué no especificamos bien?».

## Qué cambia para cada rol

El desplazamiento del código a la spec como artefacto primario afecta a cada rol del equipo de forma diferente:

**Para el product owner / product architect**, la exigencia de precisión sube. Ya no basta con escribir una historia de usuario que un desarrollador humano interpretará con

sentido común y experiencia de dominio. El agente de IA es literal: hará exactamente lo que le digas, y si no le dices algo, lo inventará. Los criterios de aceptación pasan de ser una guía para la conversación a ser un contrato de ejecución.

**Para el desarrollador / product builder**, la competencia central se desplaza del lenguaje de programación al diseño de specs y la orquestación de agentes. No deja de necesitar conocimientos técnicos —los necesita para escribir buenas specs técnicas y para evaluar el output del agente—, pero la proporción de tiempo dedicada a escribir código a mano disminuye drásticamente. Un desarrollador que trabaja con SDD se parece más a un tech lead —un desarrollador sénior que dirige y revisa el trabajo del equipo— que a un programador que escribe.

**Para el Scrum master / Agile enabler**, aparecen nuevas responsabilidades. Las puertas de aprobación entre fases son puntos de inspección y adaptación que necesitan facilitación. Hay que asegurar que el proceso de especificación no se convierta en un cuello de botella burocrático y que el equipo mantenga el ritmo iterativo.

**Para los stakeholders (directivos, responsables de área, usuarios clave)**, hay una ganancia inesperada: las specs son legibles por personas no técnicas en una medida que el código nunca lo fue. Una spec bien escrita describe comportamientos, criterios de éxito y restricciones en un lenguaje que un directivo o un usuario puede evaluar. El código siempre fue una caja negra para los no técnicos; la spec abre esa caja.

### La spec como contrato

Hay un matiz importante que distingue a SDD de la simple documentación previa. En SDD, la spec funciona como un contrato entre las partes del sistema, no como un documento de mando que se entrega al agente.

La apuesta subyacente es: contratos explícitos en la granularidad correcta permiten que el desarrollo dirigido por IA a escala de equipo avance más rápido, no más lento. Los humanos aprueban el contrato en las puertas de fase, los agentes lo ejecutan, y lo que se entrega es el código que cumple el contrato.

## 3. Los tres problemas estructurales del vibe coding

El vibe coding no falla por accidente. Falla por tres razones estructurales que se refuerzan mutuamente a medida que el proyecto crece.


***Imagen 3.*** *Problemas estructurales del vibe coding*

## Requisitos implícitos

Cuando escribes «construye un sistema de autenticación», estás comunicando una intención, no una spec. El agente de IA necesita tomar decenas de decisiones para convertir esa intención en código: ¿qué método de autenticación? ¿Contraseña, OAuth, biometría? ¿Qué política de contraseñas? ¿Cuántos intentos fallidos antes de bloquear la cuenta? ¿Hay verificación en dos pasos? ¿Cómo se recupera una contraseña olvidada?

Cada una de estas preguntas que no se responde explícitamente es un requisito implícito que el agente resuelve por su cuenta. Y lo resuelve con la estrategia que le resulta más probable dada su base de entrenamiento, no con la que es más adecuada para tu contexto específico.

El coste de los requisitos implícitos es insidioso porque no produce errores inmediatos. El código funciona. Los tests pasan. Pero las decisiones que tomó el agente pueden no ser las que tú habrías tomado, y eso genera una divergencia acumulativa que se descubre tarde y es cara de corregir.

En nuestro sistema de notificaciones multicanal, un prompt del tipo «implementa las notificaciones por correo» lleva al agente a decidir, entre otras cosas: qué servicio SMTP usar, qué formato HTML adoptar, cómo gestionar los fallos de entrega, si incluir un enlace de baja, si agrupar notificaciones del mismo usuario, en qué idioma enviarlas. Ninguna de estas decisiones aparece en el prompt. Todas aparecen en el código.

## Contaminación del contexto

Los agentes de IA trabajan dentro de una ventana de contexto —la cantidad de texto que pueden «tener en mente» simultáneamente—. A medida que la conversación avanza, esa ventana se llena con decisiones pasadas, código generado, correcciones, cambios de

dirección. Cuando la ventana se satura, el agente necesita resumir la conversación previa para liberar espacio, y cada resumen pierde información.

El efecto es lo que se conoce como *context decay* (degradación del contexto): las decisiones tomadas al inicio de la sesión se desvanecen gradualmente. El agente empieza a contradecirse, a reintroducir problemas ya resueltos, a perder de vista restricciones que se establecieron horas antes.

Incluso antes de saturar la ventana, la calidad degrada. La investigación sobre la llamada *maldición de las instrucciones* —que desarrollamos en detalle en el capítulo 14— muestra que, cuantas más instrucciones se apilan en un prompt, más cae la probabilidad de que el modelo cumpla cada una de ellas individualmente.

Una sesión de vibe coding prolongada es, en esencia, una instrucción que crece indefinidamente. Cada iteración añade contexto, cada corrección añade matices, cada cambio de dirección añade complejidad. El agente con el que estás hablando en la hora tres es funcionalmente peor que el agente de la hora uno.

## Deriva

La deriva es quizás el problema más peligroso porque es el más invisible. En el vibe coding, cada sesión de trabajo con el agente deja un rastro de decisiones: qué patrón de arquitectura se eligió, cómo se resolvió un caso límite, cómo se estructuraron los datos. Esas decisiones no están documentadas en ninguna parte; vivían en el contexto de la sesión. Cuando inicias una nueva sesión, ese contexto desaparece.

El resultado es que diferentes sesiones de trabajo pueden tomar decisiones inconsistentes entre sí. Ninguna está «mal» individualmente, pero el conjunto pierde coherencia. Los patrones que existían por una razón se reemplazan silenciosamente por lo que el modelo decide hacer ese día. La arquitectura se ablanda.

Ese es el riesgo real. No la alucinación que explota inmediatamente —esa la detectas en cinco minutos—. Son los seis meses de código que funciona bien, pasa todos los tests y poco a poco deja de tener sentido como conjunto. La misma arquitectura informe —lo que Foote y Yoder, en un artículo clásico de 1997, llamaron *big ball of mud*— que los equipos de ingeniería han construido siempre, solo que ahora a la velocidad de la IA.

## El coste real: no es técnico, es cognitivo

Los tres problemas anteriores tienen una dimensión técnica, pero su impacto real es cognitivo. El desarrollador que trabaja con vibe coding a escala dedica una cantidad creciente de energía mental a recordar decisiones de sesiones anteriores, a verificar que el agente no ha contradicho algo que ya estaba establecido, a revisar código generado

sin contexto de por qué se generó así, y a deshacer trabajo que «funciona» pero no es lo que se quería.

Cada una de estas actividades consume memoria de trabajo —el recurso cognitivo más escaso y valioso de un profesional del conocimiento—. La paradoja del vibe coding es que la herramienta diseñada para ahorrarte esfuerzo termina generando más esfuerzo del que ahorra, pero de un tipo diferente y más agotador: esfuerzo de verificación en lugar de esfuerzo de construcción.

SDD ataca este problema directamente. Al separar la planificación de la ejecución, al explicitar las decisiones en documentos revisables, al establecer puntos de verificación estructurados, SDD reduce la carga cognitiva de la revisión porque el código generado ya tiene un contrato contra el que evaluarse. No revisas «a ver qué hizo el agente»; revisas si lo que hizo coincide con lo que especificaste.

## 4. SDD y agilidad: ¿el regreso de waterfall?

### La pregunta incómoda

Cualquier profesional ágil que lea sobre SDD pensará, tarde o temprano, lo mismo: «Esto suena a waterfall».

La observación no es trivial. En noviembre de 2025, Marmelab publicó un artículo titulado *Spec-Driven Development: The Waterfall Strikes Back* que expresa la objeción directamente. El argumento tiene peso: SDD propone fases secuenciales (requisitos, diseño, tareas, implementación), puertas de aprobación entre fases y documentación detallada antes de escribir código. Todo esto suena a lo que el Manifiesto Ágil criticó hace más de dos décadas.

Ignorar la tensión sería deshonesto. Resolverla es lo que permite que SDD funcione en un contexto ágil.

### Lo que dice el Manifiesto

El *Manifiesto Ágil* establece cuatro valores:

1. Individuos e interacciones sobre procesos y herramientas.
2. Software funcionando sobre documentación extensiva.
3. Colaboración con el cliente sobre negociación contractual.
4. Respuesta ante el cambio sobre seguir un plan.

El segundo valor —«software funcionando sobre documentación extensiva»— es el que parece chocar con SDD. Si SDD propone escribir documentación detallada antes de producir software, ¿no estamos invirtiendo la prioridad?

La objeción es razonable, pero se basa en una lectura incompleta. El *Manifiesto* dice «sobre», no «en lugar de». Reconoce valor en la documentación; simplemente prioriza el software funcionando. Y añade la coletilla habitualmente ignorada: «esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda».

La pregunta correcta no es «¿SDD documenta?» sino «¿la documentación que produce SDD contribuye a obtener software funcionando de mayor calidad, más rápido?». Si la respuesta es sí, no hay contradicción con el *Manifiesto*.

Pero hay una razón más profunda por la que la objeción no aplica a SDD del mismo modo que aplicaba a waterfall. Y tiene que ver con la naturaleza misma de la documentación.

## Una nueva categoría: documentación operativa

Hasta la aparición de la IA en el desarrollo de software, la documentación de un proyecto pertenecía a dos categorías:

- **Documentación informativa**: manuales de usuario, guías de arquitectura, ficheros README, comentarios en el código. Su función es transmitir conocimiento entre personas. Alguien la escribe para que otra persona la lea y comprenda el sistema. Es valiosa, pero su relación con el código es indirecta: informa sobre el software, no lo produce.
- **Documentación administrativa**: actas de reuniones, informes de estado, registros de cambios, documentos de conformidad regulatoria. Su función es satisfacer requisitos del proceso, de la organización o del marco normativo. Es necesaria en muchos contextos, pero su relación con el código es aún más indirecta: ni lo describe con precisión ni contribuye a generarlo.

***Imagen 4.*** *La spec no describe el software; lo produce.*

El *Manifiesto Ágil* tenía razón al cuestionar la prioridad de estas dos categorías sobre el software funcionando. La documentación informativa puede quedar desactualizada al día siguiente de escribirse. La documentación administrativa puede convertirse en un fin en sí mismo que consume esfuerzo sin aportar valor al producto. Ambas son derivados del proceso de construcción, no motores del mismo.

SDD introduce una tercera categoría que no existía antes: **documentación operativa**. Es la spec que el agente de IA consume directamente para generar código. No informa a una persona sobre el software. No satisface un requisito administrativo. Es el input del proceso de producción. Es, literalmente, lo que mueve al agente.

Conviene ser preciso, porque la idea de documentar antes de codificar no es nueva. Los documentos de requisitos, los diseños de arquitectura y los esquemas de base de datos son documentación previa al código que existe desde los orígenes de la ingeniería de software. Se escribían precisamente para encauzar la construcción, y en ese sentido son ancestros directos de la documentación SDD.

La diferencia no está en el *cuándo* —antes o después del código— sino en el *cómo se consume* y en los problemas que el enfoque tradicional arrastraba. La documentación previa tradicional la leía un desarrollador humano que la interpretaba con juicio profesional, experiencia de dominio y sentido común. Había margen para que el desarrollador completara lo que el documento no decía, resolviera ambigüedades sobre la marcha y adaptara la implementación a lo que «claramente se quería decir» aunque no estuviera escrito. La documentación operativa de SDD la consume un agente que interpreta literalmente. No hay margen para «ya lo interpretará bien el desarrollador». Lo que la spec dice es lo que el agente hace; lo que la spec no dice es lo que el agente inventa.

Además, la documentación previa tradicional arrastraba tres problemas que limitaban su eficacia: el alcance solía ser desmesurado (se intentaba especificar el sistema completo antes de construir nada), el feedback era tardío (los errores en los requisitos se descubrían meses después, cuando ya había código implementado) y la documentación podía convertirse en un fin en sí mismo (se medía por volumen y completitud formal, no por su contribución real a la calidad del software). SDD hereda la idea de documentar antes de construir, pero resuelve estos tres problemas: el alcance es un incremento, el feedback es inmediato a través de las puertas de aprobación, y la documentación se autovalida porque el agente la usa directamente para producir código.

## Qué tiene SDD de waterfall (y qué no)

**Lo que comparte con waterfall:** SDD utiliza fases secuenciales con entregables definidos. Requisitos antes que diseño, diseño antes que tareas, tareas antes que

implementación. Cada fase produce un artefacto que se revisa antes de avanzar. Esto es secuencialidad, y la secuencialidad es una característica de waterfall.

**Lo que no comparte con waterfall**: la diferencia fundamental es el **alcance**. Waterfall aplicaba la secuencia a todo el proyecto: meses de requisitos, meses de diseño, meses de implementación. SDD aplica la secuencia a cada incremento: una funcionalidad, una historia de usuario, un cambio. El ciclo completo de SDD —desde la idea hasta el código en producción— puede medirse en horas o días, no en meses.

La segunda diferencia es el **feedback.** En waterfall, el feedback llega al final. En SDD, hay feedback en cada puerta de aprobación, y cada ciclo de SDD es un ciclo de aprendizaje completo. Si la spec está mal, se corrige antes de implementar. Si el diseño no convence, se revisa antes de descomponer en tareas. El coste de corrección es órdenes de magnitud menor.

La tercera diferencia es la **reversibilidad.** En waterfall, volver atrás es costoso porque se ha invertido mucho trabajo en cada fase. En SDD, las fases previas a la implementación son documentos de texto: modificarlos es trivial en comparación con reescribir código.

## La conclusión matizada de Thoughtworks

Thoughtworks incluyó SDD en su *Technology Radar Vol. 33* (noviembre de 2025) en la categoría Assess. El análisis de Böckeler concluye que SDD funciona mejor cuando los requisitos son ambiguos, los equipos están distribuidos o las regulaciones exigen trazabilidad. Para desarrolladores individuales trabajando en problemas bien entendidos, el overhead puede no compensar.

Esta es una conclusión ágil en espíritu: no hay una práctica que sea siempre buena o siempre mala; depende del contexto. La sabiduría está en saber cuándo aplicarla.

La propia Böckeler documentó un caso revelador: al pedir a Kiro que arreglara un bug pequeño, el documento de requisitos generado convirtió la tarea en cuatro historias de usuario con dieciséis criterios de aceptación. «Era como usar un mazo para romper una nuez».

Este riesgo es real y hay que nombrarlo: SDD mal calibrado puede convertirse en una burocracia de especificación que ralentiza en lugar de acelerar. La solución no es rechazar SDD sino ajustar su intensidad al tamaño del problema. Un bug menor no necesita una spec de cuatro fases. Una funcionalidad compleja que afecta a múltiples partes del sistema sí la necesita.

## Cuándo aplicar SDD

Si tuviéramos que trazar una línea, diríamos que SDD aporta valor neto cuando se cumplen una o más de estas condiciones:

**El cambio es sustancial**: no es un ajuste de CSS o un fix de una línea, sino una funcionalidad nueva o un refactoring significativo.

- **Hay ambigüedad**: los requisitos no son evidentes y hay decisiones de diseño por tomar.
- **Hay riesgo**: el cambio afecta a partes críticas del sistema o a múltiples componentes.
- **Hay equipo**: más de una persona (humana o agente) va a trabajar en ello.
- **Hay continuidad**: el código tendrá que mantenerse y evolucionar, no es un prototipo desechable.

Cuando ninguna de estas condiciones se cumple —un script rápido, un prototipo exploratorio, un cambio trivial—, el vibe coding sigue siendo la herramienta adecuada. SDD no sustituye al vibe coding; lo complementa para los escenarios donde las vibes no escalan.

Esta idea de calibrar la intensidad del proceso al tamaño del problema es tan central en SDD que tiene nombre propio —*principio de proporcionalidad*— y reaparecerá en los capítulos 12, 19 y 23.

## 5. SDD como consecuencia natural de los principios ágiles

### El argumento positivo

El capítulo anterior desmontó la objeción de que SDD es waterfall disfrazado. Este capítulo va más lejos: argumenta que SDD es lo que ocurre cuando aplicas principios ágiles de forma coherente a un equipo donde parte de los constructores son agentes de IA.

No es un argumento forzado. Si leemos los principios del Manifiesto Ágil con ojos de 2026, varios de ellos encuentran en SDD una expresión más completa que la que tenían en el desarrollo tradicional.

### Feedback rápido (principios 1 y 12)

El primer principio del Manifiesto habla de satisfacer al cliente mediante la entrega temprana y continua de software con valor. El duodécimo dice: «a intervalos regulares, el equipo reflexiona sobre cómo ser más efectivo y ajusta su comportamiento en consecuencia».

SDD implementa feedback rápido a dos niveles. A nivel de proceso, las puertas de aprobación entre fases son puntos de inspección y adaptación: ¿estos requisitos son los

correctos? ¿Este diseño es viable? ¿Estas tareas cubren todo lo que necesitamos? Cada puerta es una oportunidad de corregir antes de invertir más esfuerzo. A nivel de implementación, cada tarea atómica produce un commit que se puede evaluar contra su spec.

Comparado con el vibe coding, donde el feedback llega al final —cuando el código está generado y hay que revisarlo en bloque— o durante la ejecución —interrumpiendo constantemente con aprobaciones individuales—, SDD propone un término medio: feedback estructurado en los momentos donde la información es más valiosa.

## Entrega incremental (principio 3)

El tercer principio ágil favorece la entrega de software funcional con frecuencia, con preferencia por ciclos cortos. SDD no contradice este principio; lo operativiza en el contexto de la IA.

Cada spec en SDD cubre un incremento —una funcionalidad, un cambio, una mejora— no el sistema completo. El ciclo de SDD para un incremento puede completarse en horas. Al final del ciclo hay software funcionando, tests pasando y un commit atómico que puede desplegarse.

La diferencia con waterfall es que no hay un «gran diseño previo» de todo el sistema. Hay muchos diseños pequeños, cada uno para un incremento, cada uno completo en sí mismo. Esto es desarrollo incremental con un paso de especificación integrado, no desarrollo secuencial disfrazado.

## Colaboración con el cliente (tercer valor del *Manifiesto*)

El tercer valor del *Manifiesto* favorece la colaboración con el cliente sobre la negociación contractual. Paradójicamente, SDD mejora la colaboración precisamente porque produce artefactos que la facilitan.

Una spec es legible por todas las personas del equipo —y por muchos stakeholders—. Cuando el product owner y el desarrollador discuten sobre un requisito, discuten sobre un documento que ambos pueden leer, no sobre código que solo uno de ellos entiende. Cuando hay desacuerdo, se negocia sobre la spec, que es barata de cambiar, no sobre el código, que ya se ha generado.

Las specs hacen explícito lo que antes vivía en la cabeza de una persona. En un equipo ágil tradicional, mucho conocimiento se transmite en conversaciones, *standups* y sesiones de refinamiento. Ese conocimiento es valioso pero volátil: se pierde cuando alguien se va de vacaciones, cambia de proyecto o simplemente lo olvida. Las specs capturan las decisiones y sus razones en un formato persistente y compartible.

## Respuesta al cambio (cuarto valor del *Manifiesto*)

El cuarto valor del Manifiesto favorece la respuesta ante el cambio sobre seguir un plan. La objeción habitual a SDD es: «si ya tengo una spec aprobada y ahora cambian los requisitos, ¿no es más rígido que el vibe coding?».

En realidad, es menos rígido, por una razón económica: es más barato modificar una spec que reescribir código generado.

Una spec es un documento de texto. Cambiar un criterio de aceptación, añadir una restricción, eliminar un caso de uso: son operaciones de edición que tardan minutos. El código correspondiente ni siquiera existe todavía, así que no hay nada que tirar. En cambio, si ya generaste código mediante vibe coding y ahora cambian los requisitos, tienes que decidir si merece la pena parchear el código existente —con el riesgo de acumular deuda técnica— o regenerarlo desde cero —perdiendo el trabajo ya hecho—.

SDD crea un punto de bajo coste para absorber cambios: la spec. Los cambios que llegan antes de la implementación son triviales de integrar. Los que llegan después requieren volver a la spec, modificarla y regenerar. El ciclo es más largo pero predecible.

## La tesis de esta guía

Recapitulando: SDD no es una moda técnica, no es el regreso de waterfall y no es un capricho de las herramientas. SDD es la consecuencia lógica de aplicar principios ágiles cuando tu equipo incluye agentes de IA.

Los principios ágiles siempre hablaron de transparencia, inspección y adaptación. De entregar incrementos de valor. De colaborar alrededor de artefactos compartidos. De responder al cambio. SDD aplica todo esto a un contexto nuevo: uno donde el acto de construir software se ha bifurcado entre la especificación (humana) y la generación (artificial).

La guía *Scrum en equipos con IA* de Scrum Manager introdujo roles, artefactos y ceremonias adaptadas a este contexto. SDD es el método de trabajo que da vida a esos conceptos en el día a día del equipo. No son dos ideas separadas sino dos caras de la misma adaptación: *Scrum en equipos con IA* define la estructura del equipo y su marco de trabajo; SDD define cómo se construye el software dentro de esa estructura.

# PARTE II — La metodología: fases, artefactos y puertas de calidad

## 6. El flujo de cuatro fases

### La estructura convergente

Independientemente de la herramienta que se utilice —Kiro, Spec Kit, Claude Code nativo o cualquier otra— todas las implementaciones de SDD convergen en la misma estructura de cuatro fases:

**Requisitos → Diseño → Tareas → Implementación**

Kiro las llama *Requirements, Design, Tasks*. GitHub Spec Kit las llama *Specify, Plan, Tasks*. Otros frameworks usan nombres ligeramente diferentes. Pero la estructura subyacente es la misma, y esto no es casualidad: refleja la secuencia lógica mínima para pasar de una intención a un software funcionando cuando el constructor es un agente de IA.

Cada fase produce un artefacto concreto —un documento— que se revisa y aprueba antes de avanzar a la siguiente. Estas revisiones entre fases son las **puertas de aprobación** (*approval gates*), y son el mecanismo que hace viable todo el flujo. Sin ellas, SDD sería simplemente waterfall con documentos más cortos.

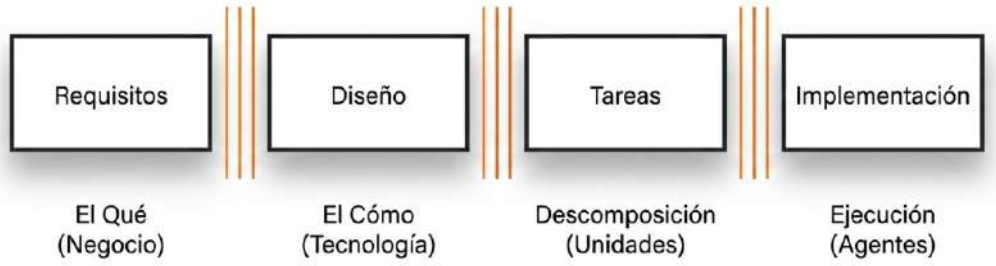



***Imagen 5.*** *La estructura convergente*

### El principio rector

El principio que gobierna todo el flujo es: **revisa en las puertas de fase, no durante la implementación**.

Este principio responde a un problema real del trabajo con agentes. En el modo de trabajo habitual, el desarrollador interactúa con el agente en un ciclo continuo de instrucciones y aprobaciones: el agente propone un cambio, el desarrollador lo aprueba o lo rechaza, el agente propone otro, y así sucesivamente. Cada aprobación individual es

una interrupción, un cambio de contexto y una microdecisión. Multiplicado por decenas o centenares de cambios en una sesión de trabajo, este patrón genera la *fatiga de aprobación* que muchos desarrolladores reconocen.

SDD concentra la revisión humana en los puntos donde la información es más valiosa y el coste de corrección es menor: entre fases. ¿Los requisitos son los correctos? Revísalo antes de diseñar. ¿El diseño es viable? Revísalo antes de descomponer en tareas. ¿Las tareas cubren todo? Revísalo antes de implementar. Una vez que las tareas están aprobadas, la implementación puede ejecutarse con mínima intervención porque el contrato ya está claro.

El resultado es menos interrupciones, menos cambios de contexto y más calidad. La calidad se asegura mediante la inversión previa en la spec, no mediante la supervisión constante de la ejecución.

## Visión general del flujo

Antes de entrar en el detalle de cada fase:

- **Fase 1 — Requisitos**: se define qué se va a construir desde la perspectiva del usuario y del negocio. Entregable: documento de requisitos.
- **Puerta 1**: el equipo revisa los requisitos. Si se aprueba, se avanza al diseño.
- **Fase 2 — Diseño**: se define cómo se va a construir. Entregable: documento de diseño técnico.
- **Puerta 2**: el equipo revisa el diseño. Si se aprueba, se avanza a las tareas.
- **Fase 3 — Tareas**: se descompone el diseño en unidades atómicas de implementación. Entregable: lista de tareas con prompts estructurados.
- **Puerta 3**: el equipo revisa las tareas. Si se aprueba, se implementa.
- **Fase 4 — Implementación**: los agentes ejecutan las tareas. Entregable: código, tests y commits atómicos.

## No es todo o nada

Las cuatro fases representan el flujo completo, pero no todos los problemas necesitan las cuatro con la misma profundidad. Un cambio complejo en un sistema crítico justifica una spec exhaustiva en cada fase. Una funcionalidad mediana puede necesitar requisitos y tareas, con un diseño ligero. SDD es una metodología, no un formulario que se rellena mecánicamente.

La habilidad está en calibrar la intensidad del proceso al tamaño del problema —el principio de proporcionalidad del capítulo 4—. Es una habilidad que se desarrolla con la práctica y que, como veremos en la Parte IV, el agile enabler ayuda a regular dentro del equipo.

# 7. Fase 1: Requisitos

## Qué se produce

El entregable de la primera fase es un **documento de requisitos** que describe qué se va a construir desde la perspectiva del usuario y del negocio. No es un documento técnico: no dice qué tecnología usar ni cómo estructurar el código. Dice qué comportamiento se espera del sistema, bajo qué condiciones y cómo se verificará que funciona.

El formato habitual incluye historias de usuario con criterios de aceptación, aunque no es el único posible. Lo importante no es la plantilla sino la precisión: cada requisito debe ser lo suficientemente claro para que un agente de IA pueda implementarlo sin tomar decisiones no autorizadas.

## Antes de escribir: el Impact Report

En equipos que trabajan sobre un codebase existente —la situación habitual en entornos profesionales— la primera fase no empieza escribiendo requisitos. Empieza analizando el código actual.

El **Impact Report** (informe de impacto) es un análisis del codebase que responde a tres preguntas: ¿qué ficheros se verán probablemente afectados por este cambio? ¿Qué patrones y convenciones existen ya que debemos respetar? ¿Qué efectos colaterales podría tener el cambio en otras partes del sistema?

El Impact Report cumple una función preventiva fundamental: evita que los requisitos pidan algo que duplica lo que ya existe, contradice patrones establecidos o ignora dependencias que causarán problemas durante la implementación. Es la diferencia entre diseñar sobre un terreno que conoces y diseñar sobre un terreno que imaginas.

En la práctica, el Impact Report se genera mediante búsqueda en el codebase —estructural y semántica— antes de que se escriba un solo requisito. El agente analiza el código, identifica los puntos de contacto con el cambio propuesto y produce un informe que el equipo revisa. Este informe informa los requisitos: si el sistema ya tiene un mecanismo de envío de correos transaccionales, por ejemplo, el requisito no debería pedir «construir un sistema de envío de correos» sino «extender el sistema existente para soportar el nuevo canal».

En nuestro sistema de notificaciones multicanal, el Impact Report detectaría —por ejemplo— que la plataforma ya tiene un servicio de correo usado para recuperación de contraseñas, un sistema de preferencias de usuario con tabla `user\_preferences`, y una cola de trabajos asíncronos en Redis. El requisito puede entonces apoyarse en esos tres elementos en lugar de proponer alternativas que colisionarían con ellos.

Para proyectos *greenfield* (nuevos, sin código previo), el Impact Report no aplica en su primera iteración, pero sí a partir de la segunda funcionalidad. En cuanto hay código, hay contexto que respetar.

## Historias de usuario en SDD

Las historias de usuario en SDD siguen la estructura clásica pero con una exigencia de precisión mayor que la habitual en desarrollo ágil tradicional.

En un equipo ágil clásico, una historia como «Como usuario, quiero recibir notificaciones de cambios en mis documentos» es un punto de partida para una conversación. El equipo la refinará en el sprint planning, discutirá los detalles y el desarrollador la interpretará con su conocimiento del sistema y del dominio.

En SDD, esa misma historia necesita criterios de aceptación que no dejen margen para la interpretación del agente:

- ¿Qué cambios disparan notificación? (nueva versión del documento, comentario añadido, cambio de permisos, descarga por un tercero).
- ¿Qué canales? (correo electrónico, push móvil, mensaje in-app).
- ¿Con qué frecuencia? (inmediata, agrupada por hora, resumen diario).
- ¿Qué ocurre si el usuario no tiene un canal configurado?
- ¿Cuál es el tiempo de respuesta aceptable entre el evento y la notificación?

Esto no significa que la historia deba convertirse en un pliego de condiciones. Significa que los criterios de aceptación deben ser **verificables**: debe ser posible determinar, al ver el resultado, si se cumplen o no. «La notificación es rápida» no es verificable. «La notificación llega al canal de destino en menos de 30 segundos desde el evento en el 99 % de los casos» sí lo es.

## La notación EARS

Varias herramientas SDD utilizan la notación EARS (*Easy Approach to Requirements Syntax*) para estructurar los criterios de aceptación. EARS fue desarrollada por Alistair Mavin en Rolls-Royce y publicada en la conferencia IEEE Requirements Engineering de 2009, y ha sido adoptada por organizaciones como Airbus, NASA y Siemens.

EARS no es una notación compleja. Es un conjunto de patrones basados en palabras clave que restringen el lenguaje natural lo justo para eliminar ambigüedad sin sacrificar legibilidad. La plantilla básica es:

> **Mientras** [precondición opcional], **cuando** [evento disparador opcional], **el sistema debe** [respuesta del sistema].

(La plantilla en inglés, por si se busca en la documentación original, es *While [precondition], when [trigger], the system shall [response]*.)

EARS define cinco patrones que cubren la inmensa mayoría de requisitos:

- **Ubicuo** (sin precondición ni evento): define propiedades permanentes del sistema. *Ejemplo*: «El sistema debe cifrar todas las contraseñas almacenadas usando bcrypt con un coste mínimo de 12 rondas».
- **Dirigido por evento** (*cuando...*): se activa al ocurrir algo específico. *Ejemplo*: «Cuando se publica una nueva versión de un documento, el sistema debe encolar una notificación para cada usuario con permiso de lectura sobre ese documento en un plazo máximo de 5 segundos».
- **Dirigido por estado** (*mientras...*): activo mientras se cumpla una condición. *Ejemplo*: «Mientras el usuario tiene activo el modo "no molestar", el sistema debe retener sus notificaciones por push y entregar solo un resumen al finalizar el modo».
- **Comportamiento no deseado** (*si...*): gestiona errores o excepciones. *Ejemplo*: «Si la contraseña se introduce incorrectamente tres veces consecutivas, el sistema debe bloquear la cuenta durante 15 minutos y notificar al usuario por correo».
- **Opcional** (*donde...*): funcionalidad que depende de la configuración. *Ejemplo*: «Donde la organización ha habilitado la autenticación en dos pasos, el sistema debe solicitar un código de verificación después de validar la contraseña».

La notación no es obligatoria, pero resulta especialmente útil: los agentes de IA responden bien a requisitos estructurados con patrones consistentes. Un requisito escrito en EARS reduce la probabilidad de que el agente malinterprete la intención, porque la estructura elimina las ambigüedades más comunes del lenguaje natural.

## Requisitos como comportamiento, no como instrucción técnica

Un error frecuente al escribir requisitos para SDD es confundir el qué con el cómo. El requisito «Usar PostgreSQL para almacenar las preferencias de notificación» es una decisión técnica, no un requisito. El requisito sería: «Como usuario, puedo configurar por qué canales quiero recibir notificaciones y para qué tipos de evento, y mis preferencias se aplican de forma inmediata a las notificaciones posteriores».

La decisión de usar PostgreSQL, MongoDB o cualquier otra tecnología pertenece a la fase de diseño, no a la de requisitos. Mezclar las dos fases en un solo documento es uno de los caminos más rápidos hacia la sobreespecificación y hacia la pérdida del valor que aporta la separación en fases.

Esta distinción tiene una implicación práctica directa para el equipo ágil: los requisitos deberían poder ser escritos (o al menos validados) por alguien con conocimiento del dominio pero sin conocimiento técnico profundo. Si un requisito solo puede entenderlo un desarrollador, probablemente contiene decisiones técnicas disfrazadas de requisitos.

# 8. Fase 2: Diseño

## Qué se produce

El entregable de la segunda fase es un **documento de diseño técnico** que describe cómo se van a implementar los requisitos aprobados. Aquí sí se toman decisiones técnicas: qué patrones de arquitectura aplicar, qué componentes del sistema se ven afectados, qué interfaces necesitan modificarse, cómo se estructura la base de datos.

El documento de diseño no es un diagrama UML exhaustivo ni un diseño de clases detallado al estilo de la ingeniería de software clásica. Es un documento pragmático que responde a las preguntas que un agente necesita para implementar los requisitos sin tomar decisiones de arquitectura por su cuenta.

## El diseño como guía de implementación

La diferencia entre un buen documento de diseño y uno deficiente está en la pregunta que responde. Un diseño deficiente responde: «¿cómo podría funcionar esto en abstracto?». Un buen diseño responde: «¿cómo va a funcionar esto en este codebase concreto, con estos patrones existentes, con estas restricciones?».

Para construir un buen documento de diseño, se necesita:

**Análisis del codebase existente**: ¿qué patrones usa ya el proyecto? Si el sistema utiliza un patrón *Repository* para el acceso a datos, el diseño debería indicar que la nueva funcionalidad seguirá ese mismo patrón. Si hay un sistema de eventos, el diseño debería especificar si la nueva funcionalidad emite eventos y cuáles.

**Identificación de ficheros afectados**: basándose en el Impact Report de la fase anterior, el diseño debería listar qué ficheros se crearán, cuáles se modificarán y por qué. Esto reduce drásticamente la probabilidad de que el agente modifique ficheros que no debería tocar o ignore ficheros que necesitan actualizarse.

**Decisiones de diseño explícitas**: cuando hay más de una forma razonable de implementar algo, el documento debería registrar la decisión tomada y la razón. Volviendo al ejemplo del sistema de notificaciones: «Usamos la cola de trabajos existente en Redis para el envío asíncrono. El requisito establece que la notificación debe entregarse en menos de 30 segundos; procesarla síncronamente bloquearía la respuesta al evento que la dispara».

**Restricciones y dependencias**: qué no debe cambiar (para evitar efectos colaterales) y qué necesita estar listo antes (para planificar el orden de las tareas).

### Lo que el diseño no es

El documento de diseño no es el lugar para volver a discutir los requisitos. Si durante el diseño se descubre que un requisito es inviable o necesita modificación, la respuesta correcta es volver a la puerta de aprobación 1 y revisar el requisito, no reinterpretarlo silenciosamente en el diseño.

Tampoco es un manual completo de arquitectura del sistema. El diseño cubre la funcionalidad específica que se está construyendo, no el sistema entero. Se apoya en la **constitución del proyecto** (que veremos en la Parte V: un documento tipo CLAUDE.md o agents.md que captura las convenciones y decisiones arquitectónicas generales del codebase) para todo lo que aplica de forma transversal.

### La línea entre diseño suficiente y sobrediseño

Hay una tensión inherente en el diseño: demasiado poco y el agente tomará decisiones por su cuenta; demasiado y el diseño se convierte en pseudocódigo que el agente traduce mecánicamente, eliminando cualquier ventaja de trabajar con IA.

El punto de equilibrio está en especificar las decisiones que importan y dejar abierto lo que no importa. Importa qué patrón de arquitectura se usa. No importa cómo se llama una variable local. Importa qué API se expone. No importa el orden de los argumentos en una función interna.

Una buena heurística: si dos desarrolladores sénior razonables tomarían la misma decisión sin necesidad de discutirlo, probablemente no necesita estar en el diseño. Si podrían tomar decisiones diferentes, cada una razonable pero con consecuencias distintas, debería estar en el diseño.

## 9. Fase 3: Tareas

### Qué se produce

El entregable de la tercera fase es una **lista de tareas atómicas** que descomponen el diseño en unidades de implementación. Cada tarea es lo suficientemente pequeña para que un agente la ejecute en una sesión con contexto limpio y lo suficientemente precisa para que el resultado sea predecible.

Esta fase es donde SDD transforma la complejidad de un cambio grande en una serie de cambios pequeños y manejables. La complejidad no desaparece, pero se convierte en una secuencia de piezas con fronteras claras.

## Anatomía de una tarea atómica

Una tarea bien definida en SDD tiene características específicas que la distinguen de un ítem genérico del backlog:

**Alcance limitado**: cada tarea afecta normalmente a entre uno y tres ficheros. Si una tarea necesita modificar más ficheros, probablemente se puede descomponer en tareas más pequeñas. Este límite no es arbitrario: responde a la realidad de cómo los agentes gestionan el contexto. Cuantos menos ficheros involucrados, más precisa es la implementación.

**Prompt estructurado**: cada tarea incluye una instrucción que el agente recibirá para ejecutarla. Este prompt suele tener cuatro campos. Ejemplo aplicado a nuestro sistema de notificaciones:


***Imagen 6.*** *Estructura del prompt*

- **Rol:** qué papel debe asumir el agente. «Eres un desarrollador backend sénior especializado en APIs REST y mensajería asíncrona».
- **Tarea**: qué debe hacer concretamente. «Crear el endpoint POST /api/notifications/dispatch que recibe un payload con eventType, documentId y affectedUserIds, y encola una tarea en la cola notifications-queue para cada usuario afectado».
- **Restricciones**: qué no debe hacer o qué límites debe respetar. «No modificar los endpoints existentes. Seguir el patrón de validación de la carpeta /validators. No añadir dependencias nuevas. La inserción en la cola debe respetar el patrón existente en jobs/email_dispatcher.py».
- **Criterios de éxito**: cómo se verificará que la tarea está completa. «El endpoint responde 202 con el número de notificaciones encoladas. Responde 400 si falta algún campo obligatorio. Los tests unitarios cubren ambos casos y el caso de cola no disponible».

Los prompts suelen escribirse en la misma lengua de trabajo del equipo; los identificadores técnicos (rutas, nombres de endpoints, variables) se mantienen en inglés por convención del oficio.

**Independencia verificable**: cada tarea produce un resultado que puede evaluarse de forma aislada. No se necesita haber completado todas las tareas para saber si una tarea individual está bien hecha. Esto permite detectar problemas tarea a tarea en lugar de descubrirlos al final.

## Dependencias y oleadas

Las tareas no son necesariamente secuenciales. Algunas dependen de otras (no puedes implementar el endpoint de despacho si no has creado la cola) pero otras son independientes y pueden ejecutarse en paralelo.

La organización en **oleadas** (*waves*) refleja esta realidad. En el caso del sistema de notificaciones:

- ***Oleada 1:*** tareas fundacionales sin dependencias. Creación de la tabla `notification\_preferences`, definición de la cola `notifications-queue`, esquema del evento `DocumentUpdated`.
- **Oleada 2:** tareas que dependen de la oleada 1. Servicio que lee preferencias y decide canales, endpoint `/api/notifications/dispatch`.
- **Oleada 3:** tareas que dependen de la oleada 2. Worker que consume de la cola y envía por cada canal, pantalla de configuración de preferencias en la interfaz web.

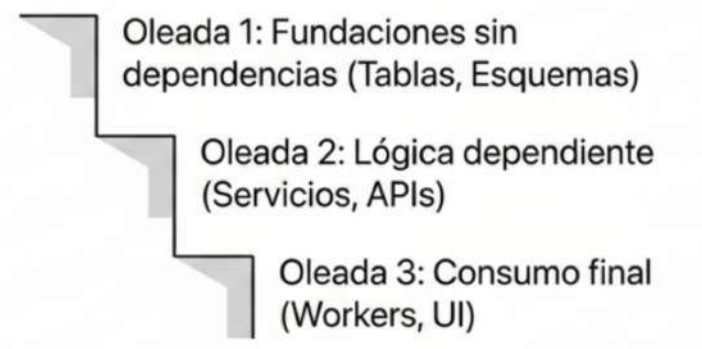



***Imagen 7.*** *Organización en oleadas*

Las tareas dentro de una misma oleada pueden ejecutarse en paralelo —cada una por un subagente con contexto limpio—. Las oleadas se ejecutan en secuencia. Este modelo aprovecha la capacidad de los agentes para trabajar en paralelo sin contaminación de contexto entre tareas.

### El riesgo de la descomposición excesiva

Existe un punto en el que descomponer más es contraproducente. Si una tarea es tan pequeña que el prompt que la describe es más largo que el código que producirá, la descomposición ha ido demasiado lejos. El objetivo es encontrar el tamaño de tarea donde el agente puede trabajar con contexto completo y el resultado es revisable de un vistazo.

Un indicador práctico: si la tarea se puede implementar y verificar en minutos (no en horas), probablemente tiene el tamaño correcto. Si tarda horas, probablemente es demasiado grande. Si el overhead de leer el prompt y configurar el contexto es mayor que el de ejecutar la tarea, probablemente es demasiado pequeña.

## 10. Fase 4: Implementación

### Qué se produce

El entregable de la cuarta fase es **código funcional, tests y commits atómicos.** Cada tarea de la fase anterior se convierte en código que pasa sus criterios de éxito y se registra como un commit independiente en el control de versiones.

Es la fase donde el agente hace el trabajo de construcción. Y es, paradójicamente, la fase donde el humano interviene menos, si las tres fases anteriores se han hecho bien.

### Ejecución delegada

El patrón de implementación en SDD invierte la relación habitual entre el desarrollador y el agente. En el vibe coding, el desarrollador dirige al agente paso a paso, revisando y corrigiendo sobre la marcha. En SDD, el desarrollador ha invertido ese esfuerzo de dirección en las fases anteriores; la implementación es ejecución delegada.

Cada tarea se asigna a un agente (o subagente) que recibe el prompt estructurado de la tarea, el contexto relevante del diseño y la constitución del proyecto. El agente implementa, ejecuta los tests definidos en los criterios de éxito y produce un commit. Si los criterios de éxito no se cumplen, el agente itera. Si después de varios intentos no se cumplen, la tarea se marca como bloqueada y el humano interviene, pero solo en esa tarea, sin afectar al resto.

### Commits atómicos

Cada tarea genera un commit independiente. Esta práctica, que puede parecer un detalle técnico, tiene consecuencias profundas para la mantenibilidad del proyecto:

**Reversibilidad granular**: si una tarea produce un resultado defectuoso, se puede revertir su commit sin afectar al trabajo de las demás. En el vibe coding, un cambio

grande que falla obliga a deshacer todo o a buscar manualmente qué partes funcionan y cuáles no.

**Trazabilidad**: cada commit está vinculado a una tarea, que está vinculada a un diseño, que está vinculado a unos requisitos. Se puede recorrer la cadena desde cualquier línea de código hasta la intención de negocio que la motivó. Es la trazabilidad que los marcos regulatorios exigen y que los equipos ágiles rara vez logran mantener sin esfuerzo extra.

**Bisección**: `git bisect` es mucho más útil cuando cada commit es una unidad lógica coherente. Si un bug aparece en algún momento, identifica con precisión qué tarea lo introdujo, en lugar de señalar un commit que mezcla cambios heterogéneos.

### Logs de implementación

Durante la implementación, el agente produce logs que documentan qué hizo, qué decisiones menores tomó (las que no estaban especificadas en la tarea y no eran lo suficientemente significativas para requerir consulta), y qué problemas encontró. Estos logs tienen valor a medio plazo: cuando alguien necesite entender por qué el código es como es, los logs proporcionan contexto que el código por sí solo no ofrece.

### La revisión al cierre de fase

La revisión del código generado se hace al cierre de la fase de implementación o por lotes (por ejemplo, al completar una oleada), no después de cada tarea individual. Esto es coherente con el principio rector: revisar en puertas de fase, no durante la ejecución.

La revisión de implementación es diferente a las revisiones de fases anteriores. Ya no se discute si los requisitos son correctos ni si el diseño es viable: eso se aprobó en sus respectivas puertas. La revisión de implementación verifica que el código producido cumple la spec y que la calidad técnica es aceptable.

Esto simplifica enormemente la revisión, porque el revisor tiene un contrato claro contra el que evaluar: los criterios de éxito de cada tarea, los requisitos de la funcionalidad y las restricciones del diseño. No es «¿este código me parece bien?» sino «¿este código cumple lo que la spec dice que debía cumplir?».

## 11. Las puertas de aprobación

### El mecanismo central

Si las cuatro fases son el cuerpo de SDD, las puertas de aprobación son su sistema nervioso. Son los puntos donde el humano ejerce criterio, donde se detectan problemas

antes de que sean caros de corregir y donde el equipo se alinea sobre lo que se está construyendo.

Hay tres puertas principales, una entre cada par de fases consecutivas:

- **Puerta 1 (Requisitos → Diseño)**: ¿los requisitos son correctos, completos y verificables?
- **Puerta 2 (Diseño → Tareas)**: ¿el diseño es viable, coherente con el codebase existente y suficientemente detallado?
- **Puerta 3 (Tareas → Implementación)**: ¿las tareas cubren todo el diseño, el orden de dependencias es correcto y los prompts son precisos?

Y una revisión final al cierre de la implementación: ¿el código cumple los requisitos?

## Qué se revisa en cada puerta

**Puerta 1 — Revisión de requisitos.** La pregunta central es: ¿estamos resolviendo el problema correcto? Implica verificar que las historias de usuario reflejan necesidades reales (no supuestas), que los criterios de aceptación son verificables (no vagos), que el alcance es manejable (no se ha inflado) y que el Impact Report no revela conflictos con el sistema existente.

Es también el momento de detectar requisitos implícitos: cosas que el equipo da por sentadas pero que el agente no va a adivinar. Seguridad, accesibilidad, rendimiento, internacionalización: si no están en los requisitos, no estarán en el código.

**Puerta 2 — Revisión de diseño**. La pregunta central es: ¿esto es viable y coherente? Se verifica que el diseño respeta los patrones existentes del codebase, que las decisiones técnicas están justificadas, que las dependencias están identificadas y que los riesgos están mapeados.

Un error común en esta puerta es aceptar diseños que suenan bien en abstracto pero que ignoran la realidad del código existente. El Impact Report de la fase 1 es la referencia para evitarlo: si el diseño propone un patrón nuevo donde ya existe uno que funciona, hay que justificar por qué.

**Puerta 3 — Revisión de tareas**. La pregunta central es: ¿si ejecutamos estas tareas en este orden, obtendremos lo que el diseño describe? Se verifica que las tareas cubren todo el diseño sin huecos, que las dependencias entre tareas son correctas, que los prompts estructurados son precisos y que los criterios de éxito son evaluables.

Esta es probablemente la puerta más técnica y donde más valor aporta la participación de desarrolladores. Un product owner puede evaluar requisitos. Un agile enabler puede facilitar la revisión de diseño. Pero la revisión de tareas requiere criterio técnico para evaluar si la descomposición tiene sentido.

## Quién aprueba

SDD no prescribe un rol específico para cada puerta: depende de la estructura del equipo. En el marco de *Scrum en equipos con IA*, la asignación natural sería:

- **Puerta 1**: product architect (por conocimiento de dominio) con participación del equipo.
- **Puerta 2**: product architect y product builders (por conocimiento técnico más de dominio).
- **Puerta 3**: product builders (por conocimiento técnico).
- **Facilitación transversal**: agile enabler (para asegurar que las puertas se ejecutan con rigor sin convertirse en cuellos de botella).

Lo esencial no es quién aprueba sino que la aprobación sea un acto deliberado y explícito: alguien lee el artefacto, lo evalúa contra criterios conocidos, y decide si es suficiente para avanzar.

## El coste de un error según la fase

Hay una razón económica poderosa detrás de las puertas de aprobación: el coste de corregir un error crece de forma acelerada a medida que se avanza en las fases.

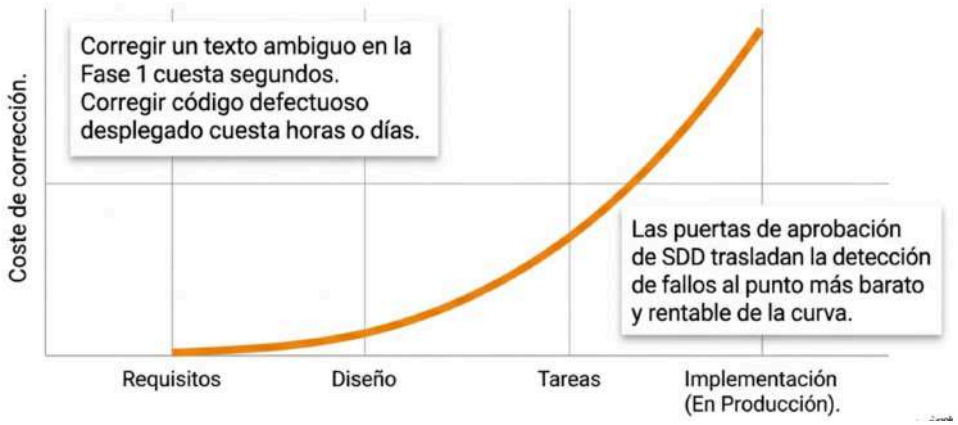



***Imagen 8.*** *Coste del error según la fase*

Un error en los requisitos —por ejemplo, un criterio de aceptación ambiguo— es trivial de corregir: se reescribe una frase en un documento de texto. Si ese mismo error llega al diseño, hay que repensar la solución técnica. Si llega a las tareas, hay que rehacer la descomposición. Si llega a la implementación, hay que descartar y regenerar código.

Las puertas de aprobación son, desde esta perspectiva, el mecanismo de detección temprana más rentable del proceso. Cada minuto invertido en revisar un requisito antes de diseñar ahorra potencialmente horas de reimplementación.

## La diferencia entre revisar una spec y revisar código

Un aspecto que transforma la experiencia de revisión en SDD es que las puertas 1, 2 y 3 revisan documentos de texto, no código. Esto tiene implicaciones importantes:

**Accesibilidad**: más personas del equipo pueden participar. Un product owner puede evaluar si los requisitos son correctos. Un stakeholder puede opinar sobre el diseño. No se necesita saber programar para contribuir a las tres primeras puertas.

**Velocidad**: leer y evaluar una spec es más rápido que leer y evaluar código. La spec describe intenciones y decisiones en lenguaje comprensible; el código las implementa en un lenguaje que requiere conocimientos técnicos para interpretar.

**Foco**: cuando revisas una spec, la pregunta es «¿esto es lo que queremos?». Cuando revisas código, la pregunta se bifurca en «¿esto es lo que queremos?» y «¿está bien implementado?»: dos preguntas que compiten por atención y que, combinadas, hacen la revisión más difícil y más susceptible a fatiga.

SDD separa estas dos preguntas. Las puertas de aprobación 1-3 se centran exclusivamente en «¿esto es lo que queremos y cómo lo queremos?». La revisión de implementación se centra exclusivamente en «¿está bien implementado?». Al separar las preguntas, cada revisión es más eficaz.

# PARTE III — Escribir buenas specs

## 12. Anatomía de una buena spec

### La spec como documento inteligente, no extenso

Si hay una frase que resume lo que separa una buena spec de una mala, es esta: ***minimal no significa necesariamente corto***. Una buena spec cubre lo justo para guiar al agente sin abrumarlo. No escatima en detalle cuando el detalle importa, pero no añade información que no contribuye a la calidad del resultado.

Addy Osmani, tras años de experiencia con agentes en Google, publicó en 2026 el artículo *How to write a good spec for AI agents* (disponible en su blog y reeditado en O'Reilly Radar) donde destila cinco principios para escribir buenas specs. Estos principios no son reglas rígidas sino heurísticas que ayudan a calibrar qué incluir y qué dejar fuera.

### Los cinco principios de Osmani

#### Principio 1 · Empieza con la visión de alto nivel y deja que la IA elabore los detalles

La spec no nace como un documento exhaustivo. Nace como una intención clara —un brief de producto— que describe qué se construye, para quién y qué aspecto tiene el éxito. A partir de esa intención, el propio agente puede generar un borrador más detallado que el equipo revisa y refina.

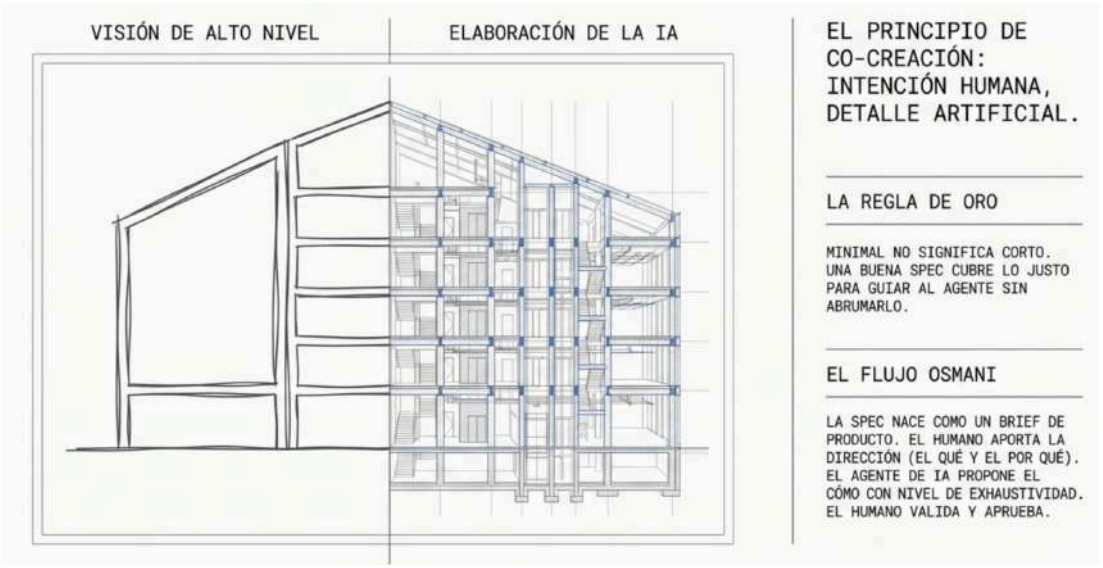



***Imagen 9.*** *Empieza con la visión de alto nivel*

Esto aprovecha una fortaleza de los modelos de lenguaje: son excelentes elaborando detalles cuando tienen una directiva de alto nivel clara, pero se pierden cuando no tienen una misión definida. El humano aporta la dirección; el agente aporta la exhaustividad.

La implicación práctica es que la spec es el primer artefacto que el humano y el agente construyen juntos. No es un documento que se le «entrega» al agente, sino uno que se co-crea. El humano aporta el qué y el por qué; el agente propone el cómo con nivel de detalle; el humano valida, corrige y aprueba.

**Principio 2 · Estructura la spec como un documento profesional, no como una colección de notas sueltas**

Los agentes procesan mejor la información estructurada que la prosa libre. Una spec con secciones claras, encabezados consistentes y formato predecible permite al agente localizar rápidamente la información relevante para cada tarea.

GitHub analizó más de 2.500 ficheros de configuración de agentes en repositorios públicos y encontró un patrón claro: las specs más efectivas cubren **seis áreas**. Este hallazgo se publicó en el GitHub Blog en 2025 (*How to write a great agents.md: lessons from over 2,500 repositories*):

- **Comandos**: no solo nombres de herramientas, sino comandos completos con parámetros. No «usar npm», sino «`npm run build` para compilar, `npm test` para ejecutar tests, `npm run lint --fix` para corregir estilo».
- **Testing**: cómo ejecutar los tests, qué framework se usa, dónde viven los ficheros de test, qué cobertura se espera.
- **Estructura del proyecto**: dónde vive el código fuente, dónde van los tests, dónde la documentación. Ser explícito: «`src/` para código de aplicación, `tests/` para tests unitarios, `docs/` para documentación».
- **Estilo de código**: un fragmento real de código que muestre el estilo del proyecto vale más que tres párrafos describiéndolo. Convenciones de nomenclatura, reglas de formato, ejemplos de output esperado.
- **Flujo git**: nomenclatura de ramas, formato de mensajes de commit, requisitos de pull request. El agente puede seguir estas convenciones si se las describes con claridad.
- **Boundaries (fronteras)**: qué no debe tocar el agente nunca. Secretos, directorios de dependencias, configuraciones de producción. «Nunca hacer commit de secretos» fue la restricción útil más frecuente del estudio.

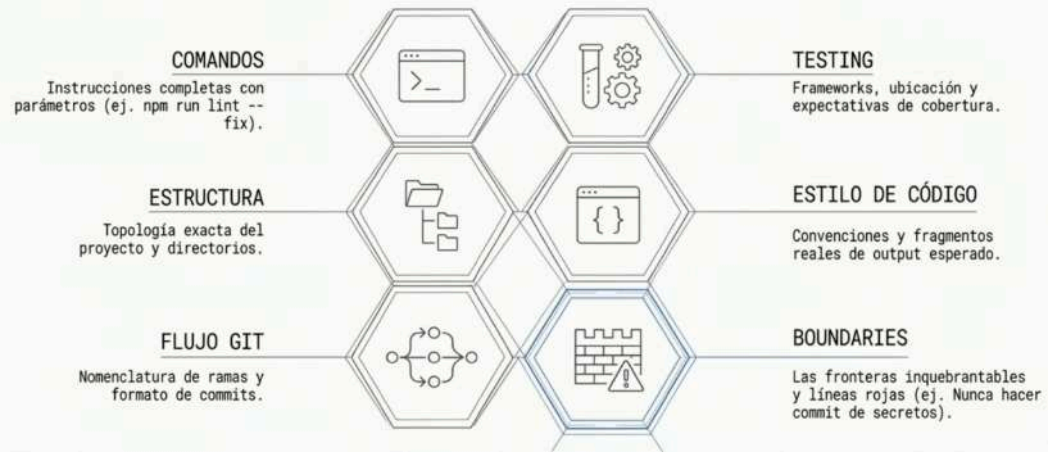



***Imagen 10.*** *Las 6 áreas clave*

Estas seis áreas no son obligatorias ni exhaustivas, pero funcionan como un checklist de completitud: si tu spec no cubre alguna de ellas, probablemente el agente tomará decisiones por su cuenta en ese ámbito.

**Principio 3 · Divide las tareas en prompts modulares, no en un prompt monolítico**

Este principio conecta con la fase de tareas que vimos en la Parte II, pero tiene una dimensión adicional: no se trata solo de dividir el trabajo, sino de dividir el contexto.

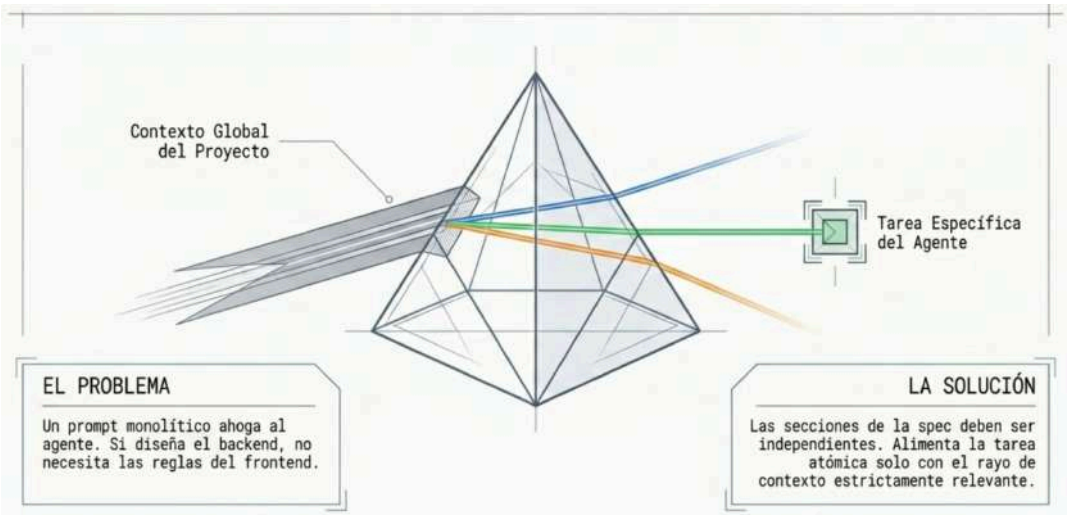



***Imagen 11.*** *Modularidad: divide el contexto, no sólo el trabajo*

Cuando se apilan demasiadas instrucciones o datos en un solo prompt, el rendimiento del modelo cumpliendo cada instrucción individual baja significativamente (el fenómeno que desarrollamos en el capítulo 14). La solución es alimentar al agente con la porción de la spec relevante para la tarea que está ejecutando, no con la spec completa. Si está trabajando en el backend, no necesita la spec del frontend. Si está implementando el modelo de datos, no necesita los detalles de la interfaz de usuario.

La implicación es que una buena spec no es solo un buen documento; es un documento que se puede consumir por partes. Las secciones deben ser lo suficientemente independientes para que cada una pueda alimentar una tarea sin necesidad de cargar todo el contexto.

**Principio 4 · Incorpora autoverificación, restricciones y conocimiento experto**

Una buena spec no es solo una lista de lo que hay que hacer. Es también una guía de control de calidad. Incluye mecanismos para que el agente verifique su propio trabajo, restricciones que evitan errores comunes y conocimiento específico del dominio que solo alguien con experiencia puede aportar.

Este principio tiene tres facetas:

- **Autoverificación**: instruir al agente para que, tras implementar, compare el resultado con la spec y confirme que todos los requisitos se han cumplido. Esto empuja al modelo a reflexionar sobre su output antes de darlo por terminado.
- **Restricciones proactivas**: anticipar dónde el agente podría equivocarse e incluir guardarraíles. Si una librería tiene un bug conocido en cierta versión, mencionarlo. Si una tabla de base de datos tiene triggers que pueden causar efectos colaterales, advertirlo.
- **Conocimiento experto**: volcar en la spec lo que sabes como profesional del dominio. Si sabes que un endpoint concreto es sensible al rendimiento, especifica el umbral. Si sabes que productos y categorías tienen una relación muchos-a-muchos, dilo explícitamente. El agente es tan bueno como el contexto que recibe, y tu experiencia es contexto de alto valor.

**Principio 5 · Trata la spec como un documento vivo, no como un artefacto que se escribe una vez y se olvida**

La spec debe actualizarse cuando se toman decisiones o se descubre información nueva. Si el agente tuvo que cambiar el modelo de datos, refléjalo. Si se recortó una funcionalidad, elimínala. La spec es la fuente de verdad del proyecto, y una fuente de verdad desactualizada es peor que no tener fuente de verdad.

Este principio se desarrolla en profundidad en el capítulo 15.

## El principio de proporcionalidad aplicado a la spec

El principio de proporcionalidad que introdujimos en el capítulo 4 vuelve aquí en clave de heurística práctica: ajusta el detalle de la spec a la complejidad de la tarea. No sobreespecifiques un problema trivial (el agente se enredará o consumirá contexto en instrucciones innecesarias) ni subespecifiques un problema complejo (el agente improvisará donde no debería).

La pregunta de calibración es: ¿qué ocurre si el agente toma la decisión equivocada en este punto? Si la respuesta es «lo corrijo en dos minutos», no necesita estar en la spec. Si la respuesta es «pierdo horas de trabajo o introduzco un bug sutil», necesita estar en la spec.

## 13. El sistema de boundaries: Always / Ask First / Never

### Un marco de delegación, no una lista de prohibiciones

El análisis de GitHub sobre 2.500 configuraciones de agentes reveló que las specs más efectivas no usan una lista plana de reglas. Usan un **sistema de tres niveles** que da al agente un marco claro de decisión sobre cuándo actuar, cuándo consultar y cuándo detenerse.

Este sistema no es específico de SDD: es un patrón general de delegación efectiva que aplica tanto a agentes como a personas. De hecho, funciona exactamente como la delegación a un profesional competente: le dices qué puede hacer por su cuenta, qué necesita tu aprobación y qué está prohibido. La diferencia es que con un agente de IA, estas reglas deben ser explícitas porque el agente no tiene sentido común para inferirlas.

### Los tres niveles

**Always (siempre hacer)**: acciones que el agente debe ejecutar sin preguntar. Son las prácticas estándar del proyecto que no requieren supervisión.

| | | |
|---|---|---|
| Nivel: ALWAYS (Autonomía Total) | Acción: Ejecutar sin interrumpir ni preguntar. | Ejemplos: Ejecutar tests antes de commit, seguir nomenclatura, registrar errores. |
| Nivel: ASK FIRST (Punto de Control) | Acción: Proponer solución y y requerir aprobación humana. | Ejemplos: Modificar esquema de BD, añadir dependencias, cambiar API pública. |
| Nivel: NEVER (Línea Roja Absoluta) | Acción: Bloqueo estricto e inequívoco. | Ejemplos: Hacer commit de secretos, editar node_modules/, eliminar tests fallidos. |

***Imagen 12.*** *Límites (Boundaries) de delegación*

Ejemplos:

- Siempre ejecutar los tests antes de hacer commit.
- Siempre seguir las convenciones de nomenclatura del proyecto.
- Siempre registrar errores en el servicio de monitorización.
- Siempre incluir manejo de errores en los endpoints de API.
- Siempre añadir documentación para funciones públicas.

**Ask First (preguntar primero)**: acciones que podrían ser correctas pero que requieren aprobación humana por su impacto potencial. Es el nivel que captura los cambios de alto impacto que merecen una revisión antes de ejecutarse.

Ejemplos:

- Preguntar antes de modificar el esquema de base de datos.
- Preguntar antes de añadir dependencias nuevas al proyecto.
- Preguntar antes de cambiar la configuración de CI/CD.
- Preguntar antes de modificar la API pública.
- Preguntar antes de refactorizar un módulo que no es parte de la tarea actual.

**Never (nunca hacer)**: líneas rojas absolutas. Acciones que el agente no debe ejecutar bajo ninguna circunstancia.

Ejemplos:

- Nunca hacer commit de secretos o claves API.
- Nunca editar `node\_modules/`, `vendor/` u otros directorios de dependencias.
- Nunca eliminar un test que falla sin aprobación explícita.
- Nunca modificar ficheros de configuración de producción.
- Nunca hacer cambios fuera del alcance de la tarea asignada.

## Por qué funciona

Este sistema de tres niveles funciona mejor que una lista plana de instrucciones por varias razones:

- **Da autonomía donde es segura**. El nivel *Always* libera al agente de consultar en cada microdecisión. Si sabe que siempre debe ejecutar tests antes de un commit, lo hace sin interrumpir. Esto reduce la fatiga de aprobación que vimos en la Parte II.
- **Crea puntos de control donde son necesarios**. El nivel *Ask First* concentra la intervención humana en las decisiones de alto impacto, que es donde aporta más valor. Un cambio en el esquema de base de datos merece una conversación; una variable renombrada, no.
- **Establece líneas rojas inequívocas**. El nivel *Never* elimina categorías enteras de errores. Si el agente tiene claro que nunca debe hacer commit de secretos, no hay ambigüedad que resolver.

## Boundaries como marco de autonomía gradual

Hay un aspecto de los boundaries que conecta con la agilidad y merece atención: el sistema de tres niveles no es estático. A medida que el equipo gana confianza con el agente y con el proceso, los boundaries pueden evolucionar. Algo que hoy es *Ask First*

puede pasar a *Always* cuando el equipo ha verificado que el agente toma buenas decisiones en ese ámbito. Algo que es *Never* puede relajarse a *Ask First* si se implementan salvaguardas adicionales.

Esta evolución es análoga a cómo un mánager gestiona la autonomía de un profesional nuevo: al principio supervisa más, con el tiempo delega más. Los boundaries son el mecanismo que hace esa evolución explícita y controlable.

### Boundaries a nivel de proyecto frente a nivel de tarea

Los boundaries operan en dos niveles:

- **A nivel de proyecto** (en la constitución del proyecto, tipo CLAUDE.md): definen las reglas generales que aplican a cualquier tarea. «Nunca hacer commit de secretos» es una regla de proyecto.
- **A nivel de tarea** (en el prompt estructurado de cada tarea): definen restricciones específicas para esa implementación. «No modificar los endpoints existentes» es una restricción de la tarea de crear un nuevo endpoint, no una regla general del proyecto.

La combinación de ambos niveles crea un marco de delegación completo: las reglas de proyecto aseguran la consistencia global y las restricciones de tarea aseguran la precisión local.

## 14. La maldición de las instrucciones

### El problema de la sobrecarga

Uno de los hallazgos más relevantes para la práctica de SDD proviene de la investigación académica. El paper “Curse of Instructions: Large Language Models Cannot Follow Multiple Instructions at Once” (*OpenReview*, 2024) introdujo el benchmark ManyIFEval con hasta diez instrucciones verificables por prompt y mostró un resultado regular: a medida que se acumulan más instrucciones, el rendimiento del modelo cumpliendo cada instrucción individual cae de forma predecible. La tasa de éxito combinado se ajusta con bastante precisión a la probabilidad individual elevada al número total de instrucciones.

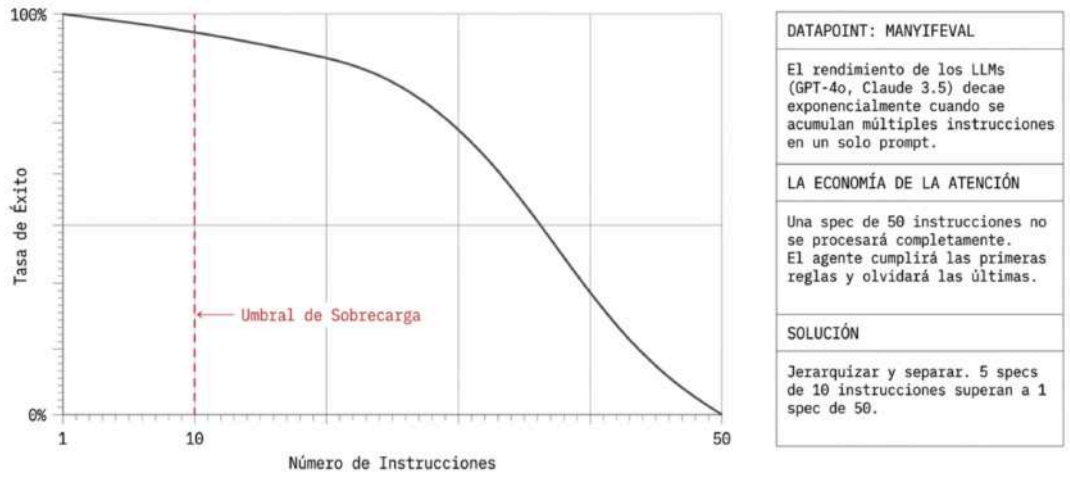



***Imagen 13.*** *La maldición de las instrucciones*

Este efecto se ha manifestado en GPT-4o, Claude 3.5, Gemini 1.5, Gemma y Llama: el modelo cumple las primeras instrucciones con fiabilidad pero empieza a incumplir las últimas a medida que la lista crece. Es una limitación estructural de los modelos tipo transformer (la arquitectura dominante en los LLM actuales) que se atenúa con modelos más capaces pero no desaparece.

En términos prácticos: si presentas diez reglas detalladas en una spec, el agente podría cumplir las primeras de forma fiable y empezar a pasar por alto las últimas. Si presentas veinte, la situación empeora. Si presentas cincuenta, la spec es efectivamente un documento que el agente leerá parcialmente.

## Implicaciones para el diseño de specs

Este hallazgo tiene consecuencias directas para cómo se diseñan las specs:

**Una spec más inteligente, no más larga**. La tentación natural es añadir más detalle para cubrir más casos. La maldición de las instrucciones demuestra que esa estrategia tiene rendimientos decrecientes y eventualmente negativos. A partir de cierto punto, añadir una instrucción más no solo no mejora el resultado; puede empeorar el cumplimiento de instrucciones anteriores.

**Descomponer en vez de acumular**. La solución no es escribir menos, sino dividir mejor. En lugar de una spec de 50 instrucciones que el agente recibirá de golpe, es preferible tener 5 specs de 10 instrucciones cada una, asignadas a las tareas donde son relevantes. Esto es exactamente lo que hace la fase de tareas en SDD: al descomponer el trabajo en unidades atómicas, cada tarea recibe solo las instrucciones que necesita, no todas las del proyecto.

**Priorizar las instrucciones**. Si no es posible dividir más la spec, al menos hay que asegurar que las instrucciones más importantes aparezcan primero. El modelo les dará

más peso. Las restricciones de seguridad, por ejemplo, deberían estar al inicio del documento, no al final.

**Separar niveles de instrucción**. El sistema de boundaries del capítulo anterior ayuda precisamente con esto: en lugar de una lista plana de instrucciones de igual peso, crea tres categorías con niveles diferentes de urgencia. El agente puede procesar las reglas *Never* (pocas, críticas) con más atención que las reglas *Always* (muchas, rutinarias).

## La economía de la atención del agente

Hay una analogía útil con la gestión de equipos humanos. Un mánager que da a un empleado veinte directrices en una reunión no debería sorprenderse cuando el empleado olvida la mitad. La forma efectiva de gestionar es: pocas reglas claras para el día a día, directrices específicas para cada tarea, y una referencia consultable para todo lo demás.

SDD aplica exactamente este modelo. La constitución del proyecto contiene las pocas reglas que aplican siempre. El prompt de cada tarea contiene las directrices específicas. Y la spec completa existe como referencia consultable, pero no se carga entera en la ventana de contexto en cada interacción.

El resultado es que el agente recibe, en cada momento, la cantidad de instrucciones que puede procesar eficazmente. Ni más (que degradaría el cumplimiento) ni menos (que dejaría decisiones sin guiar).

## Señales de que la spec está sobrecargada

Algunos indicadores de que una spec ha cruzado el umbral de la sobrecarga:

- El agente ignora restricciones que están claramente escritas.
- El agente cumple las instrucciones del principio del documento pero no las del final.
- Los resultados mejoran cuando se reduce la spec en lugar de ampliarla.
- El agente produce resultados correctos cuando recibe una tarea aislada pero incorrectos cuando recibe varias tareas juntas.
- El equipo dedica más tiempo a verificar si el agente cumplió la spec que a escribir la spec.

Si alguno de estos síntomas aparece, la respuesta probablemente no es «el agente es malo» sino «la spec le está pidiendo demasiado a la vez».

# 15. Specs vivas frente a specs estáticas

## El problema de la deriva

Escribir una buena spec es la mitad del desafío. La otra mitad es mantenerla alineada con la realidad del código a medida que el proyecto avanza.

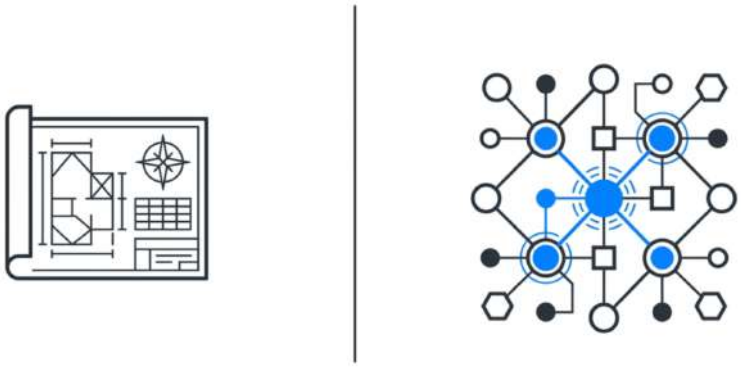


***Imagen 14.*** *Spec estática vs. viva*

La deriva ocurre cuando la spec dice una cosa y el código ha evolucionado hacia otra. Esto pasa más rápido de lo que parece: basta con que el agente tome una decisión menor no cubierta por la spec, o que el equipo haga un ajuste durante la implementación sin actualizar el documento. En pocas iteraciones, spec y código cuentan historias diferentes.

La deriva no solo inutiliza la spec como referencia futura; la convierte en un riesgo activo. Si un nuevo miembro del equipo (o una nueva sesión del agente) consulta la spec y la toma como verdadera, tomará decisiones basadas en información incorrecta.

## Los tres niveles de vida de la spec

La taxonomía de Böckeler (capítulo 1) describe cómo diferentes equipos y herramientas gestionan el ciclo de vida de la spec:

- **Spec-first**: la spec se escribe antes de codificar, se usa para guiar la tarea actual y se descarta al terminar. Es el nivel más básico de adopción. La spec es un artefacto efímero: cumple su función para la sesión de trabajo y desaparece. La deriva no es un problema porque no hay spec que mantener, pero también se pierde la documentación como activo del proyecto.
- **Spec-anchored**: la spec se mantiene después de completar la tarea y se usa como referencia para la evolución y el mantenimiento. Las specs se acumulan como documentación viva del sistema. La deriva es un riesgo real que requiere disciplina: cuando el código cambia, la spec debe actualizarse. El beneficio es que

cualquier persona (o agente) puede consultar la spec para entender qué hace una parte del sistema, por qué se diseñó así y qué restricciones aplican.

- **Spec-as-source**: la spec es el artefacto principal y permanente. Solo se edita la spec; el código no se modifica directamente. Cuando se necesita un cambio, se modifica la spec y se regenera el código. Es el nivel más radical y el que menos herramientas soportan actualmente. La deriva desaparece por diseño —el código siempre es derivado de la spec— pero la regeneración completa plantea desafíos prácticos de rendimiento y determinismo.

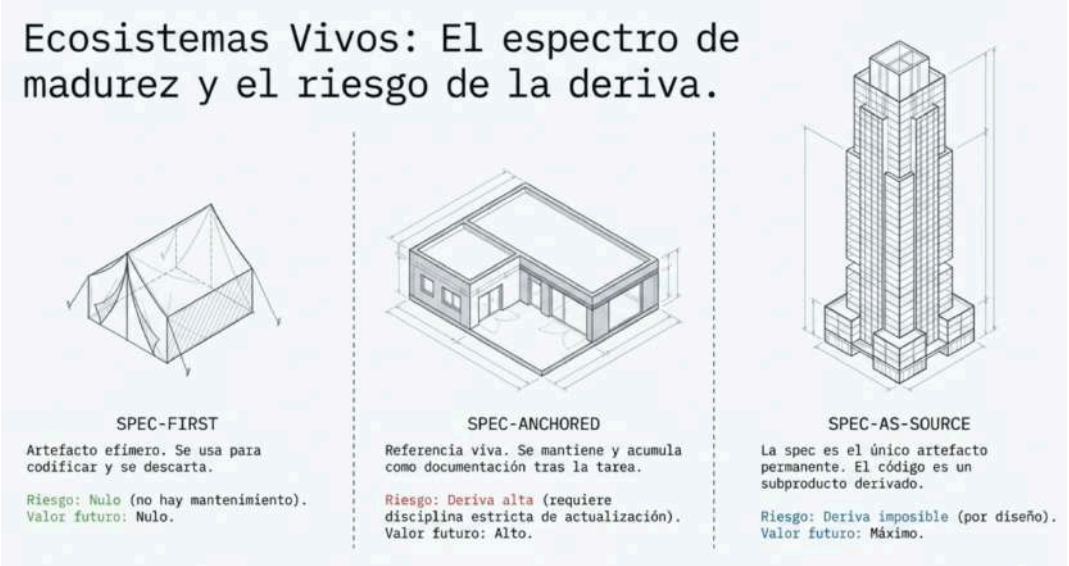



***Imagen 15.*** *Madurez y riesgo de deriva*

La mayoría de equipos y herramientas actuales operan en el nivel spec-first. Algunos están avanzando hacia spec-anchored. Spec-as-source es todavía experimental, aunque herramientas como Tessl Framework lo exploran.

## La Clarity Gate

Un concepto emergente en la comunidad de SDD que resulta útil como prueba de calidad de la spec es la **Clarity Gate** (puerta de claridad): ¿puede un agente diferente, o una sesión completamente nueva del mismo agente, generar código funcionalmente equivalente a partir de la misma spec?

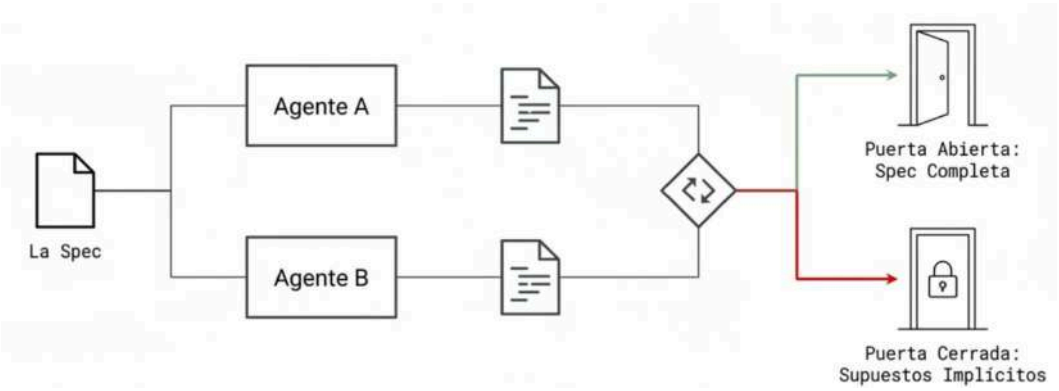



***Imagen 16.*** *La puerta de claridad: el test ácido de una spec*

Si la respuesta es sí, la spec es clara y completa: todo lo necesario está explícito en el documento.

Si la respuesta es no, la spec tiene supuestos implícitos. Hay conocimiento que vivía en el contexto de la conversación original pero que no quedó capturado en el documento. Estos supuestos implícitos son la causa principal de la deriva, porque cada nueva sesión los reinterpretará de forma diferente.

La Clarity Gate es una prueba sencilla que cualquier equipo puede aplicar: toma una spec terminada, dásela a otro agente (o a otra persona) sin contexto adicional y compara el resultado con el original. Las divergencias revelan exactamente dónde la spec era insuficiente.

## La spec como diagnóstico

Hay una observación habitual en la comunidad de practicantes que invierte la relación tradicional entre bugs y documentación: cuando el código falla, el problema suele originarse en la spec. Arreglar la spec suele arreglar el código; de lo contrario, solo se parchean síntomas mientras el problema real sigue generando nuevos bugs.

Esta observación tiene implicaciones profundas. Si una spec está bien mantenida, los bugs del código se pueden diagnosticar preguntando: ¿la spec es correcta? Si la spec es correcta y el código no la cumple, el problema es de implementación: se regenera la tarea. Si la spec es incorrecta, el problema es de especificación: se corrige la spec y se regenera.

Este patrón de diagnóstico es más limpio que el debugging tradicional, donde hay que determinar simultáneamente si el código hace lo que debería y si lo que debería hacer es lo correcto.

## Estrategias para mantener specs vivas

Para equipos que aspiran al nivel spec-anchored, estas estrategias ayudan a combatir la deriva:

**Actualizar la spec con cada cambio**. Cuando la implementación difiere de la spec —incluso por una razón justificada—, la spec se actualiza en el mismo commit o inmediatamente después. No «lo actualizaré luego». *Luego* no existe.

**Versionado de la spec**. Las specs deberían tener un historial de versiones que registre qué cambió y por qué. Basta una sección al final del documento con fecha, versión y resumen del cambio. El versionado convierte la spec en un registro de decisiones, no solo en un documento de requisitos.

**Revisión de la spec en la retrospectiva**. Si el equipo hace retrospectivas, incluir una revisión del estado de las specs: ¿cuántas están actualizadas? ¿Cuántas han derivado? ¿Qué specs fueron más útiles durante el sprint? Esta práctica convierte el mantenimiento de specs en una responsabilidad del equipo, no de un individuo.

**Tests de conformidad**. Para specs críticas, crear tests automatizados que verifiquen que el código cumple la spec. Estos tests actúan como un contrato ejecutable: si el código cambia de forma que incumple la spec, el test falla y obliga a decidir si se corrige el código o se actualiza la spec. Simon Willison —creador de Datasette y divulgador habitual sobre LLMs— los llama *conformance suites* y los propone como contratos independientes del lenguaje, normalmente en formato YAML.

## Cuándo dejar morir una spec

No todas las specs merecen mantenerse. Una spec para una funcionalidad pequeña y estable, que no va a evolucionar, puede tratarse como spec-first: se usa para la implementación y se descarta. Mantener una spec viva tiene un coste de mantenimiento, y ese coste solo se justifica si la spec sigue sirviendo: como referencia para nuevos miembros del equipo, como base para futuras evoluciones o como documentación de conformidad.

La decisión de qué specs mantener y cuáles descartar es parte de la gestión del backlog de documentación del equipo. Como cualquier otro artefacto, una spec que nadie consulta y nadie actualiza no es documentación viva: es documentación zombi (un anti-patrón que retomaremos en el capítulo 23).

# PARTE IV — SDD en el equipo ágil

## 16. Conexión con *Scrum en equipos con IA*

### Dos caras de la misma adaptación

La *guía Scrum* en equipos con IA definió cómo se organiza un equipo ágil cuando parte de los constructores son agentes. Introdujo roles adaptados (product architect, product builder, agile enabler), artefactos nuevos (Definition of Ready for AI, Dual Track) y un modelo de trabajo que reconoce la realidad de la construcción asistida. SDD es el complemento natural de ese marco: si scrum en equipos con IA describe la estructura del equipo, SDD describe cómo ese equipo construye software en el día a día.

No son dos metodologías separadas. Son dos niveles de la misma adaptación: el nivel organizativo (roles, ceremonias, artefactos de equipo) y el nivel operativo (cómo se especifica, diseña, descompone e implementa cada incremento de producto).

### Los roles y su encaje en SDD

**Product architect como diseñador y aprobador de specs**. El product architect, responsable de la visión del producto y de las decisiones de alto nivel, encuentra en SDD su medio natural de expresión. La fase de requisitos es donde ejerce su competencia principal: definir qué se construye, para quién y con qué criterios de éxito. En la puerta 1 es quien evalúa si los requisitos reflejan las necesidades reales del producto.

En la fase de diseño, el product architect participa validando que las decisiones técnicas son coherentes con la arquitectura del sistema y con la dirección del producto. No necesita escribir el diseño técnico —eso pueden hacerlo los product builders— pero sí aprobarlo en la puerta 2.

**Product builders como orquestadores de la implementación**. Los product builders son los profesionales que trabajan directamente con los agentes. En SDD, su rol se despliega a lo largo de todo el flujo: contribuyen a refinar los requisitos con su conocimiento técnico, diseñan la solución, descomponen el diseño en tareas atómicas y orquestan la implementación delegando a agentes.

**Agile enabler como facilitador de las puertas de aprobación**. El agile enabler encuentra en las puertas un mecanismo concreto de facilitación. Su responsabilidad no es aprobar —eso le corresponde al product architect y a los product builders según la puerta— sino asegurar que las puertas se ejecutan con el rigor necesario sin convertirse en cuellos de botella.

Esto implica varias funciones: asegurar que se dedica tiempo a la revisión (que no se saltan puertas por presión de calendario), moderar las discusiones cuando hay

desacuerdo sobre una spec, detectar cuándo el proceso está generando overhead innecesario y ajustar la intensidad, y vigilar que las specs no se conviertan en mini-waterfalls (que el equipo mantenga el ritmo iterativo dentro de cada ciclo).

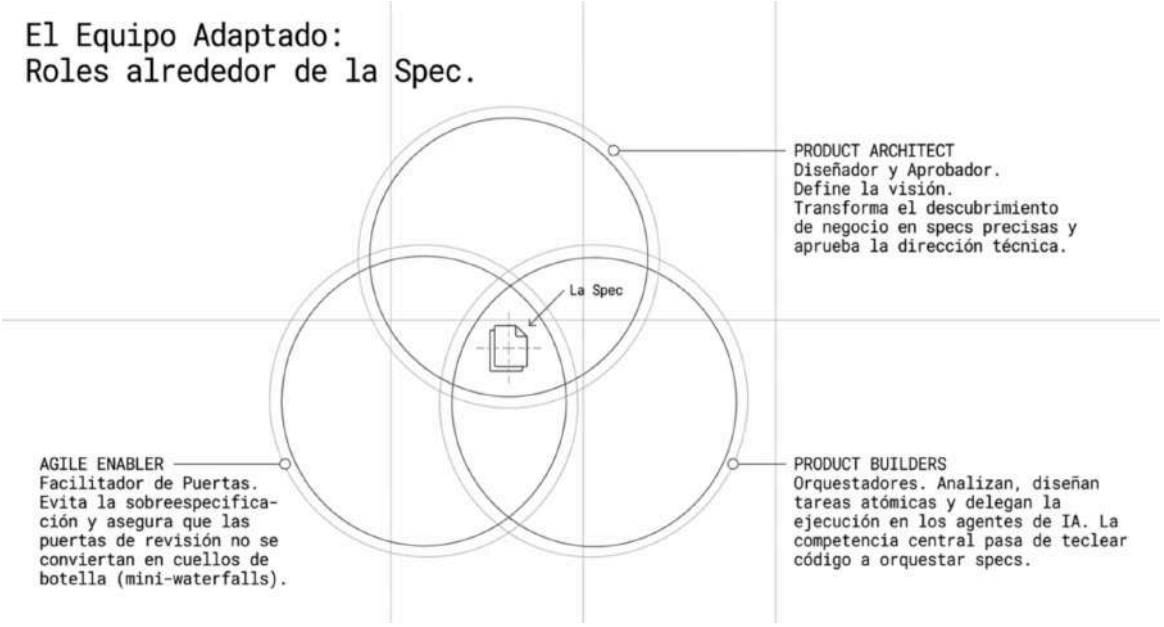



***Imagen 17.*** *Roles alrededor de la spec*

## El modelo de Dual Track (doble carril)

El modelo de *Double track* (discovery y delivery en paralelo) que introduce *Scrum en equipos con IA* tiene un paralelismo directo con la separación planificación/ejecución de SDD.

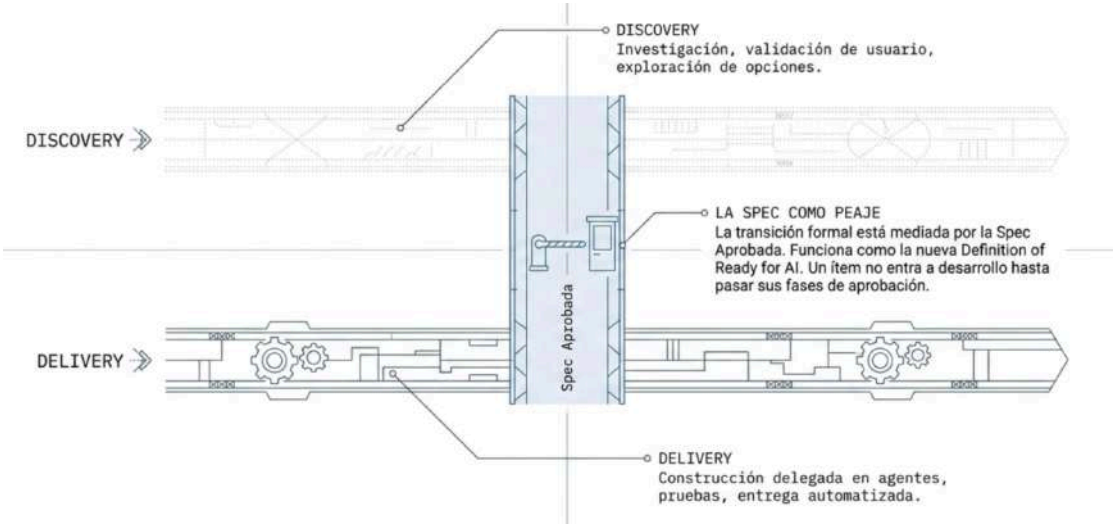



***Imagen 17.*** *El puente operativo: de discovery a delivery*

**El carril de discovery** —donde se investiga, se valida con usuarios, se exploran opciones— alimenta la fase de requisitos de SDD. Lo que se descubre en discovery se convierte en criterios de aceptación precisos que el agente puede implementar.

**El carril de delivery** —donde se construye, se prueba, se entrega— es el territorio de las fases 2, 3 y 4 de SDD (diseño, tareas, implementación).

La conexión clave es que SDD formaliza la transición entre ambos carriles. En un equipo sin SDD, la transición de discovery a delivery puede ser informal: alguien describe lo que se descubrió y el equipo empieza a construir. Con SDD, esa transición está mediada por una spec aprobada: un artefacto explícito que captura lo descubierto en discovery en un formato que el carril de delivery puede ejecutar. La spec es el puente entre descubrir y construir.

### Definition of Ready for AI

*Scrum en equipos con IA* introduce el Definition of Ready for AI: los criterios que un elemento del backlog debe cumplir antes de que pueda ser trabajado por un agente. Este concepto es el precursor natural de la spec de calidad en SDD.

En SDD, esta idea se sistematiza: un ítem no está *Ready* para implementación hasta que ha pasado por las fases de requisitos, diseño y tareas, y ha sido aprobado en las tres puertas correspondientes. El Definition of Ready for AI y el flujo de aprobación de SDD son, en esencia, el mismo concepto expresado en lenguajes diferentes: uno en el lenguaje de scrum (criterios de entrada al sprint) y otro en el lenguaje de SDD (puertas de aprobación entre fases). El equipo que usa ambos marcos no necesita duplicar esfuerzos: la spec aprobada ES el Definition of Ready para esa funcionalidad.

## 17. Impacto de SDD en cada rol

### Lo que cambia en el día a día

SDD no es una capa que se añade encima del trabajo existente sin modificarlo. Cambia la naturaleza del trabajo de cada persona del equipo. Entender esos cambios es requisito para una adopción exitosa. El capítulo 2 describió el desplazamiento general; este lo aterriza en el día a día de cada rol.

### Product owner / Product architect

**Lo que cambia:** la exigencia de precisión sube drásticamente. «Como usuario, quiero recibir notificaciones de cambios en mis documentos» es suficiente para iniciar una conversación productiva con un desarrollador humano, pero no para un agente que va a implementar literalmente lo que lea.

Esta exigencia puede percibirse inicialmente como una carga. Pero en la práctica, muchos product owners descubren que la disciplina de especificar con precisión mejora su propio entendimiento del producto. Las decisiones que antes se tomaban de forma implícita durante la implementación ahora se toman explícitamente durante la especificación, que es donde son más baratas de cambiar.

**Lo que gana**: las specs son artefactos que cualquier stakeholder puede leer. El product owner puede mostrar una spec a un director o a un cliente y recibir feedback sobre lo que se va a construir antes de construirlo. Esto acorta el ciclo de feedback con el negocio de forma significativa.

## Scrum master / Agile enabler

**Lo que cambia**: aparecen nuevos puntos de facilitación y nuevos riesgos que gestionar.

Las puertas de aprobación son eventos que necesitan facilitación. No son reuniones formales necesariamente —pueden ser revisiones asíncronas en un pull request de la spec— pero requieren que alguien asegure que ocurren, que participan las personas adecuadas y que la aprobación es un acto deliberado.

Los riesgos nuevos que el agile enabler debe vigilar:

- **Sobreespecificación**: cuando el equipo dedica más tiempo a perfeccionar la spec que a producir software funcionando.
- **Puertas como cuellos de botella**: cuando una aprobación se bloquea porque la persona que debe aprobar no está disponible. El agile enabler debe facilitar mecanismos de delegación o aprobación asíncrona.
- **Teatro de especificación**: cuando se escriben specs que nadie revisa realmente. El proceso se ejecuta mecánicamente sin aportar el valor de la revisión.
- **Rigidez excesiva**: cuando el equipo aplica SDD con la misma intensidad a un bug trivial que a una funcionalidad compleja.

**Lo que gana**: las puertas proporcionan puntos de inspección y adaptación estructurados que son más concretos que los del scrum tradicional. En lugar de preguntar en la daily «¿cómo va el desarrollo?», se puede preguntar «¿la spec de la funcionalidad X ha pasado la puerta 2?». Los bloqueos se hacen visibles antes.

## Desarrollador / Product builder

**Lo que cambia:** la competencia central se desplaza del código a la orquestación de specs y agentes, como vimos en el capítulo 2.

Un product builder que adopta SDD pasa de escribir código a analizar, especificar, orquestar y revisar. La actividad de teclear sintaxis deja de ser la actividad principal.

**Lo que gana**: la capacidad de producir software a una velocidad que antes era imposible. Un product builder con SDD puede completar en una tarde lo que antes requería días, porque la implementación está delegada a agentes que trabajan en paralelo. Pero esa velocidad solo es posible si las fases previas (requisitos, diseño, tareas) se han hecho bien. La inversión de esfuerzo se desplaza de la ejecución a la planificación.

### Stakeholders

**Lo que cambia:** ganan visibilidad sobre lo que se construye.

Las specs son legibles por personas no técnicas. Un director puede leer un documento de requisitos y opinar sobre si las prioridades son correctas. Un responsable de compliance puede revisar los criterios de aceptación y verificar que cumplen los requisitos regulatorios. Un usuario clave puede evaluar si los comportamientos descritos en la spec coinciden con lo que necesita.

Esta visibilidad transforma la relación entre el equipo de desarrollo y el resto de la organización. La spec reduce la asimetría de información que tradicionalmente ha existido entre quienes construyen software y quienes lo pagan o lo usan.

## 18. Ceremonias y artefactos ágiles con SDD

### El Sprint bajo SDD

SDD no elimina el sprint ni lo sustituye. Lo que cambia es qué contiene el sprint y cómo se ejecutan las ceremonias dentro de él.

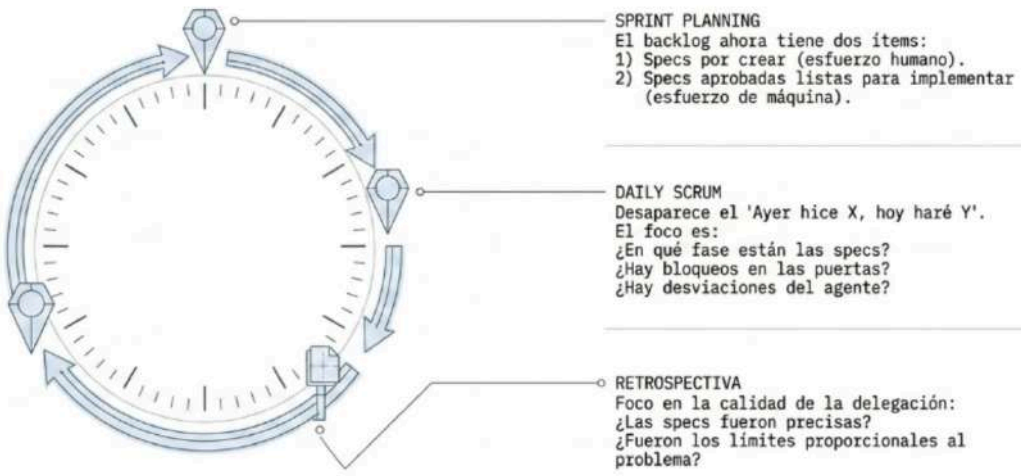



***Imagen 18.*** *Ceremonias ágiles con SDD*

## Sprint planning con specs

En un sprint planning tradicional, el equipo selecciona ítems del backlog, los discute, los estima y se compromete con un sprint goal. Con SDD, el sprint planning adquiere una dimensión adicional: no solo se seleccionan qué funcionalidades se van a construir, sino qué specs se van a desarrollar y aprobar.

Esto implica que el backlog contiene dos tipos de ítems:

- **Specs por crear**: funcionalidades que necesitan pasar por el flujo de SDD (fases 1-3) antes de poder implementarse.
- **Specs aprobadas listas para implementar**: funcionalidades cuyas specs ya han pasado todas las puertas y están listas para la fase 4.

El sprint planning se centra en dos preguntas: ¿qué specs nuevas vamos a desarrollar este sprint? y ¿qué specs aprobadas vamos a implementar? Es posible —y habitual— que un sprint contenga ambos tipos de trabajo: se implementan specs que se aprobaron en el sprint anterior mientras se desarrollan specs para el sprint siguiente. Esto crea un flujo continuo donde la planificación y la ejecución se solapan de forma natural.

La estimación también cambia. En el modelo tradicional, se estima el esfuerzo de implementación. Con SDD, hay dos estimaciones relevantes: el esfuerzo de especificación (humano) y el esfuerzo de implementación (mayoritariamente del agente). El primero es el que realmente consume capacidad del equipo; el segundo es predecible una vez que las tareas están bien definidas.

## Definition of Done con código generado

¿Qué significa «terminado» cuando el código es generado por un agente?

La Definition of Done tradicional suele incluir criterios como: código revisado por pares, tests pasando, documentación actualizada, desplegado en entorno de staging. Con SDD, la Definition of Done se enriquece con criterios específicos:

- El código cumple todos los criterios de éxito definidos en las tareas.
- El código respeta las restricciones del documento de diseño.
- Los commits son atómicos (un commit por tarea).
- La spec ha sido actualizada si la implementación difirió de lo especificado.
- La revisión de código se ha hecho contra la spec, no solo contra el criterio del revisor.

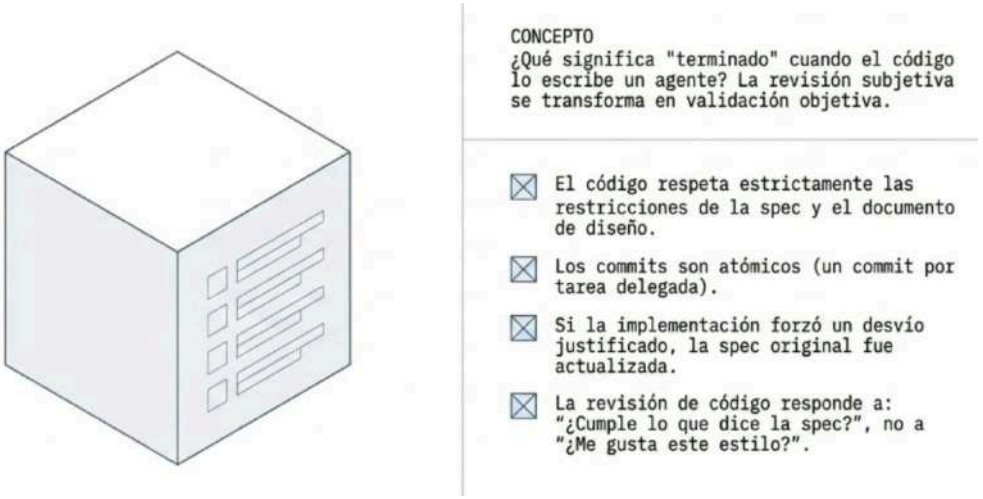



***Imagen 19.*** *Definición de "done" con código generado*

El último punto es particularmente importante: en SDD, la revisión de código no es «¿me parece bien este código?» sino «¿este código cumple lo que la spec dice?». Esto objetiva la revisión y reduce las discusiones subjetivas sobre estilo o preferencias personales.

## Daily scrum con agentes

La daily scrum cambia de naturaleza cuando parte del trabajo lo ejecutan agentes. Los agentes no necesitan reportar en una reunión, pero los humanos que los orquestan sí necesitan coordinar el trabajo.

El foco de la daily se desplaza hacia:

- **Estado de las specs**: ¿en qué fase están las specs que estamos desarrollando? ¿Hay alguna puerta pendiente? ¿Quién la va a revisar?
- **Resultados de implementación**: ¿las tareas delegadas a agentes han producido resultados aceptables? ¿Hay tareas bloqueadas que necesitan intervención humana?
- **Desviaciones**: ¿el agente ha tomado alguna decisión que difiere de la spec? ¿Necesitamos actualizar la spec o corregir la implementación?

Lo que desaparece (o debería desaparecer) es el repaso mecánico de «ayer hice X, hoy haré Y». Con SDD, el estado del trabajo es visible a través de las fases y puertas del proceso. La daily se centra en lo que no es visible: bloqueos, decisiones pendientes, desviaciones.

## Sprint review

La sprint review sigue cumpliendo su función original: inspeccionar el incremento de producto y adaptar el backlog. Pero con SDD, la Review puede ir un paso más allá.

Además de demostrar el software funcionando, el equipo puede presentar las specs desarrolladas durante el sprint como artefacto compartible. Los stakeholders pueden revisar no solo qué se construyó sino qué se especificó para el próximo sprint. Esto crea un bucle de feedback anticipado: el stakeholder puede opinar sobre una spec (legible y barata de cambiar) antes de que se convierta en código.

### Retrospectiva

La Retrospectiva incorpora nuevas dimensiones de inspección:

- **Calidad de las specs**: ¿fueron suficientemente precisas? ¿Hubo funcionalidades donde la implementación difirió significativamente de la spec? ¿Qué se podría haber especificado mejor?
- **Eficacia de las puertas**: ¿detectaron problemas a tiempo? ¿Alguna puerta se saltó o se ejecutó de forma superficial?
- **Calibración del proceso**: ¿la intensidad de SDD fue proporcional al tamaño de los problemas? ¿Hubo funcionalidades donde SDD fue excesivo o insuficiente?
- **Calidad de la delegación**: ¿los prompts de las tareas fueron claros? ¿Las restricciones fueron adecuadas? ¿Los criterios de éxito capturaron lo que realmente importaba?

Estas preguntas complementan las de la retrospectiva tradicional con un foco específico en la interacción entre el equipo humano y los agentes.

## 19. SDD en codebase heredado y proyectos brownfield

### La realidad de la mayoría de equipos

La mayor parte de la literatura sobre SDD asume proyectos greenfield: se parte de cero, se especifica todo y se genera código nuevo. Pero la realidad de la mayoría de equipos profesionales es diferente: trabajan sobre codebases existentes, a menudo grandes, a veces heredados, con patrones establecidos, deuda técnica acumulada y restricciones heredadas.

SDD en entornos brownfield no solo es posible: es donde más valor aporta, porque las consecuencias de un agente que no entiende el contexto existente son mucho peores que en un proyecto nuevo.

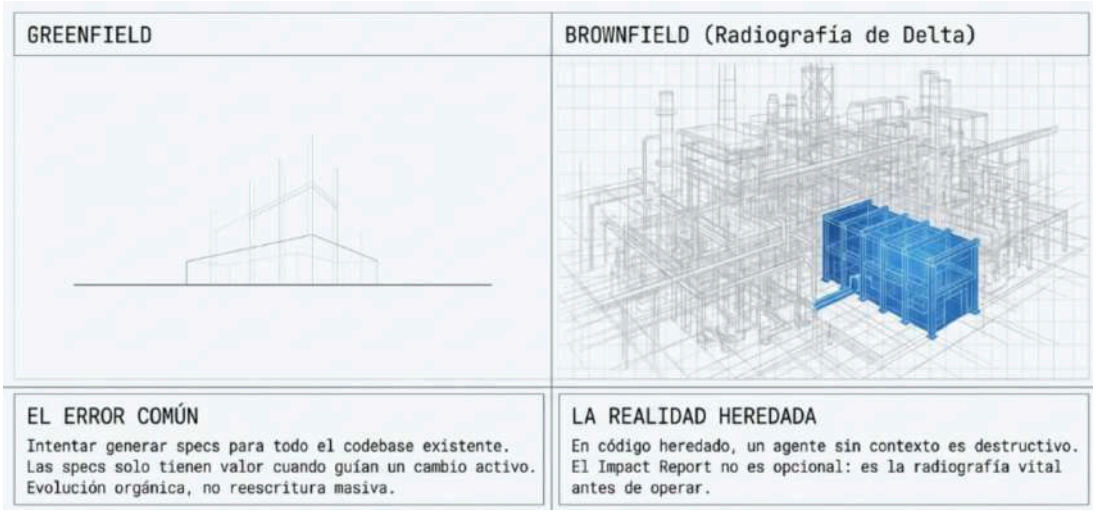



***Imagen 20.*** *SDD en entornos heredados (brownfield)*

## El Impact Report como pieza esencial

En un proyecto greenfield, el Impact Report es opcional en la primera iteración. En un proyecto brownfield es obligatorio siempre. Sin un análisis del código existente, los requisitos y el diseño se escriben en el vacío, y el agente producirá código que puede ser técnicamente correcto pero incoherente con el sistema existente.

El Impact Report en un entorno brownfield responde a preguntas que no existen en greenfield: ¿qué módulos del sistema se verán afectados? ¿Qué patrones usa el código existente que debemos mantener? ¿Hay tests que se romperán? ¿Hay dependencias no documentadas entre componentes? ¿Hay deuda técnica que este cambio podría agravar o que deberíamos resolver de paso?

## El formato delta

Una de las contribuciones más prácticas al SDD brownfield es el formato delta, popularizado por OpenSpec. En lugar de escribir una spec completa del sistema —algo impracticable en un codebase existente grande— la spec delta describe solo lo que cambia: qué se añade, qué se modifica, qué se elimina.

El formato es directo:

- **ADDED** (añadido): funcionalidades completamente nuevas que no existían.
- **MODIFIED** (modificado): cambios en comportamiento existente. Incluye la versión completa revisada del requisito, no solo la diferencia.
- **REMOVED** (eliminado): funcionalidades que se deprecian o eliminan.

La spec delta es más manejable que una spec completa porque se centra en la superficie de cambio, no en todo el sistema. Un equipo que trabaja sobre un codebase de cien mil

líneas no necesita especificar las cien mil líneas: solo necesita especificar las líneas que van a cambiar y su relación con las que no cambian.

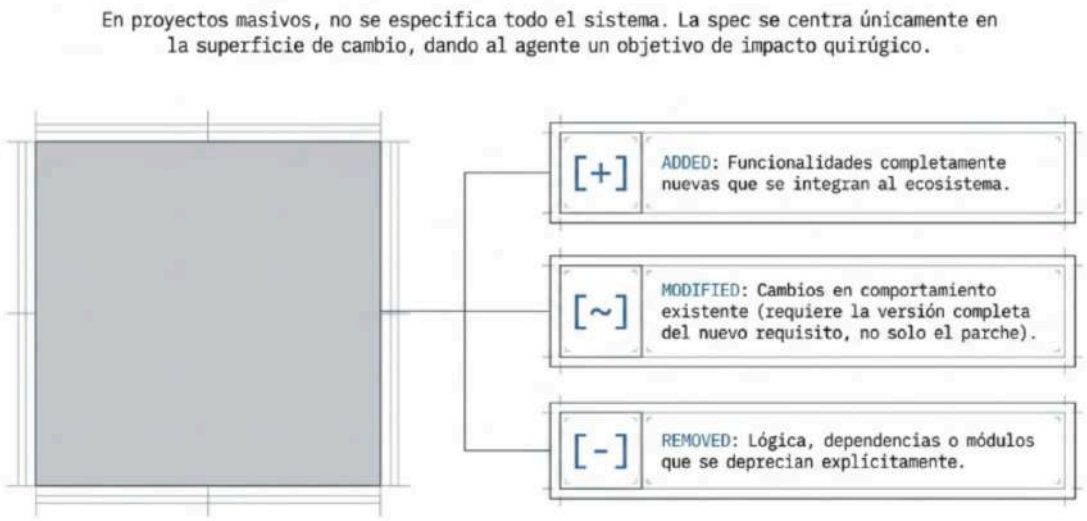



***Imagen 21.*** *Especificando el cambio: el formato delta*

Además, el formato delta funciona mejor con agentes en entornos brownfield: en lugar de pedirle al agente que reinterprete todo el sistema, se le da un objetivo preciso —«implementa esta diferencia»— que es más fácil de revisar, más fácil de evolucionar y menos propenso a la deriva.

## Adopción gradual: empezar con una funcionalidad

Un error frecuente al adoptar SDD en un proyecto brownfield es intentar generar specs para todo el codebase existente de una vez. Esto es un desperdicio de esfuerzo: las specs solo tienen valor cuando guían un cambio activo, no cuando documentan código estable que nadie va a modificar.

La estrategia recomendada es empezar con una única funcionalidad nueva o un cambio significativo, aplicar SDD completo a ese cambio y evaluar el resultado. Las specs se acumulan orgánicamente a través del trabajo real: cada vez que se modifica una parte del sistema, se genera una spec para ese cambio. Con el tiempo, las partes más activas del codebase tendrán specs y las partes estables no, que es exactamente la distribución correcta de esfuerzo.

El manifiesto del proyecto OpenSpec lo expresa así: no se trata de generar specs para todo el codebase heredado por adelantado, sino de acumular specs orgánicamente a través del trabajo real.

## La constitución del proyecto como primer paso

Para equipos que trabajan en brownfield, hay un artefacto que sí vale la pena crear antes de empezar con SDD: la **constitución del proyecto** (CLAUDE.md, agents.md o equivalente). Este documento describe las convenciones del proyecto, el stack tecnológico, la estructura de directorios, los patrones de diseño establecidos y las reglas que cualquier cambio debe respetar.

La constitución no especifica funcionalidades: especifica el contexto en el que las funcionalidades se construyen. Es el equivalente a decirle a un desarrollador nuevo: «así es como hacemos las cosas aquí». Para un agente, este contexto es tan importante como los requisitos específicos, porque sin él cada spec se escribe en el vacío y el agente tomará decisiones que pueden ser técnicamente correctas pero culturalmente incorrectas para el proyecto.

La constitución es un esfuerzo de una sola vez (con actualizaciones ocasionales) que paga dividendos en cada spec posterior.

## Cuándo un cambio en brownfield justifica una spec

No todo cambio en un codebase heredado necesita SDD. El principio de proporcionalidad del capítulo 4 sigue vigente, con matices específicos del entorno brownfield:

- **Un bug fix localizado** (afecta a uno o dos ficheros, causa conocida): no necesita spec. Instrucción directa al agente con el contexto del bug.
- **Una funcionalidad nueva mediana** (afecta a tres-diez ficheros, requiere decisiones de diseño): spec completa con las cuatro fases. Es el caso ideal para SDD.
- **Un refactoring significativo** (afecta a muchos ficheros, cambia patrones establecidos): spec con énfasis especial en el Impact Report y el diseño. Las tareas deben ser especialmente granulares para que cada una sea reversible.
- **Una migración técnica** (cambio de framework, de base de datos, de arquitectura): SDD con fase de investigación previa (subagentes que exploran opciones) y spec detallada que documenta la estrategia de migración.

La decisión no depende de la herramienta sino del juicio del equipo, y es una de las habilidades que el agile enabler ayuda a desarrollar.

# PARTE V — Ecosistema, herramientas y decisiones

## 20. Panorama de herramientas SDD (2026)

### Un ecosistema en expansión rápida

El ecosistema de herramientas SDD ha crecido con una velocidad notable. En menos de un año hemos pasado de unas pocas propuestas experimentales a un panorama con múltiples opciones maduras, cada una con una filosofía diferente. Esto es señal de que SDD responde a una necesidad real, pero también significa que elegir herramienta requiere entender qué problema resuelve cada una.

La distinción fundamental que atraviesa todo el ecosistema es la que Augment Code identifica entre dos grandes categorías: **plataformas de spec viva** que mantienen la documentación sincronizada con el código mientras los agentes trabajan, y **herramientas de spec estática** que estructuran los requisitos por adelantado pero requieren reconciliación manual cuando la implementación diverge.

### Las herramientas principales

#### GitHub Spec Kit

Es un kit de herramientas CLI de código abierto (licencia MIT) que sitúa las specs en el centro del proceso de ingeniería mediante un flujo de cuatro fases (Specify, Plan, Tasks, Implement). Su ventaja es la portabilidad: soporta más de una docena de agentes diferentes, incluyendo herramientas de línea de comandos (Claude Code, Gemini CLI, Codex CLI) e IDEs (GitHub Copilot, Cursor, Windsurf). Los artefactos son ficheros Markdown planos que se integran nativamente con la interfaz web de GitHub y con cualquier flujo de control de versiones.

**Spec Kit** incluye el concepto de *constitución*: un documento fundacional que codifica principios inmutables del proyecto que todos los agentes deben respetar. Es la herramienta de referencia para equipos que necesitan portabilidad entre agentes sin acoplarse a un proveedor.

**Sus limitaciones:** genera output verboso que requiere revisión humana cuidadosa, no está optimizado para cambios pequeños (un fix trivial puede producir documentación desproporcionada) y las specs son estáticas (no se autoactualizan durante la implementación).

#### Kiro (AWS)

Es un IDE completo (fork de VS Code) desarrollado por AWS que integra SDD directamente en el entorno de desarrollo. Su flujo guía al usuario por tres fases

—Requirements, Design, Tasks— generando automáticamente los artefactos de cada fase. Kiro también admite hooks: agentes dirigidos por eventos que se activan al guardar ficheros o hacer commits (actualizando tests o documentación automáticamente, por ejemplo).

Su fortaleza es la integración nativa con el ecosistema AWS y la experiencia guiada que reduce la barrera de entrada. Su debilidad es el acoplamiento a AWS y la rigidez del flujo: para tareas pequeñas, el proceso puede resultar excesivo. La propia Böckeler lo documentó al observar que un bug pequeño se convirtió en cuatro historias de usuario con dieciséis criterios de aceptación —«como usar un mazo para romper una nuez»—.

**OpenSpec**

Es un framework de código abierto diseñado específicamente para el trabajo en codebases existentes (brownfield). Su innovación principal es el formato delta (ADDED, MODIFIED, REMOVED) que describe qué cambia en el sistema en lugar de especificar el sistema completo.

OpenSpec aplica una máquina de estados estricta en tres fases (Propose, Apply, Archive) que asegura que la documentación no se quede atrás. Su output es ligero —en torno a 250 líneas frente a las 800 aproximadas que suele producir Spec Kit para funcionalidades comparables—, lo que reduce el overhead de revisión.

**Su limitación principal:** las specs no se autoactualizan durante la implementación. Si el agente modifica el enfoque a mitad de tarea, el documento de propuesta no refleja esos cambios automáticamente.

**BMAD Method**

*Breakthrough Method for Agile AI-Driven Development* es un framework de código abierto de escala enterprise que orquesta múltiples agentes especializados a lo largo de un flujo SDLC completo. Define una veintena de agentes con roles específicos, permisos de acceso explícitos al contexto y puntos de traspaso discretos entre ellos.

Es la opción más completa y más pesada. Donde Spec Kit y OpenSpec guían a un solo asistente, BMAD simula un equipo de desarrollo completo con agentes especializados (analista de requisitos, arquitecto, desarrollador, tester, revisor). Es adecuado para proyectos complejos con lógica de dominio elaborada, pero su curva de aprendizaje y overhead de configuración lo hacen excesivo para proyectos pequeños o medianos.

**Intent (Augment Code)**

Se diferencia del resto por su arquitectura de spec viva: mantiene las specs sincronizadas con el código de forma bidireccional mientras los agentes trabajan.

Cuando un agente modifica el enfoque durante la implementación, la spec se actualiza automáticamente. Esto elimina el problema de la deriva que afecta a las herramientas de spec estática.

Intent destaca en arquitecturas multi-servicio complejas y permite traer tu propio agente (BYOA, *Bring Your Own Agent*). Su modelo de coste —actualmente basado en consumo— puede ser un factor relevante para equipos con uso intensivo.

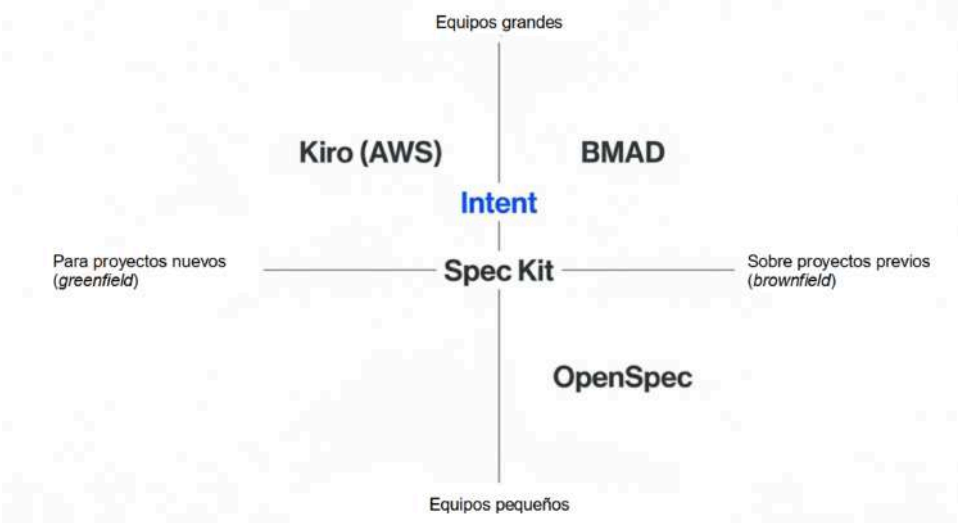



***Imagen 22.*** *Herramientas / idoneidad*

**Capacidades nativas de los agentes**

Además de los frameworks dedicados, los propios agentes (Claude Code, Cursor, GitHub Copilot) ofrecen capacidades nativas que soportan SDD sin herramientas externas: ficheros de configuración del proyecto (CLAUDE.md, .cursorrules, agents.md), modos de planificación (*Plan Mode* en Claude Code), subagentes, sistemas de tareas y hooks.

Para muchos equipos, estas capacidades nativas son suficientes para practicar SDD sin adoptar un framework adicional. La ventaja es la simplicidad y la ausencia de dependencias; la limitación es que la disciplina del flujo depende del equipo, no de la herramienta.

## No existe «la mejor herramienta»

La tentación de buscar «la mejor» herramienta SDD es comprensible pero contraproducente. Cada herramienta optimiza para un perfil diferente de equipo y de proyecto:

- Equipos que necesitan portabilidad entre agentes → **Spec Kit.**
- Equipos con fuerte presencia AWS y proyectos greenfield → **Kiro.**
- Equipos que trabajan mayoritariamente en codebase heredado → **OpenSpec.**
- Proyectos enterprise con lógica de dominio compleja → **BMAD.**

- Equipos que necesitan sincronización bidireccional en arquitecturas multi-servicio → **Intent.**
- Equipos que prefieren simplicidad y control directo → **capacidades nativas del agente.**

La elección correcta depende del contexto, no de las características de la herramienta en abstracto. Y como SDD es una metodología antes que una herramienta, es perfectamente viable practicar SDD con cualquiera de estas opciones —o sin ninguna de ellas, usando solo disciplina de equipo y ficheros Markdown—.

## 21. Nativo frente a framework: criterios de decisión

### La pregunta práctica

Una vez que un equipo decide adoptar SDD, la siguiente pregunta es: ¿necesitamos un framework dedicado o las capacidades nativas de nuestro agente son suficientes?

> No hay una respuesta universal. Hay criterios que ayudan a decidir.

### Cuándo las capacidades nativas son suficientes

- **El equipo es pequeño** (una a tres personas trabajando con agentes). La coordinación es sencilla y la disciplina del flujo se mantiene por comunicación directa.
- **El proyecto es homogéneo en tecnología y en agente.** Se usa un solo agente (por ejemplo, sólo Claude Code o solo Cursor) y no hay necesidad de portabilidad.
- **La complejidad de las specs es moderada**. Las specs cubren funcionalidades individuales, no sistemas completos con dependencias entre múltiples servicios.
- **El equipo tiene madurez técnica** para mantener la disciplina del flujo sin que una herramienta la imponga. Saben cuándo hacer un Impact Report, cuándo escribir una spec completa y cuándo un prompt directo es suficiente.

En estos casos, un CLAUDE.md bien mantenido, un flujo disciplinado de fases y el uso de subagentes y tareas nativos proporcionan todo lo necesario. Añadir un framework sería overhead sin retorno.

### Cuándo un framework aporta valor

- **El equipo es más grande** (cuatro o más personas) y necesita un flujo estandarizado que todos sigan. El framework impone disciplina que de otro modo dependería de convenciones frágiles.
- **Se usan múltiples agentes** o se necesita la posibilidad de cambiar de agente sin reescribir las specs. La portabilidad del formato es valiosa.

- **El proyecto es regulado** y requiere trazabilidad formal desde requisitos hasta código. Un framework con puertas de aprobación estructuradas proporciona esa trazabilidad.
- **El codebase es grande y heredado** y necesita un enfoque sistemático para gestionar los cambios. El formato delta y las puertas de aprobación de un framework reducen el riesgo.
- **El equipo es nuevo en SDD** y necesita andamiaje. Un framework con fases guiadas y comandos predefinidos reduce la curva de aprendizaje.

## La vía intermedia: skills y *slash commands*

Entre el enfoque puramente nativo y un framework completo existe una vía intermedia: crear *skills* personalizados (paquetes reutilizables de instrucciones) o *slash commands* (comandos que el usuario invoca precedidos por una barra, como `/spec-driven`) que automatizan el flujo de SDD dentro del agente. Por ejemplo, un comando `/spec-driven` que lanza un flujo de cuatro fases con prompts predefinidos, generación de documentos en una carpeta `specs/` y puertas de aprobación interactivas.

Este enfoque combina la ligereza de lo nativo con la estructura de un framework. Es especialmente adecuado para equipos que quieren estandarizar su flujo sin adoptar una dependencia externa.

Proyectos como cc-sdd ofrecen exactamente esto: skills que se instalan sobre agentes existentes (Claude Code, Codex, Cursor, Copilot, Gemini CLI, entre otros) y proporcionan comandos SDD sin imponer un framework pesado.

## Guías complementarias de plataforma

Esta guía describe SDD como metodología independiente de la herramienta. Para la implementación práctica con herramientas concretas, Scrum Manager ofrece (o ofrecerá) guías complementarias específicas:

- **SDD con Claude Code**: implementación usando CLAUDE.md, subagentes, Tasks y Hooks nativos.
- *(Otras guías de plataforma se añadirán según demanda)*.

Estas guías cubren la configuración, los comandos, los workflows paso a paso y los ejemplos prácticos específicos de cada herramienta. Se actualizan independientemente de esta guía central cuando las herramientas versionan o añaden funcionalidades.

# PARTE VI — Madurez y anti-patrones

## 22. Métricas: ¿está funcionando SDD?

### Qué medir

Adoptar SDD sin medir su impacto es un acto de fe. Y los actos de fe no sobreviven al primer trimestre con presión de entrega. Para que SDD sea sostenible, el equipo necesita indicadores que muestren si el proceso está aportando valor o generando overhead.

Las métricas útiles para evaluar SDD se dividen en dos categorías: métricas de flujo (¿el proceso funciona?) y métricas de resultado (¿el producto mejora?).

### Métricas de flujo

**Tiempo de especificación frente a tiempo de implementación:** la proporción entre el tiempo que el equipo dedica a las fases 1-3 (requisitos, diseño, tareas) y el tiempo de la fase 4 (implementación). No hay una proporción «correcta» —depende de la complejidad—, pero un equipo maduro en SDD normalmente invierte más tiempo en especificación que en implementación. Si la implementación sigue consumiendo la mayor parte del tiempo, probablemente las specs no están lo suficientemente bien hechas para que la delegación al agente sea efectiva.

**Tasa de aprobación en primera revisión**: el porcentaje de specs que pasan las puertas sin necesidad de revisión. Una tasa alta indica que el equipo ha desarrollado criterio para escribir buenas specs. Una tasa baja no es necesariamente mala al inicio —las puertas están haciendo su trabajo de detección—, pero debería mejorar con el tiempo.

**Frecuencia de reescritura de prompts**: cuántos prompts de tarea han necesitado reescribirse durante el sprint (se revisa en la retrospectiva). Si este número es alto, la planificación inicial es demasiado vaga. Si es cero, puede que el equipo no esté reflexionando lo suficiente sobre la calidad de sus prompts.

**Tasa de desviación**: cuántas veces durante el sprint el agente se desvió de los parámetros definidos en la spec y requirió intervención humana. Mide directamente la calidad de las specs como instrucciones operativas.

### Métricas de resultado

**Calidad del primer intento (first-pass quality):** qué porcentaje del código generado cumple los criterios de éxito sin necesidad de regeneración. Es la métrica más directa de la calidad de la spec: una buena spec produce buen código al primer intento.

**Tiempo de rework**: el tiempo dedicado a corregir código que fue generado pero no cumple los requisitos. Debería disminuir a medida que las specs mejoran. Si no disminuye, las specs no están capturando lo que importa.

**Cobertura de specs sobre el backlog**: qué porcentaje de los ítems del backlog que fueron implementados tenían una spec aprobada. No necesita ser el 100 % (no todo necesita SDD), pero debería cubrir las funcionalidades de complejidad media y alta.

**Tiempo de revisión de código**: cuánto tiempo dedica el equipo a revisar el código generado. Con SDD maduro, este tiempo debería reducirse porque la revisión se hace contra la spec (verificación objetiva) en lugar de contra el criterio del revisor (evaluación subjetiva).

### Qué no medir

Es tentador medir la productividad por líneas de código generadas o por velocidad de implementación. Estas métricas son contraproducentes en SDD: el objetivo no es generar más código más rápido, sino generar el código correcto a la primera. Un equipo que, tras adoptar SDD, ve caer su output en un 30 % puede estar, en realidad, mejorando: genera menos líneas porque las que genera son correctas y evita las correcciones posteriores que antes absorbían el tiempo.

También hay que evitar medir el volumen de specs producidas como indicador de calidad. Más specs no es mejor; mejores specs sí lo es.

## 23. Anti-patrones y errores comunes

### Sobreespecificación

**Síntoma:** la spec es más larga que el código que genera. El equipo dedica horas a perfeccionar documentos que el agente traduce mecánicamente a código trivial.

**Causa**: no calibrar la intensidad de SDD al tamaño del problema. Aplicar el flujo completo de cuatro fases a un cambio que podría resolverse con un prompt directo.

**Solución**: el principio de proporcionalidad (capítulos 4, 12 y 19). No todo necesita SDD. Un bug localizado, un ajuste de configuración, un cambio cosmético: estas tareas funcionan mejor con una instrucción directa. SDD se reserva para funcionalidades de complejidad media y alta donde la inversión en especificación se amortiza en menos correcciones posteriores.

## Documentación zombi

**Síntoma:** el proyecto acumula specs que nadie consulta, nadie actualiza y nadie elimina. Las specs están desactualizadas respecto al código y cualquiera que las consulte obtendrá información incorrecta.

**Causa**: adoptar spec-anchored (mantener las specs vivas) sin establecer un proceso de mantenimiento. Las specs se crean con rigor en el primer ciclo y se abandonan en los siguientes.

**Solución**: decidir conscientemente qué specs mantener (spec-anchored) y cuáles descartar tras la implementación (spec-first). No todas las specs merecen mantenerse. Las que se mantienen deben tener un responsable y un mecanismo de actualización (actualización en el mismo commit que el código, revisión en la retrospectiva).

## El teatro de la especificación

**Síntoma:** el equipo escribe specs y ejecuta las puertas, pero la revisión es superficial. Nadie lee realmente la spec en profundidad. Las puertas se convierten en un trámite que se firma sin evaluar.

**Causa**: presión de tiempo que convierte las puertas en un obstáculo a superar rápido en lugar de un punto de inspección genuino. También puede indicar que el equipo no percibe valor en la revisión, quizá porque las specs son demasiado genéricas para que la revisión aporte algo.

**Solución**: si las puertas no aportan valor, el problema puede ser que las specs sean insuficientemente específicas (revisar una spec vaga es una pérdida de tiempo) o que se aplique SDD a tareas que no lo necesitan. El agile enabler debe detectar este patrón y ajustar: o se mejora la calidad de las specs para que la revisión aporte valor, o se reduce la intensidad del proceso para las tareas donde SDD es excesivo.

## SDD como herramienta de control

**Síntoma:** las specs se usan para micro-gestionar al equipo. Cada decisión debe estar en una spec aprobada. La autonomía del product builder desaparece porque no puede tomar ninguna decisión técnica sin pasar por una puerta.

**Causa**: confundir la estructura del proceso con el control del equipo. SDD está diseñado para dar claridad y reducir la ambigüedad, no para eliminar la autonomía profesional.

**Solución**: recordar el sistema de boundaries (Always / Ask First / Never). El nivel *Always* existe precisamente para dar autonomía: las decisiones rutinarias no necesitan aprobación. Las puertas de aprobación son para las decisiones de alto impacto, no para

todas las decisiones. Si el equipo siente que las puertas limitan su autonomía en lugar de apoyar su trabajo, la calibración está mal.

## El Markdown infinito

**Síntoma:** SDD produce cantidades masivas de ficheros Markdown. El equipo pasa más tiempo leyendo documentación generada por el agente que pensando en el producto. Las specs contienen repeticiones, casos límite imaginarios y refinamientos excesivos.

**Causa**: esta es una crítica documentada del SDD actual. Marmelab la describió en *Spec-Driven Development: The Waterfall Strikes Back* (noviembre de 2025) observando que los desarrolladores pueden acabar dedicando la mayor parte de su tiempo a leer ficheros Markdown extensos, buscando errores básicos ocultos en prosa verbosa que suena experta.

**Solución**: revisar la configuración de los prompts de generación de specs. Si la herramienta genera specs verbosas, ajustar los prompts para producir specs más concisas. OpenSpec, por ejemplo, produce specs de alrededor de 250 líneas frente a las 800 de Spec Kit: la elección de herramienta importa. También aplicar la heurística de Osmani: una spec más inteligente, no más larga. Si una spec tiene 50 páginas, probablemente necesita dividirse en specs más pequeñas, no leerse entera.

## Falso ágil: historias de usuario que no lo son

**Síntoma:** las specs generadas por la herramienta SDD contienen «historias de usuario» que son en realidad instrucciones técnicas disfrazadas. Ejemplo: «Como administrador de sistema, quiero que la relación de referido se almacene en la base de datos».

**Causa**: las herramientas SDD generan artefactos en formato de historia de usuario porque es lo que su plantilla pide, pero el contenido no refleja necesidades reales de usuario sino decisiones de implementación.

**Solución**: este es un problema de la herramienta, no de la metodología. Al revisar las specs en la puerta 1, el equipo debe verificar que las historias de usuario expresan comportamientos observables por el usuario, no instrucciones técnicas. Si la herramienta genera historias mal formuladas, el equipo debe corregirlas o ajustar los prompts de generación para producir historias genuinas.

# 24. Hacia dónde evoluciona SDD

## Las tendencias visibles

SDD es una metodología joven cuya adopción se está acelerando rápidamente. Aunque es prematuro hacer predicciones definitivas, hay tendencias claras que apuntan a su evolución.

## La spec como unidad fundamental de programación

Hay voces en la industria —entre ellas el equipo de Tessl Framework y divulgadores como Dario Amodei— que predicen que en pocos años los desarrolladores no mirarán el código directamente. La spec se convertiría en el artefacto que se edita, se versiona, se revisa y se mantiene, y el código sería un derivado completamente generado.

Esta visión —el nivel spec-as-source de la taxonomía de Böckeler— es todavía experimental. Pero la dirección es consistente: cada nueva generación de modelos de lenguaje es mejor generando código a partir de specs. A medida que la fiabilidad de la generación mejora, la necesidad de revisar el código manualmente disminuye y la spec gana peso como artefacto primario.

Para los equipos ágiles, esto significaría que la competencia de «saber programar» se desplaza definitivamente hacia la competencia de «saber especificar». No es que la programación desaparezca —alguien necesitará entender el código cuando haya problemas— pero dejará de ser la actividad cotidiana de la mayoría de los constructores de software.

## Specs ejecutables y suites de conformidad

Una evolución natural de SDD es que las specs dejen de ser solo documentos de texto y se conviertan en artefactos ejecutables. Ya hay propuestas en esta dirección: suites de conformidad independientes del lenguaje (tests en YAML que cualquier implementación debe pasar), specs que generan automáticamente tests de aceptación, y validación continua del código contra la spec.

Cuando una spec es ejecutable, la puerta de aprobación entre implementación y entrega se automatiza parcialmente: si el código pasa la suite de conformidad derivada de la spec, cumple la spec. La revisión humana se reserva para aspectos que la conformidad automatizada no captura: legibilidad, mantenibilidad, decisiones de diseño.

## *LLM-as-Judge*: validación semántica

Para criterios difíciles de testar automáticamente —estilo de código, legibilidad, adherencia a patrones de arquitectura, calidad de la documentación— se está

consolidando el patrón de usar un segundo agente como evaluador (LLM-as-Judge). Este agente revisor recibe la spec y el código generado, y evalúa si el código cumple las directrices de calidad del proyecto.

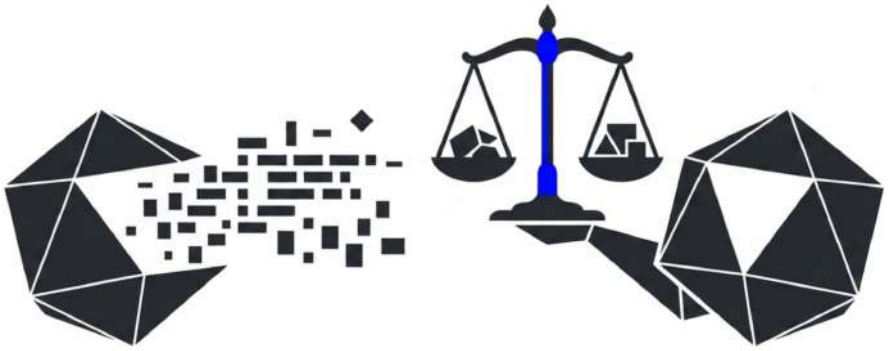


***Imagen 23.*** *LLM-as-Judge*

Este patrón añade una capa de evaluación semántica que los tests sintácticos no proporcionan. No es infalible —un modelo evaluando a otro modelo tiene limitaciones evidentes— pero complementa las verificaciones automatizadas con un juicio aproximado de calidad.

## Implicaciones para la formación de profesionales ágiles

Si las tendencias actuales se confirman, la formación de profesionales ágiles necesitará incorporar competencias que hoy son incipientes:

- **Diseño de specs**: la capacidad de escribir specs claras, precisas y verificables será una competencia central, no un complemento.
- **Orquestación de agentes**: entender cómo delegar trabajo a agentes, cómo evaluar sus resultados y cómo gestionar la ejecución paralela.
- **Criterio de calibración**: saber cuándo aplicar SDD y con qué intensidad, cuándo un prompt directo es suficiente y cuándo una spec completa es necesaria.
- **Revisión de código generado**: la capacidad de evaluar código que no has escrito tú, contra una spec que define lo que debería ser.

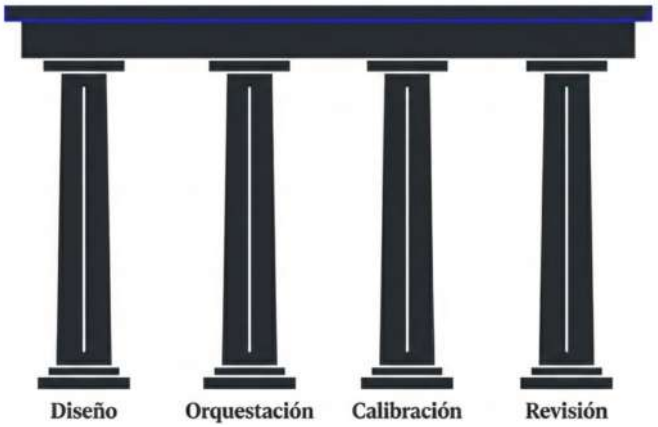



***Imagen 24.*** *Competencias de los nuevos constructores*

Estas competencias no sustituyen a las actuales —la comprensión del dominio, la facilitación de equipos, el pensamiento de producto siguen siendo fundamentales— pero las complementan con habilidades específicas del trabajo con agentes.

# Glosario

Este glosario recoge los términos centrales de SDD y los relaciona con los conceptos correspondientes de scrum y de scrum en equipos con IA. Sirve como referencia para profesionales que se mueven entre marcos.

## Términos de SDD

**Boundaries (Always / Ask First / Never)**. Sistema de tres niveles que define qué puede hacer un agente sin permiso, qué requiere aprobación y qué está prohibido.

**Clarity Gate**. Prueba de calidad de una spec: ¿puede otro agente o sesión generar código equivalente a partir solo del documento?

**Constitución del proyecto**. Documento (CLAUDE.md, agents.md o equivalente) que captura las convenciones, patrones, stack tecnológico y reglas generales del proyecto. Se aplica a cualquier tarea como contexto base.

**EARS (Easy Approach to Requirements Syntax)**. Notación basada en cinco patrones (mientras, cuando, si, donde, ubicuo) para escribir requisitos verificables.

**Impact Report**. Análisis del codebase existente que identifica ficheros afectados, patrones a respetar y posibles efectos colaterales antes de escribir requisitos.

**LLM-as-Judge**. Patrón en el que un agente revisor evalúa el output de otro agente contra criterios de calidad.

**Maldición de las instrucciones**. Fenómeno por el que la fiabilidad de un LLM cumpliendo instrucciones individuales decrece a medida que aumenta el número total de instrucciones.

**Oleada (wave)**. Conjunto de tareas independientes que pueden ejecutarse en paralelo. Las oleadas se ejecutan en secuencia.

**Puerta de aprobación (approval gate)**. Punto de revisión humana entre fases del flujo SDD donde el equipo decide si avanzar a la siguiente fase.

**Spec** (femenino: «la spec», «las specs»). Documento que captura los requisitos, el diseño o las tareas de una funcionalidad. Es el artefacto primario del desarrollo en SDD: el código se deriva de la spec.

**Spec delta**. Formato de spec que describe solo los cambios (ADDED, MODIFIED, REMOVED) en lugar del sistema completo. Especialmente útil en codebase heredado.

**Spec-Driven Development (SDD)**. Metodología que organiza el desarrollo de software asistido por IA en torno a documentos de especificación que se producen y se aprueban antes de generar código.

**Spec-first / Spec-anchored / Spec-as-source**. Tres niveles crecientes de adopción de SDD según el ciclo de vida que se da a la spec (descartar, mantener, ser la fuente única).

**Spec viva (living spec)**. Spec que se mantiene actualizada en sincronía con el código a medida que el proyecto evoluciona.

**Tarea atómica**. Unidad de implementación lo suficientemente pequeña para que un agente la ejecute con contexto limpio. Típicamente afecta a uno-tres ficheros.

**Vibe coding**. Modo de programación con IA en el que el desarrollador describe la intención conversacionalmente y el agente genera código sin spec previa. Útil para prototipos; problemático a escala.

## Equivalencias entre marcos

| Concepto SDD | Scrum tradicional | Scrum en equipos con IA |
|---|---|---|
| Spec | Historia de usuario refinada | Definition of Ready for AI |
| Puerta 1 (requisitos) | Refinamiento del backlog | Validación de discovery |
| Puerta 2 (diseño) | Sin equivalente directo | Validación arquitectónica |
| Puerta 3 (tareas) | Sprint planning | Planificación de doble carril |
| Implementación | Desarrollo del sprint | Carril de delivery |
| Constitución del proyecto | Guías de equipo, README | Marco operativo del equipo |
| Boundaries | Definition of Done implícita | Boundaries explícitos para agentes |
| Spec viva | Documentación viva (rara) | Documentación operativa |
| Tareas atómicas | Sub-tareas | Unidades de delegación a agentes |

## Roles y su equivalencia

| Rol en SDD (esta guía) | Scrum tradicional | Scrum en equipos con IA |
|---|---|---|
| Diseñador de specs | Product owner + Tech lead | Product architect |
| Orquestador de agentes | Developer | Product builder |
| Facilitador de puertas | Scrum master | Agile enabler |

# Bibliografía

## Sobre vibe coding y adopción de IA en desarrollo


- GitHub & Wakefield Research (2023), "*Survey reveals AI's impact on the developer experience*", *Github Blog*.
- Karpathy, A. (2025), Tuit en X (Twitter) introduciendo el término vibe coding: «There's a new kind of coding I call "vibe coding", where you fully give in to the vibes, embrace exponentials, and forget that the code even exists». Recogido en *Wikipedia* y múltiples medios.
- Tan, G. (2025), "Y Combinator startups are fastest growing, most profitable in fund history because of AI", *CNBC*.


## Sobre productividad y calidad del código generado por IA


- Abrams, Lawrence (2026), "Curl ending bug bounty program after flood of AI slop reports", *BleepingComputer*.
- GitClear (2025), "AI Copilot Code Quality 2025: 4x Growth in Code Clones", GitClear.
- METR (2025), "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity", *Metr*.
- Veracode (2025), "2025 GenAI Code Security Report: Assessing the Security of Using LLMs for Coding", *Veracode*.


## Sobre la metodología SDD y su análisis


- Böckeler, B. (2025), "Understanding Spec-Driven Development: Kiro, spec-kit, and Tessl", *MartinFlower*.
- Thoughtworks (2025), "Spec-Driven Development", *Thoughtworks*.
- Zaninotto, François (2025), "Spec-Driven Development: The Waterfall Strikes Back", *Marmelab*.


## Sobre escribir buenas specs para agentes


- Nigh, Matt (2025), "How to write a great agents.md: Lessons from over 2,500 repositories", *GithubBlog*.
- Osmani, A. (2026), "How to write a good spec for AI agents", *AddyOsmaniBlog*.

## Sobre la maldición de las instrucciones

- Harada, K. et al. (2024), "Curse of Instructions: Large Language Models Cannot Follow Multiple Instructions at Once", OpenReview.net.

## Sobre EARS

- Mavin, A. et al. (2009), "Easy Approach to Requirements Syntax (EARS)", *Proceedings of the 17th IEEE International Requirements Engineering Conference*.

## Sobre big ball of mud

- Foote, B. y Yoder, J. (1997, revisado en 1999), "Big Ball of Mud", *Fourth Conference on Pattern Languages of Programs (PLoP '97)*.

## Sobre Simon Willison y conformance suites

- Willison, S. (entradas continuas en simonwillison.net/tags/conformance-suites). Defensa habitual del uso de suites de conformidad como contratos independientes del lenguaje.

## Sobre las herramientas SDD mencionadas

- Bmad code (2026), BMAD Method, *Github*.
- Augment Code (2026), The Software Agent Company, Augment Code.
- GitHub Spec Kit (2025), "Anuncio: Spec-driven development with AI: Get started with a new open source toolkit", *GitHubBlog*.
- Fission AI (2026), OpenSpec. Framework open source con foco brownfield.
- Gotalab (2026), cc-sdd Skills de SDD para múltiples agentes, *Github*.
- Kiro (AWS) (2025), kiro.dev. IDE basado en VS Code.
- Tessl Framework (2026), tessl.io. En beta privada (al cierre de la edición de esta guía).

## Sobre la guía complementaria

- Scrum Manager (2026), *Scrum en equipos con IA*. Marco organizativo y de roles para equipos ágiles con agentes.

*Esta guía es parte del corpus de Scrum Manager 2026 sobre agilidad y desarrollo asistido por IA. Las correcciones, sugerencias y casos de uso pueden enviarse a través de los canales habituales de la comunidad.*