JuJu-Chu!

Sample

Yuka Ooka

Comienza tu flujo de trabajo Jujutsu×IA con `jj new`

¡El control de versiones también evoluciona en la era de la IA!

EXPANSIÓN DE DOMINIO

Características de Jujutsu

- Commits automáticos
- Undo universal
- Conflictos de primera clase

Juju-chu!

Comienza tu flujo de trabajo

Jujutsu × IA con `jj new`

Yuka Ooka

Klemiwary Books

Página de créditos

Juju-chu!
Comienza tu flujo de trabajo Jujutsu × IA con `jj new`

Copyright © 2026 Yuka Ooka

Todos los derechos reservados. Ninguna parte de este libro puede reproducirse, distribuirse ni transmitirse de ninguna forma ni por ningún medio sin la autorización previa por escrito de la autora, salvo citas breves en reseñas o según lo permita la ley.

Este libro se proporciona únicamente con fines informativos. La autora no ofrece garantía alguna sobre la exactitud o la integridad de su contenido, y no asume responsabilidad alguna por los daños que puedan derivarse de su uso.

Jujutsu (jj) y los demás nombres de productos, servicios y empresas que se mencionan aquí pueden ser marcas comerciales de sus respectivos titulares. Se usan únicamente con fines editoriales y de identificación, sin ánimo de infringir ningún derecho.

Primera edición en español: septiembre de 2026
Publicado originalmente en japonés: abril de 2026

Publicado por Klemiwary Books
Repositorio de apoyo: https://github.com/klemiwary/Juju-chu-es

Prefacio

En este momento, Jujutsu está atrayendo mucha atención.

En lo que respecta a los sistemas de control de versiones (VCS), Git ha reinado sin rival durante mucho tiempo. Esto es especialmente cierto para los ingenieros de software que entraron en la industria a partir de la década de 2010: muchos de ellos nunca han usado nada que no sea Git. GitHub, por su parte, fue el mayor artífice de que Git alcanzara la posición de estándar de facto, y hoy supera con creces los 180 millones de desarrolladores en todo el mundo. Está tan extendido que casi se podría decir que un ingeniero de software sin una cuenta de GitHub no es un verdadero ingeniero de software.

Con ese telón de fondo, es realmente raro que un VCS distinto de Git dé que hablar; tan raro que, cuando ocurre, casi parece un acontecimiento.

¿Por qué Jujutsu, y por qué ahora?

El desarrollo de Jujutsu comenzó en 2019, pero su primera versión pública (v0.3.0) no llegó hasta 2022. Y hasta 2025 no empezó a aparecer con regularidad en la comunidad de desarrollo. Ese momento coincide con la rápida expansión de los agentes de codificación de IA y con el instante en que el término «vibe coding» empezó a cuajar como palabra de moda.

Los agentes de IA ya pueden generar y modificar enormes cantidades de código a una velocidad que ninguna persona puede igualar, y el costo de escribir código (y de iterar sobre él) se ha desplomado. Y sospecho que cada vez más gente ha empezado a sentir un desajuste entre esa nueva realidad y los VCS tradicionales, diseñados partiendo de una premisa muy distinta: la de que «los humanos escriben código con cuidado, línea a línea, decidiendo conscientemente qué registrar y compartiéndolo solo cuando todo está listo».

Y entonces, casi sin darnos cuenta, ahí estaba Jujutsu, un VCS nuevo. La gente lo probaba y resultaba que encajaba sorprendentemente bien

con el ritmo del desarrollo basado en agentes. «¡Tengo que contarle esto a todo el mundo!». Así es, en mi opinión, como ha surgido la reciente oleada de entusiasmo.

«Pero eso también puedes hacerlo en Git»

Casi cada vez que un artículo introductorio se vuelve viral y Jujutsu se pone de moda, aparece sin falta el mismo estribillo: «Pero eso ya puedes hacerlo en Git». Pero el valor de Jujutsu no reside en hacer lo que Git no puede.

Piensa, por ejemplo, en la BlackBerry y el iPhone. Ambos podían hacer llamadas, navegar por la web, instalar y ejecutar aplicaciones y tomar fotos. Y cuando el iPhone apareció, no fueron pocos —sobre todo los más fieles a la BlackBerry— quienes lo rechazaron señalando justamente eso. Pero ya sabemos todos en qué quedó al final esa rivalidad. Aunque pudieran hacer las mismas cosas, había una diferencia decisiva en la experiencia de usuario que ofrecía cada uno.

«Poder hacer algo» y «poder hacerlo con comodidad» no son lo mismo. En Jujutsu, incluso al hacer lo mismo que harías en Git, la herramienta automatiza una mayor parte del trabajo, las operaciones son más fáciles de recordar y, si te equivocas, es fácil dar marcha atrás: todo resulta más tranquilizador, más seguro y más ligero.

Las palabras clave son «**baja carga cognitiva**» y «**seguridad psicológica**». En Jujutsu, absolutamente todo está pensado cuidando con esmero esos dos aspectos.

Para quién es este libro

Este libro está escrito pensando en un lector como el siguiente:

- Conoces las operaciones básicas de Git y usas Git y GitHub/GitLab a diario.
- Usas un agente de codificación de IA como Claude Code o Codex CLI, o tienes pensado empezar a hacerlo.
- No tienes experiencia con Jujutsu, o ya has empezado a usarlo pero todavía estás en el nivel de principiante.

Ten en cuenta que no es «un libro introductorio para quien aún no domina Git». Y aunque uso React como material para demostrar los

flujos de trabajo con un VCS, ese no es el tema central, así que no hace falta tener conocimientos de React ni del ecosistema de Node.js. Traslada los ejemplos a tu propio stack habitual a medida que avanzas.

Algunas notas antes de empezar

A fecha de agosto de 2026, Jujutsu está en pleno desarrollo y aún no ha alcanzado su versión estable (v1.0.0). De hecho, el propio README del proyecto lo describe como un «experimental version control system» y advierte de forma explícita de que pueden entrar breaking changes antes de la v1.0.0. Por eso, algunos nombres de comandos y comportamientos descritos en este libro pueden no coincidir con los de la versión más reciente. Dicho esto, el contenido de este libro no se limita al uso de comandos: es un tratamiento integral que abarca la filosofía de diseño de Jujutsu, su modelo mental y su aplicación al agentic coding. Aunque en el futuro haya cambios superficiales, estoy convencida de que el contenido de este libro seguirá siendo útil en los años venideros.

Cómo está organizado este libro

El capítulo 1 ofrece una visión general de Jujutsu; el capítulo 2 recorre las operaciones básicas; el capítulo 3 trata el desarrollo colaborativo con IA; el capítulo 4 cubre diversas aplicaciones avanzadas; y el capítulo 5 es una guía de preguntas frecuentes y resolución de problemas. Recomiendo leerlo en orden desde el principio, pero si prefieres ponerte manos a la obra cuanto antes, puedes empezar por el capítulo 2, y si te interesa especialmente la integración con la IA, puedes adelantarte al capítulo 3. Además, los capítulos 4 y 5 están pensados para funcionar también como referencia.

Una cosa más: este libro se desarrolla en forma de historia, a través de un diálogo entre dos personajes, una ingeniera sénior y una ingeniera júnior. Adopté este formato para que resultara ameno de leer y para poder responder, una a una, a las dudas que suele plantearse quien está empezando.

Y ahora, deja que te dé la bienvenida al mundo de Jujutsu. El conjuro para comenzar: `jj new`.

Sobre este libro

Reparto de personajes

Yukina Shibasaki

Ingeniera sénior en una empresa de servicios de internet con sede en Tokio. La contrataron por su dominio de React y, como líder técnica, montó desde cero el equipo de desarrollo front-end. Tiene la costumbre de descubrir tecnologías nuevas e insistir en adoptarlas antes de que estén del todo consolidadas. A veces eso tiene un poco hartos a sus compañeros de equipo, pero, por alguna razón, esas apuestas suelen acabar imponiéndose en el sector más adelante, así que el equipo lo deja pasar.

Kanae Akiya

Ingeniera júnior del equipo de Yukina, muy entusiasta. Se considera su discípula número uno e intenta seguir su ejemplo en todo. Le encantan el anime y el manga. Aunque le preocupa que la IA pueda quitarle el trabajo, usa activamente agentes de IA en el desarrollo y está decidida a no quedarse atrás.

Sobre el código de ejemplo

El código de ejemplo y los archivos de configuración que se presentan en este libro están disponibles en el repositorio de GitHub que aparece a continuación. Para la ubicación de cada archivo, consulta la nota al pie correspondiente en el texto principal.

- Repositorio público de apoyo de *Juju-chu! Comienza tu flujo de trabajo Jujutsu × IA con `jj new`*
  https://github.com/klemiwary/Juju-chu-es

Puedes usar libremente el código que aparece en el texto principal y en el repositorio anterior. También puedes citarlo o reproducir fragmentos en documentos públicos, videos y materiales similares sin pedir

permiso a la autora, siempre que menciones la fuente. Dicho esto, evita republicar el código en su totalidad.

Erratas

En el enlace de abajo encontrarás una lista de erratas con los errores de contenido y las erratas tipográficas del texto. Se actualizará según sea necesario.

- Erratas de *Juju-chu! — Comienza tu flujo de trabajo Jujutsu × IA con `jj new`*
  https://github.com/klemiwary/Juju-chu-es/blob/main/errata.md

Versiones de software usadas en este libro

- Jujutsu 0.45.1
- jjui 0.10.9
- JJ View 2.7.0
- Claude Code 2.1.260
- Codex CLI 0.153.2

Índice

Prefacio 3

Sobre este libro 6

Prólogo 12

Capítulo 1. ¿Qué clase de herramienta es Jujutsu? 15

1-1. ¿Encaja mal Git con el agentic coding? 15

El redundante modelo de tres estados de Git 15

El alto costo del cambio de contexto 17

Reescribir el historial es complejo y peligroso 17

Comandos con efectos secundarios graves y difíciles de revertir . . . 18

1-2. Las características de Jujutsu que destacan en la era de la IA 20

Al working copy se le hace commit automáticamente 23

Toda operación se puede deshacer 25

Los conflictos se tratan como un estado más 25

Comandos ortogonales y con poca dependencia del estado 27

Columna: cómo Git revolucionó el control de versione 30

Capítulo 2. Probemos Jujutsu 33

2-1. Configurar Jujutsu 33

2-1-1. Instalar Jujutsu 33

2-1-2. Configuración inicial 34

2-2. Un recorrido práctico por Jujutsu 36

2-2-1. Inicializar un repositorio 36

2-2-2. Comprobar el estado del repositorio 38

2-2-3. Trabajar con los changes 40

2-2-4. Interactuar con un remoto 45

2-3. Los tres tipos de log de Jujutsu 48

2-3-1. El log de revisiones (`jj log`) 48

2-3-2. El evolution log (`jj evolog`) 52

2-3-3. El operation log (`jj operation log`) 56

2-4. El modelo mental de Jujutsu 59

2-4-1. ¿Qué es un change? 59

2-4-2. La diferencia entre branch y bookmark 61
2-4-3. Trabajar en branches anónimos 62
2-5. Comandos `jj` de uso frecuente 64
Columna: las raíces de Jujutsu —¿qué clase de VCS es Mercurial? 66
Capítulo 3. El flujo de trabajo Jujutsu × IA en la práctica 69
3-1. Coordinar los agentes de IA con Jujutsu 69
3-1-1. Conseguir que los agentes de IA usen Jujutsu 69
3-1-2. Configuración de Permissions para los comandos `jj` 75
3-1-3. Ejecutar `jj fix` mediante Hooks 78
3-2. Un recorrido por el proceso de desarrollo Jujutsu × IA 84
3-2-1. Ajustar la granularidad de los changes creados por la IA 84
3-2-2. Hacer push y crear un PR 89
3-2-3. Desarrollo en paralelo con workspaces 94
Columna: las herramientas que dieron forma a Jujutsu, parte 1 103
Capítulo 4. Técnicas avanzadas de Jujutsu 106
4-1. Formas ingeniosas de especificar tus objetivos 106
4-1-1. Especificar revisiones con astucia usando revsets 106
4-1-2. Especificar archivos con astucia usando filesets 109
4-2. Comandos prácticos de usuario avanzado que conviene conocer . 111
4-2-1. `jj absorb` 111
4-2-2. `jj arrange` 112
4-2-3. `jj bookmark advance` 114
4-3. Tácticas alternativas para los Git hooks 115
4-4. Resolver conflictos de forma semiautomática 118
4-4-1. Mergiraf 118
4-4-2. Weave 122
4-5. Herramientas de UI para Jujutsu 125
4-5-1. jjui 125
4-5-2. JJ View 129
Columna: las herramientas que dieron forma a Jujutsu, parte 2 133
Capítulo 5. Guía de resolución de problemas de Jujutsu 137
5-1. FAQ 137

5-1-1. Comparación con Git 137
🤔 ¿Qué puede hacer Git que Jujutsu no? 137
🤔 ¿No hay comando merge? 138
🤔 ¿No hay comando pull? 139
🤔 Quiero hacer el equivalente al cherry-pick de Git 140
5-1-2. Operaciones y ajustes de nicho 141
🤔 ¿Puedo comprobar el contenido de un archivo en un punto dado sin mover `@`? 141
🤔 Quiero dividir un change cronológicamente 142
🤔 No quiero logs ni volcados temporales en mi historial 145
🤔 Quiero guardar la configuración de Jujutsu de un repositorio en el propio repositorio 145
🤔 Quiero renombrar un bookmark en tracking 147
5-1-3. Curiosidades sobre Jujutsu 147
🤔 ¿Es Jujutsu un wrapper de Git? 147
🤔 ¿Qué clase de persona creó Jujutsu? 149
🤔 ¿Qué significa el nombre Jujutsu y cómo se pronuncia? 150
5-2. Resolución de problemas 151
😞 Un push normal se convierte silenciosamente en un force push . 151
😞 Aparece un críptico «Error: The working copy is stale» 153
😞 Un change apareció de repente con la anotación «divergent» 155
😞 Jujutsu no rastrea mis archivos de imagen o video 157
😞 Después de hacer merge de un PR y un fetch, `@` se va por donde no debe 159
😞 Borré de GitHub un bookmark remoto en el que aún estaba trabajando 159
Columna: ¡Queremos un servicio de hosting nativo de JJ! 161

Epílogo 165

Sobre las autoras 167

Prólogo

—¡Ay, no, no, no, no...! —exclamó Kanae.

Yukina levantó la vista.

—¿A qué viene ese arrebato repentino? ¿Qué pasó?

—Tenía a Codex escribiéndome algo de código y olvidé hacer un commit de mis cambios anteriores antes de darle otra instrucción. Entonces sobrescribió mis archivos con cambios que nunca le pedí, y ahora no puedo recuperarlos...

—Ah, el impuesto de los agentes de IA. Si estuvieras en Claude Code, podrías revertirlo con el comando `/rewind`. Pero Codex CLI no tiene nada equivalente.[1]

—Últimamente Codex es más rápido y se le dan mejor los problemas difíciles, así que le he estado dando prioridad. Debí quedarme con Claude Code...

—Dicho esto, el `/rewind` de Claude Code tampoco es una bala de plata. Solo rastrea el código que escribe el agente. Si mientras tanto ejecutaste comandos de shell o hiciste ediciones manuales, esos cambios no quedan registrados. Y si intentas usar `/rewind` para retroceder varios puntos del historial, se acaba enredando con el de Git y es fácil meterse en un lío del que no puedes salir.

—Uf, o sea que ninguna de estas herramientas hace bien lo único que de verdad necesito: volver exactamente a ese momento, ¿eh?... Yukina, se te da mucho mejor que a mí trabajar con IA. ¿Tú nunca tienes este tipo de meteduras de pata?

—Yo prácticamente solo uso **Jujutsu**, así que casi nunca me topo con ese tipo de accidentes.

—**¡¿Jujutsu?!** Espera, Yukina, ¡¿tú eres una *hechicera de jujutsu*?! ¿Y encima manipulas el tiempo? Eso es fácilmente de primer grado o

[1] Estado a agosto de 2026. Codex CLI ofrecía anteriormente un comando experimental `/undo`, pero se eliminó más tarde debido a errores frecuentes y a la preocupación de que su gestión del historial entrara en conflicto con los VCS y confundiera a los usuarios.

superior, ¿no? ...Ahora que lo pienso, hasta te pareces un poco a Maki Zen'in.[2]

—Lo dices solo por los lentes, ¿verdad? No hablo del tipo de *jujutsu* que sale en *Jujutsu Kaisen*[3]. Me refiero a Jujutsu, el sistema de control de versiones de nueva generación que últimamente está dando mucho que hablar.

—Vaya, ¿existe una herramienta así? No lo sabía. En cuanto a control de versiones, yo solo conocía Git.

—Probablemente sea así para la mayoría de desarrolladores junior que entran hoy en la industria. Pero en realidad ha habido muchos VCS antes y después de Git, y el que más rápido está creciendo en popularidad ahora mismo es Jujutsu.

—Aun así, Yukina, ¿cuánto tiempo llevas usando esto de Jujutsu? Estamos en el mismo equipo y no tenía ni idea. ...Espera, pero en GitHub parece que usas Git como todo el mundo. ¿Qué pasa ahí?

—Cierto. Jujutsu es compatible con Git: puede usar directamente un repositorio de Git como su almacenamiento backend. Así que, a menos que yo lo mencione, mis compañeros de equipo ni se enteran de que uso Jujutsu. De hecho, creo que empecé a usarlo hace unos seis meses.

—Espera, ¡¿hace tanto?! ¿Te has estado guardando algo tan útil todo este tiempo? Con todo lo que hemos pasado juntas...

—Perdón, perdón. No es una herramienta que afecte directamente al trabajo en equipo, y no pensé que te fuera a interesar tanto.

—Pero si yo usara Jujutsu, podría recuperarlo igualmente aunque pase algo así, ¿verdad? ¡Estoy más que interesada! ¡Por favor, enséñame a usar Jujutsu! Yo también quiero convertirme en una hechicera de Jujutsu de primer grado y manipular el tiempo.

---

[2] Un personaje del manga *Jujutsu Kaisen*. Su energía maldita no supera la de una persona corriente, pero posee capacidades físicas sobrehumanas y combate usando diversas herramientas malditas. Como no puede ver los espíritus malditos a simple vista, normalmente lleva unos lentes especiales.

[3] Un manga de Gege Akutami, serializado en la *Weekly Shōnen Jump*. Una fantasía oscura que retrata las batallas de los hechiceros que exorcizan a los monstruos y espíritus malditos nacidos de las emociones negativas de la humanidad. La serie supera los 150 millones de copias en circulación en todo el mundo. En español el título se mantiene como *Jujutsu Kaisen*, lo que, sumado al arte marcial tradicional japonés del *jujutsu*, no hace precisamente más fácil buscar sobre este VCS.

—Bueno... está bien. Voy a sacar un rato ahora mismo para explicarte Jujutsu paso a paso.

—Espera, ¿en serio? ¿Eso no va a retrasar el proyecto?

—Si adoptar Jujutsu reduce drásticamente el tiempo que pierdes en recuperaciones, a la larga se amortiza con creces. Como líder del equipo, autorizo esto como una excepción especial.

—¡Sí! ¡Una sesión uno a uno de Jujutsu con Yukina! ¡Qué ganas!

Capítulo 1. ¿Qué clase de herramienta es Jujutsu?

1-1. ¿Encaja mal Git con el agentic coding?

—Ese accidente que acabas de tener, Kanae, no ha pasado porque fueras especialmente descuidada. Ha pasado porque Git, el VCS que usas, no se diseñó para encajar con el desarrollo moderno basado en agentes. Era un accidente que tarde o temprano tenía que ocurrir. Para ser justas, Git se diseñó hace más de veinte años, así que sería mucho pedirle que hubiera anticipado un mundo en el que los agentes de IA generan cantidades enormes de código a toda velocidad.

—...Ah. Así que era eso. ¡No es culpa mía!

—Antes de hablar de Jujutsu en sí, vamos a dejar claro dónde se queda corto Git para el desarrollo basado en agentes. Así, mientras aprendes Jujutsu, sentirás de verdad de dónde vienen su comodidad y su facilidad de uso.

El redundante modelo de tres estados de Git

—La característica principal de Git, y a la vez lo primero con lo que tropiezan quienes empiezan —dijo Yukina—, es que los cambios en tus archivos pasan por tres estados distintos antes de quedar registrados en el historial:

- **Working tree** — el lugar donde editas los archivos de verdad. También se le llama working directory.
- **Staging area** — la zona entre el working tree y el repositorio donde preparas un commit. También se le llama index.
- **Repositorio** — la base de datos que almacena todo el historial de cambios y todos los archivos.

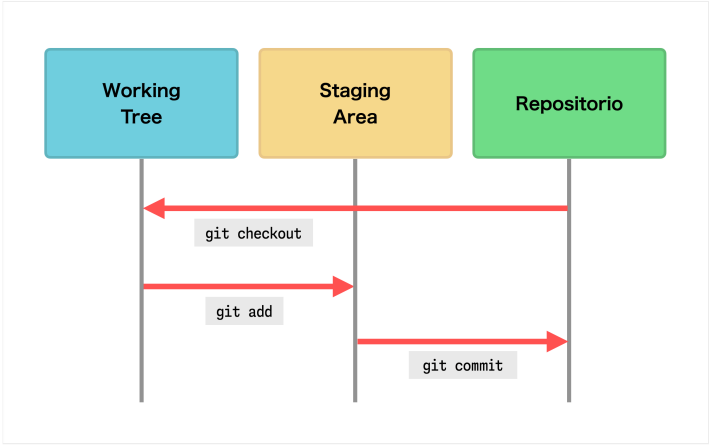

Figura 1: El modelo de tres estados de Git

—Cierto. Cuando empecé, siempre me preguntaba por qué tenía que molestarme en hacer `git add` y luego `git commit`. ¿Por qué no basta con `commit` a secas?

—En la época en que todo el desarrollo lo hacían los humanos a mano, este modelo tenía su lógica. La idea es que eliges solo algunos de los cambios de tu working tree, los pasas al staging y luego conviertes cada conjunto de cambios con sentido del staging en un commit. El staging area está pensado como un borrador de tu próximo commit. Tenerlo ayudaba a trabajar con cuidado y facilitaba producir commits de buena calidad. Recuerda que Git se creó originalmente para el desarrollo del kernel de Linux, así que soltar commits confusos cuya intención no se entiende les amargaría la vida a los mantenedores.

—Hmm, ya veo.

—Pero en el mundo de hoy, donde un agente de IA genera docenas o cientos de líneas de código repartidas en varios archivos de una sola vez, ese paso extra se ha convertido en un lastre serio. No es nada realista que un humano se ponga a ejecutar `git add` y `git commit` una y otra vez para seguirle el ritmo a una IA incansable que no para de producir código nuevo. El resultado es que la gente le da a la IA la siguiente instrucción sin haber registrado nunca el estado anterior, y luego o bien no puede volver atrás para rehacer el trabajo, o bien acaba

ahogándose en un mar de historial. Ese es justo el tipo de accidente que no deja de repetirse.

El alto costo del cambio de contexto

—En el desarrollo real —continuó Yukina—, a menudo te entran ganas de probar una idea de repente, o te cae de la nada una petición de revisión o un arreglo de bug urgente. Eso te obliga a cambiar a otro branch con el working tree todavía a medias. Cuando eso pasa en Git, echas mano de `git stash` para apartar las cosas temporalmente.

—Sí, sí, hago stash prácticamente todos los días. Es un fastidio, y si para cuando vuelves se te ha olvidado que dejaste algo en el stash, la cosa se pone fea enseguida.

—Y cuando una interrupción larga se ve interrumpida a su vez por algo aún más urgente, y acabas con varios stashes apilados, se vuelve imposible recordar qué stash guarda qué trabajo. No es nada raro toparte con un conflicto al restaurar uno y acabar descartando tus cambios de pura frustración.

—Uf, solo de imaginarlo me duele el estómago...

—Y luego está el costo de poner nombres en Git, que no es ninguna tontería. Para empezar un trabajo nuevo, primero tienes que crear un branch y darle un nombre descriptivo. Para hacer un commit necesitas un mensaje de commit, y una vez que te has decidido por uno, cambiarlo después no es imposible, pero requiere unos pasos tediosos. Incluso para algo que solo estás experimentando y podrías tirar al minuto siguiente, tienes que inventarte algún nombre antes de poder empezar. Eso añade una carga cognitiva sorprendente.

—El costo de poner nombres, ¿eh?... No me había parado a pensarlo, pero ahora que lo dices, sí que me atasco cada vez intentando decidir cómo llamar a las cosas, y me quedo bloqueada.

Reescribir el historial es complejo y peligroso

—Comparado con los tiempos en que los humanos escribían el código con cuidado a mano —prosiguió Yukina—, el costo de que una IA escriba código es muchísimo menor, así que ahora es mucho más habitual querer volver atrás para retocar algo que ya habías registrado. Pero reescribir el historial en Git es una tarea abrumadora.

—Sé a qué te refieres. Cuando de verdad tengo que editar el historial, acabo buscando por ahí o preguntándole a una IA de chat, y lo hago conteniendo la respiración todo el rato.

—En Git, un solo concepto de «reescribir el historial» está repartido entre varios comandos: `git commit --amend`, `git rebase -i`, `git reset`. Y el alcance de cada uno puede cambiar drásticamente según qué opciones le pases. Construir un modelo mental claro es difícil. Dudo que mucha gente los tenga todos memorizados y sepa elegir con confianza el adecuado para cada situación.

—Ni me lo cuentes.

—`git rebase -i` en particular tiene esa interfaz peculiar en la que editas una lista de commits en un editor de texto, y cualquier error que cometas ahí se cuela directo en tu historial. Borra una línea y ese commit desaparece; cambia `pick` por `drop` y también desaparece. En cuanto guardas y cierras el editor, el proceso empieza a ejecutarse, así que la desesperación al darte cuenta de que has editado algo mal no tiene nombre.

—Y si a mitad de camino surge un conflicto —siguió Yukina—, puedes acabar teniendo que resolver un conflicto tras otro mientras los commits se aplican de uno en uno. Para el final has perdido la cuenta de en qué commit estás siquiera, y la cosa suele reducirse a elegir entre `git rebase --abort` para empezar de cero, o aporrear `git rebase --continue` en total confusión.

—Sí, exacto. Me da tanto miedo que acabo avanzando todo lo que puedo sin hacer commit, solo para no tener que dividir ni arreglar cosas más tarde. Que es, por supuesto, justo como pasa el accidente de antes, o como acabo enviando un PR con un único commit gigante, un auténtico cajón de sastre.

Comandos con efectos secundarios graves y difíciles de revertir

—Git tiene bastantes comandos destructivos —dijo Yukina— que no puedes deshacer fácilmente una vez ejecutados. Por ejemplo, `git reset --hard`, `git clean -fd` y `git restore` barren sin piedad los cambios a los que aún no has hecho commit. Que un solo comando mal tecleado pueda borrar todo un working tree de esfuerzo que has ido acumulando con tanto trabajo es sencillamente aterrador.

—Una vez, Gemini CLI tuvo el detalle de hacer cambios que yo nunca le pedí, y cuando le dije «devuelve las cosas a como estaban», ejecutó `git restore` y borró también todo lo que había estado haciendo antes. Después me pidió perdón mil veces, pero ya era demasiado tarde.

—Con Claude Code puedes usar la configuración de Permissions para bloquear comandos concretos de Git, pero es fácil configurarla mal. El riesgo de soltar a una IA sobre Git, una herramienta donde un solo comando puede destruir de forma irreversible el trabajo que has ido acumulando, es alto.

—Además —añadió Yukina—, algún usuario avanzado de Git podría decirte: «Usa `git reflog`, con eso puedes deshacer cualquier cosa». Pero el reflog no puede restaurarlo todo en realidad. Leer el reflog es genuinamente difícil, y restaurar con seguridad al estado que querías exige un conocimiento profundo de las tripas de Git. El número de ingenieros capaces de operar el reflog con calma en pleno pánico y volver exactamente al estado que buscaban es, de nuevo, casi nulo.

Resumen: la filosofía de diseño de Git frente a la era de la IA

—Hmm —dijo Kanae—. Escuchando todo esto, Git empieza a parecerme una de esas herramientas para las que la humanidad aún no estaba preparada. ¿Por qué está diseñado así?

—Porque Git se diseñó originalmente en torno al modelo de desarrollo del kernel de Linux: «**los humanos escriben código a mano, organizan sus cambios, los agrupan en unidades listas para revisar y los comparten con cuidado**». Para un flujo de trabajo en el que los humanos se pasan los cambios entre sí con cuidado, es una herramienta fenomenal, y se ha ganado su fama de sobra a lo largo de los años. Pero en el desarrollo basado en agentes el panorama es bastante distinto:

- El ensayo y error rápido es constante.
- Enormes cantidades de código cambian de golpe.
- El contexto de trabajo cambia con frecuencia.
- Surge más a menudo la necesidad de ordenar el historial a posteriori.

—En condiciones así, los mismos rasgos que hacían que Git funcionara tan bien acaban frenando tu desarrollo. Esa es la razón de fondo

por la que los puntos débiles de Git se han hecho tan evidentes en la era de la IA.

—Ajá.

—Lo que el agentic coding realmente necesita es una herramienta en la que primero puedas montar un lío y luego ordenarlo con seguridad. Y es justo en ese punto donde un sistema de control de versiones construido sobre una filosofía de diseño distinta empezó a llamar la atención.

—¡Ah, eso es Jujutsu! ¿Verdad?

1-2. Las características de Jujutsu que destacan en la era de la IA

—No es que no confíe en ti, Yukina, pero ¿de verdad la popularidad de Jujutsu está subiendo tan rápido ahora mismo? Ni siquiera había oído hablar de un VCS así hasta que lo mencionaste.

—Bueno, a mediados de 2026 todavía está en la fase en que un grupo reducido de desarrolladores que siguen de cerca las herramientas nuevas lo respaldan con pasión. En términos de la teoría del abismo (chasm)[4], se está extendiendo entre los early adopters, pero aún no ha cruzado el abismo.

—Me encantaría enseñarte datos sólidos —continuó Yukina—, pero la verdad es que no hay una buena métrica objetiva de la popularidad de un VCS. Git ha sido tan dominante durante tanto tiempo que las encuestas a desarrolladores ni se molestan en preguntar «¿qué VCS usas?». No es una comparación del todo justa, pero echemos un vistazo al historial de estrellas de GitHub de los repositorios oficiales.

[4] Una teoría de marketing basada en el ciclo de adopción de tecnología, un modelo sociológico de cómo se difunden los nuevos productos e innovaciones. Divide el proceso de difusión en cinco segmentos de clientes e identifica el «abismo» (chasm), una profunda brecha entre el mercado inicial y el mercado mayoritario, como el obstáculo más crítico que hay que superar.

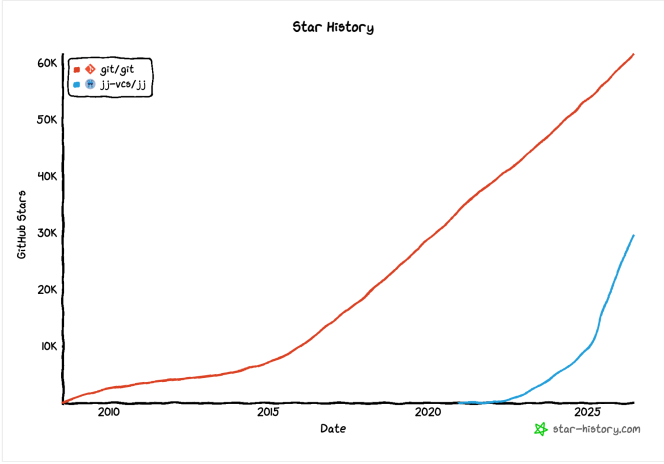

Figura 2: Evolución de las estrellas de GitHub de Git y Jujutsu (GitHub Star History, junio de 2026)

—¡Guau! La curva de Jujutsu sube como un palo de hockey. Despegó de verdad a mediados de 2025. A este ritmo, parece que el día en que adelante a Git no está tan lejos. Pero ¿qué quieres decir con «no del todo justa»?

—Git no se desarrolla realmente en GitHub. Ahí solo hay un repositorio espejo. El oficial está alojado en kernel.org, el mismo sitio que el kernel de Linux[5]. Jujutsu, en cambio, tiene su repositorio oficial en GitHub.

—Bueno, para la mayoría de los usuarios GitHub lo es prácticamente todo, así que yo diría que es un barómetro fiable de la popularidad. Así que entiendo que Jujutsu está ganando terreno rápidamente. ¿Qué hay detrás de ese despegue?

—Kanae, acabas de decir que empezó a despegar a mediados de 2025. ¿Recuerdas qué pasó por entonces?

—...Mmm, ¿qué era?

—Claude Code se lanzó oficialmente en mayo. Hasta ese momento, las herramientas de IA para programar eran sobre todo de tipo IDE, centradas en cambios de código sencillos mediante sugerencias y chat. Claude Code es una herramienta de IA para programar de tipo agente

[5] https://git.kernel.org/pub/scm/git/git.git

que funciona en la terminal: le das instrucciones y genera de forma autónoma código de alta calidad en una sola ejecución.

—Con Claude Code en escena —prosiguió Yukina—, el mundo se inclinó hacia un estado en el que la IA escribía muchísimo más código que los humanos. Para no quedarse atrás, Codex CLI y Gemini CLI siguieron su ejemplo, Cursor viró a un modelo basado en agentes, y la tendencia hacia el desarrollo basado en agentes se volvió decisiva.

—Cierto. Ahora parece historia antigua, pero hasta hace bien poco escribíamos cada línea a mano.

—El auge de Jujutsu parece ir arrastrado por la expansión de los agentes de codificación de IA como Claude Code y Codex. Yo misma me decidí a probar Jujutsu después de leer, por esa época, una tanda de artículos que lo presentaban en el contexto del desarrollo basado en agentes. Algunos de los más leídos son:

- Use Jujutsu, Not Git: Why Your Next Coding Agent Should Use Jujutsu[6]
- Avoid Losing Work with Jujutsu (jj) for AI Coding Agents[7]
- Why Jujutsu (jj) Is Perfect for AI-Generated Code[8]
- Towards an AI-Native Development Workflow (Using Jujutsu as the Backbone)[9]

—Vaya. Cuando los expertos te dicen que Jujutsu y la IA encajan tan bien, no puedes evitar querer probarlo. Pero hay algo que no entiendo. Jujutsu ya existía antes de que despegara el desarrollo basado en agentes, ¿no? No se diseñó para eso, así que ¿por qué encaja tan bien?

—El desarrollo de Jujutsu empezó en 2019 y, en aquel momento, casi nadie podía predecir un futuro en el que la IA programara de forma autónoma, así que por supuesto no se diseñó pensando en eso. Según su autor, Martin von Zweigbergk, sus objetivos eran **reducir la carga cognitiva para los humanos** y **acabar con el complejo modelo mental de Git**[10].

---

[6] https://slavakurilyak.com/posts/use-jujutsu-not-git

[7] https://www.panozzaj.com/blog/2025/11/22/avoid-losing-work-with-jujutsu-jj-for-ai-coding-agents/

[8] https://cesar.velandia.co/why-jujutsu-jj-is-perfect-for-ai-generated-code/

[9] https://ianbull.com/posts/jj-vibes

[10] «Solving Git's Pain Points with Jujutsu (with Martin von Zweigbergk) - YouTube»
https://www.youtube.com/watch?v=ulJ_Pw8qqsE

—Los agentes de IA iteran a gran velocidad sin miedo a equivocarse —agregó Yukina—, y trabajar codo con codo con ellos exige muchísimo del cerebro humano. Así que el diseño de Jujutsu, que se propuso reducir esa carga cognitiva de los humanos, acabó encajando de forma natural con esta situación como efecto secundario.

—Hmm, ya veo.

—Pues vamos a repasar, una a una, las características de Jujutsu que están llamando la atención en la era de la IA.

AI working copy se le hace commit automáticamente

—A diferencia de Git, Jujutsu no tiene staging area —dijo Yukina—. Solo hay dos estados: el **working copy** (que se corresponde con el working tree de Git) y el **repositorio**. Y aquí está la mayor característica de Jujutsu: los cambios en los archivos del working copy se confirman automáticamente. Sin que el usuario ejecute nada parecido a `git commit`, se toma un snapshot del working copy y queda registrado en el repositorio.

—¿Qué? ¿No tienes que hacer add, y ni siquiera tienes que hacer commit? ¿Cómo se consigue eso en la práctica? ¿Hay un daemon[11] vigilando el estado de los archivos o algo así?

—No. El sistema de comandos de Jujutsu usa `jj` como comando principal, con subcomandos como `jj log` y `jj diff`, y se toma un snapshot en el momento en que se ejecuta un comando `jj`. Si hay una diferencia respecto al snapshot anterior, se hace commit de esa diferencia.

[11] Un programa que se ejecuta de forma continua en segundo plano en un sistema UNIX o Linux (u otro SO similar) y que gestiona automáticamente tareas concretas, como atender peticiones web o gestionar la red.

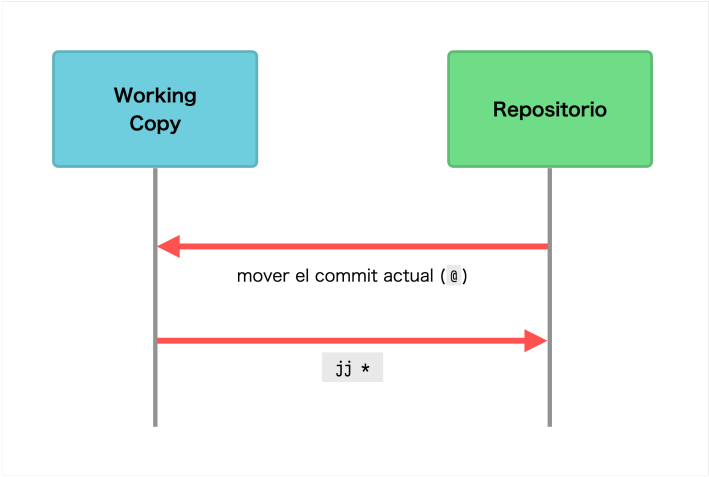

Figura 3: El modelo de dos estados de Jujutsu

—Vaya. Está bien que los errores de olvidarte de hacer add o de hacer commit simplemente desaparezcan. Pero si no controlas cuándo se hacen los commits, ¿no acaba tu historial hecho un caos?

—Te lo explicaré con más detalle luego, cuando lo pruebes de verdad, pero Jujutsu tiene un concepto llamado **change** que envuelve los commits como una sola unidad. Dicho a grandes rasgos por ahora: un change es como un contenedor que guarda varios commits en orden cronológico. El historial de Jujutsu se muestra normalmente por change, y ver los commits individuales que hay dentro de uno requiere un paso extra. Poner nombres, ajustar la granularidad y reordenar se hace todo a nivel de change. Así que no tienes que preocuparte de que los commits se acumulen por su cuenta y te ensucien el historial.

—Ah, ya veo cómo lo resuelven.

—La documentación oficial recoge esta propiedad con el término **working-copy-as-a-commit**. Gracias a su sencillo modelo de dos estados y a esta propiedad, Jujutsu no necesita nada que se corresponda con `git add`, `git commit` o `git stash`. Comparada con los VCS tradicionales, la carga cognitiva en torno a gestionar el estado de los archivos es muchísimo menor.

Toda operación se puede deshacer

—Cuando usas Git en el día a día —dijo Yukina—, ¿no tienes esos momentos de «uf, cómo me gustaría deshacer ese último comando»?

—A todas horas. La verdad es que me da rabia que no haya un deshacer como en un editor.

—Jujutsu tiene exactamente eso: el **universal undo**. Cualquier cosa que toque un remoto no se puede rebobinar a la perfección, claro, pero toda operación local se puede deshacer y rehacer.

—¡Es increíble! Solo por eso ya me dan ganas de cambiarme.

—Además del log de commits habitual, Jujutsu tiene un **operation log**. Es un registro de cada comando `jj` que has ejecutado, enlazado con los snapshots del working copy. Así que, más allá del simple deshacer, puedes rebobinar al estado del momento en que se ejecutó cualquier comando concreto, o tachar una única operación del historial como si nunca hubiera ocurrido.

—En Git siempre ejecuto el rebase y demás comandos que editan el historial con mucho cuidadito, pero con el deshacer puedo adoptar un enfoque de «pruébalo, y si no sale, deshaz y vuelve a intentarlo». Eso da tranquilidad.

—Exacto. Una vez que te liberas del miedo a hacer algo irreversible, el listón para experimentar baja drásticamente. En Git, conseguir que un agente de IA ejecute comandos que editan el historial da demasiado miedo como para intentarlo en serio, pero con Jujutsu no es ningún problema. La sensación de seguridad psicológica está a un nivel completamente distinto. Una vez que la has probado, no puedes volver atrás.

Los conflictos se tratan como un estado más

—Lo que da miedo en Git —dijo Yukina— son los conflictos durante un merge, un rebase o un cherry-pick. Cuando se produce un conflicto en Git, el resto de operaciones quedan estrictamente bloqueadas, y te ves obligada a parar de trabajar hasta que lo resuelves.

—Da miedo, sí. Cada vez que incorporo los últimos cambios del branch main antes de abrir un PR, rezo para que no salga ningún conflicto.

—Jujutsu, en cambio, trata **un conflicto como un estado más**. Cuando se produce uno, se le hace commit tal cual y no bloquea ninguna operación. Por ejemplo, si surge un conflicto durante un rebase, el rebase termina igualmente, arrastrando el conflicto consigo. La documentación oficial llama a esto **first-class conflicts**.

—Espera, ¿y eso es bueno? ¿Un conflicto no es algo que haya que resolver de inmediato? Dejarlo ahí no me deja tranquila.

—Eso de que «un conflicto hay que resolverlo de inmediato» es una suposición propia de la mentalidad Git. La razón por la que Git trata un conflicto sin resolver como una «excepción» que bloquea no es una necesidad técnica, sino una restricción de su implementación. Deja que te dé unos cuantos casos concretos en los que los first-class conflicts se sienten de maravilla:

- Estás haciendo un rebase de un branch, y resolver un conflicto requiere consultar la intención de la implementación con Alice, la compañera responsable de ese código. Pero está de vacaciones y no vuelve hasta mañana. En Git tendrías que abortar el rebase hasta entonces. En Jujutsu puedes terminar el rebase por ahora y encargarte de la resolución más tarde, o seguir adelante con un parche improvisado.
- Quieres dejar que una IA se encargue de resolver varios conflictos que surgieron durante un merge. En Git no puedes hacer commit a menos que resuelvas todos los conflictos, así que, si se equivoca en alguno, restaurar un único conflicto concreto y volver a verificarlo es incómodo. En Jujutsu puedes localizar y restaurar un solo conflicto y rehacer la resolución tantas veces como haga falta.

—Ah, ya veo. La verdad es que suena útil. Que una IA resuelva conflictos habría sido impensable en Git. Pero con Jujutsu puedes hacer que los resuelva de uno en uno, y si se equivoca en alguno, basta con deshacer y volver a intentarlo. Eso hace mucho más fácil delegar el trabajo sin echarse atrás.

—Además, en Jujutsu, cuando editas un commit anterior, los cambios de ese archivo se propagan automáticamente a sus commits descendientes. Así que, incluso al resolver un conflicto, basta con moverte al commit donde se produjo y editar el archivo ahí, sin tener que lanzar un rebase a mano. Como aplazar la resolución apenas cuesta nada después, la carga psicológica es pequeña.

Comandos ortogonales y con poca dependencia del estado

—Una cosa que noto al usar Jujutsu —dijo Yukina— es que los comandos son muy fáciles de recordar. Para los comandos cuyo objetivo es el historial en sí, el comando se corresponde uno a uno con el verbo de lo que quieres hacer. Por ejemplo, descartar historial es `jj abandon`, dividir es `jj split`, combinar es `jj squash`. No hay nada como `git checkout` o `git reset`, donde las opciones o los argumentos cambian de raíz lo que hace el comando. La operación que quieres se corresponde directamente con un comando, así que nunca tienes que preguntarte «¿cuál era ese comando?».

—Ajá, que haya menos sobrecarga de comandos es genial.

—Los comandos cuyo objetivo no es el historial en sí también están organizados en una jerarquía lógica clara. Por ejemplo, en Git las operaciones sobre branches están desperdigadas: crear uno es `git branch <branch-name>` o `git checkout -b <branch-name>` o `git switch -c <branch-name>`, listar es `git branch`, borrar es `git branch -d <branch-name>`. No es que sea muy coherente. En Jujutsu es `jj bookmark create`, `jj bookmark list`, `jj bookmark delete`. «Sobre qué actúas y qué haces» se lee directamente en la estructura del comando.

—Tienes razón, nunca conseguía recordar qué comando de branch de Git hacía qué. Lo buscaba cada vez. Pero si los comandos de Jujutsu funcionan así, hasta yo podría memorizarlos.

—Encima, Jujutsu comparte las convenciones básicas de opciones y de objetivo entre muchos comandos. La opción para indicar un punto concreto del grafo del historial es `-r/--revision`, y `-f/--from`, `-t/--into` y `-o/--onto` tienen el mismo significado en todos los comandos que editan el historial. Así que transfieres con facilidad lo que sabes de un comando a otro.

—Ya veo. Con Git da como la sensación de que tienes que memorizar un conjuro distinto para cada comando.

—En diseño de software, cuando aprender sobre un elemento te permite extrapolar a otros elementos, decimos que tiene «alta ortogonalidad». El sistema de comandos de Jujutsu es, efectivamente, muy ortogonal. Después de usarlo unas cuantas veces, te descubres tecleando «esto probablemente se escribe así...» y acertando.

—Nunca había oído la palabra «ortogonal», pero, bueno, básicamente significa simple e intuitivo, ¿no?

—Otro punto que vale la pena mencionar es que los comandos de Jujutsu dependen muy poco del estado. Los comandos de Git siempre parten de HEAD[12], y les afecta el estado del working tree y del staging area, así que ejecutar el mismo comando en situaciones distintas puede dar resultados muy diferentes. En Jujutsu no hay staging area y a los cambios se les hace commit automáticamente, así que el working copy es solo un nodo más entre muchos en el grafo del historial. Y como muchos comandos pueden tomar un punto concreto del grafo del historial como argumento, puedes operar directamente sobre esos puntos sin importar dónde esté el working copy.

—Hmm, ¿como por ejemplo?

— `jj squash` es el comando que combina historial. Si indicas explícitamente el origen y el destino con `jj squash -f <from-id> -t <to-id>`, el resultado es el mismo sin importar en qué punto del grafo se encuentre el working copy. Si intentaras hacer lo mismo en Git, primero tendrías que hacer `checkout` al lugar de destino.

—Ah, tiene sentido. Pero el enfoque de Jujutsu de indicar siempre el objetivo con una opción tiene la desventaja de más pulsaciones de teclas, ¿no?

—Por eso muchos comandos tienen valores por defecto cuando se omite una opción. Ejecutar `jj squash` sin argumentos combina el working copy con su padre. También puedes indicar solo `-f` o solo `-t`.

—Ya veo. Por cierto, ¿qué tiene de bueno que dependan poco del estado?

—Que puedes hacer la operación que quieras desde donde estés. No hace falta andar moviendo el working copy de un lado a otro. También te liberas de la carga cognitiva de tener que entender con precisión el estado actual antes de cada operación, e indicar el objetivo de forma explícita reduce los errores de manejo. Una vez que te acostumbras a Jujutsu, los comandos ortogonales y con poca dependencia del estado empiezan a resultar tan naturales que dejas de notarlos. Pero de vez en

[12] En Git, un puntero que indica dónde estás trabajando en cada momento. Normalmente apunta al último commit del branch actual.

cuando, cuando tienes que volver a Git para algo, sientes de verdad lo mucho más cómodo que es Jujutsu.

Resumen: Jujutsu es un VCS en el que empiezas en sucio y ordenas después

—Si juntas todas estas características —dijo Yukina—, empieza a asomar la filosofía que hay bajo Jujutsu. El estilo es: «empieza en sucio sin más, experimenta a tu aire y ordénalo después».

- Al working copy se le hace commit de forma automática y continua, así que no tienes que pensar en si tus ediciones están guardadas.
- No tienes que apuntar a una granularidad de cambios perfecta ni dar con los nombres adecuados desde el principio. Cuando lo necesites, puedes dividir, combinar y reordenar los changes, y renombrarlos para que reflejen cómo acabaron siendo las cosas.
- Toda operación se puede deshacer, así que puedes experimentar sin miedo a equivocarte.
- Los conflictos no son excepciones que bloquean; son un estado que un commit puede mantener. Puedes aplazar la resolución y seguir trabajando.
- Los comandos, ortogonales y con poca dependencia del estado, te permiten remodelar el historial sobre la marcha, cada vez que te llega la inspiración.

—Ya veo... Es justo lo contrario de Git, donde tienes que dejarlo todo clavado desde el principio y cualquier cambio posterior es un engorro.

—Este estilo de Jujutsu tiene mucho en común con el desarrollo ágil, que parte de que los humanos cometen errores y busca elevar la calidad mediante pequeños ajustes repetidos.

—Ah, es verdad.

—Los agentes de IA se equivocan al menos tanto como los humanos, e iteran y generan código a velocidades que los humanos no pueden igualar. La afinidad de Jujutsu con los agentes de IA no es una casualidad. Es la consecuencia natural de diseñar sin descanso pensando en el cerebro humano.

—Amable con los humanos y con la IA... Ahora sí que me pica la curiosidad. ¡Quiero probarlo cuanto antes!

Columna: cómo Git revolucionó el control de versione

El capítulo 1 ha construido su argumento desde el ángulo de que el diseño de Git ha quedado desactualizado con el tiempo. Pero no sería justo detenerse ahí sin reconocer también el enorme papel que Git ha desempeñado en la historia del control de versiones. Una de las motivaciones detrás del desarrollo de Jujutsu era resolver los puntos débiles de Git. Dándole la vuelta, eso significa sencillamente que Jujutsu es un producto construido a hombros de un gran predecesor.

Git empezó a ganar terreno en el sector a principios de la década de 2010. Antes de eso, el control de versiones significaba por lo general un producto centralizado, y Subversion[13] en particular gozaba de amplio uso. Como todos compartían un único repositorio central, muchas operaciones cotidianas dependían del servidor. Los commits y la consulta del log, en particular, exigían hablar con el repositorio central, y el trabajo sin conexión estaba muy limitado. Sacar adelante cualquier trabajo de peso sin conexión era básicamente inviable y, como un commit se compartía al instante con el resto del equipo, no podías registrar el historial en pasos finos mientras tu trabajo seguía a medias. Así que no era raro ver flujos de trabajo que despojaban a un VCS de la mitad de su valor: acumular un montón de cambios en local y hacer un único commit solo cuando por fin todo estaba listo para publicar, o meter todo el trabajo de un día en un solo commit a la hora de salir.

La expansión de Git cambió todo eso de la noche a la mañana. Una copia casi completa del historial vive siempre en la propia máquina del usuario, y la mayoría de las operaciones cotidianas se completan en local. Como haces commit contra tu propio repositorio local, puedes dar forma a tu historial con la granularidad que quieras, enteramente a tu criterio. E incluso si el servidor se cae o estás trabajando sin conexión, puedes seguir adelante.

[13] https://subversion.apache.org/

En Git, el repositorio central no es conceptualmente más que un punto de sincronización para compartir el trabajo entre los miembros del equipo; cada miembro tiene un repositorio con un historial casi completo propio. Aunque el repositorio central se corrompa, se puede restaurar a partir del repositorio de algún miembro. En Subversion los archivos de la máquina de un miembro estaban subordinados al repositorio central, pero en Git cada miembro tiene un repositorio que está en pie de igualdad con el central. Fue un cambio que transformó la estructura de poder del desarrollo y la propia mentalidad de quienes participaban en él.

Git era, además, vertiginosamente rápido en toda clase de operaciones comparado con los VCS existentes de su época. En parte porque la mayoría de las operaciones se completan en local, pero un factor importante es que su objetivo de diseño era el kernel de Linux, así que desde la fase de diseño fijó requisitos de rendimiento en torno a cosas como el poco volumen de transferencia por red y la velocidad de la consulta del log y de los merges. Donde los VCS convencionales registraban las sucesivas diferencias de un archivo desde su estado inicial, Git registra un snapshot de todo el árbol de archivos en el momento del commit. Para extraer una versión concreta, simplemente lee directamente el puntero del commit correspondiente, lo que lo hace extremadamente rápido.

Git también cambió lo que significa un branch. En Subversion, un branch era simplemente una copia de todo el proyecto creada en un directorio aparte, así que ramificar acarreaba una sobrecarga y una fricción notables. Encima, su modelo de datos seguía mal los merges, lo que hacía probables conflictos innecesarios, y a muchos equipos les disgustaba lo suficiente como para evitar los branches por completo. En Git, en cambio, el historial en sí se estructura como un DAG[14], y un branch no es más que un puntero a un commit concreto: un diseño notablemente elegante[15]. Los branches de Git son ligeros de manejar, producen menos conflictos al hacer merge

[14] Sigla de Directed Acyclic Graph (grafo acíclico dirigido): una estructura de grafo en la que los nodos conectados por aristas dirigidas nunca forman un ciclo.

[15] Se explica en «2-4-2. La diferencia entre branch y bookmark».

y se pueden crear puramente en local, todo lo cual redujo drásticamente su costo psicológico. Estilos de desarrollo como crear un feature branch para cada pequeña funcionalidad, trabajar en varios branches en paralelo o usar un branch local como espacio de pruebas personal solo se volvieron posibles una vez que Git se impuso.

Se podría decir que Git cambió el estatus mismo del control de versiones: de algo que «había que usar» bajo el control de una empresa u organización, a algo que uno «quería dominar por voluntad propia» para aumentar su propia productividad. La llegada de Git democratizó el control de versiones en sí y, a mi juicio, puso el historial y su poder por igual en manos de cada desarrollador.

Capítulo 2. Probemos Jujutsu

2-1. Configurar Jujutsu

2-1-1. Instalar Jujutsu

—De aquí en adelante quiero que aprendas Jujutsu usándolo de verdad —dijo Yukina—. Pero lo primero es lo primero: tenemos que instalarlo. Como Jujutsu está escrito en Rust, la documentación oficial abre con un procedimiento basado en Cargo, pero, a menos que seas una fan acérrima de Rust, es mejor que evites esa vía. La compilación tarda un buen rato y mantenerlo actualizado se vuelve engorroso.

—Ya —dijo Kanae—. No tengo pensado tocar Rust en un futuro próximo, así que preferiría no empezar montando todo un entorno de Rust solo para esto.

—Por suerte, las dos estamos en Mac, así que podemos instalarlo fácilmente con Homebrew, así:

```
$ brew install jj
```

—Esta parte no es obligatoria, pero añadir un poco de configuración de shell hace más cómodo ejecutar comandos. Según qué shell uses, añade una de estas líneas a tu archivo de configuración:

```
# Para zsh
source <(COMPLETE=zsh jj)

# Para fish
COMPLETE=fish jj | source
```

—¿Y esto qué hace en realidad?

—Cuando ejecutas `jj` con la variable de entorno `COMPLETE=zsh` definida, imprime un script de autocompletado de shell para zsh. Al hacer que zsh cargue ese script al arrancar, las definiciones de autocompletado más recientes del comando `jj` se generan y se aplican de forma dinámica. Así, cuando escribes parte de un subcomando o de

una opción y presionas `Tab`, te muestra una lista de candidatos o te autocompleta el comando.

—Ah, eso suena práctico. Yo también lo añadiré a mi configuración. Por cierto, en casa tengo un equipo con Windows. ¿Cómo lo instalaría ahí?

—En un entorno WSL[16], lo ideal es Homebrew, igual que en un Mac. Para un entorno nativo, el paquete está registrado en WinGet como `jj-vcs.jj`, así que instala ese. Ten en cuenta que hay algunas particularidades en torno a los saltos de línea y los enlaces simbólicos, así que léete con calma la página correspondiente de la documentación oficial[17].

—Entendido.

—Una cosa más, no directamente relacionada con Jujutsu: si aún no has instalado la GitHub CLI (`gh`)[18], este es un buen momento. Dejar que un agente de IA ejecute el comando `gh` junto con `jj` hace que tu flujo de trabajo sea bastante más eficiente.

2-1-2. Configuración inicial

—Jujutsu tiene un montón de ajustes configurables —prosiguió Yukina—, pero al principio solo importan unos pocos. Primero, vamos a registrar tu nombre de usuario y tu dirección de correo.

```
$ jj config set --user user.name "kanaetti"
$ jj config set --user user.email "kanaetti@skydome.co.jp"
```

— `jj config set`[19] es el comando para escribir valores en tu archivo de configuración. Con `--user` apunta a los ajustes a nivel de usuario; con `--repo`, a los ajustes limitados a ese único repositorio.

---

[16] Sigla de Windows Subsystem for Linux. Una funcionalidad de Microsoft para ejecutar entornos Linux en Windows. El WSL moderno usa una máquina virtual ligera con un kernel de Linux real, lo que te permite instalar y usar distribuciones como Ubuntu o Debian.

[17] "Working on Windows - Jujutsu docs" https://docs.jj-vcs.dev/latest/windows/

[18] https://cli.github.com/

[19] "`jj config set` - CLI reference - Jujutsu docs"
https://www.jj-vcs.dev/latest/cli-reference/#jj-config-set

—¿Qué pasa si empiezas a usarlo sin configurar esto?

—Faltará tu información de autor en el historial y no podrás hacer push a un remoto. Entonces tendrías que configurar tu información de usuario a posteriori y ejecutar `jj metaedit --update-author <TARGET>`[20] para actualizar la información de autor en tu historial.

—Eso suena a engorro. Me aseguraré de no olvidarlo en una instalación nueva.

—Cambiemos también el ajuste del editor. De fábrica, el editor que Jujutsu abre cuando lo necesita es nano, que a la mayoría de la gente no le resulta familiar[21].

```
# Para Vim
$ jj config set --user ui.editor "vim"

# Para VS Code
$ jj config set --user ui.editor "code --wait"
```

—¿Qué es esa opción `--wait` que le pasas al abrir VS Code?

—Cuando Jujutsu abre un editor, necesita detectar cuándo se cierra el editor para poder recoger lo que has escrito. Si abres un editor con interfaz gráfica desde la terminal, lo normal es que se lance como un proceso aparte, y tus modificaciones nunca se reflejarán en Jujutsu. Añadir `--wait` hace que VS Code mantenga bloqueado el proceso de la terminal hasta que cierras la pestaña del editor, de modo que Jujutsu puede reconocer que has terminado y recibir tus ediciones correctamente.

—Ah, ya veo, tiene sentido.

—Por cierto, si no puedes abrir VS Code escribiendo `code` en la terminal, abre la paleta de comandos con `⌘` + `Shift` + `p`, escribe «shell» y ejecuta «Shell Command: Install `code` command in PATH» de entre los resultados. Con eso funcionará.

---

[20] "`jj metaedit` - CLI reference - Jujutsu docs"
https://www.jj-vcs.dev/latest/cli-reference/#jj-metaedit

[21] El comportamiento en macOS y en Linux basado en Ubuntu cuando la variable de entorno `$EDITOR` no está definida. En Windows se abre el Bloc de notas en su lugar.

—Ahora, con eso configurado —prosiguió Yukina—, ejecutemos `jj config edit --user`[22]. El editor que indicaste antes debería abrirse y mostrar algo como esto:

```
#:schema https://docs.jj-vcs.dev/latest/config-schema.json

[user]
name = "kanaetti"
email = "kanaetti@skydome.co.jp"

[ui]
editor = "vim"
```

—¿Y dónde vive en realidad este archivo? Eso también puedes averiguarlo con `jj config`.

```
$ jj config path --user
/Users/kanaetti/.config/jj/config.toml
```

—Entonces, editar este archivo directamente también aplicará los ajustes, ¿no?

—Claro. Así que, si guardas este archivo en un directorio compartido como Dropbox y creas un enlace simbólico a él, puedes compartir los ajustes de usuario de Jujutsu entre varias máquinas.

2-2. Un recorrido práctico por Jujutsu

2-2-1. Inicializar un repositorio

—Muy bien, vamos a empezar a poner archivos bajo control de versiones con Jujutsu —dijo Yukina—. Una app de React es un buen ejemplo para aprender, creo. Usaremos Node.js como runtime y pnpm para gestionar los paquetes, así que, si aún no los tienes, configúralos primero. Y para instalarlos, usaremos mise[23].

---

[22] "`jj config edit` - CLI reference - Jujutsu docs"
https://www.jj-vcs.dev/latest/cli-reference/#jj-config-edit

[23] Un gestor de paquetes para instalar entornos de ejecución de lenguajes de programación y diversas herramientas de CLI, gestionando varias versiones de cada uno. También hace las veces de ejecutor de tareas y de gestor de variables

```
$ brew install mise
$ mise use -g node pnpm
```

—Ahora vamos a crear el proyecto de React, inicializarlo y ponerlo bajo la gestión de Jujutsu.

```
$ pnpm create vite jj-tour
◇  Select a framework:
│  React
◇  Select a variant:
│  TypeScript
◇  Which linter to use?
│  ESLint
◇  Install with pnpm and start now?
│  No
◇  Scaffolding project in /Users/kanaetti/projects/jj-tour...
└  Done.

$ cd jj-tour/
$ pnpm install
$ jj git init
Initialized repo in "."

$ ls -d .*
.git/       .jj/       .gitignore
```

—¿El comando para inicializar es `jj git init`[24], no `jj init`?

—Jujutsu usa una arquitectura de almacenamiento intercambiable y puede usar Git como su almacenamiento backend. Así que este comando inicializa un repositorio de Git con Jujutsu montado encima. Algunos artículos más antiguos sugieren usar la opción `--colocate` para que los repositorios de Jujutsu y de Git convivan en el mismo directorio, pero hoy en día ya es el comportamiento por defecto, así que no la necesitas.

—¿Y si tengo un repositorio que ya está gestionado con Git y quiero empezar a usar Jujutsu en él más adelante?

---

de entorno.
https://mise.jdx.dev/

[24] "`jj git init` - CLI reference - Jujutsu docs"
https://www.jj-vcs.dev/latest/cli-reference/#jj-git-init

—El mismo comando: `jj git init` funciona. El historial de commits que creaste con Git se importa tal cual como historial de Jujutsu.

2-2-2. Comprobar el estado del repositorio

—A continuación, comprobemos el estado actual de este repositorio —dijo Yukina.

```
$ jj status
Working copy changes:
A .gitignore
A README.md
A eslint.config.js
A index.html
⋮
A vite.config.ts
Working copy  (@) : zvxsqvvl 624524a7 (no description set)
Parent commit (@-): zzzzzzzz 00000000 (empty) (no description set)
```

—Puedes pensar en `jj status`[25] como algo que cumple más o menos la misma función que `git status`. La diferencia es que `git status` sirve para comprobar el estado de tu working tree y tu staging area, aún sin commit, mientras que `jj status` sirve para comprobar el estado de tu working copy, al que ya se le ha hecho commit. Esto es lo que te muestra el comando:

- Un resumen de qué cambios se hicieron en el working copy
- Información sobre el commit y su commit padre
- Si hay un conflicto, información sobre él

—Hay una lista entera de nombres de archivo, todos los que se crearon al montar el proyecto, alineados con una `A` delante de cada uno. Según lo que acabas de explicar, ¿eso significa que ya se les ha hecho commit?

—Exacto. Así que echemos un vistazo al historial de ese commit.

[25] "`jj status` - CLI reference - Jujutsu docs"
https://www.jj-vcs.dev/latest/cli-reference/#jj-status

Capítulo 3. El flujo de trabajo Jujutsu × IA en la práctica

3-1. Coordinar los agentes de IA con Jujutsu

3-1-1. Conseguir que los agentes de IA usen Jujutsu

—Últimamente he empezado a dejar que la IA se encargue también de Git por mí —dijo Kanae—, o sea, le digo sin más «bueno, haz commit de lo que llevamos hasta ahora y súbelo». ¿De verdad se pueden delegar las operaciones de Jujutsu en una IA así?

—Claro que se puede —respondió Yukina—. Jujutsu te permite retroceder en el historial con facilidad y seguridad, así que puedes delegar el control de versiones con más audacia que con Git. Y como no hay staging y los cambios se registran en un commit de forma automática, puedes configurarlo para que baste con encargar una tarea y te quede un historial organizado en unidades con sentido, sin tener que ir detallando los pasos de control de versiones.

—Eso suena genial. Entonces, ¿has estado combinando Jujutsu con el desarrollo basado en agentes todo este tiempo? Yo habría dicho que un agente de IA solo echa mano de Git por defecto, así que ¿qué has ido haciendo para que use Jujutsu? Ese conocimiento me sería valiosísimo, así que ¡me encantaría que lo compartieras!

—Claro. La cosa es que los agentes de IA se diferencian bastante en sus funcionalidades y sus manías, y se les añaden capacidades nuevas a un ritmo vertiginoso, así que todavía cuesta fijar las buenas prácticas. Entre los agentes de codificación, Claude Code tiene la mayor cuota de mercado y yo soy una usuaria intensiva, así que me centraré en lo que creo que es el mejor enfoque para él a fecha de agosto de 2026. También mencionaré Codex CLI cuando venga al caso.

—Sí, por favor. Esos son básicamente los dos que yo también uso a diario.

—Muy bien. Empecemos por cómo orientar a un agente de IA hacia Jujutsu en lugar de Git como su VCS. En realidad, los modelos más recientes que hay detrás de Claude Code y de Codex ya entienden los

conceptos y los comandos básicos de Jujutsu, así que con escribir «usa Jujutsu, no Git» en `CLAUDE.md` o `AGENTS.md` ya consigues que funcione hasta cierto punto.

—Ah, ¿ya lo conocen?

—Aun así, la inmensa mayoría de los datos con los que se entrenaron para tareas de programación proceden de entornos con Git, así que un flujo de trabajo nativo de Jujutsu no les sale de forma natural. Con solo esa instrucción tan simple, te puedes encontrar con cosas como estas:

- Sustituir mecánicamente cada comando `git` por el equivalente `jj` que más se le parezca
- Empezar una tarea sin comprobar en qué estado está el working-copy commit, con lo que trabajos distintos acaban mezclados en el historial
- Tratar un bookmark como un branch de Git, que avanza automáticamente con cada commit nuevo
- Manejar mal los bookmarks a la hora de hacer push
- Detener el trabajo en cuanto aparece un conflicto
- No saber reaccionar cuando surge un problema propio de Jujutsu

—Conocer los comandos no es lo mismo que haber pasado por un flujo de trabajo completo y haber desarrollado el criterio necesario —dijo Yukina—. Un modelo con buen razonamiento puede resolver bien algunos casos, pero apoyarse en eso hace que el resultado dependa de la suerte y sea difícil de reproducir. Así que necesitas una forma de enseñarle a la IA un flujo de trabajo que siga el modelo mental propio de Jujutsu.

—Tiene sentido.

—Para eso, con Claude Code creo que lo mejor es basar el flujo principalmente en las **Rules**[54]. En `CLAUDE.md` escribes únicamente que el proyecto usa Jujutsu para el control de versiones. Los pasos concretos del flujo de trabajo van en el archivo de reglas.

—¿Y por qué no escribirlo todo en `CLAUDE.md`?

—La clave está aquí: en Claude Code, `CLAUDE.md` y un archivo de reglas

[54] "Manage Claude's memory — Claude Code Docs"
https://code.claude.com/docs/en/memory

se cargan en el mismo momento[55] y, escribas la instrucción donde la escribas, tiene la misma prioridad. La única razón para separar las instrucciones en un archivo de Rules es la facilidad de mantenimiento. `CLAUDE.md` también lleva información e instrucciones propias del proyecto, así que amontonar encima un buen bloque de convenciones de control de versiones no hace más que invitar a la confusión. Separarlas hace que sean más fáciles de actualizar y de revisar, y te permite reutilizar las mismas reglas en otros repositorios. En cuanto a prohibir comandos concretos, de eso se encarga la configuración de **Permissions**. Lo veremos en detalle más adelante.

—Ajá, o sea que las Rules sirven para que todo sea más fácil de gestionar. ¿Y qué pasa entonces con las **Skills**? Si buscas «jujutsu» en skills.sh[56], aparecen un montón de skills. Solo por la cantidad, parece que se usan bastante.

—Dudo que funcionen tan bien. Las skills son referencias que un agente de IA consulta cuando las considera relevantes; no son restricciones de comportamiento siempre activas. Muchas veces hay que invocarlas explícitamente. Si le pides expresamente una operación sobre el repositorio, como «haz push de esto», es distinto. Pero cuando solo le encargas implementar una funcionalidad o depurar un problema, existe un riesgo real de que ejecute `git` sin llegar a activar la skill.

—Ya veo.

—Entonces, ¿qué pones en realidad en ese archivo de reglas? A grandes rasgos, lo siguiente:

1. Restricciones sobre los distintos comandos
2. Facilitarle a la IA la lectura de los diffs, por ejemplo pasando `--git` a los comandos que los muestran

---

[55] Este es el comportamiento cuando no hay frontmatter `paths`. Si defines un campo `paths` en el frontmatter de un archivo de reglas, ese archivo se carga y se aplica únicamente cuando el agente trabaja con archivos que coinciden con esos patrones.

[56] Un directorio abierto y sitio de ranking publicado por Vercel en enero de 2026 donde puedes buscar y gestionar skills para agentes de IA.
https://skills.sh/

3. Pasar `--ignore-working-copy` a los comandos de solo lectura cuando convenga, lo que evita que las revisiones se fragmenten y previene los errores de stale working copy
4. Cómo tratar los changes al empezar una tarea
5. Procedimientos para dividir un change y para resolver conflictos
6. Cómo manejar bien los bookmarks al hacer push
7. Qué hacer cuando se produce un error de stale working copy

—Esta lista sale en parte de los problemas que me fui encontrando de verdad al hacer que los agentes de IA usaran Jujutsu —continuó Yukina —. El resto salió de repasar casos hipotéticos con el agente uno por uno y dejar que fuera él quien me dijera qué necesitaba. Llevo unos meses trabajando con esta configuración y debería funcionar sin problemas en los flujos de desarrollo habituales que usan GitHub.

—La opción `--git` del punto 2 está ahí porque un diff con el formato de Git le resulta más fácil de interpretar a la IA, ya que es el formato con el que la entrenaron, ¿verdad? Las skills de Jujutsu que circulan por ahí no parecen prestar demasiada atención a ese aspecto. ¿Y en qué consiste el punto 3, `--ignore-working-copy`?

—Normalmente, cuando ejecutas `jj` mientras hay un diff entre el working copy y el historial, Jujutsu toma un snapshot. Esa opción desactiva ese comportamiento. Una de las razones es evitar que se tome un snapshot cada vez que cambia un archivo, lo que dividiría el evolution log en fragmentos mucho más pequeños de lo necesario.

—¿Y la otra razón?

—Mientras esperas a que la IA termine una tarea, a menudo te pasas a otro panel y haces otro trabajo, ¿no? Si tu propio comando `jj` y el comando `jj` de la IA se solapan, uno de los dos puede tomar un snapshot y avanzar el historial antes de que el otro haya terminado. Entonces el working-copy commit que uno de los procesos cree estar mirando acaba siendo más antiguo que el working-copy commit que el historial refleja en realidad, y el desajuste provoca un error. Paso esa opción para atajar en lo posible ese tipo de situaciones. Y el punto 7 cubre qué hacer cuando ocurre de todos modos.

—Hmm, también hay que pensar en ese nivel de detalle.

—Déjame comentar también cómo hacer esto con Codex CLI. Codex no tiene una función de reglas equivalente a la de Claude Code. Existe una función con el mismo nombre, *rules*, pero sirve para gestionar qué comandos pueden ejecutarse fuera del sandbox, no para darle al agente de IA una guía de comportamiento en lenguaje natural.

—Vaya problema. ¿Y entonces qué haces?

—La mejor alternativa es combinar `AGENTS.md` con *skills*. Pero, como dije, las skills solo se disparan cuando la IA decide que son relevantes, así que en este caso acabas metiendo mucho contenido en `AGENTS.md`.

—Hmm, dar con el equilibrio adecuado es delicado. No esperaba que hubiera que cambiar tanto todo el enfoque según el agente.

—Juntando todo esto, los archivos de configuración por proyecto de tus agentes de IA acaban distribuidos así[57]:

Listado 1: Distribución de los archivos de configuración de los agentes de IA

```
my-project/
  CLAUDE.md             ← indica que se use Jujutsu
  AGENTS.md             ← comportamiento básico y cuándo se disparan
las skills
  .claude/
    rules/
      jujutsu-rules.md  ← el archivo de reglas en cuestión
  .agents/
    skills/
      jujutsu/
        SKILL.md        ← el mismo contenido empaquetado como skill
para Codex
```

Listado 2: Comienzo de jujutsu-rules.md

```
# Jujutsu (jj) Rules for AI Agents

Standard jj semantics are assumed knowledge. What follows is only what
an agent cannot infer from the tool itself: this repository's
permissions, its aliases, and the conventions the team has settled on.

---

## Important Notes for AI Agents
```

[57] https://github.com/klemiwary/Juju-chu-es/tree/main/samples/ch3

```
### Command Constraints

`.claude/settings.json` denies these outright. Never plan a workflow
around them; if one is genuinely needed, describe the command and let
the user run it.

| Denied        | Why
|
| ------------- | ----------------------------------------------------
|
| `git` (all)   | Risks corrupting jj state. `jj git ...` and `gh`
remain allowed.                    |
| `jj resolve`  | Opens the interactive merge editor. Resolve conflicts
by editing the files instead. |
| `jj diffedit` | Opens the interactive diff editor.
|
| `jj arrange`  | Interactive TUI.
|

`jj split` is allowed, but **always pass filesets and `-m`. Never run
it bare, and never pass `-i` or `--tool`** because these actions
trigger an interactive diff editor.

These prompt for confirmation every time, so save them for the end of a
task rather than firing them mid-flow: `jj git push`, `jj bookmark
delete`, `jj bookmark forget`, `jj bookmark track`, `jj bookmark
untrack`.

### What jj Changes About the Workflow

- **No staging, no stash, no "unsaved change".** Saving a file puts it
in the working-copy commit (`@`) that instant. Never ask the user
whether a change should be included — it already is.
⋮
```

—Como tus repositorios existentes probablemente estén gestionados con Git, te conviene limitar esta configuración a los repositorios concretos que usan Jujutsu, en lugar de ponerla en tu configuración a nivel de usuario.

—¡Genial! ¡Me lo llevo tal cual!

Sobre las autoras

Yuka Ooka

Yuka Ooka empezó su carrera como desarrolladora de PHP y Ruby y como product manager en varias empresas de tecnología japonesas, antes de establecerse como freelance en 2016. Se interesó pronto por React, que por entonces era todavía una tecnología de nicho en Japón, y trabajó como especialista en React para diversos clientes. A partir de esa experiencia, escribió la serie *Riakuto!* (りあクト !), que caló entre los lectores en Japón y vendió más de 40.000 ejemplares en total.
Está en X como @oukayuka y escribe sobre cómo dominar Jujutsu en su blog, blog.juju-chu.com/es/blog.

El espacio de trabajo de la autora

Ilustración: Megumi Kuroki

Mangaka e ilustradora.
Se ha encargado de todas las ilustraciones de portada de las series *Riakuto!* y *Juju-chu!*.