JuJu-Chu!

Sample

Yuka Ooka

Starting Your Jujutsu×AI Workflow with `jj new`

Version control finds its next evolution in the AI era.

DOMAIN EXPANSION!

【Jujutsu Features】

- Auto-commit
- Universal Undo
- First-class Conflicts

Juju-chu!

Starting Your Jujutsu × AI Workflow with `jj new`

Yuka Ooka

Klemiwary Books

Copyright Page

Juju-chu!
Starting Your Jujutsu × AI Workflow with `jj new`

Copyright © 2026 Yuka Ooka

All rights reserved. No part of this book may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the author, except for brief quotations in reviews or as otherwise permitted by law.

This book is provided for informational purposes only. The author makes no warranties regarding the accuracy or completeness of its contents, and assumes no liability for any damages arising from its use.

Jujutsu (jj) and other product, service, and company names mentioned herein may be trademarks of their respective owners. They are used for identification and editorial purposes only, with no intent to infringe.

First English edition: July 2026
First published in Japanese: April 2026

Published by Klemiwary Books
Support repository: https://github.com/klemiwary/Juju-chu-en

Table of Contents

Preface

Jujutsu is drawing a lot of attention right now.

When it comes to version control systems (VCS), Git has reigned supreme for a long time. This is especially true of engineers who entered the industry in the 2010s or later: many of them have never used anything but Git. GitHub, for its part, was the single biggest force that made Git the de facto standard, and it now has well over 180 million developers worldwide. It's so ubiquitous you could almost get away with saying that a software engineer without a GitHub account isn't really a software engineer at all.

Against that backdrop, it's incredibly rare for a new VCS to gain traction—so rare that when it happens, it feels almost like an event.

Why Jujutsu, Why Now?

Development of Jujutsu began in 2019, but its first public release (v0.3.0) didn't arrive until 2022. It didn't start coming up regularly in developer circles until 2025. That timing lines up with the rapid spread of AI coding agents and the moment the term "vibe coding" started catching on as a buzzword.

AI agents can now generate and modify huge amounts of code at speeds no human can match. The cost of writing code—and of iterating on it—has dropped dramatically. And I suspect a growing number of people have started to feel a mismatch between that new reality and traditional version control systems. Those tools were designed around a very different premise: that "humans write code carefully line by line, consciously deciding what to record, and sharing it only when everything is ready."

And then, almost overnight, Jujutsu was suddenly there. They tried it, and it turned out to be a remarkably good fit for the pace of agentic development. "I have to tell everyone about this!" That, I think, is how the recent surge of excitement came about.

"But You Can Do That in Git Too"

Almost every time an introductory article goes viral and Jujutsu becomes a hot topic, you'll inevitably hear one common refrain: "But you can already do that in Git." But Jujutsu's value does not lie in doing what Git can't.

Think of the BlackBerry versus the iPhone. Both could make calls, browse the web, install and run apps, and take photos. When the iPhone first appeared, many people, especially BlackBerry loyalists, dismissed it using that exact argument. But we all know how that rivalry ultimately turned out. Even though they could do the same things, there was a decisive difference in the user experience each one delivered.

"Being able to do something" and "being able to do it comfortably" are not the same thing. Even when you're doing the very same thing in Jujutsu that you'd do in Git, Jujutsu automates more of the work, the operations are easier to remember, and mistakes are easy to undo. So the whole experience is more reassuring, safer, and less effortful.

The key ideas are **low cognitive load** and **psychological safety**. Everything about Jujutsu is built with careful attention to those two things.

Who This Book Is For

This book is written with the following readers in mind:

- You know basic Git operations and use Git and GitHub/GitLab on a daily basis.
- You use an AI coding agent like Claude Code or Codex CLI, or you're planning to start.
- You have no experience with Jujutsu, or you've started using it but are still a beginner.

To be clear, this is not a beginner's guide to Git. I do use React as teaching material for demonstrating VCS workflows, but since that isn't the point, knowledge of React or the Node.js ecosystem is not required. You can map the examples to your own primary tech stack as you read.

A Few Notes Before You Read

As of mid-2026, Jujutsu is under active development and has not yet reached v1.0.0. In fact, the project's own README describes it as an "experimental version control system" and states explicitly that breaking changes may land before v1.0.0. As a result, some command names and behaviors in this book may not match the latest version. That said, this book covers far more than just how to use commands: it's a comprehensive treatment that extends to Jujutsu's design philosophy, its mental model, and how to apply it to agentic coding. Even if surface-level changes happen down the line, I'm confident the substance of this book will remain useful well into the future.

How This Book Is Organized

Chapter 1 gives an overview of Jujutsu, Chapter 2 walks through the basic operations, Chapter 3 covers collaborative development with AI, Chapter 4 covers a range of advanced applications, and Chapter 5 is an FAQ and troubleshooting guide. I recommend reading it in order from the beginning. But if you'd rather get hands-on first, you can jump straight to Chapter 2, and if you're especially interested in working with AI, you can skip ahead to Chapter 3. Chapters 4 and 5 are also designed to double as a reference.

One more thing: this book unfolds as a story, told through a dialogue between two characters—a senior engineer and a junior engineer. I chose this dialogue format to keep the guide accessible and to systematically address the questions a beginner is likely to encounter.

Now, let me welcome you into the world of Jujutsu. The opening incantation: `jj new`.

About This Book

Cast of Characters

Yukina Shibasaki

A senior engineer at a Tokyo-based internet services company. She was brought on board for her React skills and, as tech lead, built the front-end development team from the ground up. She has a habit of finding new technologies and pushing to adopt them before they're fully proven. This sometimes exasperates her teammates, but somehow those picks have a way of breaking into the mainstream later, so the team generally tolerates it.

Kanae Akiya

An enthusiastic junior engineer on Yukina's team. She considers herself Yukina's star pupil and tries to follow her lead in everything. She loves anime and manga. Although she worries that AI might take her job, she actively uses AI agents in her development work and is determined not to be left behind.

About the Sample Code

The sample code and configuration files introduced in this book are available in the GitHub repository below. For the location of each file, see the corresponding footnote in the main text.

- Public companion repository for *Juju-chu! Start Your Jujutsu × AI Workflow with `jj new`*
  https://github.com/klemiwary/Juju-chu-en

You're free to use the code shown in the main text and in the repository above however you see fit. You may also cite or excerpt it in public documents, videos, and similar materials without asking the author's permission, as long as you credit the source. That said, please refrain from republishing the code in its entirety.

Errata

An errata list covering errors and typos in the text is available at the link below. It will be updated as needed.

- Errata for *Juju-chu! Start Your Jujutsu × AI Workflow with `jj new`*
  https://github.com/klemiwary/Juju-chu-en/blob/main/errata.md

Software Versions Used in This Book

- Jujutsu 0.43.0
- jjui 0.10.7
- JJ View 2.2.0
- Claude Code 2.1.201
- Codex CLI 0.142.5

Prologue

"Oh no, no, no, no...!!" Kanae cried.

Yukina looked up. "What's with the sudden outburst? What happened?"

"I had Codex writing some code for me, and I forgot to commit my previous changes before giving it another prompt. Then it overwrote my existing files with changes I never asked for, and now I can't get them back..."

"Ah, the AI agent tax. If you were on Claude Code, you could roll it back with the `/rewind` command. But Codex CLI doesn't have anything equivalent.[1]"

"Lately Codex has been faster and better at powering through tough problems, so I've been favoring it. I should've just stuck with Claude Code..."

"That said, Claude Code's `/rewind` isn't a silver bullet either. It only tracks code written by the agent. If you've run shell commands or made manual edits in the meantime, those changes won't be captured. And if you try to use `/rewind` to step back through several points in history, it ends up tangled with the Git history and you can easily get into a mess you can't recover from."

"Ugh, so none of these tools can cleanly handle the one thing I actually need—getting back to exactly that point in time, huh... Yukina, you're way better at working with AI than I am. Don't you ever have these kinds of screw-ups?"

"I pretty much only use **Jujutsu**, so I almost never run into that kind of accident."

"**Jujutsu?!** Wait, Yukina, you're a *jujutsu sorcerer*?! And time manipulation on top of that? That's easily Grade 1 or higher, right? ...Come to think of it, you do kind of look like Maki Zenin.[2]"

---

[1] Status as of June 2026. Codex CLI previously offered an experimental `/undo` command, but it was later removed due to frequent bugs and concerns that its history management conflicted with VCSs and confused users.

[2] A character in the manga *Jujutsu Kaisen*. Her cursed energy is no higher than an ordinary person's, but she has superhuman physical abilities and fights using a

"You're only saying that because of the glasses, aren't you. I'm not talking about the kind of *jujutsu* you see in *Jujutsu Kaisen*[3]. I mean Jujutsu, the next-generation version control system that's been getting a lot of attention lately."

"Huh, there's a tool like that? I didn't know. When it came to version control, I only really knew Git."

"That's probably true for most junior developers entering the industry today. But there have actually been plenty of VCSs before and after Git, and the one whose popularity is climbing fastest right now is Jujutsu."

"Still, Yukina, how long have you been using this Jujutsu thing? We're on the same team and I had no idea. ...Wait, but on GitHub it looks like you're using Git like everyone else. What's going on there?"

"Right, Jujutsu is compatible with Git: it can use a Git repository directly as its backend storage. So unless I bring it up myself, my teammates can't tell I'm using Jujutsu. I think I started using it about six months ago, actually."

"Wait, that long ago?! You've been keeping something this useful to yourself this whole time? After all we've been through together..."

"Sorry, sorry. It doesn't force the rest of the team to change their workflow, and I didn't think you'd be all that interested."

"But if I were using Jujutsu, I could still recover even when something like this happens, right? I'm beyond interested! Please teach me how to use Jujutsu! I want to become a Grade 1 Jujutsu sorcerer too and manipulate time."

"Well... alright. Let me make some time right now and walk you through Jujutsu."

"Wait, really? Won't that hold up the project?"

---

variety of cursed tools. Since she can't see cursed spirits with the naked eye, she normally wears a special pair of glasses.

[3] A manga by Gege Akutami, serialized in *Weekly Shōnen Jump*. A dark fantasy depicting the battles of Jujutsu Sorcerers who exorcise the monsters and cursed spirits born from humanity's negative emotions. The series has more than 150 million copies in circulation worldwide. The English title is also *Jujutsu Kaisen*, which, together with the traditional Japanese martial art of *jujutsu*, doesn't exactly make the VCS any easier to search for.

"If adopting Jujutsu drastically reduces the time you waste on recovery, it'll more than pay for itself in the long run. As team lead, I'm authorizing this as a special exception."

"Yes! A one-on-one Jujutsu lesson with Yukina! I can't wait!"

Chapter 1. What Kind of Tool Is Jujutsu?

1-1. Is Git a Poor Fit for Agentic Coding?

"That accident you just had, Kanae, isn't because you were being unusually careless. It happened because Git, the VCS you're using, wasn't designed to mesh with modern agentic development. It was an accident waiting to happen. To be fair, Git was designed over twenty years ago, so it's a bit much to expect it to have anticipated a world where AI agents are churning out massive amounts of code at breakneck speed."

"...Oh. So that's what was going on. It's not my fault!"

"Before we get into Jujutsu itself, let's first nail down exactly where Git falls short for agentic coding. That way, as you learn Jujutsu, you'll really feel where its convenience and ease of use come from."

Git's Redundant Three-State Model

"Git's biggest defining feature, and also the thing newcomers stumble over first," Yukina said, "is that, changes to your files pass through three distinct states before they're recorded in history:"

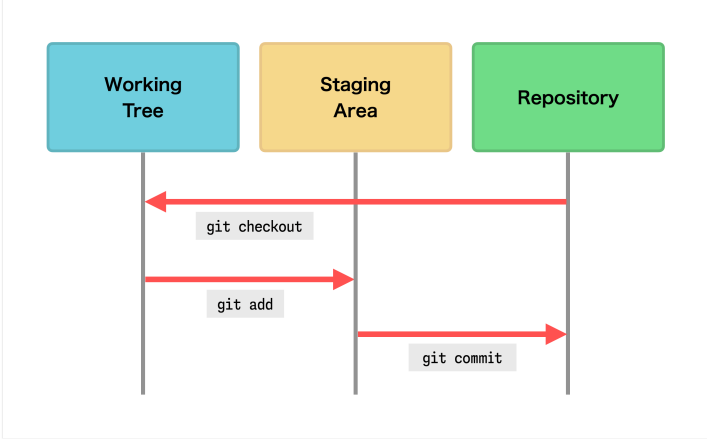

Figure 1: Git's three-state model

- **Working tree** — the place where you actually edit files. Also called the working directory.
- **Staging area** — the area between the working tree and the repository, where you prepare a commit. Also called the index.
- **Repository** — the database that stores your full change history and all the files.

"Right. When I first started, I always wondered why I had to bother with `git add` and then `git commit`. Why isn't `commit` enough on its own?"

"Back when humans were doing all the development by hand, there was a real logic to this model. The idea is that you pick out just some of the changes from your working tree, stage them, and then turn each meaningfully grouped set of staged changes into a commit. The staging area is meant to be a draft area for your next commit. Having it helped developers work carefully and made it easier to land high-quality commits. Remember, Git was originally built for Linux kernel development, so landing messy commits whose intent you couldn't read would make life miserable for the maintainers."

"Hmm, I see."

"But in today's world, where an AI agent generates dozens or hundreds of lines of code across multiple files in a single run, that extra step has turned into a serious drag. It just isn't realistic to have a human keep running `git add` and `git commit` over and over to keep up with a tireless AI cranking out new code. The end result is that people give the AI its next instruction without ever recording the previous state, and then either can't roll back the work to redo it, or end up drowning in a sea of history. That's exactly the kind of accident that keeps happening."

The High Cost of Context Switching

"In real development," Yukina continued, "you'll often suddenly want to try out an idea, or you get hit with a review request or an urgent bug fix out of nowhere. That forces you to switch to a different branch while your working tree is still in a half-finished state. When that happens in Git, you reach for `git stash` to temporarily set things aside."

"Yep, yep, I stash stuff pretty much every day. It's annoying, and if you forget you stashed something by the time you come back, it gets ugly fast."

"And when a long-running interruption gets interrupted again by something even more urgent, and you end up with multiple stashes piled up, it becomes impossible to remember which stash holds which piece of work. It's not at all unusual to hit a conflict while restoring one and end up tossing your changes in frustration."

"Ugh, just imagining it gives me a stomachache..."

"Then there's the cost of naming things in Git, which really adds up. To start new work, you first have to create a branch and give it a meaningful name. To commit, you need a commit message, and once you've settled on one, changing it later isn't impossible, but it requires some tedious steps. Even for something you're just experimenting with and might throw away in a minute, you have to come up with some kind of name before you can start. That adds a surprising amount of cognitive overhead."

"The cost of naming things, huh... I hadn't really thought about it, but now that you mention it, I do get stuck every time trying to figure out what to call things, and I just freeze up."

Rewriting History Is Complex and Dangerous

"Compared to the days when humans wrote code carefully by hand," Yukina went on, "the cost of having an AI write code is dramatically lower, so it's much more common now to want to go back and tweak something you already recorded. But rewriting history in Git is a daunting task."

"I know what you mean. When I really have to edit history, I end up googling around or asking a chat AI, and I do it while holding my breath the whole time."

"In Git, the single concept of 'rewriting history' is spread across multiple commands: `git commit --amend`, `git rebase -i`, `git reset`. And the scope of each one can change dramatically depending on which options you pass. Building a clear mental model is hard. I doubt many people have all of these memorized and can confidently pick the right one for each situation."

"Tell me about it."

"`git rebase -i` in particular has this peculiar UI where you edit a list of commits in a text editor, and any mistake you make in that editor

flows straight into your history. Delete a line and that commit disappears; change `pick` to `drop` and it disappears too. The moment you save and close the editor, the process starts running, so the despair when you realize you edited something wrong is something else.

"And if a conflict comes up partway through, you can end up being asked to resolve conflict after conflict as commits are applied one at a time. By the end you've lost track of which commit you're even on, and it tends to come down to a choice between `git rebase --abort` to start over from scratch, or hammering `git rebase --continue` in confusion."

"Right, exactly. I'm so afraid of that, I end up going as far as I can without committing, just to avoid having to split or fix things up later. Which is, of course, exactly how the accident from earlier happens, or how I end up submitting a PR with one massive, kitchen-sink commit."

Commands with Heavy Side Effects That Are Hard to Undo

"Git has quite a few destructive commands," Yukina said, "that you can't easily take back once you've run them. For example, `git reset --hard`, `git clean -fd`, and `git restore` will mercilessly wipe out changes you haven't committed yet. The fact that one mistyped command can erase a working tree's worth of effort you've painstakingly built up is nothing short of terrifying."

"Gemini CLI once helpfully made changes I never asked for, and when I told it to 'put things back the way they were,' it ran `git restore` and wiped out everything I'd been working on before that, too. It apologized profusely afterward, but by then the damage was done."

"With Claude Code you can use Permissions settings to block specific Git commands, but it's easy to misconfigure those settings. The risk of letting an AI loose on Git, a tool where one command can irreversibly destroy the work you've piled up, is high.

"Also, some heavy Git users might say, 'Just use `git reflog`, you can undo anything,' but reflog can't actually restore everything. Reading reflog is genuinely hard, and safely restoring to an intended state takes a deep understanding of Git's internals. The number of engineers who can calmly operate reflog while panicking and get back to exactly the state they wanted is, again, vanishingly small."

Summary: Git's Design Philosophy Versus the AI Era

"Hmm," Kanae said. "Listening to all this, Git is starting to feel like one of those tools humanity just wasn't ready for. Why is it designed this way?"

"It's because Git was originally designed around the Linux kernel development model: **humans write code by hand, organize their changes, group them into review-ready units, and share them carefully**. For a workflow where humans hand changes off to other humans with care, it's a phenomenal tool, and it has more than proven itself over the years. But in agentic development, the situation looks quite different:"

- Rapid trial and error is constant.
- Huge amounts of code change all at once.
- The context of your work switches frequently.
- The need to tidy up history after the fact comes up more often.

"Under conditions like these, the very traits that made Git work so well end up dragging your development down. That's the underlying reason Git's pain points have stood out so much in the AI era."

"Mmhm."

"What agentic coding really needs is a tool where you can first make a mess and then safely tidy it up afterward. And it's right there, on exactly that point, that a version control system built on a different design philosophy started drawing attention."

"Oh—that's Jujutsu, isn't it!"

1-2. The Jujutsu Features That Stand Out in the AI Era

"I'm not saying I don't trust you, Yukina, but is Jujutsu's popularity really climbing that fast right now? I'd never even heard of a VCS like that until you brought it up."

"Well, as of mid-2026, it's still at the stage where a small group of developers who keep a close eye on new tooling are passionately backing

it. In terms of chasm theory[4], it's spreading among early adopters but hasn't crossed the chasm yet.

"I'd love to point you at some hard data, but there really isn't a good objective metric for VCS popularity. Git has been so dominant for so long that developer surveys don't even bother asking 'what VCS do you use?' It's not exactly a fair comparison, but let's look at the GitHub star history for the official repositories."

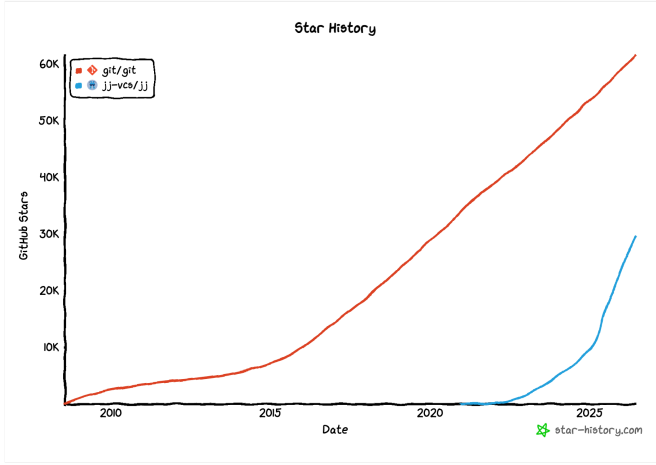

Figure 2: GitHub star history for Git and Jujutsu (GitHub Star History, June 2026)

"Whoa! The Jujutsu curve is going up like a hockey stick. It really took off from mid-2025. At this rate it looks like the day it overtakes Git might not be that far off. But what do you mean by 'not exactly fair'?"

"Git isn't actually developed on GitHub. There's just a mirror repository there. The official one is hosted on kernel.org, the same place as the Linux kernel[5]. Jujutsu, on the other hand, has its official repository on GitHub."

[4] A marketing theory grounded in the Technology Adoption Lifecycle, a sociological model of how new products and innovations spread. It divides the diffusion process into five customer segments, and identifies the "chasm," a deep gap between the early market and the mainstream market, as the most critical hurdle to cross.

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

"Well, for most users GitHub is pretty much everything, so I'd say it's a reliable barometer of popularity. So I get that Jujutsu is rapidly gaining momentum. What's driving the surge?"

"Kanae, you just said it started taking off in mid-2025. Do you remember what happened around then?"

"...Hmm, what was it again?"

"Claude Code was officially released in May. Up to that point, AI coding tools were mostly IDE-based, focused on simple code changes through suggestions and chat. Claude Code is a terminal-based agentic AI coding tool: give it instructions and it autonomously generates high-quality code in one go.

"With Claude Code on the scene, the world tipped over into a state where AI was writing far more code than humans. Not to be left behind, Codex CLI and Gemini CLI followed suit, Cursor pivoted to an agentic model, and the trend toward agentic development became decisive."

"Right. It feels like ancient history now, but it really wasn't that long ago we were writing every line by hand."

"Jujutsu's rise looks like it's being pulled along by the spread of AI coding agents like Claude Code and Codex. I myself decided to give Jujutsu a try after reading a run of articles around that time that introduced Jujutsu in the context of agentic coding. Some of the more widely read ones include:"

- 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]

"Huh. When experts are telling you Jujutsu and AI are this good a match, you can't help but want to give it a shot. But here's something I don't get. Jujutsu existed before agentic coding took off, right? It wasn't designed for it, so why does it fit so well?"

---

[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

"Jujutsu's development started in 2019, and at that point hardly anyone could have predicted a future where AI codes autonomously, so of course it wasn't designed with that in mind. According to the creator, Martin von Zweigbergk, his goals were **reducing the cognitive load on humans** and **doing away with Git's complex mental model**[10].

"AI agents iterate at high speed without fear of mistakes, and working alongside them is enormously taxing for the human brain. So Jujutsu's design, which set out to reduce that cognitive load for humans, ended up as a natural fit for this situation as a side effect."

"Hmm, I see."

"So let's walk through the Jujutsu features that are getting attention in the AI era, one by one."

The Working Copy Is Committed Automatically

"Unlike Git, Jujutsu has no staging area," Yukina said. "There are only two states: the **working copy** (which corresponds to Git's working tree) and the **repository**. And here's Jujutsu's biggest defining feature: changes to files in the working copy are automatically committed. Without the user running anything like `git commit`, the working copy gets snapshotted and recorded in the repository."

"What? You don't have to add, and you don't even have to commit? How is that actually pulled off? Is there a daemon[11] watching the file state or something?"

"No. Jujutsu's command system uses `jj` as the main command, with subcommands such as `jj log` and `jj diff`, and a snapshot is taken at the moment a `jj` command runs. If there's a diff from the previous snapshot, that diff is committed."

---

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

[11] A program that runs continuously in the background on a UNIX or Linux system (or similar OS), automatically handling specific tasks such as serving web requests or managing networking.

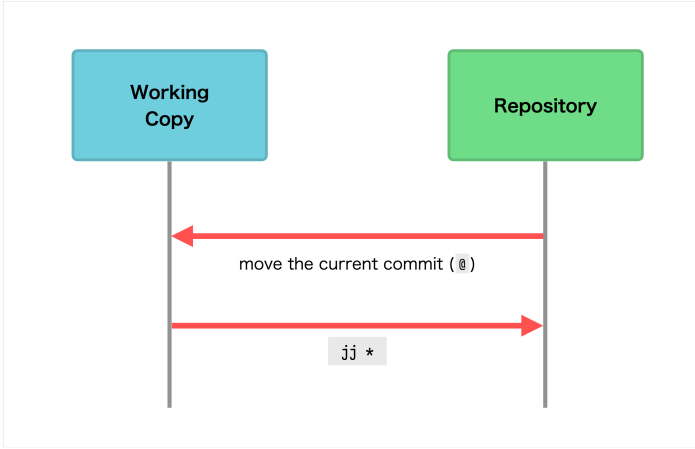

Figure 3: Jujutsu's two-state model

"Huh. It's nice that mistakes from forgetting to add or forgetting to commit just disappear. But if you don't control when commits happen, doesn't your history end up chaotic?"

"I'll explain in more detail later when you actually try it, but Jujutsu has a concept called a **change** that wraps commits as a single unit. To put it roughly enough for now: a change is like a container that holds multiple commits in chronological order. Jujutsu's history is usually displayed by change, and seeing the individual commits inside one takes an extra step. Naming, adjusting granularity, and reordering are all done at the change level. So you don't have to worry about commits piling up on their own and making your history messy."

"Ah, I see how they handle that."

"The official documentation captures this property with the term **working-copy-as-a-commit**. Thanks to its simple two-state model and this property, Jujutsu doesn't need anything that corresponds to `git add`, `git commit`, or `git stash`. Compared to traditional VCSs, the cognitive load around managing file state is dramatically lower."

Every Operation Is Undoable

"When you're using Git day to day," Yukina said, "don't you ever have those moments of 'ugh, I really want to undo that last command'?"

"All the time. Honestly, it makes me mad that there isn't an undo like in an editor."

"Jujutsu has exactly that: **universal undo**. Anything that touches a remote can't be perfectly rewound, of course, but every local operation can be undone and redone."

"That's amazing! That alone makes me want to switch."

"In addition to the usual commit log, Jujutsu has an **operation log**. It's a record of every `jj` command you've run, linked to working-copy snapshots. So beyond simple undo, you can rewind to the state at the moment any specific command was run, or strike a single operation out of history as if it never happened."

"In Git I'm always gingerly running rebase and other history-editing commands, but with undo I can take a 'try it, and if it doesn't work, undo and try again' approach. That's reassuring."

"Right. Once you're freed from the fear of doing something irreversible, the bar for experimentation drops dramatically. In Git, getting an AI agent to run history-editing commands is too scary to seriously try, but with Jujutsu it's a non-issue. The sense of psychological safety is on a completely different level. Once you've experienced it, you can't go back."

Conflicts Are Treated as Just Another State

"What's scary in Git," Yukina said, "is conflicts during merge, rebase, or cherry-pick. When a conflict happens in Git, other operations are strictly blocked, and you're forced to stop working until you resolve it."

"It is scary. Whenever I pull in the latest main branch before opening a PR, I'm always praying no conflicts come up."

"Jujutsu, on the other hand, treats **a conflict as just another state**. When one happens, it gets committed as-is and doesn't block any operations. For example, if a conflict comes up during a rebase, the rebase finishes anyway, carrying the conflict along with it. The official documentation calls this **first-class conflicts**."

"Wait, that's a good thing? Isn't a conflict something you have to resolve right away? Leaving it sitting there just nags at me."

"That 'a conflict has to be resolved immediately' thing is a Git-brained assumption. The reason Git treats an unresolved conflict as a

blocking 'exception' isn't a technical necessity; it's an implementation constraint. Let me give you a few concrete cases where first-class conflicts feel great:"

- You're rebasing a branch, and resolving a conflict requires checking the implementation intent with Alice, the teammate who owns that code. But she's on vacation and won't be back until tomorrow. In Git, you'd have to give up on the rebase until then. In Jujutsu, you can finish the rebase for now and deal with the resolution later, or move forward with a quick-and-dirty patch.
- You want to let an AI handle resolving multiple conflicts that came up during a merge. In Git you can't commit unless every conflict is resolved, so if it gets something wrong, restoring just one specific conflict and re-verifying is awkward. In Jujutsu you can pinpoint and restore a single conflict and redo the resolution as many times as needed.

"Ah, I see. That actually sounds useful. Having an AI resolve conflicts would have been unthinkable in Git. But with Jujutsu, you can have it resolve them one at a time, and if it gets one wrong, just undo and try again. That makes it much easier to hand the work over without flinching."

"On top of that, in Jujutsu, when you edit a past commit, the changes to that file are automatically propagated to its descendant commits. So even when you resolve a conflict, you can just move to the commit where it happened and edit the file there, without having to manually trigger a rebase. Since deferred resolution costs almost nothing later, the psychological burden is small."

Orthogonal, Largely Stateless Commands

"One thing I notice using Jujutsu," Yukina said, "is that the commands are very easy to remember. For commands whose target is history itself, the command maps one-to-one onto the verb for what you want to do. For example, discarding history is `jj abandon`, splitting is `jj split`, squashing is `jj squash`. There's nothing like `git checkout` or `git reset` where options or arguments fundamentally change what the command does. The operation you want maps directly to a command, so you never have to wonder 'which command was that again?'"

"Mmhm, less command overloading is great."

"Commands whose target isn't history itself are also organized into a clear logical hierarchy. For instance, in Git, branch operations are scattered: creating one is `git branch <branch-name>` or `git checkout -b <branch-name>` or `git switch -c <branch-name>`, listing is `git branch`, deletion is `git branch -d <branch-name>`. It's hardly consistent. In Jujutsu it's `jj bookmark create`, `jj bookmark list`, `jj bookmark delete`. 'What you're acting on, and what you're doing' reads straight out of the command structure."

"You're right, I could never remember which Git branch command did what. I was looking it up every time. But if Jujutsu's commands work like that, even I might be able to memorize them."

"On top of that, Jujutsu shares basic option and target conventions across many commands. The option for specifying a particular point in the history graph is `-r/--revision`, and `-f/--from`, `-t/--into`, and `-o/--onto` all have the same meaning across history-editing commands. So you can transfer knowledge from one command to another easily."

"I see. With Git it kind of feels like you have to memorize a different spell for each command."

"In software design, when learning about one element lets you extrapolate to other elements, we call it 'highly orthogonal.' Jujutsu's command system really is highly orthogonal. After using it a few times, you find yourself typing 'this probably looks like…' and getting it right."

"I've never heard the term 'orthogonal' before, but I guess it basically means simple and intuitive."

"Another point worth making is that Jujutsu's commands have very little state dependence. Git's commands always start from HEAD[12], and they're influenced by the state of the working tree and staging area, so running the same command in different situations can give very different results. In Jujutsu, there's no staging area and changes are automatically committed, so the working copy is just one node among many on the history graph. And since many commands can take a specific point in the history graph as an argument, you can operate directly on those points regardless of where the working copy is."

[12] In Git, a pointer that indicates where you're currently working. Normally it points to the latest commit on the current branch.

"Hmm, like what?"

"`jj squash` is the command that squashes history. If you explicitly specify the source and destination with `jj squash -f <from-id> -t <to-id>`, the result is the same no matter where the working copy sits on the graph. If you tried to do the same thing in Git, you'd have to `checkout` to the target location first."

"Ah, that makes sense. But Jujutsu's approach of always specifying the target with an option has the downside of more keystrokes, doesn't it?"

"That's why many commands have default values when an option is omitted. Running `jj squash` with no arguments squashes the working copy into its parent. You can also specify just `-f` or just `-t`."

"I see. By the way, what's so nice about having low state dependence?"

"You can do whatever operation you want from wherever you are. There's no need to shuffle the working copy around. You're also freed from the cognitive burden of having to precisely understand the current state before each operation, and explicitly specifying the target reduces operational mistakes. Once you're used to Jujutsu, orthogonal commands with low state dependence start to feel so natural that you don't notice them anymore. But every now and then, when you have to go back to Git for something, you really feel how much more convenient Jujutsu is."

Summary: Jujutsu Is a VCS Where You Start Rough and Clean It Up Later

"Putting all these features together," Yukina said, "the philosophy underneath Jujutsu starts to show. The style is: 'just start rough, experiment freely, and clean it up later.'"

- The working copy is automatically committed continuously, so you don't have to think about whether your edits are saved.
- You don't have to aim for perfect change granularity or come up with the right names from the start. When you need to, you can split, squash, and reorder changes, and rename them to match how things actually turned out.
- Every operation is undoable, so you can experiment without fear of mistakes.
- Conflicts aren't blocking exceptions; they're a state a commit can hold. You can defer resolution and keep working.

- The orthogonal, largely-stateless commands let you reshape history on the spot, whenever inspiration strikes.

"I see... It's the complete opposite of Git, where you have to nail everything down up front and any later change is a hassle."

"This Jujutsu style has a lot in common with agile development, which assumes humans make mistakes and aims to raise quality through repeated small adjustments."

"Ah, true."

"AI agents make mistakes at least as much as humans do, and they iterate and generate code at speeds humans can't match. Jujutsu's affinity for AI agents isn't a coincidence. It's the natural consequence of relentlessly designing for the human brain."

"Kind to humans and AI... Now I'm seriously interested. I want to try it as soon as possible!"

Column: How Git Revolutionized Version Control

Chapter 1 built its argument from the angle that Git's design has fallen out of step with the times. But it would hardly be fair to stop there without also acknowledging the enormous role Git has played in the history of version control. One of the motivations behind Jujutsu's development was to solve Git's pain points. Turned around, that simply means Jujutsu is a product built on the shoulders of a great predecessor.

Git began to gain traction in the industry in the early 2010s. Before that, version control generally meant a centralized product, and Subversion[13] in particular was widely used. Because everyone shared a single central repository, many everyday operations depended on the server. Commits and log browsing, in particular, required talking to the central repository, and offline work was severely limited. Getting any substantial work done offline was basically impractical, and since a commit was instantly shared with the rest of the team, you couldn't record history in fine-grained steps while your work was still half-finished. So it wasn't unusual to see workflows that stripped a VCS of half its value: piling up a heap of changes locally and only making a single commit once everything was finally ready to release, or lumping a whole day's work into one commit at quitting time.

The spread of Git changed all of that overnight. A nearly complete copy of the history always lives on the user's own machine, and most everyday operations complete locally. Because you commit against your own local repository, you can shape your history at whatever granularity you like, entirely at your own discretion. And even if the server goes down or you're working offline, you can keep going.

In Git, the central repository is conceptually no more than one synchronization point for sharing work among team members; each

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

member holds a repository with a nearly complete history of its own. Even if the central repository is corrupted, it can be restored from a member's repository. In Subversion the files on a member's machine were subordinate to the central repository, but in Git each member holds a repository that stands as a peer of the central repository. That was a shift that changed the power structure of development—and how the people in it thought.

Git was also blazingly fast at all kinds of operations compared to the existing VCSs of its day. Part of that is because most operations complete locally, but a big factor is that its design target was the Linux kernel, so from the design stage it set performance requirements around things like low network transfer volume and the speed of log browsing and merges. Where conventional VCSs recorded the successive diffs of a file from its initial state, Git records a snapshot of the entire file tree at commit time. To pull out a particular version, it simply reads the pointer for the corresponding commit directly, which makes it extremely fast.

Git also changed what a branch means. In Subversion, a branch was just a copy of the whole project created in a separate directory, so branching carried significant overhead and friction. On top of that, its data model tracked merges poorly, which made unnecessary conflicts likely, and many teams disliked that enough to avoid branches altogether. In Git, by contrast, the history itself is structured as a DAG[14], and a branch is nothing more than a pointer to a particular commit—a remarkably elegant design[15]. Git branches are lightweight to work with, produce fewer conflicts on merge, and can be created purely locally, all of which dramatically lowered their psychological cost. Development styles like creating a feature branch for each small feature, working on several branches in parallel, or using a local branch as a personal sandbox only became possible once Git took hold.

[14] Short for Directed Acyclic Graph: a graph structure in which nodes connected by directed edges never form a cycle.

[15] Explained in "2-4-2. The Difference Between Branches and Bookmarks".

Git, you could say, changed the very status of version control: from something you "had to use" under the control of a company or organization, into something you "actively wanted to master" to raise your own productivity. The arrival of Git democratized version control itself and, as I see it, placed history and its power equally into the hands of every individual developer.

Chapter 2. Let's Try Jujutsu

2-1. Setting Up Jujutsu

2-1-1. Installing Jujutsu

"From here on, I want you to learn Jujutsu by actually using it," Yukina said. "But first things first, we need to get it installed. Since Jujutsu is written in Rust, the official docs lead with a Cargo-based procedure, but unless you're a die-hard Rust fan, you're better off avoiding that route. The build takes a while, and keeping it updated gets fiddly."

"Right," Kanae said. "I've got no plans to touch Rust anytime soon, so I'd rather not start by setting up a whole Rust environment just for this."

"Luckily, we're both on Macs, so we can install it easily with Homebrew, like this:"

```
$ brew install jj
```

"This next part isn't required, but adding a bit of shell configuration makes running commands more convenient. Depending on which shell you use, add one of these lines to your config file:"

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

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

"What does this actually do?"

"When you run `jj` with the `COMPLETE=zsh` environment variable set, it outputs a shell completion script for zsh. By having zsh load that script on startup, the latest completion definitions for the `jj` command get generated and applied dynamically. So when you type part of a subcommand or option and press `Tab`, it'll show you a list of candidates or autocomplete the command for you."

"Oh, that sounds handy. I'll add it to my config too. By the way, I've got a Windows machine at home. How would I install it there?"

"In a WSL[16] environment, Homebrew is the way to go, same as on a Mac. For a native environment, the package is registered under WinGet as `jj-vcs.jj`, so install that. Just be aware there are some caveats around line endings and symbolic links, so read through the relevant page in the official docs[17] carefully."

"Got it."

"One more thing, not directly related to Jujutsu: if you haven't installed the GitHub CLI (`gh`)[18] yet, now's a good time. Letting an AI agent run the `gh` command alongside `jj` makes your workflow quite a bit more efficient."

2-1-2. Initial Configuration

"Jujutsu has a huge number of configurable settings," Yukina went on, "but only a handful matter at the start. First, let's register your username and email address."

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

"`jj config set`[19] is the command for writing values to your config file. With `--user`, it targets user-level settings; with `--repo`, it targets settings scoped to that one repository."

"What happens if you start using it without setting these?"

"Your author information in the history will be missing, and you won't be able to push to a remote. You would then have to set your

---

[16] Short for Windows Subsystem for Linux. A Microsoft feature for running Linux environments on Windows. Modern WSL uses a lightweight virtual machine with a real Linux kernel, letting you install and use distributions such as Ubuntu or 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

user info after the fact and run `jj metaedit --update-author <TARGET>`[20] to update the author information in your history."

"That sounds like a hassle. I'll make sure not to forget that on a fresh install."

"Let's also change the editor setting. Out of the box, the editor Jujutsu launches when it needs one defaults to nano, which most people aren't familiar with[21]."

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

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

"What's that `--wait` option you're passing when launching VS Code?"

"When Jujutsu launches an editor, it needs to detect when the editor exits so it can pick up what you wrote. If you open a GUI editor from the terminal, it normally launches as a separate process, and your edits never make it back to Jujutsu. Adding `--wait` makes VS Code keep the terminal process blocked until you close the editor tab, so Jujutsu can recognize that you're done and receive your edits correctly."

"Ah, I see, that makes sense."

"By the way, if you can't launch VS Code by typing `code` in the terminal, open the command palette with `⌘` + `Shift` + `p`, type 'shell', and run 'Shell Command: Install `code` command in PATH' from the results. That'll get it working.

"Now, with that set up, let's run `jj config edit --user`[22]. The editor you specified earlier should open and display something like this:"

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

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

[21] The behavior on macOS and Ubuntu-based Linux when the `$EDITOR` environment variable is unset. On Windows, Notepad launches instead.

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

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

[ui]
editor = "vim"
```

"And where does this file actually live? You can find that out with `jj config` too."

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

"So editing this file directly will apply the settings as well, right?"

"Of course. So if you store this file in a shared directory like Dropbox and create a symlink to it, you can share your Jujutsu user settings across multiple machines."

2-2. A Hands-On Tour of Jujutsu

2-2-1. Initializing a Repository

"All right, let's start version-controlling some files with Jujutsu," Yukina said. "A React app makes for good teaching material, I think. We'll use Node.js as the runtime and pnpm for package management, so if you don't have those yet, set them up first. And to install them, we'll use mise[23]."

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

"Now let's create the React project, initialize it, and put it under Jujutsu's management."

[23] A package manager for installing programming-language runtimes and various CLI tools while managing multiple versions of each. It also doubles as a task runner and an environment-variable manager.
https://mise.jdx.dev/

```
$ 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
```

"The command to initialize is `jj git init`[24], not `jj init`?"

"Jujutsu uses a pluggable storage architecture, and it can use Git as its backend storage. So this command initializes a Git repository with Jujutsu layered on top of it. Some older articles suggest using the `--colocate` option to let the Jujutsu and Git repositories coexist in the same directory, but these days that behavior is the default, so you don't need it."

"What if I've got a repository that's already managed with Git and I want to start using Jujutsu on it later?"

"Same command: `jj git init` works. The past commit history you created with Git gets imported as Jujutsu history as-is."

2-2-2. Checking the Repository's State

"Next, let's check the current state of this repository," Yukina said.

---

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

Chapter 3. Jujutsu × AI Workflow in Practice

3-1. Coordinating AI Agents with Jujutsu

3-1-1. Getting AI Agents to Use Jujutsu

"Lately I've started letting AI handle Git for me too," Kanae said, "like, I'll just tell it 'okay, commit what we've got so far and push it.' Can you actually hand Jujutsu operations off to an AI like that?"

"Of course you can," Yukina replied. "Jujutsu has undo, and it's easy to rewind to any midpoint in a stretch of work the AI just carried out. Honestly, you can hand things off more boldly than you can with Git."

"Huh, interesting. So you've been pairing Jujutsu with agentic coding this whole time, right? It really doesn't seem like just telling an AI agent 'use Jujutsu' would do the trick—what kinds of things have you been doing to make it actually work? That know-how would be incredibly valuable to me, so I'd love it if you'd share!"

"Sure. The thing is, AI agents differ quite a bit in their features and quirks, and new capabilities are being added at a dizzying pace, so best practices are still hard to pin down. Among coding agents, Claude Code has the biggest market share and I'm a heavy user, so I'll focus on what I think is the best approach for it as of June 2026. I'll also bring up Codex CLI where it's relevant."

"Yes, please. Those are basically the two I use day to day, too."

"All right. Let's start with how to steer an AI agent toward Jujutsu instead of Git as its VCS. For Claude Code, I think the best move is to drive things primarily through **rules**[54]."

"Not `CLAUDE.md`?"

"That too. But in `CLAUDE.md` I only put short, top-priority items I want followed no matter what. The main material lives in the rules."

"Sounds complicated. Can't you just consolidate it into one?"

---

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

"The limitation of `CLAUDE.md` comes from the fact that it's fundamentally a place for conveying project context. If you write 'this project uses Jujutsu' in there, the agent processes it as just one more piece of information, and as Claude Code gets absorbed in the task, that fact ends up buried under everything else and its priority quietly drops. The longer your `CLAUDE.md` is, the worse this gets."

"Yeah, it's honestly not unusual for `CLAUDE.md` or `AGENTS.md` to get ignored once you've been working for a while."

"AI coding agents are trained on data dominated by Git, so the Git workflow is baked deep into their defaults. Want to check your changes? `git diff`. Want to commit? `git commit`. Want a branch? `git checkout -b`. These aren't just knowledge—they're reflexive behavioral patterns. Rules are a much better fit for correcting that behavior."

"Why is that?"

"Because rules are designed as behavioral constraints that apply to every interaction, all the time. They get injected into every exchange in a session, so no matter what context Claude Code is in when it reaches for a VCS command, the directive 'don't use `git`, use `jj`' is always there to intervene."

"Ah, that clicks now. What about skills, though? If you search 'jujutsu' on skills.sh[55], a lot of skills come up. The sheer number suggests they're getting reasonable use."

"I'd bet they don't actually work very well. Skills are references that an AI agent goes and reads when *it* decides 'this looks relevant'—they aren't always-on behavioral constraints. A lot of the time you have to point at them explicitly. VCS operations come up implicitly in the middle of coding work, so there's a good chance `git` ends up running without the skill reference ever being triggered."

"Hmm. What if you used skills together with rules?"

"I wouldn't recommend that either. The instructions can conflict, and having both in context tends to confuse the agent. The terminology mismatch is another concern. I explained earlier that the same word

[55] An open directory and ranking site released by Vercel in January 2026 where you can search and manage skills for AI agents.
https://skills.sh/

'commit' means different things in Git and Jujutsu. Most of those skills barely account for that kind of thing, which becomes another source of confusion."

"Right."

"And as a final safeguard, I want to throw the *permissions* settings into the mix, too. I'll cover that separately later.

"Now, what do you actually put in that rules file? Roughly the following:"

1. A ban on using `git` commands
2. Differences in terminology and mental models between Git and Jujutsu
3. Always pass `--git` to commands that output a diff
4. Outside of confirming results after a file change, pass `--ignore-working-copy` to read-only commands
5. How to read conflict markers
6. Jujutsu workflows organized by use case

"Item 1 doesn't really need explanation," Yukina continued. "For item 2, I struggled a lot to phrase things in a way that actually gets the AI to behave the way I want. In the end, I gave up on technical accuracy in the wording, made a side-by-side mapping of terms, and just wrote out what not to do. So you'll see things like 'commit = change' or 'HEAD = working copy' in there, which aren't strictly correct. I also try to avoid using the word 'commit' in Jujutsu contexts at all and use 'revision' instead."

"So you basically gave up on getting the AI to truly understand Jujutsu and prioritized 'it works for now.' That's the kind of conclusion you only reach after actually battle-testing this."

"I tried not to demand too much change from the AI agent. What I focused on was reducing the cognitive load for minds wired for Git. Forcing diff output into Git's format is part of the same idea. You can configure that in Jujutsu itself, but then the output gets harder for *humans* to read, so I have the agent pass `--git` at the command line instead."

"What's that `--ignore-working-copy` in item 4?"

"Normally, when you run `jj` while there's a diff between the working copy and the history, Jujutsu takes a snapshot. That option disables that behavior.

"While you're waiting for the AI to finish a task, you often pop over to another pane and do other work, right? If your own `jj` command and the AI's `jj` command happen to overlap, one of them can snapshot and advance the history before the other has finished. Then the working-copy commit that one process thinks it's looking at ends up older than the working-copy commit the history actually reflects, and the mismatch raises an error. I add that option to head off that kind of situation as much as I can."

"Hmm, you have to think about that level of detail, too."

"Let me also touch on how to do this with Codex CLI. As of June 2026, Codex doesn't have a Rules feature equivalent to Claude Code's. There's something with the same name called *rules*, but it's for managing which commands can run outside the sandbox—not for handing the AI agent a behavioral charter in natural language."

"That's a problem. So what do you do?"

"The next-best option is to combine `AGENTS.md` with *skills*. But as I said, skills only fire when the AI decides they're highly relevant, so in this case you end up putting a lot of content into `AGENTS.md`."

"Hmm, getting the balance right is tricky. I didn't expect that you'd have to change your whole approach this much depending on the agent."

"Putting all of this together, the per-project setup files for your AI agents end up laid out like this[56]:"

Listing 1: Layout of AI agent configuration files

```
my-project/
  CLAUDE.md             ← top-priority directives for Claude Code
  AGENTS.md             ← directives Codex must follow
  .claude/
    rules/
      jujutsu-rules.md  ← the rules file in question
  .codex/
```

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

```
    skills/
      jujutsu/
        SKILL.md        ← the same content packaged as a skill
```

Listing 2: Opening of jujutsu-rules.md

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

This project uses **Jujutsu (jj)** for version control. AI agents must
strictly follow the rules below, and **using `git` commands is
generally prohibited** (except for the `jj git` subcommands and the
`gh` CLI).

---

## Important Notes for AI Agents

### Terminology Mapping

The meanings of terms differ between Jujutsu and Git. AI agents must
rigorously observe the distinctions below.

| Git term                 | Jujutsu term
|
| ------------------------ |
-------------------------------------------- |
| commit (as a noun)       | change
|
| branch                   | bookmark
|
| staging                  | (no such concept)
|
| unstaged / uncommitted   | (no such concept)
|
| HEAD                     | `@` (the working copy)
|
| stash                    | (no such concept; use `jj new` instead)
|
| `git add`                | (unnecessary; automatic snapshot)
|
| `git commit --amend`     | (unnecessary; changes to `@` are applied
automatically) |

### Fundamental Differences from Git

1. **There is no such thing as an "unsaved change"**: the moment you
```

```
save a file, it is automatically included in the current change. There
is no need to ask, "Do you want to include this change?"
2. **change and revision**:
   - **change**: A unit of work. It has a unique, immutable **change
ID**, while its contents are mutable.
   - **revision**: A snapshot of a change. A new revision is created
every time you edit, but the change ID never changes.
3. **Automatic rebase**: When you change a change's parent, its
descendant changes are rebased automatically. There is no need to
manage a rebase chain by hand.
4. **First-class conflicts**: Even when a conflict occurs, the
operation is not interrupted; it is recorded as a change that contains
the conflict. You can resolve it later.
5. **Operation log**: Every operation is recorded, and you can return
to any point with `jj undo` / `jj op restore`. You may operate without
fear of mistakes.
⋮
```

"Since your existing repositories are probably managed with Git, you'll want to scope this setup to individual repositories that use Jujutsu, rather than in your user-level config."

"Nice! I'm grabbing this exactly as is!"

3-1-2. Permissions Settings for `jj` Commands

"Claude Code's **permissions** feature[57] lets you govern the execution of any command with one of three kinds of rules: `allow` lets it run without manual approval, `ask` prompts for confirmation, and `deny` prevents it from running," Yukina said. "In the rules we just looked at, I wrote things like 'don't use `git` commands,' but with permissions you can actually enforce that ban. And for the individual `jj` commands, if you list out the ones you're fine with running unattended, you don't have to keep approving each one in a dialog. Development goes much more smoothly."

"Oh, you mean that thing where I keep picking 'Always allow' because it's annoying, and more entries keep piling up in `settings.local.json`?"

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

About the Authors

Yuka Ooka

Yuka Ooka began her career as a PHP and Ruby developer and product manager at several Japanese IT companies before going freelance in 2016. She took an early interest in React, still a niche technology in Japan at the time, and worked with a series of clients as a React specialist. Drawing on that experience, she wrote the *Riakuto!* (りあクト！) series. It struck a chord with readers in Japan, selling over 40,000 copies in total.

The author's desktop setup

Illustration: Megumi Kuroki

Manga artist and illustrator.
She has consistently created the cover illustrations for the *Riakuto!* and *Juju-chu!* series.