

Del código a la especificación:

Cómo los LLMs están transformando el desarrollo de software



Matías Toro

Doctor en Ciencias Mención Computación por la Universidad de Chile. Profesor Asistente del Departamento de Ciencias de la Computación de la misma Universidad, e Investigador Joven del Instituto Milenio Fundamentos de los Datos. Líneas de investigación: lenguajes de programación, sistemas de tipos y privacidad de datos.

✉ mtoro@dcc.uchile.cl



Resumen / Los modelos de lenguaje de gran escala (LLMs) están redefiniendo el desarrollo de software. Lo que comenzó como asistentes de autocompletado ha evolucionado hacia agentes autónomos capaces de escribir, testear y desplegar aplicaciones completas. Este artículo examina cómo esta transformación desplaza el valor profesional desde la escritura de código hacia la especificación precisa de intención. Se presenta un panorama desde la *generación conversacional* hasta los sistemas multi-agente autónomos, con atención al “Spec-Driven Development” y al emergente “Harness Engineering”. El análisis se complementa con observaciones desde la experiencia docente y de desarrollo del autor.

Introducción

El cuello de botella del desarrollo de software se está desplazando. La pregunta ya no es si la inteligencia artificial puede programar (puede) sino cómo dirigirla de forma confiable a escala. A medida que los agentes de IA asumen tareas cada vez más complejas, el valor profesional migra desde la escritura de código hacia la especificación precisa de intención, el diseño de arquitectura y el aseguramiento de calidad.

Dos casos recientes ilustran hasta dónde ha llegado esta transformación. A principios de 2026, un equipo de entre tres y siete ingenieros de OpenAI construyó un producto interno con más de un millón de líneas de código en cinco meses, sin escribir código directamente. Utilizaron agentes de Codex para toda la implementación (lógica de aplicación, tests, CI, documentación y observabilidad) en un décimo del tiempo estimado para hacerlo a mano [1]. En paralelo, un investigador de Anthropic puso a trabajar 16 instancias de Claude sobre un repositorio compartido para construir un compilador de C en Rust. En cerca de 2.000 sesiones, los agentes produjeron 100.000 líneas de código capaces de compilar el *kernel* de Linux en tres arquitecturas [2]. En ambos casos, el humano no escribió código: diseñó el entorno, escribió especificaciones y tests, y supervisó.

Este artículo examina el espectro de enfoques que han emergido para trabajar con LLMs en desarrollo de software, con énfasis en dos marcos convergentes: el Spec-Driven Development (SDD), que pone la especificación en el centro del proceso, y el Harness Engineering (ingeniería de arnés), que convierte esa especificación en infraestructura viva para gobernar agentes autónomos.

El espectro

La interacción entre desarrolladores y LLMs admite grados. Existe un espectro con niveles crecientes de estructura y autonomía (ver Tabla 1).

He recorrido cada parte de este espectro. Partí usando LLMs para generar fragmentos de código o corregir errores puntuales. Después transiciné a GitHub Copilot y su autocompletado, que al día de hoy sigo usando en clases para escribir código trivial o implementar casos repetitivos. Ahora trabajo con Claude Code, un agente que ejecuta tareas completas en el terminal. He llegado a confiar tanto en lo que produce que en muchos casos apenas miro el código resultante y ejecuto el agente con permisos sin restricciones. No planifiqué esta progresión; simplemente cada herramienta fue quedando corta para lo que necesitaba.

Enfoque	Interacción	Autonomía	Caso de uso
Generación conversacional	Conversacional	Nula: el humano revisa manualmente	Prototipos rápidos, exploración
Desarrollo asistido	Copiloto integrado	Baja: sugerencias inline	Aceleración de escritura de código
Programación agéntica	Loop autónomo supervisado	Media: el agente ejecuta, el humano supervisa	Tareas completas (código + tests + PR)
Multi-agente autónomo	Múltiples agentes interactuando	Alta: agentes revisan a otros agentes	Sistemas completos a escala

Tabla 1 / Espectro de enfoques para el desarrollo de software con LLMs, ordenados por grado creciente de estructura y autonomía del agente.

A medida que aumenta la autonomía del agente, más importante se vuelven la especificación, la arquitectura y las restricciones.

Generación conversacional es el punto de entrada más accesible: interacción conversacional con un chat para generar código. Útil para prototipos, pero sin estructura para proyectos mayores. El *desarrollo asistido* da un paso más: el humano escribe código con copilotos y autocompletado inteligente.

La *programación agéntica* introduce un cambio cualitativo: el agente ejecuta tareas completas (código, tests, *pull requests*) mientras el humano supervisa. El *Ralph Loop* [3], creado por Geoffrey Huntley y nombrado por Ralph Wiggum de Los Simpson (ver Figura 1), ejemplifica este patrón. En su forma más pura, es un bucle bash infinito que reinicia un agente de IA sobre el mismo repositorio. La clave es que el progreso no se almacena en la ventana de contexto del modelo (que se satura y degrada con el uso) sino en los archivos y el historial de Git: cada iteración comienza con contexto fresco, lee el estado actual del disco, ejecuta una tarea, corre los tests, y si todo pasa, hace *commit*. El humano no está *en* el loop sino *sobre* él: diseña el entorno, escribe los tests que actúan como contrapresión, observa los patrones de fallo y ajusta.

El *desarrollo autónomo multi-agente* lleva esta idea al extremo. En OpenAI, los ingenieros interactúan con el sistema casi enteramente a través de prompts, y la revisión de código se delega a otros agentes [1]. La noción de “dirigir” a la IA se difumina cuando múltiples agentes asumen roles de arquitectos, desarrolladores, revisores y especialistas en calidad.

Un patrón emerge a lo largo de este espectro: a medida que aumenta la autonomía del agente, *más importante* se vuelve la especificación, la arquitectura y las restricciones. Como señala Jonathan Aldrich: en este nuevo mundo, “todo el valor está en la ingeniería de software”: requisitos, especificación, arquitectura, diseño, aseguramiento de calidad y gestión [4]. El rol del ingeniero se desplaza aguas arriba en el proceso.

Spec-Driven Development

La autonomía creciente exige especificaciones más rigurosas. El *Spec-Driven Development* (SDD) propone escribir la



Figura 1 / Personaje Ralph Wiggum de Los Simpson.

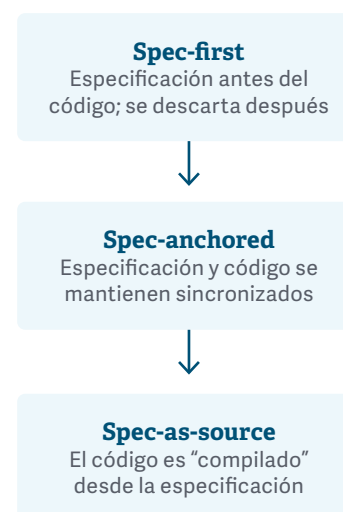


Figura 2 / Variantes del Spec-Driven Development. La flecha indica el grado creciente de acoplamiento entre especificación y código. En *spec-as-source*, el código se trata como artefacto compilado a partir de la especificación.

especificación antes de generar código y usarla como ancla del ciclo de vida del software. Muñoz-Fernández et al. [5] identifican tres variantes progresivas (ver Figura 2).

En *spec-first*, la especificación se escribe antes del código pero puede descartarse después. En *spec-anchored*, se mantiene como referencia viva: los cambios deben reflejarse en ambos artefactos. En *spec-as-source*, el código generado se trata como “compilado”: los cambios se hacen en la especificación y se regenera el código. Esta última variante exige que la especificación sea lo bastante precisa para producir código equivalente si se partiera desde cero.



¿Qué es una buena especificación?

Esta pregunta es central. Una especificación útil para SDD debe ser precisa para que un agente genere código correcto, y a la vez lo bastante abstracta para no convertirse en código disfrazado de prosa. Muchos desarrolladores han convergido en Gherkin, un lenguaje del Behavior-Driven Development (BDD) que expresa requisitos en formato Given-When-Then:

```
Feature: Login de usuario
  Scenario: Login exitoso
    Given el usuario está en la página de login
    When el usuario ingresa credenciales válidas
    Then el usuario es redirigido al dashboard
```

Gherkin es legible para humanos y máquinas, y conecta con frameworks de testing automatizado. Queda abierta la pregunta de si herramientas como tipos, contratos o verificación formal pueden asistir en especificaciones más rigurosas.

Frameworks para SDD

Fowler [6] analiza tres frameworks representativos. GitHub Spec-kit (Microsoft) propone un flujo de cuatro etapas: Constitución → Especificación → Plan → Tareas. La constitución define reglas inmutables; las especificaciones, requisitos funcionales sin detalles técnicos; el plan traduce esos requisitos en decisiones de arquitectura; y las tareas descomponen el plan en unidades implementables. Ofrece comandos en Copilot (/specify, /plan, /task, /implement, /clarify) y solicita aclaraciones ante ambigüedades. Amazon Kiro ofrece un enfoque más liviano, cercano a spec-first, mientras que Tessel explora otras variantes.

SDD en la práctica

He probado SDD y la experiencia es mixta. Para proyectos grandes, donde la complejidad justifica la inversión inicial, funciona bien: la especificación alinea al agente y reduce iteraciones fallidas. Para proyectos pequeños, es burocracia excesiva; se logra lo mismo dando instrucciones cortas al agente e iterando paso a paso.

Hay un problema más sutil. Frameworks como Spec-kit generan múltiples archivos automáticamente: constituciones, especificaciones técnicas, planes, tareas. En teoría, estos documentos guían al agente. En la práctica, uno deja

El contexto es un recurso escaso: un archivo gigantesco eclipsa la tarea y la documentación pertinente. Cuando todo es "importante", nada lo es.

de leer los archivos .md generados y confía en que todo está bien, reproduciendo el mismo patrón que se da con el código: la especificación se vuelve otro artefacto que nadie revisa. La ironía es evidente: SDD promete que la especificación gobierna, pero cuando se automatiza su generación, se pierde la supervisión humana que le da valor.

Harness Engineering

Cuando se delega la escritura de código a agentes autónomos, la especificación deja de ser un documento y se convierte en infraestructura. El equipo de OpenAI lo describió con claridad: su trabajo principal pasó a ser diseñar entornos, especificar intención y construir ciclos de retroalimentación [1]. Esta disciplina, denominada *Harness Engineering* (ingeniería de arnés), fue popularizada por Mitchell Hashimoto y formalizada por OpenAI durante 2025–2026.

La metáfora es deliberada: así como un arnés canaliza la fuerza de un caballo en una dirección útil, el *harness* de software canaliza la capacidad generativa de los LLMs mediante restricciones, documentación y ciclos de retroalimentación. Restringir el espacio de soluciones hace a los agentes más productivos [7]: cuando pueden generar cualquier cosa, desperdician tokens explorando caminos sin salida.

El fracaso del enfoque monolítico

Una lección temprana fue que un gran archivo de instrucciones monolítico (AGENTS.md) fracasa de maneras predecibles [1]. El contexto es un recurso escaso: un archivo gigantesco eclipsa la tarea y la documentación pertinente. Cuando todo es "importante", nada lo es. Un manual monolítico se convierte en un cementerio de reglas caducas que nadie mantiene.

La solución: tratar el AGENTS.md como *tabla de contenidos*, no como enciclopedia. La base de conocimiento se organiza en un directorio docs/ estructurado como fuente de verdad, mientras un AGENTS.md breve (~100 líneas) actúa como mapa [1]. Linters y CI validan que la documentación esté actualizada e interconectada, y un agente recurrente

de “mantenimiento documental” escanea documentación obsoleta y abre PRs de corrección [8].

El arnés importa más que el modelo

La evidencia más directa del valor del harness proviene de LangChain, un popular framework open-source para construir aplicaciones con LLMs. Su agente de programación subió de 52,8% a 66,5% en Terminal Bench 2.0, un benchmark que evalúa la capacidad de agentes para resolver tareas reales de ingeniería de software en entornos de terminal, saltando del puesto 30 al top 5 sin cambiar el modelo subyacente; sólo mejoraron el arnés [7]. La calidad del entorno puede importar más que la capacidad del modelo. La Tabla 2 resume las prácticas claves.

Desafíos abiertos

El SDD y el Harness Engineering ofrecen marcos prometedores, pero están lejos de ser soluciones maduras.

Seguridad

Veracode [9] reporta que los modelos actuales alcanzan corrección sintáctica superior al 95%, pero la tasa de seguridad permanece estancada en ~55%, idéntica a la de hace dos años. Un estudio de repositorios públicos [10] encontró más de 110.000 issues introducidos por IA que persisten sin resolver, con las vulnerabilidades de seguridad como las más persistentes (41,1% sobreviven en la versión más reciente). Lo preocupante es que los desarrolladores tienden a confiar en exceso en el código generado, aceptándolo sin validación [10].

Sin especificaciones que incluyan requisitos de seguridad y restricciones verificables, la productividad puede erosionarse en deuda técnica acumulada.

El otro lado de la moneda es igual de inquietante. Los modelos están alcanzando un nivel de capacidad que genera riesgos propios. Un caso concreto: Claude Mythos Preview, un modelo de Anthropic orientado a ciberseguridad, identificó miles de vulnerabilidades zero-day de alta severidad en sistemas operativos, navegadores y software ampliamente usado, incluyendo una vulnerabilidad de 27 años en OpenBSD y una falla de 16 años en FFmpeg que herramientas automatizadas habían testeado 5 millones de veces sin detectar [11]. Anthropic decidió no lanzarlo al público, optando por desarrollar salvaguardas y un Programa de Verificación Cibernética antes de habilitar acceso más amplio [11]. Que un modelo encuentre vulnerabilidades que sobrevivieron décadas de revisión humana plantea una pregunta incómoda: ¿qué pasa cuando esta capacidad se vuelve accesible sin controles?

¿Es esto Cascada?

Una objeción frecuente al SDD es que se parece a Cascada: definir todo antes de implementar. El contraargumento: Cascada fracasó no porque las especificaciones fueran malas, sino porque el costo de descubrir errores era catastrófico. En SDD con agentes, el ciclo de retroalimentación se reduce a minutos [12]; eso constituye un cambio de categoría. Sin embargo, para trabajo exploratorio, un enfoque más conversacional sigue siendo más efectivo; el espectro de la Sección 2 es un conjunto de herramientas complementarias, no una jerarquía.

Práctica	Descripción
Ingeniería de contexto	AGENTS.md como tabla de contenidos; docs/ estructurado como fuente de verdad; documentación versionada y validada mecánicamente.
Restricciones arquitectónicas	Capas de dependencia estrictas (Types → Config → Service → UI); validadas por tests estructurales y CI.
Ciclos de retroalimentación	Compilador, suite de tests, telemetría y observabilidad como señales para que el agente se autocorrija iterativamente.
Revisión agente-a-agente	Agentes que revisan código de otros agentes, eliminando el cuello de botella humano en la revisión de código.
Mantenimiento documental	Agentes recurrentes que escanean documentación obsoleta y abren PRs de corrección automáticamente.

Tabla 2 / Prácticas clave del Harness Engineering, adaptado de las experiencias de OpenAI [1], LangChain [7] y Anthropic [8].



Restringir el espacio de soluciones hace a los agentes más productivos: cuando pueden generar cualquier cosa, desperdician tokens explorando caminos sin salida.

El problema de la confianza excesiva

Desde la docencia, veo un fenómeno que ilustra un riesgo transversal a todos estos enfoques. En las entregas de tareas realizadas con IA, muchos estudiantes ni siquiera revisan el código que entregan. En una ocasión, recibí una entrega que parecía correcta a primera vista: usaba construcciones avanzadas que hacían sentido superficial. Al revisarla con más cuidado, todo estaba incorrecto: el modelo había alucinado con coherencia. Cuando cité al estudiante a explicar su código, me dijo que había usado IA. No podía explicar una sola línea...

Este patrón no es exclusivo de estudiantes. La tendencia a confiar en el código generado sin revisarlo es una forma de sesgo de automatización, y se intensifica con cada mejora en la calidad de los modelos. Paradójicamente, cuanto mejor se vuelve la IA, más peligroso se vuelve dejar de supervisarla.

Desafíos transversales

A estos problemas se suman la memoria (cada sesión de un agente comienza sin recuerdo de las anteriores [8]), la economía de tokens (los mejores frameworks serán los que produzcan buen software con menos tokens), y la ingeniería de contexto, es decir, gestionar qué información ve el agente en cada momento. Especificaciones sin contexto adecuado producen implementaciones incoherentes; contexto sin especificaciones produce actividad sin dirección [13].

Implicaciones para la ingeniería y la academia

Esta transformación tiene consecuencias directas sobre el perfil profesional. Como observa Aldrich [4], muchas posiciones de “Ingeniero de Software” en la práctica involucran sólo programación. Ahora, el ingeniero planifica y supervisa agentes, lo que exige expertise en ciencias de la computación para revisar y dirigir agentes, habilidades en gestión de agentes, y conocimiento profundo de ingeniería de software: requisitos, especificación, arquitectura, QA y gestión de proyectos.

Para la academia, las preguntas son concretas. ¿Cuál es el conocimiento mínimo para ser *efectivo* en cualquier tarea de computación con estas herramientas? ¿Puede un estudiante construir un compilador con IA sin cursar lenguajes de pro-

gramación? Politz propone que las habilidades fundamentales persisten (definir datos, escribir ejemplos, dar firmas, escribir tests) aunque ahora el cuerpo de la función lo rellena la IA [14]. La disciplina se redistribuye en vez de desaparecer.

En la práctica, esto sugiere al menos dos dimensiones formativas distintas: usar LLMs como componente (la IA como API para construir software que sería difícil de desarrollar sin ella) y usar agentes para construir software (diseño de prompts, SDD, ciclos de retroalimentación). Ambas requieren habilidades complementarias, y ambas están en evolución.

Reflexión final

Creo que los agentes van a escribir la mayoría del código en producción. No todo (hay código especializado que necesita ser creado o supervisado de cerca por humanos) pero sí la mayor parte. La historia de la ingeniería de software es la historia de encontrar mejores formas de expresar intención y verificar resultados. Los LLMs aceleran esa trayectoria.

Me preocupa el acceso. Estas herramientas son costosas, y el costo puede dispararse a medida que los modelos se vuelven más capaces. Si eso ocurre, el riesgo es que el desarrollo de software asistido por IA se convierta en tecnología de privilegiados. Democratizar estas herramientas es urgente para evitar una nueva brecha digital.

Hay también un lado positivo que vale la pena destacar: más personas, incluyendo quienes antes no podían, van a poder desarrollar software. Eso amplía quién participa en la construcción del mundo digital, y es algo valioso.

¿Seguirán vigentes en seis meses técnicas como el Ralph Loop o el SDD? Los principios subyacentes (especificación rigurosa, restricciones mecánicas, ciclos de retroalimentación) seguirán siendo válidos, especialmente con modelos más pequeños o de código abierto. Quien domine la especificación y el diseño de entornos para agentes dominará el desarrollo de software en los años por venir. **B**

Referencias

- [1] OpenAI, “Harness engineering: leveraging Codex in an agent-first world”, OpenAI Blog, febrero 2026.
- [2] N. Carlini, “Building a C compiler with a team of parallel Claudes”, Anthropic Engineering Blog, febrero 2026.
- [3] G. Huntley, “The Ralph Loop”, ghuntley.com, 2025.
- [4] J. Aldrich, “It’s Software Engineering — Really!”, presentación, 2025.
- [5] A. Muñoz-Fernández et al., “SDD: Spec-Driven Development with AI Agents”.
- [6] M. Fowler, “Exploring Gen AI: SDD Tools”, martinofowler.com, 2026.
- [7] M. Fowler, “Harness Engineering”, martinofowler.com, febrero 2026.
- [8] Anthropic, documentación sobre harnesses para agentes de larga ejecución, 2026.
- [9] Veracode, “Spring 2026 GenAI Code Security Update”, Veracode Blog, marzo 2026.
- [10] Y. Liu et al., “Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild”, 2026.
- [11] Anthropic, “Glasswing: Claude Mythos Preview”, anthropic.com, 2026.
- [12] A. Cloudstar, “Spec-Driven Development 2026: Future of AI Coding or Waterfall?”, alexcloudstar.com, marzo 2026.
- [13] WeBuild-AI, “Aligning Spec-Driven Development and Context Engineering for 2026”, webuild-ai.com, febrero 2026.
- [14] J. G. Politz, “Skills for AI-assisted programming”, presentación, 2025.