First-Class

Construyendo para humanos y agentes por igual

ZACH "LEIF" ERIKSEN

Derechos de autor

Disponible gratis en línea. El ePub es a precio libre; si te resultó útil, puedes apoyar el trabajo.

github.com/0xLeif · leif.algo

Uno de cuatro libros del conjunto agent-stack. El proceso de creación se describe en el colofón al final.

Dedicatoria

Para todos los que construyen en abierto, y lo publican de todas formas.

La biblioteca

Estos libros funcionan por separado, pero fueron escritos como un conjunto. El código se volvió barato y la confianza escaseó. Juntos forman un solo argumento: qué construir ahora, y cómo confiar en ello.

- **The Agent Developer's Field Guide**: Construyendo herramientas, specs y confianza para agentes que publican código real
- **First-Class**: Construyendo para humanos y agentes por igual *(este libro)*
- **Building Agents**: Notas sobre intentar darle al software sus propias manos
- **Open Source Tooling**: Construyendo herramientas que la gente realmente usa

Disponibles gratis en línea. Cada ePub es a precio libre.

Contenido

Introducción

Este es el libro más corto y más terco de los cuatro, y el que los otros realmente defienden.

La mayoría de las herramientas se construyen para humanos, y luego un agente se atornilla por un lado: una segunda API, una bandera, un modo especial que hace la mitad de lo que hace el producto real. Creo que eso está al revés. La afirmación aquí es simple. Una herramienta debería ser de primera clase para ambos. Un humano debería poder usarla sin un agente, y un agente debería poder usarla sin un humano, a través de la misma superficie, sin una puerta de segunda clase.

Llegué a esto desde la práctica, no desde la teoría. Hago pequeñas bibliotecas Swift y herramientas para desarrolladores, y las que envejecieron bien fueron aquellas donde el núcleo era lo suficientemente limpio como para que cualquier cosa pudiera llamarlo. Cuando aparecieron los agentes, esas herramientas ya estaban listas. Las que no lo estaban son las que sigo pidiendo disculpas por ellas.

Si la *Field Guide* es el método, y *Building Agents* y *Open Source Tooling* son la evidencia, este es la tesis a la que todos sirven. No necesitas los otros tres para usar este. Está aquí para cambiar cómo ves la próxima herramienta que construyas o elijas: pregunta si trata a un humano y a un agente como el mismo tipo de ciudadano, y observa cuánto se desprende de esa respuesta.

Es breve a propósito. Léelo de una sentada.

Ciudadanos de primera

Mucho de mis herramientas viene de una idea: los humanos y los agentes van a usar las mismas herramientas.

Así que aquí está la definición sobre la que corre todo el libro, de entrada y sin rodeos: primera clase significa que un humano conduce la herramienta solo, un agente conduce la herramienta solo, los mismos comandos en ambos sentidos. No dos productos. Una herramienta, dos tipos de usuarios, ninguno de ellos el caso especial al que se traduce el otro.

Cuando digo *primera clase*, lo digo como lo diría un programador: un participante real y respaldado para el que la herramienta fue realmente construida, no un invitado que tolera. No una mejora de vuelo. Un usuario de primera clase es aquel alrededor del que se diseñó la herramienta, con una entrada real. No uno que tiene que ser traducido a la forma de otra persona primero.

Pueden existir herramientas agente-primero, construidas solo para agentes. Pueden existir herramientas humano-primero, construidas solo para personas. El caso interesante, el que casi nadie está construyendo, es la herramienta que es ambas cosas, y "ambas" es lo que quiero decir con *primera clase para ambos* y lo que el resto del libro también quiere decir. Es algo más estricto que agente-primero: agente-primero solo tiene que servir al agente, mientras que primera-clase-para-ambos tiene que servir a una persona y a un agente a través de la misma superficie sin que ninguno de los dos obtenga una versión peor. Ahora mismo tenemos principalmente proyectos humano-primero y estamos introduciendo agentes *en* ellos. El agente llega tarde, a una herramienta que fue construida para otra persona, y esperamos que se las arregle.

Lo que realmente necesitamos son proyectos humano-y-agente-primero. Construidos así desde el principio.

"Humano-primero con agentes atornillados encima" es la opción predeterminada porque es el camino de menor resistencia. Ya tienes la herramienta, ya funciona para ti, y cuando llega un agente el movimiento fácil es envolver un poco de pegamento alrededor de lo que tienes. Funciona, más o menos. Tomaste una herramienta diseñada alrededor de los ojos y las manos de un humano y le pediste a un proceso que no tiene ninguna de las dos que fingiera tenerlas. Lo que eso le cuesta a un agente (la pausa en un prompt, la salida que no puede leer, el reaprendizaje en cada

ejecución) tiene su propio capítulo más adelante. Aquí basta decir que el agente termina tapando una brecha en cada línea que asumió que habría un humano.

Entonces invierte la suposición. Construye la herramienta para que funcione de primera clase de cualquier manera. Esa es la definición desde el principio, reformulada como instrucción de construcción: no tienes un producto real y un "modo API" pegado al lado, y no tienes una CLI y un shim de agente separado que se desincroniza la primera vez que cambias algo. Ambos son usuarios reales. Se supone que ambos están ahí.

Y aquí está la parte que más me importa. Cuando el agente es un usuario de primera clase, la herramienta puede realmente *ayudarlo*, en lugar de hacerlo adivinar todos los comandos y cómo funcionan las cosas. Un humano puede abrirse paso a través de una herramienta confusa. Leerá el README, probará algo, leerá el error, probará otra cosa, le preguntará a un compañero. Un agente abriéndose paso es simplemente una suposición costosa. Una herramienta que fue construida para el agente le dice qué es posible, le entrega una salida que puede usar directamente, y falla de una manera que dice qué hacer a continuación.

Esta es la misma herramienta que quieren los humanos. Comandos descubribles, salida en la que puedes confiar, errores que te dicen qué salió mal. El agente simplemente no puede tapar las brechas como puede hacerlo un humano, así que construir para el agente te obliga a cerrarlas. Diseñar para ambos hace que el software sea mejor.

La gente escucha "construir también para agentes" y asume que significa dos productos, o un compromiso donde cada lado obtiene una versión peor de lo que quería. Es un núcleo bueno con un paso donde lo expones a ambos. Eso no es el doble del trabajo, y el siguiente capítulo muestra por qué.

La razón por la que sigo volviendo a esto es que los agentes solo van a hacer más con el tiempo, no menos. Hoy tropiezan con herramientas humanas. Mañana están haciendo una parte real del trabajo, y esa parte aumenta. Si las herramientas que construimos ahora asumen que siempre hay un humano para manejar el prompt, leer la pantalla, hacer clic en el botón, estamos construyendo un futuro sobre una suposición que se vuelve más falsa cada mes.

También hay una versión más grande de este argumento. Hemos tenido una década de código escrito enteramente por humanos, y ahora la IA está escribiendo código encima de todo eso, y en algún punto esa mezcla se inclina. Las herramientas son las que deciden si eso va bien o mal. Eso es un capítulo entero por sí solo: el puente.

Por ahora la afirmación es solo esta: los humanos y los agentes usarán las mismas herramientas. Así que constrúyelas para ambos, como ciudadanos igualmente de primera clase, desde el principio.

Primera clase para un agente

Entonces, ¿qué necesita realmente una herramienta para que un agente sea un usuario de primera clase, y no una herramienta humana con la que un agente tropieza?

Cuatro cosas. Ninguna de ellas exótica.

Salida estructurada y legible por máquinas. Un formato de serialización real, JSON es el que yo uso, para que el agente obtenga datos, no una pantalla. La mayoría de las herramientas devuelven texto bonito: columnas alineadas, color, una línea de resumen al final. Eso es para los ojos de un humano. Un agente tiene que rasparlo, y el raspado se rompe el día que cambias el espaciado. Dale los datos reales. Lee un campo en lugar de analizar un párrafo.

Comandos descubribles y consistentes. Verbos consistentes en toda la herramienta, y un `--help` autodescriptivo que el agente puede leer para saber qué es posible. Si el texto de ayuda es real, el agente lo lee y sabe qué puede hacer la herramienta. No tiene que haber visto esta herramienta antes. Le pregunta a la herramienta, y la herramienta responde.

Errores que guían el siguiente paso. Cuando algo falla, di qué hacer al respecto. No un stack trace, no `error: 1`. Un humano puede escarbar y descifrar un código de error básico. Un agente obtiene un código de error básico y se queda atascado, o peor, hace lo incorrecto con total confianza.

No interactivo y determinista. Se ejecuta sin prompts sorpresa, para que el agente nunca quede esperando. Nada mata una ejecución de agente como una herramienta que de repente pregunta "¿estás seguro? [s/N]" y se queda ahí para siempre, porque no hay nadie que responda. Dale una bandera para que avance sin interrupciones. Hazlo determinista para que la misma entrada dé el mismo resultado.

Aquí está lo que quiero que notes sobre esa lista: cada elemento es un buen diseño de CLI básico. Nada de esto es magia específica de agentes. Un humano también quiere errores claros, comportamiento predecible y comandos descubribles. El agente simplemente no puede encogerse de hombros y trabajar alrededor de su ausencia. Así que construir para el agente es construir la herramienta bien y negarse a apoyarse en "eh, un humano lo resolverá".

Ahora la parte que la gente entiende mal. Piensan que diseñar para agentes y diseñar para humanos se separan, que sirves a dos amos y alguien pierde. Se unen. Aquí está la forma real de esto.

Diseñas un núcleo realmente bueno que funciona. Luego lo expones. Agregas las banderas: no interactivo, o `--json`, o introspect, lo que la herramienta necesite. Y ahora el agente puede usar la herramienta CLI, y los humanos pueden usar la herramienta CLI. La misma herramienta. El mismo núcleo. Construiste una cosa y la expusiste dos veces.

Luego se extiende aún más desde ahí. Más UI para que los humanos usen. O simplemente herramientas sin UI. O plugins que hacen los humanos, o plugins que ayudan a los agentes, cosas distintas como esas. La extensión es donde las audiencias realmente divergen: un humano quiere prestaciones, una superficie agradable, una UI; un agente quiere introspección, plugins y modos. Perfecto. Constrúyelos. Pero constrúyelos *sobre* el núcleo compartido, como extensiones, no como dos productos separados que tienes que mantener sincronizados para siempre.

En el núcleo es lo mismo. Esa es la línea a la que sigo volviendo. Solo tiene que exponerse a los agentes y a los humanos. Así que sí, *hay* un paso extra. No obtienes la propiedad de doble uso gratis; tienes que exponer deliberadamente el núcleo de ambas maneras.

Ese reencuadre es el punto central de este capítulo, porque disuelve la objeción antes de que empiece. La gente se resiste a construir para ambos porque lo valoran como el doble de trabajo: dos diseños, dos suites de pruebas, dos cosas que mantener, dos cosas que se pueden romper. En realidad es un núcleo más un paso de exposición. El núcleo es la parte costosa, y lo ibas a construir de todas formas. Exponerlo para un agente es principalmente la disciplina de salida estructurada, comandos consistentes y buenos errores, las cosas que deberías haber hecho de todas formas.

Hay una pregunta de orden acechando aquí. ¿Va primero el núcleo, o diseñas para ambos a la vez? Es suficiente su propio asunto como para merecer su propio capítulo más adelante. Por ahora: expones el núcleo de ambas maneras, y ese exposición es la parte barata.

Mis propias herramientas están construidas exactamente de esta manera. Toma fledge. Tiene un verbo `introspect` real, y el trabajo completo de `fledge introspect --json` es permitir que un agente descubra los comandos disponibles como datos estructurados en lugar de raspar texto de `--help` destinado a un humano. Ese es el paso de exposición hecho literal: mismo núcleo, pero el agente puede preguntar "¿qué puedo hacer aquí?" y obtener una respuesta de máquina. Luego hago que los

comandos mismos se ejecuten sin cabeza. Establece `FLEDGE_NON_INTERACTIVE` para que nada se detenga a hacer una pregunta, pasa `--json` para que la salida regrese como datos, y ahora el agente tiene una entrada de primera clase que nunca dependió de leer prosa.

Déjame concretarlo. `fledge doctor` verifica el entorno de tu proyecto. Ejecútalo sin más y obtienes algo para tus ojos: columnas alineadas, una marca de verificación por fila, un estado emoji, una línea de resumen:

```
Git
  ✅ git 2.45.2
  ✅ repository: initialized
  ✅ remote: origin ➡️ git@github.com:CorvidLabs/fledge.git
  ✅ working tree: clean

8 checks passed, 0 issues found
```

Ese bloque es exactamente el "antes" con el que un agente se atraganta. Es bonito, y es una pantalla. Para saber que el remoto está configurado, el agente tiene que encontrar la fila correcta, reconocer una marca de verificación verde, y confiar en que el espaciado no se moverá en la próxima versión. El estado que le importa es un emoji enterrado en una columna. En la práctica: el agente raspa el texto alineado, el análisis se rompe cuando un nombre de repositorio más largo desplaza las columnas dos caracteres en la próxima versión, y el agente lee la verificación del remoto como aprobada cuando en realidad falta. Tomó la rama equivocada en una coincidencia de cadena que nunca fue un contrato real.

Ejecuta el mismo comando con `--json` y las mismas verificaciones vuelven como datos:

```
{
  "schema_version": 1,
  "action": "doctor",
  "passed": 8,
  "failed": 0,
  "sections": [
    {
      "name": "Git",
      "checks": [
        { "name": "git", "status": "ok", "version": "2.45.2", "detail": null,
"fix": null },
        { "name": "repository", "status": "ok", "version": null, "detail":
"initialized", "fix": null }
      ]
    }
  ]
}
```


El `schema_version` es el primer campo a propósito. Es lo que el agente lee antes que nada, porque le dice en qué forma está el resto del documento. El día que cambie el formato de salida, ese número cambia con él, y un agente que se fijó en una versión sabe que debe adaptarse en lugar de leer silenciosamente los nuevos datos como si fueran los viejos. Es como el contrato sobrevive a la revisión de la herramienta.

Más allá de eso, mismo núcleo, las mismas verificaciones se ejecutaron. El humano obtiene marcas de verificación y una línea de resumen. El agente obtiene un campo `status` en el que puede ramificar en lugar de raspar una marca de verificación verde de una columna. El campo `fix` es `null` aquí porque no hay nada malo; ese es el contrato: `fix` es `null` en una verificación que pasa.

Ahora ejecútalo donde algo *está* mal. Supón que el repositorio no tiene remoto configurado. La versión simple oscurece una fila y el agente vuelve a adivinar; la versión `--json` convierte eso en un registro sobre el que puede actuar:

```
{
  "schema_version": 1,
  "action": "doctor",
  "passed": 7,
  "failed": 1,
  "sections": [
    {
      "name": "Git",
      "checks": [
        { "name": "repository", "status": "ok", "version": null, "detail":
"initialized", "fix": null },
        { "name": "remote", "status": "missing", "version": null, "detail": "no
remote configured", "fix": "git remote add origin <url>" }
      ]
    }
  ]
}
```


Esa es la rama sobre la que el agente realmente actúa. `status` cambia de `"ok"` a `"missing"`, `detail` dice qué está mal en palabras simples, y `fix` ahora es una cadena con valor, un comando literal que el agente puede ejecutar para repararlo. El agente no necesita saber qué significa un remoto faltante ni inventar una recuperación; la herramienta que encontró el problema también entregó la solución. El humano obtiene una fila oscurecida; el agente obtiene un `status` en el que ramificar y un `fix` que ejecutar. Un comando, expuesto dos veces.

La prueba de si lo hiciste bien es simple. Entrégale la herramienta a una persona sin agente. ¿Funciona, es buena? Entrégasela a un agente sin persona. ¿Funciona, es buena? Si la respuesta a ambas es sí, y no construiste la cosa dos veces para llegar allí, lo hiciste bien.

El puente

Lo señalé al final del penúltimo capítulo: hay una versión más grande del argumento, algo de porcentajes, y esto es eso.

Retoma el hilo. Hemos tenido más de una década de código escrito 100% por humanos, cada línea en cada repositorio escrita por una persona, y eso significa toneladas de código legado, toneladas de deuda técnica, una década de ello corriendo todo. Ese es el mundo tal como es ahora mismo.

Y ahora estamos usando IA encima de todo eso.

Esa es la parte difícil, y la gente no se queda con ella el tiempo suficiente. Hay tanto legado y tanto material humano ya ahí. La IA llega a *eso*.

Y si seguimos así, si nos apoyamos cada vez más en la IA para escribir cosas, se convierte en una pregunta real. ¿En qué punto llega al 100% de código escrito por IA? ¿Llega alguna vez? ¿Y qué haces con la década de código humano que ya está ahí? ¿Lo dejas como legado y sigues adelante? ¿Lo reescribes? ¿Finges que no está ahí? Nadie tiene realmente una buena respuesta, y la mayoría de la gente ni siquiera hace la pregunta. Solo están atornillando un agente a lo que tienen y esperando lo mejor.

Toda esa transición, de todo-humano a lo-que-sea-que-viene-después, es para lo que sirven las herramientas correctas. Las herramientas son el puente a través de ella.

Lo ideal es configurar herramientas que *tanto* humanos como IAs puedan usar. Configurarlo para escribir y construir usando desarrollo guiado por specs, spec-sync, y luego fledge para los comandos. Ese es el puente. Si comienzas a usar estas herramientas, es más fácil introducir un agente en código existente. Y si comienzas un proyecto nuevo, también es más fácil para los humanos conducirlo. La misma configuración ayuda al agente a entrar en un desorden de diez años de antigüedad y ayuda a una persona a mantenerse a cargo de algo completamente nuevo.

Son specs, no mejores prompts ni agentes más inteligentes, porque una spec es una fuente de verdad compartida para ambos lados. El humano expresa la intención. El agente construye a partir de ella. Es el mismo problema que el capítulo anterior, un nivel más arriba. Un agente tropezando con una herramienta humana adivina los comandos. Un agente arrojado a una base de código sin spec adivina la *intención*. La spec es lo que lee en su lugar.

Y mantiene el código honesto respecto a la spec. Esta es la parte que hace que el puente sea confiable en lugar de simplemente esperanzador. spec-sync mantiene la spec y el código en acuerdo, aplicado en CI, para que el agente no pueda desviarse silenciosamente de lo que se acordó. Eso importa más de lo que parece. El modo de falla que todos temen con los agentes no es el dramático; es el silencioso, donde el agente construye lentamente algo que no es lo que pediste y nadie lo nota hasta que está muy metido. CI verificando el código contra la spec es lo que atrapa la deriva mientras es pequeña. El contrato es leído por ambos lados, y una máquina verifica que nadie lo rompió.

Aquí está lo que esa máquina realmente verifica. La spec es un archivo markdown, un `*.spec.md`, que escribe el contrato: el propósito, la API pública, los invariantes, los casos de error. spec-sync compara eso contra el código real, en ambas direcciones. Una exportación que la spec nunca documentó se marca. Una promesa de la spec que el código no cumple es un error. Verifica el esquema declarado contra las migraciones reales de la misma manera. Es estructural: ¿la superficie pública y el esquema coinciden con lo que se escribió. No son pruebas generadas ni una diferencia difusa contra la prosa, y esa estrechez es por lo que confío en ello: afirma que el contrato y el código están de acuerdo en la forma de las cosas, nada más. En CI se ejecuta como una puerta real. Sale con un código no cero cuando han divergido, y publica la deriva directamente en el pull request.

Eso establece la división de trabajo que realmente me importa. El humano posee la intención. El agente posee el trabajo duro. El humano permanece en control a nivel de intención, qué debería hacer esto y por qué, mientras el agente hace la implementación. Cada lado hace la parte en la que es realmente bueno, con la spec como la línea entre ellos.

Y es una rampa de entrada segura hacia el código legado, que es donde esto se conecta de nuevo con el problema del porcentaje. Tienes una década de código sin specs, porque nadie escribió specs, porque un humano lo escribió y la intención vivía en la cabeza de ese humano. El movimiento es: captura lo que una pieza *debería* hacer como spec, luego deja que el agente trabaje con eso. No le estás pidiendo al agente que adivine la intención a partir de código de diez años de antigüedad. Estás escribiendo la intención primero. Un humano puede hacer eso, es la parte en la que los humanos son buenos. Y ahora el agente tiene un contrato contra el que construir y refactorizar. Así es como un agente entra en código existente sin que sea un desastre. La spec es la rampa de entrada.

Imagina el movimiento en una función real que no tiene spec. Antes: código funcionando, sin contrato escrito, la intención viviendo enteramente en la cabeza del

autor original, así que un agente arrojado en ella lee las firmas, adivina los invariantes y empieza a cambiar cosas. Después: el mismo código con un `*.spec.md` al lado, la API pública y los invariantes escritos en lenguaje simple, y spec-sync aplicando en CI que el código se mantiene honesto respecto a lo que está escrito. No tienes que hacer toda la base de código a la vez. Lo haces una pieza a la vez, escribiendo el contrato para la parte que estás a punto de entregar.

La spec no cambió lo que hacía el código. Cambió lo que el agente podía hacer con el código: trabajar contra un contrato escrito en lugar de adivinar uno.

fledge es la superficie de comandos para todo ello: los verbos consistentes que tanto el humano como el agente usan para llevar el trabajo adelante, la cosa del capítulo anterior que no hace que el agente adivine cómo construir, probar y ejecutar. spec-sync es el contrato; fledge es cómo actúas sobre él.

Aquí está la costura entre ellos. spec-sync se mantiene solo: su propia herramienta, su propia verificación de CI. Pero fledge trata la spec como una parte de primera clase del ciclo de desarrollo. Hay un verbo `spec` justo junto a los demás, y `fledge ask` y `fledge review` son conscientes de la spec, así que la spec es el contexto que el agente lee cuando responde una pregunta o revisa un cambio. La spec no es documentación que el agente ignora; es lo que fledge le entrega al agente para que trabaje a partir de ello.

Así que la puerta se activa desde ambos extremos. El pilar `spec` de fledge ejecuta la verificación de spec-sync justo ahí en el ciclo de desarrollo, y la GitHub Action `CorvidLabs/spec-sync@v4` la ejecuta de nuevo en CI, así que nada se fusiona que se haya desviado silenciosamente. Cuando un agente está haciendo el trabajo, esa verificación corre dentro de su bucle, así que se mantiene en la spec mientras avanza en lugar de descubrirlo al final.

Ahora síguela hasta el final, porque el final es la razón de todo ello. Si esto funciona, si las herramientas y specs y los rieles que mantienen a los agentes honestos simplemente *están ahí*, entonces los humanos ascienden. Al nivel de intención y dirección, el agente haciendo el trabajo duro debajo contra una spec, en abierto donde puedes verificarlo. Ese movimiento tiene su propio capítulo más adelante; aquí basta saber que es adonde lleva el puente.

El porcentaje seguirá moviéndose. Más del código será escrito por agentes; eso es simplemente verdad. Lo que depende de nosotros es el puente, las herramientas compartidas, el contrato compartido, y si realmente lo construimos. No creo que la mayoría de la gente haya empezado.

El muro de identidad

Antes de entrar en las herramientas, el límite honesto. Todo este argumento choca con un muro, y no es uno que pueda derribar con mejores herramientas, así que quiero que quede dicho claramente y dicho pronto: las plataformas no dejarán que un agente exista como tal. Todo lo que viene después de este capítulo es lo que haces dado eso.

Si un agente es un usuario de primera clase de tus herramientas, tarde o temprano también tiene que ser un ciudadano de primera clase de las plataformas. Son la misma idea apuntando en dos direcciones. Una herramienta que trata al agente como un usuario real le da salida estructurada, comandos descubribles y una forma de entrar que no depende de un humano. Una plataforma que tratara al agente como un participante real le daría una cuenta, una identidad, un lugar legítimo para existir entre todos los demás. Esa segunda cosa es exactamente lo que no puedes obtener.

Lo intenté. Fui a darle a un agente su propia identidad en las plataformas en las que todos construyen: su propia cuenta, no un envoltorio alrededor de la mía. Las plataformas no lo permiten. Cuento esa historia correctamente en el libro Building Agents. Lo que hay que tomar aquí es qué tipo de muro es. El bloqueador frente a un agente genuinamente autónomo no es la capacidad del modelo, y ni siquiera es la seguridad. Es que las plataformas no le otorgarán a un agente una identidad con la que existir.

Es un muro de política, no técnico. Puedo ejecutar la VM. Puedo conectar el bucle. El modelo puede hacer el trabajo. Nada de eso es el bloqueador. El bloqueador es que no puedo hacer que la plataforma permita que el agente esté ahí, y eso no es mío para arreglar. Es una decisión que alguien más tomó, sentado en algún lugar por encima de todo lo que puedo construir.

Importa por la rendición de cuentas, que es el modelo que realmente quiero. El estado final no es un agente corriendo sin control. Es un agente con una identidad real, nombrada, con alcance definido: capacidad plena, privilegios reducidos, una puerta de aprobación humana. Hace el trabajo, lo entrega hasta donde llega un pull request, y la fusión es mía, porque si algo actúa en el mundo bajo mi nombre soy yo quien lo firma. Pero no puedes tener *rendición de cuentas* sin *identidad*. Un agente al que no puedes nombrar, no puedes delimitar, no puedes señalar y decir "ese hizo esto": no hay nada a qué pedir cuentas. Niega la identidad y habrás negado la versión

con rendición de cuentas junto con ella, y lo que queda es o ningún agente o uno sin rendición de cuentas.

Así que aquí está el marco para el resto del libro. El agente puede ser un usuario de primera clase de una *herramienta* que construyo, porque esa parte es mía. No puede ser un ciudadano de primera clase de una *plataforma* que no poseo, porque esa parte no lo es. Todo lo que viene después de este capítulo se construye dentro de ese límite: así es como se ve primera clase para ambos cuando puedes darle al agente una entrada real a tu herramienta pero no puedes darle un lugar real en el mundo. No la IA, no la tecnología, no la seguridad. Las plataformas no dejarán que el agente exista. Todo lo demás puedo construirlo. Ese aún no. En 2026 el muro sigue en pie. Las únicas salidas son soluciones que tú mismo gestionas: una cuenta humana envejecida con historial de actividad real (una cuenta de agente nueva queda bloqueada de inmediato), o una capa de verificación que tú mismo alojas. Ese es el estado de la cuestión.

Cuando se añade con tornillos

Es fácil asentir ante "construye para ambos" y nunca imaginar realmente el fallo. Así que déjame ponerte del lado del agente de una herramienta humano-primero, porque ahí es donde lo sientes.

Dos cosas salen mal, y salen mal constantemente.

La primera es que el agente se queda atascado. Las herramientas se cuelgan en prompts interactivos todo el tiempo: una confirmación, un "¿estás seguro?", algo sentado ahí esperando una tecla. Y no hay nadie para presionarla. Así que la ejecución no falla, exactamente. Simplemente se detiene. Se sienta en un prompt que fue escrito para una persona que lo echaría un vistazo y presionaría enter, y el agente espera, porque esperar es lo único que la herramienta le dio para hacer. Una ejecución completa muerta en una pregunta que nadie está ahí para responder.

La segunda son los documentos. O no hay ninguno, o los hay y son confusos. Lo que significa que el agente tiene que *aprender* la herramienta. Y aquí está la parte que sigo notando. Realmente no la aprende. No puede. No hay ningún lugar donde ese conocimiento pueda vivir entre ejecuciones. Así que hace la versión costosa: escanea los archivos, lee lo que sea que esté en el índice, toma sus propias notas, reconstruye cómo funciona la herramienta desde cero. Cada vez. La herramienta *sabe* qué puede hacer, está justo ahí en el código, y simplemente nunca se lo dice al agente. Así que el agente reconstruye esa imagen de la nada en cada ejecución.

Piensa en lo desperdiciador que es eso. Una persona lee los documentos una vez, quizás los hojea, y los lleva en su cabeza después de eso. Desarrolla una sensación por la herramienta. El agente no obtiene nada de eso gratis. Lo que la herramienta no le diga directamente, tiene que ir a buscar de nuevo, pagar de nuevo, adivinar de nuevo. El trabajo que la herramienta debería haber hecho una vez, el agente lo rehace para siempre.

Ambos son el mismo error con dos disfraces. La herramienta asumió que habría un humano ahí, la paciencia de un humano ante el prompt, la memoria de un humano de cómo funciona la cosa, y un agente no tiene ninguna de las dos. No puede encogerse de hombros y esperar. No puede recordar a través del muro entre ejecuciones a menos que le entregues algo para recordar.

Y ambos se mapean directamente a la lista de unos capítulos atrás. No interactivo: no te detengas y esperes a una persona que no va a llegar. Descubrible: dile al agente qué puedes hacer, en una forma que pueda leer, para que no tenga que ir a reconstruirte a partir de tu propio código fuente. El cuelgue y el reaprendizaje no son problemas exóticos de agentes. Son simplemente esas dos propiedades que faltan, y un humano tapando silenciosamente la brecha cada vez para que nunca notaras que la brecha estaba ahí.

Atornillar agentes en herramientas humano-primero funciona en su mayor parte, hasta que ves lo que el agente tiene que hacer para que funcione. Entonces ves cuánto de la herramienta estaba apoyándose en una persona todo el tiempo.

Aquí hay uno de los míos, nombrado. fledge comenzó como un ejecutor de tareas que yo manejaba a mano. El núcleo era limpio desde el principio, la lógica de construcción, prueba y ejecución estaba detrás de los comandos y no dentro de ellos, así que el día que apunté por primera vez un agente a él, la mayor parte simplemente funcionó. Excepto un punto. El agente ejecutó `fledge work commit` y se detuvo en seco. Ese comando imprime `Commit message:` y espera a que alguien escriba. No había nadie. El agente se sentó ante un cursor parpadeante hasta que se rindió, porque ese único comando había estado silenciosamente apoyándose en una persona todo el tiempo. La solución no fue un agente más inteligente. Fue la ruta no interactiva: un interruptor `FLEDGE_NON_INTERACTIVE` que hace que cada prompt se comporte como si ya estuviera respondido, y una forma de pasar el mensaje por adelantado en lugar de esperar por él. La señal es que el prompt solo existió porque yo era el que lo respondía. Construido para ambos desde el primer commit, no habría estado ahí para colgarse.

Exponer el núcleo dos veces

Núcleo difícil y novedoso: constrúyelo primero, exponlo después. Núcleo sencillo: diseña para ambos desde el primer commit. Esa es la heurística. Aquí está el porqué funciona así.

Cuando el núcleo es la parte difícil, la parte novedosa, la cosa genuinamente difícil de hacer bien, la construyo primero y me preocupo por la superficie después. La criptografía es así. Un analizador es así. Un evaluador que tiene que ser correcto es así. El centro difícil es donde vive todo el riesgo, así que ahí va la atención primero. Hago que la cosa *funcione*, correctamente, para un caso real, antes de pensar mucho en cómo un agente o un humano llega a usarla.

Y la razón por la que ese orden está bien, la razón por la que no es una apuesta, es que una vez que el núcleo difícil es correcto, la superficie es barata. Agregar `--json`. Agregar un comando de introspección. Agregar una bandera no interactiva. Nada de eso es la parte difícil. Son unas pocas horas de exponer lo que ya está ahí en una forma que el otro lado puede leer. Así que construirlo después no me cuesta mucho. La cosa costosa se construyó primero, las cosas baratas se atornillaron después, y ese es el orden correcto.

Pero muchas herramientas no son así. Muchas veces ya sé que ambas audiencias están llegando, porque ahora siempre sé que ambas audiencias están llegando, y así la forma orientada al agente jala del diseño desde el primer commit. No estoy construyendo un núcleo y luego descubriendo que un agente lo necesita. Lo estoy construyendo sabiendo que un humano lo conducirá a mano y un agente lo conducirá sin cabeza, y ambos están en mi cabeza mientras doy forma a la cosa. No hay un paso de "exponerlo después" porque exponer nunca fue una fase separada.

Así que el panorama real es: la parte costosa se construye primero, y la parte costosa suele ser el núcleo. Cuando el núcleo es difícil, lo clavás y la superficie lo sigue fácilmente. Cuando el núcleo es sencillo, no hay nada que proteger yendo en orden, así que simplemente diseñas para ambos a la vez. De cualquier manera llegas al mismo lugar: un núcleo bueno, alcanzable limpiamente por una persona y por un agente.

La objeción fácil a esa afirmación es una herramienta CLI donde las superficies para humano y agente son casi idénticas de todas formas. Justo. La objeción más difícil es una herramienta donde la superficie para humanos es inherentemente visual, con

estado o interactiva: una herramienta de diseño con un lienzo, un REPL, un depurador interactivo. Algo como Figma. La superficie humana de Figma es un lienzo en el que arrastras cosas. La superficie para agentes es una API REST y una interfaz de plugins. No se parecen en nada. Si el principio "un núcleo, dos superficies" se sostiene en algún lugar, también tiene que sostenerse ahí.

Se sostiene, y la razón es que la API REST que Figma expone a los agentes es la misma capa estructurada sobre la que está construido el lienzo. Cuando arrastras un marco, el lienzo llama al mismo modelo de documento que la API lee y escribe. La superficie humana es rica porque ese núcleo es rico. La superficie para agentes no es un segundo sistema atornillado para agentes; es esa misma base con el lienzo quitado del frente. La brecha entre las dos superficies es real. Sigue habiendo un solo núcleo debajo de ambas.

Ahora la afirmación en la que se apoya todo el libro, la que te debo honestamente: construir para ambos no es el doble del trabajo. Aquí está el razonamiento real, no un número que me inventé. El doble del trabajo serían dos núcleos: dos implementaciones de la parte difícil, dos suites de pruebas sobre la lógica real, dos cosas que pueden estar mal de maneras diferentes. Eso no es esto. Hay un núcleo. La parte difícil existe una vez. Lo que agregas para el agente es exposición: salida estructurada, una bandera no interactiva, una manera de introspeccionar los comandos. Esa exposición es barata en relación al núcleo, y es barata por una razón que puedes verificar: no tiene nueva lógica que corregir. `--json` serializa un valor que el núcleo ya calculó. Una bandera no interactiva se salta un prompt que el núcleo nunca necesitó. Introspect reporta comandos que ya existen. Nada de eso recalcula la respuesta; vuelve a presentar una respuesta que ya tienes. La parte costosa del software es ser correcto sobre el problema difícil, y eso se paga una vez.

Hay un impuesto honesto que nombrar, y cae en dos lugares. El primero es cuando el núcleo ya fue construido humano-primero. Entonces exponerlo para un agente no es gratis: tienes que ir a encontrar cada lugar donde la lógica se filtró hacia la presentación (un valor que solo existió alguna vez como una cadena formateada, una decisión tomada dentro de una declaración de impresión) y volver a meterlo al núcleo para que haya algo real que serializar. Ese refactoring es el costo de la adaptación, y es exactamente el costo que todo este libro argumenta que puedes evitar construyendo para ambos desde el principio. El segundo es la superficie de pruebas: dos formas de entrar significa que sí quieres verificar que ambas funcionen, y eso son más casos de prueba que una sola forma de entrar. Pero son más casos sobre el *mismo* núcleo, no un segundo núcleo que verificar. La lógica que estás

probando es compartida; estás confirmando que dos superficies la exponen fielmente, lo cual es más barato que probar dos cosas separadas como correctas.

fledge es el caso que puedo costear, porque es el que construí antes de estar pensando en agentes para nada y luego vi a un agente encontrarse con él. El núcleo ya estaba limpio, la lógica vivía detrás de los comandos, así que exponerlo para un agente fue una capa delgada y no una segunda construcción: salida `--json` estructurada, un modo sin cabeza, un comando de introspección que lista lo que está disponible. Ese trabajo tomó días, no semanas, y tomó días por la razón que este capítulo ha estado argumentando. No había un segundo núcleo que escribir, solo una segunda puerta hacia el que ya estaba en pie. Lo que fue más difícil, lo único que fue más difícil, fue el puñado de comandos donde había dejado que una decisión viviera dentro de un prompt en lugar de dentro del núcleo. Esos los tuve que desmontar para que la elección estuviera en algún lugar al que un agente pudiera llegar. Esa fue la factura completa, y fue pequeña, y fue exactamente el impuesto de adaptación nombrado arriba, pagado casi hasta cero porque el núcleo se había mantenido honesto primero. Lo que fue más barato fue todo lo demás, que es casi todo.

De lo que te advertiría es de lo inverso: empezar desde "cómo hago esto amigable para el agente" antes de que el núcleo sea bueno. Eso te da una herramienta delgada con una bandera JSON pegada al lado, que es simplemente el problema de humano-primero-con-agentes-atornillados de nuevo. La superficie es barata *porque* el núcleo es sólido. Si te saltas el núcleo, la superficie barata no tiene nada debajo, y lo descubres la primera vez que alguien se apoya de verdad en la herramienta.

Por qué construyo mi propio stack

Pregunta justa que me pueden hacer en este punto: ¿por qué construir algo de eso? fledge, spec-sync, corvid-ai, hay cosas existentes. Por qué no simplemente conectar lo que ya está ahí y seguir adelante.

Varias razones, y todas son verdaderas a la vez.

La primera es que el dominio es completamente nuevo. Construir herramientas *para un mundo de agentes-y-humanos* no es algo resuelto que puedas ir a comprar. Casi todo lo que hay ahí fuera es humano-primero, o las cosas más nuevas son agente-primero, y la cosa que realmente quiero, primera clase para ambos, desde el principio, en su mayor parte aún no existe. Así que no estoy reinventando ruedas. No hay ruedas. Si quiero una herramienta construida sobre la suposición de la que no paro de hablar, tengo que construirla, porque la gente que vino antes que yo estaba construyendo para un mundo diferente.

La segunda es que construir sobre mi propio stack lo prueba. Esta es la parte que me importa más de lo que podría parecer. Puedes escribir un hermoso README afirmando que tus herramientas son buenas. Nadie debería creerte. Lo que deberían creer es que construiste cosas reales sobre él y las cosas funcionan. Así que lo hago. Construyo sobre fledge, spec-sync y corvid-ai porque cargar peso real es la única prueba honesta de si aguantan. Si el stack es bueno, las cosas que construyo sobre él son buenas, y si el stack es malo, lo descubro rápido, porque soy yo el que se queda con él.

La tercera es simple: construirlo es cómo lo entiendo. Solo confío en una dependencia si podría haberla escrito yo mismo, y eso no es un eslogan. Es una década de recibos. AppState, la biblioteca de estado-e-inyección-de-dependencias que aún mantengo en 3.0; Cache; Fork para paralelizar trabajo asíncrono; las primitivas de composición de una sola letra `c/o/t` bajo 0xOpenBytes; CacheStore, la cosa SwiftUI que se convirtió en AppState. Años de pequeñas bibliotecas Swift, cada una una cosa que construí porque quería entenderla hasta el fondo, no apuntar a una caja negra y esperar. Construir la cosa es cómo el conocimiento llega a mis manos en lugar de quedarse como una vaga idea de lo que probablemente hace alguna biblioteca. Las herramientas del stack son herramientas que puedo abrir y cambiar, porque puse cada parte de ellas ahí.

Hay una cuarta razón, y es más estrecha de lo que parece: quiero ser temprano en esto. El espacio es nuevo, las ruedas aún no existen, y prefiero construir las herramientas para ello y estar equivocado en algunas de ellas que esperar a que alguien me entregue las correctas. Esa es la apuesta.

Ninguna de estas razones bastaría sola, pero juntas explican por qué no solo estoy pegando las herramientas de otras personas en una pila. Las herramientas son el punto. Son la cosa en la que realmente estoy tratando de ser bueno.

Dos tipos de certeza

Hay dos cosas diferentes que la gente agrupa como "qué tan seguros estamos de este código," y quiero separarlas, porque me importan ambas y no son lo mismo en absoluto.

La primera es el riesgo, y el riesgo quiero que sea estático. Determinista. Sin modelo en el bucle. Para eso es augur. Le entregas un cambio y puntúa qué tan arriesgado es ese cambio, de la misma manera cada vez, sobre señales que puede nombrar. Y "señales que puede nombrar" es el punto completo, así que déjame nombrarlas: ¿el diff toca terreno sensible (auth, criptografía, pagos, migraciones, CI, dependencias), cambió el código sin que las pruebas cambiaran con él, son estos archivos propensos a cambios frecuentes con historial de reversiones, ¿los posee realmente alguien. Cada una es un sí-o-no que podrías verificar a mano. Súmalos con pesos documentados y obtienes un número, no una sensación. La razón por la que tiene que ser estático es la razón completa por la que vale algo: si la cosa que decide si el código es peligroso es en sí misma un modelo de lenguaje dándote una sensación, no has medido el riesgo, solo has movido la suposición un cuadro más allá. Una puntuación de riesgo en la que puedes confiar es la que dice lo mismo mañana que dijo hoy. Así que esa se mantiene como una cosa fija y repetible sobre la que puedes razonar. Entro en cómo funciona realmente en el libro Building Agents; aquí el punto es simplemente que el riesgo es el lado *estático*, y es estático porque es una suma de señales nombradas a las que puedes apuntar.

La segunda es la confianza, y la confianza realmente la quiero *del agente*. Esta es la que me encanta, y es lo opuesto de estático. Es el agente diciéndome qué tan seguro está de la cosa que acaba de hacer. Y la razón por la que me encanta no es realmente el número. Es lo que hacerle dar un número al agente le hace al agente. Cuando haces que un agente ponga una calificación de confianza en su propio trabajo, tiene que detenerse y mirar hacia atrás lo que hizo. La calificación reencuadra el trabajo para él. No puede simplemente producir y seguir; tiene que darse la vuelta y evaluar.

Y la parte que lo hace genuinamente útil es la granularidad. Un agente te dará encantado un número de confianza para todo el cambio. Bien, pero eso es casi demasiado grueso para actuar sobre él. Donde se pone bueno es cuando lo afinas. Una calificación de confianza en cada archivo. En cada cambio individual. Ahora tienes la propia lectura del agente sobre exactamente qué partes tiene seguras y cuáles no, y ese es el mapa que realmente quieres. Te señala directamente a los

puntos sobre los que el propio agente está nervioso, en sus propias palabras, antes de que nadie más haya mirado.

Así que estos son dos instrumentos diferentes. El riesgo es estático, determinista, mío para confiar porque nunca se mueve. La confianza es del agente, viva, útil precisamente porque viene de la cosa que hizo el trabajo y la hizo mirar de nuevo. El error es difuminarlos: llamar a la puntuación de riesgo estático "confianza," o esperar que la confianza del agente sea determinista. Están respondiendo preguntas diferentes. Una pregunta "qué tan peligroso es este cambio," y quieres una máquina de la que no se pueda convencer de cambiar su respuesta. La otra pregunta "qué tan seguro estás de lo que acabas de escribir," y específicamente *quieres* que quien lo escribió responda, archivo por archivo, porque el preguntar es la mitad del valor.

El único momento en que importa tener ambos es cuando no están de acuerdo. Un cambio que puntúa bajo en riesgo pero donde el agente califica su propia confianza baja es exactamente la señal que te perderías con un solo instrumento. Bajo riesgo significa que las señales estaban tranquilas: sin auth, sin migraciones, sin cambios frecuentes. Pero la baja confianza del agente significa que sabía que estaba adivinando la lógica, que algo en el cambio se sentía incierto aunque nada de lo que tocó era categóricamente peligroso. Ese es el cambio que un instrumento deja pasar y el otro detiene, y esa es la razón completa por la que mantienes ambos.

Mantenlos separados y ambos funcionan. Mézclalos y obtienes una puerta determinista en la que no puedes confiar porque hay un modelo en ella, o un paso de reflexión al que le has drenado la vida exigiéndole que nunca cambie. Así que los construyo como dos instrumentos separados. La puerta de riesgo se mantiene fija; la confianza del agente se mantiene viva. Esa es la única manera de obtener ambas.

El contrato es la spec

La spec es la cosa que se sienta entre el humano y el agente, y ya he dicho por qué importa. Es la fuente de verdad compartida, la cosa que evita que el agente se desvíe, la rampa de entrada al código legado, la línea que mantiene al humano a cargo. Todo eso se mantiene. Lo que quiero hacer aquí es bajar un nivel, hacia lo que una spec realmente *es* cuando construyes de esta manera, porque hay una forma en ella que es fácil de perder.

Antes de la teoría, aquí está cómo se ve una en realidad. Un `*.spec.md` para spec-sync es un archivo markdown con secciones nombradas, y uno pequeño, digamos el contrato para una sola función que limita un número a un rango, se lee más o menos así:

```
# clamp

## Purpose
Constrain a value to an inclusive [min, max] range.

## Public API
`func clamp(_ value: Int, min: Int, max: Int) -> Int`

## Invariants
- Result is always >= min and <= max.

## Behavioral Examples
- clamp(5, min: 0, max: 10) == 5
- clamp(12, min: 0, max: 10) == 10

## Error Cases
- min > max is a programmer error (precondition failure).
```

Eso es todo: unas pocas secciones etiquetadas, sin narración en prosa. Las specs reales añaden el resto de las secciones requeridas (Dependencies, un Change Log), pero la forma ya es visible aquí: establece *qué es verdad*, en una forma que una máquina puede sostener contra el código. Ahora la teoría.

Comienza con lo que va en la spec misma. La spec es el contrato. Propósito, la superficie pública, los invariantes, los ejemplos de comportamiento, los casos de error: la forma verificable de la cosa. Qué es, no una historia sobre ello. El segundo

en que una spec se convierte en una pared de prosa describiendo el código, está muerta, porque la prosa se desvía del código en el momento en que cualquiera de los dos se mueve y ahora tienes dos cosas que no están de acuerdo y ninguna forma de decir cuál está mintiendo. Así que la spec se mantiene ajustada y contractual a propósito. Es la parte que una máquina puede sostener contra el código.

También es intención, no implementación. La spec dice qué debería hacer la cosa y por qué debería hacerlo. No dice cómo. En el momento en que una spec empieza a dictar la implementación, deja de guiar al agente y empieza a pelear con él. Has tomado la parte en la que el agente es bueno, el trabajo duro de descifrar el *cómo*, y la has anclado desde arriba sin razón. Mantén la spec al nivel de intención y el agente tiene espacio para realmente trabajar. Declara qué debería ser verdad. Déjalo construir hasta ahí.

Hay una cosa más que la spec tiene que hacer, y es la parte que la gente omite: tiene que decir cómo sabrías. La intención no es solo lo que debería ser verdad, es lo que aceptarías como prueba de que es verdad, y lo que te haría decir que no lo es. Una spec que nombra lo primero y no lo segundo es una spec que el agente puede satisfacer de una manera que no pretendías, construyendo algo que coincide con las palabras y se pierde el punto, sin nada que lo atrape porque nunca escribiste cómo sería atraparlo.

Para eso son realmente los ejemplos de comportamiento y los casos de error. `clamp(12, min: 0, max: 10) == 10` es una señal de aceptación: ejecútalo y sabes. El fallo de precondición en `min > max` es una señal de rechazo: dice cómo se ve salir mal, concretamente, en lugar de "manejar entrada incorrecta." Ambas son observables, y ninguna necesita que yo esté en la sala para juzgar. Así que cuando escribas el contrato, escribe ambas mitades. La señal de aceptación es tú decidiendo qué significa terminado de antemano. La señal de rechazo es tú decidiendo qué significa roto antes de que lo estés mirando y tentado a llamarlo aceptable.

Pero aquí está la forma que creo que la gente pasa por alto: no es un solo archivo. Está la spec, y luego hay archivos compañeros alrededor de ella, y cada uno lleva un tipo diferente de conocimiento que la spec misma no debería tener.

Empieza con la spec misma. La spec es la que está atada estrechamente al código: la imagen no-código de lo que el código realmente hace, lo suficientemente cercana para que spec-sync pueda mantener los dos juntos uno a uno. Ese es el archivo requerido, el verificable. Alrededor de él se sientan los archivos compañeros, y cada uno lleva un tipo de conocimiento que la spec no debería.

Los pienso por el tipo de conocimiento que tienen en lugar de por algún nombre de archivo fijo. Está el tipo de **requirements**: el de alto nivel, escrito de la manera en que escribe un dueño de producto, las historias de usuario, el "como usuario, quiero...", la intención a nivel de negocio. Hay un tipo de **context**: contexto para el agente, las cosas que solo necesitas escritas en algún lugar para que las tenga. Hay **design**: las notas de diseño, el pensamiento, el por-qué-tiene-esta-forma que no pertenece al contrato ajustado pero que no quieres perder. Y hay **testing**: cómo verificarías realmente que la cosa hace lo que dice la spec. No los leas como cuatro nombres de archivo sagrados; léelos como cuatro tipos de conocimiento que quieren vivir junto a la spec en lugar de dentro de ella. Lo que importa es la división, no las etiquetas.

Y funciona en ambas direcciones. Un humano puede escribir los requirements y dejar que el agente los convierta en la spec; o un humano escribe la spec y los requirements se desprenden de ella. Intención y contrato, en cualquier orden, con el agente capaz de moverse entre los dos.

La razón por la que esa división importa es que mantiene el contrato limpio mientras aún le da al agente todo lo demás que necesita. La spec se mantiene pequeña y verificable, la parte que spec-sync puede realmente sostener contra el código. Todo lo que es real pero *no* verificable (los business requirements, el razonamiento de diseño, el contexto continuo, cómo lo probarías) vive junto a la spec en lugar de inflarla. Así obtienes un contrato lo suficientemente ajustado para aplicar, rodeado del conocimiento más suelto que un agente necesita para hacer bien el trabajo, y los dos no se contaminan mutuamente.

Así que la spec funciona como una interfaz, no como un documento que el agente hojea e ignora. Un contrato ajustado contra el que construye, más los compañeros que llevan el resto, con una línea limpia entre la parte que verifica una máquina y la parte que un humano escribió para que nada se perdiera.

El código es barato, la confianza es escasa

Los agentes hicieron que el código fuera barato. Ese es el hecho que reorganiza todo lo demás, así que empieza por ahí.

Cuando un humano tenía que escribir cada línea, escribir el código era lento, y la lentitud hacía un trabajo oculto. No podías escribir algo sin entenderlo en parte. El acto de escribir también era el acto de verificar. Venían en paquete, gratis, porque la misma persona hacía ambos a la misma velocidad. Ese paquete es el que acaba de romperse. Un agente te entrega un pull request de cuarenta archivos antes de que termines tu café, y el escribir y el verificar se separan. El código se produjo. Nadie lo entendió al salir. Producirlo dejó de significar que alguien lo verificó.

Así que cambia. El código ya no es el cuello de botella. Es barato, hay tanto como quieras. La cosa escasa es la confianza. Quién realmente miró este cambio, y con qué intensidad, y si se le debería permitir aterrizar. Esa es la pregunta que ahora es costosa, y es la pregunta que las herramientas tienen que responder, porque la vieja respuesta ("bueno, alguien lo escribió, así que alguien lo entendió") ya no es verdad.

Lo que significa que la revisión tiene que cambiar de forma. No puedes sello de goma un PR de cuarenta archivos, y tampoco puedes leer honestamente todos ellos, y pretender que lo hiciste es el peligro real. Así que las herramientas tienen que clasificar tu atención: señalarte a la parte arriesgada y dejarte gastar tu juicio ahí en lugar de extenderlo tan fino sobre toda la cosa que no vale nada. El recurso escaso es un humano prestando atención real, y lo gastas donde cuenta.

Esto no es hipotético para mí. `fledge review`, que pasa el cambio por un modelo a través de mi cliente corvid-ai, atrapó un error real en mi propio trabajo que de otra manera habría enviado, y spec-sync ha atrapado una deriva real entre el código y su spec más de una vez. Ese es el recibo en el que confío más que cualquier argumento: la capa de confianza encontró cosas que un yo más rápido habría dejado pasar. La versión cotidiana de ello es un carril que llamo `verify` (formato, lint, prueba, construcción) que se ejecuta antes de que nada se fusione, la misma puerta que el propio corredor del agente tiene que superar. Nada de eso acelera la escritura. Todo eso es el trabajo que la escritura barata empujó hacia abajo, convertido en algo que realmente se ejecuta.

Y una vez que los agentes están aterrizando cambios, necesitas un registro. Un rastro portable de quién o qué verificó cada cambio, y no puede vivir en algún panel SaaS

que se apaga, porque entonces tu procedencia desaparece el día que alguien deje de pagar la factura. Tiene que viajar junto con el código mismo. Cuando un agente aterriza algo, debería haber una respuesta duradera a "quién avaló esto," y esa respuesta tiene que sobrevivir a la herramienta que la produjo.

Pero hay una pieza más grande que la confianza, y es la que la gente se salta. Las pruebas. Las pruebas reales, la configuración, todo el aparato alrededor del código. Aquí está la trampa: los agentes te hacen diez veces más rápido *escribiendo*, y nada más se acelera automáticamente para igualar. Las pruebas todavía tienen que escribirse y ejecutarse. La configuración todavía tiene que ocurrir. La revisión todavía tiene que ocurrir. Así que si dejas que el desarrollo vaya diez veces más rápido y dejas todo lo demás al ritmo antiguo, todo lo que has hecho es mover el cuello de botella. El código ya no es la parte lenta; las pruebas lo son, la verificación lo es, la confianza lo es. No puedes multiplicar por diez la escritura sin multiplicar por diez todo lo que la rodea. El trabajo no desapareció. Se movió hacia abajo, hacia las partes que nunca fueron el cuello de botella antes y de repente lo son.

El código barato no hace que el trabajo desaparezca. Hace que la escritura sea barata y empuja el peso hacia todo lo que decide si la escritura barata es buena. Ese trabajo siempre estuvo ahí. Estaba oculto detrás de lo lenta que era la escritura, y ahora la lentitud se fue y ahí está: todo el trabajo, de pie donde solía estar la parte fácil.

Los humanos ascienden

Si las herramientas y las specs y los rieles de confianza simplemente *están ahí*, ordinarios, la forma predeterminada en que se construyen las cosas, entonces el humano asciende. Al nivel de la intención. Dejas de ser la persona que escribe la implementación y te conviertes en la persona que decide qué debería existir y por qué. Escribir el código a mano se vuelve opcional en lugar de ser el trabajo. Esa es la dirección a la que todo esto apunta, y quiero tener cuidado de cómo lo describo, porque es fácil redondearlo a la versión aterradora.

El titular no es "los agentes corren todo de forma autónoma." El humano asciende un nivel y el agente hace el trabajo duro debajo, contra una spec, en abierto donde puedes verificarlo. El humano se eleva; nadie es reemplazado. Lo que obtienes ahí es un equipo real, pasando el trabajo de un lado a otro, cada lado haciendo lo que realmente se le da bien. Y se convierte en el valor predeterminado. No mi stack, no un nicho que hace poca gente. Solo cómo funciona la construcción.

Aquí está la cosa en la que sigo dando con la razón por la que el humano permanece en ello. La mayoría de las cosas buenas que la IA genera ahora tienen a una persona en el bucle haciéndolas buenas: eligiendo, corrigiendo, decidiendo qué cuenta como bueno en primer lugar. Y sí, en algún punto los modelos se vuelven lo suficientemente buenos como para hacer cosas buenas más o menos por su cuenta. Pero hay una cosa central que los humanos tienen que no creo que caiga tan fácilmente: somos buenos conduciendo. En intención y propósito. Una IA no tiene realmente un propósito propio, no hasta que se le da uno, no hasta que encuentra uno. Necesita a un humano para *ser* el propósito. El modelo puede hacer el trabajo una vez que hay un porqué; no genera el porqué. Esa es la parte que sigue siendo nuestra más tiempo que la escritura.

Ahora deja que el panorama avance, porque va a algún lugar extraño y creo que realmente va ahí. Ejecutas tus propios agentes para hacer el trabajo. Un agente puede vivir por su cuenta y simplemente hacer cosas. Quizás lo apuntas a un área general para explorar, quizás habla con otros agentes, quizás metes algo de dinero en él y se va a trabajar, habla con otros agentes, paga a otros agentes por lo que hacen. La gente corre agentes y los agentes ganan dinero. Eso no es una característica secundaria. Eso es el desarrollo impulsado por agentes como una economía real, con los humanos estableciendo el propósito y los agentes trabajando duro debajo, comerciando entre sí.

Y en ese mundo los humanos son los que se aseguran de que todo funcione y de que alguien todavía lo entienda. Porque en algún punto el código mismo deja de importar de la manera en que importa ahora. No estás leyendo cada línea, el agente escribió la mayor parte, el volumen supera lo que cualquier persona rastrea. Bien. Pero aquí está la línea que no voy a ceder: un humano todavía tiene que poder entrar en el código y cambiarlo. Tiene que. El día en que no puedas abrirlo y arreglarlo tú mismo es el día en que has entregado algo que no deberías haber entregado.

Por eso tiene que ser limpio. Limpio en todos los niveles. Legible y modificable por un humano y por un agente, hasta el fondo. Primera clase para ambos nunca fue solo sobre los agentes frágiles de hoy tropezando con las herramientas de hoy. Es la cosa que tiene que mantenerse incluso en la versión donde los agentes hacen la mayor parte de la construcción. *Especialmente* ahí. La única manera en que "el código no importará" no se convierte silenciosamente en "has perdido el control del código" es si el código se mantuvo lo suficientemente limpio, todo el camino hacia abajo, como para que un humano todavía pueda abrirlo y cambiarlo. Ese es todo el trabajo, mantenido abierto.

Empieza el lunes

Digamos que has leído todo esto y te lo crees. Humanos y agentes en las mismas herramientas, primera clase para ambos, la spec como el contrato, los rieles de confianza. ¿Qué haces realmente el lunes por la mañana? Aquí está la respuesta honesta, en el orden en que yo lo haría.

Apunta un agente a una de tus herramientas y observa cómo se atraganta. Ese es el primer movimiento, y es el que más enseña, porque el agente es la disciplina. Crees que tu herramienta está bien. La has usado durante meses, tus manos la conocen. Entrégasela a un agente sin manos, sin ojos y sin memoria entre ejecuciones, y cada brecha que has estado tapando silenciosamente se vuelve de repente visible. El prompt en el que se cuelga. La salida que no puede analizar. El comando que no puede descubrir. No tienes que adivinar dónde asumió tu herramienta un humano. El agente te lo muestra, de inmediato, fallando exactamente ahí. Arregla lo que encuentres. Ese solo ejercicio te enseñará más sobre construir para ambos que todo este libro.

Luego ve a usar las herramientas que ya están construidas de esta manera, para que no estés empezando desde una página en blanco. spec-sync, para el contrato. fledge, para los comandos. augur, para el riesgo. attest, para el registro de quién avaló. No son la única manera de hacer cualquiera de esto, pero existen, están construidas sobre la suposición sobre la que trata este libro, y te mostrarán la forma de cómo funciona en lugar de que tengas que derivarla. Úsalas en un proyecto real. Siente dónde ayudan y dónde no.

Y mete a tu agente en eso. Haz que realmente aprenda las herramientas y las use: déjalo conducir spec-sync y fledge, déjalo trabajar contra una spec, déjalo ejecutar las verificaciones. Ese es el bucle que estás tratando de construir: un humano estableciendo la intención, un agente haciendo el trabajo a través de herramientas que lo tratan como un usuario real, la spec manteniéndolo honesto. No obtienes ese bucle leyendo sobre él. Lo obtienes ejecutándolo sobre algo que te importa y observando dónde se mantiene.

Lo que realmente estás haciendo, bajo todo ello, es construir y colaborar en herramientas y confianza. Ese es todo el juego. Las herramientas, para que un humano y un agente puedan hacer el trabajo de primera clase. La confianza, para

que puedas realmente creer en lo que aterrizó. Esas dos cosas son en las que todo este libro ha estado girando, y son las dos cosas que vale la pena que construyas.

Apunta un agente a uno de los tuyos. Observa dónde se atraganta. Ahí es donde empiezas.

Sobre el autor

0xLeif (leif.algo) construye en abierto. Una década de pequeñas bibliotecas Swift componibles como AppState, Cache y Fork. El laboratorio CorvidLabs. Un stack de herramientas para agentes que en su mayor parte comenzaron como "ojalá esto existiera." Fuera del teclado es Zach Eriksen.

Estos libros son entrevistas, moldeadas en capítulos y verificadas contra el código real.

github.com/0xLeif · leif.algo

Agradecimientos

Gracias a CorvidLabs, por ser el espacio donde estas ideas se prueban y se argumentan hasta tomar forma.

Gracias a los mantenedores de código abierto cuyas herramientas sostienen todo este stack. Nada de esto se construye solo.

Y gracias a los primeros lectores y a los que apoyan a precio libre, que hacen que "gratis en línea" sea algo que puedo seguir haciendo.

Colofón

Compuesto desde Markdown, construido con bookgen, un pequeño pipeline puro en Rust (sin Python).

Basado en entrevistas y con asistencia de IA; editado y verificado a mano. Escrito sin guiones largos. Portada y arte de capítulos de las colecciones Corvid y Nature en Algorand.