JuJu-Chu!

Yuka Ooka

Comece seu fluxo de trabalho Jujutsu×IA com `jj new`

O controle de versões também evolui na era da IA!

EXPANSÃO DE DOMÍNIO

Características do Jujutsu

- Commits automáticos
- Undo universal
- Conflitos de primeira classe

Juju-chu!

Comece seu fluxo de trabalho Jujutsu × IA com `jj new`

Yuka Ooka

Klemiwary Books

Página de créditos

Juju-chu!
Comece seu fluxo de trabalho Jujutsu × IA com `jj new`

Primeira edição em português: setembro de 2026
Publicado originalmente em japonês: abril de 2026

Publicado pela Klemiwary Books
Repositório de apoio: https://github.com/klemiwary/Juju-chu-pt

Prefácio

Neste momento, o Jujutsu está atraindo muita atenção.

Quando o assunto é sistema de controle de versão (VCS), o Git reina soberano há muito tempo. Isso vale especialmente para os engenheiros de software que entraram no mercado a partir dos anos 2010: muitos deles nunca usaram nada além do Git. O GitHub, por sua vez, foi o maior responsável por levar o Git à posição de padrão de fato, e hoje conta com bem mais de 180 milhões de desenvolvedores no mundo todo. É algo tão onipresente que quase daria para dizer que um engenheiro de software sem uma conta no GitHub não é lá um engenheiro de software de verdade.

Nesse cenário, é extremamente raro que um VCS diferente do Git ganhe tração — tão raro que, quando acontece, parece quase um acontecimento.

Por que Jujutsu, por que agora?

O desenvolvimento do Jujutsu começou em 2019, mas sua primeira versão pública (v0.3.0) só chegou em 2022. E foi apenas a partir de 2025 que ele passou a aparecer com regularidade nos círculos de desenvolvedores. Esse momento coincide com a rápida disseminação dos agentes de codificação de IA e com a época em que o termo "vibe coding" começou a pegar como palavra da moda.

Os agentes de IA agora conseguem gerar e modificar quantidades enormes de código a uma velocidade que nenhum humano consegue igualar, e o custo de escrever código — e de iterar sobre ele — caiu drasticamente. E suspeito que um número cada vez maior de pessoas passou a sentir um descompasso entre essa nova realidade e os sistemas de controle de versão tradicionais, projetados a partir de uma premissa muito diferente: a de que "os humanos escrevem código com cuidado, linha por linha, decidindo conscientemente o que registrar e compartilhando apenas quando tudo está pronto".

E então, quase da noite para o dia, o Jujutsu simplesmente apareceu. As pessoas experimentaram, e ele se encaixou surpreendentemente bem no ritmo do desenvolvimento baseado em agentes de IA. “Preciso contar isso para todo mundo!” É assim, a meu ver, que surgiu a recente onda de entusiasmo.

“Mas isso você também faz no Git”

Quase toda vez que um artigo introdutório viraliza e o Jujutsu vira assunto, aparece sem falta o mesmo refrão: “Isso o Git já faz”. Mas o valor do Jujutsu não está em fazer o que o Git não consegue.

Pense, por exemplo, no BlackBerry e no iPhone. Os dois faziam ligações, navegavam na web, instalavam e rodavam aplicativos e tiravam fotos. E quando o iPhone surgiu, muita gente, sobretudo os mais fiéis ao BlackBerry, o rejeitou apontando exatamente esse argumento. Mas todos sabemos como essa rivalidade terminou no fim das contas. Mesmo que os dois pudessem fazer as mesmas coisas, havia uma diferença decisiva na experiência de uso que cada um oferecia.

“Conseguir fazer algo” e “conseguir fazer com conforto” não são a mesma coisa. Mesmo quando você faz no Jujutsu exatamente o que faria no Git, o Jujutsu automatiza uma parte maior do trabalho, as operações são mais fáceis de memorizar e os erros são fáceis de reverter. Assim, toda a experiência fica mais tranquila, mais segura e menos trabalhosa.

As ideias-chave são **baixa carga cognitiva** e **segurança psicológica**. No Jujutsu, absolutamente tudo foi pensado com um cuidado especial voltado a esses dois aspectos.

Para quem é este livro

Este livro foi escrito pensando nos seguintes leitores:

- Você conhece as operações básicas do Git e usa Git e GitHub/GitLab no dia a dia.
- Você usa um agente de codificação de IA como o Claude Code ou o Codex CLI, ou pretende começar.
- Você não tem experiência com Jujutsu, ou já começou a usar mas ainda está no nível de iniciante.

Deixando claro: este não é um guia de Git para iniciantes. Eu uso React como material didático para demonstrar os fluxos de trabalho com um VCS, mas, como esse não é o ponto central, não é preciso ter conhecimento de React nem do ecossistema Node.js. Você pode adaptar os exemplos à sua própria stack principal à medida que avança na leitura.

Algumas observações antes de ler

Em agosto de 2026, o Jujutsu está em desenvolvimento ativo e ainda não chegou à versão estável (v1.0.0). Na verdade, o próprio README do projeto o descreve como um "experimental version control system" e afirma explicitamente que breaking changes podem entrar antes da v1.0.0. Por isso, alguns nomes de comandos e comportamentos descritos neste livro podem não corresponder à versão mais recente. Dito isso, o conteúdo deste livro vai muito além de apenas como usar comandos: é um tratamento abrangente que se estende à filosofia de design do Jujutsu, ao seu modelo mental e a como aplicá-lo à programação agêntica. Mesmo que mudanças superficiais aconteçam mais adiante, estou convencida de que a essência deste livro continuará útil por muito tempo.

Como este livro está organizado

O capítulo 1 apresenta uma visão geral do Jujutsu, o capítulo 2 percorre as operações básicas, o capítulo 3 trata do desenvolvimento colaborativo com IA, o capítulo 4 cobre uma variedade de aplicações avançadas e o capítulo 5 é um guia de perguntas frequentes e resolução de problemas. Recomendo ler na ordem, desde o começo. Mas, se você preferir logo pôr a mão na massa, pode ir direto para o capítulo 2, e se tiver um interesse especial em trabalhar com IA, pode pular para o capítulo 3. Os capítulos 4 e 5 também foram pensados para funcionar como referência.

Mais uma coisa: este livro se desenrola como uma história, contada através de um diálogo entre duas personagens — uma engenheira sênior e uma engenheira júnior. Escolhi esse formato de diálogo para manter o guia acessível e para responder, uma a uma, às dúvidas que quem está começando costuma ter.

Agora, me permita te dar as boas-vindas ao mundo do Jujutsu. O encantamento de abertura: `jj new`.

Sobre este livro

Elenco de personagens

Yukina Shibasaki

Engenheira sênior em uma empresa de serviços de internet sediada em Tóquio. Foi contratada graças ao seu domínio de React e, como tech lead, montou do zero o time de desenvolvimento front-end. Tem o costume de descobrir tecnologias novas e insistir em adotá-las antes de estarem totalmente consolidadas. Às vezes isso tira os colegas de time do sério, mas, por algum motivo, essas apostas costumam acabar estourando mais tarde, então o time faz vista grossa.

Kanae Akiya

Engenheira júnior do time da Yukina, muito entusiasmada. Ela se considera a discípula número um da Yukina e tenta seguir os passos dela em tudo. Adora anime e mangá. Embora tenha receio de que a IA possa tirar o emprego dela, usa ativamente agentes de IA no seu trabalho de desenvolvimento e está decidida a não ficar para trás.

Sobre o código de exemplo

O código de exemplo e os arquivos de configuração apresentados neste livro estão disponíveis no repositório do GitHub abaixo. Para a localização de cada arquivo, consulte a nota de rodapé correspondente no texto principal.

- Repositório público de apoio de *Juju-chu! Comece seu fluxo de trabalho Jujutsu × IA com `jj new`*
  https://github.com/klemiwary/Juju-chu-pt

Você pode usar como quiser o código que aparece no texto principal e no repositório acima. Você também pode citá-lo ou usar trechos dele em documentos públicos, vídeos e materiais semelhantes sem pedir

permissão à autora, desde que mencione a fonte. Dito isso, evite republicar o código na íntegra.

Errata

No link abaixo está disponível uma lista de erratas com os erros e as gralhas do texto. Ela será atualizada conforme necessário.

- Erratas de — *Juju-chu! Comece seu fluxo de trabalho Jujutsu × IA com `jj new`*
  https://github.com/klemiwary/Juju-chu-pt/blob/main/errata.md

Versões de software usadas neste livro

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

Sumário

Prólogo

— Ai, não, não, não, não...!! — exclamou Kanae.

Yukina ergueu os olhos.

— Que grito foi esse do nada? O que aconteceu?

— Estava deixando o Codex escrever código para mim e esqueci de commitar as mudanças anteriores antes de mandar outro prompt. Aí ele sobrescreveu meus arquivos existentes com mudanças que eu nunca pedi, e agora não consigo mais trazer de volta...

— Ah, o imposto dos agentes de IA. Se você estivesse no Claude Code, poderia reverter isso com o comando `/rewind`. Mas o Codex CLI não tem nada equivalente.[1]

— Ultimamente o Codex tem sido mais rápido e melhor para destravar problemas difíceis, então eu vinha dando preferência para ele. Eu devia ter ficado no Claude Code mesmo...

— Dito isso, o `/rewind` do Claude Code também não é uma bala de prata. Ele só rastreia o código escrito pelo agente. Se nesse meio-tempo você rodou comandos de shell ou fez edições manuais, essas mudanças não são capturadas. E se você tentar usar o `/rewind` para voltar vários pontos no histórico, ele acaba se embolando com o do Git e é fácil entrar numa bagunça da qual não dá para sair.

— Aff, então nenhuma dessas ferramentas resolve direito a única coisa de que eu realmente preciso: voltar exatamente àquele momento, né... Yukina, você é muito melhor do que eu para trabalhar com IA. Você nunca comete esse tipo de vacilo?

— Eu praticamente só uso o **Jujutsu**, então quase nunca passo por esse tipo de acidente.

— **Jujutsu?!** Espera, Yukina, você é uma *feiticeira de jujutsu*?! E ainda por cima manipula o tempo? Isso é facilmente grau 1 ou mais, né? ...

---

[1] Situação em agosto de 2026. O Codex CLI oferecia anteriormente um comando experimental `/undo`, mas ele foi removido depois devido a bugs frequentes e à preocupação de que seu gerenciamento de histórico entrasse em conflito com os VCS e confundisse os usuários.

Agora que eu parei para pensar, você até se parece um pouco com a Maki Zen'in.[2]

— Você só está dizendo isso por causa dos óculos, né? Não estou falando do tipo de *jujutsu* que aparece em *Jujutsu Kaisen*[3]. Estou me referindo ao Jujutsu, o sistema de controle de versão de nova geração que anda chamando bastante atenção ultimamente.

— Nossa, existe uma ferramenta assim? Eu não sabia. Quando o assunto era controle de versão, eu só conhecia o Git mesmo.

— Provavelmente é assim para a maioria dos desenvolvedores júnior que entram no mercado hoje. Mas, na verdade, já existiram vários VCS antes e depois do Git, e o que mais está crescendo em popularidade agora é o Jujutsu.

— Mas me diz uma coisa, Yukina, há quanto tempo você usa esse tal de Jujutsu? A gente trabalha no mesmo time e eu não fazia ideia. ...Espera, mas no GitHub parece que você usa Git como todo mundo. Como assim?

— Pois é. O Jujutsu é compatível com o Git: ele pode usar um repositório Git diretamente como armazenamento de backend. Então, a menos que eu mesma toque no assunto, meus colegas de time não têm como perceber que eu uso Jujutsu. Aliás, acho que comecei a usar faz uns seis meses.

— Espera, faz tanto tempo assim?! Você ficou guardando uma coisa tão útil só para você esse tempo todo? Logo eu, sua parceira de time...

— Desculpa, desculpa. É uma ferramenta que não obriga o resto do time a mudar o fluxo de trabalho, e eu não achei que você fosse se interessar tanto.

— Mas, se eu estivesse usando Jujutsu, eu ainda conseguiria recuperar tudo, mesmo que acontecesse algo assim, né? Estou mais do que inter-

---

[2] Uma personagem do mangá *Jujutsu Kaisen*. A energia amaldiçoada dela não é maior que a de uma pessoa comum, mas ela tem capacidades físicas sobre-humanas e luta usando diversas ferramentas amaldiçoadas. Como não consegue ver os espíritos amaldiçoados a olho nu, normalmente usa óculos especiais.

[3] Um mangá de Gege Akutami, serializado na *Weekly Shōnen Jump*. Uma fantasia sombria que retrata as batalhas dos feiticeiros que exorcizam os monstros e espíritos amaldiçoados nascidos das emoções negativas da humanidade. A série já passa de 150 milhões de exemplares em circulação no mundo todo. Em português o título também se mantém como *Jujutsu Kaisen*, o que, somado à arte marcial japonesa do *jujutsu* — da mesma família do jiu-jitsu brasileiro —, não torna exatamente mais fácil pesquisar sobre o VCS.

essada! Por favor, me ensina a usar Jujutsu! Eu também quero virar uma feiticeira de Jujutsu grau 1 e manipular o tempo.

— Bom... tudo bem. Deixa eu arranjar um tempo agora mesmo para te explicar o Jujutsu passo a passo.

— Espera, sério? Isso não vai atrasar o projeto?

— Se adotar o Jujutsu reduzir drasticamente o tempo que você perde com recuperação, no médio e longo prazo isso compensa de sobra. Como líder do time, autorizo em caráter excepcional.

— Isso! Uma aula particular de Jujutsu com a Yukina! Mal posso esperar!

Capítulo 1. Que tipo de ferramenta é o Jujutsu?

1-1. O Git combina mal com a programação agêntica?

— Esse acidente que você acabou de ter, Kanae, não aconteceu porque você estava sendo especialmente descuidada. Aconteceu porque o Git, o VCS que você usa, não foi projetado para se encaixar no desenvolvimento moderno baseado em agentes de IA. Era um acidente anunciado. Para ser justa, o Git foi projetado há mais de vinte anos, então é pedir demais que ele tivesse antecipado um mundo em que os agentes de IA despejam quantidades enormes de código em altíssima velocidade.

— ...Ah. Então era isso. A culpa não é minha!

— Antes de entrar no Jujutsu em si, vamos primeiro deixar claro exatamente onde o Git fica devendo para a programação agêntica. Assim, enquanto você aprende Jujutsu, vai sentir na prática de onde vêm a conveniência e a facilidade de uso dele.

O redundante modelo de três estados do Git

— A maior característica do Git, e ao mesmo tempo a primeira coisa em que quem está começando tropeça — disse Yukina —, é que as mudanças nos seus arquivos passam por três estados distintos antes de serem registradas no histórico:

- **Working tree** — o lugar onde você de fato edita os arquivos. Também chamado de working directory.
- **Staging area** — a zona entre o working tree e o repositório onde você prepara um commit. Também chamado de index.
- **Repositório** — o banco de dados que armazena todo o histórico de mudanças e todos os arquivos.

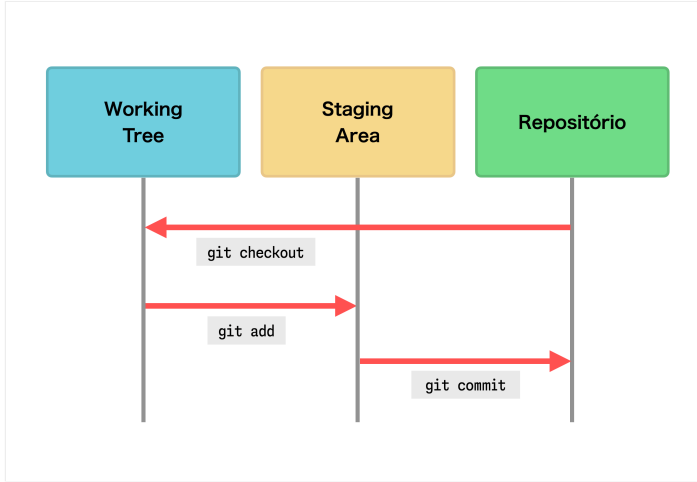

Figura 1: O modelo de três estados do Git

— É verdade. Quando eu comecei, sempre me perguntava por que eu precisava me dar ao trabalho de fazer `git add` e depois `git commit`. Por que só o `commit` não basta?

— Na época em que os humanos faziam todo o desenvolvimento à mão, esse modelo tinha uma lógica de verdade. A ideia é que você escolhe só algumas das mudanças do seu working tree, passa para o staging e então transforma cada conjunto de mudanças com sentido em um commit. A staging area é pensada como uma área de rascunho para o seu próximo commit. Tê-la ajudava a trabalhar com cuidado e facilitava produzir commits de boa qualidade. Lembre que o Git foi criado originalmente para o desenvolvimento do kernel do Linux, então mandar commits confusos, cuja intenção não dá para entender, deixaria a vida dos mantenedores um inferno.

— Hmm, entendi.

— Mas no mundo de hoje, onde um agente de IA gera dezenas ou centenas de linhas de código espalhadas por vários arquivos de uma só vez, esse passo extra virou um empecilho sério. Não é nada realista um humano ficar executando `git add` e `git commit` várias e várias vezes para acompanhar uma IA incansável que não para de produzir código novo. O resultado final é que as pessoas dão a próxima instrução para a IA sem nunca ter registrado o estado anterior e depois ou não con-

seguem voltar atrás no trabalho para refazê-lo, ou acabam se afogando num mar de histórico. É exatamente esse tipo de acidente que não para de acontecer.

O alto custo da troca de contexto

— No desenvolvimento real — continuou Yukina —, muitas vezes você de repente tem vontade de testar uma ideia, ou é atingida do nada por um pedido de revisão ou uma correção de bug urgente. Isso te obriga a trocar para um branch diferente com o working tree ainda pela metade. Quando isso acontece no Git, você recorre ao `git stash` para deixar as coisas de lado temporariamente.

— Isso, isso, eu faço stash quase todo dia. É um saco, e se, quando você volta, já esqueceu que deixou algo no stash, a coisa fica feia rapidinho.

— E quando uma interrupção longa é interrompida de novo por algo ainda mais urgente, e você acaba com vários stashes empilhados, fica impossível lembrar qual stash guarda qual trabalho. Não é nada incomum bater num conflito ao restaurar um deles e acabar jogando fora as suas mudanças de pura frustração.

— Ugh, só de imaginar já me dá dor de estômago...

— Depois tem o custo de dar nomes no Git, que não é nada desprezível. Para começar um trabalho novo, primeiro você tem que criar um branch e dar a ele um nome com sentido. Para fazer um commit você precisa de uma mensagem de commit e, uma vez que você se decide por uma, mudá-la depois não é impossível, mas exige alguns passos tediosos. Mesmo para algo que você só está experimentando e pode jogar fora no minuto seguinte, você tem que inventar algum nome antes de poder começar. Isso adiciona uma quantidade surpreendente de sobrecarga cognitiva.

— O custo de dar nomes, né... Eu não tinha parado para pensar nisso, mas agora que você fala, eu realmente empaco toda vez tentando decidir como chamar as coisas, e simplesmente travo.

Reescrever o histórico é complexo e perigoso

— Comparado com a época em que os humanos escreviam o código com cuidado, à mão — prosseguiu Yukina —, o custo de fazer uma IA escrever código é muitíssimo menor, então agora é muito mais comum

querer voltar atrás para ajustar algo que você já tinha registrado. Mas reescrever o histórico no Git é uma tarefa intimidante.

— Sei bem o que você quer dizer. Quando eu realmente preciso editar o histórico, acabo pesquisando no Google ou perguntando para uma IA de chat, e faço tudo prendendo a respiração o tempo todo.

— No Git, o conceito único de "reescrever o histórico" está espalhado por vários comandos: `git commit --amend`, `git rebase -i`, `git reset`. E o alcance de cada um pode mudar drasticamente dependendo de quais opções você passa. Construir um modelo mental claro é difícil. Duvido que muita gente tenha todos eles memorizados e saiba escolher com confiança o adequado para cada situação.

— Nem me fala.

— O `git rebase -i` em particular tem aquela interface peculiar em que você edita uma lista de commits em um editor de texto, e qualquer erro que você cometa ali vai direto para o seu histórico. Apague uma linha e aquele commit desaparece; troque `pick` por `drop` e ele também desaparece. No instante em que você salva e fecha o editor, o processo começa a rodar, então o desespero ao perceber que editou algo errado é de outro nível.

— E se um conflito aparece no meio do caminho — seguiu Yukina —, você pode acabar tendo que resolver um conflito atrás do outro conforme os commits são aplicados um a um. Lá pelo fim você já perdeu a conta de em qual commit está, e a coisa costuma se resumir a escolher entre `git rebase --abort` para começar tudo de novo, ou martelar `git rebase --continue` sem entender nada.

— É, exatamente. Eu tenho tanto medo disso que acabo indo o mais longe possível sem commitar, só para não ter que dividir ou consertar as coisas depois. O que é, claro, exatamente como acontece o acidente de agora há pouco, ou como eu acabo mandando um PR com um único commit gigante, um verdadeiro balaio de gato.

Comandos com efeitos colaterais pesados e difíceis de reverter

— O Git tem vários comandos destrutivos — disse Yukina — que você não consegue recuperar facilmente depois de executá-los. Por exemplo, `git reset --hard`, `git clean -fd` e `git restore` varrem sem piedade as mudanças que você ainda não commitou. O fato de um único comando

digitado errado poder apagar de uma vez todo o trabalho que você foi acumulando com dedicação no working tree é simplesmente aterrorizante.

— Uma vez, o Gemini CLI teve a gentileza de fazer mudanças que eu nunca pedi e, quando eu disse "põe as coisas de volta como estavam", ele rodou `git restore` e apagou também tudo o que eu vinha fazendo antes disso. Depois pediu desculpas mil vezes, mas o estrago já estava feito.

— Com o Claude Code você pode usar as configurações de Permissions para bloquear comandos específicos do Git, mas é fácil configurar errado. O risco de soltar uma IA sobre o Git, uma ferramenta em que um único comando pode destruir de forma irreversível o trabalho que você foi acumulando, é alto.

— Além disso — acrescentou Yukina —, algum usuário avançado de Git pode dizer: "Usa o `git reflog`, com ele dá para desfazer qualquer coisa", mas o reflog não consegue restaurar tudo, na verdade. Ler o reflog é genuinamente difícil, e restaurar com segurança até o estado pretendido exige um conhecimento profundo das entranhas do Git. O número de engenheiros capazes de operar o reflog com calma no meio do pânico e voltar exatamente ao estado que queriam é, de novo, ínfimo.

Resumo: a filosofia de design do Git frente à era da IA

— Hmm — disse Kanae. — Ouvindo tudo isso, o Git começa a parecer uma dessas ferramentas para as quais a humanidade ainda não estava preparada. Por que ele é projetado assim?

— É porque o Git foi originalmente projetado em torno do modelo de desenvolvimento do kernel do Linux: **os humanos escrevem código à mão, organizam suas mudanças, agrupam-nas em unidades prontas para revisão e as compartilham com cuidado**. Para um fluxo de trabalho em que os humanos passam mudanças a outros humanos com cuidado, é uma ferramenta fenomenal, e provou seu valor de sobra ao longo dos anos. Mas no desenvolvimento baseado em agentes de IA o cenário é bem diferente:

- O ciclo rápido de tentativa e erro é constante.
- Enormes quantidades de código mudam de uma vez.

- O contexto do seu trabalho muda toda hora.
- A necessidade de organizar o histórico depois surge com mais frequência.

— Sob condições como essas, os próprios traços que faziam o Git funcionar tão bem acabam atrapalhando o seu desenvolvimento. Essa é a razão de fundo pela qual os pontos problemáticos do Git ficaram tão em evidência na era da IA.

— Aham.

— O que a programação agêntica realmente precisa é de uma ferramenta em que você primeiro possa fazer uma bagunça e depois arrumá-la com segurança. E é justamente nesse ponto que um sistema de controle de versão construído sobre uma filosofia de design diferente começou a chamar atenção.

— Ah, é o Jujutsu, né!

1-2. As características do Jujutsu que se destacam na era da IA

— Não é que eu não confie em você, Yukina, mas a popularidade do Jujutsu está mesmo subindo tão rápido assim agora? Eu nunca nem tinha ouvido falar de um VCS desse tipo até você mencionar.

— Bom, em meados de 2026 ele ainda está no estágio em que um grupo pequeno de desenvolvedores atentos às ferramentas novas o apoia com paixão. Em termos da teoria do abismo (chasm)[4], está se espalhando entre os early adopters, mas ainda não atravessou o abismo.

— Eu adoraria te mostrar dados concretos — continuou Yukina —, mas realmente não existe uma boa métrica objetiva para a popularidade de um VCS. O Git foi tão dominante por tanto tempo que as pesquisas com desenvolvedores nem se dão ao trabalho de perguntar "que VCS você usa?". Não é bem uma comparação justa, mas vamos dar uma olhada no histórico de estrelas do GitHub dos repositórios oficiais.

[4] Uma teoria de marketing baseada no ciclo de adoção de tecnologia, um modelo sociológico de como novos produtos e inovações se difundem. Ela divide o processo de difusão em cinco segmentos de clientes e identifica o "abismo" (chasm), uma fenda profunda entre o mercado inicial e o mercado mainstream, como o obstáculo mais crítico a ser superado.

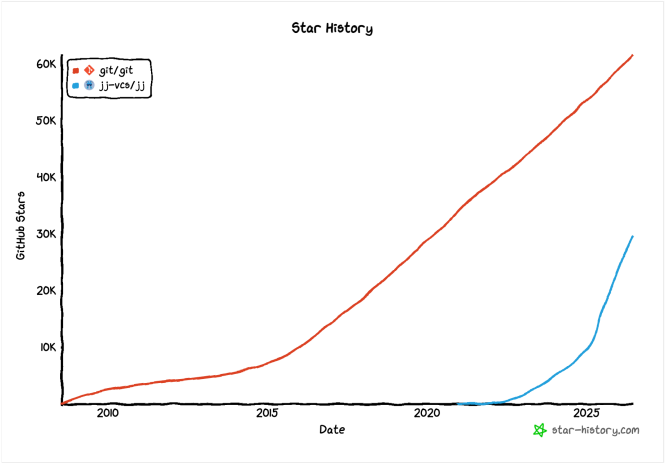

Figura 2: Histórico de estrelas do GitHub de Git e Jujutsu (GitHub Star History, junho de 2026)

— Uau! A curva do Jujutsu está subindo feito um taco de hóquei. Ela decolou de verdade a partir de meados de 2025. Nesse ritmo, parece que o dia em que ele vai ultrapassar o Git não está tão longe. Mas o que você quer dizer com "não é bem justa"?

— O Git não é desenvolvido de fato no GitHub. Lá existe apenas um repositório espelho. O oficial fica hospedado no kernel.org, o mesmo lugar do kernel do Linux[5]. O Jujutsu, por outro lado, tem seu repositório oficial no GitHub.

— Bom, para a maioria dos usuários o GitHub é praticamente tudo, então eu diria que é um barômetro confiável de popularidade. Ou seja, o Jujutsu está mesmo ganhando força rapidamente. O que está impulsionando essa alta?

— Kanae, você acabou de dizer que ele começou a decolar em meados de 2025. Você lembra o que aconteceu por volta dessa época?

— ...Hmm, o que foi mesmo?

— O Claude Code foi lançado oficialmente em maio. Até então, as ferramentas de IA para programar eram principalmente baseadas em IDE, voltadas para mudanças simples de código por meio de sugestões e chat. O Claude Code é uma ferramenta agêntica de codificação com

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

IA que roda no terminal: você dá as instruções e ele gera código de alta qualidade de forma autônoma, de uma vez só.

— Com o Claude Code em cena — prosseguiu Yukina —, o mundo mergulhou de vez num cenário em que a IA escrevia muito mais código do que os humanos. Para não ficar para trás, o Codex CLI e o Gemini CLI seguiram o exemplo, o Cursor migrou para um modelo baseado em agentes de IA, e a tendência rumo ao desenvolvimento de software com agentes de IA se tornou decisiva.

— É verdade. Agora parece história antiga, mas não faz tanto tempo assim que a gente escrevia cada linha à mão.

— A ascensão do Jujutsu parece estar sendo puxada pela disseminação dos agentes de codificação de IA como o Claude Code e o Codex. Eu mesma decidi experimentar o Jujutsu depois de ler, por essa época, uma série de artigos que apresentavam o Jujutsu no contexto da programação agêntica. Alguns dos mais lidos são:

- 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]

— Nossa. Quando os especialistas te dizem que Jujutsu e IA combinam tão bem, fica impossível não querer testar. Mas tem uma coisa que eu não entendo. O Jujutsu já existia antes de a programação agêntica decolar, né? Ele não foi projetado para isso, então por que se encaixa tão bem?

— O desenvolvimento do Jujutsu começou em 2019 e, naquele momento, quase ninguém conseguia prever um futuro em que a IA programaria de forma autônoma, então é claro que ele não foi projetado pensando nisso. Segundo o autor, Martin von Zweigbergk, os objetivos dele eram **reduzir a carga cognitiva dos humanos** e **acabar com o**

[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

complexo modelo mental do Git[10].

— Os agentes de IA iteram em alta velocidade sem medo de errar — emendou Yukina —, e trabalhar lado a lado com eles exige muitíssimo do cérebro humano. Então o design do Jujutsu, que se propôs a reduzir essa carga cognitiva dos humanos, acabou se encaixando de forma natural nessa situação, como um efeito colateral.

— Hmm, faz sentido.

— Então vamos ver, uma a uma, as características do Jujutsu que estão chamando atenção na era da IA.

O working copy é commitado automaticamente

— Diferentemente do Git, o Jujutsu não tem staging area — disse Yukina. — Há apenas dois estados: o **working copy** (que corresponde ao working tree do Git) e o **repositório**. E aqui está a maior característica do Jujutsu: as mudanças nos arquivos do working copy são commitadas automaticamente. Sem que o usuário execute nada parecido com `git commit`, é tirado um snapshot do working copy e ele é registrado no repositório.

— O quê? Você não precisa fazer add e nem precisa commitar? Como é que isso funciona na prática? Tem um daemon[11] vigiando o estado dos arquivos, ou algo assim?

— Não. O sistema de comandos do Jujutsu usa `jj` como comando principal, com subcomandos como `jj log` e `jj diff`, e um snapshot é tirado no momento em que um comando `jj` roda. Se houver diferença em relação ao snapshot anterior, essa diferença é commitada.

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

[11] Um programa que roda continuamente em segundo plano em um sistema UNIX ou Linux (ou SO semelhante), tratando automaticamente de tarefas específicas, como atender requisições web ou gerenciar a rede.

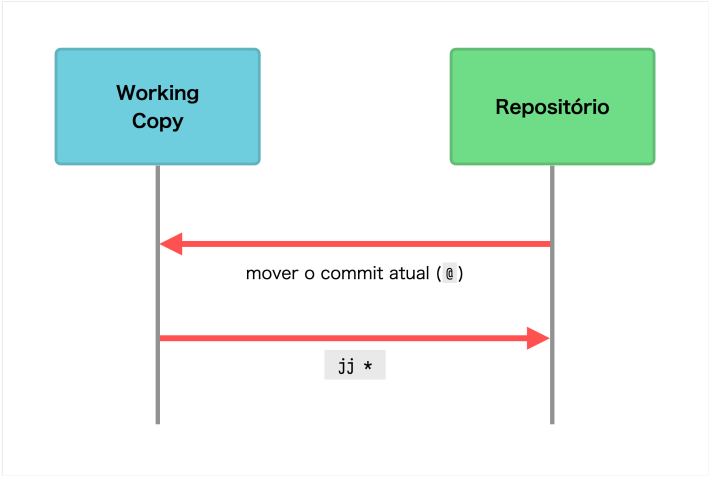

Figura 3: O modelo de dois estados do Jujutsu

— Nossa. É bom que os erros de esquecer de dar add ou de commitar simplesmente desapareçam. Mas, se você não controla quando os commits acontecem, o seu histórico não acaba virando um caos?

— Vou explicar com mais detalhe depois, quando você for testar de verdade, mas o Jujutsu tem um conceito chamado **change** que encapsula os commits como uma única unidade. Para colocar de forma bem simplificada por enquanto: um change é como um contêiner que guarda vários commits em ordem cronológica. Normalmente o histórico do Jujutsu é exibido por change, e ver os commits individuais dentro de um deles exige um passo a mais. Nomear, ajustar a granularidade e reordenar é tudo feito no nível do change. Então você não precisa se preocupar com os commits se acumulando por conta própria e sujando o seu histórico.

— Ah, entendi como eles resolvem isso.

— A documentação oficial descreve essa propriedade com o termo **working-copy-as-a-commit**. Graças ao seu modelo simples de dois estados e a essa propriedade, o Jujutsu não precisa de nada que corresponda a `git add`, `git commit` ou `git stash`. Comparada com os VCS tradicionais, a carga cognitiva em torno de gerenciar o estado dos arquivos é muitíssimo menor.

Toda operação pode ser desfeita

— Quando você usa Git no dia a dia — disse Yukina —, nunca tem aqueles momentos de "puxa, eu queria muito desfazer esse último comando"?

— O tempo todo. Sinceramente, me irrita que não exista um desfazer como num editor.

— O Jujutsu tem exatamente isso: o **universal undo**. Qualquer coisa que envolva um remoto não dá para rebobinar com perfeição, claro, mas toda operação local pode ser desfeita e refeita.

— Isso é incrível! Só isso já me dá vontade de mudar.

— Além do log de commits habitual, o Jujutsu tem um **operation log**. É um registro de cada comando `jj` que você executou, ligado aos snapshots do working copy. Então, além do simples desfazer, você pode rebobinar para o estado do momento em que qualquer comando específico foi executado, ou riscar uma única operação do histórico como se ela nunca tivesse acontecido.

— No Git eu sempre executo o rebase e outros comandos de edição de histórico com o coração na mão, mas com o desfazer eu posso adotar uma abordagem de "testa, e se não der certo, desfaz e tenta de novo". Isso dá uma tranquilidade.

— Exato. Uma vez que você se liberta do medo de fazer algo irreversível, a barreira para experimentar cai drasticamente. No Git, fazer um agente de IA executar comandos de edição de histórico dá medo demais para tentar a sério, mas com o Jujutsu isso não é problema nenhum. A sensação de segurança psicológica está num nível completamente diferente. Depois que você experimenta, não tem como voltar atrás.

Os conflitos são tratados como só mais um estado

— O que dá medo no Git — disse Yukina — são os conflitos durante um merge, um rebase ou um cherry-pick. Quando um conflito acontece no Git, as outras operações ficam rigorosamente bloqueadas, e você é obrigada a parar de trabalhar até resolvê-lo.

— Dá medo mesmo. Sempre que eu puxo o main mais recente antes de abrir um PR, fico rezando para não aparecer nenhum conflito.

— O Jujutsu, por outro lado, trata **um conflito como só mais um estado**. Quando um acontece, ele é commitado como está e não bloqueia nenhuma operação. Por exemplo, se um conflito surge durante um rebase, o rebase termina mesmo assim, carregando o conflito junto. A documentação oficial chama isso de **first-class conflicts**.

— Espera, isso é bom? Um conflito não é uma coisa que você tem que resolver na hora? Deixar ele parado ali fica me incomodando.

— Essa ideia de que "um conflito tem que ser resolvido na hora" é uma suposição de quem tem a cabeça no Git. A razão de o Git tratar um conflito não resolvido como uma "exceção" que bloqueia não é uma necessidade técnica; é uma restrição de implementação. Deixa eu te dar alguns casos concretos em que os first-class conflicts são ótimos:

- Você está fazendo o rebase de um branch, e resolver um conflito exige confirmar a intenção da implementação com a Alice, a colega responsável por esse código. Mas ela está de férias e só volta amanhã. No Git, você teria que deixar o rebase de lado até lá. No Jujutsu, você pode terminar o rebase por enquanto e cuidar da resolução depois, ou seguir em frente com um remendo rápido e tosco.
- Você quer deixar uma IA cuidar de resolver vários conflitos que surgiram durante um merge. No Git você não consegue fazer commit a menos que todos os conflitos estejam resolvidos, então, se ela erra em algum, restaurar só um conflito específico e verificar de novo é incômodo. No Jujutsu você consegue localizar e restaurar um único conflito e refazer a resolução quantas vezes for preciso.

— Ah, entendi. Isso realmente parece útil. Deixar uma IA resolver conflitos seria impensável no Git. Mas, com o Jujutsu, você pode fazer ela resolver um de cada vez e, se errar em algum, é só desfazer e tentar de novo. Isso torna muito mais fácil delegar o trabalho sem hesitar.

— Além disso, no Jujutsu, quando você edita um commit anterior, as mudanças naquele arquivo são propagadas automaticamente para os commits descendentes. Então, mesmo ao resolver um conflito, basta você ir até o commit onde ele aconteceu e editar o arquivo ali, sem precisar disparar um rebase na mão. Como adiar a resolução quase não custa nada depois, a carga psicológica é pequena.

Comandos ortogonais e com pouca dependência de estado

— Uma coisa que eu noto usando Jujutsu — disse Yukina — é que os comandos são muito fáceis de lembrar. Para os comandos cujo alvo é o próprio histórico, o comando corresponde um a um ao verbo do que você quer fazer. Por exemplo, descartar histórico é `jj abandon`, dividir é `jj split`, combinar é `jj squash`. Não existe nada como `git checkout` ou `git reset`, onde as opções ou os argumentos mudam radicalmente o que o comando faz. A operação que você quer corresponde diretamente a um comando, então você nunca precisa se perguntar "qual era mesmo aquele comando?".

— Aham, ter menos sobrecarga de comandos é ótimo.

— Os comandos cujo alvo não é o próprio histórico também são organizados em uma hierarquia lógica clara. Por exemplo, no Git as operações de branch estão espalhadas: criar um é `git branch <branch-name>` ou `git checkout -b <branch-name>` ou `git switch -c <branch-name>`, listar é `git branch`, apagar é `git branch -d <branch-name>`. Difícil chamar isso de consistente. No Jujutsu é `jj bookmark create`, `jj bookmark list`, `jj bookmark delete`. "Sobre o que você atua e o que você faz" se lê direto na estrutura do comando.

— Você tem razão, eu nunca conseguia lembrar qual comando de branch do Git fazia o quê. Ficava procurando toda vez. Mas se os comandos do Jujutsu funcionam assim, até eu talvez consiga memorizar.

— Além disso, o Jujutsu compartilha as convenções básicas de opções e de alvo entre muitos comandos. A opção para indicar um ponto específico no grafo do histórico é `-r/--revision`, e `-f/--from`, `-t/--into` e `-o/--onto` têm o mesmo significado em todos os comandos de edição de histórico. Então você transfere com facilidade o que aprendeu de um comando para outro.

— Entendi. Com o Git dá meio que a sensação de que você tem que decorar um feitiço diferente para cada comando.

— No design de software, quando aprender sobre um elemento permite extrapolar para outros elementos, dizemos que ele tem "alta ortogonalidade". O sistema de comandos do Jujutsu é realmente muito ortogonal. Depois de usar algumas vezes, você se pega digitando "isso provavelmente se escreve assim..." e acertando.

— Eu nunca tinha ouvido o termo “ortogonal” antes, mas acho que basicamente quer dizer simples e intuitivo, né.

— Outro ponto que vale mencionar é que os comandos do Jujutsu têm pouquíssima dependência de estado. Os comandos do Git sempre partem do HEAD[12], e sofrem influência do estado do working tree e da staging area, então executar o mesmo comando em situações diferentes pode dar resultados bem distintos. No Jujutsu não há staging area e as mudanças são commitadas automaticamente, então o working copy é só um nó entre muitos no grafo do histórico. E como muitos comandos podem receber um ponto específico do grafo do histórico como argumento, você pode operar diretamente sobre esses pontos independentemente de onde o working copy esteja.

— Hmm, tipo o quê?

— O `jj squash` é o comando que combina histórico. Se você especificar explicitamente a origem e o destino com `jj squash -f <from-id> -t <to-id>`, o resultado é o mesmo, não importa em que ponto do grafo o working copy esteja. Se você tentasse fazer a mesma coisa no Git, teria que dar um `checkout` no local de destino antes.

— Ah, faz sentido. Mas a abordagem do Jujutsu de sempre especificar o alvo com uma opção tem a desvantagem de mais teclas digitadas, não tem?

— É por isso que muitos comandos têm valores padrão quando uma opção é omitida. Executar `jj squash` sem argumentos absorve o working copy no seu pai. Você também pode indicar só `-f` ou só `-t`.

— Entendi. Aliás, o que tem de tão bom em ter pouca dependência de estado?

— Você pode fazer a operação que quiser de onde estiver. Não precisa ficar movendo o working copy de um lado para o outro. Você também se livra da carga cognitiva de ter que entender com precisão o estado atual antes de cada operação, e indicar o alvo de forma explícita reduz os erros de operação. Depois que você se acostuma com o Jujutsu, os comandos ortogonais e com pouca dependência de estado passam a parecer tão naturais que você nem repara mais neles. Mas de vez em

[12] No Git, um ponteiro que indica onde você está trabalhando no momento. Normalmente aponta para o commit mais recente do branch atual.

quando, quando você precisa voltar para o Git para alguma coisa, sente de verdade o quanto o Jujutsu é mais conveniente.

Resumo: o Jujutsu é um VCS em que você começa no rascunho e organiza depois

— Juntando todas essas características — disse Yukina —, a filosofia que está por baixo do Jujutsu começa a aparecer. O estilo é: "comece logo no rascunho, experimente à vontade e organize depois".

- O working copy é commitado de forma automática e contínua, então você não precisa pensar se as suas edições estão salvas.
- Você não precisa mirar uma granularidade de changes perfeita nem achar os nomes certos desde o começo. Quando precisar, você pode dividir, combinar e reordenar os changes, e renomeá-los para refletir como as coisas de fato ficaram.
- Toda operação pode ser desfeita, então você pode experimentar sem medo de errar.
- Os conflitos não são exceções que bloqueiam; são um estado que um commit pode manter. Você pode adiar a resolução e continuar trabalhando.
- Os comandos ortogonais e com pouca dependência de estado deixam você remodelar o histórico na hora, sempre que a inspiração bater.

— Entendi... É exatamente o oposto do Git, onde você tem que deixar tudo definido lá no começo e qualquer mudança depois é um transtorno.

— Esse estilo do Jujutsu tem muito em comum com o desenvolvimento ágil, que parte do princípio de que os humanos cometem erros e busca elevar a qualidade por meio de pequenos ajustes repetidos.

— Ah, é verdade.

— Os agentes de IA erram pelo menos tanto quanto os humanos, e iteram e geram código em velocidades que os humanos não conseguem acompanhar. A afinidade do Jujutsu com os agentes de IA não é coincidência. É a consequência natural de um design pensado obsessivamente para o cérebro humano.

— Gentil com os humanos e com a IA... Agora fiquei mesmo interessada. Quero testar o quanto antes!

Coluna: como o Git revolucionou o controle de versão

O capítulo 1 construiu seu argumento pelo ângulo de que o design do Git ficou fora de sintonia com os tempos. Mas não seria nada justo parar por aí sem também reconhecer o enorme papel que o Git desempenhou na história do controle de versão. Uma das motivações por trás do desenvolvimento do Jujutsu foi resolver os pontos problemáticos do Git. Visto pelo outro lado, isso significa simplesmente que o Jujutsu é um produto construído sobre os ombros de um grande predecessor.

O Git começou a ganhar terreno no setor no início da década de 2010. Antes disso, controle de versão significava em geral um produto centralizado, e o Subversion[13] em particular era muito usado. Como todos compartilhavam um único repositório central, muitas operações do dia a dia dependiam do servidor. Os commits e a consulta ao log, em especial, exigiam conversar com o repositório central, e o trabalho offline era bastante limitado. Fazer qualquer trabalho de peso offline era basicamente inviável e, como um commit era compartilhado na hora com o resto do time, você não conseguia registrar o histórico em passos finos enquanto o seu trabalho ainda estava pela metade. Por isso não era raro ver fluxos de trabalho que tiravam de um VCS metade do seu valor: acumular um monte de mudanças localmente e fazer um único commit só quando tudo finalmente estava pronto para publicar, ou juntar o trabalho de um dia inteiro em um só commit na hora de ir embora.

A disseminação do Git mudou tudo isso da noite para o dia. Uma cópia quase completa do histórico vive sempre na própria máquina do usuário, e a maioria das operações do dia a dia se completa localmente. Como você faz commit no seu próprio repositório local, pode dar forma ao seu histórico na granularidade que quiser, inteiramente a seu critério. E mesmo que o servidor caia ou você esteja trabalhando offline, dá para continuar.

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

No Git, o repositório central não passa, conceitualmente, de um ponto de sincronização para compartilhar o trabalho entre os membros do time; cada membro mantém um repositório com um histórico quase completo próprio. Mesmo que o repositório central seja corrompido, ele pode ser restaurado a partir do repositório de algum membro. No Subversion, os arquivos na máquina de um membro eram subordinados ao repositório central, mas no Git cada membro mantém um repositório que fica em pé de igualdade com o central. Foi uma mudança que transformou a estrutura de poder do desenvolvimento — e a própria mentalidade de quem participava dele.

O Git também era absurdamente rápido em todo tipo de operação em comparação com os VCS existentes na sua época. Parte disso se deve ao fato de que a maioria das operações se completa localmente, mas um fator importante é que seu alvo de design era o kernel do Linux, então, desde a fase de design, ele estabeleceu requisitos de desempenho em torno de coisas como o baixo volume de transferência pela rede e a velocidade da consulta ao log e dos merges. Enquanto os VCS convencionais registravam as sucessivas diferenças de um arquivo a partir do seu estado inicial, o Git registra um snapshot de toda a árvore de arquivos no momento do commit. Para extrair uma versão específica, ele simplesmente lê diretamente o ponteiro do commit correspondente, o que o torna extremamente rápido.

O Git também mudou o que significa um branch. No Subversion, um branch era apenas uma cópia do projeto inteiro criada em um diretório separado, então ramificar trazia uma sobrecarga e um atrito consideráveis. Além disso, seu modelo de dados rastreava mal os merges, o que tornava prováveis conflitos desnecessários, e muitos times detestavam isso a ponto de evitar branches por completo. No Git, por outro lado, o próprio histórico é estruturado como um DAG[14], e um branch não passa de um ponteiro para um commit

[14] Sigla de Directed Acyclic Graph (grafo acíclico dirigido): uma estrutura de grafo em que nós conectados por arestas dirigidas nunca formam um ciclo.

específico — um design notavelmente elegante[15]. Os branches do Git são leves de manusear, produzem menos conflitos no merge e podem ser criados de forma puramente local, o que reduziu drasticamente o seu custo psicológico. Estilos de desenvolvimento como criar um feature branch para cada funcionalidade pequena, trabalhar em vários branches em paralelo ou usar um branch local como espaço de testes pessoal só se tornaram possíveis depois que o Git se firmou.

Dá para dizer que o Git mudou o próprio status do controle de versão: de algo que você "tinha que usar" sob o controle de uma empresa ou organização para algo que você "queria dominar por vontade própria" para aumentar a sua própria produtividade. A chegada do Git democratizou o próprio controle de versão e, a meu ver, colocou o histórico e seu poder igualmente nas mãos de cada desenvolvedor individual.

[15] Explicado em "2-4-2. A diferença entre branch e bookmark".

Capítulo 2. Vamos experimentar o Jujutsu

2-1. Configurando o Jujutsu

2-1-1. Instalando o Jujutsu

— De agora em diante, quero que você aprenda Jujutsu usando ele de verdade — disse Yukina. — Mas vamos por partes: primeiro precisamos instalá-lo. Como o Jujutsu é escrito em Rust, a documentação oficial começa com um procedimento baseado em Cargo, mas, a menos que você seja uma fã ferrenha de Rust, é melhor evitar esse caminho. A compilação demora um bom tempo e manter tudo atualizado fica trabalhoso.

— Certo — disse Kanae. — Não tenho planos de mexer com Rust tão cedo, então prefiro não começar montando um ambiente Rust inteiro só para isso.

— Por sorte, nós duas estamos no Mac, então dá para instalar fácil com o Homebrew, assim:

```
$ brew install jj
```

— Esta próxima parte não é obrigatória, mas adicionar um pouco de configuração de shell deixa a execução dos comandos mais prática. Dependendo de qual shell você usa, adicione uma destas linhas ao seu arquivo de configuração:

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

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

— E o que isso faz, na prática?

— Quando você executa `jj` com a variável de ambiente `COMPLETE=zsh` definida, ele imprime um script de autocompletar do shell para o zsh. Ao fazer o zsh carregar esse script na inicialização, as definições de autocompletar mais recentes do comando `jj` são geradas e aplicadas de

forma dinâmica. Assim, quando você escreve parte de um subcomando ou de uma opção e aperta `Tab`, ele mostra uma lista de sugestões ou autocompleta o comando para você.

— Ah, isso parece útil. Vou adicionar na minha configuração também. Aliás, lá em casa eu tenho uma máquina com Windows. Como eu instalaria lá?

— Num ambiente WSL[16], o Homebrew é o caminho, igual no Mac. Para um ambiente nativo, o pacote está registrado no WinGet como `jj-vcs.jj`, então instale esse. Só fique atenta a alguns pontos sobre finais de linha e links simbólicos, então leia com calma a página correspondente da documentação oficial[17].

— Certo.

— Mais uma coisa, que não é diretamente relacionada ao Jujutsu: se você ainda não instalou a GitHub CLI (`gh`)[18], este é um bom momento. Permitir que um agente de IA execute o comando `gh` junto com o `jj` deixa o seu fluxo de trabalho bem mais eficiente.

2-1-2. Configuração inicial

— O Jujutsu tem um número enorme de ajustes configuráveis — continuou Yukina —, mas no começo só uns poucos importam. Primeiro, vamos registrar o seu nome de usuário e o seu endereço de e-mail.

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

— `jj config set`[19] é o comando para escrever valores no seu arquivo de configuração. Com `--user`, ele atua sobre os ajustes no nível de usuário; com `--repo`, os ajustes limitados àquele único repositório.

---

[16] Sigla de Windows Subsystem for Linux. Um recurso da Microsoft para rodar ambientes Linux no Windows. O WSL moderno usa uma máquina virtual leve com um kernel Linux real, o que permite instalar e usar distribuições como Ubuntu ou 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

— O que acontece se você começar a usar sem configurar isso?

— As suas informações de autor vão ficar faltando no histórico, e você não vai conseguir fazer push para um remoto. Aí você teria que configurar a sua informação de usuário depois e executar `jj metaedit --update-author <TARGET>`[20] para atualizar a informação de autor no seu histórico.

— Isso parece um saco. Vou tomar cuidado para não esquecer disso numa instalação nova.

— Vamos mudar também a configuração do editor. Por padrão, o editor que o Jujutsu abre quando precisa de um é o nano, que a maioria das pessoas não conhece bem[21].

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

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

— O que é essa opção `--wait` que você passa ao abrir o VS Code?

— Quando o Jujutsu abre um editor, ele precisa detectar quando o editor é fechado para conseguir pegar o que você escreveu. Se você abre um editor gráfico pelo terminal, o normal é que ele seja iniciado como um processo separado, e as suas edições nunca voltam para o Jujutsu. Adicionar `--wait` faz o VS Code manter o processo do terminal bloqueado até você fechar a aba do editor, de modo que o Jujutsu consegue reconhecer que você terminou e receber as suas edições corretamente.

— Ah, entendi, faz sentido.

— Aliás, se você não conseguir abrir o VS Code digitando `code` no terminal, abra a paleta de comandos com `⌘` + `Shift` + `p`, digite "shell" e execute "Shell Command: Install `code` command in PATH" que aparece nos resultados. Com isso vai funcionar.

---

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

[21] O comportamento no macOS e no Linux baseado em Ubuntu quando a variável de ambiente `$EDITOR` não está definida. No Windows, o Bloco de Notas é aberto no lugar.

— Agora, com isso configurado — prosseguiu Yukina —, vamos executar `jj config edit --user`[22]. O editor que você indicou antes deve abrir e mostrar algo assim:

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

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

[ui]
editor = "vim"
```

— E onde esse arquivo fica, afinal? — continuou Yukina. — Isso você também consegue descobrir com o `jj config`.

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

— Então editar esse arquivo diretamente também aplica as configurações, né?

— Claro. Então, se você guardar esse arquivo em um diretório compartilhado como o Dropbox e criar um link simbólico para ele, dá para compartilhar as suas configurações de usuário do Jujutsu entre várias máquinas.

2-2. Um tour prático pelo Jujutsu

2-2-1. Inicializando um repositório

— Beleza, vamos começar a versionar alguns arquivos com o Jujutsu — disse Yukina. — Um app React é um bom material didático, eu acho. Vamos usar o Node.js como runtime e o pnpm para gerenciar os pacotes, então, se você ainda não os tem, configure-os primeiro. E para instalá-los, vamos usar o mise[23].

---

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

[23] Um gerenciador de pacotes para instalar runtimes de linguagens de programação e diversas ferramentas de CLI, gerenciando várias versões de cada

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

— Agora vamos criar o projeto React, inicializá-lo e colocá-lo sob o gerenciamento do 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
```

— O comando para inicializar é `jj git init`[24], não `jj init`?

— O Jujutsu usa uma arquitetura de armazenamento plugável e pode usar o Git como armazenamento de backend. Então esse comando inicializa um repositório Git com o Jujutsu montado por cima. Alguns artigos mais antigos sugerem usar a opção `--colocate` para que os repositórios do Jujutsu e do Git convivam no mesmo diretório, mas hoje em dia esse comportamento é o padrão, então você não precisa dela.

— E se eu tiver um repositório que já é gerenciado com Git e quiser começar a usar Jujutsu nele mais tarde?

---

uma. Também funciona como executor de tarefas e gerenciador de variáveis de ambiente.
https://mise.jdx.dev/

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

— O mesmo comando: `jj git init` funciona. O histórico de commits que você criou com o Git é importado como histórico do Jujutsu do jeito que está.

2-2-2. Verificando o estado do repositório

— A seguir, vamos verificar o estado atual deste repositório — disse 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)
```

— Você pode pensar no `jj status`[25] como algo que cumpre mais ou menos a mesma função que o `git status`. A diferença é que o `git status` serve para verificar o estado do seu working tree e da sua staging area ainda não commitados, enquanto o `jj status` serve para verificar o estado do seu working copy, que já foi commitado. Isto é o que o comando mostra:

- Um resumo de quais mudanças foram feitas no working copy
- Informações sobre o commit e o seu commit pai
- Se houver um conflito, informações sobre ele

— Tem uma lista inteira de nomes de arquivo, todos os que foram criados quando o projeto foi montado, alinhados com um `A` na frente de cada um. Pelo que você acabou de explicar, isso quer dizer que eles já foram commitados?

— Exato. Então vamos dar uma olhada no histórico desse commit.

---

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

Capítulo 3. Fluxo de trabalho Jujutsu × IA na prática

3-1. Coordenando os agentes de IA com o Jujutsu

3-1-1. Fazendo os agentes de IA usarem o Jujutsu

— Ultimamente eu comecei a deixar a IA cuidar do Git para mim também — disse Kanae —, tipo, eu só falo "beleza, faz commit do que a gente tem até agora e dá push". Dá para delegar as operações do Jujutsu para uma IA assim de verdade?

— Claro que dá — respondeu Yukina. — O Jujutsu deixa você voltar no histórico com tranquilidade e segurança, então dá para delegar o controle de versão com mais ousadia do que com o Git. E, como não existe staging e as mudanças entram num commit automaticamente, dá para montar as coisas de um jeito que basta passar uma tarefa para você ficar com um histórico organizado em unidades que fazem sentido, sem precisar detalhar os passos de controle de versão.

— Isso é ótimo. Então você vem combinando Jujutsu com desenvolvimento baseado em agentes de IA esse tempo todo, né? Eu imaginava que um agente de IA só fosse recorrer ao Git por padrão, então o que você vem fazendo para ele usar o Jujutsu? Esse conhecimento seria valiosíssimo para mim, então eu adoraria que você compartilhasse!

— Claro. A questão é que os agentes de IA se diferenciam bastante nas funcionalidades e nas manias, e capacidades novas vêm sendo adicionadas num ritmo vertiginoso, então ainda é difícil fixar as boas práticas. Entre os agentes de codificação, o Claude Code tem a maior fatia de mercado e eu sou uma usuária intensiva, então vou focar no que eu acho que é a melhor abordagem para ele em agosto de 2026. Também vou trazer o Codex CLI quando for pertinente.

— Sim, por favor. Esses são basicamente os dois que eu também uso no dia a dia.

— Beleza. Vamos começar por como orientar um agente de IA para o Jujutsu em vez do Git como VCS. Na verdade, os modelos mais recentes por trás do Claude Code e do Codex já entendem os conceitos e os

comandos básicos do Jujutsu, então só escrever "use Jujutsu, não Git" no `CLAUDE.md` ou no `AGENTS.md` já resolve parte do problema.

— Ah, eles já sabem usar?

— Ainda assim, eles foram treinados para tarefas de programação majoritariamente em ambientes com Git, então um fluxo de trabalho nativo do Jujutsu não é natural para eles. Só com essa instrução simples, você pode esbarrar em coisas assim:

- Trocar mecanicamente cada comando `git` pelo equivalente `jj` mais parecido
- Começar uma tarefa sem verificar em que estado está o working-copy commit, misturando trabalhos distintos no histórico
- Tratar um bookmark como um branch do Git, que avança automaticamente a cada commit novo
- Lidar mal com os bookmarks na hora de dar push
- Parar o trabalho assim que aparece um conflito
- Não saber reagir quando surge um problema específico do Jujutsu

— Conhecer os comandos não é a mesma coisa que ter passado por um fluxo de trabalho completo e desenvolvido o discernimento necessário ao longo do caminho — disse Yukina. — Um modelo com bom raciocínio até pode se sair bem em alguns casos, mas se apoiar nisso faz o resultado depender da sorte e ficar difícil de reproduzir. Então você precisa de um jeito de ensinar à IA um fluxo de trabalho que siga o modelo mental do próprio Jujutsu.

— Faz sentido.

— Para isso, com o Claude Code eu acho que o melhor é basear o fluxo principalmente nas **Rules**[54]. No `CLAUDE.md` você escreve apenas que o projeto usa Jujutsu para controle de versão. Os passos concretos do fluxo de trabalho ficam no arquivo de Rules.

— E por que não escrever tudo no `CLAUDE.md`?

— É o seguinte: no Claude Code, o `CLAUDE.md` e um arquivo de Rules são carregados no mesmo momento[55], e tanto faz em qual dos dois você

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

[55] Este é o comportamento quando não há frontmatter `paths`. Se você definir um campo `paths` no frontmatter de um arquivo de Rules, esse arquivo é carregado e

escreve a instrução: a prioridade é a mesma. O único motivo para separar as instruções num arquivo de Rules é a facilidade de manutenção. O `CLAUDE.md` também carrega informações e instruções específicas do projeto, então acrescentar um bloco grande de convenções de controle de versão só gera confusão. Separar esse conteúdo facilita atualizá-las e revisá-las, e permite reaproveitar as mesmas regras em outros repositórios. Quanto a proibir comandos específicos, disso quem cuida é a configuração de **Permissions**. A gente vê isso em detalhe mais adiante.

— Aham, então as Rules são mais uma questão de manter tudo administrável. E as **Skills**, o que você acha? Se você busca "jujutsu" no skills.sh[56], aparece um monte de skills. Só a quantidade já sugere que elas são bastante usadas.

— Duvido que funcionem tão bem assim. As skills são materiais de referência que um agente de IA consulta quando *ele* considera que são relevantes; não são restrições de comportamento sempre ativas. Na maioria das vezes, você precisa invocá-las explicitamente. Se você pede expressamente uma operação no repositório, tipo "dá push nisso", tudo bem. Mas, quando só pede que ele implemente uma funcionalidade ou depure um problema, há um risco real de o agente executar `git` sem sequer ativar a skill.

— Entendi.

— Então, o que você põe de fato nesse arquivo de Rules? Em linhas gerais, o seguinte:

1. Restrições sobre os diversos comandos
2. Facilitar a leitura dos diffs pela IA, por exemplo passando `--git` aos comandos que os exibem
3. Passar `--ignore-working-copy` aos comandos somente leitura quando fizer sentido, o que evita a fragmentação das revisões e previne os erros de stale working copy
4. Como tratar os changes no início de uma tarefa

---

aplicado apenas quando o agente trabalha com arquivos que correspondem a esses padrões.

[56] Um diretório aberto e site de ranking lançado pela Vercel em janeiro de 2026 onde você pode buscar e gerenciar skills para agentes de IA.
https://skills.sh/

5. Procedimentos para dividir um change e para resolver conflitos
6. Como lidar bem com os bookmarks na hora de dar push
7. O que fazer quando ocorre um erro de stale working copy

— Esta lista vem em parte dos problemas que eu realmente enfrentei ao fazer os agentes de IA usarem o Jujutsu — continuou Yukina. — O resto saiu de percorrer casos hipotéticos com o agente um por um e deixar que ele mesmo me dissesse do que precisava. Já faz alguns meses que eu trabalho assim e, para os fluxos de trabalho de desenvolvimento comuns que passam pelo GitHub, deve dar conta sem problemas.

— A opção `--git` do item 2 está aí porque um diff no formato do Git é mais fácil para a IA interpretar, já que é o formato com que ela foi treinada, né? As skills de Jujutsu que circulam por aí não parecem dar muita atenção a esse aspecto. E o que é o item 3, `--ignore-working-copy`?

— Normalmente, quando você roda `jj` enquanto há um diff entre o working copy e o histórico, o Jujutsu tira um snapshot. Essa opção desativa esse comportamento. Um dos motivos é evitar que um snapshot seja tirado toda vez que um arquivo muda, o que dividiria o evolution log em partes muito menores do que o necessário.

— E o outro motivo?

— Enquanto você espera a IA terminar uma tarefa, muitas vezes você pula para outro painel e faz outro trabalho, né? Se o seu próprio comando `jj` e o comando `jj` da IA acabam se sobrepondo, um dos dois pode tirar um snapshot e avançar o histórico antes de o outro ter terminado. Aí o working-copy commit que um dos processos acha que está olhando acaba mais antigo do que o working-copy commit que o histórico de fato reflete, e o descompasso gera um erro. Eu passo essa opção para evitar ao máximo esse tipo de situação. E o item 7 cobre o que fazer quando isso acontece mesmo assim.

— Hmm, você tem que pensar até nesse nível de detalhe.

— Deixa eu comentar também como fazer isso com o Codex CLI. O Codex não tem uma funcionalidade Rules equivalente à do Claude Code. Existe uma funcionalidade com o mesmo nome, *rules*, mas serve para gerenciar quais comandos podem rodar fora do sandbox, não para dar ao agente de IA diretrizes de comportamento em linguagem natural.

— Que problema. E aí, o que você faz?

— A segunda melhor opção é combinar o `AGENTS.md` com *skills*. Mas, como eu disse, as skills só disparam quando a IA decide que são relevantes, então, neste caso, você acaba colocando bastante conteúdo no `AGENTS.md`.

— Hmm, achar o equilíbrio certo é delicado. Eu não esperava que você tivesse que mudar tanto toda a abordagem dependendo do agente.

— Juntando tudo isso, os arquivos de configuração por projeto dos seus agentes de IA acabam dispostos assim[57]:

Listagem 1: Disposição dos arquivos de configuração dos agentes de IA

```
my-project/
  CLAUDE.md             ← instrui a usar o Jujutsu
  AGENTS.md             ← comportamento básico e quando as skills
disparam
  .claude/
    rules/
      jujutsu-rules.md  ← o arquivo de Rules em questão
  .agents/
    skills/
      jujutsu/
        SKILL.md        ← o mesmo conteúdo empacotado como skill para o
Codex
```

Listagem 2: Início do 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

### Command Constraints

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

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

```
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 os seus repositórios existentes provavelmente são gerenciados com Git, é melhor limitar essa configuração aos repositórios específicos que usam Jujutsu, em vez de colocá-la na sua configuração no nível de usuário.

— Ótimo! Vou pegar isso exatamente como está!

[58] "Configure Permissions — Claude Code Docs"
https://code.claude.com/docs/en/permissions

Sobre as autoras

Yuka Ooka

Yuka Ooka começou a carreira como desenvolvedora de PHP e Ruby e como product manager em várias empresas de tecnologia japonesas, antes de se tornar autônoma em 2016. Ela se interessou cedo por React, que na época ainda era uma tecnologia de nicho no Japão, e trabalhou como especialista em React para diversos clientes. A partir dessa experiência, escreveu a série *Riakuto!* (りあクト！), que caiu no gosto dos leitores no Japão e vendeu mais de 40.000 exemplares no total.
Ela está no X como @oukayuka e escreve sobre como dominar o Jujutsu em seu blog, juju-chu.com/pt/blog.

O setup de trabalho da autora

Ilustração: Megumi Kuroki

Mangaká e ilustradora.
Assina as ilustrações de capa das séries *Riakuto!* e *Juju-chu!* desde o início.