

MANUAL PRÁCTICO · SPEC-DRIVEN DEVELOPMENT

OpenSpec desde cero

Edición imprimible de la web: todo el contenido en un solo documento, en el mismo orden que el menú lateral. La versión en línea siempre es la más reciente.

explore › **propose** › `revisión` › **apply** › **verify** › **archive**

OpenSpec: desarrollo guiado por especificaciones con IA

Primero se acuerda la spec; después el agente implementa contra ella.

1 Fundamentos

- 🔗 Humanos y agentes de IA acuerdan qué construir antes de escribir código, y ese acuerdo vive en Git.
- 📁 `openspec/specs/` Fuente de verdad: cómo se comporta el sistema hoy, por dominios.
- ⚠️ `openspec/changes/` Cada cambio es un delta: solo describe lo que añade, modifica o elimina.
- 📋 Brownfield primero: se introduce en un proyecto existente con un piloto acotado, sin documentarlo todo de golpe.
- 🔧 `openspec init --tools` Agnóstico de herramienta: admite más de 40 asistentes.

Amplía: [Qué es OpenSpec](#) · [Proyectos existentes](#)

3 El ciclo diario con /opsx

- 🔍 `/opsx:explore`
Investiga el problema sin escribir artefactos ni código.
- ➕ `/opsx:propose`
Crea el change con proposal, specs, design y tasks.
- 🔧 `/opsx:apply`
Implementa las tareas y las marca al completarlas.
- ✅ `/opsx:verify`
Comprueba que el código cumple specs, design y tasks.
[perfil ampliado](#)
- 📁 `/opsx:archive`
Fusiona las delta specs y archiva el change.

Amplía: [Flujo /opsx](#) · [Comandos de chat](#)

2 Los cuatro artefactos de un change



Amplía: [Conceptos](#) · [Formato de specs](#)

4 En equipo

- 📋 Backlog en Jira o Azure DevOps: el agente lee el ticket vía MCP, en solo lectura.
- 🛡️ `openspec validate --all --strict` En la pipeline de cada PR, con una versión fija de OpenSpec.
- 🔒 `openspec/specs/` Regla de oro: no se edita a mano; solo cambia con `/opsx:archive` o `/opsx:sync`.

Amplía: [Jira](#) · [Azure DevOps](#) · [Validación en CI](#) · [Adopción](#)

Cómo usar este manual

Una ruta práctica para pasar de no haber usado nunca OpenSpec a trabajar con él a diario, con tu asistente de IA y el backlog de tu equipo.

A quién va dirigido

A desarrolladores y responsables técnicos que ya trabajan con asistentes de IA y quieren que el agente implemente a partir de especificaciones revisadas, no de prompts sueltos. No se asume ningún conocimiento previo de OpenSpec.

Ruta recomendada

Léelo de principio a fin la primera vez. Después, los anexos sirven como consulta rápida.

Momento	Qué hacer	Capítulos
Día 1	Entender el modelo mental: specs, changes y deltas.	01 , 02
Día 2	Instalar OpenSpec, ejecutar openspec init en un repositorio de pruebas y completar un primer cambio.	03 , 04
Día 3	Aprender a escribir specs, recorrer el flujo /opsx y las recetas por tipo de cambio.	05 , 06 , 07
Día 4	Redactar config.yaml y las instrucciones del agente para un proyecto real del equipo, y elegir las skills que lo acompañan.	08 , 09 , 10 , 12
Día 5	Primer change real y pequeño: un bugfix o una funcionalidad acotada.	07
Sprints siguientes	Conectar el backlog y la pipeline, fijar el gobierno del cambio y extender el método al equipo.	13 , 14 , 15 , 16 , 17

VERSIÓN DE REFERENCIA

El manual está escrito sobre OpenSpec v1.14.0. Las versiones anteriores a la 1.x usaban comandos y ficheros distintos; ante cualquier discrepancia, manda la documentación oficial de la versión que tengas instalada.

Convenciones del documento

- `openspec ...` son comandos de terminal (CLI).
- `/opsx:...` son comandos de chat dentro de tu asistente de IA.
- Los comandos aparecen en PowerShell y en bash, uno tras otro.
- El ejemplo que recorre el manual es un sistema neutro de gestión de reservas, sin datos personales.

Índice

EMPIEZA

00 Cómo usar este manual

Ruta recomendada por días y a quién va dirigido

01 ¿Qué es OpenSpec?

OpenSpec es un framework ligero de spec-driven development: humanos y agentes de IA acuerdan qué construir antes de escribir código, y ese acuerdo vive en Git.

02 Conceptos: specs, changes y deltas

Los tres conceptos de OpenSpec: specs como fuente de verdad, changes como unidad de trabajo y deltas ADDED, MODIFIED y REMOVED, con ejemplos de reservas.

03 Instalación

Instala OpenSpec 1.14.0, inicializa un repositorio con una herramienta de IA, entiende la estructura generada y conoce los comandos de terminal esenciales.

04 Tu primer cambio en 30 minutos

Tutorial práctico sobre un repositorio vacío: `openspec init`, `/opsx:propose`, `/opsx:apply` y `/opsx:archive` con un único requisito de gestión de reservas.

GUÍAS

05 Escribir specs

Cómo se escribe una spec principal y una delta spec (ADDED, MODIFIED, REMOVED) en OpenSpec, con buenas prácticas y una checklist de revisión.

06 Flujo `/opsx` de principio a fin

Los comandos `/opsx` de los perfiles core y ampliado, el flujo completo de un change hasta archivarlo en la PR y qué revisar en cada punto de control.

07 Recetas: feature, bugfix, refactor, paralelo

Siete recetas para combinar `/opsx` según el trabajo: funcionalidad pequeña, bugfix, exploración, paralelo, refactor, control incremental y aprendizaje.

08 Proyectos existentes (brownfield)

Cómo introducir OpenSpec en un proyecto existente: sin documentar todo de golpe, con un piloto acotado, cambios pequeños y los riesgos habituales mitigados.

09 config.yaml y schemas personalizados

Cómo ajustar OpenSpec a tu proyecto con openspec/config.yaml (schema, context, rules y operations) y cómo crear schemas de flujo personalizados.

CON TU AGENTE

10 Claude Code

Cómo configurar Claude Code para trabajar con OpenSpec: estructura versionada, permisos, hook de validación, modelos por fase y subagente verificador.

11 Otros agentes (Copilot, Cursor, Codex...)

OpenSpec es agnóstico de herramienta: qué agentes soporta openspec init 1.14, qué se mantiene igual que con Claude Code y qué cambia.

12 Skills complementarias

Qué skills de terceros conviene enchufar alrededor de /opsx:* (interrogar el ticket, TDD, diagnóstico de bugs, revisión, retro), cuáles no y cómo instalarlas.

EN EQUIPO

13 Integración con Jira

Cómo conectar OpenSpec con Jira sin duplicar información: mapeo de conceptos, convención de nombres, servidor MCP de solo lectura y estados por automatización.

14 Integración con Azure DevOps

Cómo integrar OpenSpec con Azure Boards y Azure Repos: convención AB#, servidor MCP local, validación en la branch policy y estados por automatización.

15 Pipeline y validación en CI

Cómo validar las specs en integración continua con Azure Pipelines o GitHub Actions, con la versión de OpenSpec fijada y una regla opcional que exija change.

16 Gobierno del cambio

Cuándo una spec manda de verdad: espectro de autoridad, verja frente a detección temprana, reparto de papeles, cierre calculado y lo que OpenSpec deja fuera.

17 Plan de adopción y riesgos

Plan por fases para adoptar OpenSpec en un equipo: elección del piloto, definición de hecho, métricas, riesgos con su mitigación y checklist de arranque.

REFERENCIA

A Comandos de chat /opsx:*

Referencia de los comandos de chat /opsx:* que genera openspec init en OpenSpec 1.14: qué hace cada uno, cuándo usarlo y qué workflows añade el perfil ampliado.

B CLI openspec

Referencia de la CLI openspec 1.14.0 extraída de su ayuda: instalación, comandos, subcomandos y flags más usados, con variantes para PowerShell y bash.

C Plantillas

Plantillas para copiar: CLAUDE.md, openspec/config.yaml, /ticket-propose, el subagente spec-verifier, la validación en CI y las plantillas de PR y ticket.

D Glosario

Glosario de OpenSpec y del desarrollo guiado por specs: spec, change, delta spec, archive, spec-anchored, verja, MCP y más, cada uno con ancla estable.

E FAQ

Preguntas frecuentes sobre OpenSpec 1.14: specs y changes, cuándo archivar, bugs, validación y verja, skills, Jira y Azure DevOps, idioma y telemetría.

RECURSOS

F Documentación, vídeos y comunidad

Recursos externos para aprender OpenSpec: documentación oficial, artículos de la comunidad y vídeos, con su tipo e idioma y un filtro por tipo.

CÓMO SE HIZO

G Esta web, construida con OpenSpec

Historial real de esta web: los changes de OpenSpec archivados y los dominios de spec actuales, generados desde el repositorio.

Colofón

Ficha de la edición y fuentes

¿Qué es OpenSpec?

OpenSpec es un framework ligero de spec-driven development: humanos y agentes de IA acuerdan qué construir antes de escribir código, y ese acuerdo vive en Git.

OpenSpec es un framework de código abierto (licencia MIT, desarrollado por Fission-AI) para hacer **spec-driven development** con asistentes de IA de programación. Su objetivo es concreto: que personas y agentes se pongan de acuerdo sobre **qué** hay que construir antes de escribir código, y que ese acuerdo quede versionado en el repositorio.

El problema que resuelve

Sin especificaciones, el trabajo con agentes se apoya en prompts vagos. El agente rellena huecos con suposiciones, el conocimiento se pierde en el historial del chat y nadie puede revisar la intención antes de ver el código.

OpenSpec introduce un paso intermedio y revisable: el agente redacta una **propuesta de cambio** (qué, por qué, cómo y lista de tareas) que el equipo revisa y aprueba **antes** de implementar.

Principios

1. **Las specs son la fuente de verdad.** `openspec/specs/` describe cómo se comporta el sistema hoy, organizado por dominios.
2. **Los cambios son deltas.** Cada cambio vive en su carpeta dentro de `openspec/changes/` y solo describe lo que añade, modifica o elimina.
3. **Brownfield primero.** No hay que documentar todo el sistema para empezar: solo se escriben specs de lo que se va a tocar. La documentación crece con el trabajo real.
4. **Ligero e iterativo.** Sin fases rígidas; los artefactos se pueden editar en cualquier momento.

5. **Agnóstico de herramienta.** Funciona con Claude Code, GitHub Copilot, Cursor, Codex, Gemini CLI, OpenCode, Windsurf y muchos otros.
6. **Todo en Git.** Specs y cambios se versionan junto al código y se revisan en las PR.

Cómo funciona

```
openspec/  
├─ config.yaml          # contexto y reglas para la IA  
├─ specs/<dominio>/     # fuente de verdad: comportamiento actual  
└─ changes/<cambio>/    # cambio en curso  
    ├─ proposal.md      # por qué y alcance  
    ├─ specs/           # delta specs (ADDED / MODIFIED / REMOVED)  
    ├─ design.md        # decisiones técnicas  
    └─ tasks.md         # checklist de implementación
```

El ciclo básico se maneja con comandos en el chat del agente:

`/opsx:propose` → *revisión humana* → `/opsx:apply` → `/opsx:archive`

Al archivar, las delta specs se fusionan en `openspec/specs/` y el cambio pasa a `openspec/changes/archive/`. Si hay incertidumbre previa, `/opsx:explore` sirve para investigar sin escribir nada.

NOVEDAD RESPECTO A VERSIONES ANTERIORES

`/opsx:update` (experimental) revisa los artefactos de planificación de un cambio y los mantiene coherentes entre sí, sin tocar código. Es útil tras recibir feedback en la revisión de la propuesta.

Comparativa con alternativas

Enfoque	Puntos fuertes	Limitaciones
OpenSpec: ligero, deltas, brownfield primero	Agnóstico de herramienta y modelo; poca ceremonia; encaja con código existente	Menos guiado que Spec Kit en proyectos desde cero
GitHub Spec Kit: fases rígidas	Muy exhaustivo	Pesado; mucha configuración; orientado a greenfield
Kiro (AWS): IDE con specs integradas	Experiencia integrada	Atado a su IDE y a sus modelos
Sin specs: prompts directos	Rápido para prototipos	Resultados impredecibles; sin trazabilidad

Qué no es OpenSpec

- **No es un generador de código.** El código lo escribe el agente que uses.
- **No sustituye al backlog.** El backlog guarda el qué de negocio y el estado; OpenSpec guarda el comportamiento del sistema y el detalle técnico del cambio.
- **No sustituye a los tests.** Los escenarios de las specs deben acabar traducidos en tests automatizados.

Siguiente paso

[Conceptos: specs, changes y deltas](#)

Conceptos: specs, changes y deltas

Los tres conceptos de OpenSpec: specs como fuente de verdad, changes como unidad de trabajo y deltas ADDED, MODIFIED y REMOVED, con ejemplos de reservas.

OpenSpec se reduce a tres conceptos: **specs** (cómo se comporta el sistema hoy), **changes** (una modificación en curso) y **deltas** (lo que un change altera de las specs). Los ejemplos de esta página usan un dominio neutro: la gestión de reservas.

Specs: la fuente de verdad

Las specs viven en `openspec/specs/<dominio>/spec.md` y se organizan **por dominio funcional** (reservas, pagos), no por capas técnicas. Describen comportamiento observable, no implementación: nada de clases o tablas, salvo que formen parte del contrato.

- Cada **Requirement** declara un comportamiento con lenguaje normativo (SHALL / MUST).
- Cada requisito tiene uno o más **Scenario** en formato GIVEN / WHEN / THEN.

```
# Reservas
```

```
### Requirements
```

```
#### Requirement: Alta de reserva
```

The system SHALL allow a customer to create a reservation for an available time slot.

```
##### Scenario: Alta correcta
```

- GIVEN an available time slot
- WHEN the customer submits a reservation for that slot
- THEN the system confirms the reservation
- AND the slot is no longer available

```
##### Scenario: Franja no disponible
```

- GIVEN a time slot already reserved
- WHEN the customer submits a reservation for that slot
- THEN the system rejects it with code `'SLOT_NOT_AVAILABLE'`

Las palabras normativas y las palabras clave de los escenarios van en inglés para que el validador y el agente las reconozcan. Los nombres de requisito y de escenario y los términos de dominio pueden ir en tu idioma, siempre con un criterio uniforme en todo el repositorio.

Changes: la unidad de trabajo

Un change es una carpeta en `openspec/changes/<nombre>/` con todo lo necesario para proponer, revisar e implementar una modificación:

Artefacto	Contenido	Quién lo revisa
<code>proposal.md</code>	Intención, alcance y enfoque	Producto / responsable técnico
<code>specs/</code>	Delta specs con escenarios	Todo el equipo (es la parte clave de la revisión)
<code>design.md</code>	Decisiones técnicas	Responsable técnico

Artefacto	Contenido	Quién lo revisa
tasks.md	Checklist de implementación con progreso	Quien implementa

Varios changes pueden convivir en paralelo sin pisarse, porque cada uno vive en su carpeta. Cuando se completa, /opsx:archive lo mueve a openspec/changes/archive/.

Deltas: lo que cambia

Dentro de un change, las specs no se reescriben enteras: solo se declara qué se añade, qué se modifica y qué se elimina, en openspec/changes/<nombre>/specs/<dominio>/spec.md.

ADDED

Un requisito nuevo.

```
### ADDED Requirements
```

```
#### Requirement: Cancelación de reserva
```

```
The system SHALL allow a customer to cancel a reservation up to 24 hours before its start.
```

```
#### Scenario: Cancelación a tiempo
```

- GIVEN a reservation starting in more than 24 hours
- WHEN the customer cancels it
- THEN the reservation is marked as cancelled
- AND the slot becomes available again

MODIFIED

Un requisito existente cuyo comportamiento cambia. Se reescribe **completo**, no solo la diferencia. Es la forma natural de documentar un bug: se describe el comportamiento correcto, no «arreglar X».

```
### MODIFIED Requirements
```

```
### Requirement: Alta de reserva
```

The system SHALL allow a customer to create a reservation for an available time slot **starting in the future**.

```
### Scenario: Franja en el pasado
```

- GIVEN a time slot that already started
- WHEN the customer submits a reservation for that slot
- THEN the system rejects it with code `'SLOT_IN_THE_PAST'`

REMOVED

Un comportamiento que desaparece. Exige motivo y migración.

```
### REMOVED Requirements
```

```
### Requirement: Reserva por correo electrónico
```

Reason: sustituido por el formulario web.

Migration: las reservas pendientes por correo se migran manualmente antes del despliegue.

Del delta a la spec principal

Al ejecutar `/opsx:archive`, OpenSpec fusiona los deltas en `openspec/specs/` y archiva el change. Así la fuente de verdad solo cambia a través de un change revisado.

Un refactor que no altera el comportamiento no lleva delta: se archiva con `openspec archive <cambio> --skip-specs`.

Reglas prácticas

1. **Un escenario por regla de negocio y por caso de error.** Los errores también son comportamiento.

2. **Códigos de error estables** en lugar de mensajes literales: los clientes y los tests dependen de ellos.
3. **Cada escenario, un test automatizado.** La spec sin test es un deseo.
4. **No hagas backfilling masivo.** Las specs crecen change a change.

Siguiente paso

[Instalación](#)

Instalación

Instala OpenSpec 1.14.0, inicializa un repositorio con una herramienta de IA, entiende la estructura generada y conoce los comandos de terminal esenciales.

Requisitos

- **Node.js 20.19.0 o superior** (recomendado una LTS reciente).
- **Git**.
- Un **asistente de IA compatible** (por ejemplo Claude Code). OpenSpec prepara los comandos para él, pero no lo incluye.

Instalar la CLI

PowerShell

```
npm install -g @fission-ai/openspec@latest  
openspec --version
```

bash

```
npm install -g @fission-ai/openspec@latest  
openspec --version
```

También puedes instalarlo con pnpm, yarn, bun o nix. Esta web está escrita contra la versión **1.14.0**.

La telemetría solo recoge el nombre del comando y la versión. En entornos corporativos conviene desactivarla; en CI ya se desactiva sola.

PowerShell

```
openspec config set telemetry.enabled false
# alternativa por variable de entorno
$env:OPENSPEC_TELEMETRY = "0"
```

bash

```
openspec config set telemetry.enabled false
# alternativa por variable de entorno
export OPENSPEC_TELEMETRY=0
```

Inicializar un repositorio

Desde la raíz del proyecto, de forma interactiva:

PowerShell

```
cd MiProyecto
openspec init
```

bash

```
cd MiProyecto
openspec init
```

Para saltarte el selector, indica la herramienta con `--tools`. Es la ruta rápida y la que se usa en el tutorial:

PowerShell

```
openspec init --tools claude
```

bash

```
openspec init --tools claude
```

CAMBIOS RESPECTO A LA 1.13

--tools admite más de 40 herramientas, además de los valores especiales all y none, y acepta una lista separada por comas. Ejecuta `openspec init --help` para ver los identificadores válidos de tu versión.

Opciones útiles de init:

Opción	Efecto
--tools <lista>	Herramientas de IA para las que generar comandos y skills (all, none o lista)
--language <idioma>	Idioma de los artefactos nuevos. No sobrescribe un config.yaml existente: si ya existe, indica el idioma en context
--profile <perfil>	Perfil de workflows: core o custom
--force	Limpia ficheros legacy de versiones anteriores

Estructura generada

```
MiProyecto/
├── .claude/                                # skills y comandos de OpenSpec (con --
tools claude)
├── openspec/
│   ├── config.yaml                        # contexto y reglas para la IA
│   ├── specs/                            # fuente de verdad, por dominio
│   ├── changes/                          # cambios en curso
│   └── archive/                          # cambios completados
```

Con el perfil core y Claude Code se generan 6 skills y 6 comandos en `.claude/`. No los edites a mano: `openspec update` los regenera y los sobrescribe.

PowerShell

```
● ● ●
openspec update
```

bash

```
● ● ●
openspec update
```

Comandos de chat

Los comandos `/opsx:*` se escriben en el chat de tu agente, no en la terminal.

Comando	Qué hace	Perfil
<code>/opsx:propose <nombre></code>	Crea el change con proposal, delta specs, design y tasks	core
<code>/opsx:explore</code>	Investiga el problema y el código sin escribir nada	core

Comando	Qué hace	Perfil
<code>/opsx:apply [<nombre>]</code>	Implementa las tareas de <code>tasks.md</code> y las marca	core
<code>/opsx:sync</code>	Sincroniza las delta specs con las specs principales sin archivar	core
<code>/opsx:archive</code>	Fusiona los deltas en specs/ y archiva el change	core
<code>/opsx:update</code>	Revisa los artefactos de planificación y los mantiene coherentes	core, experimental
<code>/opsx:verify, /opsx:new, /opsx:continue, /opsx:ff, /opsx:bulk-archive, /opsx:onboard</code>	Control incremental, verificación, archivado masivo y tutorial guiado	Perfil ampliado

Para activar el perfil ampliado:

PowerShell



```
openspec config profile
```

bash



```
openspec config profile
```

NUEVO Y EXPERIMENTAL

`/opsx:update` no existía en la 1.13. Solo modifica artefactos de planificación y nunca toca código. Al ser experimental, su comportamiento puede cambiar entre versiones.

config.yaml

Fija el schema, inyecta contexto en los artefactos y define reglas por artefacto:

```
schema: spec-driven

context: |
  Dominio: gestión de reservas (francjas, clientes, cancelaciones).
  Las APIs públicas deben mantener compatibilidad hacia atrás.

rules:
  specs:
    - Escenarios en formato Given/When/Then
  tasks:
    - Cada tarea incluye su test asociado
```

- context se aplica a **todos** los artefactos. Incluye solo restricciones que el agente no pueda deducir leyendo el código.
- rules.<artefacto> se aplica solo al artefacto indicado.

BLOQUE NUEVO: OPERATIONS

config.yaml admite además operations.apply.guidance y operations.archive.guidance: orientación para la ejecución de apply y archive, separada de las reglas de los artefactos. No aparecía en la documentación de la 1.13.

Comandos de terminal esenciales

Comando	Uso
openspec init [path]	Inicializa el repositorio
openspec update [path]	Regenera skills y comandos tras actualizar la versión (--force)
openspec list	Lista changes; con --specs, --changes, --archived o --json

Comando	Uso
<code>openspec show [item]</code>	Detalle de un change o spec (<code>--json</code> , <code>--type</code>)
<code>openspec validate [item]</code>	Valida estructura y formato (<code>--all</code> , <code>--strict</code> , <code>--json</code> , <code>--no-interactive</code>)
<code>openspec archive [change]</code>	Archiva un change (<code>-y</code> , <code>--skip-specs</code>)

En un repositorio recién inicializado, `openspec validate --all --strict --no-interactive` responde «No items found to validate.» con código de salida 0, así que puedes dejarlo en CI desde el primer día.

Siguiente paso

[Tu primer cambio en 30 minutos](#)

Tu primer cambio en 30 minutos

Tutorial práctico sobre un repositorio vacío: `openspec init`, `/opsx:propose`, `/opsx:apply` y `/opsx:archive` con un único requisito de gestión de reservas.

Vas a recorrer el ciclo completo de OpenSpec sobre un repositorio vacío con **un solo requisito pequeño** de gestión de reservas: `openspec init`, `/opsx:propose`, `/opsx:apply` y `/opsx:archive`. Cada paso termina con lo que deberías ver, para que verifiques sin adivinar.

Antes de empezar

- OpenSpec 1.14.0 instalado y Node.js 20.19.0 o superior (ver la página de instalación).
- Git.
- **Un agente de IA compatible** con comandos de chat, por ejemplo Claude Code. Los pasos 2 a 4 se ejecutan en su chat, no en la terminal. Esta guía usa Claude Code; con otra herramienta, cambia el valor de `--tools`.

NOTA

El agente es no determinista: los textos exactos de lo que genere variarán. Verifica la **estructura** (ficheros, secciones, SHALL, escenarios), no las palabras.

1. Crear el repositorio e inicializar OpenSpec (5 min)

PowerShell

```
mkdir reservas-demo
cd reservas-demo
git init
openspec init --tools claude
```

bash

```
mkdir reservas-demo
cd reservas-demo
git init
openspec init --tools claude
```

Deberías ver estas rutas nuevas:

```
reservas-demo/
├─ .claude/                # skills y comandos /opsx:*
└─ openspec/
    ├─ config.yaml
    ├─ specs/              # vacía
    └─ changes/            # vacía (aparecerá archive/ al archivar)
```

Comprueba que la validación no falla en un repositorio vacío:

PowerShell

```
openspec validate --all --strict --no-interactive
```

bash

```
openspec validate --all --strict --no-interactive
```

Deberías ver «No items found to validate.» y código de salida 0.

Abre el agente en esta carpeta. Un contexto mínimo en `openspec/config.yaml` mejora lo que genere; añade, bajo el context existente:

```
context: |
  Dominio: gestión de reservas (frangas horarias, clientes).
  Documentación y specs en español; palabras normativas (SHALL) y
  GIVEN/WHEN/THEN en inglés.
```

2. Proponer el cambio con `/opsx:propose` (8 min)

En el chat del agente:

```
/opsx:propose reservas-cancelacion
```

Describe el requisito cuando te lo pida, o pégalo directamente: «Un cliente puede cancelar una reserva hasta 24 horas antes de su inicio. Si falta menos, se rechaza».

Deberías ver una carpeta nueva con cuatro artefactos:

```
openspec/changes/reservas-cancelacion/
├─ proposal.md
├─ specs/
│   └─ reservas/
│       └─ spec.md          # delta spec
├─ design.md
└─ tasks.md
```

Revisa a mano, antes de seguir, lo que importa. Este es el punto de control humano del flujo:

1. `specs/reservas/spec.md` empieza por `### ADDED Requirements` y contiene un requisito con `SHALL` y al menos dos escenarios: cancelación a tiempo y cancelación fuera de plazo.
2. `tasks.md` es una lista de casillas - `[ ]`, con las tareas de implementación y de test.
3. `proposal.md` explica el porqué y el alcance sin salirse de este único requisito.

Si algo no cuadra, pídeselo al agente en el mismo chat y deja que edite el mismo change.

Valida el resultado:

PowerShell

```
● ● ●
openspec validate reservas-cancelacion --strict --no-interactive
openspec list --changes
```

bash

```
● ● ●
openspec validate reservas-cancelacion --strict --no-interactive
openspec list --changes
```

Deberías ver el change como válido y listado con su progreso de tareas, todavía a 0.

3. Implementar con `/opsx:apply` (10 min)

```
/opsx:apply
```

Con un único change activo no hace falta indicar el nombre; si tuvieras varios, usa `/opsx:apply reservas-cancelacion`. El agente implementa las tareas de `tasks.md` y las va marcando.

Deberías ver:

- Código y tests nuevos en tu repositorio (el lenguaje depende de lo que el agente elija o de lo que indiques en `context`).
- Todas las casillas de `tasks.md` en - [x].
- Cada escenario del delta con un test cuyo nombre lo referencia. Es una buena práctica de trazabilidad: si falta alguno, pídeselo.

Comprueba el avance desde la terminal:

PowerShell

```
openspec list --changes  
git status --short
```

bash

```
openspec list --changes  
git status --short
```

Deberías ver el change con todas las tareas completadas y los ficheros nuevos sin confirmar en Git.

4. Archivar con /opsx:archive (5 min)

```
/opsx:archive
```

OpenSpec fusiona el delta en la spec principal y mueve el change a archive/.

Deberías ver:

```
openspec/  
├─ specs/  
│   └─ reservas/  
│       └─ spec.md          # ahora contiene "Cancelación de reserva"  
└─ changes/  
    └─ archive/  
        └─ <fecha>-reservas-cancelacion/
```

Verifica el estado final y confirma el trabajo:

PowerShell

```
openspec list --specs  
openspec validate --all --strict --no-interactive  
git add -A  
git commit -m "feat: cancelación de reservas"
```

bash

```
openspec list --specs  
openspec validate --all --strict --no-interactive  
git add -A  
git commit -m "feat: cancelación de reservas"
```

Deberías ver la spec reservas listada y la validación sin errores.

CONSEJO

Archiva siempre antes del merge, dentro de la misma PR: así el código y las specs actualizadas entran juntos en main y nunca quedan desalineados.

Si algo falla

- **El agente no reconoce /opsx:propose.** Reinicia el agente en la carpeta del repositorio para que cargue `.claude/`, o regenera los ficheros con `openspec update`.
- **validate --strict falla.** Lee el mensaje: casi siempre es un requisito sin SHALL o sin escenarios, o un encabezado mal escrito. Corrígelo en el delta y vuelve a validar.
- **El change es mayor de lo previsto.** Reduce el alcance a un único requisito antes de aplicar: un change pequeño es más fácil de revisar.

Siguiente paso

[Escribir specs: formato y buenas prácticas](#)

Escribir specs

Cómo se escribe una spec principal y una delta spec (ADDED, MODIFIED, REMOVED) en OpenSpec, con buenas prácticas y una checklist de revisión.

Una spec describe **qué hace el sistema**, no cómo está construido. OpenSpec distingue entre las specs principales (openspec/specs/), que reflejan el comportamiento vigente, y las delta specs de cada change, que describen solo lo que ese change añade, modifica o elimina.

Reglas del formato

- Cada requisito empieza con ##### Requirement: <nombre> y usa una palabra normativa en inglés: **SHALL** o **MUST**.
- Cada requisito tiene al menos un ##### Scenario: <nombre> (cuatro almohadillas; con tres, el validador no lo reconoce).
- Los escenarios se escriben con **GIVEN / WHEN / THEN / AND** en negrita y en lista.
- Una capacidad por carpeta: openspec/specs/<capacidad>/spec.md.
- El nombre del requisito es su identificador: las deltas lo localizan por nombre exacto.

Ejemplo de spec principal

openspec/specs/reservas/spec.md

Reservas

Purpose

Gestión de reservas de salas por parte de los usuarios registrados.

Requirements

Requirement: Crear reserva

El sistema SHALL permitir reservar una sala libre para una franja horaria.

Scenario: Reserva de una sala libre

- ****GIVEN**** la sala "Norte" está libre de 10:00 a 11:00
- ****WHEN**** un usuario la reserva para esa franja
- ****THEN**** la reserva queda en estado "confirmada"

Scenario: Reserva duplicada

- ****GIVEN**** la sala "Norte" ya está reservada de 10:00 a 11:00
- ****WHEN**** otro usuario intenta reservarla para esa franja
- ****THEN**** el sistema rechaza la petición con el código `'RESERVA_DUPLICADA'`

Delta specs

Dentro de un change, la delta spec vive en

openspec/changes/<change>/specs/<capacidad>/spec.md y agrupa los requisitos por operación. Al archivar, OpenSpec aplica cada operación sobre la spec principal.

ADDED: requisito nuevo

openspec/changes/cancelacion-reservas/specs/reservas/spec.md

```
+++ ADDED Requirements
+
+#### Requirement: Cancelar reserva
+El sistema SHALL permitir cancelar una reserva confirmada hasta 2 horas
antes de su inicio.
+
+#### Scenario: Cancelación dentro de plazo
+- **GIVEN** una reserva confirmada que empieza dentro de 5 horas
+- **WHEN** su titular la cancela
+- **THEN** la reserva pasa a estado "cancelada" y la sala queda libre
```

MODIFIED: cambiar un requisito existente

MODIFIED **reescribe el requisito completo**: copia el bloque entero de la spec principal y edítalo. Lo que no incluyas se pierde al archivar. En el ejemplo, la línea eliminada es la versión anterior y las insertadas son lo nuevo.

openspec/changes/limite-reservas/specs/reservas/spec.md

```
+++ MODIFIED Requirements

#### Requirement: Crear reserva
-El sistema SHALL permitir reservar una sala libre para una franja horaria.
+El sistema SHALL permitir reservar una sala libre para una franja horaria de
4 horas como máximo.

#### Scenario: Reserva de una sala libre
- **GIVEN** la sala "Norte" está libre de 10:00 a 11:00
- **WHEN** un usuario la reserva para esa franja
- **THEN** la reserva queda en estado "confirmada"

+#### Scenario: Franja demasiado larga
+- **WHEN** un usuario intenta reservar una sala durante 5 horas
+- **THEN** el sistema rechaza la petición con el código `FRANJA_EXCEDIDA`
```

En el fichero real no dejas la línea antigua: los marcadores solo muestran qué cambia respecto a la spec principal.

REMOVED: retirar un requisito

```
openspec/changes/retirar-lista-espera/specs/reservas/spec.md
```

```
### REMOVED Requirements
```

```
-### Requirement: Lista de espera
```

```
**Reason**: la lista de espera apenas se usaba y complicaba la liberación de salas.
```

```
**Migration**: se avisa a los usuarios cuando una sala se libera; las entradas en lista de espera se eliminan.
```

REMOVED siempre lleva **Reason** (por qué se retira) y **Migration** (qué ocurre con quien dependía de ese comportamiento).

Buenas prácticas

- Escribe comportamiento observable, no detalles de implementación (nada de tablas, clases o endpoints concretos salvo que sean el contrato).
- Un requisito, una regla. Si un requisito necesita «y además», probablemente son dos.
- Cubre el caso feliz y al menos un caso de error con su código (RESERVA_DUPLICADA). Usa códigos de error estables, no mensajes literales: el frontend y los tests dependen de ellos.
- Usa datos de ejemplo concretos y neutros: salas, franjas, estados. Nunca datos personales.
- Cada escenario debe poder convertirse en un test cuyo nombre contenga Scenario: <nombre>.
- No rellenes specs de todo el sistema de golpe (*backfilling*): las specs crecen change a change, solo sobre lo que tocas.

Checklist de revisión

- ¿Cada requisito usa SHALL o MUST?
- ¿Cada requisito tiene al menos un escenario con #####?
- ¿Los escenarios cubren el caso feliz y los errores relevantes?
- ¿Los MODIFIED incluyen el requisito completo, con todos sus escenarios?
- ¿Los REMOVED explican **Reason** y **Migration**?
- ¿Pasa openspec validate --strict?

Siguiente paso

[Flujo /opsx de principio a fin](#)

Flujo /opsx de principio a fin

Los comandos /opsx de los perfiles core y ampliado, el flujo completo de un change hasta archivarlo en la PR y qué revisar en cada punto de control.

Los comandos /opsx:* se ejecutan en el chat de tu asistente de IA y guían cada fase de un change: explorar, proponer, implementar y archivar. Esta página reúne todos los comandos, el flujo de equipo recomendado y los puntos donde conviene revisar.

Comandos del perfil core

Comando	Qué hace
/opsx:explore	Investiga el problema y el código sin escribir nada; sirve para aclarar ideas antes de proponer.
/opsx:propose	Crea el change y genera proposal, delta specs, design y tasks en un paso.
/opsx:apply	Implementa las tareas de tasks.md y las marca al completarlas.
/opsx:archive	Fusiona las delta specs en las specs principales y mueve el change a archive/.
/opsx:sync	Fusiona las delta specs del change en las specs principales sin archivarlo .
/opsx:update	<i>(Experimental)</i> Revisa los artefactos de planificación de un change y los mantiene coherentes entre sí. No toca código.

PRECAUCIÓN

La KB (escrita para 1.13) describe /opsx:sync como «sincroniza el estado». En 1.14 lo que hace es fusionar las delta specs en openspec/specs/ sin archivar el change. /opsx:update es nuevo y no figura en la KB.

Comandos del perfil ampliado

Se activan eligiendo el perfil con `openspec config profile`.

Comando	Qué hace
<code>/opsx:new</code>	Crea el change vacío, sin generar artefactos.
<code>/opsx:continue</code>	Genera el siguiente artefacto pendiente, uno cada vez.
<code>/opsx:ff</code>	Genera de una vez todos los artefactos que faltan.
<code>/opsx:verify</code>	Comprueba que la implementación cumple specs, design y tasks.
<code>/opsx:bulk-archive</code>	Archiva varios changes terminados a la vez.
<code>/opsx:onboard</code>	Recorrido guiado de un change completo sobre tu propio repositorio.

Flujo completo del equipo

Flujo de un change

Ticket del backlog

- └─ rama con el mismo nombre que el change
 - └─ `/opsx:explore` (opcional, si hay incertidumbre)
 - └─ `/opsx:propose` → proposal, specs, design, tasks
 - └─ — revisión de la propuesta —
 - └─ `/opsx:apply` → código + tests
 - └─ `/opsx:verify` (perfil ampliado)
 - └─ `/opsx:archive` → specs principales actualizadas
- └─ PR: código + delta specs + archive
 - └─ — revisión de código, CI en verde —
- └─ merge a main

El **archive** va dentro de la misma **PR**, antes del merge. Así main nunca tiene código sin su spec ni specs que describan algo que no se ha fusionado.

Cambios grandes: PR de specs primero

Para cambios de alto impacto, abre primero una PR **solo con el change** (proposal, specs, design y tasks). Cuando se aprueba, implementa en la misma rama o en otra. Para cambios pequeños basta con revisar las specs dentro de la PR de código.

Qué revisar en cada punto de control

El flujo tiene tres momentos de revisión humana. Cada uno mira algo distinto; si se mezclan, la revisión acaba siendo de trámite.

Momento	Revisor	Foco
Tras propose	Responsable técnico o equipo	¿Resuelve el problema correcto? ¿Hay escenarios para el caso feliz, los errores y los límites (por ejemplo, RESERVA_DUPLICADA o una franja fuera de horario)? ¿El diseño es coherente con la arquitectura?
Tras verify	Quien implementa	¿Cada escenario tiene código y un test cuyo nombre contiene Scenario: <nombre>? ¿tasks.md refleja la realidad, sin tareas marcadas que no estén hechas?
PR	Revisor de código	Código, delta specs y archive en la misma PR; openspec validate --all --strict y los tests en verde en la pipeline.

Siguiente paso

[Recetas: feature, bugfix, refactor, paralelo](#)

Recetas: feature, bugfix, refactor, paralelo

Siete recetas para combinar /opsx según el trabajo: funcionalidad pequeña, bugfix, exploración, paralelo, refactor, control incremental y aprendizaje.

No todos los cambios necesitan el mismo recorrido. Estas recetas parten del flujo completo y lo ajustan al tipo de trabajo; los ejemplos usan una aplicación de gestión de reservas.

Receta 1: funcionalidad pequeña (camino rápido)

1. `/opsx:propose cancelacion-reservas`
2. Revisa proposal y specs.
3. `/opsx:apply`
4. `/opsx:archive`

Receta 2: bugfix

Igual que la receta 1, pero la propuesta se formula como **comportamiento correcto** y la delta spec usa **MODIFIED** sobre el requisito afectado. El bug queda documentado como requisito, con un escenario que lo cubre.

Chat

```
/opsx:propose fix-reserva-duplicada
```

Receta 3: explorar primero

Para cambios con incertidumbre:

1. `/opsx:explore` sobre el área afectada. En Claude Code encaja con el *plan mode* (Shift+Tab), que impide editar mientras investigas.
2. Cuando haya claridad, `/opsx:propose`: la propuesta aprovecha el contexto de la exploración.

Receta 4: cambios en paralelo

Cada change vive en su propia carpeta, así que no hay conflictos entre ellos. Para retomar uno concreto: `/opsx:apply exportacion-pdf`. Para trabajar en paralelo de verdad, combina OpenSpec con **git worktrees** (un directorio y una rama por change):

PowerShell

```
git worktree add ..\reservas-exportacion -b exportacion-pdf
openspec list
```

bash

```
git worktree add ../reservas-exportacion -b exportacion-pdf
openspec list
```

`openspec list` muestra los changes activos y su progreso.

Receta 5: refactor sin cambio de comportamiento

Propuesta y tasks, sin delta specs: el comportamiento no cambia, así que no hay nada que fusionar. Archiva saltando la actualización de specs:

PowerShell



PowerShell

```
openspec archive refactor-repositorio-reservas --skip-specs
```

bash



bash

```
openspec archive refactor-repositorio-reservas --skip-specs
```

Receta 6: control incremental (cambios grandes)

Con el perfil ampliado: /opsx:new → /opsx:continue (un artefacto cada vez, revisando entre medias) → /opsx:ff cuando el resto ya está claro.

Receta 7: aprender sobre el código real

/opsx:onboard (perfil ampliado) guía un change completo sobre tu propio repositorio. Es la forma más rápida de que alguien nuevo vea el ciclo entero con código que conoce.

Siguiente paso

[Proyectos existentes \(brownfield\)](#).

Proyectos existentes (brownfield)

Cómo introducir OpenSpec en un proyecto existente: sin documentar todo de golpe, con un piloto acotado, cambios pequeños y los riesgos habituales mitigados.

OpenSpec está pensado para proyectos que ya tienen código. La clave es no intentar describir todo el sistema antes de empezar: las specs crecen change a change, solo en las zonas que tocas.

Sin documentación masiva

- **No hagas backfilling:** escribir specs de todo el sistema de entrada cuesta semanas, se desactualiza y nadie lo revisa.
- Cada change describe solo el comportamiento que añade o modifica. Al archivarlo, ese fragmento pasa a openspec/specs/.
- Con el tiempo, las zonas con más actividad quedan bien especificadas; las que nadie toca no necesitan spec.

Elegir un piloto

- Un equipo pequeño y un repositorio con actividad frecuente.
- Un área de negocio acotada, por ejemplo el módulo de reservas de salas.
- Alguien que haga de referente y revise las primeras propuestas.
- Una duración cerrada (unas semanas) con revisión al final.

Primeros changes

1. Empieza por changes **pequeños y reales:** un bugfix o una funcionalidad menor, no una épica.

2. Antes de proponer, usa `/opsx:explore` para que el agente lea el código del área afectada.
3. Revisa con atención la primera delta spec: es la que fija el estilo de las siguientes.
4. Archiva dentro de la misma PR y comprueba que `openspec/specs/` refleja lo esperado.

Riesgos y mitigaciones

Riesgo	Mitigación
Specs que se desactualizan respecto al código	Archive obligatorio dentro de la PR; <code>openspec validate</code> en CI.
Sobrecarga de proceso en cambios triviales	Cambios editoriales o triviales sin change; receta rápida para lo pequeño.
El agente inventa comportamiento que no existe	<code>/opsx:explore</code> antes de proponer y revisión humana de las specs.
Specs demasiado ligadas a la implementación	Revisión con la checklist de Escribir specs .
Abandono tras el piloto	Métricas sencillas y retrospectiva al cerrar el piloto.

Definition of Done: añadidos

- El change está archivado en la misma PR y `openspec validate --all --strict` pasa.
- Cada escenario nuevo o modificado tiene un test cuyo nombre contiene `Scenario:<nombre>`.
- Las delta specs describen comportamiento, no implementación.

Siguiente paso

[config.yaml y schemas personalizados](#)

config.yaml y schemas personalizados

Cómo ajustar OpenSpec a tu proyecto con openspec/config.yaml (schema, context, rules y operations) y cómo crear schemas de flujo personalizados.

openspec/config.yaml adapta los artefactos que genera el agente a tu proyecto: qué flujo usar, qué contexto debe conocer y qué reglas cumple cada artefacto. Si el flujo estándar no encaja, puedes definir tu propio schema.

Estructura de config.yaml

openspec/config.yaml

```
schema: spec-driven

context: |
  Aplicación de gestión de reservas de salas.
  Las fechas se guardan en UTC y se muestran en la zona horaria de la sala.
  Los códigos de error de negocio usan MAYÚSCULAS_CON_GUIONES
  (RESERVA_DUPLICADA).

rules:
  proposal:
    - Indica qué equipos o integraciones se ven afectados.
  specs:
    - Cada requisito incluye al menos un escenario de error con su código.
  tasks:
    - Cada tarea de implementación tiene su tarea de test.

operations:
  apply:
    guidance: |
      Ejecuta los tests del módulo afectado tras cada tarea.
  archive:
    guidance: |
      Comprueba que openspec validate --all --strict pasa antes de archivar.
```

- **schema:** el flujo de artefactos que se usa por defecto (spec-driven es el estándar).
- **context:** se aplica a **todos** los artefactos. Incluye solo restricciones que el agente no puede deducir leyendo el código; repetir lo obvio solo gasta contexto.
- **rules.<artefacto>:** reglas específicas de un artefacto (proposal, specs, design, tasks).
- **operations:** orientación para la ejecución de apply y archive.

NOTA

`operations.apply.guidance` y `operations.archive.guidance` son nuevos en 1.14 y no aparecen en la KB. Separan las instrucciones de ejecución de las reglas de cada artefacto.

CLAUDE.md o config.yaml

Va en <code>CLAUDE.md</code>	Va en <code>config.yaml</code>
Convenciones generales del repositorio (comandos, estilo, git)	Contexto de dominio que deben reflejar los artefactos
Instrucciones para cualquier tarea del agente	Reglas por artefacto (<code>rules</code>)
Herramientas y entorno de desarrollo	Orientación para <code>apply</code> y <code>archive</code> (<code>operations</code>)

`openspec init --language` no sobrescribe un `config.yaml` existente: si ya lo tienes, ajusta el idioma a mano.

Schemas personalizados

Un schema define qué artefactos tiene un change y en qué orden se generan. Puedes partir del estándar o crear uno desde cero.

PRECAUCIÓN

`openspec schema` está marcado como **experimental** en 1.14: los comandos y el formato pueden cambiar entre versiones.

PowerShell

```
PowerShell

# Copiar el schema estándar al proyecto para modificarlo
openspec schema fork spec-driven mi-flujo
# O crear uno nuevo desde cero
openspec schema init rapido --artifacts proposal,tasks
# Validar estructura y plantillas
openspec schema validate mi-flujo
# Ver de dónde se resuelve el schema
openspec schema which
# Crear un change con un schema concreto
openspec new change mi-cambio --schema mi-flujo
# Listar schemas y plantillas disponibles
openspec schemas
openspec templates
```

bash

```
bash

# Copiar el schema estándar al proyecto para modificarlo
openspec schema fork spec-driven mi-flujo
# O crear uno nuevo desde cero
openspec schema init rapido --artifacts proposal,tasks
# Validar estructura y plantillas
openspec schema validate mi-flujo
# Ver de dónde se resuelve el schema
openspec schema which
# Crear un change con un schema concreto
openspec new change mi-cambio --schema mi-flujo
# Listar schemas y plantillas disponibles
openspec schemas
openspec templates
```

Para que un schema sea el predeterminado del proyecto, indícalo en la clave schema de config.yaml.

Siguiente paso

[Claude Code](#)

Claude Code

Cómo configurar Claude Code para trabajar con OpenSpec: estructura versionada, permisos, hook de validación, modelos por fase y subagente verificador.

OpenSpec genera los comandos `/opsx:*` y los skills, pero el trabajo en equipo exige algo más: permisos acotados, validación automática y un reparto claro de contexto. Esta página recoge la configuración de Claude Code que funciona bien con OpenSpec y que conviene versionar en el repositorio.

Estructura versionada en el repositorio

```
.claude/
├─ commands/opsx/...      # generados por openspec init (no editar)
├─ skills/...             # generados por openspec init (no editar)
├─ agents/spec-verifier.md # subagente de verificación independiente
└─ settings.json          # permisos, hooks y variables de entorno
.mcp.json                 # servidores MCP del backlog, sin secretos
CLAUDE.md                 # stack y convenciones de código
openspec/config.yaml      # contexto de dominio y reglas de specs
```

Con `openspec init --tools claude` y el perfil `core` se generan 6 skills y 6 comandos en `.claude/`. No los edites a mano: `openspec update` los regenera y los sobrescribe.

Versiona todo salvo las preferencias personales, que van en `.claude/settings.local.json`. Así todo el equipo trabaja igual.

CLAUDE.md frente a config.yaml

Fichero	Cuándo se usa	Qué poner
CLAUDE.md	Siempre, en toda la sesión	Stack, arquitectura, convenciones de código, comandos de build y test, reglas del repositorio
openspec/config.yaml	Al generar artefactos OpenSpec	Glosario de dominio, reglas de proposal, specs, design y tasks

Una regla imprescindible en CLAUDE.md: **nunca se edita openspec/specs/ directamente; solo cambia mediante /opsx:archive o /opsx:sync**. Si ya usas config.yaml, repasa [su guía](#) para no duplicar contexto en ambos ficheros.

Las plantillas completas de CLAUDE.md, config.yaml, del comando /ticket-propose y del subagente están en [Plantillas](#); aquí no se repiten.

Permisos, variables y hooks

Un .claude/settings.json razonable para OpenSpec:

.claude/settings.json

```
{
  "env": { "OPENSPEC_TELEMETRY": "0" },
  "permissions": {
    "allow": [
      "Bash(openspec list:*)",
      "Bash(openspec show:*)",
      "Bash(openspec status:*)",
      "Bash(openspec validate:*)"
    ],
    "ask": [
      "Bash(openspec archive:*)",
      "Bash(git push:*)",
      "Edit(/openspec/specs/**)",
      "Write(/openspec/specs/**)"
    ],
    "deny": ["Read(/.env)", "Read(/.env.*)"]
  },
  "hooks": {
    "Stop": [
      {
        "hooks": [
          {
            "type": "command",
            "command": "openspec validate --all --strict --no-interactive
1>&2 || exit 2"
          }
        ]
      }
    ]
  }
}
```

- **Lectura sin fricción, escritura con confirmación.** Los comandos de solo lectura de openspec se permiten; archive y las ediciones sobre openspec/specs/ piden confirmación, que es justo el punto donde conviene que mires.
- **Hook Stop.** Cuando Claude termina un turno se validan las specs. Si fallan, el código de salida 2 impide finalizar y devuelve los errores al agente para que los corrija. Es una

validación local, sin esperar a la CI.

- **Telemetría.** `OPENSPEC_TELEMETRY=0` evita enviar métricas desde los puestos del equipo.
- **Herramientas de escritura de un MCP.** Si conectas un servidor MCP del backlog, consulta sus herramientas con `/mcp` y añade a `deny` las que modifican datos (actualizar work items, transicionar issues). El agente lee el backlog pero no lo cambia.

Modelos por fase

Fase	Modelo recomendado	Cómo fijarlo
Explore y propose	opus	<code>model</code> : opus en el frontmatter del comando; modo <code>plan</code> (Mayús+Tab) para explorar
Apply	sonnet	<code>/model sonnet</code> : más rápido y barato, suficiente si <code>tasks.md</code> es bueno
Verify	opus en un subagente	Contexto limpio: no se autojustifica
Archive	cualquiera	Operación casi mecánica

Usa alias (opus, sonnet) en lugar de versiones concretas para que la configuración no caduque.

Subagente verificador

Un subagente definido en `.claude/agents/spec-verifier.md` revisa, sin modificar nada, que cada requisito y escenario del change tenga código y test, ejecuta los tests y detecta tareas marcadas como hechas que no lo están. Complementa a `/opsx:verify` porque trabaja en un contexto limpio.

Se invoca desde la conversación:

Usa el subagente `spec-verifier` sobre el cambio `rechazo-solicitudes-duplicadas`

Acceso y licencias

- **Claude Team o Enterprise** incluye puestos de Claude Code: lo más sencillo para un piloto.
- **Proveedores alternativos** (Amazon Bedrock, Google Vertex AI, Microsoft Foundry) son una opción si prefieres facturar por el contrato de tu nube y mantener el tratamiento de datos en tu tenant. Verifica la configuración vigente en la documentación de Claude Code.

DATOS PERSONALES Y MODELO

Código, specs y tickets que envías al modelo pueden contener datos personales. Antes de adoptar el flujo en un equipo, valida el tratamiento con tu delegado de protección de datos. Detalles en el [plan de adopción](#).

Entorno de trabajo

- Funciona igual en la **terminal** y en la **extensión de VS Code**: los comandos `/opsx:*`, el `.mcp.json` y el `settings.json` son los mismos.
- Para cambios en paralelo usa **git worktrees**, con una sesión de Claude Code por worktree.

Siguiente paso

[Otros agentes](#)

Otros agentes (Copilot, Cursor, Codex...)

OpenSpec es agnóstico de herramienta: qué agentes soporta openspec init 1.14, qué se mantiene igual que con Claude Code y qué cambia.

OpenSpec no depende de un asistente concreto: el flujo (proposal, specs, design, tasks) vive en ficheros del repositorio, y el agente es quien los lee y los escribe. Lo que cambia entre herramientas es cómo se instalan los comandos y skills.

Qué agentes soporta

`openspec init --tools` acepta `all`, `none` o una lista separada por comas. En la versión 1.14 la lista incluye, entre otras:

- `claude` (Claude Code)
- `codex`
- `cursor`
- `github-copilot`
- `gemini`
- `opencode`
- `cline`
- `amazon-q`
- `kiro`
- `roocode`
- `devin` (`windsurf` se acepta como alias)
- `agents`

La lista completa y vigente está en la ayuda de tu versión:

```
openspec init --help
```

LA LISTA CAMBIA ENTRE VERSIONES

Las herramientas soportadas crecen con cada versión de OpenSpec. No te fíes de listas copiadas: consulta `openspec init --help` en la versión que fijes en tu proyecto.

Instalación en un proyecto

Puedes configurar varios agentes a la vez si el equipo no usa el mismo:

```
openspec init --tools claude,cursor,github-copilot
```

Para GitHub Copilot existe además la opción `--copilot-cloud` (y su contraria `--no-copilot-cloud`), que prepara los ficheros del agente de codificación en la nube sin preguntar.

Qué es común a todos

- **Los artefactos son los mismos.** `openspec/changes/`, `openspec/specs/` y `openspec/config.yaml` no dependen del agente.
- **El flujo es el mismo.** Propose, apply, archive; la guía de [flujo /opsx](#) vale para cualquiera.
- **La validación es la misma.** `openspec validate --all --strict --no-interactive` funciona desde la CLI, la uses o no con un agente.

Qué cambia

- **Dónde se instalan comandos y skills.** Cada herramienta tiene su propia carpeta o fichero de instrucciones; `openspec init` los genera y `openspec update` los regenera.

No los edites a mano.

- **Cómo se invocan.** Los comandos `/opsx:*` son una convención de las herramientas con comandos propios; en otras puedes pedir la misma acción en lenguaje natural o con el skill correspondiente. Comprueba en la documentación de tu agente cómo se lanzan.
- **Permisos, hooks y subagentes.** El `settings.json`, el hook Stop y los subagentes de la [página de Claude Code](#) son específicos de esa herramienta. Si tu agente no tiene equivalente, mueve la validación a la [CLI](#).
- **Modelos.** El reparto por fase es una buena práctica, pero cómo se fija el modelo depende del agente.

FUENTE DE VERDAD

Para detalles de una herramienta concreta, manda la documentación oficial de OpenSpec y la de ese agente. Esta página solo recoge lo que es verificable con la propia CLI.

Siguiente paso

[Skills complementarias](#)

Skills complementarias

Qué skills de terceros conviene enchufar alrededor de `/opsx:*` (interrogar el ticket, TDD, diagnóstico de bugs, revisión, retro), cuáles no y cómo instalarlas.

OpenSpec decide **qué** se construye y deja constancia en Git. Cómo trabaja el agente en cada fase (preguntar antes de proponer, escribir el test en rojo, diagnosticar un bug) se puede reforzar con *skills*: carpetas con un `SKILL.md` que empaquetan un procedimiento reutilizable. Esta página analiza la colección de Matt Pocock (licencia MIT, [ver Recursos](#)), pensada como piezas sueltas que el equipo controla. No compite con OpenSpec: se enchufa alrededor.

RECOMENDACIÓN PARA UN PILOTO · CONTRASTADA EL 25 DE OCTUBRE DE 2026

El catálogo de skills cambia a menudo. Los veredictos de esta página son un punto de partida para un piloto, no una norma. Antes de adoptar una skill, revisa su `CHANGELOG.md` y mide su efecto con las métricas del [plan de adopción](#).

Dos tipos de skill

Tipo	Quién la invoca	Para qué sirve
Invocada por el usuario	Solo la persona, como comando (<code>/grill-me</code>)	Orquestrar un flujo
Invocada por el modelo	La persona o el agente, cuando la tarea encaja	Aportar una disciplina reutilizable (<code>tdd</code> , <code>code-review</code>)

Una skill invocada por el usuario puede apoyarse en skills invocadas por el modelo, pero no en otras skills de usuario. Por eso las skills de disciplina (`tdd`) combinan bien con `/opsx:apply`.

Qué usar en cada fase

Fase	Skill	Cómo encaja	Veredicto
Antes de <code>/opsx:propose</code>	<code>grill-with-docs</code>	Interroga el ticket hasta cerrar ambigüedades (límites, errores, casos simultáneos) y deja el glosario y las decisiones por escrito	Adoptar
Contexto de dominio	<code>domain-modeling</code>	Afina el lenguaje del dominio contra casos límite; necesita un glosario único (ver conflictos)	Adoptar
Propuesta	<code>to-spec</code>	Solapa con <code>/opsx:propose</code> y publica la spec en el gestor de tickets: rompe «el qué en el backlog, el cómo en Git»	No usar
Desglose	<code>to-tickets</code>	Solapa con <code>tasks.md</code> y duplicaría las subtareas	No usar
Implementación	<code>tdd</code>	Refuerza <code>/opsx:apply:</code> cada escenario empieza como un test en rojo	Adoptar
Bugfix	<code>diagnosing-bugs</code>	Reproduce en rojo, minimiza, formula hipótesis y deja el test de regresión, que es el escenario del requisito MODIFIED	Adoptar
Verificación	<code>code-review</code>	Su eje «Spec» comprueba si el código implementa fielmente la spec; compáralo con el subagente verificador	Probar

Fase	Skill	Cómo encaja	Veredicto
PR	pr	Aporta riesgo de merge y evidencias antes y después; fusióvalo con tu plantilla de PR	Inspirarse
Retrospectiva	retro	Propone mejoras del entorno del agente (reglas, hooks, comprobaciones) que entran por PR	Adoptar
Deuda técnica	improve-codebase-architecture	Detecta candidatos de refactor, que entran como changes sin delta specs	Adoptar
Sesiones largas	handoff	Compacta la conversación para que otra sesión continúe un change grande	Opcional
Contexto del agente	writing-for-agents	Ayuda a redactar CLAUDE.md, config.yaml y skills propias	Adoptar

Conflictos que resolver antes de instalar

1. **Un solo glosario.** Las skills de dominio mantienen un GLOSSARY.md, y OpenSpec tiene el glosario en el context de openspec/config.yaml. Deja el glosario en GLOSSARY.md, en la raíz, y haz que config.yaml lo referencie («Usa la terminología de GLOSSARY.md»).
2. **Una sola palabra para «spec».** La spec de to-spec es un PRD con historias de usuario. En el repositorio, *spec* significa siempre spec de OpenSpec: requisitos con SHALL y escenarios.
3. **Un solo comando de implementación.** La colección trae implement. Usa /opsx:apply, que marca tasks.md, y añade a CLAUDE.md: «Durante apply, sigue la disciplina de la skill tdd».

4. **Gestor de tickets.** El asistente de configuración solo trae plantillas para GitHub, GitLab y Markdown local. Con Jira o Azure DevOps, describe el flujo en prosa: «lectura vía MCP, nunca escritura; los estados los mueven automatizaciones nativas». Más en [Integración con Jira](#).
5. **Carpeta compartida.** `openspec init` y `openspec update` generan sus propias skills en `.claude/skills/`. Las de terceros conviven en la misma carpeta con nombres distintos; no edites las de OpenSpec.

Instalarlas como ficheros versionados

Hay dos vías: un plugin que se actualiza solo o ficheros editables dentro del repositorio. En los repositorios del equipo, usa la segunda e instala solo las skills que hayas adoptado:

```
PowerShell o bash

npx skills@latest add mattpocock/skills -a claude-code
```

El comando pregunta qué skills instalar y las copia a `.claude/skills/`. Haz commit de esa carpeta. Para actualizar, ejecuta `npx skills@latest update` en una rama y revisa el diff en la PR como cualquier otro código. No es un comando de OpenSpec, así que su sintaxis puede cambiar: compruébala en el README del repositorio.

SKILLS Y CADENA DE SUMINISTRO

Una skill que se actualiza sola cambia el comportamiento del agente sin pasar por una revisión: es un riesgo de **cadena de suministro**, igual que una dependencia sin versión fijada. Con ficheros versionados, cada cambio se revisa como código. No combines plugin y ficheros, porque duplicarías las skills.

Cómo introducirlas en el plan de adopción

- **Piloto manual:** solo `grill-with-docs` y `tdd`. Mide si bajan las iteraciones en la revisión de la propuesta.

- **Automatización:** prueba code-review junto al subagente verificador y quédate con el que detecte más huecos reales.
- **Retrospectivas:** ejecuta retro sobre una sesión representativa y convierte sus hallazgos en cambios de CLAUDE.md o de los hooks, por PR.

Siguiente paso

[Integración con Jira](#)

Integración con Jira

Cómo conectar OpenSpec con Jira sin duplicar información: mapeo de conceptos, convención de nombres, servidor MCP de solo lectura y estados por automatización.

Jira y OpenSpec responden a preguntas distintas. Si intentas sincronizar todo en ambos sentidos, divergen en pocas semanas. El principio que funciona es **no duplicar**.

Principio: cada cosa en su sitio

- **Jira** guarda el *qué* de negocio, los criterios de aceptación y el **estado**. Es donde trabaja quien prioriza.
- **El repositorio (OpenSpec)** guarda el *cómo* técnico y es la **fuentes de verdad del comportamiento**. Es donde trabaja el equipo de desarrollo.

Mapeo de conceptos

Jira	OpenSpec
Epic o Feature	Dominio en <code>openspec/specs/<dominio>/</code> y varios changes
Story	Un change (<code>openspec/changes/<id>/</code>)
Criterios de aceptación	Escenarios WHEN/THEN en la delta spec
Bug	Change con un requisito MODIFIED que describe el comportamiento correcto
Deuda técnica o refactor	Change sin delta specs (<code>openspec archive --skip-specs</code>)
Sub-tasks	<code>tasks.md</code> ; elige una sola fuente y no las dupliques en Jira
Estado (To Do, In Progress, Done)	Derivado de eventos de Git (rama, PR, merge)

Convención de nombres

Es la integración más importante, y no necesita ninguna herramienta:

- **El nombre del change es el nombre de la rama:** `res-123-rechazo-solicitudes-duplicadas`.
- **Commits y PR llevan la clave del ticket** (RES-123) para que Jira los enlace con los *smart commits*.
- **La primera línea de `proposal.md` enlaza al ticket.** Puedes forzarlo con una regla en `rules.proposal` de [config.yaml](#).

Conectar el agente a Jira con MCP

Atlassian ofrece un servidor MCP remoto oficial (Rovo MCP) con acceso a Jira, Confluence y otros productos. Cada persona se autentica con su cuenta mediante OAuth 2.1, así que no hay tokens en el repositorio.

`.mcp.json`

```
{
  "mcpServers": {
    "atlassian": {
      "type": "http",
      "url": "https://mcp.atlassian.com/v2/mcp"
    }
  }
}
```

En Claude Code, autentica desde `/mcp`. La autenticación con API token existe como alternativa para escenarios no interactivos, si tu administrador la habilita.

REVISADO EL 4 DE OCTUBRE DE 2026

El endpoint cambia con el tiempo. Según la guía oficial de Atlassian, el actual es `/v2/mcp`, y el antiguo `/v1/sse` dejó de soportarse tras el 30 de junio de 2026. Compruébalo en la [guía oficial](#) antes de configurarlo.

Del ticket a la propuesta

Un comando propio, por ejemplo `/ticket-propose RES-123`, encadena los pasos habituales:

1. Lee el ticket vía MCP, en solo lectura.
2. Si los criterios son ambiguos, para y pregunta.
3. Ejecuta `/opsx:propose` con el nombre `<clave>-<slug>`, enlaza el ticket y convierte cada criterio en un escenario.
4. Presenta un resumen para revisión antes de pasar a `apply`.

La plantilla completa está en [Plantillas](#).

Estados: automatizaciones, no el agente

No delegues las transiciones de estado en el modelo: es poco fiable y deja mala trazabilidad. Usa eventos de Git con *Jira Automation*:

- Rama creada → **In Progress**.
- Pull request fusionada → **Done**.

Como mucho, permite al agente **comentar** en el ticket el enlace a la propuesta, con permisos mínimos.

DATOS PERSONALES Y RGPD

Los tickets pueden contener datos personales (capturas, correos, identificadores de clientes) que el servidor MCP enviaría al modelo. Revisa el tratamiento con tu delegado de protección de datos antes de conectarlo, empieza con permisos de solo lectura y no guardes secretos en el `.mcp.json` versionado.

Si el ticket cambia después

Si el equipo edita los criterios en Jira después de aprobar la propuesta, hay que actualizar el change (con `/opsx:update`) o crear uno nuevo. Decide cuál de los dos desde el primer día y déjalo escrito.

Siguiente paso

[Integración con Azure DevOps](#)

Integración con Azure DevOps

Cómo integrar OpenSpec con Azure Boards y Azure Repos: convención AB#, servidor MCP local, validación en la branch policy y estados por automatización.

El planteamiento es el mismo que con [Jira](#): el backlog guarda el *qué* y el *estado*, y el repositorio guarda el *cómo* y la fuente de verdad del comportamiento. Esta página recoge lo que cambia con Azure DevOps.

Mapeo de conceptos

Azure Boards	OpenSpec
Epic o Feature	Dominio en <code>openspec/specs/<dominio>/</code> y varios changes
User Story o PBI	Un change
Criterios de aceptación	Escenarios WHEN/THEN en la delta spec
Bug	Change con un requisito MODIFIED
Task	<code>tasks.md</code> ; no la dupliques en Boards

Convención de nombres

- **Nombre del change = nombre de la rama:** `ab123-rechazo-solicitudes-duplicadas`.
- **Commits y PR con AB#123:** Azure DevOps enlaza automáticamente el work item.
- **Primera línea de propuesta1.md:** enlace al work item.

Servidor MCP de Azure DevOps

Microsoft ofrece dos servidores:

- **Remoto**, en `https://mcp.dev.azure.com/{organizacion}`. Usa autenticación con Microsoft Entra.
- **Local**, el paquete `@azure-devops/mcp`, que ejecutas con `npx` y se autentica, por ejemplo, con Azure CLI.

REVISADO EL 4 DE OCTUBRE DE 2026

El servidor remoto ya está disponible de forma general, pero Claude Code, Claude Desktop, Cursor y ChatGPT **todavía no pueden conectarse** a él: Entra no admite aún el registro dinámico de clientes OAuth que necesitan. Funcionan Visual Studio y VS Code. Con Claude Code, usa el servidor local, que sí está soportado. Verifica el estado en el [repositorio oficial](#) antes de decidir.

Configuración del servidor local, con carga selectiva de dominios y sin secretos en el repositorio:

.mcp.json

```
{
  "mcpServers": {
    "ado": {
      "type": "stdio",
      "command": "npx",
      "args": [
        "-y", "@azure-devops/mcp", "${ADO_ORG}",
        "-d", "core", "work", "work-items", "repositories",
        "--authentication", "azcli"
      ]
    }
  }
}
```

Necesitas haber hecho `az login` y definir la variable de entorno `ADO_ORG`. Incluye siempre el dominio `core`; añade solo los demás dominios que necesites.

Estados: automatización nativa

Con AB#123 en la PR, activa la opción de completar los work items enlazados al completar la PR. El agente no debe transicionar estados; como mucho, puede comentar el enlace a la propuesta.

Validación en la branch policy

Con Azure Repos, el trigger pr: del YAML no se aplica. Registra la pipeline de validación como **Build Validation** en la branch policy de main. La definición de la pipeline está en [Pipeline y validación en CI](#).

DATOS PERSONALES Y RGPD

Los work items pueden contener datos personales que el servidor MCP enviaría al modelo. Revisa el tratamiento con tu delegado de protección de datos antes de conectarlo, usa permisos mínimos y de solo lectura, y añade a deny las herramientas de escritura del servidor.

Siguiente paso

[Pipeline y validación en CI](#)

Pipeline y validación en CI

Cómo validar las specs en integración continua con Azure Pipelines o GitHub Actions, con la versión de OpenSpec fijada y una regla opcional que exija change.

La validación local (el hook Stop de [Claude Code](#)) avisa pronto, pero no obliga. Lo que hace obligatoria la disciplina es una comprobación en la CI que bloquee la fusión.

Qué validar

El comando es el mismo en cualquier plataforma:

```
openspec validate --all --strict --no-interactive
```

Comprueba que todas las specs y changes cumplen el formato. `--strict` convierte las advertencias en errores y `--no-interactive` evita preguntas en un entorno sin terminal.

Pipeline mínima

Azure Pipelines

azure-pipelines.openspec.yml

```
trigger: none
pool: { vmImage: ubuntu-latest }
steps:
  - task: NodeTool@0
    inputs: { versionSpec: '22.x' }
  - script: npm i -g @fission-ai/openspec@1.14.0
    displayName: Install OpenSpec
  - script: openspec validate --all --strict --no-interactive
    displayName: Validate specs
  env: { OPENSPEC_TELEMETRY: '0' }
```

Con Azure Repos, regístrala como **Build Validation** en la branch policy de main.

GitHub Actions

.github/workflows/openspec.yml

```
name: openspec
on:
  pull_request:
  push:
    branches: [main]
env:
  OPENSPEC_TELEMETRY: '0'
jobs:
  validate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 22
      - run: npm i -g @fission-ai/openspec@1.14.0
      - run: openspec validate --all --strict --no-interactive
```

FIJA LA VERSIÓN

Instala siempre una versión concreta de OpenSpec, nunca la última. Así los builds son reproducibles y una actualización no rompe la pipeline un martes cualquiera. Actualiza la versión en una rama dedicada.

Regla opcional: no hay código sin change

Puedes hacer fallar la CI si una PR modifica el código fuente sin tocar `openspec/changes/`, salvo que lleve una etiqueta `no-spec`. Define la excepción desde el primer día: hotfixes, configuración y cambios editoriales son casos legítimos, y sin una salida explícita la regla se acaba saltando de formas peores.

Revisa de vez en cuando cuántas PR usan `no-spec`: si crecen, el proceso se está erosionando.

Automatización futura

Cuando el flujo esté asentado, puedes ejecutar Claude Code en modo no interactivo (`claude -p`) en la pipeline con el prompt del verificador y publicar el resultado como comentario en la PR. No lo hagas en el primer sprint: mide antes el valor de la revisión humana y el coste en tokens por PR.

Siguiente paso

[Gobierno del cambio](#)

Gobierno del cambio

Cuándo una spec manda de verdad: espectro de autoridad, verja frente a detección temprana, reparto de papeles, cierre calculado y lo que OpenSpec deja fuera.

OpenSpec da la estructura (specs vigentes, changes con deltas y un validador), pero no decide por sí solo **quién ratifica, qué bloquea un merge ni cuándo un cambio está cerrado**. Esta página recoge el marco de gobierno que conviene montar alrededor de la herramienta. Se apoya en el documento técnico *Desarrollo de Software Basado en Especificaciones* de Roberto Canales, que compara OpenSpec, Spec Kit, Kiro y Tessl por mecanismos (ver [Recursos](#)).

Funciona no es lo mismo que es conforme

Que la cancelación de reservas «funcione» se comprueba con una demo. Que sea **conforme** exige compararla con un comportamiento acordado de antemano, por ejemplo «una reserva solo se cancela sin coste con 24 h de antelación», y dejar evidencia reproducible: el escenario y su test en verde en la pipeline.

De ahí sale la primera regla: **la conversación con el agente no puede ser el único sitio donde vive la intención**. Lo que tenga que sobrevivir a la sesión va a ficheros versionados. CLAUDE.md aporta contexto, las skills procedimiento y los hooks automatismo. Los tres mejoran al agente, pero ninguno especifica el comportamiento del producto. Eso lo hace la spec.

Espectro de autoridad de la spec

Modelo	Qué manda después de implementar	Riesgo típico
Spec-first	El código. La spec se escribió antes y luego envejece	Documentación que miente a los seis meses
Spec-anchored	La spec ratificada sigue siendo normativa; la deriva se controla con revisión y tests	Revisiones «de trámite» que dejan pasar deriva
Spec-as-source	La spec. La salida generada no se edita: se cambia la spec y se regenera	Exige regeneración repetible y detectar ediciones manuales

OpenSpec encaja en **spec-anchored por cambios**: cada change propone deltas sobre openspec/specs/, que sigue siendo la verdad vigente tras archivar. Es el punto de partida recomendado. Mueve un módulo a spec-as-source solo cuando tengas regeneración repetible, detección de ediciones manuales y una CI que lo proteja.

Detección temprana frente a verja

Una [verja](#) combina tres piezas: una **comprobación ejecutable**, una **política obligatoria** y un **bloqueo efectivo**. Si falta una, no es una verja.

Mecanismo	Qué es	Bloquea el merge
Hook Stop de Claude Code	Detección temprana: avisa en local al terminar el turno	No
openspec validate en el portátil	Comprobación sin política: depende de que alguien la ejecute	No

Mecanismo	Qué es	Bloquea el merge
Pipeline con <code>openspec validate --all -strict</code> , build y tests, registrada como validación obligatoria de la rama <code>main</code>	Verja	Sí

Mantén las tres, pero no las confundas: un verde local no protege `main`. Junto a la verja funciona el [tringuete](#), que permite avanzar sin dejar que la calidad retroceda. Por ejemplo, la cobertura de escenarios de un dominio no puede bajar de un change al siguiente. La configuración de la pipeline está en [Validación en CI](#).

Quién propone, quién ratifica, quién comprueba

Papel	Responsable	Ejemplo con reservas
Propone	El agente	<code>/opsx:propose</code> genera la delta spec de cancelación con sus escenarios
Ratifica	Una persona	El revisor aprueba la PR de la propuesta sobre un commit concreto
Comprueba	Un programa	La pipeline ejecuta el validador y el test de cada escenario

Ningún mecanismo debe permitir que el modelo ratifique su propio resultado. En la práctica, esto implica tres cosas:

- **Ratificación ligada a una versión.** La aprobación es una aprobación de PR o un comentario con el commit revisado, no un «ok» en el chat.
- **Tests aprobados protegidos.** Escribe primero los tests de los escenarios, revísalos con la propuesta y comprueba que fallan (en rojo) antes de `/opsx:apply`. En la PR, cualquier cambio posterior en esos tests se revisa aparte, para que el agente no los «ajuste» hasta que pasen.
- **Specs protegidas.** `openspec/specs/` solo cambia al archivar dentro de la PR.

El cierre se calcula, no se declara

Un change no está terminado porque el agente diga «he terminado». Está terminado cuando el [cierre calculado](#) lo deduce de artefactos y resultados:

- todas las filas de la matriz de cobertura verificadas;
- la CI en verde sobre el mismo commit que se va a fusionar;
- las aprobaciones requeridas presentes.

La matriz de cobertura es la tabla que produce el subagente verificador y que acompaña a la PR:

Requirement	Scenario	Código	Test	Estado
Cancelación sin coste	Cancelación con más de 24 h	CancelarReserva	Scenario: Cancelación con más de 24 h	Cubierto
Cancelación sin coste	Cancelación con menos de 24 h	CancelarReserva	—	Sin cubrir

Una sola fila «sin cubrir» bloquea la Definition of Done. Para que la trazabilidad funcione, enlaza cada eslabón por su identificador: intención (ticket) → change → requirement → escenario → test → código → ejecución de CI.

Lo que OpenSpec deja fuera

OpenSpec aporta la separación entre spec vigente y propuesta, deltas por capacidad (ADDED, MODIFIED, REMOVED y RENAMED), un validador determinista de estructura y un histórico navegable. Estas son las carencias que tiene que cubrir el equipo:

Carencia	Cómo cubrirla
La sincronización puede adelantar la «verdad vigente» antes del cierre autorizado	Fusionar en openspec/specs/ solo al archivar dentro de la PR, y proteger esa carpeta con CODEOWNERS y rama protegida
config.yaml orienta al agente, pero no concede permisos ni protege rutas	Permisos reales: deny de escritura sobre openspec/specs/** en la configuración del agente y política de rama en el repositorio
«Válido» no significa «correcto»: el validador comprueba la forma, no el significado	Revisión humana de la delta spec con checklist y trazabilidad de cada escenario a su test
Ninguna herramienta garantiza el comportamiento	Los tests son del proyecto: cada escenario tiene un test automatizado en la verja

NO HAY GANADOR UNIVERSAL

La comparativa del documento concluye que la elección depende de dónde deba vivir la autoridad. Spec Kit aporta una «constitución», pero no separa la spec vigente de la propuesta. Kiro integra la disciplina en el IDE, pero sus puertas son un modo de uso, no una política. Tessl versiona el contexto, pero versionar no equivale a ratificar.

Medir antes de optimizar

El gobierno añade pasos, así que tiene que demostrar valor. Usa las métricas del [plan de adopción](#). Si sube la trazabilidad, pero también los defectos y el tiempo de ciclo sin reducir riesgos, el enfoque no está funcionando y hay que replantearlo.

Siguiente paso

[Plan de adopción y riesgos](#)

Plan de adopción y riesgos

Plan por fases para adoptar OpenSpec en un equipo: elección del piloto, definición de hecho, métricas, riesgos con su mitigación y checklist de arranque.

Adoptar desarrollo guiado por specs sin añadir burocracia al flujo Scrum exige hacerlo de forma incremental y medible. Este plan lo divide en fases con un criterio de salida claro.

Plan por fases

Fase	Duración	Alcance	Criterio de salida
0. Preparación	1 semana	Formación, elección del piloto, revisión con protección de datos	El equipo tiene el modelo mental y el piloto está elegido
1. Piloto manual	Sprints 1–2	<code>openspec init</code> , <code>CLAUDE.md</code> , <code>config.yaml</code> , convención de nombres, revisión de specs en la PR y las skills <code>grill-with-docs</code> y <code>tdd</code> . Sin MCP	5 o más changes archivados y feedback del equipo
2. Integración con el backlog	Sprint 3	MCP de solo lectura y comando de <code>ticket</code> a propuesta	Propuestas generadas desde tickets sin retrabajo mayor

Fase	Duración	Alcance	Criterio de salida
3. Automatización	Sprint 4 en adelante	Validación en CI como validación obligatoria (la verja), hook Stop, subagente verificador, CODEOWNERS sobre openspec/specs/ y estados por automatización	Pipeline obligatoria en main
4. Extensión	Posterior	Resto de proyectos y revisión automática en CI	Adopción en todos los repositorios activos

Elegir el proyecto piloto

Busca una aplicación en mantenimiento activo, con cobertura de tests razonable, un equipo motivado y un impacto de negocio medio (no crítico). Empieza con **cambios reales y pequeños** de la semana: un bug con regla de negocio, un endpoint nuevo con escenarios de error y un refactor.

Definition of Done

Añade a la definición de hecho del equipo:

- ☐ Change creado y propuesta revisada antes de implementar (o etiqueta no-spec justificada).
- ☐ Tests de los escenarios escritos y vistos en rojo antes de implementar, en cambios con comportamiento nuevo o bugfix.
- ☐ Cada escenario cubierto por un test automatizado; la tabla del verificador no tiene filas «sin cubrir».
- ☐ Change archivado en la misma PR y openspec/specs/ actualizado.

- `□ openspec validate --strict` en verde en la pipeline; el verde local no basta.

Métricas para evaluar el piloto

- **Retrabajo**: iteraciones de código tras la revisión. Se espera que baje.
- **Defectos escapados** en historias con spec frente a las que no la tienen.
- **Tiempo de ciclo** por historia: igual o menor tras la curva de aprendizaje.
- **Cobertura de escenarios**: porcentaje con test; el objetivo es el 100 %.
- **Omisiones detectadas antes del merge**, por la revisión de la propuesta o por el verificador.
- **Coste en tokens** por historia.
- **Percepción del equipo**, con una encuesta breve en la retrospectiva.

Criterio de corte: si sube la trazabilidad pero también los defectos y el tiempo de ciclo sin reducir riesgos, el enfoque no ha demostrado valor y hay que replantearlo.

Riesgos y mitigaciones

Riesgo	Mitigación
Revisión de specs «de trámite»: se aprueba sin leer	Checklist de revisión, rotación de revisores y repaso en las retros
Rellenar specs de todo el código existente	Política brownfield: solo specs de lo que se toca. Ver proyectos existentes
Datos personales en tickets y código enviados al modelo	Revisión con protección de datos, plantilla de ticket sin datos reales y deny de ficheros sensibles
Specs desalineadas con el código	Archive dentro de la PR, validación en la pipeline y subagente verificador

Riesgo	Mitigación
«Válido» confundido con «correcto»: specs bien formadas pero erróneas pasan la pipeline	El validador solo comprueba la forma; revisión humana del significado y trazabilidad de cada escenario a su test. Ver Gobierno del cambio
openspec/specs/ actualizado fuera de una PR	Archive solo dentro de la PR, CODEOWNERS y rama protegida sobre esa carpeta
El agente ajusta los tests para que pasen	Tests de los escenarios escritos y revisados antes de implementar; cualquier cambio posterior en ellos se revisa en la PR
El agente modifica el backlog	MCP en solo lectura y estados por automatizaciones nativas
Skills de terceros que se actualizan solas	Cambian el comportamiento del agente sin revisión: instálalas como ficheros versionados y actualízalas por PR. Ver Skills complementarias
Solape de comandos de skills con /opsx:*	Selección cerrada de skills; una sola fuente para specs y tareas
Versiones y tutoriales desactualizados	Versión fijada y openspec update controlado
Rotura por actualizar OpenSpec	Versión fijada en CI y actualización en una rama dedicada
Limitación del MCP remoto de Azure DevOps con Claude Code	Servidor local; revisa el estado periódicamente
Cambios sin spec por urgencia	Etiqueta no-spec explícita y visible, con revisión periódica de su uso
Coste de tokens	Modelo por fase y medición por historia

DATOS PERSONALES Y RGPD

Antes de conectar un agente a tickets, código o repositorios con datos personales, informa a tu delegado de protección de datos y deja por escrito qué datos pueden llegar al modelo. Es la condición previa de la fase 2 del plan.

Checklist de arranque del piloto

- ☐ Node 20.19 o superior y OpenSpec instalado en los puestos del equipo.
- ☐ Telemetría desactivada.
- ☐ `openspec init` ejecutado con las herramientas que usa el equipo.
- ☐ `CLAUDE.md` y `openspec/config.yaml` redactados con el stack y el dominio reales.
- ☐ Skills seleccionadas instaladas como ficheros versionados en `.claude/skills/`.
- ☐ Glosario único (`GLOSSARY.md`) referenciado desde `config.yaml`.
- ☐ Convención de nombres acordada y documentada.
- ☐ Plantilla de PR y Definition of Done actualizadas.
- ☐ Delegado de protección de datos informado.
- ☐ Primer change real elegido para el sprint.

Las plantillas de PR y de ticket están en [Plantillas](#).

Antes de abrir la PR

Repasa esta lista en cada change, no solo durante el piloto:

- ☐ Cada requisito usa SHALL o MUST y describe comportamiento observable.
- ☐ Todos los criterios de aceptación de la historia están cubiertos por escenarios, incluidos los errores y los límites.
- ☐ Los MODIFIED reescriben el requisito completo y los REMOVED indican motivo y migración.
- ☐ Las specs no contienen detalles de implementación: esos van en `design.md`.
- ☐ `openspec validate <change> --strict` pasa sin errores.

- ☐ La rama y el change se llaman igual.
- ☐ `tasks.md` refleja la realidad: nada marcado que no esté hecho.
- ☐ El subagente verificador no deja escenarios sin cubrir.
- ☐ El change está archivado y `openspec/specs/` se actualiza en esta misma PR.
- ☐ Las migraciones de datos, si las hay, tienen plan de vuelta atrás.

Siguiente paso

[Comandos de chat](#)

Comandos de chat /opsx:*

Referencia de los comandos de chat /opsx:* que genera openspec init en OpenSpec 1.14: qué hace cada uno, cuándo usarlo y qué workflows añade el perfil ampliado.

Los comandos /opsx:* se escriben en el chat de tu asistente de IA, no en la terminal. openspec init los genera para la herramienta elegida; con Claude Code quedan en .claude/commands/opsx/ junto a sus skills en .claude/skills/. Esta página recoge los que genera la versión 1.14.0 con el perfil **core**.

NOTA

No edites a mano los ficheros generados: openspec update los regenera al subir de versión y sobrescribe tus cambios.

Comandos del perfil core

Comando	Qué hace	Cuándo usarlo
/opsx:explore	Modo exploración: investiga el problema y el código sin escribir artefactos ni código.	Antes de proponer, cuando hay incertidumbre.
/opsx:propose <nombre>	Crea el change y genera proposal, delta specs, design y tasks en un solo paso.	Para iniciar cualquier cambio de comportamiento.
/opsx:apply [<nombre>]	Implementa las tareas de tasks.md y las marca al completarlas.	Cuando la propuesta está revisada y aprobada.
/opsx:update [<nombre>]	(Experimental) Revisa los artefactos de planificación de un change y los mantiene coherentes entre sí. No toca código.	Tras el feedback de la revisión de la propuesta.

Comando	Qué hace	Cuándo usarlo
<code>/opsx:sync</code> <code>[<nombre>]</code>	Fusiona las delta specs del change en openspec/specs/ sin archivarlo.	Cuando necesitas las specs principales al día antes de archivar.
<code>/opsx:archive</code> <code>[<nombre>]</code>	Fusiona las delta specs y mueve el change a openspec/changes/archive/.	Al terminar, dentro de la misma PR que el código.

Ejemplo en el dominio de reservas

```
/opsx:explore ¿Qué pasa hoy si dos clientes reservan la misma franja?
/opsx:propose rechazo-reservas-solapadas
/opsx:apply rechazo-reservas-solapadas
/opsx:archive rechazo-reservas-solapadas
```

Diferencias con la KB (1.13)

- `/opsx:sync` ya no «sincroniza el estado»: fusiona las delta specs en las specs principales sin archivar.
- `/opsx:update` es nuevo en 1.14.
- Los tutoriales anteriores a la 1.x usaban `/openspec:proposal` y `project.md`; hoy son los comandos `/opsx:*` y `openspec/config.yaml`.

Workflows del perfil ampliado

`openspec init` con el perfil core no genera más comandos. Con `openspec config profile` puedes activar workflows adicionales; cada uno añade su comando de chat con el mismo prefijo:

Workflow	Qué hace
new	Crea el change vacío, sin generar artefactos.
continue	Genera el siguiente artefacto pendiente, uno cada vez.
ff	Genera de una vez todos los artefactos que faltan.
verify	Comprueba que la implementación cumple specs, design y tasks.
bulk-archive	Archiva varios changes terminados a la vez.
onboard	Tutorial guiado que recorre un change completo sobre tu propio código.

El uso detallado de estos workflows está en [Flujo /opsx de principio a fin](#).

Siguiente paso

[CLI openspec](#)

CLI openspec

Referencia de la CLI openspec 1.14.0 extraída de su ayuda: instalación, comandos, subcomandos y flags más usados, con variantes para PowerShell y bash.

La CLI openspec se ejecuta en la terminal. Inicializa el repositorio, valida y archiva changes, y genera las instrucciones que usan los comandos de chat. Esta referencia parte de la ayuda de la versión fijada (`openspec --help` y `openspec <comando> --help`), no de la KB escrita para 1.13.

CONSEJO

Cualquier comando acepta `--help`. Si esta página y tu terminal no coinciden, comprueba tu versión con `openspec --version`: manda la ayuda de la versión instalada.

Instalación y versión

Requiere Node.js 20.19 o superior. Fija la versión en lugar de usar `@latest`.

PowerShell

```
PowerShell

npm install -g @fission-ai/openspec@1.14.0
openspec --version
# Versión instalada y si hay actualización disponible
openspec version --check
# Desactivar la telemetría anónima
openspec config set telemetry.enabled false
$env:OPENSPEC_TELEMETRY = "0"
```

bash

```
bash

npm install -g @fission-ai/openspec@1.14.0
openspec --version
# Versión instalada y si hay actualización disponible
openspec version --check
# Desactivar la telemetría anónima
openspec config set telemetry.enabled false
export OPENSPEC_TELEMETRY=0
```

Opciones globales: `--no-color` desactiva el color de la salida y `-h / --help` muestra la ayuda.

Comandos de primer nivel

Configuración del proyecto

Comando	Qué hace
<code>openspec init [ruta]</code>	Inicializa OpenSpec en el proyecto. <code>--tools claude</code> evita el asistente interactivo; <code>--language</code> fija el idioma de los artefactos nuevos; <code>--profile core</code> o <code>custom</code> ; <code>--force</code> limpia ficheros legacy.
<code>openspec update [ruta]</code>	Regenera los ficheros de instrucciones (comandos y skills) tras subir de versión. <code>--force</code> los regenera aunque estén al día.

Comando	Qué hace
<code>openspec config</code>	Configuración global: <code>path</code> , <code>list</code> , <code>get</code> , <code>set</code> , <code>unset</code> , <code>reset</code> , <code>edit</code> y <code>profile</code> .
<code>openspec completion</code>	Autocompletado de la shell: <code>generate</code> , <code>install</code> y <code>uninstall</code> .
<code>openspec version</code>	Versión instalada; con <code>--check</code> comprueba si hay actualización.
<code>openspec feedback <mensaje></code>	Envía feedback sobre OpenSpec; <code>--body</code> añade detalle.

Changes y specs

Comando	Qué hace
<code>openspec new change <nombre></code>	Crea la carpeta de un change. Admite <code>--description</code> , <code>--goal</code> y <code>--schema</code> .
<code>openspec list</code>	Lista los changes activos; <code>--specs</code> lista las specs y <code>--archived</code> los archivados.
<code>openspec show [elemento]</code>	Muestra un change o una spec; <code>--type change</code> o <code>--type spec</code> si el nombre es ambiguo, <code>--json</code> para máquinas.
<code>openspec view</code>	Dashboard interactivo de specs y changes.
<code>openspec status</code>	Estado de los artefactos de un change (<code>--change <id></code>) o de todos (<code>--all</code>).
<code>openspec validate [elemento]</code>	Valida changes y specs: <code>--all</code> , <code>--changes</code> , <code>--specs</code> , <code>--strict</code> , <code>--json</code> , <code>--no-interactive</code> .

Comando	Qué hace
<code>openspec archive [change]</code>	Fusiona las delta specs en <code>openspec/specs/</code> y mueve el change a <code>archive/</code> . <code>--skip-specs</code> para cambios sin comportamiento; <code>-y / --yes</code> sin confirmaciones.
<code>openspec change</code>	Subcomandos <code>show</code> , <code>list</code> y <code>validate</code> sobre changes (<code>openspec change list</code> está obsoleto: usa <code>openspec list</code>).
<code>openspec spec</code>	Subcomandos <code>show</code> , <code>list</code> y <code>validate</code> sobre specs.

Schemas e instrucciones

Comando	Qué hace
<code>openspec schemas</code>	Lista los schemas de workflow disponibles.
<code>openspec schema</code>	<i>(Experimental)</i> <code>which</code> , <code>validate</code> , <code>fork</code> e <code>init</code> para schemas propios.
<code>openspec templates</code>	Rutas resueltas de las plantillas de cada artefacto de un schema.
<code>openspec instructions [artefacto]</code>	Instrucciones enriquecidas para un artefacto, <code>apply</code> o <code>archive</code> ; es lo que consultan las skills.

Raíz, stores y vistas de trabajo

Comando	Qué hace
<code>openspec doctor</code>	Salud de las relaciones en la raíz OpenSpec resuelta.
<code>openspec context</code>	Contexto de trabajo de la raíz resuelta.

Comando	Qué hace
<code>openspec store</code>	Stores: repositorios OpenSpec independientes registrados en la máquina (setup, register, unregister, remove, list, doctor). Muchos comandos aceptan <code>--store <id></code> .
<code>openspec workset</code>	Vistas de trabajo personales y locales: create, list, open, remove.

Uso diario

PowerShell

```

PowerShell

# Inicializar sin preguntas para Claude Code
openspec init --tools claude
# Crear un change del dominio de reservas
openspec new change rechazo-reservas-solapadas
# Ver estado y validar antes de la PR
openspec status --change rechazo-reservas-solapadas
openspec validate rechazo-reservas-solapadas --strict
# Archivar dentro de la PR
openspec archive rechazo-reservas-solapadas --yes

```

bash

```

bash

# Inicializar sin preguntas para Claude Code
openspec init --tools claude
# Crear un change del dominio de reservas
openspec new change rechazo-reservas-solapadas
# Ver estado y validar antes de la PR
openspec status --change rechazo-reservas-solapadas
openspec validate rechazo-reservas-solapadas --strict
# Archivar dentro de la PR
openspec archive rechazo-reservas-solapadas --yes

```

Validación en CI



```
openspec validate --all --strict --no-interactive
```

En un repositorio recién inicializado devuelve «No items found to validate.» con código 0, así que el job de CI no falla antes del primer change.

Refactor sin cambio de comportamiento



```
openspec archive refactor-repositorio-reservas --skip-specs
```

Siguiente paso

[Plantillas](#)

Plantillas

Plantillas para copiar: CLAUDE.md, openspec/config.yaml, /ticket-propose, el subagente spec-verifier, la validación en CI y las plantillas de PR y ticket.

Adapta nombres de proyecto, rutas y convenciones antes de usarlas. Los ejemplos usan un dominio neutro de gestión de reservas.

SIN DATOS PERSONALES

Las plantillas de ticket y de PR no deben contener datos personales reales: lo que escribas en ellas puede llegar al modelo de IA.

CLAUDE.md

En la raíz del repositorio. Claude Code lo carga siempre: reserva para él el stack, los comandos y las convenciones de código.

Contexto del proyecto

Stack

- Backend: .NET 8, ASP.NET Core Web API, Clean Architecture (Api / Application / Domain / Infrastructure)
- Persistencia: EF Core + SQL Server, migraciones code-first
- Frontend: Angular (standalone components, signals)
- Tests: xUnit + FluentAssertions + Testcontainers; Angular con Jest

Comandos

- Build: ``dotnet build``
- Tests backend: ``dotnet test``
- Tests frontend: ``npm run test`` (desde /src/Web)

Convenciones

- Casos de uso en Application; validación con FluentValidation.
- Errores de negocio con códigos estables (p. ej. RESERVA_SOLAPADA), mapeados a ProblemDetails.
- Nada de lógica de negocio en controladores.

Spec-driven development (OpenSpec)

- Todo cambio de comportamiento pasa por un change de OpenSpec (/opsx:propose).
- NUNCA editas openspec/specs/ directamente: solo cambia mediante /opsx:archive.
- Cada Scenario de una spec debe tener un test automatizado.
- Nombre del change = nombre de la rama = `<clave-ticket>-<slug>`.

openspec/config.yaml

Dominio, glosario y reglas por artefacto. Incluye solo restricciones que el agente no puede inferir leyendo el código.

openspec/config.yaml

```
schema: spec-driven

context: |
  Dominio: gestión de reservas (recursos, franjas, reservas).
  Glosario:
    - franja: intervalo de tiempo reservable de un recurso
    - reserva: asignación confirmada de una franja a un cliente
  Las APIs públicas mantienen compatibilidad hacia atrás.

rules:
  proposal:
    - Primera línea: enlace al ticket de origen (Jira PROJ-XXXX o Azure Boards AB#XXXX)
    - Incluir impacto en base de datos y plan de rollback
    - Identificar contratos de API o eventos afectados (breaking o no)
  specs:
    - Formato Given/When/Then; un escenario por regla de negocio y por caso de error
    - Cada criterio de aceptación del ticket tiene al menos un Scenario
    - Si un criterio es ambiguo, registrarlo como pregunta abierta en proposal.md
  design:
    - Referenciar patrones existentes antes de proponer nuevos
  tasks:
    - Cada tarea incluye su test asociado
    - Migraciones de base de datos en tarea separada

operations:
  apply:
    guidance:
      - Ejecuta los tests afectados tras cada bloque de tareas
  archive:
    guidance:
      - Confirma que el subagente spec-verifier no reporta escenarios sin cubrir
```

El bloque operations (orientación para apply y archive) aparece en la plantilla que genera la 1.14 y no figura en la KB.

Comando /ticket-propose

Comando de chat propio en `.claude/commands/ticket-propose.md`: crea un change a partir de un ticket leído por MCP en solo lectura.

```
.claude/commands/ticket-propose.md
```

```
---
```

```
description: Crea un cambio OpenSpec a partir de un ticket de Jira o Azure Boards
```

```
argument-hint: <PROJ-1234 | AB#1234>
```

```
model: opus
```

```
---
```

1. Lee el ticket \$ARGUMENTS con el MCP correspondiente (título, descripción, criterios de aceptación, enlaces). Solo lectura: no modifiques el ticket.
2. Si hay criterios ambiguos o contradictorios, lístalos y PARA; pregúntame antes de seguir.
3. Ejecuta `/opsx:propose` con el nombre `<clave-ticket-en-minúsculas>-<slug-kebab-case>`.
 - `proposal.md`: enlace al ticket en la primera línea.
 - `specs`: al menos un Scenario por criterio de aceptación.
4. Muéstrame un resumen de `proposal.md` y de las delta specs para revisión antes de `/opsx:apply`.

Subagente spec-verifier

Subagente en `.claude/agents/spec-verifier.md`: verifica la implementación sin modificar nada, después de `/opsx:apply` y antes de abrir la PR.

```
.claude/agents/spec-verifier.md
```

```
---
```

```
name: spec-verifier
```

```
description: Verifica de forma independiente que la implementación de un  
cambio OpenSpec cumple sus specs. Úsalo después de /opsx:apply y antes de  
abrir la PR.
```

```
tools: Read, Grep, Glob, Bash
```

```
model: opus
```

```
---
```

```
No modifiques ningún fichero.
```

1. Lee openspec/changes/<cambio>/ (proposal, specs, tasks).
2. Para cada Requirement y Scenario, localiza el código que lo implementa y el test que lo cubre.
3. Ejecuta los tests de los proyectos afectados.
4. Devuelve:
 - Tabla: Requirement | Scenario | Código | Test | Estado (cubierto / parcial / sin cubrir)
 - Tareas marcadas [x] en tasks.md que no están realmente implementadas
 - Riesgos detectados (breaking changes, migraciones, seguridad)

Validación en Azure Pipelines

Regístrala como validación de build en la política de main. La versión de OpenSpec va fijada.

```
azure-pipelines.openspec.yml
```

```
trigger: none
```

```
pool: { vmImage: ubuntu-latest }
```

```
steps:
```

- task: NodeTool@0
 - inputs: { versionSpec: '22.x' }
- script: npm i -g @fission-ai/openspec@1.14.0
 - displayName: Install OpenSpec
- script: openspec validate --all --strict --no-interactive
 - displayName: Validate specs
- env: { OPENSPEC_TELEMETRY: '0' }

La variante para GitHub Actions está en [Validación en CI](#).

Plantilla de pull request

.github/pull_request_template.md

Ticket

PROJ-1234 / AB#1234

Change OpenSpec

`openspec/changes/archive/<fecha>-proj-1234-...` (o `no-spec` justificado)

Checklist

- [] Propuesta y delta specs revisadas antes de implementar
- [] Cada Scenario tiene test automatizado
- [] spec-verifier sin elementos "sin cubrir"
- [] Change archivado en esta PR (specs/ actualizado)
- [] Pipeline verde (openspec validate + build + tests)
- [] Migraciones revisadas y con plan de rollback (si aplica)

Plantilla de ticket

Story o PBI compatible con OpenSpec: redactar los criterios en Given/When/Then facilita que el agente los convierta en escenarios sin interpretar.

Ticket (Story / PBI)

Como <rol> **quiero** <objetivo> **para** <beneficio>

Criterios de aceptación

1. Dado <contexto>, cuando <acción>, entonces <resultado>
2. Dado <contexto>, cuando <acción errónea>, entonces <error esperado>

Fuera de alcance

- ...

Notas (sin datos personales reales)

Siguiente paso

[Glosario](#)

Glosario

Glosario de OpenSpec y del desarrollo guiado por specs: spec, change, delta spec, archive, spec-anchored, verja, MCP y más, cada uno con ancla estable.

Cada término tiene un ancla estable derivada de su nombre (en minúsculas, sin tildes y con guiones), por ejemplo `/referencia/glosario/#delta-spec`. Úsala para enlazar desde otras páginas.

Conceptos de OpenSpec

SDD

Spec-Driven Development, desarrollo guiado por specs: una especificación estructurada guía la implementación del agente de IA.

Spec

Documento en `openspec/specs/<dominio>/spec.md` que describe el comportamiento actual del sistema. Es la fuente de verdad.

Requirement

Regla de comportamiento dentro de una spec, redactada con palabras normativas como SHALL o MUST.

Scenario

Caso concreto de un requirement, con la forma WHEN / THEN (y opcionalmente GIVEN). Cada escenario debe tener un test automatizado.

Change

Carpeta en `openspec/changes/<nombre>/` con una propuesta de cambio: `proposal`, `delta specs`, `design` y `tasks`.

Delta spec

Fragmento de `spec` dentro de un `change` que marca requisitos como `ADDED`, `MODIFIED` o `REMOVED` respecto a la `spec` principal.

Proposal

Artefacto `proposal.md` de un `change`: intención, alcance, enfoque y enlace al ticket de origen.

Design

Artefacto `design.md` de un `change` con las decisiones técnicas y las alternativas descartadas.

Tasks

Artefacto `tasks.md` de un `change`: checklist de implementación que el agente marca al completar cada tarea.

Archive

Operación que fusiona las `delta specs` en `openspec/specs/` y mueve el `change` a `openspec/changes/archive/`.

Schema

Definición de qué artefactos tiene un `change` y en qué orden se generan. El `schema` por defecto es `spec-driven`.

Perfil

Conjunto de workflows que `openspec init` genera como comandos de chat: el perfil core o uno personalizado con `openspec config profile`.

Store

Repositorio OpenSpec independiente registrado en la máquina, útil para compartir specs entre varios repositorios.

Gobierno del cambio

Spec-first

La spec se escribe antes de implementar, pero después manda el código y la spec envejece. Ver [Gobierno del cambio](#).

Spec-anchored

La spec ratificada sigue siendo normativa tras implementar; la deriva se controla con revisión y tests. Es la posición de OpenSpec, que trabaja por changes.

Spec-as-source

La salida generada no se edita: se cambia la spec y se regenera el código. Exige regeneración repetible y detectar ediciones manuales.

Verja

Comprobación ejecutable más política obligatoria más bloqueo efectivo; por ejemplo, la validación de specs, el build y los tests registrados como requisito para fusionar en main. Un hook local no es una verja.

Trinquete

Regla que permite avanzar sin dejar que la calidad retroceda; por ejemplo, que la cobertura de escenarios de un dominio no baje entre changes.

Cierre calculado

Un change se da por cerrado cuando el sistema lo deduce de artefactos y resultados (matriz de cobertura completa, CI en verde sobre el mismo commit, aprobaciones presentes), no porque el agente lo declare.

Entorno y equipo

MCP

Model Context Protocol: protocolo para que el agente de IA acceda a herramientas externas como Jira o Azure DevOps.

Skill

Instrucciones empaquetadas que el agente carga cuando la tarea encaja; `openspec init` genera una por cada comando de chat del perfil.

Brownfield

Proyecto existente, frente a *greenfield* (proyecto desde cero). En OpenSpec solo se escriben specs de lo que se toca.

Siguiente paso

[FAQ](#).

FAQ

Preguntas frecuentes sobre OpenSpec 1.14: specs y changes, cuándo archivar, bugs, validación y verja, skills, Jira y Azure DevOps, idioma y telemetría.

Empezar

¿Tengo que documentar todo el sistema antes de empezar?

No. OpenSpec está pensado también para proyectos existentes: solo se escriben specs de lo que se va a cambiar. La documentación crece con cada change archivado. Más en [proyectos existentes](#).

¿Qué diferencia hay entre openspec/specs y openspec/changes?

specs/ describe cómo se comporta el sistema hoy (fuente de verdad). changes/ contiene propuestas en curso, expresadas como deltas (ADDED / MODIFIED / REMOVED). Al archivar un change, sus deltas se fusionan en specs/.

¿Cómo lo aprendo rápido?

Sigue [tu primer cambio](#) en un repositorio de pruebas y consulta [Recursos](#). El workflow onboard del perfil ampliado guía un change completo sobre tu propio código.

Flujo de trabajo

¿Cuándo se archiva un change?

Antes del merge, dentro de la misma PR, para que código y specs entren juntos en main.

¿Qué pasa con los refactors que no cambian comportamiento?

Se crea un change con propuesta y tareas, sin delta specs, y se archiva con `openspec archive <change> --skip-specs`.

¿Y los bugs?

Se proponen como comportamiento correcto con un requisito MODIFIED. El bug queda documentado y protegido por un test.

¿Qué hago si la revisión pide cambios en la propuesta?

Itera sobre el mismo change. En 1.14, `/opsx:update` revisa los artefactos de planificación y los mantiene coherentes entre sí sin tocar código.

¿Cómo evito que las specs se queden desactualizadas?

Archive en la PR, `openspec validate --all --strict --no-interactive` en la pipeline, hook Stop en Claude Code, subagente verificador, CODEOWNERS sobre `openspec/specs/` y una regla en `CLAUDE.md` que prohíba editar specs/ directamente.

Si `openspec validate` pasa, ¿la spec es correcta?

No. El validador comprueba la estructura y el formato, no el significado. La corrección la garantizan la revisión humana de la delta spec y el test de cada escenario. Más en [Gobierno del cambio](#).

¿El hook `Stop` sustituye a la pipeline?

No. El hook detecta errores pronto en local, pero no bloquea un merge. La verja real es la pipeline registrada como validación obligatoria en main. Más en [Gobierno del cambio](#).

¿Qué hago con un cambio urgente sin tiempo para specs?

Etiqueta no-spec visible en la PR. Revisa su uso en las retrospectivas; si se abusa, el proceso se erosiona.

¿OpenSpec sustituye a los tests?

No. Cada Scenario debe tener un test automatizado. Las specs dicen qué debe pasar; los tests lo garantizan.

Equipo y herramientas

¿Qué modelos de IA puedo usar?

OpenSpec es agnóstico: depende de la herramienta. `openspec init --tools` admite más de 40 asistentes. Con Claude Code se suele usar un modelo más capaz para explorar, proponer y verificar, y otro más rápido para implementar.

¿Funciona con Jira y con Azure DevOps?

Sí. El backlog guarda el qué y el estado; OpenSpec, el cómo. El agente lee los tickets por MCP y los estados se actualizan con automatizaciones nativas basadas en eventos de Git. Más en [Jira](#) y [Azure DevOps](#).

¿Debe el agente mover los tickets de estado?

No. Mejor automatizaciones de Jira o Azure DevOps disparadas por rama, PR y merge. El agente, como mucho, comenta en el ticket.

¿Duplico las sub-tasks en Jira y en tasks.md?

No. Elige una sola fuente; recomendamos tasks.md.

¿Qué pasa si front y back están en repositorios distintos?

Puedes guardar las specs en el repositorio que define el contrato (normalmente el backend) o usar stores: repositorios OpenSpec independientes que se gestionan con `openspec store`.

¿Las skills de terceros sustituyen a OpenSpec?

No. Son piezas sueltas que refuerzan fases concretas: interrogar el ticket antes de proponer, TDD y diagnóstico de bugs al implementar, revisión y retrospectiva al cerrar. Evita las que solapan con `/opsx:propose` o con tasks.md. Más en [Skills complementarias](#).

¿Instalo las skills como plugin o como ficheros?

En los repositorios del equipo, como ficheros versionados en `.claude/skills/`, para revisar cada actualización por PR. El plugin que se actualiza solo, mejor para uso personal. Más en [Skills complementarias](#).

¿Dónde vive el glosario de dominio?

En un único GLOSSARY.md en la raíz, referenciado desde openspec/config.yaml. Así lo usan tanto OpenSpec como las skills de dominio. Más en [Skills complementarias](#).

¿En qué idioma escribo las specs?

Palabras normativas (SHALL/MUST, GIVEN/WHEN/THEN) en inglés; términos de dominio en el idioma del negocio. Con `openspec init --language` los artefactos nuevos se generan en ese idioma; si ya tienes config.yaml, indícalo en context.

Privacidad

¿Envía datos OpenSpec a algún sitio?

Solo telemetría anónima (nombre de comando y versión), desactivable con `openspec config set telemetry.enabled false` o la variable `OPENSPEC_TELEMETRY=0`. En CI se desactiva sola. El contenido de specs y código lo procesa el modelo de IA de la herramienta que uses, no OpenSpec.

Siguiente paso

[Documentación, vídeos y comunidad](#)

Documentación, vídeos y comunidad

Recursos externos para aprender OpenSpec: documentación oficial, artículos de la comunidad y vídeos, con su tipo e idioma y un filtro por tipo.

Enlaces externos seleccionados para profundizar en OpenSpec. La mayoría están en inglés; el idioma de cada recurso aparece junto a su tipo.

TUTORIALES ANTIGUOS

Las versiones anteriores a la 1.x usaban comandos y ficheros distintos, como `/openspec:proposal` o `project.md`. Los actuales son los comandos `/opsx:*` y `openspec/config.yaml`. Ante cualquier discrepancia, manda la documentación oficial de la versión que tengas instalada.

Filtrar por tipo

☒ Todos ☐ Oficial ☐ Comunidad ☐ Vídeo

Documentación oficial

[CLI](#) Oficial · Inglés

[Commands](#) Oficial · Inglés

[Core Concepts](#) Oficial · Inglés

[Customization](#) Oficial · Inglés

[Comunidad en Discord](#) Oficial · Inglés

[Documentación oficial \(carpeta docs/\)](#) Oficial · Inglés

[Editing Changes](#) Oficial · Inglés

[Examples & Recipes](#) Oficial · Inglés

[Existing Projects](#) Oficial · Inglés

[Explore First](#) Oficial · Inglés

[FAQ](#) Oficial · Inglés

[Getting Started](#) Oficial · Inglés

[Glossary](#) Oficial · Inglés

[How Commands Work](#) Oficial · Inglés

[El propio proyecto desarrollado con OpenSpec](#) Oficial · Inglés

[openspec.dev](#) Oficial · Inglés

[Repositorio de OpenSpec en GitHub](#) Oficial · Inglés

[Schema por defecto spec-driven](#) Oficial · Inglés

[Stores \(Beta\)](#) Oficial · Inglés

[Supported Tools](#) Oficial · Inglés

[Troubleshooting](#) Oficial · Inglés

[Workflows](#) Oficial · Inglés

Comunidad

[Deep dive de arquitectura de OpenSpec](#) Comunidad · Inglés

[intent-driven.dev — OpenSpec](#) Comunidad · Inglés

[Linear MCP + OpenSpec](#) Comunidad · Inglés

[Mapping OpenSpec to Jira](#) Comunidad · Inglés

[Skills de Matt Pocock \(grill-with-docs, tdd, diagnosing-bugs, code-review, retro...\)](#)

Comunidad · Inglés

[openspec-mcp \(servidor MCP de la comunidad\)](#) Comunidad · Inglés

[Roberto Canales — Desarrollo de Software Basado en Especificaciones](#) Comunidad · Español

Vídeos

[Spec-Driven Development for AI Agents: I Tried OpenSpec and Others](#) Vídeo · Inglés

[Build with OpenSpec: Spec-Driven Development in Claude Code](#) Vídeo · Inglés

[Getting Started with OpenSpec — Setup Tutorial](#) Vídeo · Inglés

[OpenSpec GitHub Tutorial](#) Vídeo · Inglés

[Spec-Driven Development: GitHub Spec Kit & OpenSpec Explained](#) Vídeo · Inglés

[The OpenSpec Tutorial Every Developer Needs](#) Vídeo · Inglés

Ruta de aprendizaje sugerida

1. Lee Getting Started y Core Concepts y mira un vídeo de instalación.
2. Instala OpenSpec en un repositorio de pruebas y sigue [tu primer cambio](#).
3. Lee Existing Projects y Examples & Recipes.
4. Escribe un `config.yaml` real para un proyecto de tu equipo.
5. Haz un primer cambio real y pequeño sobre un proyecto existente.

Esta web, construida con OpenSpec

Historial real de esta web: los changes de OpenSpec archivados y los dominios de spec actuales, generados desde el repositorio.

Esta web se desarrolla con el flujo que enseña: cada cambio pasa por un change de OpenSpec antes de llegar a main. Esta página se genera al compilar el sitio a partir de la carpeta openspec/ del repositorio.

Changes archivados

2026-10-11 [w15-infografia-sdd](#)

El autor quiere una infografía-resumen de OpenSpec («mapa visual» del método) visible nada más entrar en la web y en el manual. La referencia visual (`` docs/Desarrollo_guiado_por_especificaciones (1).png ``, generada con IA) tiene el estilo deseado pero no es publicable: contiene rutas y comandos erróneos (`` openspec/apecs/ ``, `` ropax:archive ``), una tabla ilegible, texto corrupto y un enfoque «.NET y Angular» ajeno al sitio. En una web que enseña OpenSpec, publicar comandos incorrectos resta credibilidad y contradice la regla de no inventar comandos.

2026-10-05 [fix-cursor-terminal](#)

En w13 el `` design.md `` (D2) fija que las animaciones de la terminal se reproducen una sola vez para no distraer ni gastar CPU, pero el cursor de la terminal parpadea en bucle infinito (`` animation: blink 1s steps(1) infinite ``). El spec se cumple con movimiento reducido, pero diseño y código se contradicen.

2026-10-05 [w13-redisenio-landing](#)

La landing `` / `` usa el hero y las tarjetas por defecto de Starlight. No transmite que el sitio trata de un flujo de trabajo en terminal con agentes ni da peso al ciclo `` /opxs ``. Se ha elegido la dirección `**1a "Terminal en vivo"**` de `` docs/Diseño web OpenSpecs innovador/Landing Rediseño.dc.html ``: hero a dos columnas con una terminal que "ejecuta" el ciclo, franja de cuatro pasos, frases numeradas y tarjetas de recorrido con uno recomendado.

2026-10-05 [w14-manual-pdf](#)

El sitio enseña OpenSpec en 25 páginas repartidas en siete secciones. Hay lectores que quieren estudiarlo sin conexión, imprimirlo o compartirlo con su equipo como un único documento. Hoy la única forma es imprimir página a página desde el navegador, sin portada, sin índice y con el cromó de Starlight (sidebar, cabecera, buscador) en cada hoja.

2026-10-04 [w04-landing](#)

La landing actual (`/ `) es el placeholder splash de Starlight: un tagline y un único botón "Empieza aquí". No explica qué es OpenSpec, no muestra el ciclo `/opsx` y no orienta a los tres perfiles de lector (quien empieza, quien adopta en equipo, quien consulta). El objetivo de conversión del sitio es "Empieza en 30 minutos" (`/empieza/primer-cambio/`) y la landing no apunta ahí.

2026-10-04 [w05-seccion-empieza](#)

The four "Empieza" pages are placeholders ("Página en preparación"). They are the start of the main reading path (Landing → Qué es → Conceptos → Instalación → Primer cambio → Escribir specs) and the landing CTA already points at them, so the site cannot teach OpenSpec until they exist.

2026-10-04 [w06-seccion-guias](#)

Las cinco páginas de Guías son marcadores ("Página en preparación"). `/guias/formato-de-specs/` es el destino del último "Siguiendo paso" de Empieza (w05), así que el recorrido Inicio termina hoy en una página vacía. Sin Guías, el lector aprende a instalar OpenSpec pero no a escribir specs ni a usar el flujo `/opsx` con criterio.

2026-10-04 [w07-agentes-y-equipo](#)

Las seis páginas de Con tu agente y En equipo son marcadores ("Página en preparación"). `/agentes/claude-code/` es el destino del último "Siguiendo paso" de Guías (w06), así que el recorrido termina hoy en una página vacía. Sin estas secciones, el lector sabe escribir specs pero no cómo configurar su agente ni cómo llevar OpenSpec a un equipo con backlog, CI y un plan de adopción.

2026-10-04 [w08-referencia-y-recursos](#)

Las cinco páginas de Referencia y `/recursos/` son marcadores ("Página en preparación"). Referencia es la entrada del recorrido "Consulta" de la landing y el destino del último "Siguiendo paso" de En equipo (w07), así que hoy el lector termina en páginas vacías. Además,

la KB se escribió para OpenSpec 1.13 y la CLI fijada es la 1.14.0: una referencia de comandos copiada de la KB enseñaría comandos y flags que ya no coinciden.

2026-10-04 [w09-como-se-hizo](#)

The site teaches OpenSpec but does not yet show its own OpenSpec history: ``/como-se-hizo/`` is a placeholder. Rendering the archived changes and current specs from the repository at build time turns the site itself into a living, always-current example of the workflow.

2026-10-04 [w10-seo-avanzado](#)

El sitio ya emite idioma, título, canónica y sitemap (spec ``seo``), pero al compartir una URL no hay tarjeta enriquecida (Open Graph/Twitter), los buscadores no reciben datos estructurados (fecha de revisión, migas de pan) y no existe ``robots.txt`` que anuncie el sitemap. Además ``description`` no es obligatoria en las guías, así que nada impide publicar una página sin meta description o con una longitud inútil para el snippet.

2026-10-04 [w12-despliegue-vercel](#)

El sitio aún no tiene un despliegue declarado: faltan cabeceras de seguridad, política de caché y previews por PR revisables. Desplegar pronto (orden recomendado 01 → 02 → 03 → 12) permite revisar cada change en un preview real de Vercel, como exige la Definition of Done.

2026-10-03 [w01-fundacion-starlight](#)

El sitio sigue siendo la plantilla de Starlight (título "My Docs", sidebar de ejemplo, idioma inglés). Antes de escribir contenido (changes 03–09) hace falta una base estable: idioma, estructura de navegación con slugs definitivos, 404 propia y SEO base, verificada con tests E2E desde el primer día.

2026-10-03 [w02-calidad-ci](#)

Hoy la CI solo valida specs y ejecuta ``astro check`` + build. No ejecuta ningún test (el script ``test:e2e`` existe pero no se lanza en el workflow), no comprueba enlaces rotos y no hay tests unitarios ni de accesibilidad. Los siguientes changes (w03–w12) añaden contenido y comportamiento en bloque; sin una puerta de calidad que falle de forma fiable, los errores llegarán a ``main``.

2026-10-03 [w03-esquema-de-contenido](#)

Las guías enseñan una herramienta que cambia de versión con frecuencia. Sin metadatos tipados (versión de OpenSpec, fecha de revisión, nivel, duración) el lector no sabe si una

página está vigente ni si es para él, y nada impide publicar una guía sin esos datos. Este change es prerequisite de las secciones de contenido (05–08).

Dominios de spec actuales

[busqueda](#) 1 requisito

[calidad](#) 8 requisitos

[contenido](#) 34 requisitos

[despliegue](#) 6 requisitos

[landing](#) 14 requisitos

[navegacion](#) 4 requisitos

[seo](#) 12 requisitos

[showcase](#) 5 requisitos

Colofón

TÍTULO	OpenSpec desde cero · Manual práctico
AUTOR	Jesús Pedro Rodríguez
VERSIÓN DE REFERENCIA	OpenSpec v1.14.0
REVISIÓN	Octubre de 2026
EDICIÓN EN LÍNEA	https://openspec-ochre.vercel.app/

Este documento se genera automáticamente a partir del mismo contenido que la web, que es siempre la edición más reciente. Si encuentras una diferencia entre ambos, manda la web.

Fuentes principales

- [OpenSpec \(Fission-AI\)](#) — Repositorio, CLI y documentación oficial (carpeta docs/).
- openspec.dev — Web oficial del proyecto.
- [Roberto Canales — Desarrollo de Software Basado en Especificaciones](#) — Marco de gobierno del cambio: espectro de autoridad, verja y cierre calculado.
- [Matt Pocock — Skills](#) — Colección de skills de agente que complementan el flujo /opsx.

El catálogo completo de documentación, artículos y vídeos está en [Recursos](#).