DOCUMENTO METODOLÓGICO · GUÍA DE APLICACIÓN

# De Scrum a Spec-Driven Development

*Un método para equipos que ya trabajan con Scrum: cómo aplicarlo, cómo reutilizar sus ceremonias y cómo medir la madurez de la adopción*

Jorge Roberts

Versión 1.0 · octubre de 2026

**Resumen.** Cuando un agente de inteligencia artificial genera la mayor parte del código, el costo de construir cae en órdenes de magnitud, mientras que el costo de decidir qué construir y de validar lo construido se mantiene, porque sigue dependiendo de personas. Scrum se diseñó sobre la estructura de costos inversa: iterar sobre código era la forma más barata de aprender. Este documento presenta Spec-Driven Development (SDD) como respuesta a ese cambio y lo hace en términos operativos, para un lector que conoce Scrum y nada más. Se define el ciclo de trabajo, sus artefactos y sus puntos de control (gates); se muestra cómo cada ceremonia y artefacto de Scrum se conserva con un propósito nuevo; se propone un conjunto de métricas que reemplaza a los story points y la velocity; se discute por qué SDD no es un regreso a la cascada; y se describe un modelo de madurez de tres niveles (spec-first, spec-anchored y spec-as-source), con spec-anchored como objetivo y los mecanismos concretos para llegar a él. Se incluye un ejemplo completo en una aplicación móvil, un plan de transición de ocho semanas y un catálogo de los modos de falla de la adopción parcial.

***Palabras clave:*** spec-driven development, Scrum, desarrollo asistido por agentes, diseño por contrato, quality gates, modelo de madurez.

---

## 1 Introducción

Scrum resolvió un problema concreto: en contextos de alta incertidumbre, especificar todo de antemano era caro y casi siempre equivocado. La alternativa fue construir algo pequeño, mostrarlo, aprender y corregir en el ciclo siguiente. Esa estrategia funciona mientras construir sea la forma más barata de descubrir. El sprint, la estimación en story points, la velocity y la demo como momento de descubrimiento son consecuencias de ese supuesto.

Con agentes de IA en el circuito de desarrollo, el supuesto deja de valer. Generar una implementación cuesta minutos. Lo que sigue costando lo mismo es lo que hace una persona: entender el problema, decidir el comportamiento correcto y verificar que lo generado lo cumple. Si un equipo incorpora agentes y conserva intacto el método, obtiene más código con la misma calidad de decisión, y el cuello de botella se traslada a la revisión, donde se acumula sin que ninguna métrica de Scrum lo muestre.

El fenómeno tiene nombre: la **paradoja de la productividad de la IA**. Los asistentes de codificación aumentaron de manera notable la velocidad individual, pero las métricas del sistema (tiempo de entrega, frecuencia de despliegue, tasa de fallas en cambios) se mantuvieron estables o empeoraron en muchas organizaciones. La explicación es un diagnóstico equivocado del problema original: el cuello de botella nunca fue la velocidad de escritura, sino la ambigüedad sobre qué había que construir. Acelerar la generación sobre requisitos ambiguos no produce mejor software: amplifica la disfunción existente, con más soluciones incorrectas, más deuda técnica y más horas de revisión y corrección.

Spec-Driven Development reorganiza el trabajo alrededor de ese nuevo cuello de botella. El equipo itera sobre la especificación, que es barata de corregir, y genera el código contra lo aprobado. La especificación deja de ser documentación y pasa a ser el contrato contra el cual se acepta o se rechaza el resultado.

**QUÉ OBTIENE EL LECTOR**

1. **Un método aplicable desde la primera semana**: el ciclo SDD, sus artefactos, una plantilla de especificación y un ejemplo completo (secciones 3 y 4).
2. **Las ceremonias de Scrum reconvertidas**: qué se conserva de cada una, qué cambia, qué artefacto produce y cuál es su anti-patrón (sección 6).
3. **Un modelo de madurez de tres niveles** (spec-first, spec-anchored y spec-as-source), con spec-anchored como objetivo, los mecanismos para pasar de un nivel al otro y una autoevaluación (sección 8).

**Alcance.** El documento es agnóstico de herramienta. Los conceptos se aplican con cualquier agente de codificación y cualquier sistema de control de versiones. Se asume un equipo de producto de cinco a diez personas que hoy trabaja con Scrum y que ya usa, o va a usar, agentes de IA para generar código.

# 2 Marco conceptual

## 2.1 *Definiciones*

**Especificación (spec).** Documento versionado que describe el comportamiento esperado de una funcionalidad en términos verificables: precondiciones, transiciones, poscondiciones, invariantes, criterios de aceptación y exclusiones explícitas. No describe cómo implementarlo.

**Contrato.** La spec aprobada. Es la referencia contra la cual se acepta o rechaza todo lo que se genera. La noción proviene del diseño por contrato (Meyer, 1992), que a su vez formaliza la terna precondición, acción, poscondición de Hoare (1969).

**Plan técnico.** Documento derivado de la spec que describe cómo se va a implementar: componentes afectados, decisiones de diseño, dependencias y riesgos. Lo propone el agente y lo valida una persona.

**Gate.** Punto de control que bloquea el avance hasta que una persona con autoridad (o una prueba automática) aprueba el artefacto. Un gate que no bloquea es una sugerencia.

**Contexto del proyecto (constitución).** Conjunto mínimo de archivos versionados que el agente lee antes de cada tarea: principios y reglas no negociables, tecnologías aprobadas y prohibidas, arquitectura de referencia y requisitos de seguridad. Es la memoria institucional del equipo escrita para una máquina, y evita que el agente invente sus propias convenciones. Los agentes lo leen; solo las personas lo modifican.

**Doble perspectiva.** SDD exige dos especificaciones que convergen antes de generar: la de *producto*, que define qué se construye y por qué (visión, recorridos de usuario, criterios orientados al valor), y la de *ingeniería*, que define cómo debe construirlo el agente (el contexto del proyecto). La spec de cada funcionalidad vive en la intersección de ambas.

**Criterio de aceptación ejecutable.** Prueba automática derivada de la spec antes de que exista el código. Reemplaza la Definition of Done como acuerdo verbal.

**Deriva (drift).** Divergencia entre lo que dice la spec y lo que hace el código. Aparece cada vez que alguien corrige comportamiento en el código sin actualizar antes la spec.

**Calibración.** Corrección del contexto del proyecto (no del artefacto puntual) cuando un mismo error aparece más de una vez. Es el mecanismo por el cual el sistema aprende.

**Validador.** Persona con autoridad explícita para rechazar un artefacto en un gate. Su rechazo no se revierte por presión de plazo.

## 2.2 *Las tres diferencias que importan*

Scrum y SDD comparten el rechazo a la planificación predictiva exhaustiva, la entrega incremental y la orientación al valor. Las diferencias son tres, y el resto del método se deduce de ellas.

**Primera: cambia el sustrato de la iteración.** Los dos modelos iteran. Scrum itera sobre código: construye una hipótesis, la expone y la corrige en el ciclo siguiente. SDD itera sobre el contrato: refina la spec, la valida y genera contra lo aprobado. La razón es económica. Corregir un párrafo de la spec cuesta minutos. Corregir la misma decisión después de que un agente la propagó a decenas de archivos, esquemas y pruebas cuesta días, y suele hacerse mal bajo presión de

entrega. La relación entre el momento en que se detecta un defecto y su costo de corrección está documentada desde hace décadas (Boehm, 1981). Los agentes la acentúan, porque propagan una decisión equivocada a más lugares y más rápido que una persona.

**Segunda: cambia el mecanismo de cumplimiento.** En Scrum, el cumplimiento de los acuerdos depende de la disciplina del equipo, y un acuerdo incumplido se discute en la retrospectiva. En SDD el cumplimiento depende de gates que bloquean el avance: la IA propone, las personas deciden y el flujo lo impone. Un equipo que conserva las specs pero no el bloqueo adoptó el vocabulario de SDD, no el método.

**Tercera: cambia lo que se mide.** Los story points miden el esfuerzo de construir. Cuando construir cuesta minutos de cómputo, la métrica pierde su referente y empieza a premiar volumen. En SDD se estima y se mide el esfuerzo de validación, que es el recurso escaso. El corolario es contraintuitivo y conviene decirlo el primer día: una funcionalidad grande pero aislada y bien entendida puede estimarse bajo, y un cambio chico que toca el modelo de usuarios o un esquema heredado puede estimarse alto. Lo que pesa es el riesgo de validar, no el tamaño.

**Tabla 1.** Comparación estructural entre Scrum y SDD.

| Dimensión | Scrum | Spec-Driven Development |
|---|---|---|
| **Supuesto económico** | Construir es la forma más barata de aprender. | Construir es casi gratis; decidir y validar son lo caro. |
| **Objeto de la iteración** | El incremento de código. | La especificación. |
| **Fuente de verdad** | El código y la conversación del equipo. | La spec aprobada y el contexto del proyecto. |
| **Cumplimiento** | Disciplina colectiva; retrospectiva. | Gates bloqueantes con validador nombrado. |
| **"Terminado"** | Definition of Done acordada. | Criterios ejecutables derivados de la spec, en verde. |
| **Unidad de estimación** | Story points (esfuerzo de construcción). | Esfuerzo de validación, ponderado por riesgo. |
| **Unidad de entrega** | El sprint. | La funcionalidad validada, cuando pasa sus gates. |
| **Recurso escaso** | Capacidad de desarrollo. | Capacidad de validación de personas expertas. |
| **Aprendizaje del sistema** | En las personas. | En las personas y en el contexto versionado. |

## 2.3 *Lo que se conserva y lo que se retira*

La mitad de la resistencia a la transición viene de suponer que SDD elimina Scrum. No es así. Se conservan la entrega incremental (más granular que en Scrum, porque se entrega por funcionalidad validada y no por sprint), la adaptación al cambio (ahora se ejecuta sobre la spec), la demo al cliente (como comunicación de resultados, no como descubrimiento), la retrospectiva (que gana peso, porque pasa a auditar el sistema de trabajo) y el refinamiento (que se vuelve la ceremonia central).

> *Se retiran cuatro cosas: la historia de usuario como insumo directo de implementación, los story points y la velocity, el sprint como unidad de entrega y la estimación en horas de desarrollo. Las cuatro se calibraron sobre una estructura de costos que dejó de existir. Mantenerlas "solo para reportar" es el atajo que más daña la transición (sección 10).*

La historia de usuario no desaparece: queda como enunciado de valor en la parte superior de la spec. Lo que se retira es su uso como orden de trabajo. Un agente que recibe una historia de usuario completa por probabilidad todo lo que la historia no dice, y lo hace sin avisar.

## 2.4 *SDD no es cascada*

La objeción más frecuente de un equipo ágil es que SDD es la cascada de los años setenta con otro nombre: documentación densa antes del código y parálisis por análisis. La equivalencia es superficial, porque confunde escribir la especificación *temprano* con *congelarla*. En cascada la documentación es un artefacto de gobierno, diseñado para obtener aprobaciones y resistir el cambio. En SDD la spec es un mecanismo de retroalimentación, pensado para iterarse y versionarse a medida que la implementación revela realidades nuevas (Brooker, 2026).

**Tabla 2.** Propiedades estructurales de cascada y SDD.

| Propiedad | Cascada | Spec-Driven Development |
|---|---|---|
| **Naturaleza del documento** | Artefacto rígido de gobierno, para obtener aprobaciones formales por fase. | Mecanismo activo de retroalimentación técnica; artefacto vivo. |

| Propiedad | Cascada | Spec-Driven Development |
|---|---|---|
| **Formato** | Documentos estáticos (procesador de texto, PDF, wikis aisladas). | Texto estructurado legible por máquina (Markdown, YAML), versionado junto al código. |
| **Momento de la validación** | Al final, en una fase separada, meses después del diseño. | Continua, dentro del ciclo de cada funcionalidad, en el pipeline. |
| **Respuesta al cambio** | Resistente: volver atrás exige burocracia que desalienta adaptarse. | Adaptativa: revisar la spec es la respuesta esperada a lo que se aprende. |
| **Deriva documento-código** | Reconciliación manual y agotadora; el documento queda obsoleto casi de inmediato. | Detección automática mediante pruebas derivadas del contrato. |
| **Roles** | Analistas que escriben aislados y desarrolladores que construyen en silencio. | Colaboración: las mismas personas escriben la intención y validan, en ciclo con el agente. |
| **Alcance y tiempo de ciclo** | Macro: todo el sistema, durante meses, antes de la primera línea. | Micro: una funcionalidad por vez, en ciclos de horas o días. |

La diferencia decisiva es el tratamiento de la deriva. En cascada, mantener el documento sincronizado con un código que cambia todo el tiempo cuesta demasiado esfuerzo humano, así que el documento se abandona y el código asume el rol de fuente de verdad. En SDD, spec, código y pruebas conviven en el mismo repositorio y bajo el mismo control de versiones: actualizar la spec es un acto de refactorización semántica, no un trámite.

### 2.5 *De dónde viene*

SDD no es una invención aislada, sino la convergencia instrumentada de prácticas previas. Del **diseño por contrato** (Hoare, 1969; Meyer, 1992) toma la terna precondición, transición y poscondición, y la noción de invariante. Del **Behavior-Driven Development** (North, 2006) toma la descripción del comportamiento mediante ejemplos concretos en lenguaje estructurado, del tipo *dado, cuando, entonces*. La limitación histórica de BDD era que los escenarios quedaban como documentación consultiva o alimentaban pruebas que costaba mucho mantener. Con agentes, esos escenarios se vuelven ejecutables y, además, restringen lo que el agente puede generar.

También converge con la gestión de requisitos. La distinción clásica entre *requisito* (la necesidad de negocio que motiva el trabajo) y *especificación* (la descripción precisa de cómo se satisface) es exactamente la que SDD vuelve operativa: la intención vive en el backlog, la especificación en el repositorio, y el puente entre ambas es trazable. En industrias reguladas, como la aeronáutica (DO-178C), los dispositivos médicos (IEC 62304) o la automotriz (ISO 26262), esa trazabilidad deja de ser un esfuerzo manual al final del proyecto y pasa a ser un subproducto del ciclo diario.

## 3 El ciclo Spec-Driven

El ciclo tiene seis fases y tres gates humanos, más un circuito de calibración. La Figura 1 lo resume. Las flechas de retorno son tan importantes como las de avance: definen a dónde vuelve el trabajo cuando algo falla.

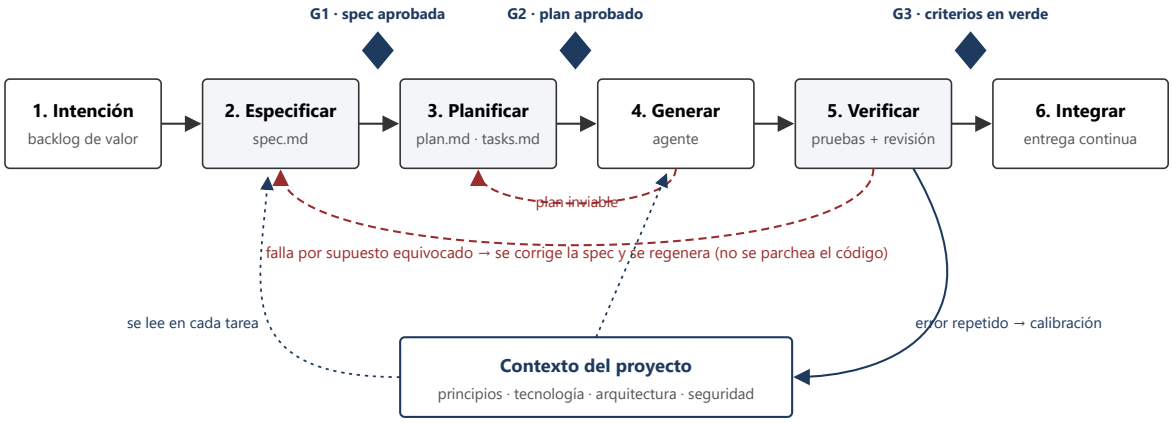



**Figura 1.** Ciclo Spec-Driven: seis fases, tres gates y dos circuitos de retorno. La línea discontinua roja indica a dónde vuelve el trabajo ante una falla; la azul, el aprendizaje que queda en el contexto del proyecto.

## 3.1 *Las fases, una por una*

**Tabla 3.** Entradas, actividades, salida y control de cada fase.

| Fase | Entrada | Actividad | Salida | Control |
|---|---|---|---|---|
| **1. Intención** | Necesidad de negocio, feedback, métrica. | El Product Owner enuncia el resultado buscado y por qué importa, sin solución. | Ítem en el backlog de intenciones. | Ninguno; es insumo. |
| **2. Especificar** | Intención priorizada y contexto del proyecto. | Se redacta la spec (con o sin ayuda del agente), se resuelven ambigüedades y se declaran las incógnitas. Se derivan los criterios de aceptación. | `spec.md` | **G1**: el validador de negocio y el técnico aprueban. |
| **3. Planificar** | Spec aprobada. | El agente propone el plan técnico y lo descompone en tareas pequeñas, cada una verificable por sí misma (micro-incrementos que se integran por separado, como pull requests apilados). | `plan.md`, `tasks.md` | **G2**: el validador técnico aprueba diseño y riesgos. |
| **4. Generar** | Plan y tareas aprobados. | El agente implementa tarea por tarea, con las pruebas derivadas de la spec escritas primero. Ante una bifurcación de diseño no prevista, se detiene y presenta opciones cerradas en lugar de adivinar. | Código y pruebas en una rama. | Pruebas locales; ninguna persona escribe código salvo excepción documentada. |
| **5. Verificar** | Rama con implementación. | Corren las pruebas automáticas. La revisión humana se concentra en lo que las pruebas no cubren: seguridad, decisiones de diseño, casos no previstos. | Informe de verificación. | **G3**: criterios en verde y revisión aprobada. |
| **6. Integrar** | Verificación aprobada. | Se integra a la rama principal y se despliega con la cadencia que el producto permita. | Funcionalidad entregada. | Pipeline de integración continua. |

## 3.2 *Qué hacer cuando algo falla*

La regla más importante del ciclo es que **la corrección va al nivel donde se originó el error**. Si el error está en la spec, se corrige la spec y se regenera. Si es un error que ya ocurrió antes, se corrige el contexto del proyecto. Parchear el código directamente es la forma más rápida de volver a Scrum sin darse cuenta: el código pasa a contradecir la spec y, en dos semanas, nadie la lee.

**Tabla 4.** Decisión ante un resultado incorrecto.

| Situación | Decisión | Justificación |
|---|---|---|
| El resultado no cumple la spec, pero la spec es correcta. | **Regenerar** | El agente se desvió. Se rechaza y se regenera con el mismo contrato; si se repite, se calibra. |
| El resultado cumple la spec, pero la spec estaba mal. | **Corregir spec y regenerar** | El error es de decisión, no de ejecución. Corregir el código dejaría la spec desactualizada. |
| El mismo tipo de error aparece por segunda vez. | **Calibrar** | Es un problema del contexto, no del artefacto. Se agrega o corrige la regla correspondiente para que no vuelva. |
| El resultado está mal de raíz. | **Descartar** | Descartar es barato y vuelve a un contexto limpio. Iterar sobre un resultado malo arrastra sus supuestos errados. |
| Nadie sabe cómo funciona la parte del sistema afectada. | **Extraer conocimiento o spike** | Primero lectura asistida del código y entrevistas; si no alcanza, un spike acotado (sección 3.3). |

## 3.3 *Dónde sigue habiendo iteración: spikes gobernados*

SDD no elimina la experimentación. La desplaza: se itera libremente en la especificación, en prototipos de interfaz que se muestran al usuario y en spikes técnicos. Lo que no se hace es descubrir el comportamiento de producción construyéndolo. Un spike es legítimo cuando cumple cinco condiciones:

1. Responde una pregunta escrita de antemano, que la spec no puede resolver sin evidencia.
2. Tiene un límite de tiempo declarado, normalmente de horas, no de días.
3. Su entregable es una **decisión** registrada en la spec, no código.
4. El código del spike se descarta. No se promueve a producción aunque funcione.

5. Un validador revisa la decisión resultante con el mismo criterio que cualquier spec.

La condición cuatro es la que más se viola. Un spike promovido a producción introduce un componente que ninguna spec gobierna y que ningún criterio validó, y convierte la excepción en método.

### 3.4 *Supervisión proporcional al riesgo*

No todo artefacto merece el mismo escrutinio. Si los validadores revisan todo con la misma intensidad, se convierten en el cuello de botella; si revisan poco, aprueban por cansancio. La alternativa es graduar la supervisión según el riesgo del artefacto y la confianza que dejan las validaciones automáticas.

**Tabla 5.** Niveles de supervisión.

| Riesgo | Qué incluye | Tratamiento |
|---|---|---|
| **Bajo** | Código repetitivo, transformaciones mecánicas, cambios cubiertos por pruebas en verde. | Avanza, queda registrado y se revisa por muestreo. |
| **Medio** | Implementaciones con lógica de negocio acotada. | Criterio contextual. Specs y planes se revisan siempre; la implementación, según su criticidad. |
| **Alto** | Dominio complejo, seguridad, datos sensibles, trabajo regulado, artefactos de las primeras fases del ciclo o casos en que el agente declaró incertidumbre. | Revisión humana obligatoria. Nada avanza aguas abajo hasta la aprobación. |

Hay una razón empírica para no bajar la guardia en el tramo alto: el **sesgo de complacencia con la automatización**. Un resultado que se ve coherente reduce la atención del revisor, y los modelos están diseñados para sonar seguros aun cuando no lo están. La coherencia no es evidencia de corrección. Por eso la regla de G3 es explícita: nunca se aprueba algo porque "se ve bien".

## 4 Los artefactos

### 4.1 *Estructura mínima del repositorio*

Todos los artefactos viven en el mismo repositorio que el código y se versionan con él. Esto es una condición técnica, no una preferencia: si la spec vive en una wiki y el código en un repositorio, la deriva es inevitable.

```
repositorio/
├── context/                    # lo lee el agente antes de cada tarea
│   ├── principles.md           # reglas no negociables del producto y del equipo
│   ├── tech-policy.md          # tecnologías aprobadas / prohibidas, versiones
│   ├── architecture.md         # capas, módulos, patrones de referencia
│   └── security.md             # qué no se registra, cómo se autentica, datos sensibles
├── specs/
│   └── 042-cupon-checkout/
│       ├── spec.md             # el contrato (G1)
│       ├── plan.md             # el diseño técnico (G2)
│       └── tasks.md            # tareas pequeñas y verificables
├── criteria/                   # reglas de revisión: contra qué valida una persona
│   └── ui-review.md
├── lib/ … test/                # código y pruebas generadas
└── CHANGELOG.md
```

El contexto del proyecto empieza con cuatro archivos breves y crece solo cuando una falla concreta lo justifica. Un contexto de cuarenta páginas escrito antes de la primera corrida es tan inútil como no tenerlo: el agente lo lee entero y prioriza mal. La carpeta `criteria/` resuelve un problema distinto: que dos validadores no aprueben cosas distintas ante el mismo artefacto.

### 4.2 *Anatomía de una spec*

Una spec bien escrita responde seis preguntas. Las primeras cuatro siguen la lógica del diseño por contrato; las dos últimas existen porque el lector es un agente, que no pregunta cuando le falta información.

**Tabla 6.** Secciones obligatorias de una spec.

| Sección | Pregunta que responde | Error típico si falta |
|---|---|---|
| **Precondiciones** | ¿Qué tiene que ser cierto antes de que esto ocurra? | El agente valida a medias o no valida, y aparecen estados imposibles. |
| **Transiciones** | ¿Qué cambia, en qué orden y con qué reglas de cálculo? | Reglas de negocio inferidas: redondeos, prioridades y combinaciones arbitrarias. |
| **Poscondiciones** | ¿Qué tiene que ser cierto después? | El resultado "funciona" en pantalla pero deja el sistema en un estado inconsistente. |
| **Invariantes** | ¿Qué nunca puede dejar de ser cierto, pase lo que pase? | Efectos laterales sobre otras partes del sistema que nadie pidió revisar. |
| **Criterios de aceptación** | ¿Con qué ejemplos concretos se demuestra que está bien? | Verificación por impresión visual: "se ve bien". |
| **Fuera de alcance e incógnitas** | ¿Qué no se hace y qué no se sabe todavía? | El agente completa lo no dicho con lo más probable estadísticamente, no con lo correcto. |

**La prueba de los dos ingenieros.** Una spec está lista cuando dos ingenieros que la leen por separado construirían lo mismo. Si tendrían que ponerse de acuerdo en algo, ese algo falta en la spec. Es la versión SDD de la Definition of Ready, y es el criterio principal del gate G1.

## 4.3 *Ejemplo completo: cupón de descuento en el checkout de una app móvil*

El ejemplo usa una aplicación Flutter, pero la estructura es idéntica en cualquier stack. A la izquierda, la historia de usuario tal como entraría a un sprint. A la derecha, la spec que entra al ciclo SDD.

**HISTORIA DE USUARIO (INSUMO DE SCRUM)**

```
Como comprador
quiero aplicar un cupón de descuento
para pagar menos.

Criterios:
- Puedo ingresar el código en el checkout.
- Se aplica el descuento.
- Veo el nuevo total.
```

En Scrum, esta historia funcionaba porque el desarrollador preguntaba durante el sprint. Un agente no pregunta: decide al menos diez cosas no escritas. Por ejemplo, si se combinan cupones, si el descuento alcanza al envío o a los impuestos, cómo se redondea, qué pasa si el cupón vence durante el checkout, si funciona sin conexión, qué mensaje ve el usuario ante cada error y si el uso se consume al aplicarlo o al pagar.

Cada una de esas decisiones queda implícita en el código y aparece recién en producción, o en la demo, que es lo mismo pero más tarde.

**SPEC (INSUMO DE SDD)**

```
# 042 · Cupón de descuento en checkout
Valor: reducir abandono en el último paso.

## Precondiciones
G1 Carrito con ≥ 1 ítem y sesión activa.
G2 Código: 4 a 16 caracteres [A-Z0-9].
G3 Requiere conexión. Sin red no se valida
   localmente: se muestra E4.

## Transiciones
T1 Validar el cupón contra el servidor.
T2 Descuento sobre el subtotal de productos;
   excluye envío e impuestos.
   Redondeo half-up a 2 decimales.
T3 Un cupón por pedido: el nuevo reemplaza
   al anterior, con aviso al usuario.

## Poscondiciones
P1 El total mostrado coincide con el que
   recalcula el servidor al pagar.
P2 El uso se consume al confirmar el pago,
   no al aplicar el cupón.

## Invariantes
I1 El total nunca es negativo.
I2 El código no se registra en logs.

## Errores
E1 inexistente · E2 vencido · E3 no aplica
a estos productos · E4 sin conexión.

## Criterios de aceptación
AC1 Subtotal 100,00 y cupón 15 %: total 85,00.
AC2 Cupón vencido: E2 y total sin cambio.
AC3 Golden test de la pantalla con cupón
    aplicado, tema claro y oscuro.

## Fuera de alcance
Cupones combinables. Cupones sobre envío.
## Incógnitas
¿Se muestra el ahorro en el resumen? (PO)
```

A partir de la spec, las pruebas se derivan **antes** de generar la implementación. Cada criterio de aceptación se convierte en una prueba con su identificador, de modo que la trazabilidad entre contrato y código es directa:

```
// test/checkout/coupon_test.dart — derivado de specs/042, antes del código
test('AC1: cupón 15% sobre subtotal 100,00 deja total 85,00', () {
  final cart = Cart(subtotal: Money.parse('100.00'), shipping: Money.parse('8.00'));
  final result = applyCoupon(cart, Coupon.percent('PROMO15', 15));
  expect(result.productsTotal, Money.parse('85.00'));   // T2: excluye envío
});

test('AC2: cupón vencido devuelve E2 y no modifica el total', () {
  final cart = Cart(subtotal: Money.parse('100.00'));
  final result = applyCoupon(cart, Coupon.expired('VIEJO10'));
  expect(result.error, CouponError.expired);           // E2
  expect(result.productsTotal, Money.parse('100.00'));
});

testWidgets('AC3: resumen con cupón aplicado (golden)', (tester) async {
  await tester.pumpWidget(CheckoutSummary.fixture(couponApplied: true));
  await expectLater(find.byType(CheckoutSummary),
      matchesGoldenFile('goldens/042_cupon_aplicado_claro.png'));
});
```

El golden test merece una mención aparte. En interfaces generadas por agentes, la revisión visual manual es el punto donde la validación se degrada primero, porque el revisor termina aprobando "lo que se ve razonable". Comparar contra una imagen de referencia aprobada convierte un criterio subjetivo en uno ejecutable. La imagen de referencia se aprueba una vez, en G3, y desde ahí cualquier regresión visual bloquea la integración.

## 4.4 *De lo vago a lo verificable*

**Tabla 7.** Formulaciones que no sirven para un agente y su reemplazo.

| Formulación vaga | Qué hace el agente con ella | Reemplazo verificable |
|---|---|---|
| **"Que sea rápido"** | Elige un objetivo arbitrario y optimiza contra él sin decirlo. | Percentil 95 del tiempo de respuesta bajo un valor, medido en un punto nombrado. |
| **"Manejar los errores"** | Inventa su propia lista de errores y tiende a ocultarlos. | Lista cerrada de errores con código, mensaje al usuario y qué se registra. |
| **"Validar la entrada"** | Valida a medias y deja huecos de seguridad. | Tipos, rangos y comportamiento ante cada violación, campo por campo. |
| **"Seguir las buenas prácticas"** | Trae patrones ajenos al proyecto. | Referencia al archivo del contexto que gobierna ese aspecto. |
| **"Como lo que ya existe"** | Infiere el patrón muestreando el repositorio, con resultados inconsistentes. | El archivo exacto que es la referencia y qué debe ser distinto. |
| **"Responsive"** | Decide los puntos de quiebre. | Anchos de referencia y comportamiento esperado en cada uno, con golden test por ancho. |

La contracara: una spec tiene que **declarar lo que no sabe**. Una spec que oculta su incertidumbre para parecer completa es más peligrosa que una que la declara, porque el agente va a resolver la incógnita en silencio. La sección "Incógnitas" bloquea G1 hasta que cada pregunta tenga dueño y respuesta.

Para equipos que buscan más rigor sintáctico existen plantillas de redacción como EARS (*Easy Approach to Requirements Syntax*; Mavin et al., 2009), que restringen cada requisito a unas pocas formas, por ejemplo "cuando <evento>, el sistema debe <respuesta>". No son obligatorias, pero reducen la ambigüedad y permiten revisar specs de manera automática antes de enviarlas al agente.

## 4.5 *Calibración: cómo aprende el sistema*

La señal de calibración es precisa: **si un validador tiene que dar el mismo feedback dos veces, ya no es feedback, es un problema de calibración**. El arreglo no va al artefacto sino al contexto del proyecto, para que el error no se repita en ninguna tarea futura.

**Tabla 8.** Dónde se corrige cada tipo de falla.

| Tipo de falla | Síntoma | Dónde se corrige |
|---|---|---|
| **Alucinación técnica** | El agente usa una biblioteca o API que no existe o no está aprobada. | `tech-policy.md` |
| **Violación de estilo** | Funciona, pero rompe los patrones acordados. | `architecture.md` o estándares de código |
| **Error de negocio** | Interpreta mal una regla o un invariante. | La spec y, si es transversal, un archivo de dominio en el contexto |
| **Seguridad o cumplimiento** | Secreto en el código, registro de datos sensibles, valor por defecto inseguro. | `security.md` y reglas de `criteria/` |
| **Criterio ambiguo** | Dos validadores aprueban cosas distintas ante el mismo artefacto. | `criteria/` |

**Procedimiento.** (1) Al rechazar un artefacto por una causa repetible, el validador lo marca con una etiqueta de calibración y su categoría. (2) Una vez por semana, el dueño del contexto revisa las etiquetas y prioriza por patrón, no por volumen. (3) Se redacta el cambio en el contexto. (4) **Se prueba el arreglo**: se vuelve a ejecutar en aislamiento la misma tarea que falló, con el contexto actualizado. Si el agente produce el resultado correcto sin guía adicional y el gate pasa limpio, el cambio se integra referenciando las etiquetas. Si no pasa, no se integra, aunque haya presión de plazo.

> *Primero el contexto, después el sistema. Ante una falla, la pregunta es si al agente le falta contexto o si el sistema de trabajo está mal diseñado. En la mayoría de los casos es contexto. Solo se escala a cambios de herramienta, de flujo o de prompts cuando la falla persiste con reglas claras.*

### 4.6 *Trazabilidad*

Cada artefacto lleva identificadores heredados del artefacto del que deriva: un requisito `FR-012` origina la spec `042`, cuyos criterios `AC1..AC3` se convierten en pruebas con el mismo identificador, y cuyas decisiones de diseño quedan como registros de arquitectura (`ADR-007`). La regla es que ningún identificador se inventa aguas abajo: todos se heredan. Así, cuando un requisito cambia semanas después, se sabe qué specs, pruebas y componentes afecta, y se evita mantener pruebas de funcionalidades obsoletas.

## 5 Roles: de Scrum a SDD

SDD no requiere roles nuevos. Requiere redistribuir la autoridad de validación y liberar agenda para ejercerla. Este es el movimiento más subestimado de la transición: no se trata de renombrar puestos, sino de cambiar quién puede decir "esto no avanza" y de asegurar que esa persona tenga tiempo para decirlo.

**Tabla 9.** Redistribución de responsabilidades por rol.

| Rol en Scrum | Deja de hacer | Empieza a hacer | Señal de que quedó a medias |
|---|---|---|---|
| **Product Owner** | Describir funcionalidades en tarjetas. | Declarar resultados, reglas de negocio e invariantes verificables. Valida la parte de negocio en G1 y responde las incógnitas. | El validador técnico tiene que interpretar la intención antes de poder validar. |
| **Scrum Master** | Facilitar impedimentos individuales y cuidar el timebox. | Gobernar el flujo: latencia de validación, retrabajo, carga de los validadores y costo de cómputo. Sostiene la regla "spec antes que código". | Su tablero de métricas se actualiza pero no dispara decisiones. |
| **Desarrollador senior** | Escribir la mayor parte del código. | Validar planes y resultados (G2 y G3), calibrar el contexto y escribir las reglas de `criteria/`. | Edita el código generado en lugar de rechazarlo y corregir la spec. |
| **Desarrollador** | Implementar tareas de punta a punta. | Redactar specs, conducir al agente, derivar pruebas, resolver la integración. | Conversa con el agente hasta que "anda", sin spec de por medio. |
| **QA** | Verificar al final del sprint. | Derivar las pruebas desde la spec antes de que exista código; cuidar los golden tests. | Las pruebas se siguen escribiendo después del código. |
| **Arquitecto o tech lead** | Revisar pull requests uno por uno. | Mantener el contexto del proyecto y diseñar los gates. Revisa patrones, no artefactos. | Entra a desbloquear tareas a mano: cada vez que lo hace, el sistema no aprende. |

**QUÉ DEJAR FIRMADO ANTES DE EMPEZAR**

- Quién valida en cada gate y para qué tipo de cambio, con la frase explícita de que su rechazo no se revierte por plazo.
- Quién es dueño del contexto del proyecto y decide cuándo una falla repetida se convierte en regla.
- Quién valida del lado del cliente o del negocio la spec y los criterios de aceptación, no solo la demo.
- Qué porcentaje de la agenda de cada validador queda liberado de construcción. Sin este dato, los gates se convierten en colas.

# 6 Las ceremonias de Scrum, reconvertidas

Las ceremonias de Scrum se conservan. Lo que cambia es su producto: cada una deja de producir acuerdos verbales y pasa a producir un artefacto versionado. La cadencia de sprint también puede conservarse, pero como **cadencia de gobierno** (cuándo se revisan prioridades, métricas y el sistema de trabajo) y no como unidad de entrega. Lo que pasa los gates se integra en el momento; no espera al cierre del sprint.




**Figura 2.** Cadencia sugerida para un ciclo de gobierno de dos semanas. Las ceremonias marcan el ritmo de decisión; la entrega ocurre de forma continua dentro del flujo.

Las fichas siguientes describen cada ceremonia con la misma estructura: qué propósito conserva, qué cambia, cómo se organiza la agenda, qué artefacto deja y cuál es su anti-patrón más frecuente.

## 6.1 Sprint Planning → Planning de intenciones

*cada 2 semanas · 60 a 90 min*

| | |
|---|---|
| **Propósito que conserva** | Acordar qué valor se persigue en el próximo ciclo y comprometer capacidad de forma realista. |
| **Qué cambia** | No se estima esfuerzo de construcción ni se desglosan tareas técnicas: eso lo hace el agente en la fase de planificación, y se valida en G2. La capacidad que se compromete es la de **validación**, no la de desarrollo. La estimación cambia de objeto: se cuantifica la ambigüedad, la complejidad lógica del dominio y el riesgo de integración. Lo aislado y bien entendido estima bajo; lo que toca seguridad o sistemas heredados estima alto. |
| **Agenda** | (1) Revisión de métricas del ciclo anterior, 10 min. (2) Objetivos de negocio del ciclo, 15 min. (3) Selección de intenciones del backlog que pasan a especificación, con estimación por ambigüedad y riesgo, 30 min. (4) Chequeo de carga de cada validador contra su techo, 10 min. (5) Asignación de dueños de spec, 10 min. |
| **Artefacto que deja** | Lista de specs a redactar en el ciclo, cada una con dueño, validadores asignados y nivel de riesgo. |
| **Anti-patrón** | Comprometer specs según la velocidad del agente. El agente nunca es el cuello de botella; el validador sí. Si nadie queda por encima de su techo de validación, la planificación es realista. |

## 6.2 Refinamiento → Taller de especificación

*2 veces por semana · 60 min*

| | |
|---|---|
| **Propósito que conserva** | Llevar los ítems del backlog a un estado en que se puedan ejecutar sin sorpresas. |
| **Qué cambia** | Pasa a ser la ceremonia central del método. Se trabaja sobre el texto de la spec, con el agente como asistente para detectar ambigüedades y casos borde. Termina con la spec lista para G1 o con preguntas escritas y con dueño. |
| **Agenda** | (1) Lectura de la spec en borrador, 10 min. (2) Pedirle al agente que liste ambigüedades, supuestos y casos borde no cubiertos, 10 min. (3) Resolver o registrar cada uno como incógnita con dueño, 25 min. (4) Prueba de los dos ingenieros, 10 min. (5) Derivar criterios de aceptación, 5 min. |
| **Artefacto que deja** | `spec.md` lista para G1, o bloqueada con incógnitas explícitas. |
| **Anti-patrón** | Aprobar en refinamiento una spec "casi lista" para no frenar el flujo. Una incógnita sin resolver se convierte en una decisión del agente. |

## 6.3 Daily → Daily de calibración

*diaria · 15 min*

| | |
|---|---|
| **Propósito que conserva** | Sincronizar al equipo y detectar bloqueos temprano. |
| **Qué cambia** | La pregunta deja de ser "en qué estoy" y pasa a ser "qué falló ayer en el sistema y qué regla corregimos". Se revisa el estado del flujo, no el de las personas. |
| **Agenda** | (1) Artefactos rechazados ayer y causa: spec, ejecución o contexto, 5 min. (2) Errores repetidos que requieren calibración, 5 min. (3) Desvíos del agente y conflictos con el contexto; trabajo esperando validación por más de un día y quién lo destraba, 5 min. |
| **Artefacto que deja** | Una corrección en el contexto del proyecto, o una tarea de calibración asignada. Si una daily no deja ninguna de las dos durante una semana, el equipo no está aprendiendo o no está registrando. |
| **Anti-patrón** | Volver a la ronda de estado individual. Una tarea abierta más de unas horas sin avance no es progreso: es un agente bloqueado o un contexto faltante. |

## 6.4 Sprint Review → Review de resultados

*cada 2 semanas · 45 min*

| | |
|---|---|
| **Propósito que conserva** | Mostrar a los interesados lo entregado y recoger su opinión. |
| **Qué cambia** | Deja de ser el momento de validación técnica, que ya ocurrió en los gates, y de descubrimiento, que ya ocurrió en la spec. Se muestran resultados de negocio y se toman decisiones sobre las próximas intenciones. |
| **Agenda** | (1) Funcionalidades entregadas en el ciclo, contra la spec que las originó, 20 min. (2) Indicadores de negocio afectados, 10 min. (3) Cambios de opinión del negocio, que se registran como cambios a specs o nuevas intenciones, 15 min. |
| **Artefacto que deja** | Decisiones de negocio registradas como nuevas intenciones o como modificaciones a specs existentes. |
| **Anti-patrón** | Que el cliente vuelva a usar la demo como gate de aprobación. Si eso pasa, el sprint vuelve a ser la unidad de entrega y el método regresa a Scrum. |

## 6.5 Retrospectiva → Retro del sistema

*cada 2 semanas · 60 min*

| | |
|---|---|
| **Propósito que conserva** | Mejorar la forma de trabajar del equipo. |
| **Qué cambia** | Gana peso y foco. Además del clima del equipo, audita el sistema: calidad de las specs, efectividad de los gates, calibraciones del ciclo y carga de los validadores. |
| **Agenda fija** | (1) Tasa de aceptación en primera pasada y su tendencia. (2) Retrabajo y su causa raíz. (3) Carga y latencia de los validadores. (4) Costo de cómputo por funcionalidad validada. (5) Recorrido de la tabla de adopción parcial (sección 10). (6) Una sola acción de mejora con dueño. |
| **Artefacto que deja** | Cambios concretos en el contexto, en `criteria/` o en la configuración de los gates. |
| **Anti-patrón** | Que la retro discuta el método en abstracto en lugar de los datos del ciclo. |

### 6.6 *Los artefactos de Scrum, reconvertidos*

**Tabla 10.** Equivalencias entre artefactos.

| Artefacto de Scrum | Equivalente en SDD | Qué cambia |
|---|---|---|
| **Product Backlog** | Backlog de intenciones + specs | Se separan dos estados: intención (qué valor) y spec (qué comportamiento). Solo las specs aprobadas son ejecutables. |
| **Sprint Backlog** | Specs comprometidas en el ciclo | Se compromete capacidad de validación, no de construcción. |
| **Historia de usuario** | Enunciado de valor dentro de la spec | Deja de ser la orden de trabajo; es la primera línea del contrato. |
| **Definition of Ready** | Gate G1 y prueba de los dos ingenieros | Pasa de checklist verbal a control bloqueante. |
| **Definition of Done** | Gate G3: criterios ejecutables en verde | Pasa de acuerdo del equipo a pruebas que bloquean la integración. |
| **Incremento** | Funcionalidad validada | Se entrega cuando pasa sus gates, sin esperar el fin del sprint. |
| **Burndown** | Tablero de flujo y validación | Muestra dónde espera el trabajo, no cuánto falta construir. |

## 7 Métricas

El cambio de métricas es el que define si la transición es real. Lo que se mide se optimiza, y si se siguen midiendo story points y velocity, el equipo va a optimizar volumen de código generado, que es exactamente lo que no hay que optimizar.

**Tabla 11.** Reemplazo de métricas.

| En Scrum | Por qué deja de servir | Qué se mide en SDD |
|---|---|---|
| **Velocity** | Con generación casi gratis mide volumen, no valor: puede subir mientras la calidad baja. | Funcionalidades validadas y entregadas por ciclo. |
| **Story points** | Miden el esfuerzo de construir, que ya no es lo caro. | Esfuerzo de validación estimado por riesgo y complejidad. |
| **Capacidad en horas** | La construcción no consume horas del equipo; la validación sí. | Capacidad de validación por persona, con un techo sostenible. |
| **Burndown** | El sprint ya no es la unidad de entrega. | Tiempo de espera en cada gate. |

**Tabla 12.** Tablero mínimo de SDD. Los umbrales son orientativos y cada equipo los ajusta con su propia línea base.

| Métrica | Definición | Umbral orientativo | Decisión que dispara |
|---|---|---|---|
| **Aceptación en primera pasada** | Artefactos aprobados en el primer intento / artefactos presentados al gate. | ≥ 80 % | Por debajo: revisar calidad de specs y contexto antes que el agente. |
| **Retrabajo** | Artefactos regenerados o corregidos / artefactos totales. | ≤ 20 % | Por encima: buscar la causa raíz en la daily y calibrar. |
| **Latencia de validación** | Tiempo entre que un artefacto llega a un gate y su decisión. | < 1 día | Alta: falta agenda de validadores. Muy baja en artefactos complejos: posible aprobación sin revisión. |
| **Utilización de validadores** | Carga de validación asignada / capacidad declarada. | ≤ 85 % | Por encima: redistribuir o bajar el compromiso del ciclo. |
| **Tasa de deriva** | Commits que cambian comportamiento sin cambio previo de spec / commits de comportamiento. | 0 % | Cualquier valor mayor: conversación inmediata; es la señal temprana de regreso a Scrum. |
| **Errores repetidos** | Rechazos con la misma causa que un rechazo anterior. | Tendencia a la baja | Estable o en alza: la calibración no está funcionando. |
| **Defectos escapados** | Defectos en producción por funcionalidad entregada. | ≤ línea base | Por encima: los gates aprueban sin criterio suficiente. |
| **Costo por funcionalidad validada** | Costo de cómputo y horas de validación / funcionalidades entregadas. | Tendencia a la baja | Estable: el sistema no está aprendiendo; revisar calibración. |

Un patrón merece atención especial: **velocity subiendo y retrabajo subiendo al mismo tiempo**. Es la firma de un equipo que generó más código sin mejorar sus decisiones. Si el tablero todavía muestra velocity, es el primer dato a mirar.

## 8 Modelo de madurez

Adoptar SDD no es un evento binario. Es un continuo, y lo que define en qué punto está un equipo es una sola variable: **cuánta autoridad tiene la spec sobre el código a lo largo del tiempo**. Böckeler (2025), a partir del análisis de las herramientas que hoy implementan el enfoque, distingue tres niveles: *spec-first*, *spec-anchored* y *spec-as-source*. Este documento adopta esa escala y toma una posición explícita sobre cada nivel.

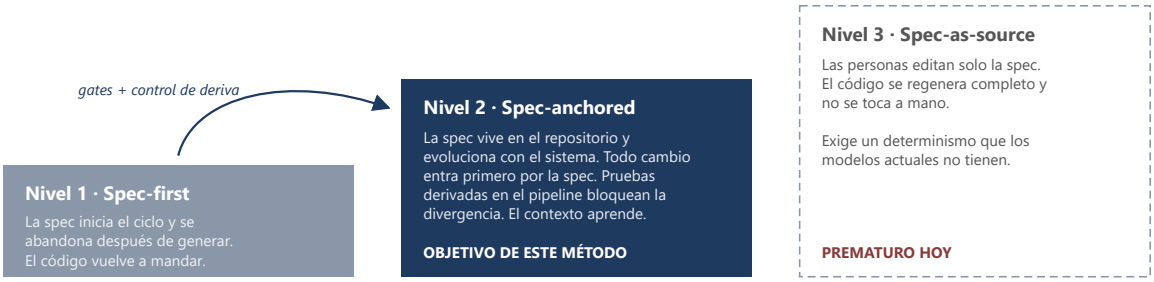



**Figura 3.** Los tres niveles de madurez según la autoridad de la spec sobre el código. El paso del nivel 1 al 2 se construye con dos mecanismos: gates que bloquean y detección automática de deriva.

| **Nivel 1 · Spec-first** | *"La spec como punto de partida" · insuficiente* |
|---|---|
| **Definición** | Toda funcionalidad empieza con una spec escrita antes de que el agente genere código. La spec le da al agente la claridad necesaria para resolver la tarea inmediata con precisión. |
| **Qué la caracteriza** | La naturaleza efímera de la spec. Una vez que el código se genera y se aprueba, la spec pierde protagonismo. Cuando aparece un cambio de requisito o un defecto, se corrige directamente el código y el documento original queda congelado. |
| **Consecuencia** | En pocas semanas la spec es un artefacto arqueológico: refleja la intención original, no el estado del sistema. El código vuelve a ser la fuente de verdad por defecto y la deriva se acumula sin que nadie la mida. |
| **Dónde aparece** | Es el modo de uso más común de las herramientas actuales: specs en ramas temporales que se descartan después de integrar la funcionalidad. También es donde caen los equipos sin disciplina de gates. |
| **Valor** | Real pero acotado: elimina la ambigüedad en la generación inicial. Es la puerta de entrada y el punto de partida obligado de la transición, no un destino. |
| **Prácticas mínimas** | Plantilla de spec con las seis secciones (Tabla 6). Contexto del proyecto con los cuatro archivos mínimos. Pruebas derivadas de los criterios de aceptación. Story points retirados. |

| Nivel 2 · Spec-anchored | *"La spec anclada y viva" · objetivo* |
|---|---|
| **Definición** | La spec no se descarta después de generar: queda anclada al repositorio y evoluciona en paralelo con el software durante toda la vida de la funcionalidad. |
| **La regla que lo define** | Ante cualquier cambio (un requisito nuevo, un defecto, un arreglo en producción, una decisión del negocio a mitad de camino) se actualiza **primero** la spec para reflejar el nuevo estado deseado, y recién entonces se regenera o modifica el código. El código nunca es la fuente de verdad: la spec sí. |
| **Cómo se garantiza** | Las pruebas derivadas de la spec corren en el pipeline de integración continua y funcionan como puente entre intención y realidad técnica: si el código diverge del comportamiento especificado, las pruebas fallan y la integración se bloquea. Un control adicional rechaza commits que cambian comportamiento sin un cambio de spec asociado. |
| **Qué más exige** | Gates G1, G2 y G3 configurados como controles técnicos, no como acuerdos. Validadores nombrados con agenda. Carpeta `criteria/`. Calibración del contexto ante errores repetidos (sección 4.5). Ceremonias reconvertidas. Tablero de la Tabla 12. Trazabilidad de identificadores entre spec, pruebas y código. |
| **Por qué es el objetivo** | Es el equilibrio pragmático entre estructura formal y flexibilidad operativa. Captura casi todo el valor del enfoque (el contrato como fuente de verdad, el aprendizaje acumulado, la auditabilidad) sin apostar a un determinismo que los modelos todavía no ofrecen. Es la opción más robusta para mantener la higiene arquitectónica en el tiempo. |
| **Evidencia observable** | Tasa de deriva en cero. Arreglos de producción que empiezan por la spec. Un integrante nuevo entiende una funcionalidad leyendo su spec, sin leer el código. Costo por funcionalidad validada en descenso sostenido. |

| Nivel 3 · Spec-as-source | *"La spec como código fuente" · prematuro* |
|---|---|
| **Definición** | La spec se convierte en el único archivo fuente válido. Las personas editan exclusivamente specs, en lenguaje natural estructurado o sintaxis declarativa, y nunca modifican ni inspeccionan el código. Cada cambio en la spec dispara una regeneración automática de componentes, interfaces y esquemas. |
| **Cómo se ve** | Existen entornos experimentales en los que cada archivo generado lleva un encabezado que prohíbe editarlo a mano, porque cualquier cambio manual se sobrescribe en la próxima generación. |
| **Por qué no es objetivo** | Primero, supone un determinismo que los modelos no tienen: la misma spec puede producir implementaciones distintas en corridas distintas, y eso obliga a reverificar todo en cada regeneración. Segundo, exige una infraestructura de generación y verificación muy madura antes de tener evidencia de que el método funciona en el equipo. |
| **Posición** | No diseñar la adopción apuntando a spec-as-source "por las dudas". Es optimización prematura sobre infraestructura no validada. El mínimo viable es spec-anchored funcionando de punta a punta en una rebanada real. Se valida lo simple antes de escalar el diseño. |

## 8.1 *Cómo se pasa de spec-first a spec-anchored*

La diferencia entre los dos niveles no está en la calidad de las specs, sino en **qué pasa después de la primera entrega**. Un equipo puede escribir specs excelentes y seguir en el nivel 1 si corrige el código directamente cuando algo cambia. Por eso el paso se construye con mecanismos, no con buenas intenciones:

**Tabla 13.** Mecanismos que convierten spec-first en spec-anchored.

| Mecanismo | Qué evita | Cómo se implementa |
|---|---|---|
| **Gates bloqueantes** | Que una spec "casi lista" o un plan no revisado lleguen a generación. | Protección de rama, revisiones obligatorias por tipo de cambio, pipeline que falla. |
| **Pruebas derivadas en CI** | Que el código diverja del comportamiento especificado sin que nadie lo note. | Cada criterio AC*n* tiene su prueba con el mismo identificador; la integración se bloquea si falla. |
| **Control de deriva** | Arreglos "chicos" directo en el código. | Regla en el pipeline o en la revisión: un commit de comportamiento debe referenciar un cambio de spec. |
| **Circuito rápido para urgencias** | Que la presión de un incidente justifique saltear la spec. | Versión abreviada de G1 (spec mínima del arreglo, un validador) con revisión completa posterior. |

| Mecanismo | Qué evita | Cómo se implementa |
|---|---|---|
| **Reconstrucción de specs heredadas** | Que el código existente quede para siempre fuera del método. | Cuando se toca una funcionalidad sin spec, primero se escribe su spec actual, con lectura asistida del código, y después se especifica el cambio. |
| **Calibración del contexto** | Pagar el mismo error una y otra vez. | Sección 4.5: el arreglo va al contexto, se prueba aislado y se versiona. |

> *La directiva que mantiene al equipo en spec-anchored: si se detecta una anomalía, o el negocio cambia el diseño durante la implementación, se actualiza primero la spec o el contexto. No se parchea el código. Es lo que impide degradar a spec-first con deriva.*

### 8.2 *Autoevaluación*

El cuestionario ubica al equipo. Un equipo está en un nivel cuando responde afirmativamente todo el bloque correspondiente. Un equipo con huecos en el bloque 1 todavía no adoptó SDD: adoptó su vocabulario.

**Tabla 14.** Cuestionario de madurez.

| # | Bloque | Pregunta (sí / no) |
|---|---|---|
| 1 | Spec-first | ¿Toda funcionalidad nueva tiene una spec escrita antes de generar código? |
| 2 | Spec-first | ¿Las specs incluyen precondiciones, transiciones, poscondiciones, criterios, fuera de alcance e incógnitas? |
| 3 | Spec-first | ¿Existe el contexto del proyecto versionado en el repositorio y el agente lo lee en cada tarea? |
| 4 | Spec-first | ¿Se retiraron los story points y la velocity de todos los tableros y reportes? |
| 5 | Spec-anchored | ¿Hay validadores nombrados por gate, con agenda liberada para validar? |
| 6 | Spec-anchored | ¿Es técnicamente imposible integrar sin pasar G3? |
| 7 | Spec-anchored | ¿Las pruebas se derivan de la spec antes de que exista el código y corren en CI? |
| 8 | Spec-anchored | ¿Los arreglos de producción y el mantenimiento también empiezan por la spec? |
| 9 | Spec-anchored | ¿El pipeline o la revisión detectan cambios de comportamiento sin cambio de spec? |
| 10 | Spec-anchored | ¿Cada error repetido terminó en una regla del contexto, probada antes de integrarse? |
| 11 | Spec-anchored | ¿Cada ceremonia produce su artefacto y el tablero de métricas dispara decisiones? |
| 12 | Spec-anchored | ¿Un integrante nuevo puede entender una funcionalidad leyendo solo su spec? |

## 9 Plan de transición

La transición no es un cambio de proceso que se anuncia. Es una secuencia de etapas con criterios de salida verificables, ejecutada sobre un **piloto acotado** (un equipo y una línea de producto) antes de extenderse. La secuencia tiene cuatro etapas: preparación, arranque, calibración y operación estable. A lo largo de ellas se completan seis movimientos de cambio organizacional.

### 9.1 *Etapa 0 · Preparación: condiciones de arranque*

El piloto no empieza hasta que las cuatro condiciones siguientes se cumplen. Pueden trabajarse en paralelo, pero el cumplimiento parcial no habilita el arranque: cada condición faltante reaparece después como una falla más cara.

**Tabla 15.** Condiciones de arranque.

| Condición | Qué se verifica | Si falla |
|---|---|---|
| **1. Compromiso de roles** | Validadores nombrados por gate, dueño del contexto y validador de negocio, con autoridad aceptada por escrito. "Nombrado" significa una persona concreta con autoridad para decir "esto no es válido". | No se empieza. Un gate sin validador real es un trámite. |
| **2. Capacidad de validación** | Que las personas nombradas tengan agenda real para validar al ritmo previsto. Es el cuello de botella habitual. La capacidad se declara como supuesto y se mide en el arranque. | Se reduce el alcance del piloto, no la validación. |

| Condición | Qué se verifica | Si falla |
|---|---|---|
| **3. Herramientas y entorno** | Agente de codificación, repositorio, pipeline de CI y entorno de pruebas funcionando de punta a punta. Cualquier integrante puede ejecutar el ciclo sin esperar permisos. El acceso del agente a modelos pasa por un canal aprobado, sin claves personales. | El arranque se convierte en una semana de configuración. |
| **4. Configuración del método** | Estructura de repositorio creada (sección 4.1), contexto mínimo versionado, plantilla de spec y checklist de G1 disponibles, protección de rama configurada. | Las primeras specs se escriben sin estándar y hay que rehacerlas. |

En esta misma etapa se mide la **línea base** del equipo con Scrum: tiempo de ciclo, defectos escapados y retrabajo. Sin línea base, la mejora no se puede demostrar y la conversación con el negocio queda en el terreno de las percepciones.

## 9.2 *Etapa 1 · Arranque: una rebanada vertical*

Se elige la pieza más chica de alcance real que cumpla tres condiciones: recorre todas las capas del sistema (interfaz, lógica, datos), activa los tres gates y produce artefactos medibles. El arranque no es una demo: es un **diagnóstico**. Su objetivo es comprobar que el sistema de trabajo funciona y medir los supuestos con los que se planificó.

Salidas obligatorias: el ciclo completo ejecutado de punta a punta; la primera medición de aceptación en primera pasada, retrabajo, latencia de validación y costo de cómputo; un backlog de calibración abierto con las primeras fallas registradas; y la capacidad de validación medida, que reemplaza a la supuesta.

## 9.3 *Etapa 2 · Calibración: estabilizar el sistema*

Durante cuatro a seis semanas el equipo escala a tres o cuatro specs en paralelo y cada ciclo sigue la secuencia planificar, ejecutar, validar, medir y calibrar. En esta etapa se instalan los mecanismos de la Tabla 13 y el equipo pasa de spec-first a spec-anchored. La calibración (sección 4.5) es la actividad principal: cada falla repetida se convierte en una regla del contexto, probada antes de integrarse.

Criterios de salida: el equipo opera el ciclo sin acompañamiento externo; los validadores aplican los criterios de manera consistente entre sí; la capacidad de validación se mantiene estable entre ciclos; no hay deriva de costo ni de calidad; y los caminos para escalar a otros equipos están definidos.

## 9.4 *Etapa 3 · Operación estable*

Cada requerimiento nuevo entra como un ítem trazable y recorre el ciclo de la Figura 1. La daily es de calibración desde el primer día, la retro audita el sistema y el tablero de la Tabla 12 dispara decisiones. Una señal operativa simple: una tarea que lleva más de unas horas "en curso" es una bandera roja. El trabajo en SDD fluye; cuando se estanca, casi siempre hay un agente bloqueado o un contexto faltante.

## 9.5 *Los seis movimientos de cambio*

En paralelo con las etapas se completan seis movimientos, cada uno con dueño y criterio de cierre. Si alguno queda a medias, el resultado es una adopción parcial, que falla peor que no transicionar (sección 10).

**Tabla 16.** Los seis movimientos.

| # | Movimiento | Qué implica | Criterio de cierre |
|---|---|---|---|
| 1 | **Reasignar la autoridad de validación** | Nombrar por escrito quién puede decir "esto no avanza" en cada gate y liberar su agenda de construcción. | Validadores nombrados, con veto y con porcentaje de agenda asignado. |
| 2 | **Convertir el backlog de a poco** | Clasificar por destino: lo que entra pronto, lo que es intención sin definir y lo que ya no aplica. Convertir a spec solo lo primero. Si no hay backlog suficiente, hacer primero la definición de producto (brief, PRD, épicas). | Primeras specs aprobadas en G1; el resto queda como intenciones. |
| 3 | **Escribir el contexto mínimo** | Los cuatro archivos del contexto, breves, más lo que la rebanada del arranque realmente toque. El resto llega por calibración. | Archivos versionados y revisados por una persona. |
| 4 | **Cambiar las métricas** | Retirar story points y velocity de todos los tableros y reportes el mismo día. Publicar el tablero de la Tabla 12. | Cero story points; tablero nuevo en uso. |
| 5 | **Convertir las ceremonias** | Aplicar las fichas de la sección 6. No se acortan: cambian de producto. | Cada ceremonia deja su artefacto durante dos ciclos seguidos. |

| # | Movimiento | Qué implica | Criterio de cierre |
|---|---|---|---|
| 6 | **Preparar al cliente o al negocio** | Explicar que las primeras semanas son de calibración y sumar a alguien del negocio a la validación de la spec, no solo de la demo. | Validador de negocio nombrado y curva de calibración aceptada. |

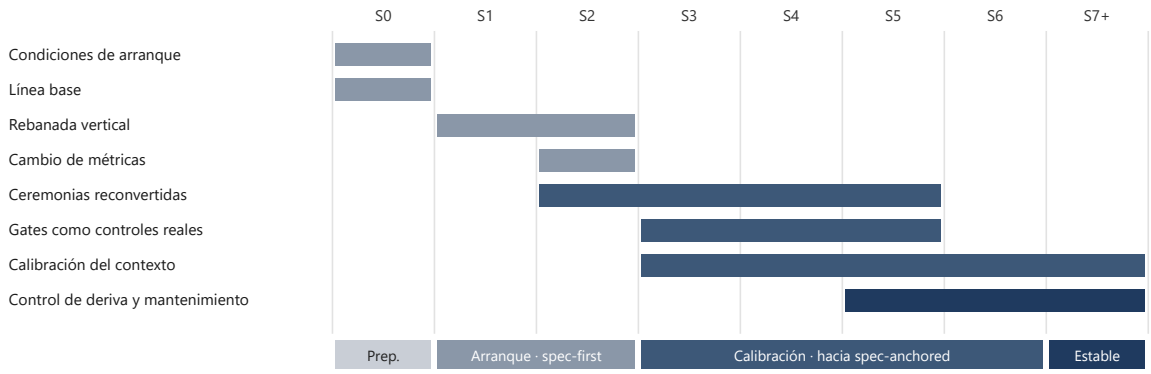



**Figura 4.** Cronograma orientativo para un piloto. La banda inferior muestra la etapa y el nivel de madurez esperado. Spec-anchored se consolida cuando llega el primer ciclo de mantenimiento y los arreglos también entran por la spec.

> *Una spec, una corrida, calibración, y recién después volumen. Convertir el backlog completo a specs antes de la primera corrida es el error de arranque más común: se escriben veinte specs con los mismos defectos que la primera corrida habría revelado.*

## 10 Riesgos: la adopción parcial

El modo de falla más frecuente no es rechazar SDD: es adoptarlo a medias. Cada fila de la tabla siguiente describe una combinación que parece una transición pero no lo es, junto con la señal que permite detectarla. Se recomienda recorrerla en cada retrospectiva durante los primeros tres meses.

**Tabla 17.** Combinaciones de adopción parcial.

| Qué se hizo a medias | Qué se rompe | Cómo se detecta |
|---|---|---|
| **Specs sí, gates bloqueantes no** | Nadie lee la spec. En dos semanas el código vuelve a ser la fuente de verdad. | Commits que corrigen comportamiento sin cambio previo de spec. |
| **Gates sí, autoridad real no** | El validador aprueba por disponibilidad, no por criterio. El riesgo se transfiere con firma. | Latencia de validación muy baja en artefactos complejos. |
| **Contexto sí, calibración no** | El mismo error se paga una y otra vez; el costo por funcionalidad no baja. | Mismo motivo de rechazo repetido en el ciclo. |
| **Specs sí, story points también** | Se optimiza volumen y no validación. | Velocity y retrabajo subiendo al mismo tiempo. |
| **Specs sí, sprint como unidad de entrega** | El sprint se vuelve una cola que retiene trabajo terminado. | Funcionalidades aprobadas esperando el cierre para integrarse. |
| **Specs sí, criterios de revisión no** | Validación por intuición: dos validadores aprueban cosas distintas. | Desacuerdos entre validadores sobre el mismo tipo de artefacto. |
| **Excepción "para urgencias"** | La excepción se vuelve el camino por defecto en el primer trimestre difícil. | Conversaciones largas con el agente sin spec; arreglos directos en producción. |
| **Equipo transicionado, negocio no** | La demo vuelve a ser el gate de facto y con ella el sprint. | Pedidos de demo fuera del flujo; conversaciones sobre fechas de sprint. |
| **Roles renombrados, agendas no liberadas** | El validador lo es de título y constructor de hecho. Los gates se vuelven colas. | Utilización de validadores por encima del techo; latencia en alza. |
| **Spec-as-source de entrada** | Sobre-ingeniería de herramientas antes de validar el método. | Semanas invertidas en tooling sin una funcionalidad de punta a punta. |

### 10.1 *Riesgos operativos*

Además de los riesgos de adopción, el trabajo con agentes trae riesgos técnicos propios. Ninguno se resuelve con disciplina individual: todos requieren una regla escrita.

**Tabla 18.** Riesgos operativos y mitigación.

| Riesgo | Qué pasa | Mitigación |
|---|---|---|
| **Vibe coding** | Una instrucción vaga produce código que compila y parece competente, pero inventa patrones: espaciados distintos en componentes vecinos, colores fuera de la marca, dependencias ajenas al stack. La velocidad es transitoria; la deuda, persistente. | Anclar la intención en una spec restrictiva y en el contexto. G2 lo bloquea. |
| **Saturación del contexto** | Las sesiones largas llenan la ventana de contexto del modelo. La compactación interrumpe el trabajo y produce "amnesia": el agente olvida restricciones, incluidas las de seguridad. | Sesiones cortas por tarea; compactar a propósito antes del límite; dejar las decisiones en archivos, no en la conversación. |
| **Seguridad tardía** | Cambiar autenticación, credenciales o accesos cuando la lógica generada ya es voluminosa obliga a reconstruir partes enteras. | `security.md` desde el primer día; seguridad en las primeras specs. |
| **Costo de inferencia** | Modelos pesados en sesiones largas agotan el presupuesto. | Modelo de razonamiento para specs y planes críticos; modelo liviano para implementación estructural. El costo por funcionalidad validada es una señal de salud. |
| **Permisos del agente** | Un agente con edición directa de archivos y permisos amplios tiene un radio de impacto enorme. | Permisos progresivos y auditables. Nunca desactivar las confirmaciones al empezar; ampliar solo con precisión demostrada. |
| **Dependencia de personas clave** | El flujo funciona porque alguien lo tiene en la cabeza; si se va, no se replica. | Método codificado y versionado: plantillas, gates, contexto y criterios. Lo que no está escrito no existe. |
| **Fatiga documental** | Specs exhaustivas demasiado temprano, con rendimiento decreciente. | La retro ajusta el nivel de detalle: la spec tiene que alcanzar para que dos ingenieros construyan lo mismo, no más. |

## 11 Conclusión

SDD no reemplaza la agilidad: la desplaza al lugar donde hoy es barata. Un equipo que viene de Scrum ya tiene la mayor parte de lo que necesita: cadencia, ceremonias, cultura de entrega incremental y hábito de inspección y adaptación. Lo que cambia es el objeto sobre el que itera (la spec), el mecanismo que garantiza los acuerdos (los gates) y lo que mide (la validación). El modelo de madurez sirve para que esa transición sea observable: en spec-first el equipo escribe antes de generar; en spec-anchored la spec es la fuente de verdad viva del sistema y todo cambio entra por ella; spec-as-source queda como horizonte, no como objetivo.

El indicador de éxito no es la velocidad percibida ni el volumen de código generado.

> *"El indicador de éxito es la ausencia de sobresaltos. Si el equipo se percibe calmo, predecible y ligeramente aburrido, el método está funcionando como fue diseñado."*

# A Anexo: plantilla de spec

```
# <id> · <nombre de la funcionalidad>
Valor: <historia de usuario o resultado de negocio en una línea>
Dueño: <nombre> · Validadores: negocio <nombre>, técnico <nombre> · Riesgo: bajo | medio | alto

## Contexto          qué existe hoy y qué parte del sistema se toca
## Precondiciones    G1..Gn  qué tiene que ser cierto antes
## Transiciones      T1..Tn  qué cambia, en qué orden, con qué reglas de cálculo
## Poscondiciones    P1..Pn  qué tiene que ser cierto después
## Invariantes       I1..In  qué nunca puede dejar de ser cierto
## Errores           E1..En  código · mensaje al usuario · qué se registra
## Criterios de aceptación  AC1..ACn  ejemplos concretos, cada uno con su prueba
## Requisitos no funcionales  rendimiento, accesibilidad, seguridad (medibles)
## Fuera de alcance  lo que explícitamente no se hace
## Incógnitas        pregunta · dueño · fecha límite  (bloquea G1 hasta resolverse)
## Historial         fecha · cambio · motivo  (obligatorio en spec-anchored)
```

# B Anexo: checklist del gate G1

| ✓ | Condición |
|---|---|
| ☐ | El enunciado de valor dice qué resultado de negocio se busca, no qué pantalla se construye. |
| ☐ | Cada transición tiene sus reglas de cálculo explícitas (redondeos, prioridades, orden). |
| ☐ | Cada error tiene código, mensaje al usuario y comportamiento de registro. |
| ☐ | Cada criterio de aceptación es un ejemplo concreto con valores, no una descripción. |
| ☐ | Lo que no se hace está escrito en "Fuera de alcance". |
| ☐ | No quedan incógnitas sin dueño ni respuesta. |
| ☐ | La spec no contradice ningún archivo del contexto del proyecto. |
| ☐ | Dos personas que la leyeron por separado describen la misma implementación. |

## Referencias


Böckeler, B. (2025). Understanding Spec-Driven Development: Kiro, spec-kit, and Tessl. *martinfowler.com*.

Boehm, B. W. (1981). *Software Engineering Economics*. Prentice Hall.

Brooker, M. (2026). Spec Driven Development isn't Waterfall. *brooker.co.za*.

GitHub (2025). *Spec Kit: toolkit for Spec-Driven Development*. github.com/github/spec-kit.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 576–580.

Mavin, A., Wilkinson, P., Harwood, A. y Novak, M. (2009). Easy Approach to Requirements Syntax (EARS). *17th IEEE International Requirements Engineering Conference*, 317–322.

Meyer, B. (1992). Applying "Design by Contract". *IEEE Computer*, 25(10), 40–51.

North, D. (2006). Introducing BDD. *Better Software*.

Schwaber, K. y Sutherland, J. (2020). *The Scrum Guide*. scrumguides.org.