SPEC-DRIVEN DEVELOPMENT

Preguntas y respuestas

01

¿Se llega con los requisitos levantados, o se empieza vago? ¿Cómo evitar que se vuelva vibe coding con artefactos?

No necesitas llegar con los requisitos levantados. Ese es justamente el punto de SDD: el proceso los levanta. Lo que sí necesitas es llegar con la intención clara, o sea saber qué problema resuelves, para quién, y cómo se ve que quedó listo.

La distinción fina es esta. Estar vago sobre la solución está bien, incluso es sano, porque no conviene predecidir la tecnología. Estar vago sobre el problema es fatal, porque ahí no hay proceso que te salve. La fase de especificación, y sobre todo la de clarificación, son la máquina que convierte una idea difusa en requisitos precisos.

Lo de que se vuelva vibe coding con artefactos es la trampa más real de todas. Pasa cuando tratas los archivos como papeleo, aceptas lo que el agente escriba sin cuestionarlo, y terminas con specs bonitas que nadie razonó. Es la misma deriva de siempre, pero con pasos de más.

El antídoto es uno solo: en cada compuerta, el cuello de botella tienes que ser tú. Si lees la spec y la apruebas sin pelear nada, es teatro. La prueba ácida es esta: si no puedes explicar por qué una decisión quedó en el plan, no hiciste SDD, lo hizo el agente y tú miraste. El valor no son los documentos, es el pensamiento que los documentos te obligan a hacer.

02

Además de apps, ¿en qué otros escenarios sirve? ¿Se podría aplicar a repositorios de infraestructura?

SDD no está casado con las apps. Sirve donde haya un agente generando algo y una intención que debe preservarse y verificarse.

Infraestructura como código es de los mejores casos, incluso mejor que una interfaz gráfica. El estado deseado se describe de forma declarativa, que ya es casi una spec, y los criterios de aceptación son muy comprobables. Se vería así: la constitución con tus no-negociables (nada de buckets públicos, todo por módulos, política de etiquetas, límites de costo), la spec con la capacidad que quieres sin decir todavía si es Terraform o Pulumi, la clarificación preguntando por región, redes, cifrado y retención, el plan ya con la tecnología, y la verificación apoyada en el plan de cambios más políticas como `Checkov` o `Conftest`. Ahí el prueba primero es el plan y las políticas, y el revisor separado son las validaciones de política como código.

Fuera de infra también encaja bien en pipelines de datos, en diseño de APIs donde el contrato es la spec, en scripts y automatizaciones, y hasta en cosas que no son código.

La regla para saber si vale la pena es simple: SDD sirve cuando equivocarse sale caro y cuando el listo se puede definir como un estado que se puede comprobar. Para explorar o para algo desechable, estorba.

03

¿Cómo lograr que el análisis de necesidades apunte a lo que quiero y encuentre vacíos? Por ejemplo, que en el café se cuestione la moneda base, la normativa contable, el cierre Z, el cierre X o abrir caja.

Esta es la pregunta más fina de todas, porque una clarificación genérica jamás va a preguntar por el cierre Z si nada se lo sugiere.

La clave incómoda es esta: el agente solo encuentra los vacíos para los que tiene contexto. Sin más, va a preguntar lo genérico, como errores, casos borde y validación, pero no va a saber que un punto de venta tiene normas contables a menos que le des ese lente. El conocimiento del dominio es una entrada, no algo que el agente adivine. Hay tres palancas, de mayor a menor impacto.

Primera: mete el dominio en la constitución o el steering. Si ahí dice que esto es un punto de venta que cumple prácticas contables de un café en Colombia, con arqueo de caja, moneda base y manejo de IVA, la clarificación va a jalar de esos hilos. El vacío que quieres que encuentre necesita un gancho en el contexto.

Segunda, y es la de más rendimiento: pídele al agente que clarifique poniéndose otra piel. Algo como: revisa esta spec como lo haría un contador y un cajero con veinte años de experiencia, ¿qué falta que un desarrollador no preguntaría? Cambias la persona que revisa y caen otros vacíos. Puedes hacer varias pasadas, como contador, cajero, dueño y auditor, y de ahí salen la moneda base, el IVA, el cierre Z y el X, la base inicial de caja, las propinas, y la diferencia entre anular y devolver.

Tercera: siembra tú los vacíos que ya sospechas. No dependas del agente para todo. La clarificación es una red de seguridad, no un oráculo.

Y ojo con una confusión común: el análisis de consistencia (el `/analyze` de spec-kit) revisa que la spec, el plan y las tareas no se contradigan entre sí, pero no valida contra la realidad del negocio. Eso último solo lo da el dominio, el tuyo o el que simules.

04

¿Cómo asegurar que cada spec tenga una revisión tipo pentesting tras su implementación?

Esto encaja redondo con SDD, porque ya tienes las piezas: el revisor separado y las compuertas. Solo hay que volver la revisión de seguridad una fase con su propio artefacto. Cuatro capas.

En la constitución dejas la seguridad como no-negociable: validación en las fronteras de confianza, nada de secretos en el código, autorización en cada acceso, mínimo privilegio. Así cada spec la hereda.

En la spec agregas criterios de aceptación de seguridad en formato EARS. Por ejemplo: si la entrada del cajero trae un total negativo, entonces el sistema deberá rechazarla. Con eso la seguridad pasa a ser comprobable y no un deseo.

Después de implementar corres una sesión aparte cuyo único trabajo es atacar, y le das la spec para que sepa qué es lo correcto: actúa como pentester, con esta spec y este diff enumera vectores de abuso, entradas maliciosas y fallos de autorización. Ese es tu pase tipo pentest, repetible por cada spec.

Y lo respaldas con herramientas, no solo con el modelo. Un análisis estático como `Semgrep` o `CodeQL`, escaneo de dependencias, y `gitleaks` para secretos. El modelo razona y prioriza, las herramientas dan piso firme.

Lo honesto que hay que decir: un pentest hecho con un modelo no reemplaza uno de verdad en algo serio. Es una primera línea barata y repetible que caza el ochenta por ciento obvio. Para el café, además, como es local-first y sin backend, el modelo de amenazas cambia y se concentra en la integridad de los datos en el navegador, el XSS si pintas entradas del usuario, y el riesgo de las dependencias. Eso mismo es una buena lección: el modelo de amenazas sale de la arquitectura.

05

¿Se podría integrar con ponytail? ¿Sería recomendable y cuáles serían los efectos?

Revisé el repo. ponytail es un skill o plugin que hace que el agente se comporte como el dev senior más perezoso, bajo la idea de que el mejor código es el que nunca escribiste. Antes de escribir, sube una escalera: ¿esto tiene que existir?, ¿ya está en el repo?, ¿lo resuelve la librería estándar?, ¿hay algo nativo?, ¿es una línea?, y solo entonces el mínimo que funcione. Y algo clave: no recorta la validación en las fronteras de confianza, ni la seguridad, ni la accesibilidad, ni el manejo de pérdida de datos. Funciona con varios agentes, incluidos Claude Code y Kiro, tiene niveles (`lite`, `full`, `ultra`, `off`) y un comando para revisar sobre-ingeniería en el diff.

¿Choca con SDD? No. Son ortogonales y se complementan. SDD gobierna el qué y el proceso. ponytail gobierna el cómo en la implementación, o sea escribir el mínimo que cumpla la spec. Encaja bien con nosotros porque la constitución del café ya dice simplicidad sobre completitud, y ponytail es justo eso llevado al momento de generar el código. Los agentes tienden a sobre-construir, y SDD por sí solo no frena eso. ponytail sí.

Lo que sí vigilaría: inyecta reglas en cada turno, y durante la especificación y el plan a veces quieres que el agente piense a fondo, no que sea perezoso. Como tiene niveles, yo lo pondría apagado o en `lite` mientras diseñas, y en `full` al implementar. La única tensión de fondo es que la spec define lo que se necesita y ponytail corta lo que no. Mientras los criterios de aceptación de la spec sean la fuente de verdad, que corte lo de más es bueno. El riesgo solo aparece si recorta algo que la spec sí pedía, y para eso está la verificación de SDD.

En resumen: sí, vale la pena probarlo, pero como capa opcional de la fase de implementación, no como pieza central. Es un buen tema para profundizar en un espacio aparte. El repo es `github.com/DietrichGebert/ponytail`.