Episode Book

EPISODE #5643

Do Git Branches Actually Nest? AI Agents and Stacked PRs

PUBLISHED	30 September 2026 at 08:39 UTC
DURATION	23:03
LISTEN	myweirdprompts.com/episode/git-branch-nesting-ai-agents/
HOSTS	Corn, Herman, Hilbert
SCRIPT MODEL	deepseek-flash
VOICES	chatterbox-regular (chatterbox-local)
PIPELINE	V5.2
BOOK GENERATED	19 October 2026 at 17:38 UTC

SYNOPSIS

Daniel asks whether the branch-per-feature rule still makes sense when an AI agent's whole session lasts five minutes, and whether branches can be nested for multi-agent repositories. The answer starts with a correction: in Git, branches are just pointers to commits — there is no parent-child relationship anywhere in the object model. What people call nesting is either a topology (branch B created from the tip of branch A) or a naming convention (slashes in a branch name are purely cosmetic). From there, the conversation traces the real history — Atlassian's Feature Branch Workflow, Vincent Driessen's Gitflow and its often-quoted sub-teams passage, and Martin Fowler's Patterns for Managing Source Code Branches, including Collaboration Branches, Team Integration Branches, and the idea that frequency reduces difficulty. Kent Beck's "illusion of frozen time" gets its due. Then the modern answer: worktrees for isolation, stacked pull requests for integration, and GitHub's own rule — nest exactly as deep as the dependency chain runs, and no deeper.

TRANSCRIPT



Corn

In August, GitHub shipped a feature specifically designed for AI agents. And the reason they built it is that agents were dumping one thousand seven hundred and twenty-one line pull requests into repositories and nobody could review them.



Herman

One seven two one. That's a diff, not a code review.



Corn

That's a diff nobody reads. And Daniel's been circling this exact problem from a different angle. He's been away from heavily shared repositories long enough that the collaborative machinery of Git has started to feel like folklore he half remembers. Lately he's been using Claude through the app, and it insists on the old religion. A fresh branch for every iteration, then a pull request for the changes. Meanwhile, people working straight on the command line have drifted into looser habits.



Herman

Looser meaning what, exactly?



Corn

Committing to main, mostly. Working in place. Treating the branch like a formality.



Corn

So Daniel's asking whether the branch-per-feature rule still makes sense when the whole agent session lasts five minutes. He wants to know how deeply branches can actually be nested inside one another. And he wants to know what a multi-contributor, multi-agent repository looks like when you divide it into branches and sub-branches for maximum effectiveness with minimum bureaucracy.



Herman

That's three questions.



Corn

There's a fourth underneath them. He says the best results come when we build on top of long-standing human development practice rather than driving over it. He wants that tested, not assumed.



Herman

Good. Because I think the honest answer involves telling him the premise is slightly wrong.



Corn

In what way?



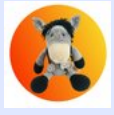
Herman

Branches don't nest. Not in Git. There is no such thing.



Corn

Start there.



Herman

A branch in Git is a pointer to a commit. That's it. A file in the refs directory containing forty hex characters. There is no parent-child relationship between branches anywhere in the object model. You cannot ask Git what a branch's parent branch is, because the question doesn't parse.



Corn

So when people say they've nested a branch inside another branch...



Herman

They mean one of two things. Either a topology, where branch B was created from the tip of branch A, so B's history contains all of A's commits. That's real, and it matters, and it's what stacked pull requests are built on. Or they mean a naming convention. Feature slash auth slash login slash fix. That's a string with slashes in it. Git does not care. The slashes are cosmetic.



Corn

Purely cosmetic.



Herman

Entirely. You could name a branch feature slash auth slash login slash fix slash final slash v2 slash actually-final and Git would treat it identically to a branch called bob. Same object, same pointer, same everything. The slashes exist for humans and for scripts that parse them.



Corn

Which is a thread we're coming back to.



Herman

We are. But let's give Daniel his ceilings first, because he asked for depth and there are real numbers. GitHub's recommended maximum is five thousand branches per repository. A single reference name has a practical limit of about four kilobytes, and that's a filesystem constraint, not a Git one.



Corn

Four kilobytes of branch name.



Herman

Which is roughly four thousand characters of path. You will hit human patience long before you hit that. And then there's the one everybody cites wrongly. Git's core dot max tree depth defaults to four thousand ninety-six.



Corn

That sounds like a nesting limit.



Herman

It governs directory trees. Not branches. It's there to stop Git from blowing the stack on a pathologically deep folder structure. It has nothing to do with how many branches you can stack on top of each other. People see the number and assume it's the answer to Daniel's question. It isn't.



Corn

So the technical ceiling on nesting is effectively unbounded.



Herman

Effectively. Which means the binding constraint is somewhere else entirely. It's cognitive and operational. The deeper the chain, the more rebasing, the more merge conflicts, and the harder the review.



Corn

So if nesting isn't a Git feature, what were all these branching rules actually for? Because every one of them was a response to a specific way collaboration goes wrong.



Herman

Every single one. Start with the Feature Branch Workflow, the way Atlassian codified it. All feature development happens on a dedicated branch, never on main. The point is that main never contains broken code. That's a large advantage for continuous integration, because your CI pipeline can trust the trunk.



Corn

And it composes.



Herman

It composes with everything else. You can layer it under other workflows without changing the core rule. Then there's Gitflow, Vincent Driessen's model. Feature branches plus multiple long-lived primary branches. Master, develop, plus release, hotfix, and feature branches. That's the one everybody implemented in twenty twelve and half of them regretted.



Corn

But there's a detail in Driessen's original post that almost nobody quotes.



Herman

There is, and it's the closest thing in the canon to what Daniel is reaching for. He explicitly describes sub-teams. I'll quote it, because the wording matters. Each developer may also pull changes from other peers to form sub teams. Useful to work together with two or more developers on a big new feature, before pushing the work in progress to origin prematurely.



Corn

So Driessen's answer to parallel work on one feature was a collaboration branch.



Herman

A shared branch that several people push to, which then merges down. That's a collaboration pattern. It is not a nesting feature. Gitflow never had a concept of a branch inside a branch. It had a convention that humans agreed to follow.



Corn

Which is the pattern of this entire episode.



Herman

It is. And the person who wrote the modern canonical reference on all of this is Martin Fowler. Patterns for Managing Source Code Branches. If Daniel reads one thing after this, it should be that.



Corn

Give me the pattern names.



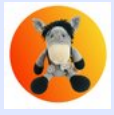
Herman

Mainline. Healthy Branch. Mainline Integration. Feature Branching. Continuous Integration. Pre-Integration Review. Release Branch. Hotfix Branch. And then the two that matter for the multi-agent case. Collaboration Branch and Team Integration Branch.



Corn

Define those two.



Herman

A Collaboration Branch is a shared branch where several developers work on the same feature before it's ready for mainline. A Team Integration Branch is a step further, a branch where a team's work is integrated and tested together before it goes to the trunk. Both exist because work in progress needs somewhere to live that isn't main.



Corn

And Fowler's overarching theme is that branches should be integrated frequently, with effort focused on a healthy mainline.



Herman

Which is the load-bearing insight for the whole AI question. He has a section called Frequency Reduces Difficulty. Frequent integration means smaller merges, less work, and critically, less risk and less uncertainty.



Corn

Less uncertainty is the interesting one.



Herman

It's the whole point. Fowler says the problem with big merges is not so much the work involved with them, it's the uncertainty of that work. And that uncertainty leads to integration fear.



Corn

Integration fear.



Herman

People stop merging because they're afraid of what the merge will do. So the branch gets longer, which makes the merge worse, which makes the fear worse. It's a ratchet.



Corn

And the failure mode he stresses isn't the textual conflict.



Herman

Semantic conflicts. Code merges cleanly and breaks at runtime. Git has no opinion about whether your function still does what its caller expects. It only knows whether the lines overlap.



Corn

Which is exactly what parallel agents produce.



Herman

Exactly what they produce. Two agents editing different files, no textual conflict, and the build is broken because one of them changed a signature the other one calls.



Corn

Fowler names a trade-off too.



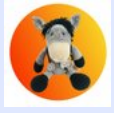
Herman

He does. Feature Branching implies a lower bound on change-set size, because a branch has to contain a cohesive feature. Continuous Integration decouples feature length from integration frequency. The rule of thumb there is that everyone commits to the mainline every day.



Corn

And then there's the Kent Beck quote.



Herman

Which is the philosophical statement of what branching buys. We give individuals the illusion of frozen time. But this is an illusion and eventually the price for it comes due. Who pays? When? How much?



Corn

Frozen time. That's what a branch is. A private universe where nobody else's changes exist yet.



Herman

And the bill arrives at merge time. Beck's asking who pays it. The answer in a team is usually whoever drew the short straw on the conflict resolution.



Corn

So the traditional workflows divide work into branches and sub-branches, and the sub-branch is a collaboration branch.



Herman

A collaboration branch or a team integration branch. And when does it work best? When the shared work is one feature that several people need to touch before it's coherent. It works badly when the sub-branch is just a place to park work that nobody wants to integrate.



Corn

Which is where the AI agents come in.



Herman

That's the human case. Now put an agent in the chair. One that finishes a task in five minutes and immediately wants to start the next one on top of it.



Corn

And here's where I want to push back on Daniel's framing slightly, because the research points somewhere cleaner than deep nesting.



Herman

Go on.



Corn

The industry is converging on two things. Worktrees for isolation, and stacked pull requests for integration. That's the actual answer to his question, and it's worth saying plainly.



Herman

Say it plainly then.



Corn

A worktree is a separate working directory attached to the same repository. Each agent gets its own. So agents and terminal tabs are isolated from each other, no conflicts, no stepping on each other's files. That's the isolation primitive everybody landed on.



Herman

And it's a real Git feature, not a convention. Git worktree add. It's been in core since two point five.



Corn

Then on the integration side, GitHub shipped native stacked pull requests. Public preview. Two or more PRs in the same repository where the bottom PR targets main, and each subsequent PR targets the branch of the PR below it.



Herman

So the stack is a dependency chain.



Corn

GitHub's own phrasing. Stacked branches form a dependency chain, where each branch builds on the one below it. And their example stack is four layers. Feature catalog data, then feature search API on top of it, then feature chat grounding, then feature grounded UI.



Herman

Catalog data feeds the search API, which feeds the grounding, which feeds the UI. Each layer depends on the one below.



Corn

And GitHub gives an operational rule for when to add a layer. If code in one layer depends on something in another, the dependency must be in the same branch or a lower one. Create a new branch when you start a different concern that depends on what you've built so far.



Herman

That's the answer to how deep you can nest. You nest exactly as deep as the dependency chain runs. And no deeper.



Corn

Which is a better answer than any number.



Herman

It's the right answer. Because it's derived from the thing that actually matters, which is what depends on what.



Corn

And GitHub is explicit that this is for agents. Their pitch is that stacks are fit for high volume development, often with AI agents. The line is: an agent completes one task, then starts the next task that builds on it. That sequence maps directly onto a stack.



Herman

That's a product team telling you what they built it for.



Corn

It is. And the blog post that frames it is from August, by Julia Muiruri. The title is Turn one giant AI-generated pull request to a reviewable stack. The example PR in that post was one thousand seven hundred twenty-one lines.



Herman

Which is the opening number of this episode.



Corn

It's the whole problem in one figure. The agent did the work correctly. It just did all of it at once, in one diff, and no human can review that. Muiruri's line is that for agents largely trained on how code has traditionally been written over the years, this pattern is their default way of shipping.



Herman

One giant PR. Because that's what the training data looks like.



Corn

Now the tooling, because Daniel will want it. It's a gh extension. You install github slash gh stack. And then there's a separate install to teach agents how stacks work, which is gh skill install, or npx skills add.



Herman

Teaching the agent the workflow, not just giving the human the command.



Corn

Both. Then the commands are gh stack add, gh stack push, gh stack submit, gh stack rebase, gh stack sync. Add a layer, push it, submit the stack as PRs, rebase the whole chain, sync it with main.



Herman

And the constraints, because there are real ones.



Corn

All branches must be in the same repository. Cross-fork stacks are not supported. And stacks don't work in GitHub Desktop.



Herman

The cross-fork one is the killer for open source.



Corn

It is. If you're contributing to somebody else's repository from your own fork, stacks are not available to you. That's a large chunk of the open source world excluded on day one.



Herman

And the rebase footgun.



Corn

GitHub's own blog warns about it. There's a one-click server-side cascading rebase. It resets the committer to whoever clicked the button, the resulting commits aren't signed, and if branch protection expects signed commits, that one click quietly breaks it.



Corn

It's doing all the work. You click rebase, the stack updates, everything looks green, and your branch protection is now silently not applying because the commits lost their signatures.



Herman

That's a bad afternoon for somebody.



Corn

Now, Claude specifically. Because Daniel's question is partly about what Claude enforces versus what CLI users have drifted into.



Herman

Claude Code drives git through the actual git and gh command line tools. That matters, because it means everything it does is reviewable and reversible with the tools you already have.



Corn

It creates branches, writes commit messages from the diff, opens PRs through gh, resolves conflicts. And it adds a co-authored-by trailer by default.



Herman

Which you can disable in the project instructions file.



Corn

So the branch-per-iteration behavior Daniel is seeing isn't some proprietary agent workflow. It's Claude using the standard tooling the standard way.



Herman

Which raises the question of whether the friction is worth it. And the honest answer is that it depends on whether a human reviewer exists downstream.



Corn

Split the case.



Herman

Tom Crawshaw wrote a guide for AI Architects on exactly this. His argument is that the friction is context dependent, and his framing is sharp. Git's entire branching model was designed to protect developers from each other. When you're the only developer in the codebase, that problem doesn't exist.



Corn

So for a solo operator, the branch is ceremony with no beneficiary.



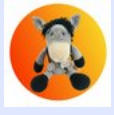
Herman

That's his position. His cautionary tale is a branch called chore slash scrub mcp secrets that sat unmerged long enough to accumulate one hundred ninety-one files. Accounting scripts, session logs, agent state.



Corn

One hundred ninety-one files on a chore branch.



Herman

Twenty minutes of cleanup, he says. And the point isn't that the branch was bad. It's that a branch nobody merges is a branch that accretes. It becomes a junk drawer.



Corn

And the counterpoint is the bug report.



Herman

Which is the sharpest thing in the whole research. Bug report seven nine four three two on the Claude Code repository, filed in July. Claude created a worktree and branch that silently tracked origin main.



Corn

Silently tracked.



Herman

The branch existed. It looked like a branch. But its upstream was set to origin main, so when Claude pushed, it pushed to main. Unreviewed. Direct to the trunk. Which triggered an auto-deploy to production.



Corn

That's the failure pattern the branch was supposed to prevent.



Herman

It's the exact failure pattern. The branch was there, and the branch did nothing, because the upstream was wrong. Environment was Claude Code two point one point two one five on Git two point five point one.



Corn

And the reporter's diagnosis is the interesting part.



Herman

The reporter's argument is that the root cause is a permission model organized around tool names, not around what an invocation actually does. So the system can ask permission for the push tool, and the push tool gets approved, and nobody asks whether this particular push is going to main.



Corn

The permission is for the verb, not the object.



Herman

For the verb, not the object. Which is a design problem, not a bug.



Corn

And that's the case for keeping the ceremony even when it feels clunky.



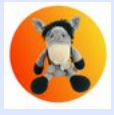
Herman

That's the case. The branch is cheap. The unreviewed production deploy is not.



Corn

But there's still something the research doesn't solve, and I want to end this segment on it rather than on a checklist.



Herman

Intent conflicts.



Corn

Worktrees solve isolation. Stacks solve integration order. Neither solves two agents making architecture changes that cannot both be true. One replaces a class while another extends it. Both diffs are clean. Both merge. The result is incoherent.



Herman

And there's a tool aimed at exactly that. Foremerge, on Hacker News this month. It describes itself as a git-like coordination layer that sits above git.



Corn

Above git, because git can't see the problem. Git sees text. Intent isn't text.



Herman

And then there's the comment from April that I keep thinking about. Somebody wrote that it's easier to pile on a lot of changes with AI assisted workflows, and then said: I've actually stopped pretending I can review everything in detail because it makes me a bottleneck.



Corn

Stopped pretending.



Herman

That's the sentence. Because the whole branch-and-PR apparatus assumes a reviewer who reads. If the reviewer has quietly given up, the apparatus is theater.



Corn

And there's a detail in all of this that Hilbert has been sitting on since we started.



Hilbert

The script split on the forward slash and took the second field as the ticket number.



Corn

Say that again.



Hilbert

I spent a stretch maintaining a repository where the branch names were the only documentation. No tickets, no changelog, just the names. And we had a release script that read them. Split on the slash, second field is the ticket, file the commit under that ticket's release bucket.



Herman

So the naming convention was load-bearing.



Hilbert

It was load-bearing because we made it load-bearing. Somebody created a branch called feature slash feature slash feature slash fix. Three slashes. The script took the second field, which was the word feature, and filed every commit on that branch under a release bucket named feature. For eleven days.



Corn

Eleven days.



Hilbert

Nobody looked at that bucket. It was a staging bucket. The first person to notice was a customer.



Herman

How did it get fixed?



Hilbert

We renamed the branch and moved the commits. Took an afternoon. The script got a check after that, counted the delimiters, refused anything with more than one slash. That check is still the right answer, by the way.



Corn

So the convention that Git doesn't enforce is a convention that rots.



Hilbert

Git will let you lie to it. That's the part your research doesn't capture. You said the slashes are cosmetic. They are, to Git. They're not cosmetic to the shell script somebody wrote in a hurry, and they're not cosmetic to the person reading the branch list at nine in the morning trying to work out what's in flight.



Herman

The structure is consumed downstream.



Hilbert

Everything downstream assumes the name means something. Git never promised it did. So the real ceiling on how deep you nest isn't five thousand branches or four kilobytes. It's how many layers of your own naming convention you can add before something that reads those names stops working.



Corn

And you don't find out until a customer tells you.



Hilbert

You don't find out until a customer tells you.



Corn

Which leaves the question we can't quite answer yet. If the honest answer to how deep branches can nest is as deep as the dependency chain and no deeper, what happens when agents can generate dependency chains faster than humans can review them?



Herman

Then the chain gets longer than the review capacity, and the review becomes a formality. Which is the thing the HN commenter already admitted to.



Corn

The second open question. There's no formal branch-per-iteration standard for AI agents. We looked. It's a Claude Code product default, not an industry practice.



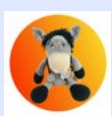
Herman

Which means it's a bet. Anthropic is betting that the ceremony is worth it. GitHub is betting that stacks plus worktrees is the shape. Neither has been settled by contact with teams at scale.



Corn

Fowler's frequency argument says small frequent integrations are better. But the ceremony around them was designed for human coordination, and nobody has solved the intent-conflict problem that parallel agents create.



Herman

The tension stays open. Build on the human practice, yes. But be honest that part of the human practice is a reviewer who reads, and that's the part under strain.



Corn

That's the one thing worth taking from this. The rules were never about the branches. They were about making sure somebody looked.



Herman

The branch is only as good as the person who reads it.



Corn

Thanks as always to Hilbert Flumingtop, who produces this show and who has now made all of us check our delimiter counts.



Herman

This has been My Weird Prompts.



Corn

The human-AI collaboration podcast. If you want to send us something, email us at show at my weird prompts dot com. We'll be back soon.