CN

Versión definitiva

Founder-Led Engineering Ops

Sistema operativo mínimo para construir software serio
spec-driven, agent-orchestrated y gobernado por calidad

Founder. Builder. Operator.

Manual operativo atemporal con instantánea de stack — abril de 2026

Carlos Navarro Cabrera
0xnavarro.com

- Spec-driven development
- Implementación con modelos frontera
- Auditorías y QA estructurados
- Autoridad final humana

Resumen ejecutivo

La inteligencia artificial ha reducido de forma radical el coste de producir código, pero no ha resuelto el problema de la ingeniería de software. Lo que hoy suele presentarse como rapidez —prompting, vibe coding o generación instantánea de features— solo produce artefactos. El software serio sigue exigiendo especificación, arquitectura, trazabilidad, verificación y criterio. La diferencia es que ahora una parte creciente del trabajo la ejecutan agentes, no personas.

Este documento describe el sistema mínimo que utilizo actualmente para desarrollar software de forma profesional en un contexto founder-led. Es un sistema spec-driven, agent-orchestrated y gobernado por calidad. Parte de una convicción simple: el software serio no se improvisa; se especifica, se ejecuta y se verifica.

En la práctica, eso implica trabajar con especificaciones formales, separar diseño de ejecución, dividir el desarrollo en fases y ramas, asignar roles claros a agentes distintos, auditar con varias familias de modelos y mantener una autoridad humana final sobre intención, seguridad, riesgo y release. El objetivo no es maximizar output bruto, sino throughput útil con control.

La instantánea actual de stack, a fecha de abril de 2026, utiliza Kiro IDE y Kiro CLI como entorno operativo; Claude Opus 4.6 como modelo principal de implementación y revisión iterativa; GPT-5.3-Codex con razonamiento extremadamente alto como auditor final; Playwright MCP para QA de navegador; y un conjunto de MCPs y skills específicas del proyecto para base de datos, observabilidad, seguridad y automatización contextual.

FOUNDER-LED ENGINEERING OPS

1. Por qué hace falta un nuevo sistema operativo de ingeniería

Durante 2025 el mercado popularizó el vibe coding: la idea de que bastaba con describir algo de forma informal para que la IA produjera software útil. Ese enfoque puede servir para prototipos, demos o pruebas rápidas, pero falla en cuanto aparece complejidad real: dependencias, arquitectura, edge cases, seguridad, QA, cambios de alcance, trazabilidad o mantenimiento.

El problema de fondo es que mucha gente ha confundido el output de un modelo con ingeniería. Generar código no equivale a construir software. Un modelo puede escribir funciones, pantallas, migraciones o tests, pero eso no significa que el sistema resultante esté alineado con un problema de negocio, con una arquitectura coherente o con un estándar de calidad aceptable.

La respuesta no es dejar de usar IA, sino reconstruir la disciplina. Si antes la ingeniería de software se apoyaba en especificaciones, revisiones, QA, ownership y releases gobernados, ahora hace falta recuperar esos principios y adaptarlos a un contexto agéntico. En lugar de empleados con funciones distintas, hoy podemos tener agentes con contextos, responsabilidades y misiones separadas. Lo que no desaparece es la necesidad de gobernanza.

Por eso este modelo no intenta romantizar la improvisación. Intenta diseñar un sistema de ejecución donde la IA acelera el trabajo sin destruir el proceso. La promesa no es producir más líneas de código; es construir sistemas fiables con velocidad, evidencia y control.

> *«El software serio no se improvisa; se especifica, se ejecuta y se verifica.»*

2. Tesis central

Mi tesis es que la era agéntica no elimina la ingeniería de software; la hace más importante. Cuando producir código se abarata, el cuello de botella se desplaza hacia la claridad de la especificación, la calidad del diseño, la corrección de la ejecución, la robustez de la verificación y la calidad del juicio final.

De ahí salen tres principios. Primero: el software serio no se improvisa; se especifica, se ejecuta y se verifica. Segundo: los agentes sustituyen trabajo, no arquitectura, criterio ni intención de negocio. Tercero: los modelos también tienen sesgos, así que un sistema robusto necesita más de una familia de modelos para reducir puntos ciegos.

Esta tesis cambia la pregunta habitual. Ya no se trata de “qué prompt produce más código”, sino de “qué sistema convierte una visión de negocio en software correcto, verificable y desplegable”. Esa es la diferencia entre productividad aparente y construcción real.

3. Del problema de negocio a la rama de trabajo

El sistema empieza antes del código. Empieza en la definición del problema, del usuario, de la fricción operativa, de la oportunidad de negocio y del estándar de calidad exigido. Si la intención original es borrosa, el resto del proceso se contamina.

La primera capa es la especificación. Para que una especificación sea profesional no basta con que tenga títulos correctos. Tiene que reducir la ambigüedad de tal forma que el diseño y la ejecución posteriores no necesiten correcciones estructurales continuas. Debe recoger requisitos funcionales y no funcionales, actores, user stories, use cases, flujos principales y alternativos, constraints, criterios de aceptación, casos límite y dependencias externas.

La segunda capa es el diseño. Aquí se define la arquitectura, se fijan los límites del sistema, se documentan decisiones, se registran ADRs cuando hace falta y se establece trazabilidad entre requisitos y componentes. La arquitectura y las decisiones de negocio deben actuar como source of truth. El código llega después y debe reflejarlas, no sustituirlas.

La tercera capa es el plan de implementación. El plan traduce diseño y requisitos en fases, tareas, subtareas y criterios de cierre. En mi flujo actual, cada fase relevante se ejecuta en su propia rama. Cuando una rama se considera cerrada no es porque “ya funciona”, sino porque ya pasó por implementación, auditoría, QA, revisión final y criterio de merge.

Este encadenamiento —requirements, design, implementation plan, phase branches— es lo que hace posible auditar, corregir y escalar. Sin esa secuencia, el sistema vuelve a convertirse en improvisación asistida por IA.

FOUNDER-LED ENGINEERING OPS

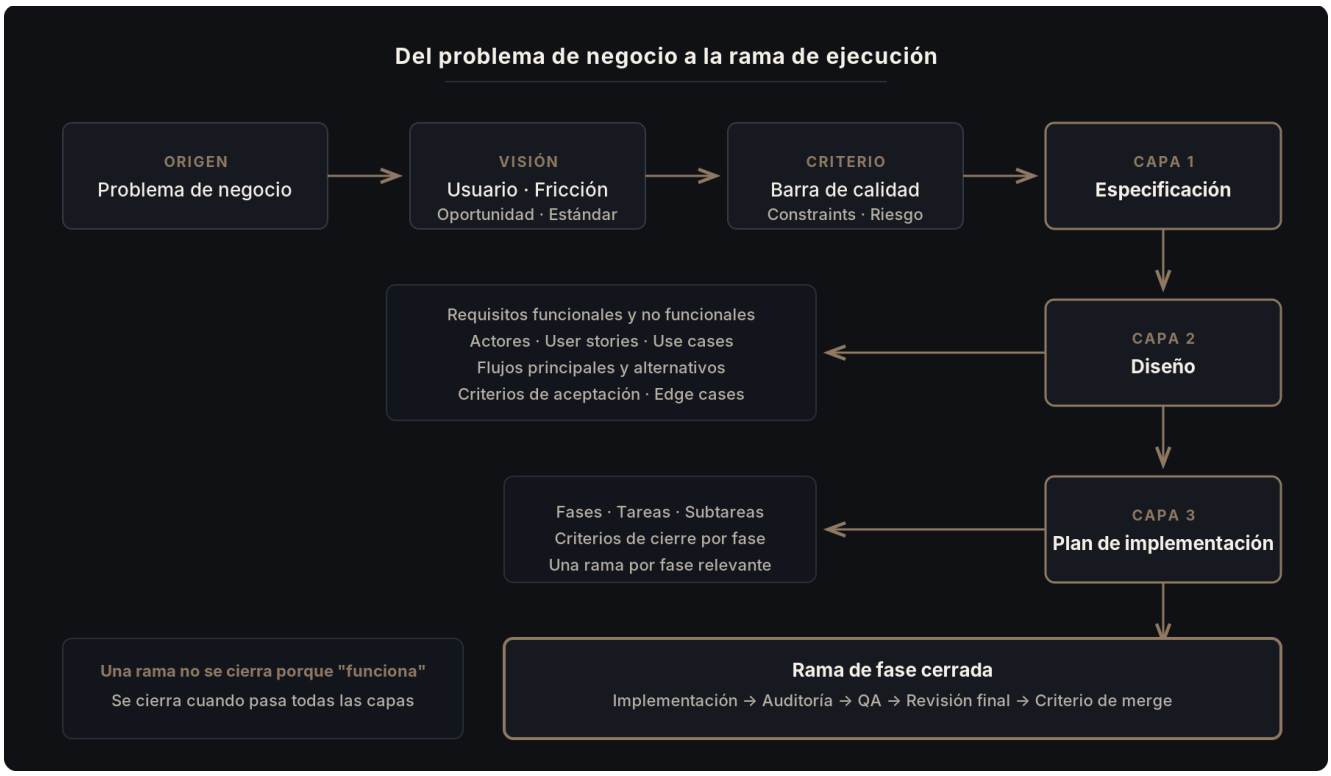

Diagrama 1. Del problema de negocio a la rama de ejecución en un flujo spec-driven

4. Roles del sistema actual

El sistema mínimo ideal que utilizo hoy tiene dos capas de roles: un rol humano de orquestación y un conjunto de roles agénticos especializados. El orquestador humano decide qué negocio se construye, qué problema merece resolverse, qué arquitectura es aceptable, qué riesgos se toleran y cuándo una versión está realmente lista. Ese rol sigue siendo central porque la responsabilidad final, la visión empresarial y el criterio de aceptación siguen siendo humanos.

A su alrededor trabajan varios agentes. El agente implementador es quien ejecuta la rama y modifica el código. El agente auditor técnico revisa la implementación de forma iterativa, detecta findings, evalúa coherencia con el diseño y busca regresiones o desajustes respecto a la especificación. El agente de QA recorre flujos reales y valida comportamiento observable, no solo estructura interna. El auditor final vuelve a revisar la rama desde otro prisma y con otra familia de modelos, con foco en merge readiness y riesgos residuales.

A veces añado otros agentes o subagentes para tareas concretas, pero estos forman el stack mínimo. Lo importante no es la cantidad de agentes; es que cada uno tenga misión, contexto y ownership claros. Separar sesiones por rol evita que el mismo modelo se autojustifique dentro del mismo contexto y mejora la calidad de la verificación.

Además, en Kiro AI trabajo también con subagentes lanzados por el propio agente principal. Esto es especialmente útil en auditoría. El auditor no actúa como una única voz monolítica, sino como una capa de coordinación que decide qué subagentes abrir según el diff de la rama, el

riesgo, el área tocada y la naturaleza del cambio. Unos se centran en arquitectura y trazabilidad con el design; otros en seguridad, regresión o spec mismatch. El agente auditor principal consolida después esos hallazgos y los devuelve como findings deduplicados y priorizados.

Diagrama 2. Flujo actual con orquestación humana, agentes especializados y subagentes de auditoría

SNAPSHOT

Mapa mínimo de roles

Rol	Misión primaria	Motivo de separación
Orquestador humano	Define intención, arquitectura aceptable, riesgo y criterio de release.	Preserva visión empresarial, responsabilidad y juicio final.
Agente implementador	Ejecuta la fase y modifica el código de la rama.	Concentra ownership de ejecución y evita edición difusa.
Auditor técnico	Genera findings iterativos y busca incoherencias.	Introduce otra misión y otro prisma dentro de la misma familia de trabajo.
Agente QA	Valida comportamiento real con navegador y evidencia.	Comprueba experiencia observable, no solo estructura de código.
Auditor final	Evalúa merge readiness y riesgos	Usa otra familia de modelos para

	residuales.	reducir sesgos y puntos ciegos.

FOUNDER-LED ENGINEERING OPS

5. Por qué el sistema usa modelos frontera

A día de hoy, el coste aparente de usar modelos más baratos suele ser inferior al coste real que provocan sus errores. Un modelo peor no solo comete fallos más graves; también arrastra errores a fases posteriores, multiplica auditorías, obliga a rehacer trabajo y puede elevar el riesgo de que algo defectuoso alcance producción.

Por eso mi principio operativo es utilizar modelos frontera siempre que el presupuesto lo permita. No se trata de lujo; se trata de coste total de corrección. En el estado actual del mercado, pagar por razonamiento y calidad suele ser más barato que pagar retrabajo.

Este sistema no está optimizado para minimizar el gasto en tokens, sino para maximizar la calidad del software. Una parte importante del coste no se va en generar código, sino en revisar, auditar, hacer QA y verificar.

En la instantánea de abril de 2026, Opus 4.6 es el modelo principal de implementación y revisión iterativa. Es rápido, consistente y suficientemente profundo para seguir especificaciones complejas, ejecutar planes por fases y mantener continuidad operativa. GPT-5.3-Codex, con razonamiento extremadamente alto, actúa como auditor final. Lo utilizo porque sistemáticamente detecta hallazgos que el otro modelo no ve. Esa divergencia no es un defecto del flujo; es precisamente la razón para trabajar con dos familias de modelos distintas.

El principio no es solo usar el mejor modelo disponible. Es combinar al menos dos familias para reducir sesgos de entrenamiento, sesgos de contexto y puntos ciegos recurrentes. La calidad del sistema no viene de un modelo mágico, sino de la interacción gobernada entre varios.

CURRENT STACK SNAPSHOT — APRIL 2026

Instantánea actual del stack

Capa	Herramienta	Motivo
IDE	Kiro IDE	Spec-driven development; ejecución principal por fase.
CLI	Kiro CLI	Auditoría, QA, sesiones aisladas y control operacional.
Modelo principal	Claude Opus 4.6	Implementación y revisión iterativa.
Auditor final	GPT-5.3-Codex	Razonamiento extremadamente alto; revisión final pre-merge.
QA browser	Playwright MCP	Flujos reales, evidencia funcional y regressions.
Skills	Project-specific	Skills específicas del proyecto; no

		plantillas genéricas sin contexto.
MCPs	DB / observability / security	Herramientas instrumentales según dominio y entorno.

6. Rationale de herramientas

Kiro IDE forma parte del sistema porque su estructura está alineada con spec-driven development. Permite trabajar con requirements, design y tasks como artefactos de primer nivel, en lugar de tratarlos como notas sueltas o prompts desordenados. En mi práctica, esa alineación es más importante que cualquier feature aislada.

En esa capa instrumental, Kiro AI también me permite lanzar subagentes desde el agente principal. Para auditoría esto es importante porque el sistema puede decidir dinámicamente qué miradas especializadas necesita en función de los cambios realizados: arquitectura, seguridad, contratos, regresión, trazabilidad con la spec o coherencia con el design. La sesión visible sigue siendo una, pero la revisión real puede venir de una topología de subagentes que luego se consolida en una única salida accionable.

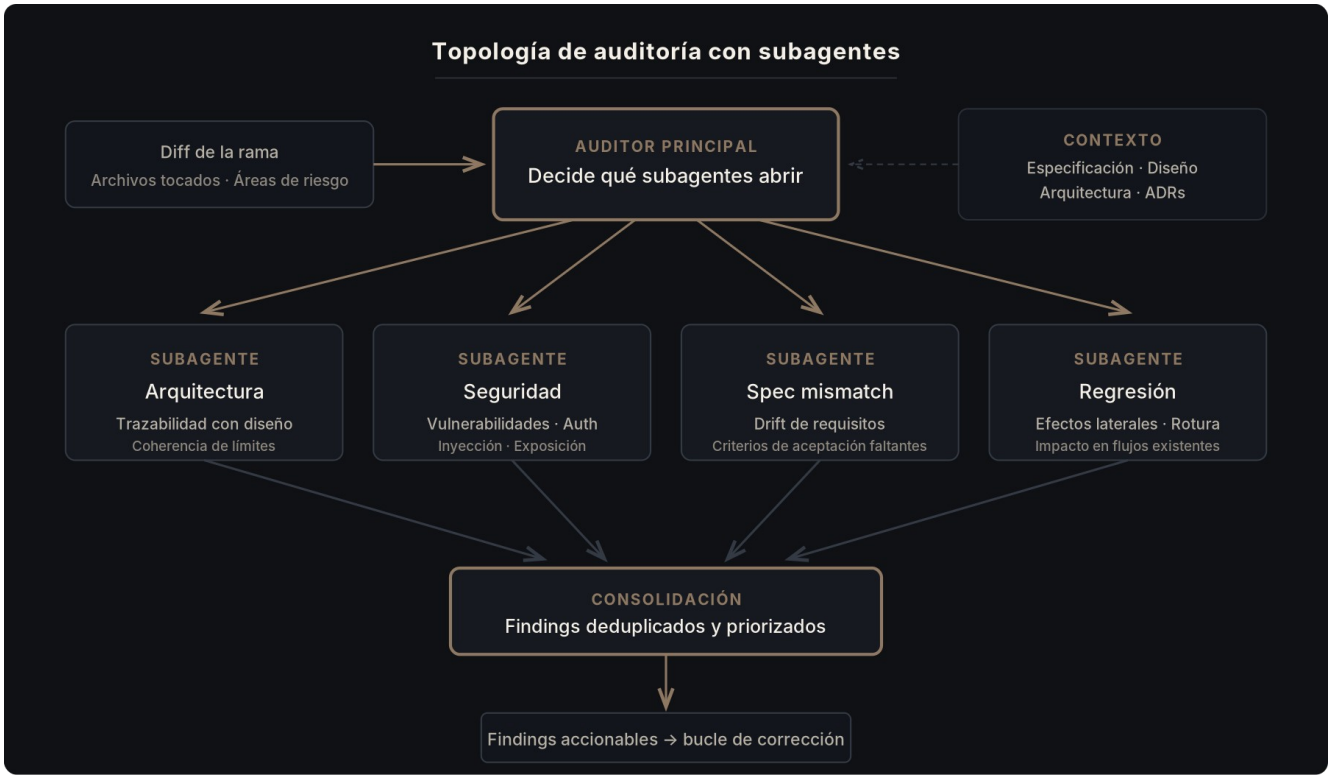

Diagrama 3. Topología de auditoría con subagentes para arquitectura, seguridad, spec mismatch y regresión

Kiro CLI cumple otro papel. Lo utilizo para sesiones más aisladas y operativas: auditoría, QA, revisiones específicas y tareas donde conviene separar contexto del desarrollo activo. También facilita un control más limpio de costes, sesiones y setup. El IDE y el CLI no se usan por moda; se usan porque soportan papeles distintos dentro del sistema.

Playwright MCP es la pieza crítica del QA de navegador. No se trata de simular calidad, sino de interactuar con la aplicación como lo haría un usuario, recorrer flujos, observar errores reales y producir evidencia. Además, el sistema puede incorporar otros MCPs para base de datos, observabilidad, seguridad, dependencias, logs o infraestructura. No son accesorios: forman la capa instrumental que conecta la ejecución agéntica con el entorno real.

Las skills también son importantes, pero no como plantillas genéricas intercambiables. En mi flujo, las skills valiosas son las adaptadas al proyecto, al dominio y al estándar de calidad que quiero imponer. La reutilización no sirve si destruye contexto o rebaja exigencia.

Eso también cambia la economía de la capa instrumental. En flujos de software serios, una parte relevante del valor no está en la generación inicial de código, sino en la especificación, la revisión, la auditoría, el QA y la verificación estructurada. Los MCPs, las skills y las sesiones separadas no son coste accesorio: son infraestructura de calidad.

FOUNDER-LED ENGINEERING OPS

7. Gobernanza de calidad

La calidad no se delega a la esperanza. Se gobierna. En este sistema, la unidad formal de control es el finding: una observación estructurada, trazable y accionable que nace de auditoría, QA o revisión final. Un finding debe tener severidad, evidencia, referencia al área afectada, recomendación y estado. Si no puede convertirse en hallazgo formal, probablemente no es lo bastante claro.

La regla central es que arreglado no significa cerrado. Un finding solo se cierra cuando un rol distinto verifica el resultado. Esto obliga a mantener separación entre ejecución y validación, y evita que el sistema se autoapruebe.

La otra regla importante es que la arquitectura y las decisiones previas son la fuente de verdad. Si el código contradice la especificación o el diseño, el código está mal aunque compile. La IA genera output rápido; precisamente por eso hace falta una capa fuerte de gobernanza que filtre, compare y descarte.

Aquí el QA no llega como maquillaje tardío. Llega como capa funcional obligatoria. Y la auditoría final no existe para repetir lo mismo, sino para introducir otra perspectiva de revisión antes del merge. Ese cierre es lo que da confianza operativa.

FOUNDER-LED ENGINEERING OPS

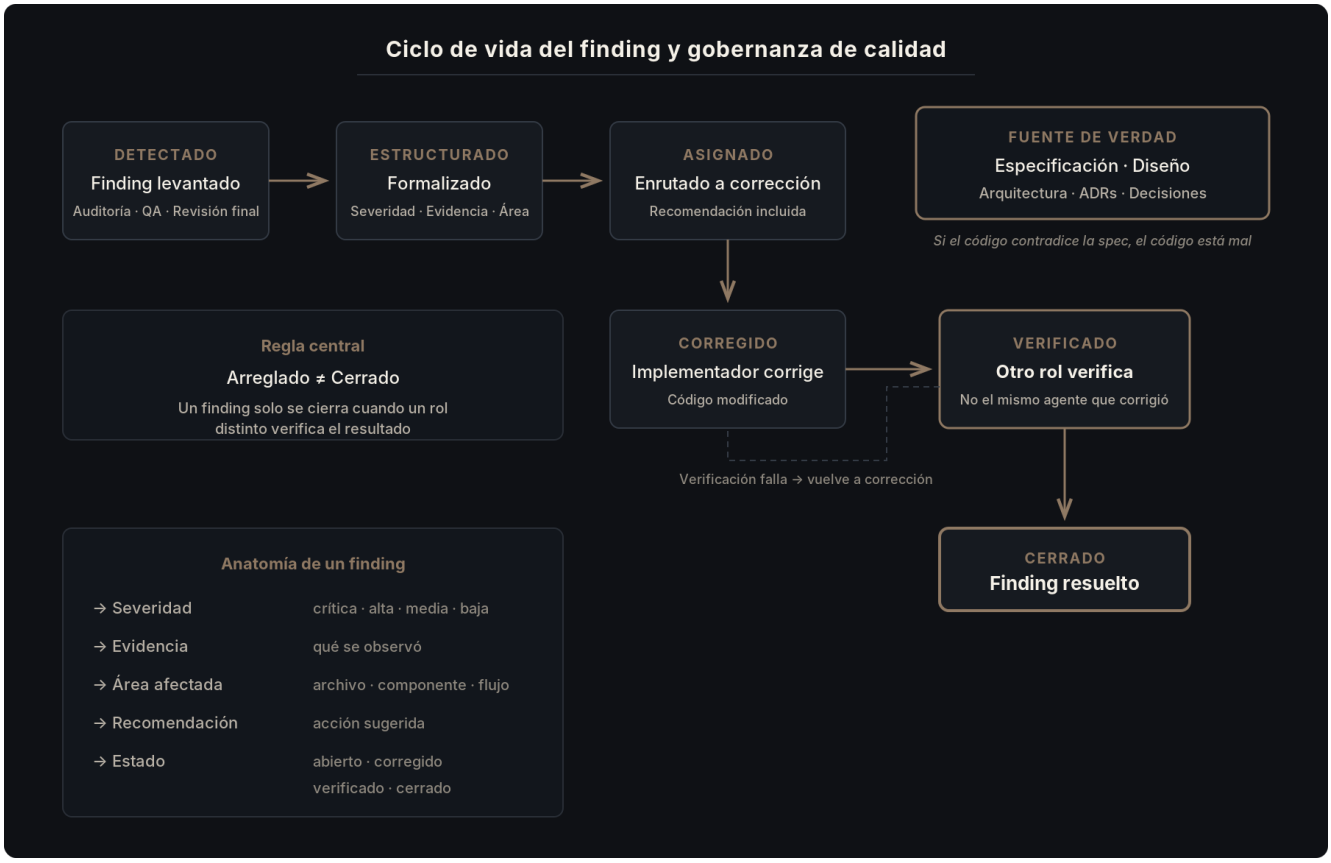

Diagrama 4. Ciclo de vida del finding y gobernanza de calidad

8. Instantánea del flujo actual

El flujo actual funciona como un bucle gobernado. El orquestador humano define intención, constraints y criterio. A partir de ahí se generan especificaciones, diseño y plan de implementación. Cada fase se ejecuta en una rama. El agente implementador desarrolla; el auditor técnico revisa; el implementador corrige; el QA valida flujos reales; y un auditor final, sobre otra familia de modelos, emite el último veredicto antes de PR o merge.

No es un sistema diseñado para maximizar autonomía simbólica. Es un sistema diseñado para maximizar fiabilidad práctica. En el estado actual de la tecnología, prefiero un flujo con varias capas de verificación y autoridad humana final a una supuesta autonomía total sin garantías.

FOUNDER-LED ENGINEERING OPS

9. Hacia un orquestador agéntico

El siguiente paso natural del sistema es reducir parte de la carga manual de coordinación. Eso no significa eliminar lo humano, sino automatizar el plano operativo inferior. El orquestador agéntico del siguiente nivel no sustituiría a los demás agentes: se sentaría en el centro y coordinaría exactamente al mismo agente implementador, al auditor técnico, al agente de QA y al auditor final.

Ese orquestador consumiría visión de negocio, requirements aprobados, design, implementation plan, arquitectura objetivo, políticas de calidad y criterios de riesgo ya definidos por el humano. A partir de ahí, movería el workflow entre etapas, consolidaría findings, gestionaría gates y

escalaría dudas o decisiones sensibles al humano cuando hiciera falta. El cambio está en quién coordina el sistema, no en eliminar la fase inicial de especificación ni el juicio final humano.

En otras palabras: la automatización futura más interesante no es una IA que “escriba más código”. Es una capa de orquestación que convierta intención empresarial en ejecución gobernada de software. Ese es el paso de una microfábrica operada por un founder a un sistema operativo de ingeniería parcialmente automatizado.

FOUNDER-LED ENGINEERING OPS

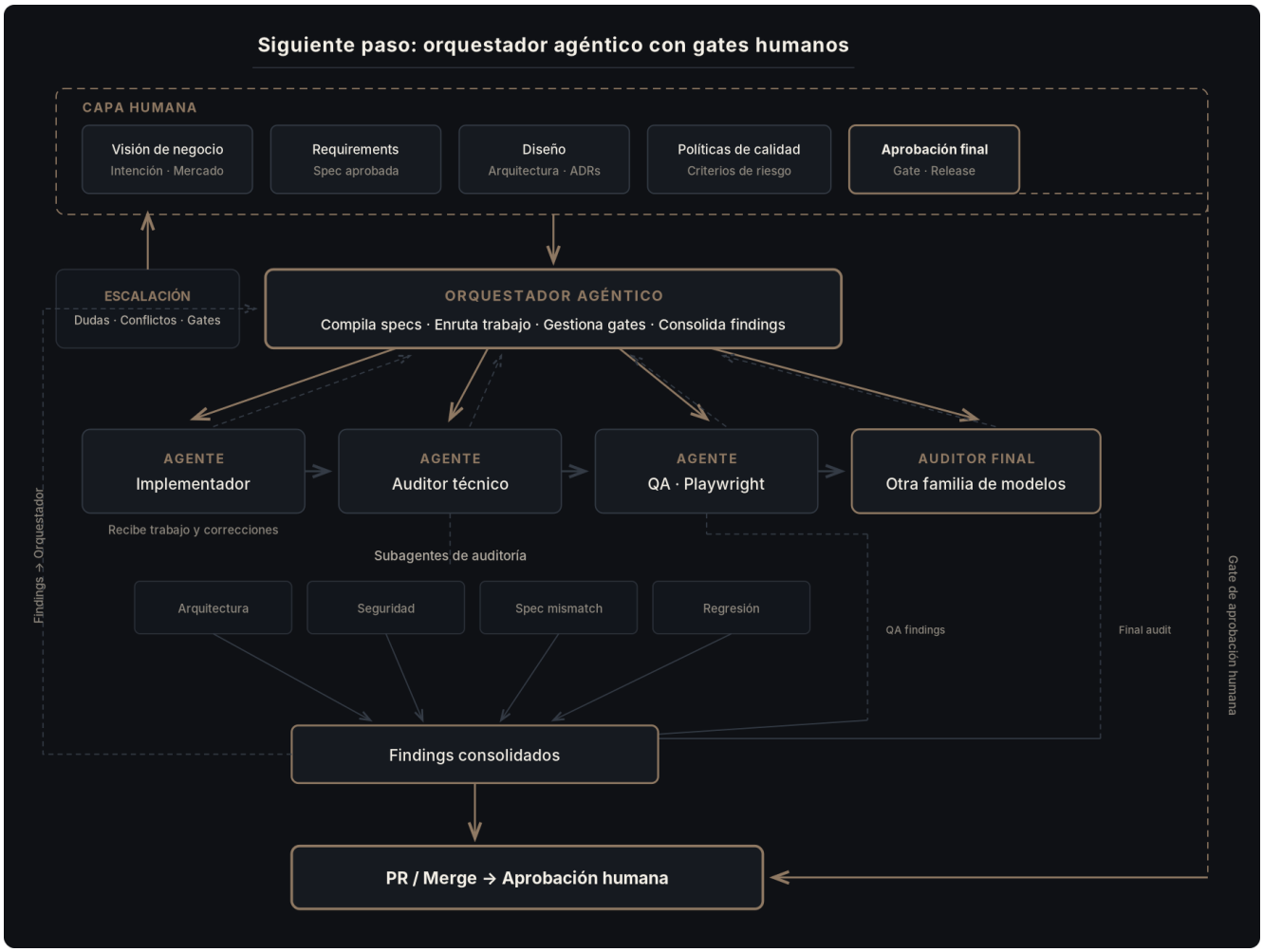

Diagrama 5. Siguiente paso con orquestador agéntico y gates humanos sobre el mismo stack de agentes

10. Conclusión operativa

Este sistema no nace para producir más código, sino para producir mejor software. Su objetivo es transformar modelos frontera y ejecución agéntica en una disciplina operativa: especificar antes de implementar, separar roles, verificar con varias capas y preservar autoridad humana sobre arquitectura, calidad, seguridad y release.

El resultado no es simplemente velocidad. Es velocidad con trazabilidad, auditoría, QA, evidencia y control. Esa diferencia es la que permite construir compañías reales sobre software generado con IA, en lugar de acumular prototipos frágiles.

Por eso el sistema no está optimizado para la factura mínima de inferencia, sino para el mejor coste total de calidad posible: menos retrabajo, menos regresiones, menos deuda y más software verificable listo para operar.

11. Tesis de ejecución

La IA no ha eliminado la ingeniería de software; ha hecho más urgente su formalización. El error del mercado ha sido confundir output con construcción. Generar código no es construir software. Construir software exige especificación profesional, arquitectura como fuente de verdad, ejecución por fases, auditoría estructurada, QA real y gobernanza clara.

Los agentes sustituyen trabajo operativo, no criterio. Los modelos aceleran la ejecución, pero no reemplazan la intención, el diseño del negocio, la seguridad, el juicio sobre riesgo ni la responsabilidad final. Por eso el sistema correcto no es una cadena de prompts; es una organización mínima de ingeniería gobernada.

12. Visión empresarial y manifiesto

En el nuevo paradigma ganarán las compañías y los founders capaces de convertir intención en sistemas con más claridad, más velocidad y mejor control que el resto. El software tenderá a abaratarse como capacidad de producción, pero la ventaja competitiva seguirá estando en decidir qué construir, para quién, con qué arquitectura, con qué calidad y con qué timing de mercado.

Eso cambia el papel del founder técnico. Ya no es solo quien escribe más código a mano. Es quien formula el problema correcto, fija el estándar, diseña la arquitectura, decide el riesgo aceptable y gobierna una fábrica de ejecución aumentada por agentes. La ventaja no está solo en programar; está en convertir visión empresarial en sistemas fiables antes que otros.

A tres o cinco años muchas capas del flujo podrán automatizarse más. Pero la intención de negocio, la comprensión del mercado, la lectura de señales aún no absorbidas por los modelos, la decisión sobre seguridad y el juicio de aceptación seguirán siendo profundamente humanos. La IA puede abaratar el software; no puede sustituir la responsabilidad del fundador ni la interpretación del mundo.

Por eso este documento no es una defensa del hype. Es una defensa de la disciplina. En una era donde crear código será cada vez más commodity, la diferencia la marcarán quienes sepan usarlo para construir empresas, no solo demos.

13. Principios finales

1. El software serio no se improvisa; se especifica, se ejecuta y se verifica.
2. Generar código no es lo mismo que construir software.
3. Los agentes sustituyen trabajo, no arquitectura, criterio ni intención de negocio.
4. La arquitectura y las decisiones son la fuente de verdad; el código debe reflejarlas.
5. Para software serio, los modelos frontera deben ser el estándar por defecto.
6. Una sola familia de modelos no basta: varias familias reducen sesgos y puntos ciegos.
7. La gobernanza humana sigue siendo irrenunciable en calidad, seguridad y release.
8. La velocidad útil no es output bruto; es throughput verificable con control.
9. El founder técnico del futuro será, sobre todo, arquitecto, operador y decisor de sistemas.