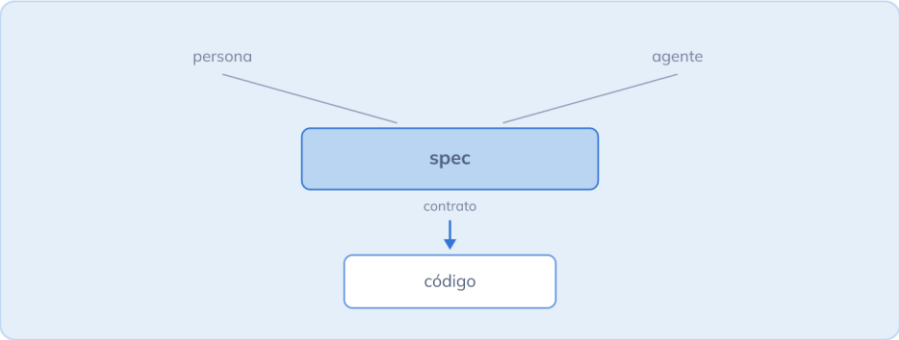
Scrum Manager
Uncovering better ways

Guía

Spec Driven Development

en equipos ágiles

Desarrollo de los temas del State of the art



Actualizado a septiembre de 2026

Scrum Manager® · Skill Arena

Sobre este documento

Este documento desarrolla en profundidad los temas del mapa de referencia (State of the art) para dirigir el desarrollo de *software* con IA mediante especificaciones (*Spec Driven Development*) en equipos ágiles, que está disponible en: Skill Arena.

Úsalo como material de consulta para mantener y contrastar tu práctica con el conocimiento profesional actual.

Se complementa con la plataforma de entrenamiento y evaluación en Skill Arena. En el área Spec Driven Development en equipos ágiles puedes realizar pruebas de entrenamiento para contrastar y mejorar tu nivel de conocimiento y si lo deseas también puedes obtener un diploma que acredita curricularmente la solvencia y vanguardia profesional en esta área.

Estado del conocimiento

El conocimiento sobre Spec Driven Development evoluciona de forma extremadamente rápida: método, herramientas y prácticas se renuevan en cuestión de meses. Los fundamentos ya están asentados y el método ha ganado respaldo académico e institucional durante 2026, pero el instrumental sigue en plena formación y la evidencia sobre cuánto acelera realmente el desarrollo con IA está en revisión. En los distintos apartados del documento, las etiquetas (ESTABLECIDO, EN CONSOLIDACIÓN, EMERGENTE) ayudan a identificar la madurez de cada concepto:

ESTABLECIDO consenso asentado; conocimiento que se da por necesario.
EN CONSOLIDACIÓN gana adopción con rapidez; aún no universal pero ya relevante.
EMERGENTE frontera reciente; alta relevancia y alta volatilidad.

Índice

Capítulos del desarrollo, en el orden del State of the art.

Bloque A · El paradigma: por qué SDD

1. Del *vibe coding* al desarrollo guiado por especificaciones
2. La spec como artefacto primario del desarrollo
3. SDD y agilidad: no es el regreso de *waterfall*
4. El principio de proporcionalidad

Bloque B · El método: fases, envolturas y puertas

5. El flujo: el núcleo de cuatro fases y sus dos envolturas
6. La constitución del proyecto
7. Las puertas de aprobación (*approval gates*)
8. Tareas atómicas, oleadas y commits atómicos
9. Especificar sobre *codebase* existente (brownfield)

Bloque C · Escribir buenas specs

10. La spec inteligente, no extensa
11. La notación EARS para criterios de aceptación
12. El sistema de boundaries: *Always* / *Ask First* / *Never*
13. La maldición de las instrucciones (*instruction overload*)
14. Specs vivas, deriva y la *Clarity Gate*

Bloque D · SDD en el equipo y su ecosistema

15. Encaje en roles y ceremonias ágiles
16. Trazabilidad y supervisión humana como obligación
17. Antipatrones: teatro de especificación y documentación zombi
18. El ecosistema de herramientas

BLOQUE A · EL PARADIGMA: POR QUÉ SDD

Capítulo 1. Del *vibe coding* al desarrollo guiado por especificaciones · ESTABLECIDO

El vibe coding es productivo para prototipos y tiene un techo; SDD responde a ese techo escribiendo el contrato antes que el código.

Introducción

En febrero de 2025, Andrej Karpathy popularizó la expresión vibe coding para describir una forma de programar con IA que muchos ya practicaban sin nombre. Se describe en lenguaje natural lo que se quiere, el modelo genera el código, se prueba y, si falla, se le devuelve el error hasta que sale del paso. No hay diseño previo ni especificación. El *prompt* es la intención, el código es la respuesta y la conversación es el método.

En este primer capítulo veremos por qué ese enfoque funciona hasta cierto punto, qué evidencia hay sobre sus límites y con qué cuidado conviene manejarla, y por qué el desarrollo guiado por especificaciones surge como respuesta. Es la base conceptual del resto de la guía, y afecta por igual a quien programa y a quien no.

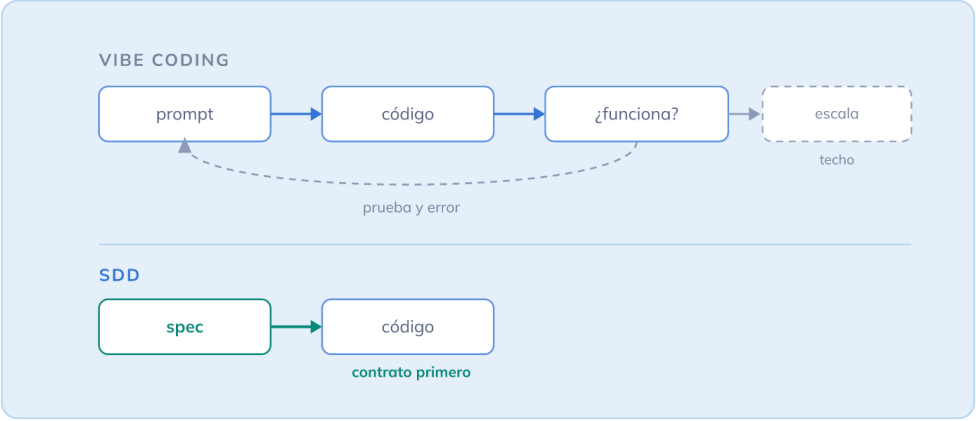


Figura 1. *SDD invierte el orden: el vibe coding conversa con el agente y SDD le entrega un contrato antes de generar código.*

1.1 Dónde funciona el vibe coding

Conviene empezar reconociendo su mérito. Para prototipos, exploración de ideas, herramientas internas de un solo uso y todo aquello donde importa más llegar pronto que llegar bien, el vibe coding es extraordinariamente productivo. Permite probar una hipótesis en una tarde, enseñar algo tangible a un cliente antes de decidir si merece inversión y automatizar tareas pequeñas que nunca habrían justificado un desarrollo formal.

El problema está en su techo, que aparece antes de lo que la mayoría espera. A partir de cierta escala, la conversación deja de sostener el diseño. Aparecen requisitos implícitos que nadie escribió y que el agente resuelve a su manera, contaminación del contexto por la acumulación de intentos fallidos y deriva arquitectónica, porque cada parche se decide en el momento y sin memoria de los anteriores.

1.2 La evidencia, con cuidado

El dato más citado del área sigue siendo el ensayo controlado que METR publicó en julio de 2025 con desarrolladores experimentados de proyectos de código abierto. Con IA tardaban un 19 % más en completar sus tareas, mientras creían ir un 20 % más rápido. La brecha entre la velocidad percibida y la medida se convirtió en el argumento central de quienes defendían poner método al trabajo con agentes.

Ese dato sigue publicado y se puede citar, pero ya no como medición contundente ni vigente. En febrero de 2026 METR anunció que cambiaba el diseño de su experimento. Su estudio de seguimiento, con 57 desarrolladores, 143 repositorios y más de 800 tareas, dio un −18 % para los diez participantes del estudio original y un −4 % para los cuarenta y siete nuevos, con intervalos de confianza que en los dos casos cruzan el cero. La causa que METR declara son sesgos de selección graves, y su conclusión es que su estimación es una cota inferior y que hoy es probable que los desarrolladores estén más acelerados por la IA de lo que midieron en 2025.

El argumento no depende de una cifra. La brecha entre lo que se percibe y lo que se mide sigue documentada, pero ha pasado de «está medido» a «está en revisión». Un profesional que cite el 19 % sin fecharlo ni matizarlo está afirmando algo que su propia autora ya no sostiene.

Lo que sí se sostiene, y con evidencia más reciente, es el diagnóstico del coste. El informe de DORA sobre el retorno de la inversión en desarrollo asistido por IA describe dos fenómenos con nombre. La curva J es la caída temporal de productividad que precede a la captura de valor, el coste de matrícula de la transformación. El impuesto de verificación (*verification tax*) es el esfuerzo añadido de comprobar que el código generado es fiable, seguro y coherente con la arquitectura. Conviene decirlo con honestidad: DORA respalda el problema, y no nombra las especificaciones entre los factores de retorno. La solución que enseña esta guía se defiende por otras vías.

1.3 El coste real es cognitivo

Cuando alguien escribe código, su memoria de trabajo retiene el contexto de lo que acaba de escribir. Cuando revisa código generado por otro, la carga de comprensión es distinta y se acumula con cada iteración. La paradoja del vibe coding es que una herramienta diseñada para ahorrar esfuerzo de construcción termina generando esfuerzo de verificación, que es más agotador y peor repartido, porque llega todo al final y sin el mapa mental que da haber construido.

El impuesto de verificación no desaparece con ninguna metodología. Lo que cambia es dónde se paga. Pagarlo en la especificación, antes de generar una línea, cuesta menos que pagarlo revisando código cuyo diseño nadie decidió. Esa es la intuición que da lugar a SDD.

1.4 La respuesta: el desarrollo guiado por especificaciones

El desarrollo guiado por especificaciones (*spec-driven development*, SDD) invierte el orden. En lugar de saltar al código, se produce una especificación que define qué se construye, cómo y en qué pasos, y solo después de revisarla y aprobarla se implementa. La definición de Birgitta Böckeler es escueta y basta: SDD significa escribir una *spec* antes de escribir código con IA. Donde el vibe coding trata al agente como un interlocutor con el que se conversa, SDD lo trata como un ejecutor que recibe un contrato claro.

SDD es una metodología y no una herramienta ni un marco de trabajo. Es una forma de organizar el trabajo que puede implementarse con instrumentos distintos, como veremos en el capítulo 18. Y conviene despejar desde ahora tres malentendidos frecuentes. SDD no consiste en documentar después de construir, porque el orden es justo lo que importa. No exige pliegos de doscientas páginas, porque una buena spec es inteligente y no extensa, como desarrolla el capítulo 10. Y no supone abandonar la agilidad, sino aplicarla de forma coherente cuando parte del equipo son agentes, que es el asunto del capítulo 3.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Explicar qué es el vibe coding y en qué contextos es la herramienta adecuada.
- Reconocer las tres señales del techo del vibe coding a escala: requisitos implícitos, contaminación de contexto y deriva arquitectónica.
- Citar la evidencia de METR con su fecha y su matización posterior, y distinguir lo que está medido de lo que está en revisión.
- Explicar la curva J y el impuesto de verificación del informe de DORA, y qué respaldan y qué no.
- Definir SDD como metodología y enunciar su premisa: la spec precede al código.
- Distinguir SDD de la documentación posterior, de los pliegos exhaustivos y de un supuesto abandono de la agilidad.

Errores y antipatrones frecuentes

Tratar el vibe coding como un enemigo que erradicar. Es la herramienta correcta para prototipos y exploración. SDD lo complementa donde la conversación no escala y no lo sustituye en todo.

Citar el 19 % de METR como dato vigente. El estudio se rediseñó en 2026 y su propia autora considera probable que la aceleración real sea mayor. El argumento se sostiene en la brecha percepción-medición, no en la cifra.

Atribuir a DORA un aval de SDD. El informe describe el impuesto de verificación y la curva J, pero no nombra las especificaciones entre los factores de retorno. Respalda el problema, no la solución.

Creer que SDD es una herramienta que se instala. Es una forma de organizar el trabajo. Las herramientas la soportan, no la definen.

Para profundizar

· METR — We are changing our developer productivity experiment design (2026)

· METR — Measuring the impact of early-2025 AI on experienced developer productivity

· DORA — ROI of AI-assisted Software Development

· Böckeler — Understanding Spec-Driven Development

BLOQUE A · EL PARADIGMA: POR QUÉ SDD

Capítulo 2. La spec como artefacto primario del desarrollo · ESTABLECIDO

El centro del trabajo se desplaza de escribir código a especificar con claridad qué se construye, por qué y bajo qué restricciones.

Introducción

Addy Osmani lo resume en una frase que conviene tener a mano: la IA no es el cuello de botella; tu spec lo es. Durante décadas, la competencia central del profesional del software ha sido escribir código. SDD propone que esa competencia se desplaza sin desaparecer, porque entender código sigue siendo necesario para revisar y decidir, pero deja de ser el acto central. Lo que hace productivo a un equipo con agentes es su capacidad de especificar con claridad qué construir, por qué, bajo qué restricciones y con qué criterios de éxito.

En este capítulo veremos qué significa tratar la spec como artefacto primario, por qué funciona como un contrato y no como una orden, y qué cambia para cada rol del equipo, incluidos los que no programan.

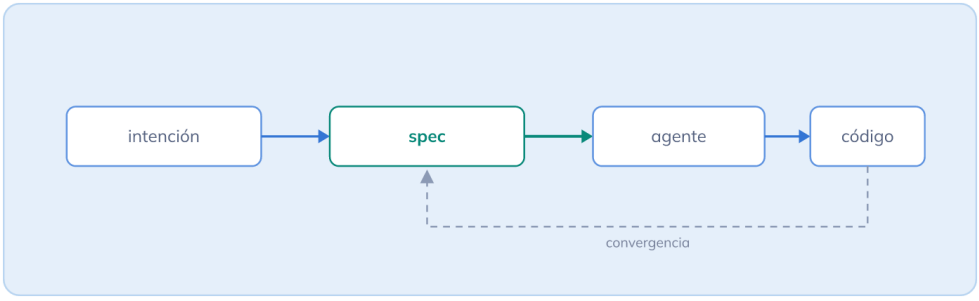


Figura 2. *La spec como contrato: la intención de las personas se convierte en un contrato que el agente ejecuta y contra el que se verifica el código.*

2.1 El código como consecuencia

La analogía más útil es la de la arquitectura. Un arquitecto no coloca ladrillos, diseña planos que otros ejecutan, y la calidad del edificio depende de la calidad de los planos.

La spec es el plano y el código es el edificio. Cuando algo falla, la primera pregunta deja de ser qué defecto tiene el código y pasa a ser qué no especificamos bien.

La razón es que el agente es literal. Cada decisión que la spec no toma es una decisión que el agente toma por su cuenta, y cada una de ellas es un punto de divergencia potencial entre lo que se quería y lo que se obtiene. Una historia de usuario que una persona interpretaría con sentido común, un agente la implementará al pie de la letra o inventará lo que le falte. Especificar deja de ser un trámite previo y se convierte en la actividad que determina el resultado.

2.2 Contrato, no documento de mando

Lo que distingue SDD de la simple documentación previa es que la spec funciona como un contrato entre las partes del sistema. Las personas lo aprueban en las puertas de fase, los agentes lo ejecutan y lo que se entrega es código que cumple el contrato. La apuesta de fondo del método es que los contratos explícitos, en la granularidad correcta, permiten que el desarrollo con IA a escala de equipo avance más rápido y con menos sorpresas.

Durante 2026 esta idea ha ganado formulación académica. El trabajo de Díaz, Gayoso, Cimmino y Pérez sobre desarrollo guiado por especificaciones para la ingeniería de software agéntica describe las especificaciones como el sustrato contractual entre humanos y agentes, y sostiene que son lo que devuelve la responsabilidad y la verificabilidad que la asistencia informal de la IA erosiona. Los propios autores presentan su propuesta como un primer paso hacia un consenso entre academia e industria, todavía pendiente de validación empírica, y conviene citarla con esa misma prudencia.

Aprobado, no lanzado. Una spec que se entrega al agente sin que nadie la haya leído con criterio no es un contrato, es una orden con apariencia de método. El valor está en la aprobación deliberada, como veremos en el capítulo 7.

2.3 Qué cambia para cada rol

El desplazamiento afecta a todo el equipo y no solo a quien programa, porque especificar con precisión es una competencia compartida.

- ***Product owner* y product architect.** La exigencia de precisión sube. Los criterios de aceptación pasan de ser una guía para la conversación a ser parte de un contrato de ejecución.
- **Desarrollador y *product builder*.** La competencia se desplaza del lenguaje de programación al diseño de *specs* y a la orquestación y revisión de agentes. El perfil se parece más al de un responsable técnico que al de un programador.
- ***Agile enabler*.** Aparecen las puertas de aprobación como puntos de inspección que hay que facilitar sin que se conviertan en cuellos de botella.
- ***Stakeholders*.** Ganan visibilidad, porque una spec es legible por personas sin formación técnica en una medida que el código nunca lo fue.

Estos roles son los que describe la skill «*Scrum* en equipos con IA» de Skill Arena para el equipo que trabaja con agentes. El capítulo 15 vuelve sobre ellos con la vista puesta en las ceremonias.

2.4 Legibilidad y responsabilidad

Que la spec sea legible tiene una consecuencia que va más allá de la comodidad. Reduce la asimetría de información entre quien construye y quien paga o usa el producto. Un *stakeholder* puede leer una spec y opinar sobre prioridades o sobre conformidad antes de que se convierta en código, algo que con el código en bruto nunca estuvo a su alcance.

Y devuelve una responsabilidad que la asistencia informal difumina. Cuando el código sale de una conversación que nadie conserva, no hay a qué remitirse cuando algo va mal. Cuando sale de una spec aprobada, la cadena de decisiones está escrita y se puede revisar, discutir y corregir. El capítulo 16 muestra que esa trazabilidad tiene además un valor regulatorio cuando el producto entra en el ámbito de alto riesgo.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Explicar el desplazamiento de la competencia central del código a la especificación.
- Aplicar la analogía del plano y el edificio para razonar sobre la relación entre la spec y el código.
- Distinguir la spec como contrato, aprobado en puertas y ejecutado por agentes, de un documento de mando.
- Citar la formulación académica de la spec como sustrato contractual con la prudencia que sus autores piden.
- Anticipar cómo cambia el trabajo de cada rol del equipo, incluidos los no técnicos.

Errores y antipatrones frecuentes

Creer que el código deja de importar. Hay que revisarlo, probarlo y validarlo. Lo que cambia es la dirección causal, porque la spec produce el código y no al revés.

Especificar con la imprecisión del desarrollo tradicional. Lo que una persona completaría con sentido común, un agente literal lo implementa al pie de la letra o lo inventa.

Tratar la spec como una orden unidireccional. Funciona como un contrato aprobado con criterio, y su valor está en esa aprobación.

Reservar la especificación a los perfiles técnicos. La precisión que exige la spec es competencia de todo el equipo, y los stakeholders son quienes más ganan con su legibilidad.

Para profundizar

· Addy Osmani — How to write a good spec for AI agents

· Díaz, Gayoso, Cimmino y Pérez — SDD for Agentic Software Engineering (arXiv, 2026)

· Skill Arena — Scrum en equipos con IA

· Martin Fowler — Exploring Gen AI (serie)

BLOQUE A · EL PARADIGMA: POR QUÉ SDD

Capítulo 3. SDD y agilidad: no es el regreso de *waterfall* · EN CONSOLIDACIÓN

SDD usa fases y puertas, pero difiere de waterfall en alcance, feedback y reversibilidad, y la crítica que lo niega tiene nombres y merece leerse.

Introducción

Cualquier profesional ágil pensará, tarde o temprano, que SDD suena a waterfall. Fases secuenciales, puertas de aprobación y documentación antes del código son los rasgos que la agilidad nació para combatir. La objeción es razonable, y resolverla es lo que permite adoptar SDD sin traicionar los principios. En este capítulo la abordamos de frente, y le damos la palabra a quienes la sostienen, porque la crítica tiene nombres y sigue viva.

Figura 3. *Tres ejes que separan SDD de waterfall: el alcance es el incremento, el feedback llega en cada puerta y volver atrás cuesta lo que cuesta editar un documento.*

3.1 Qué comparte con waterfall y qué no

SDD comparte con waterfall la secuencialidad, porque los requisitos preceden al diseño y el diseño a las tareas. Difiere en tres ejes que son los que importan.

1. **Alcance.** Waterfall aplicaba la secuencia al proyecto entero, con meses por fase. SDD la aplica a cada incremento, y el ciclo completo puede medirse en horas o en días.
2. **Feedback.** En waterfall llegaba al final. En SDD llega en cada puerta de aprobación, de modo que cada ciclo es un ciclo de aprendizaje completo.

3. **Reversibilidad.** Volver atrás en waterfall era caro. En SDD las fases previas a la implementación son documentos de texto, y modificarlos resulta trivial frente a reescribir código.

Aplicar la secuencia al proyecto entero sí sería waterfall. Aplicarla a un incremento pequeño, con revisión en cada paso y con la posibilidad de rehacer un documento en minutos, es otra cosa.

3.2 Una categoría nueva de documentación

La documentación tradicional era informativa, porque transmitía conocimiento entre personas, o administrativa, porque satisfacía requisitos del proceso. El Manifiesto Ágil cuestionó con razón que se priorizara sobre el software que funciona. SDD introduce una tercera categoría, la documentación operativa, que es la spec que el agente consume directamente para generar código. No informa a una persona ni cumple un trámite, es el insumo del proceso de producción.

Por eso la objeción del Manifiesto no le aplica del mismo modo. El texto dice «software funcionando sobre documentación extensiva», y «sobre» no significa «en lugar de». Una documentación que contribuye de forma directa al software funcionando está del lado correcto de esa preferencia.

3.3 La crítica con nombre

François Zaninotto, de Marmelab, publicó en noviembre de 2025 un artículo cuyo título es ya una tesis, «Spec-Driven Development: The Waterfall Strikes Back». No se queda en el parecido. Enumera fallos concretos que ha visto en la práctica, como documentos enormes, pérdida del contexto del código, trabajo de revisión duplicado y una falsa sensación de seguridad, y sostiene que la utilidad del método queda casi limitada a los proyectos nuevos. Propone como alternativa el desarrollo en lenguaje natural (*natural language development*), que consiste en descomponer en piezas pequeñas y comprobables e iterar sobre los fallos.

Desde el mundo ágil, Yuval Yeret ha planteado la pregunta de si SDD supone un paso adelante o un paso atrás para el desarrollo de producto. Su respuesta es matizada, y el matiz es lo interesante para el lector de esta guía. Depende de dónde viva la spec y de quién la escriba, que es justo lo que desarrolla el capítulo 15.

Leer la crítica de primera mano. Los fallos que Zaninotto enumera son los antipatrones del capítulo 17 vistos desde fuera. Quien adopta SDD hace bien en conocerlos antes de que aparezcan.

3.4 Dónde lo sitúa la industria

El Technology Radar de Thoughtworks mantiene SDD en el anillo «Assess», sin haberlo promovido a «Trial». Recomienda evaluarlo cuando los requisitos son ambiguos, los equipos están distribuidos o hace falta trazabilidad, y advierte de que puede no

compensar en problemas triviales. Es la posición de una organización que lo usa y que no lo da por consolidado, y coincide con el estado de esta ficha.

El propio ecosistema ha incorporado la objeción a su discurso. OpenSpec, una de las herramientas ligeras del área, declara su filosofía en tres frases: fluido y no rígido, iterativo y no waterfall, y *brownfield* primero. Cuando una herramienta se define por oposición a una crítica, la crítica ha hecho su trabajo.

Bien entendido, SDD puede leerse como la aplicación coherente de los principios ágiles cuando parte del equipo son agentes. Hay feedback rápido en las puertas, entrega incremental con una spec por incremento, colaboración alrededor de un artefacto legible y respuesta al cambio, porque modificar una spec es más barato que reescribir código.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Articular las tres diferencias entre SDD y waterfall: alcance, feedback y reversibilidad.
- Explicar la categoría de documentación operativa y por qué la objeción del Manifiesto Ágil no le aplica igual.
- Resumir la crítica de Zaninotto y la alternativa que propone, y situar la pregunta de Yeret.
- Situar el juicio del Technology Radar de Thoughtworks sobre cuándo SDD aporta valor y cuándo no compensa.
- Argumentar SDD como aplicación coherente de los principios ágiles en equipos con agentes.

Errores y antipatrones frecuentes

Rechazar SDD por parecer waterfall. La semejanza es superficial. Las diferencias de alcance, feedback y reversibilidad son las que importan.

Aplicar la secuencia al proyecto entero. Eso sí sería waterfall. SDD aplica la secuencia a cada incremento pequeño.

Leer el Manifiesto como una prohibición de documentar. Dice «sobre», no «en lugar de». La documentación operativa contribuye de forma directa al software que funciona.

Despachar la crítica como resistencia al cambio. Zaninotto describe fallos reales que aparecen cuando el método se aplica mal. Conocerlos es la mejor prevención.

Para profundizar

· Thoughtworks Technology Radar — Spec-driven development

· Marmelab — Spec-Driven Development: The Waterfall Strikes Back

· Yuval Yeret — Is Spec-Driven Development a step forward or back for product development?

· OpenSpec

· Manifiesto Ágil

BLOQUE A · EL PARADIGMA: POR QUÉ SDD

Capítulo 4. El principio de proporcionalidad · ESTABLECIDO

Ajustar la intensidad del proceso al tamaño del problema; mal calibrado, SDD se convierte en burocracia de especificación.

Introducción

SDD no es un formulario que se rellena de forma mecánica. El principio que gobierna su aplicación sensata es el de proporcionalidad, que consiste en ajustar la intensidad del proceso al tamaño del problema. Un defecto menor no necesita una spec de cuatro fases, y una funcionalidad compleja que afecta a varias partes del sistema sí la necesita. En este capítulo veremos el riesgo de la sobreaplicación, la pregunta que permite calibrar y la evidencia reciente que refuerza el principio desde otro ángulo.

4.1 El riesgo de la sobreaplicación

La propia Böckeler documentó un caso revelador. Al pedir a una herramienta de SDD que arreglara un defecto pequeño, el documento de requisitos generado convirtió la tarea en cuatro historias de usuario con dieciséis criterios de aceptación. Lo describió como usar un mazo para romper una nuez. SDD mal calibrado se convierte en una burocracia de especificación que ralentiza en lugar de acelerar, y la solución consiste en ajustar la intensidad, no en rechazar el método.

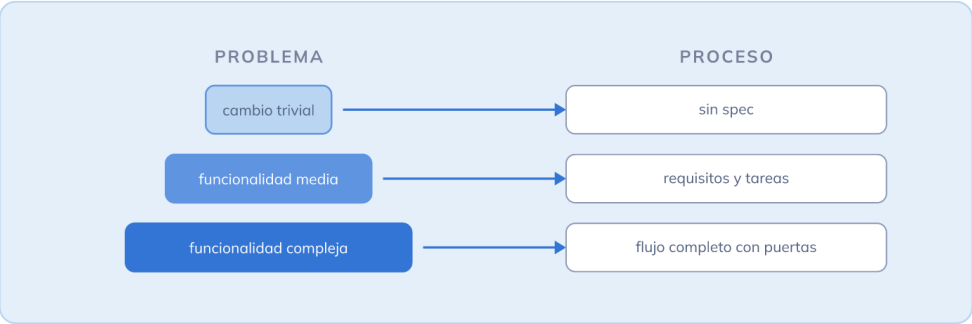


Figura 4. *El principio de proporcionalidad: la intensidad del proceso se ajusta al tamaño del problema.*

La sobreaplicación tiene además un coste oculto que no aparece en el reloj. Cuando el equipo aprende que cualquier cambio dispara el flujo completo, empieza a evitar el flujo,

y con él las puertas que sí protegían. La rigidez excesiva no produce rigor, produce atajos.

4.2 La pregunta de calibración

La heurística práctica es una sola pregunta. Qué ocurre si el agente toma la decisión equivocada en este punto. Si la respuesta es «lo corrijo en dos minutos», ese punto no necesita especificarse. Si la respuesta es «pierdo horas o introduzco un defecto sutil», sí.

SDD aporta valor neto cuando el cambio es sustancial, hay ambigüedad, hay riesgo, interviene más de una persona o agente, o el código debe mantenerse en el tiempo. Cuando ninguna de esas condiciones se cumple, el vibe coding sigue siendo la herramienta adecuada. Un guion de un solo uso, un prototipo para una demostración o un cambio de una línea no merecen el flujo completo.

4.3 El gradiente que muestra la evidencia

La evidencia acumulada durante 2026 refuerza el principio desde otro ángulo. El informe de DORA describe que el retorno de la IA es alto en trabajo nuevo y sencillo y mucho menor en código heredado y complejo, de modo que la intensidad del proceso no puede ser la misma en los dos extremos. Donde la IA acierta casi sola, especificar mucho es un coste sin retorno. Donde la IA tropieza con la historia del sistema, la spec es lo que evita que el tropiezo llegue a producción.

El principio, por tanto, no solo protege del exceso. También indica dónde invertir. El capítulo 9 desarrolla el caso del código existente, que es donde la proporcionalidad se inclina hacia especificar más.

4.4 Saber cuándo no usar SDD

Saber cuándo no aplicar SDD forma parte de dominarlo. La sabiduría ágil es contextual, y no hay práctica que sea siempre buena o siempre mala. El principio de proporcionalidad reaparece en la calibración de la spec, en el capítulo 10, y en los antipatrones de equipo, en el capítulo 17, porque la mayoría de los fallos del método son fallos de calibración.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Enunciar el principio de proporcionalidad y aplicarlo para decidir la intensidad de SDD ante un cambio dado.
- Usar la pregunta de calibración, el coste de una decisión equivocada, para decidir qué especificar.
- Identificar las condiciones en las que SDD aporta valor neto frente a las que no.
- Explicar el gradiente entre trabajo nuevo y código heredado que describe la evidencia, y qué implica para la intensidad del proceso.
- Reconocer la sobreaplicación como un modo de fallo y no como rigor.

Errores y antipatrones frecuentes

Aplicar SDD con la misma intensidad a todo. El defecto trivial y la funcionalidad compleja no merecen el mismo proceso. Igualarlos genera burocracia y, después, atajos.

Confundir exhaustividad con calidad. Más fases y más criterios no hacen mejor SDD. A menudo son el mazo para la nuez.

Olvidar que el vibe coding sigue teniendo su sitio. Para guiones, prototipos y cambios triviales, especificar es un coste sin retorno.

Para profundizar

· Böckeler — Understanding Spec-Driven Development

· DORA — ROI of AI-assisted Software Development

· Scrum Manager — Scrum en la era de la IA

BLOQUE B · EL MÉTODO: FASES, ENVOLTURAS Y PUERTAS

Capítulo 5. El flujo: el núcleo de cuatro fases y sus dos envolturas

· ESTABLECIDO

Requisitos, diseño, tareas e implementación siguen siendo el núcleo; la constitución lo precede y la convergencia lo cierra.

Introducción

Con independencia de la herramienta, el método tiene una columna vertebral estable, una secuencia de cuatro fases con un documento por fase. Kiro las llama requisitos, diseño y tareas; Spec Kit, especificar, planificar y descomponer en tareas. La estructura de fondo es la misma porque refleja la secuencia lógica mínima para pasar de una intención a software que funciona cuando el constructor es un agente.

Lo que ha cambiado desde la primera edición de esta guía es lo que rodea a ese núcleo. Las implementaciones de referencia han asentado dos envolturas, una antes de los requisitos y otra después de la implementación, y ya no es exacto decir que todas convergen en cuatro fases. En este capítulo veremos el núcleo, las envolturas y el principio que gobierna el conjunto.

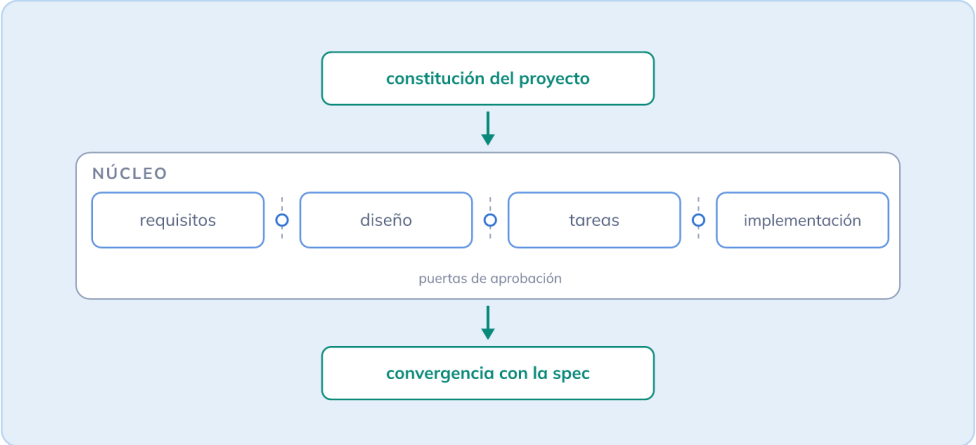


Figura 5. *El núcleo de cuatro fases y sus dos envolturas: la constitución precede a los requisitos, la convergencia cierra la implementación y la revisión se concentra en las puertas.*

5.1 Las cuatro fases y sus entregables

1. **Requisitos.** Qué se construye, desde la perspectiva del usuario y del negocio. No es técnico. Describe el comportamiento esperado, las condiciones y cómo se verificará.
2. **Diseño.** Cómo se construye. Aquí sí hay decisiones técnicas, como patrones, componentes afectados y estructura de datos, coherentes con el código que ya existe.
3. **Tareas.** El diseño se descompone en unidades atómicas de implementación, cada una con su prompt estructurado, como desarrolla el capítulo 8.
4. **Implementación.** Los agentes ejecutan las tareas y producen código, pruebas y *commits*. Es la fase en la que la persona interviene menos, si las anteriores se hicieron bien.

Cada fase produce un documento, y cada documento se revisa antes de pasar a la siguiente. Esa revisión es lo que el capítulo 7 llama puerta de aprobación.

5.2 Las dos envolturas

GitHub Spec Kit, la implementación más adoptada, publicó su versión 1.0 en agosto de 2026 y documenta un proceso de seis pasos. El primero establece los principios del proyecto en una constitución. Después vienen los cuatro del núcleo. El sexto comprueba que la implementación converge con lo especificado. A ellos se suman comandos de calidad para clarificar requisitos, analizar la coherencia entre documentos y generar listas de comprobación.

La primera envoltura, la constitución, recoge lo que se decide una vez para todo el proyecto, y tiene entidad suficiente para merecer el capítulo 6. La segunda, la convergencia, es la revisión final que compara el resultado con la spec, y responde a la pregunta que ningún documento intermedio podía responder, la de si lo construido es lo que se acordó.

Kiro llega a lo mismo por otro camino. Ofrece flujos de requisitos primero o de diseño primero, según se quiera partir del comportamiento esperado o de una idea

técnica ya madura, y un análisis profundo opcional de los requisitos antes de pasar al diseño. La convergencia de las dos herramientas de referencia en la misma forma, un núcleo envuelto, es lo que permite describir el método sin atarlo a ninguna de ellas.

5.3 El principio rector

El principio que gobierna el flujo se resume en revisar en las puertas de fase, no durante la implementación. En el trabajo conversacional, quien programa aprueba microdecisiones una a una, lo que produce fatiga de aprobación. SDD concentra la revisión humana donde la información vale más y el coste de corrección es menor, entre fases. Una vez aprobadas las tareas, la implementación se ejecuta con intervención mínima, porque el contrato ya está claro.

No es todo o nada. Las cuatro fases y las dos envolturas representan el flujo completo, y no todos los problemas las necesitan con la misma profundidad. Una funcionalidad mediana puede llevar requisitos y tareas con un diseño ligero. Calibrar la intensidad es el principio de proporcionalidad del capítulo 4 aplicado al flujo.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Enumerar las cuatro fases del núcleo y el entregable de cada una.
- Explicar las dos envolturas, la constitución y la convergencia, y por qué obligan a reformular la descripción del flujo.
- Describir cómo Spec Kit y Kiro llegan a la misma forma por caminos distintos.
- Aplicar el principio rector, revisar en las puertas y no durante la implementación, y razonar por qué reduce la fatiga de aprobación.
- Calibrar la profundidad de cada fase según el tamaño del problema.

Errores y antipatrones frecuentes

Mezclar el qué y el cómo. Las decisiones técnicas en la fase de requisitos son uno de los caminos más rápidos a la sobreespecificación.

Revisar durante la implementación en vez de en las puertas. Reintroduce la fatiga de aprobación que SDD busca eliminar.

Dar por terminado el incremento al acabar la implementación. Sin el paso de convergencia nadie ha comprobado que lo construido coincide con lo especificado, y la deriva empieza ahí.

Aplicar las fases con igual profundidad siempre. El flujo se calibra. No es un formulario obligatorio.

Para profundizar

· GitHub Spec Kit

· Kiro — especificaciones

· Kiro — buenas prácticas de specs

· Böckeler — Understanding Spec-Driven Development

BLOQUE B · EL MÉTODO: FASES, ENVOLTURAS Y PUERTAS

Capítulo 6. La constitución del proyecto · EN CONSOLIDACIÓN

Lo que se decide una vez para todo el proyecto, frente a lo que se decide en cada incremento, y el artefacto que lo conserva entre sesiones del agente.

Introducción

En la primera edición de esta guía, la constitución era media frase dentro del capítulo de los *boundaries*. Hoy es el primer paso del método en la implementación de referencia, tiene contenido documentado y se enseña como primera competencia en la formación del área. En este capítulo veremos qué es, qué contiene, dónde vive y por qué preserva el criterio entre sesiones distintas de un agente que no recuerda nada de una a otra.

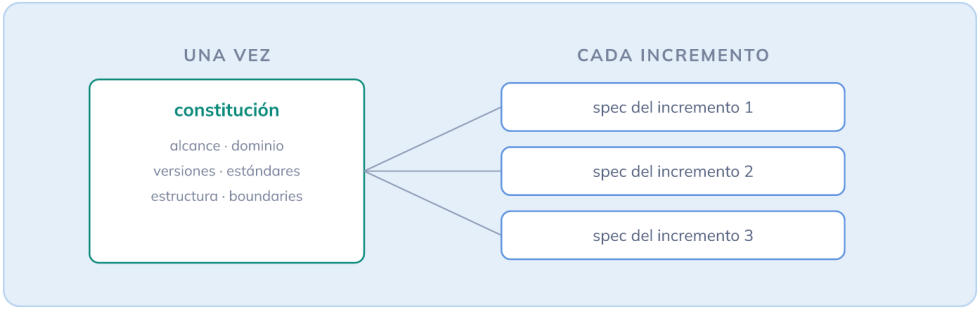


Figura 6. *Una vez frente a cada incremento: la constitución fija lo que vale para todo el proyecto y las specs deciden lo propio de cada incremento.*

6.1 Qué es y qué problema resuelve

La constitución recoge lo que se decide una vez para todo el proyecto, frente a lo que se decide en cada incremento. Es el reglamento fundacional que alinea el ciclo de vida del desarrollo y persiste por encima de las specs individuales. Resuelve un problema distinto del de la spec, que dice qué se construye, y del de los boundaries, que dicen qué puede hacer el agente. La constitución dice en qué proyecto estamos y con qué reglas se trabaja en él.

El problema que resuelve es la falta de memoria del agente. Cada sesión empieza de cero, y sin un documento que fije el contexto, las decisiones estructurales se vuelven a tomar en cada conversación, con resultados distintos. La constitución es lo que da continuidad al criterio y reduce lo que la formación del área llama deuda cognitiva, el esfuerzo de volver a explicar lo mismo cada vez.

6.2 Qué contiene

Los equipos de Thoughtworks que trabajan con SDD sobre bases de código existentes han documentado qué contiene una constitución que funciona de verdad.

- **Alcance del proyecto.** Qué es y qué no es el sistema, para que el agente no lo expanda por su cuenta.
- **Contexto de dominio.** El vocabulario del negocio y las reglas que un recién llegado necesitaría conocer.
- **Versiones tecnológicas.** Lenguajes, marcos y bibliotecas con sus versiones, para que no se introduzcan otras.
- **Estándares de código.** Convenciones de nomenclatura, de pruebas y de estilo.
- **Estructura del repositorio.** La arquitectura elegida, por ejemplo hexagonal o por capas, y dónde va cada cosa.

A esa lista se añaden los boundaries de proyecto, las reglas generales sobre lo que el agente hace siempre, consulta antes o no hace nunca, que el capítulo 12 desarrolla. Y conviene que sea corta. Una constitución que intenta contenerlo todo cae en la maldición de las instrucciones del capítulo 13.

6.3 Dónde vive y cómo se mantiene

En Spec Kit la constitución se crea con el primer comando del flujo y vive en el repositorio, versionada con el código. En el resto del ecosistema el soporte habitual es un fichero de instrucciones para agentes en la raíz del proyecto, con AGENTS.md como el formato más extendido, como veremos en el capítulo 12. Lo importante no es el nombre del fichero, sino que esté donde el agente lo lea al empezar y donde el equipo lo revise al cambiar una decisión estructural.

Se mantiene como cualquier decisión de arquitectura. Cambia poco, cambia con deliberación y cada cambio se revisa, porque afecta a todo lo que se construya después. Un incremento que necesite saltarse una regla de la constitución es una señal de que la regla, o el incremento, merecen una conversación.

Una constitución no es una spec larga. Lo que vale para un solo incremento va en su spec. Lo que vale para todos va en la constitución. Mezclarlos engorda la constitución y desactualiza la spec.

6.4 Por qué se enseña primero

El curso de DeepLearning.AI sobre desarrollo guiado por especificaciones con agentes de codificación enseña a escribir la constitución como primera competencia, antes que las specs de funcionalidad, con tres razones. Preserva el contexto entre sesiones del agente, mejora la fidelidad a la intención y reduce la deuda cognitiva. Es también el orden natural del trabajo. Cuando la constitución existe, cada spec puede ser más corta, porque no repite lo que ya está decidido.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Distinguir lo que se decide una vez para todo el proyecto de lo que se decide en cada incremento.
- Enumerar el contenido de una constitución útil y justificar cada elemento.
- Situar la constitución en el flujo del método y en el repositorio del proyecto.
- Mantener la constitución como una decisión de arquitectura, con cambios deliberados y revisados.
- Explicar por qué escribir la constitución antes que las specs acorta estas y preserva el criterio entre sesiones.

Errores y antipatrones frecuentes

Meter en la constitución lo que es propio de un incremento. La constitución engorda, la spec se desactualiza y el agente recibe reglas que no aplican a su tarea.

No tener constitución y confiar en la memoria de la conversación. Cada sesión vuelve a decidir la arquitectura, y la deriva estructural aparece en semanas.

Escribirla una vez y no volver a mirarla. Una constitución que contradice las decisiones vigentes del equipo es peor que ninguna, porque el agente le hace caso.

Para profundizar

· Thoughtworks Technology Radar — GitHub Spec Kit

· GitHub Spec Kit

· DeepLearning.AI — Spec-Driven Development with Coding Agents

BLOQUE B · EL MÉTODO: FASES, ENVOLTURAS Y PUERTAS

Capítulo 7. Las puertas de aprobación (*approval gates*) · ESTABLECIDO

Los puntos donde la persona ejerce criterio entre fases, y el contrapunto reciente: una puerta mal dimensionada deja de proteger y empieza a frenar.

Introducción

Si las fases son el cuerpo de SDD, las puertas de aprobación son su sistema nervioso. Son los puntos donde la persona ejerce criterio, se detectan los problemas antes de que sean caros y el equipo se alinea. Hay tres puertas principales, una entre cada par de fases del núcleo, y la revisión de convergencia al cierre. En este capítulo veremos qué se revisa en cada una, la lógica económica que las justifica y el contrapunto que la evidencia de 2026 añade.

Figura 7. *Tres puertas y el coste de un error: cuanto más tarde se detecta un error, más cuesta corregirlo, y por eso la revisión se concentra entre fases.*

7.1 Qué se revisa en cada puerta

- **Puerta 1, de requisitos a diseño.** Si resolvemos el problema correcto. Historias que reflejan necesidades reales, criterios verificables, alcance manejable y requisitos implícitos detectados, como seguridad, rendimiento o accesibilidad.
- **Puerta 2, de diseño a tareas.** Si el diseño es viable y coherente. Respeta los patrones del código existente, sus decisiones están justificadas y sus dependencias y riesgos identificados.
- **Puerta 3, de tareas a implementación.** Si estas tareas, en este orden, producen el diseño. Cobertura sin huecos, dependencias correctas, prompts precisos y criterios de éxito evaluables.
- **Convergencia.** Si lo construido coincide con lo especificado. Es la envoltura final del capítulo 5, y la única revisión que mira código.

7.2 El coste de un error según la fase

Hay una razón económica detrás de las puertas. El coste de corregir un error crece de forma acelerada con cada fase que atraviesa. Un criterio de aceptación ambiguo se

corrige en requisitos reescribiendo una frase. Si llega al diseño, hay que repensar la solución. Si llega a las tareas, hay que rehacer la descomposición. Si llega a la implementación, hay que descartar y regenerar código. Cada minuto invertido en revisar un requisito ahorra horas de reimplementación.

7.3 Revisar una spec frente a revisar código

Las tres primeras puertas revisan documentos de texto, y eso tiene tres implicaciones. La accesibilidad, porque más personas del equipo, incluidas las que no programan, pueden participar. La velocidad, porque leer una spec es más rápido que leer código. Y el foco, porque la pregunta es si esto es lo que queremos, separada de la de si está bien implementado, que se reserva para la convergencia. Separar las dos preguntas hace cada revisión más eficaz.

La aprobación es un acto, no un trámite. Una puerta que se cruza sin leer no detecta nada y da una garantía falsa. El capítulo 17 lo llama teatro de especificación.

7.4 El contrapunto: cuando la puerta frena

La evidencia de 2026 añade algo que la primera edición de esta guía no tenía. El informe de DORA identifica la revisión manual como el cuello de botella que se satura cuando la velocidad de generación de código sube. Las puertas concentran la revisión, y por eso mismo son el punto donde el sistema se atasca si la capacidad de revisar no crece al ritmo de la capacidad de generar. Una puerta mal dimensionada deja de proteger y empieza a frenar.

Kief Morris ha explorado la misma cuestión desde otro ángulo, el de cómo se reparten personas y agentes dentro de los bucles de la ingeniería de software. Su pregunta es en qué puntos de cada bucle aporta la persona un criterio que el agente no tiene, y en cuáles su intervención solo añade latencia. Es la pregunta que hay que hacerse al diseñar las puertas de un equipo concreto, y su respuesta cambia con la confianza acumulada, como veremos con los boundaries en el capítulo 12.

Las respuestas prácticas al atasco no pasan por eliminar la puerta. Pasan por delegar la aprobación en quien tiene criterio, admitir la aprobación asíncrona, dimensionar las specs para que la revisión sea corta y automatizar la parte de la comprobación que no requiere criterio, como el paso de análisis de coherencia que Spec Kit añade entre documentos.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Describir qué se revisa en cada una de las tres puertas y en la revisión de convergencia.
- Explicar la lógica económica: el coste de corregir crece de forma acelerada por fase.

- Justificar por qué revisar specs es más accesible, rápido y enfocado que revisar código.
- Reconocer una puerta saturada y aplicar respuestas que la descongestionen sin eliminarla.
- Asegurar que la aprobación es un acto deliberado y explícito.

Errores y antipatrones frecuentes

Saltarse puertas por presión de calendario. Es donde el error es más barato de corregir. Saltarlas traslada el coste a las fases caras.

Aprobar sin leer. Una puerta que no se ejerce de verdad no detecta nada, y es peor que no tenerla, porque da una garantía falsa.

Mezclar «¿es lo que queremos?» con «¿está bien implementado?». Separar las dos preguntas es lo que hace eficaz cada revisión.

Responder al atasco de la puerta eliminándola. El cuello de botella se resuelve con delegación, asincronía, specs más cortas y comprobaciones automáticas, no renunciando al criterio.

Para profundizar

· Kief Morris — Humans and Agents in Software Engineering Loops

· DORA — ROI of AI-assisted Software Development

· GitHub Spec Kit

BLOQUE B · EL MÉTODO: FASES, ENVOLTURAS Y PUERTAS

Capítulo 8. Tareas atómicas, oleadas y commits atómicos · EN CONSOLIDACIÓN

Descomponer el diseño en unidades pequeñas y verificables, ejecutarlas en oleadas y registrar un commit *por tarea.*

Introducción

La fase de tareas 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. En este capítulo veremos la anatomía de una tarea atómica, su organización en oleadas y la práctica del commit atómico, y el límite a partir del cual descomponer deja de ayudar.

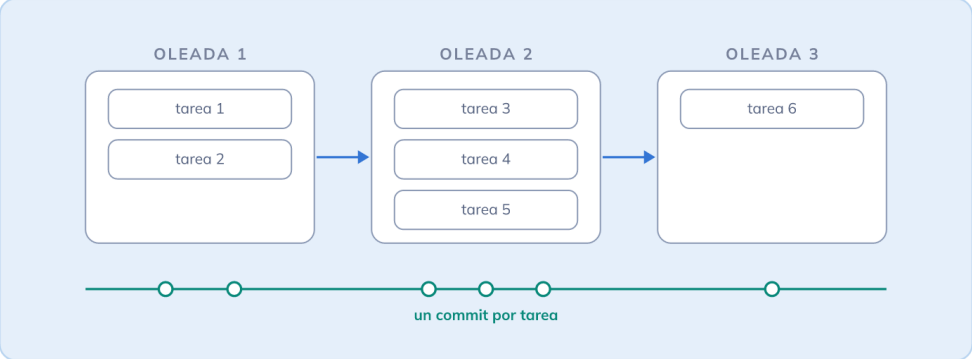


Figura 8. *De la tarea al commit: las tareas independientes se ejecutan en paralelo dentro de una oleada, las oleadas se encadenan y cada tarea cierra con su propio commit.*

8.1 Anatomía de una tarea atómica

Una tarea bien definida afecta normalmente a entre uno y tres ficheros. Si necesita más, probablemente se puede descomponer. Incluye un prompt estructurado con cuatro campos: el rol que asume el agente, la tarea concreta, las restricciones que debe respetar y los criterios de éxito con los que se verificará. Y es verificable de forma aislada, porque su resultado puede evaluarse sin haber completado las demás.

Los cuatro campos no son burocracia. El rol fija el punto de vista, la tarea fija el alcance, las restricciones son los boundaries de tarea del capítulo 12 y los criterios de éxito son lo que permite a la puerta de convergencia comprobar la tarea sin releer todo el código.

8.2 Dependencias y oleadas

Las tareas no son necesariamente secuenciales. Las que no dependen entre sí se agrupan en una misma oleada y se ejecutan en paralelo, cada una por un subagente con contexto limpio, y las oleadas se ejecutan en secuencia. Este modelo aprovecha la capacidad de los agentes para trabajar en paralelo sin que el contexto de una tarea contamine el de otra, y es el patrón de orquestador y trabajadores que describe la guía de Anthropic sobre agentes eficaces.

El orden de las oleadas lo dictan las dependencias del diseño. Primero lo que crea las estructuras de las que dependen las demás, después lo que las usa, al final lo que integra. Una tarea que espera a otra de su misma oleada es una señal de que la descomposición no respetó las dependencias.

8.3 Commits atómicos

Cada tarea genera un commit independiente, y eso tiene tres consecuencias. La reversibilidad granular, porque una tarea defectuosa se revierte sin afectar a las demás. La trazabilidad, porque cada commit se vincula a una tarea, a un diseño y a unos requisitos, que es la cadena que el capítulo 16 muestra útil ante un regulador. Y la bisección eficaz, porque se identifica con precisión qué tarea introdujo un defecto.

El riesgo de la descomposición excesiva. Si el prompt que describe una tarea es más largo que el código que producirá, la descomposición ha ido demasiado lejos. El tamaño correcto es el que permite al agente trabajar con contexto completo y a la persona revisar el resultado de un vistazo, en minutos y no en horas.

La granularidad sigue siendo materia de criterio profesional más que de consenso cerrado, y por eso la ficha conserva su estado. Lo que sí está asentado es la dirección. Piezas pequeñas, verificables por separado y con su propio registro en el historial.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Definir una tarea atómica por su alcance limitado, su prompt de cuatro campos y su verificabilidad aislada.
- Organizar tareas en oleadas según sus dependencias, aprovechando el paralelismo de subagentes con contexto limpio.
- Justificar el commit atómico por su reversibilidad granular, su trazabilidad y su utilidad para la bisección.
- Reconocer el tamaño correcto de una tarea y evitar tanto la sobrecarga como la descomposición excesiva.

Errores y antipatrones frecuentes

Tareas que tocan demasiados ficheros. Reducen la precisión de la implementación. Conviene descomponer más.

Descomposición excesiva. Si el prompt es más largo que el código que produce, el coste supera el beneficio.

Commits que mezclan varias tareas. Pierden la reversibilidad granular y la trazabilidad, y hacen inútil la bisección.

Para profundizar

· Anthropic — Building effective agents

· GitHub Spec Kit

BLOQUE B · EL MÉTODO: FASES, ENVOLTURAS Y PUERTAS

Capítulo 9. Especificar sobre *codebase* existente (brownfield) · EN CONSOLIDACIÓN

En un proyecto con historia la spec describe el delta, no el sistema, y fija lo que existe para que el agente no lo sobrescriba.

Introducción

La literatura sobre SDD nació pensando en proyectos que empiezan de cero. La realidad de la mayoría de los equipos es otra, el trabajo sobre bases de código existentes, a menudo grandes y heredadas. En esos entornos, que el sector llama brownfield, la primera fase empieza mirando el código que ya está, antes de escribir un solo requisito. Y es donde SDD aporta más valor, porque un agente que ignora el contexto existente causa más daño que uno que trabaja sobre un lienzo en blanco.

En este capítulo veremos qué se analiza antes de especificar, cómo se fija lo que existe, por qué el caso nuevo deja de serlo enseguida y una nota sobre el nombre que esta guía usaba antes para este análisis.

Figura 9. *El delta sobre lo que existe: la spec describe el cambio, fija lo que no debe romperse y deja el sistema entero fuera del contrato.*

9.1 El delta, no el sistema

En un proyecto con historia, la spec no describe el sistema entero, describe el delta. Antes de escribir un requisito conviene responder tres preguntas. Qué ficheros se verán afectados por el cambio. Qué patrones, convenciones y dependencias existen ya y deben respetarse. Y qué no debe romperse, es decir, qué efectos colaterales podría tener el cambio en otras partes del sistema. Las respuestas salen de una búsqueda en el código, estructural y semántica, que el agente puede hacer y el equipo revisa.

Ese análisis previo tiene una función preventiva. Evita que los requisitos pidan algo que duplica lo existente, contradice patrones establecidos o ignora dependencias. Si el

sistema ya tiene un mecanismo de envío de correos, el requisito no debe pedir «construir un sistema de correos», sino «extender el mecanismo existente para el nuevo canal». Es la diferencia entre diseñar sobre un terreno que se conoce y diseñar sobre uno que se imagina.

9.2 Fijar lo que existe

El segundo movimiento consiste en dejar por escrito lo que existe para que el agente no lo sobrescriba. Una especificación completa del sistema es impracticable a escala, y además innecesaria. Lo que hace falta es fijar las piezas que el cambio toca y las que no debe tocar. Kiro ofrece generar specs de lo que ya está construido, para que el código heredado quede descrito en el mismo lenguaje que el código nuevo. OpenSpec va más lejos y se declara directamente brownfield primero, con la separación entre la verdad vigente y los cambios propuestos que describe el capítulo 14.

La experiencia de campo confirma el énfasis. Los equipos de Thoughtworks informan de que usan SDD sobre todo en entornos brownfield, precisamente donde la intención poco clara y los supuestos ocultos generan retrabajo. El método rinde más donde el agente tiene más que respetar.

Proporcionalidad invertida. El capítulo 4 mostró que el retorno de la IA es alto en trabajo nuevo y sencillo y bajo en código heredado y complejo. El código heredado es justo donde la spec compensa más, porque es donde el agente se equivoca más.

9.3 Cuándo un proyecto nuevo deja de serlo

En un proyecto nuevo, el análisis del código existente no aplica en la primera iteración, pero sí a partir de la segunda funcionalidad. En cuanto hay código, hay contexto que respetar, y el *greenfield* dura lo que dura el primer incremento. El análisis alimenta además la fase de diseño, que lista qué ficheros se crearán y cuáles se modificarán, y la constitución, que fija los patrones que las siguientes specs darán por supuestos.

9.4 Nota terminológica

La primera edición de esta guía llamaba «Impact Report» al análisis previo que describe este capítulo. Al revisar el tema comprobamos que ese nombre no se usa en el área ni en la documentación de las herramientas, y esta edición lo retira. La práctica es la misma, y la describimos por lo que hace, especificar el delta y fijar lo que existe, en lugar de por un nombre que solo existía en nuestro material.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Responder las tres preguntas del análisis previo sobre un código existente: qué se ve afectado, qué hay que respetar y qué no debe romperse.
- Especificar el delta de un cambio sin intentar describir el sistema completo.

- Fijar por escrito lo que existe para que el agente no lo sobrescriba, con las herramientas que lo facilitan.
- Reconocer cuándo un proyecto nuevo pasa a comportarse como uno existente.
- Explicar por qué el código heredado es donde el método compensa más.

Errores y antipatrones frecuentes

Escribir requisitos sin mirar el código existente. Lleva a pedir cosas que duplican o contradicen lo que ya hay.

Intentar especificar el sistema entero. Es impracticable a escala y no hace falta. La spec describe el delta y fija las piezas que el cambio toca.

Asumir un proyecto nuevo cuando ya hay código. La mayoría del trabajo real es sobre código heredado, e ignorarlo es la fuente principal de efectos colaterales.

Tratar el análisis previo como un trámite. Si no se revisa de verdad, no cumple su función preventiva.

Para profundizar

· OpenSpec

· Kiro — buenas prácticas de specs

· Thoughtworks Technology Radar — GitHub Spec Kit

BLOQUE C · ESCRIBIR BUENAS SPECS

Capítulo 10. La spec inteligente, no extensa · ESTABLECIDO

Una buena spec cubre lo justo para guiar al agente sin abrumarlo; mínima no significa corta, y el prompt se trata como un artefacto que se mantiene.

Introducción

Si una frase resume lo que separa una buena spec de una mala es que mínima no significa necesariamente corta. Una buena spec no escatima detalle cuando importa y no añade información que no contribuye al resultado. En este capítulo veremos los principios que se han asentado para escribirla, las áreas que una spec eficaz cubre, la vuelta de tuerca que aportó en 2026 el desarrollo guiado por prompts estructurados y cómo se calibra el detalle.

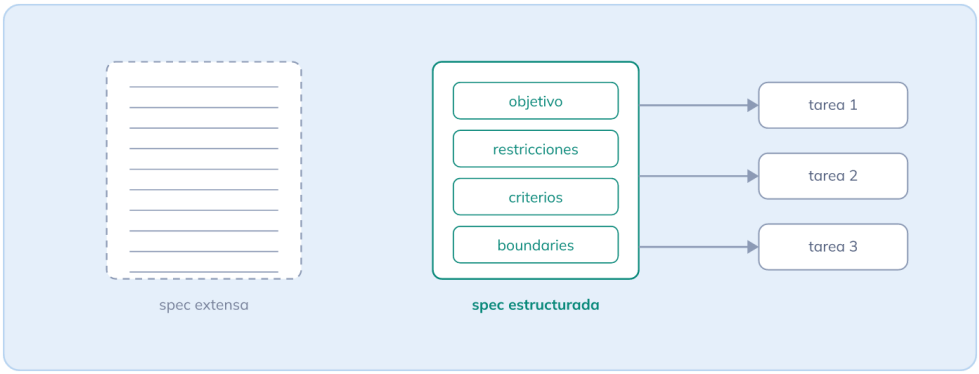


Figura 10. *La spec inteligente, no extensa: cada tarea recibe la porción de la spec que le corresponde, no el documento completo.*

10.1 Los cinco principios de Osmani

Addy Osmani destiló cinco principios que siguen siendo el marco práctico de referencia, y ha seguido desarrollándolos en su trabajo sobre ingeniería agéntica y en su libro *Beyond Vibe Coding*.

1. **Empezar con la visión de alto nivel y dejar que la IA elabore los detalles.** La persona aporta el qué y el porqué, el agente propone el cómo con detalle y la persona valida. La spec se crea entre los dos, no se entrega hecha.
2. **Estructurar la spec como un documento profesional.** Los agentes procesan mejor la información estructurada, con secciones, listas y criterios explícitos.
3. **Dividir las tareas en prompts modulares.** Alimentar al agente con la porción de la spec relevante para su tarea y no con la spec completa, como desarrolla el capítulo 13.
4. **Incorporar autoverificación, restricciones y conocimiento experto.** Instruir al agente para que compare su resultado con la spec, anticipar dónde puede equivocarse y volcar el conocimiento de dominio que solo aporta el profesional.

5. **Tratarla como un documento vivo.** Actualizarla cuando se toman decisiones o se descubre información nueva, porque una fuente de verdad desactualizada es peor que no tenerla, como veremos en el capítulo 14.

10.2 Las áreas que cubre una spec eficaz

El análisis de GitHub sobre más de dos mil quinientos ficheros de configuración de agentes, que Osmani toma como respaldo, identificó seis áreas presentes en las specs eficaces: los comandos del proyecto, la estrategia de pruebas, la estructura del proyecto, el estilo de código, el flujo de trabajo con *git* y los boundaries. Buena parte de esas áreas pertenecen a la constitución del capítulo 6, y por eso una constitución bien hecha hace más cortas todas las specs que vienen después.

10.3 El prompt como artefacto: SPDD y el lienzo REASONS

En abril de 2026, Wei Zhang y Jessie Jie Xia, de Thoughtworks, publicaron en la serie de Martin Fowler su propuesta de desarrollo guiado por prompts estructurados (*structured-prompt-driven development*, SPDD). Su tesis es que el prompt es un artefacto mantenido, versionado y gobernado, y no algo que se genera una vez y se descarta. Es la misma idea que SDD aplica a la spec, llevada a la pieza que el agente recibe en cada tarea.

Su aportación práctica es una plantilla de siete partes, el lienzo REASONS, cuyas iniciales en inglés nombran los requisitos, las entidades, el enfoque, la estructura, las operaciones, las normas y las salvaguardas. Dialoga bien con los principios de Osmani. Los cinco principios dicen cómo pensar la spec, y el lienzo ofrece un molde concreto para que no falte ninguna de las partes que el agente necesita. Es la propuesta de una consultora, de este mismo año y sin adopción medida, y por eso la presentamos como material del capítulo y no como una práctica asentada.

10.4 La proporcionalidad aplicada a la spec

El principio del capítulo 4 vuelve aquí como heurística de detalle. El detalle de la spec se ajusta a la complejidad de la tarea, con la misma pregunta de calibración. Qué ocurre si el agente toma la decisión equivocada en este punto. Si se corrige en dos minutos, no necesita estar en la spec. Si cuesta horas o introduce un defecto sutil, *sí*.

Curar, no acumular. Añadir detalle para cubrir más casos tiene rendimientos decrecientes y después negativos, por la maldición de las instrucciones del capítulo 13. La spec inteligente selecciona lo que el agente necesita saber para esa tarea.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Enunciar y aplicar los cinco principios de Osmani para escribir una spec.
- Cubrir las seis áreas de una spec eficaz, distinguiendo las que pertenecen a la constitución de las propias de cada incremento.

- Explicar la tesis del prompt como artefacto mantenido y usar el lienzo REASONS como molde de una spec de tarea.
- Calibrar el detalle de la spec con la pregunta sobre el coste de una decisión equivocada.
- Distinguir una spec inteligente de una spec simplemente larga.

Errores y antipatrones frecuentes

Añadir detalle para cubrir más casos. Tiene rendimientos decrecientes y luego negativos. La spec inteligente cura, no acumula.

Entregar la spec completa para cada tarea. Sobrecarga el contexto. Conviene alimentar solo la porción relevante.

Generar el prompt y descartarlo. El prompt de una tarea es un artefacto que se revisa y se reutiliza, y descartarlo obliga a redescubrir en cada sesión lo que ya se había decidido.

Escribir la spec una vez y olvidarla. Una spec desactualizada induce a error a quien la consulta.

Para profundizar

· Addy Osmani — How to write a good spec for AI agents

· Addy Osmani — Agentic engineering

· Zhang y Xia — Structured-Prompt-Driven Development

· AGENTS.md

BLOQUE C · ESCRIBIR BUENAS SPECS

Capítulo 11. La notación EARS para criterios de aceptación · EN CONSOLIDACIÓN

Patrones que estructuran el lenguaje natural lo justo para eliminar la ambigüedad sin perder legibilidad.

Introducción

Varias herramientas de SDD usan la notación EARS (*Easy Approach to Requirements Syntax*) para estructurar los criterios de aceptación. La desarrolló Alistair Mavin en Rolls-Royce y la presentó en 2009, y la han adoptado organizaciones del sector aeroespacial y de la ingeniería de sistemas. No es compleja. Es un conjunto de patrones basados en palabras clave que reducen la ambigüedad del lenguaje natural sin sacrificar la legibilidad, y eso la hace útil para agentes.

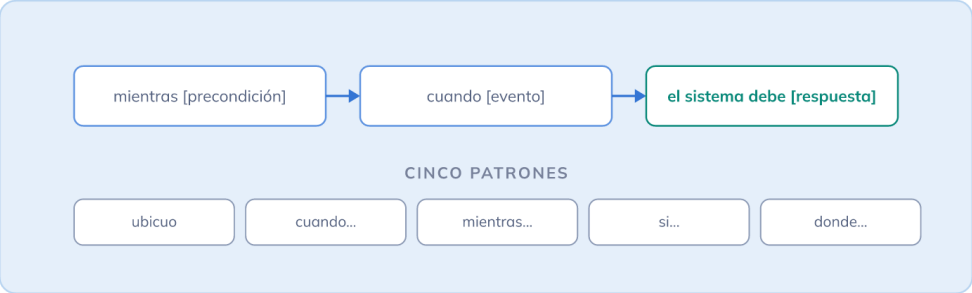


Figura 11. *La plantilla EARS: precondición, evento y respuesta del sistema, con cinco patrones que cubren la mayoría de los requisitos.*

11.1 La plantilla y los cinco patrones

La plantilla básica encaja tres piezas: «Mientras [precondición], cuando [evento], el sistema debe [respuesta]». Sobre ella, EARS define cinco patrones que cubren la mayoría de los requisitos.

- **Ubicuo.** Una propiedad permanente del sistema. «El sistema debe cifrar todas las contraseñas con un algoritmo de coste configurable».
- **Dirigido por evento** (cuando...). Se activa al ocurrir algo. «Cuando se publica una versión de un documento, el sistema debe encolar una notificación en menos de cinco segundos».
- **Dirigido por estado** (mientras...). Activo mientras se cumple una condición. «Mientras el usuario tiene activado el modo no molestar, el sistema debe retener las notificaciones».
- **Comportamiento no deseado** (si...). Gestiona errores y situaciones anómalas. «Si la contraseña se introduce mal tres veces, el sistema debe bloquear la cuenta quince minutos».
- **Opcional** (donde...). Funcionalidad condicionada a la configuración. «Donde la organización ha habilitado el segundo factor, el sistema debe pedir un código tras la contraseña».

Los patrones se combinan. Un requisito puede ser dirigido por estado y por evento a la vez, y la plantilla lo admite sin perder claridad.

11.2 Por qué funciona con agentes

Los agentes responden bien a requisitos estructurados con patrones consistentes, porque la forma fija reduce el margen de interpretación. Un requisito en EARS tiene menos probabilidad de malinterpretarse porque la estructura elimina las ambigüedades más comunes del lenguaje natural, como el sujeto implícito, la condición que no se sabe si es previa o simultánea o la respuesta sin plazo.

La notación no es obligatoria. Es especialmente útil para que los criterios de aceptación sean verificables, es decir, para que al ver el resultado se pueda determinar si se cumplen. «La notificación es rápida» no es verificable. «La notificación llega al canal en menos de treinta segundos en el 99 % de los casos» sí lo es. EARS empuja hacia la segunda forma. La integración en Kiro, que la usa para generar los criterios de sus documentos de requisitos, ha devuelto al primer plano una notación que llevaba más de una década en un sector muy concreto.

Usar donde reduce ambigüedad. EARS se aplica donde el criterio es lo bastante importante para merecer la forma fija. Forzarla en cada línea de la spec convierte una ayuda en una obligación, y su estado de adopción refleja que se usa así, con criterio.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Reconocer y aplicar la plantilla EARS y sus cinco patrones a criterios de aceptación reales.
- Elegir el patrón adecuado según la naturaleza del requisito.
- Explicar por qué los requisitos estructurados reducen la malinterpretación del agente.
- Redactar criterios verificables en lugar de vagos.

Errores y antipatrones frecuentes

Criterios de aceptación vagos. «Rápido», «intuitivo» o «seguro» sin umbral no son verificables ni para una persona ni para un agente.

Forzar EARS donde no aporta. Es una herramienta y no una obligación. Se usa donde reduce ambigüedad de verdad.

Confundir el patrón. Usar «cuando» para una propiedad permanente o «mientras» para un evento puntual desvirtúa el criterio.

Para profundizar

· Alistair Mavin — EARS

· Kiro — buenas prácticas de specs

BLOQUE C · ESCRIBIR BUENAS SPECS

Capítulo 12. El sistema de boundaries: *Always* / *Ask First* / *Never*

· EN CONSOLIDACIÓN

Un marco de delegación de tres niveles, en dos ámbitos, con un formato de instrucciones para agentes que se ha vuelto casi un estándar.

Introducción

Las specs más eficaces no usan una lista plana de reglas, sino un sistema de tres niveles que da al agente un marco de decisión claro sobre cuándo actuar, cuándo consultar y cuándo detenerse. Funciona como la delegación a un profesional competente, con la diferencia de que con un agente las reglas deben ser explícitas, porque carece del sentido común que le permitiría inferirlas. En este capítulo veremos los tres niveles, por qué funcionan y cómo evolucionan, los dos ámbitos en que operan y el formato en que el ecosistema ha convergido para escribirlos.

Figura 12. *Tres niveles de delegación: lo que el agente hace siempre, lo que consulta antes y lo que no hace nunca, con el nivel intermedio moviéndose según crece la confianza.*

12.1 Los tres niveles

- **Always, siempre hacer.** Acciones que el agente ejecuta sin preguntar. Ejecutar las pruebas antes de un commit, seguir las convenciones de nomenclatura, registrar los errores, incluir el manejo de errores en cada punto de entrada.
- **Ask First, preguntar primero.** Acciones que podrían ser correctas pero requieren aprobación por su impacto. Modificar el esquema de la base de datos, añadir dependencias nuevas, cambiar la configuración de integración continua, modificar la *API* pública.
- **Never, nunca hacer.** Líneas rojas absolutas. Hacer commit de secretos, editar directorios de dependencias, eliminar una prueba que falla sin aprobación, modificar la configuración de producción.

12.2 Por qué funciona y cómo evoluciona

El sistema da autonomía donde es segura, porque Always libera de consultar microdecisiones. Crea puntos de control donde hacen falta, porque Ask First concentra la intervención humana en lo de alto impacto. Y establece líneas rojas inequívocas, porque Never elimina categorías enteras de error. Los tres niveles hacen segura la autonomía del agente en la fase de implementación y permiten subir esa autonomía sin subir el riesgo.

Además, el sistema no es estático. A medida que el equipo gana confianza, algo que hoy es Ask First puede pasar a Always, y algo que se creía inocuo puede subir a Ask First tras un incidente. Es autonomía gradual, análoga a cómo una persona delega cada vez más en un profesional que demuestra criterio, y es la respuesta práctica a la pregunta de Morris del capítulo 7 sobre dónde debe entrar la persona en cada bucle.

12.3 Dos ámbitos: proyecto y tarea

Los boundaries operan en dos niveles. En la constitución del proyecto, del capítulo 6, fijan las reglas generales que valen para todo lo que se construya. En cada tarea añaden las restricciones concretas de esa implementación, que es el tercer campo del prompt de cuatro campos del capítulo 8. La combinación da consistencia global y precisión local, y la distinción evita el error más común, que es escribir en la constitución restricciones que solo valen para una tarea.

12.4 AGENTS.md, el formato que se ha vuelto estándar

El soporte del ecosistema ha dejado de ser cosa de cada herramienta. AGENTS.md, un fichero de instrucciones en la raíz del repositorio, se ha convertido en el formato más parecido a un estándar universal de instrucciones para agentes. Lo leen de forma nativa una veintena de herramientas de codificación, lo han adoptado más de sesenta mil proyectos y su especificación está tutelada desde diciembre de 2025 por la Agentic AI Foundation, dentro de la Linux Foundation.

Para el equipo, la consecuencia práctica es que la constitución y los boundaries de proyecto tienen un lugar donde vivir que cualquier agente entiende, con independencia del proveedor. Escribirlos una vez en ese formato evita mantener versiones distintas para cada herramienta, y es la segunda señal que la primera edición de esta guía dejó en vigilancia y que se ha resuelto en positivo.

El formato no sustituye al criterio. Que el fichero exista y lo lean todas las herramientas no dice nada de si sus reglas son las correctas. El contenido lo decide el equipo, y se revisa con la misma deliberación que la constitución.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Clasificar acciones en Always, Ask First y Never con criterio.

- Explicar por qué el sistema de tres niveles supera a una lista plana de reglas.
- Gestionar la evolución de los boundaries como autonomía gradual, en las dos direcciones.
- Distinguir los boundaries de proyecto, en la constitución, de los boundaries de tarea, en el prompt.
- Situar AGENTS.md como formato de instrucciones para agentes y explicar qué resuelve para el equipo.

Errores y antipatrones frecuentes

Una lista plana de prohibiciones. Sin niveles, el agente no sabe qué puede hacer solo, qué consultar y qué evitar.

Boundaries que nunca evolucionan. Mantener todo en Ask First de forma indefinida desaprovecha la confianza ganada y satura de consultas.

Confundir el ámbito de proyecto con el de tarea. «Nunca hacer commit de secretos» es de proyecto. «No tocar los puntos de entrada existentes» es de la tarea concreta.

Mantener un fichero de instrucciones por herramienta. Con un formato común, las versiones se desincronizan sin necesidad y el agente recibe reglas distintas según quién lo invoque.

Para profundizar

· AGENTS.md

· Agentic AI Foundation

· Addy Osmani — How to write a good spec for AI agents

BLOQUE C · ESCRIBIR BUENAS SPECS

Capítulo 13. La maldición de las instrucciones (*instruction overload*) · ESTABLECIDO

Más contexto degrada la capacidad de recuperar lo importante; la contramedida es dar a cada tarea su porción y sistematizar el contexto reutilizable.

Introducción

Uno de los hallazgos más relevantes para SDD proviene de la investigación sobre los propios modelos. Añadir instrucciones a un agente no mejora su comportamiento de forma indefinida. Pasado cierto punto, más contexto degrada la capacidad de recuperar lo importante, y lo que queda en el medio de una ventana larga es lo que peor se recuerda. En este capítulo veremos el hallazgo y su cuerpo de evidencia, la disciplina con nombre que ha nacido de él, sus implicaciones para el diseño de specs y la práctica con la que los equipos lo resuelven.

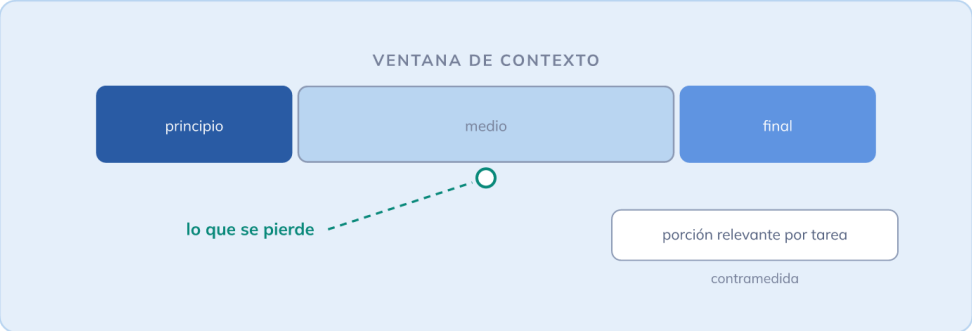


Figura 13. *La maldición de las instrucciones: lo que queda en el medio de una ventana de contexto larga es lo que peor se recuerda, y la contramedida es dar a cada tarea su porción relevante.*

13.1 El hallazgo y su evidencia

El trabajo conocido como la maldición de las instrucciones mostró un resultado regular. A medida que se acumulan instrucciones en un prompt, el rendimiento del modelo al cumplir cada una de ellas cae de forma predecible. Los modelos cumplen las primeras con fiabilidad y empiezan a incumplir las últimas a medida que la lista crece. Con diez reglas detalladas, el agente puede cumplir las primeras y pasar por alto las últimas. Con cincuenta, leerá la spec de forma parcial.

El fenómeno tiene además cuerpo de investigación propio. El trabajo sobre lo que se pierde en el medio (*lost in the middle*) mostró que los modelos recuperan mejor la información situada al principio y al final del contexto que la del centro. Y la investigación sobre la degradación del contexto (*context rot*) ha comprobado que la capacidad de recuperación decae al crecer el número de *tokens* en todos los modelos grandes, con independencia del tamaño nominal de su ventana. Ese doble asentamiento, del nombre y de la evidencia, es lo que hace que esta edición del mapa considere establecido lo que la anterior daba en consolidación.

13.2 De la ingeniería de prompts a la ingeniería de contexto

La respuesta del sector ha sido darle nombre a una disciplina. La ingeniería de contexto se distingue ya de la ingeniería de prompts, porque la pregunta cambia. Ya no pregunta cómo redactar la instrucción. Pregunta qué debe ocupar la ventana en cada momento y qué conviene desalojar, comprimir o delegar a otro agente. La guía de Anthropic sobre ingeniería de contexto para agentes lo formula como la gestión de un recurso escaso, la atención del modelo, que se administra como se administra cualquier presupuesto.

13.3 Implicaciones para el diseño de specs

Las consecuencias para quien escribe specs son cuatro.

- **Una spec más inteligente y no más larga.** Añadir una instrucción puede empeorar el cumplimiento de las anteriores.
- **Descomponer en vez de acumular.** Cinco specs de diez instrucciones, asignadas a las tareas donde son relevantes, rinden más que una de cincuenta. Es lo que hace la fase de tareas del capítulo 8.
- **Priorizar.** Si no se puede dividir más, las instrucciones más importantes, como las de seguridad, van al principio, donde el modelo les da más peso.
- **Separar niveles.** El sistema de boundaries del capítulo 12 crea categorías de urgencia distinta en lugar de una lista plana de igual peso.

Hay señales que delatan una spec sobrecargada. El agente ignora restricciones escritas, cumple lo del principio y no lo del final, mejora cuando se reduce la spec, o acierta con tareas aisladas y falla con varias juntas. Cuando aparecen, el problema no es que el agente sea malo, sino que la spec le pide demasiado a la vez.

13.4 Sistematizar el contexto reutilizable

La contramedida operativa que los equipos han asentado tiene dos partes. La primera es dar a cada tarea la porción de spec que necesita, en lugar de cargar el documento completo en cada interacción. La segunda es sistematizar el contexto reutilizable en ficheros dedicados que el agente carga cuando los necesita, en lugar de engordar un único documento. Thoughtworks lo documenta como la práctica con la que sus equipos mantienen la coherencia y previenen la inflación de instrucciones.

Ese conocimiento institucional, como las convenciones de la casa, los procedimientos de despliegue o los patrones arquitectónicos propios, es lo que el modelo no trae de fábrica. El ecosistema lo empaqueta hoy en piezas con nombre, las *agent skills*, con un formato de fichero que las describe y que el agente invoca cuando la tarea lo pide. La formación del área cierra su temario justamente ahí, automatizando el flujo con skills propias del equipo. Es la respuesta a la maldición de las instrucciones que el capítulo describe, y por ahora vive dentro de este capítulo y no en una ficha propia.

Pocas reglas siempre, las demás cuando toque. La constitución contiene las pocas reglas que aplican siempre. El prompt de cada tarea, las específicas. Las piezas

reutilizables se cargan cuando hacen falta. Y la spec completa existe como referencia, sin cargarse entera en cada interacción.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Explicar la maldición de las instrucciones y su efecto sobre el cumplimiento, con la evidencia que la sostiene.
- Distinguir la ingeniería de contexto de la ingeniería de prompts.
- Responder a la sobrecarga descomponiendo, priorizando y separando niveles, en lugar de añadiendo detalle.
- Reconocer las señales de una spec sobrecargada.
- Sistematizar el conocimiento institucional reutilizable en piezas que el agente carga cuando las necesita.

Errores y antipatrones frecuentes

Responder a un fallo añadiendo más instrucciones. Empeora el problema. La respuesta es dividir mejor, no escribir más.

Listas largas de reglas de igual peso. El modelo no las pondera. Las críticas van primero y separadas por nivel.

Cargar la spec completa en cada interacción. Satura el contexto. Se alimenta solo la porción relevante.

Repetir en cada spec el conocimiento de la casa. Las convenciones y los procedimientos se escriben una vez en una pieza reutilizable, y la spec remite a ella.

Para profundizar

· Anthropic — Effective context engineering for AI agents

· Curse of Instructions (OpenReview)

· Liu et al. — Lost in the Middle (arXiv)

· Chroma — Context Rot

· Thoughtworks Technology Radar — GitHub Spec Kit

BLOQUE C · ESCRIBIR BUENAS SPECS

Capítulo 14. Specs vivas, deriva y la *Clarity Gate* · EN CONSOLIDACIÓN

Mantener la spec alineada con el código que evoluciona; la deriva es el riesgo principal a medio plazo y hay un patrón sencillo para gestionarla.

Introducción

Escribir una buena spec es la mitad del desafío. La otra mitad es mantenerla alineada con el código a medida que el proyecto avanza. La deriva ocurre cuando la spec dice una cosa y el código ha evolucionado hacia otra, y pasa deprisa. Basta una decisión menor del agente que la spec no cubría, o un ajuste durante la implementación que nadie llevó al documento. Una spec derivada es un riesgo activo, porque quien la consulta decide sobre información incorrecta. En este capítulo veremos los tres niveles de vida de una spec, la prueba que detecta los supuestos implícitos y las estrategias para mantenerla viva.

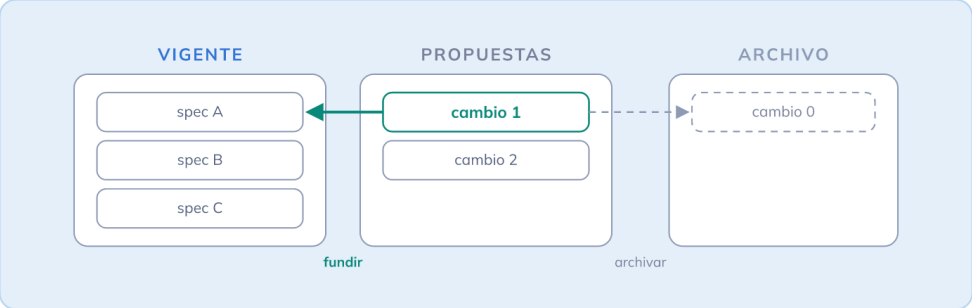


Figura 14. *Verdad vigente y cambios propuestos: las specs vigentes viven separadas de las propuestas de cambio, y lo completado se archiva en lugar de mezclarse con lo vivo.*

14.1 Los tres niveles de vida de la spec

La taxonomía de Böckeler describe cómo distintos equipos gestionan el ciclo de vida.

- **Spec primero** (*spec-first*). Se escribe antes de codificar, guía la tarea actual y se descarta. La deriva no es problema porque no hay spec que mantener, pero se pierde la documentación como activo.
- **Spec anclada** (*spec-anchored*). Se mantiene como documentación viva del sistema. La deriva es un riesgo real que exige disciplina, porque cuando el código cambia, la spec se actualiza.
- **Spec como fuente** (*spec-as-source*). La spec es el artefacto principal, solo se edita la spec y el código se regenera. La deriva desaparece por diseño, pero la regeneración plantea retos de rendimiento y determinismo.

La primera edición de esta guía dejaba el tercer nivel como frontera abierta y en vigilancia. La señal se ha resuelto en sentido negativo. El motor de Tessl, la referencia de ese nivel, sigue en beta cerrada, y la propia compañía ha desplazado el peso de su producto hacia el registro de especificaciones y la gobernanza. Spec como fuente sigue siendo experimental, y la mayoría de los equipos trabaja hoy en el segundo nivel.

14.2 La Clarity Gate

La Clarity Gate, o puerta de claridad, es una prueba de calidad con una sola pregunta. Puede un agente distinto, o una sesión nueva del mismo agente, generar código funcionalmente equivalente leyendo solo la spec. Si sí, la spec es clara y completa. Si no, tiene supuestos implícitos, conocimiento que vivía en el contexto de la conversación original y no quedó escrito, y esos supuestos son la causa principal de la deriva.

Es una prueba que cualquier equipo puede aplicar. Se da la spec a otro agente sin contexto y se compara el resultado. Lo que el segundo agente hace distinto es lo que la spec no decía. Y es también un diagnóstico más limpio que la depuración tradicional. Si la spec es correcta y el código no la cumple, el problema es de implementación y se regenera la tarea. Si la spec es incorrecta, se corrige la spec y se regenera.

14.3 Estrategias para mantener specs vivas

Para los equipos que trabajan con specs ancladas, las prácticas que se han asentado son cuatro. Actualizar la spec en el mismo commit que el cambio, porque «luego» no existe. Versionarla, con un historial de qué cambió y por qué. Revisar el estado de las specs en la retrospectiva, para decidir cuáles siguen vivas. Y, para las specs críticas, crear pruebas de conformidad que verifiquen que el código cumple la spec.

Mientras tanto ha aparecido un patrón práctico y sencillo, el de OpenSpec, que separa lo que es verdad vigente de lo que solo es propuesta de cambio. Las specs del sistema viven en un directorio, las propuestas de cambio en otro, y cuando un cambio se completa, su spec se funde con la vigente y la propuesta se archiva. El patrón no depende de la herramienta. Cualquier equipo puede adoptarlo con dos carpetas y una regla, y resuelve la confusión más frecuente, la de no saber si lo que se lee describe lo que hay o lo que se quería.

Lo completado se archiva. Una spec que describe un cambio ya hecho no es documentación viva del sistema, es historia. Fundir lo aprobado con la verdad vigente y archivar la propuesta mantiene legible lo que importa.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Explicar la deriva y por qué una spec desactualizada es un riesgo activo.
- Distinguir los tres niveles de vida de la spec, situar el estado real del tercero y elegir el adecuado al contexto.
- Aplicar la Clarity Gate para detectar supuestos implícitos en una spec y usarla como diagnóstico ante un fallo.
- Emplear las estrategias de mantenimiento: actualización en el mismo commit, versionado, revisión en retrospectiva y pruebas de conformidad.

- Organizar las specs de un proyecto separando la verdad vigente de las propuestas de cambio y archivando lo completado.

Errores y antipatrones frecuentes

«Lo actualizo luego». Luego no existe. La spec se actualiza en el mismo commit que el cambio o deriva.

Mantener viva toda spec por defecto. El mantenimiento tiene coste, y una spec que nadie consulta ni actualiza es documentación zombi, como veremos en el capítulo 17.

Asumir que la spec está clara sin probarlo. La Clarity Gate revela supuestos implícitos que no se ven desde dentro de la conversación original.

Mezclar en el mismo sitio lo vigente y lo propuesto. Quien lee no sabe si describe lo que hay o lo que se quería, y el agente tampoco.

Para profundizar

· Böckeler — Understanding Spec-Driven Development

· OpenSpec

· Tessl

BLOQUE D · SDD EN EL EQUIPO Y SU ECOSISTEMA

Capítulo 15. Encaje en roles y ceremonias ágiles · EN CONSOLIDACIÓN

Cómo SDD redistribuye el trabajo del equipo, dónde vive la spec respecto al backlog *y al* sprint*, y la tensión entre eficiencia individual y capacidad del equipo.*

Introducción

SDD no es una capa que se añade sin modificar el trabajo existente. Cambia la naturaleza del trabajo de cada rol y la forma de las ceremonias. La skill «Scrum en equipos con IA» de Skill Arena describe la estructura del equipo que trabaja con agentes, con sus roles y sus artefactos. SDD describe cómo ese equipo construye software en el día a día. Son dos caras de la misma adaptación, y en este capítulo veremos la segunda: qué hace cada rol, dónde vive la spec, qué cambia en las ceremonias y qué tensión nueva tiene que gestionar el equipo.

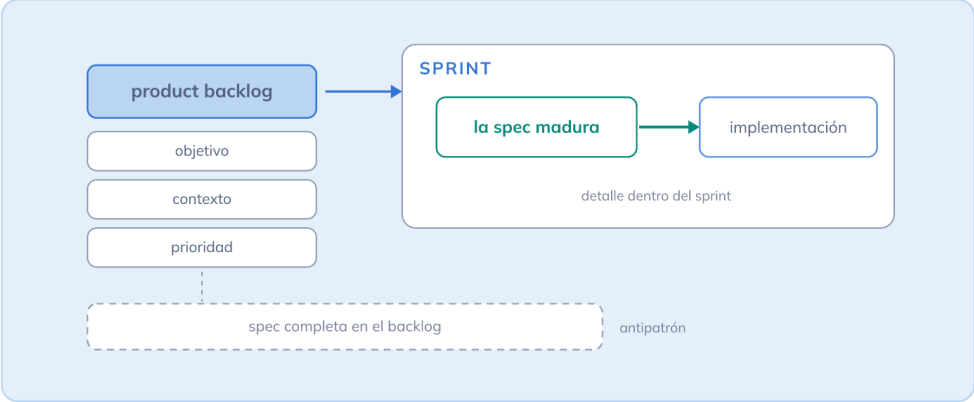


Figura 15. *Dónde vive la spec: el product backlog acuerda objetivo, contexto y prioridad, y el contrato madura dentro del sprint.*

15.1 Los roles

- **Product owner y product architect.** La exigencia de precisión sube, porque los criterios de aceptación pasan de guía a contrato. A cambio, las specs son artefactos que cualquier stakeholder puede leer, lo que acorta el ciclo de feedback con el negocio.
- **Desarrollador y product builder.** La competencia se desplaza del código a analizar, especificar, orquestar y revisar. Gana la capacidad de producir a una velocidad antes imposible, pero solo si las fases previas se hicieron bien, porque la inversión se desplaza de la ejecución a la planificación.
- **Agile enabler.** Facilita las puertas de aprobación como puntos de inspección y vigila los riesgos nuevos, que el capítulo 17 cataloga, para que el equipo mantenga el ritmo iterativo.
- **Stakeholders.** Ganan visibilidad. Pueden leer una spec y opinar sobre prioridades o conformidad antes de que se convierta en código, y la asimetría de información entre quien construye y quien paga o usa se reduce.

15.2 Dónde vive la spec

Durante 2026 ha aparecido doctrina práctica con una tesis nítida. La spec no sustituye al backlog ni al refinamiento. En el *product backlog* y en el refinamiento el equipo acuerda objetivos, contexto y prioridad, y el contrato detallado madura dentro del sprint, como parte de hacer el trabajo. Es ahí donde se decide el detalle, con el trabajo ya empezado y con la información que solo aparece al empezar.

El error de integración más habitual consiste en llevar la spec completa al product backlog. Convierte el backlog en un depósito de documentos que caducan antes de ejecutarse, devuelve al equipo la sensación de waterfall que el capítulo 3 desmontaba y desplaza la conversación de refinamiento, que es donde se acuerda el porqué, hacia un detalle técnico que todavía no toca. La respuesta de Yeret a su propia pregunta va en esta línea. SDD es un paso adelante cuando la spec nace dentro del equipo que construye, y un paso atrás cuando se convierte en un documento que alguien entrega desde fuera.

El backlog acuerda, el sprint especifica. Objetivo, contexto y prioridad en el backlog. Requisitos, diseño y tareas dentro del sprint. La spec aprobada es la *Definition of Ready* de esa funcionalidad, dicha en el lenguaje de SDD.

15.3 Las ceremonias con SDD

El sprint pasa a contener dos tipos de elementos, specs por crear y specs aprobadas listas para implementar, y la planificación del sprint decide cuántas de cada tipo entran. La estimación se desdobla en el esfuerzo de especificación, que es humano y el que consume capacidad real del equipo, y el de implementación, que recae sobre todo en el agente y es más predecible.

La *Definition of Done* se enriquece con cuatro condiciones: el código cumple los criterios de éxito de las tareas, respeta el diseño, los commits son atómicos y la spec se ha actualizado si la implementación difirió. La reunión diaria se centra en el estado de las specs y en las desviaciones, en lugar del repaso mecánico de tareas. Y la revisión de código cambia de pregunta. Se revisa contra la spec, es decir, si cumple lo que dice, y no solo contra el gusto del revisor.

15.4 La paradoja del individuo y el equipo

El trabajo académico del área describe una tensión que el equipo tiene que gestionar. La eficiencia individual sube con los agentes, mientras la capacidad de revisión del equipo se degrada y la estabilidad del sistema baja. Cada persona produce más, y el equipo en conjunto absorbe peor lo producido. Es el mismo cuello de botella que DORA describe desde la entrega, visto desde la organización del trabajo.

La propuesta de Díaz, Gayoso, Cimmino y Pérez es usar las especificaciones como gobierno del comportamiento de los agentes a escala de equipo, con un doble *harness*: controles técnicos alrededor del agente y gobernanza metodológica alrededor del

equipo. El término enlaza con otra skill de Skill Arena, «Harness Engineering», que trata la primera mitad de esa pareja, y el paper identifica además cinco patrones de interacción entre personas y agentes que redefinen los roles humanos. Para el equipo, la lección práctica es que las puertas, la constitución y la Definition of Done son los instrumentos con los que la organización recupera la capacidad de revisión que la velocidad individual erosiona.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Describir cómo cambia el trabajo de cada rol del equipo con SDD, incluidos los no técnicos.
- Situar la spec respecto al product backlog, el refinamiento y el sprint, y reconocer el antipatrón de llevar la spec completa al backlog.
- Explicar los dos tipos de elementos del sprint, el desdoblamiento de la estimación y las condiciones que SDD añade a la Definition of Done.
- Revisar código contra la spec y no solo contra el criterio del revisor.
- Explicar la paradoja entre eficiencia individual y capacidad del equipo, y qué instrumentos del método la compensan.

Errores y antipatrones frecuentes

Tratar SDD como asunto solo de desarrolladores. La especificación es trabajo compartido, y los roles no técnicos ganan voz y visibilidad.

Llevar la spec completa al product backlog. El backlog acuerda objetivo, contexto y prioridad. El contrato madura dentro del sprint, con el trabajo ya empezado.

Estimar solo la implementación. El esfuerzo que consume capacidad del equipo es el de especificación, no el del agente.

Medir el éxito por la producción individual. Si el equipo no puede revisar lo que cada persona produce, la velocidad se convierte en inestabilidad.

Para profundizar

· Díaz, Gayoso, Cimmino y Pérez — SDD for Agentic Software Engineering (arXiv, 2026)

· Yuval Yeret — Is Spec-Driven Development a step forward or back for product development?

· Skill Arena — Scrum en equipos con IA

· Skill Arena — Harness Engineering

· Scrum Manager — Scrum en la era de la IA

BLOQUE D · SDD EN EL EQUIPO Y SU ECOSISTEMA

Capítulo 16. Trazabilidad y supervisión humana como obligación · EMERGENTE

Desde agosto de 2026 el marco de alto riesgo del Reglamento europeo de IA exige registro, supervisión humana y transparencia; SDD no es obligatorio, pero lo que produce sirve al expediente.

Introducción

Desde el 2 de agosto de 2026 se aplica el marco de alto riesgo del Reglamento europeo de inteligencia artificial. Es un hecho reciente, con práctica profesional todavía por hacerse, y por eso esta ficha entra en el mapa como emergente. En este capítulo veremos qué exige la norma, dónde está la frontera entre lo que regula y lo que no, qué aporta SDD al expediente de conformidad y qué no conviene afirmar.

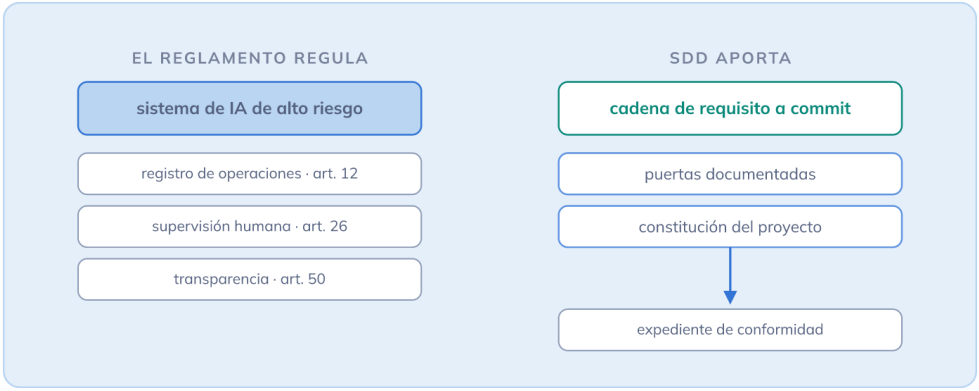


Figura 16. *Lo que regula la norma y lo que aporta SDD: el Reglamento regula el sistema de alto riesgo que se construye, y la cadena de requisito a commit y las puertas documentadas son material para su expediente.*

16.1 Qué exige el marco de alto riesgo

Para los sistemas de IA clasificados como de alto riesgo, el Reglamento exige tres cosas que interesan a este tema. El registro automático de las operaciones del sistema, para que su funcionamiento pueda reconstruirse (artículo 12). La supervisión humana asignada a personas con la competencia, la formación y la autoridad necesarias, que recae en el responsable del despliegue (artículo 26). Y la transparencia sobre el contenido generado por IA (artículo 50). Son obligaciones para quien construye y despliega el sistema, con plazos que ya han vencido para las categorías principales.

16.2 La frontera: el sistema, no la herramienta

Conviene precisar el alcance con cuidado, porque es fácil afirmar algo falso. Lo que el Reglamento regula es el sistema de IA de alto riesgo que el equipo construye, por ejemplo un sistema que decide sobre el acceso a un crédito o a un empleo. No regula el uso de IA para programar ese sistema. Que un equipo use agentes de codificación no lo convierte en sujeto de estas obligaciones, y que no los use no lo exime si su producto es de alto riesgo.

Utilidad, no obligación. SDD no es obligatorio por el Reglamento. Lo que ocurre es que, cuando el producto entra en el ámbito de alto riesgo, los artefactos que SDD produce de todos modos resultan aprovechables para demostrar conformidad. La relación es de conveniencia, y así hay que decirla.

16.3 Lo que SDD aporta al expediente

Cuando el producto sí está regulado, dos piezas que el método ya tenía cambian de valor. La primera es la cadena que va del requisito al código. Cada commit atómico se vincula a una tarea, cada tarea a un diseño y cada diseño a unos requisitos, como vimos en el capítulo 8, y esa cadena es exactamente la trazabilidad que un expediente de conformidad necesita para explicar por qué el sistema hace lo que hace.

La segunda son las puertas de aprobación documentadas. Cada puerta deja constancia de que una persona con criterio revisó y aprobó un documento en un momento concreto, y eso es evidencia de supervisión humana sobre el proceso de construcción. A ellas se suma la constitución del proyecto, que fija por escrito las restricciones que el sistema debe respetar y funciona como declaración de política técnica. Ninguna de las tres piezas se crea para el regulador, y las tres le sirven.

16.4 Lo que no aporta y por qué la ficha es emergente

SDD no cubre por sí mismo las obligaciones del Reglamento. El registro automático de operaciones es una propiedad del sistema en producción, no de su proceso de construcción, y la transparencia sobre el contenido generado se resuelve en el producto, no en la spec. Un equipo que confunda la trazabilidad de su método con la conformidad de su sistema estará cometiendo un error serio.

La ficha es emergente porque la intersección entre SDD y el Reglamento está por explorar en la práctica. Aún no hay expedientes de conformidad publicados que aprovechen la trazabilidad de una spec, ni doctrina de las autoridades sobre qué evidencia del proceso de desarrollo consideran suficiente. Lo que sí es seguro es que el equipo que trabaja con specs aprobadas y commits trazables llega a esa conversación con más material que el que trabaja a partir de conversaciones que nadie conserva.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Enunciar las tres obligaciones del marco de alto riesgo que afectan al desarrollo: registro, supervisión humana y transparencia.
- Situar la frontera entre lo que el Reglamento regula, el sistema de alto riesgo que se construye, y lo que no, el uso de IA para programar.
- Explicar qué artefactos de SDD resultan aprovechables para un expediente de conformidad y por qué.

- Distinguir la trazabilidad del proceso de construcción de la conformidad del sistema en producción.
- Comunicar la relación entre SDD y el Reglamento como utilidad y no como obligación.

Errores y antipatrones frecuentes

Afirmar que SDD es obligatorio por el Reglamento. El Reglamento regula el sistema de alto riesgo, no el método con que se programa. La relación es de utilidad.

Creer que usar agentes convierte al equipo en sujeto regulado. Lo que se regula es lo que el producto hace, con independencia de cómo se construyó.

Confundir trazabilidad del proceso con conformidad del sistema. El registro de operaciones y la transparencia se resuelven en producción. La spec aporta evidencia del proceso, no del sistema.

Ignorar el asunto por ser reciente. Los plazos han vencido, y el equipo que construye productos de alto riesgo necesita el material que SDD produce de todos modos.

Para profundizar

· Reglamento europeo de IA — artículo 12, registros

· Reglamento europeo de IA — artículo 26, obligaciones del responsable del despliegue

· Reglamento europeo de IA — artículo 50, transparencia

BLOQUE D · SDD EN EL EQUIPO Y SU ECOSISTEMA

Capítulo 17. Antipatrones: teatro de especificación y documentación zombi · EN CONSOLIDACIÓN

Los modos de fallo característicos de SDD mal aplicado, la crítica externa que los describe y el papel del agile enabler para vigilarlos.

Introducción

Reconocer los antipatrones forma parte del dominio del método tanto como conocer el flujo. Son los puntos donde SDD deja de aportar valor y empieza a estorbar, y aparecen con facilidad si no se calibra con el principio de proporcionalidad. En este capítulo veremos los cuatro modos de fallo característicos, la crítica externa que los describe sin piedad, una pregunta afín sobre el bucle del agente y el papel de quien los vigila.

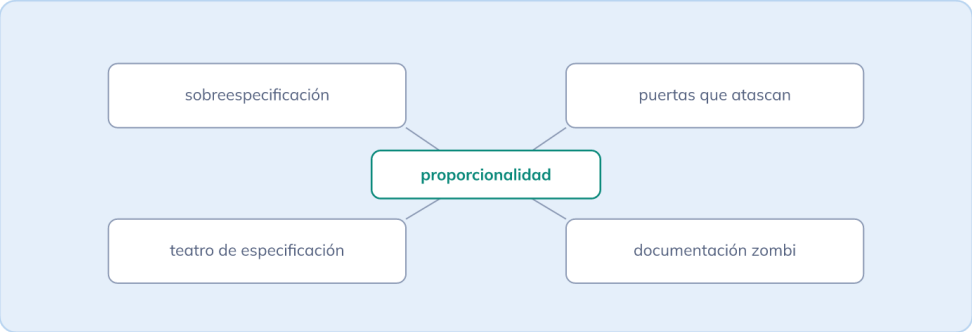


Figura 17. *Cuatro modos de fallo: sobreespecificación, puertas que atascan, revisión que no se ejerce y specs que nadie consulta, los cuatro con la proporcionalidad como remedio.*

17.1 Los cuatro antipatrones

- **Sobreespecificación.** El equipo invierte más tiempo en perfeccionar la spec que en producir software que funciona. Es el mazo para la nuez del capítulo 4 llevado al proceso del equipo, y su forma extrema es la rigidez, aplicar la misma intensidad a un defecto trivial y a una funcionalidad compleja.
- **Puertas como cuellos de botella.** Una aprobación se bloquea porque quien debe aprobar no está disponible o no da abasto. La respuesta es la del capítulo 7, delegar, admitir la aprobación asíncrona y acortar las specs, y no eliminar la puerta.
- **Teatro de especificación.** Se escriben specs que nadie revisa de verdad, y el proceso se ejecuta de forma mecánica sin aportar el valor de la revisión. Una puerta que no se ejerce es peor que su ausencia, porque da una garantía falsa.
- **Documentación zombi.** Specs que nadie consulta ni actualiza. Una spec que no sirve no es documentación viva, es lastre, y decidir qué specs mantener y cuáles dejar morir forma parte de la gestión, como vimos en el capítulo 14.

17.2 La crítica externa

La crítica de Zaninotto que vimos en el capítulo 3 merece leerse de primera mano, porque describe estos fallos con la nitidez de quien los ha sufrido. Documentos enormes que

nadie lee entero. Pérdida del contexto del código, porque la spec habla de un sistema imaginado y no del que existe. Trabajo de revisión duplicado, primero de la spec y después del código que no la cumple. Y una falsa sensación de seguridad, la de creer que el proceso protege cuando solo se ejecuta.

Cada uno de esos fallos tiene su remedio en un capítulo de esta guía, y esa correspondencia es la mejor prueba de que el método, bien aplicado, ya los tiene previstos. La crítica describe lo que pasa cuando se aplica mal, y por eso es tan útil.

17.3 Teatro o valor: la misma pregunta al bucle del agente

Böckeler ha planteado la misma pregunta que este capítulo hace a las specs, pero al bucle del agente. Al examinar si el desarrollo guiado por pruebas dentro de ese bucle aporta valor real o solo teatro, encuentra que un agente puede escribir pruebas que pasan sin que las pruebas comprueben nada relevante, del mismo modo que un equipo puede escribir specs que se aprueban sin que nadie las lea. El paralelismo es instructivo. La forma de una práctica no garantiza su fondo, y la pregunta que hay que hacerse ante cualquier ritual del método es si alguien ejerce criterio en él o si solo se ejecuta.

La señal del teatro. Cuando una puerta nunca rechaza nada, o una spec nunca cambia después de su revisión, o unas pruebas nunca fallan, lo más probable no es que todo esté bien. Es que nadie está mirando.

17.4 El papel del agile enabler

Estos riesgos no se corrigen solos. El agile enabler asegura que se dedica tiempo real a la revisión sin que las puertas se vuelvan trámite, modera los desacuerdos sobre las specs, detecta cuándo el proceso genera coste sin retorno y ajusta la intensidad, y vigila que las specs no se conviertan en pequeños proyectos en cascada. Es la función que mantiene SDD ágil en la práctica, y la razón de que el método necesite a alguien que mire el proceso además de a quienes lo ejecutan.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Identificar los cuatro antipatrones de SDD mal aplicado y el capítulo de la guía que remedia cada uno.
- Distinguir el teatro de especificación y la documentación zombi de la práctica que aporta valor.
- Resumir la crítica externa a SDD y usarla como lista de comprobación preventiva.
- Aplicar la pregunta «teatro o valor» a cualquier ritual del método, incluidas las pruebas del agente.
- Describir el papel del agile enabler en la vigilancia y corrección de estos riesgos.

Errores y antipatrones frecuentes

Confundir rigor con cantidad de documentación. Más specs no hacen mejor SDD. La sobreespecificación es un fallo, no una virtud.

Mantener puertas que nadie ejerce. El teatro de especificación da una garantía falsa. Conviene aprobar con criterio o replantear la puerta.

Acumular specs zombi. Una spec que nadie consulta ni actualiza es lastre. Decidir qué dejar morir es parte de la gestión.

Fiarse de que las pruebas pasan. Unas pruebas que nunca fallan pueden no estar comprobando nada. La pregunta es qué verifican, no cuántas son.

Para profundizar

· Marmelab — Spec-Driven Development: The Waterfall Strikes Back

· Böckeler — TDD inside the agent loop: theater or actual value?

· Böckeler — Understanding Spec-Driven Development

BLOQUE D · SDD EN EL EQUIPO Y SU ECOSISTEMA

Capítulo 18. El ecosistema de herramientas · EMERGENTE

El instrumental crece deprisa y la lista concreta caduca antes que el método; lo que conviene aprender son los criterios de elección.

Introducción

El ecosistema de herramientas alrededor de SDD es la zona más volátil del tema. La primera edición de esta guía dedicaba este capítulo a una fotografía del panorama a junio de 2026, con versiones y cifras, y advertía de que caducaría antes que el resto. Tres meses después había caducado punto por punto, lo que demuestra que la advertencia funcionaba y que la fotografía no era el formato adecuado. En esta edición hacemos otra cosa. Damos los criterios con los que se elige una herramienta, que no caducan, y las señales que indican que el método se está institucionalizando.

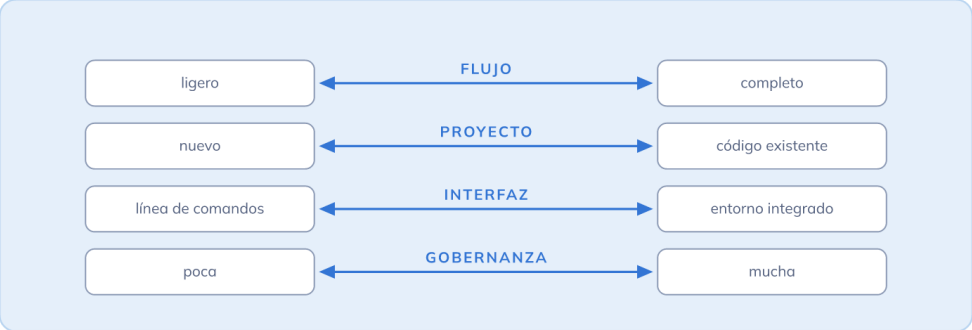


Figura 18. *Cuatro criterios de elección: flujo ligero o completo, proyecto nuevo o código existente, línea de comandos o entorno integrado, y cuánta gobernanza hace falta.*

18.1 Por qué no damos una lista

Una lista de herramientas con sus versiones dentro de un documento caduca en semanas. La implementación de referencia publica versiones con frecuencia, las alternativas aparecen y desaparecen, y las cifras de adopción cambian a diario. Un profesional que aprende la lista aprende algo que dejará de ser cierto antes de que lo necesite. Un profesional que aprende los criterios puede evaluar cualquier herramienta que aparezca, incluidas las que no existen todavía.

18.2 Los cuatro criterios

Los criterios que separan hoy unas herramientas de otras son cuatro.

1. **Flujo ligero o completo.** Algunas herramientas implementan el flujo entero, con constitución, fases, puertas y convergencia. Otras son deliberadamente ligeras y se limitan a organizar las specs y sus cambios. El principio de proporcionalidad del capítulo 4 se aplica también a la elección del instrumento.
2. **Proyecto nuevo o código existente.** Hay herramientas pensadas para empezar de cero y herramientas que se declaran brownfield primero, con capacidad de

describir lo que ya existe. El capítulo 9 explica por qué la segunda familia importa más de lo que la literatura sugería.

3. **Línea de comandos o entorno integrado.** Una herramienta de línea de comandos se añade al flujo existente del equipo y funciona con el agente que ya usa. Un entorno integrado impone su propio flujo y su propio agente, a cambio de una experiencia más guiada.
4. **Gobernanza y trazabilidad.** Cuánto necesita el equipo que la herramienta registre, versione y permita auditar. Es el criterio que sube de peso cuando el producto entra en el ámbito del capítulo 16.

La advertencia de fondo se mantiene. SDD es el método y no la herramienta, y dominar una herramienta no es dominar SDD. Los fundamentos de los bloques A, B y C son estables, y las herramientas que los implementan son la capa que más se mueve.

18.3 Señales de institucionalización

Sin dar cifras que caducan, hay señales de 2026 que indican que el método se está asentando como práctica y no como moda. La implementación de referencia publicó su versión 1.0 en agosto de 2026, al cumplir un año, con un razonamiento revelador. Renuncia a prometer la estabilidad que un 1.0 solía significar, porque refactorizar se ha vuelto barato, y reivindica en su lugar la adaptabilidad. El método tiene ya entrada propia en el Technology Radar de Thoughtworks, formación reglada en un curso de DeepLearning.AI y literatura académica, como vimos en los capítulos 2 y 15.

Y el ecosistema se ha ampliado en lugar de concentrarse. Alrededor de la implementación de referencia han cuajado opciones con perfiles distintos, desde entornos integrados hasta herramientas ligeras orientadas al código existente, y las comparativas que se publican tienen ya media docena de nombres estables. La proliferación no es dispersión. Es la señal de que hay demanda suficiente para que varias respuestas convivan.

Contrastar cualquier fotografía, incluida esta. El panorama concreto se consulta en las comparativas que se mantienen al día, no en un documento con fecha. Lo que este capítulo ofrece es el criterio para leerlas.

18.4 Por qué la ficha sigue siendo emergente

La ficha es emergente porque el instrumental sigue en plena formación, y a la vez es la que da la señal más clara de que el método se institucionaliza. Las dos cosas son compatibles. Un método se asienta cuando sus herramientas empiezan a competir, y las herramientas compiten cambiando deprisa. Invertir el aprendizaje en el método, y en los criterios para elegir el instrumento, es lo que protege al profesional de la obsolescencia de cualquier lista.

Lo que debes saber hacer

Al dominar este tema, un profesional es capaz de:

- Explicar por qué una lista fechada de herramientas no es la forma adecuada de aprender el ecosistema.
- Aplicar los cuatro criterios de elección a una herramienta concreta y justificar la elección para un equipo dado.
- Distinguir el método de SDD del instrumento que lo implementa.
- Reconocer las señales de institucionalización del método y qué indica cada una.
- Consultar las comparativas al día en lugar de tomar una fotografía como referencia permanente.

Errores y antipatrones frecuentes

Confundir el método con la herramienta. Dominar una herramienta no es dominar SDD. La herramienta cambia y el método permanece.

Elegir la herramienta sin entender el método. Lleva a aplicar el flujo de forma mecánica, sin criterio sobre cuándo y con qué intensidad.

Elegir la herramienta más completa por defecto. Un flujo completo para un equipo que necesita organizar specs ligeras es el mazo para la nuez en versión de instrumento.

Tomar cualquier panorama como permanente. Es la capa más volátil. Se contrasta con la edición vigente del mapa y con las comparativas al día.

Para profundizar

· GitHub Spec Kit — releases

· Spec Kit turns one and ships 1.0.0

· Thoughtworks Technology Radar — GitHub Spec Kit

· Böckeler — Understanding Spec-Driven Development

· DeepLearning.AI — Spec-Driven Development with Coding Agents