EPISODE 1352

[INTRODUCTION]

[00:00:00] KP: In a version control system, a monorepo is a management strategy in which all of your code is contained in one potentially large but complete repository. The monorepo is in stark contrast to an alternative approach in which software teams independently manage microservices with dedicated repos or deliver software as libraries to be imported in other projects. The monorepo strategy has been followed at noteworthy companies such as Google, Facebook and Microsoft.

Derrick Stolee is a software engineer working at GitHub. He joins us today to talk about strategies for monorepos and innovations to the way Git works in order to better support this style of repository.

[INTERVIEW]

[00:00:45] KP: Derrick, welcome to Software Engineering Daily.

[00:00:49] DS: Hey, thanks for having me.

[00:00:49] KP: Can you share a little bit about your journey as a software developer?

[00:00:55] DS: Oh, well, I guess I could say I started with as a little kid wanting to build video games was the initial reason I started getting into it. But very shortly into undergrad, even then I was really sure I was going to do it, but I realized that video game programming is not as fun as playing video games or designing video games. And I decided that I was going to broaden my view a bit. But also while I was doing undergrad, I fell in love with algorithms and data structures and started pursuing more my math side of my degree. And I ended up going to go to a Ph.D. program for math and computer science and working in a very theoretical stuff, computational complexity theory, and doing computational experiments for graph theory problems, and ended up being a faculty for a few years until I decided that that wasn't really working for me because I wasn't programming enough. Really, that was the most enjoyable part of my day was when I got to sit down and focus on a project about myself and really make progress on it. But the life of a faculty does not really promote that.

So I made the switch to working as a programmer. At that time I took a job at Microsoft. I am now at GitHub, but it's kind of due to that same job at Microsoft got reverse acquired. Like I was working for Nat Friedman at Microsoft when the acquisition happened, and then he's like, “let's bring all the people who work on Git into GitHub.” And so that entire period about almost six years now, I’ve been very focused on Git performance. First it was on the Azure repos server-side for really making sure we've got the scale ready to host things like the Windows and Office mono repos. And for the last four years I’ve been focusing on the core Git client and the open source project there and doing a little bit on the Git for Windows side, but also with the core Git for clients. So everybody who uses Git has a lot of the code that I’ve been working on to really make sure that the client performance is up to snuff. and really accelerating what you can do with it.

[00:02:55] KP: Well, forgive me for not appreciating some of the more advanced features of Git, but I feel like it has been stable and reliable for as long as I’ve been using it. What else is there to do?

[00:03:05] DS: Well, you are absolutely right that Git is a very mature product and the maintainer has really said that. And so a lot of what we do is make sure that it's stable. And the only thing we're doing is I like to change things under the hood. Not too many user visible features, but have it something like maybe there's a new data structure or something in the backend. So one of the ones that I’ve built was the commit graph file format. So we've got things like your commits are typically just plain text files that are then packed into PAC files, and that means that they're compressed text files that are placed in this really, really compact single file so you can load them up really quickly, except you're still needing to go and parse those plain text files to get information like what is the commit IDEs for its parents? Or what is its message? What are the dates for when it was committed? But that means that whenever you're doing a walk, like through Git log, or doing file history, you have to go and open up hundreds, or thousands, or even millions of commits and parse that plaintext data. And the commit graph file says, “Well, let's take the data that's really just a digraph and let's encode that in a structured form.” So instead of doing these constant hops to the back file, decompressing and parsing, we can just go to the structured format and hop around that in more of a random access way that allows you to move it more quickly.

And then you can do more interesting algorithmic stuff by adding more data into that that helps with walks. Like if you do gitlog--graph to see a nice visualization, if you don't have the commit graph, that can take multiple seconds just to show you the first few commits, because it has to walk the entire commit graph even to show you one answer. But with the commit graph file that you can get those things incrementally and you can do it really fast, tens of milliseconds.

[00:04:53] KP: Very cool. You'd mention the monorepo a couple times. For listeners who aren't familiar with that idea, what is a mono repo? Sorry. I’m saying it weird. I’m going to ask that again. For listeners who don't know what a mono repo is, what is a mono repo?

[00:05:09] DS: There are lots of things. I like to just think of it as a really big repo. Usually it's something where an entire business is basing all of their code in one place. So one of the famous examples is Google has a monorepo using a custom version control system that they have built. And it contains pretty much every code, all the code that's inside of Google entirely. Outside of some projects that are open source and in Git, but like anything that's built for Google is in this one codebase.

When we're thinking about monorepo on our side we're thinking like Windows as a business, Windows as a product. So the code that goes into building Microsoft Windows is in one repo. Microsoft Office is one business, one product. Like all the code for Microsoft Office is in one Git repository. And the biggest thing there is that it gives you this ability to say I can actually cut a tag and say this is the version that we've shipped to users. Even if there's lots of different components within there, you can say this is the thing that we've agreed upon as being the one version. And all the things that integrate together, we've said that. We don't have to go to multiple different places to pull these different versions of things and package them together. They're all right there.

[00:06:20] KP: So that makes sense. It's appealing and maybe helps us avoid whether you call it DLL, hell or whatever dependency hell what you're going to bump into. I guess you avoid that. But it seems like you take on some other baggage. Doesn't it make it harder for people to get that repo on to their machine and to merge PRs and that sort of thing?

[00:06:39] DS: Yeah. There's the aspect that now that we have everything in one place, yes, dependency problems are simpler, because as the code moves you see that the other things that depended on it are like – That you would break a build if you change a dependency incorrectly. And that's really immediate feedback. But you have this problem that because there's so much code and it's moving so quickly because thousands of engineers are contributing to it as part of their day job, that means that it's growing in scale to a part where Git really wasn't designed.

For many, many years, like the Linux kernel repository was the gold standard for a big Git repo, but it pales in comparison to the size of these repositories. So we need to do a lot of interesting things to make sure that Git works at that scale. And even by breaking some of the expectations of Git. One of the big things is Git was designed as a distributed version control system, which means that everybody has their own copy of the repository. And you share commits over the network, but you still have a full copy of everything you care about.

This is really good for the mailing list workflow of the Linux kernel, which is I'm just going to send you an email that shows you how to change the code. Not even a commit. Then you apply it wherever you want in your commits. That model doesn't work at monorepo scale, because I think if you really closely packed the Windows repository, at least even at the base, like the very first commit, it was 100 gigabytes for a single PAC file. And just to get started by downloading 100 gigabytes of data, that's really not a good way to get going. Developers are going to be very angry that it takes so long to just get started.

So we start by removing that requirement of needing absolutely everything. The start of that is now called Git's partial clone feature, which allows you to say, "Yes, give me the full commit history." And I'm going to start getting some of the Git data, but I'm not going to get every version of every file in the entire history. Let me get just things I'm going to check out, a tip for instance. And then if I come across a new checkout and I have some files that are missing, I'll go ask across the network to the server. And that'll help me get me the data I need for that operation only on an as needed basis. And that really reduces the amount of time to get started.

[00:08:54] KP: From the perspective of a developer who's comfortable on the command line or whatever tool they use to interact with Git, are they going to have to do anything different to work under those circumstances?

[00:09:06] DS: Well, the nice thing with Git partial clone is that it just means that you need to start your repository from scratch with Git clone and then the special option is – filter=blob:none. And it just gets you that initial clone to be starting fast.

From there, it just works like a normal Git repository. You do your operations. You just need to keep in mind that if you're going to go do a Git checkout on a new commit you never checked out before, you might need your network connection to be accessing. You might need your network to be working so you can access the remote. But that's not really atypical, right? If we're developing, we probably have a network connection because we want to be able to quickly go and search code in a different repository or quickly look up some documentation. So it's very rare that we're actually working without a network connection. So that's not going to require people to change their workflow too much.

[00:09:56] KP: Very true. On a greenfield project, is monorepo your default choice or are there a set of criteria you consider in deciding on the approach?

[00:10:07] DS: What I like to tell people is if it ships together, it merges together. So if you have things that are linked so much that there's a source dependency of any kind or you really want to make sure that you're testing that integration between components, then having them be in the same repository gives you that benefit, because you can do CI checks at pull request time or release time to make sure that all those components are working well, as opposed to worrying about which thing is in what state as you go along.

If you're in a greenfield project, that's probably most of what you care about, right? If you're dealing with small, you don't have the forces of scale to be forcing you to think about breaking it up into smaller pieces. I know that there're a lot of people who work in microservices and want to say, “Well, I’m going to start a new service. Let me create a new repository to do that.” But I find that that also makes means that you are needing to rebuild all the things you need in that repository to make just a microservice work from scratch, right? You need to build all your pipelines and all of your authentication things. And all that needs to be somehow pulled in to that repository. When if you just said, “Well, let me create a new directory in the monorepo that represents this microservice, and I can match the flow of all the other microservices, and I can test all my microservices the same way,” then you can get a nice microservice in a monorepo pattern that tends to work out pretty nicely.

[00:11:26] KP: Well, it's no surprise to me that Windows and Office would follow monorepo patterns. I am surprised to hear about microservices fitting into that approach. There're a lot of companies now that there is forest of different microservices that all interact with one another. Do you see that as a common trend that people even in that world are adopting monorepo?

[00:11:50] DS: I think that everyone has their own opinion on this. And when I see people working across many, many repos, to me that seems incredibly confusing. And like how can anyone work across team lines? It seems like the whole idea of being able to collaborate across teams becomes a lot harder when everyone's in their own repository and they've got their own rules in each one. And so the benefits of monorepo in terms of being able to do code sharing and having a unified workflow for everyone in the company, I find those benefits to be really good. But other people say that that's too much. Other people might say that that is too heavy-handed to say that everyone needs to conform to what one group is doing. If you're doing lots of things in different languages, say, “This microservice is in this one and this other one is in that one.” If you want to have that freedom to very quickly change completely everything when you make a new service, then maybe that's something you're interested in doing. But to me that just screams disorganization, right? How can somebody else chip in if everything's going to look completely different when they want to go look at what you're doing?

[00:12:57] KP: So for a team that's bought in on monorepo and they're intending to build quite a big project, hire a lot of people. They have a vision down the line then it's going to be this massive monolith application, are there any preliminary steps they should take to make sure their repository is healthy and not going to be bloated with things and stuff like that down the road?

[00:13:20] DS: The biggest things you can do is to make sure, for instance, that really large binaries are out of your repository. Git is really good at handling source files, right? Plain text files. And compressing them really well in its storage and being able to communicate that. But when you get to a large binary that it doesn't know how to diff, then those things don't really work as well inside of Git repositories.

There are tools like Git LFS that can help, but that doesn't change exactly how much data is being stored. It's changing where it's being stored. So if you can move those to something like a package manager instead, right? Sometimes you have test artifacts that are just large binaries. You just want to say I want to test this version of this really this binary format and I want to make sure that it works. Can you put that in a package somewhere? As opposed to in your source control.

The other things to do is to try to keep your build system componentized. As your build gets bigger, you want to make sure that developers can incrementally work and build only a slice of the cone and make sure that they are – When they're doing a small amount of work, they do a small amount of builds. You can rely on CI and your PR builds to kind of double check that the whole system works and not block the developer as they're doing their local compilation.

And the more you can do that, it actually starts to feed back into the source control side, because you can use something like Git sparse checkout to limit how many files you have in your working directory, which makes the Git operations more quick. And it means that if you're in a partial clone, again, the sparse checkout limits how many files you need so you download even less data as you're working. But it really requires your build system to be organized in a way that you can say, "I only need these directories in my working directory in order to do my work and continue being productive."

[00:15:06] KP: So let's say I'm working on something like fraud detection at an e-commerce company. So I'm going to need – Maybe there's some objects that represent the transaction and the user. I'm going to want all that. But I don't need the auth system, or the checkouts, or products, or any of the store. I need maybe a specific kind of sub-modules. It seems pretty intuitive to me that there's a way I can sparse check out just my needs. But those needs could be in all different pieces and nooks and crannies of the repo. What do I have to do to successfully execute on sparse checkout versus what does Git do for me behind the scenes?

[00:15:43] DS: Well, I like to think of Git is the substrate. It is the fundamental medium in which we're sharing code. And so it doesn't have an opinion about the build, for instance. It's something that it will learn from what you tell it to do. And you can just say like Git sparse check out and that to set it up. And then Git sparse checkout set to say these are the directories I care about. So in terms of how to interact with it, it's very simple once you know what those directories are.

But in terms of architecture, you can think about how can you set it up such that you have a lot of fan out and you're not going really deep into these nooks and crannies. And one architecture I've seen a lot is where you have these dependencies between services, but the services themselves can separate the code of their contract versus their implementation. So for instance, you're doing this fraud detection and you need to know about users and you have to call the user service. Well, you need to know the information about what that user service is promising, but you don't need to know its implementation in order to do your work, right? You can create this fraud detection service with this idea of the contract. And maybe even in your tests you have some ways of mocking that user service to make sure that you have unit level tests that you're checking on the developer machine. But then maybe as you push, you then do the integration tests across the fraud detection and user implementations to make sure that everything is actually making sense. You didn't make a mistake and your mocks have a different promise than what the user service is actually doing. And that allows you to then say, "I can cut out the user service implementation from my working directory, because I don't need it. I shouldn't need it. I shouldn't even know about it to do what I'm doing over here." And that's the kind of way you want to be able to think about how can I segment and componentize my working directory so that these things are possible?

[00:17:36] KP: Does monorepo put a technology group at risk of their code leaking out or maybe by I guess having developers have only smaller access you could limit the damage of their blast radius? What are your thoughts on security?

[00:17:51] DS: Security is a really important problem. And we think a lot about it on our side. We want to make sure that our servers are secure, that authentication is important. So that way the server-side is never a component that could be leaked and accessed incorrectly. On the client side, you absolutely do have some benefits that you only have a certain amount of data. Although because you're authenticated to the model repo, I don't think that there's been any work in saying you're only authenticated to see a portion of the monorepo, right? You can say I can read this repo. I will download what I request, and whatever I request will be there. So if somebody went through and said I want to check out the whole repo. That is allowed, but it does take a long time. So having them say this is the scope that I care about means that there's less data on their machine to possibly be leaked.

[00:18:41] KP: And have you seen any best practices or do you have any best practices or maybe recommendations of what you've seen work really well for organizations specifically around getting new developers onboarded to these large projects?

[00:18:55] DS: Right. When you have a repository this big, you're usually thinking hundreds or thousands of engineers. At that scale, it’s worth having a team of up to a dozen or more engineers whose focus is on keeping those engineers productive. And that means that they are the ones who are investigating what are our build tools? What are our best ways of making sure that developers can get started in the repository? So maybe they're the ones who are recommending, “H, y when you get started, here are the instructions for partial clone. You should do it this way.” Or, “This is the way we've set up our build system so you can use sparse checkout,” right? They're the ones who are educating the other engineers to make sure that they are working.

And then when those engineers are having problems or friction as they're developing, they go to that team. And it's kind of a communication throughout the entire career of those workers. And I see that a lot in both. Windows has a team like this. Office has a team like this. And they are really invested in developer experience. And that helps in so many different aspects. And having that one team this unified allows to satisfy the needs of all the different teams and make sure that everyone is kind of moving in the same direction and you don't see this bifurcation of, “Well, people over here do it this way. And people over there do it another way,” because that's what they configured out. But if there's this team that's coordinating and educating that, that's typically necessary to scale.

[00:20:19] KP: And do you see people having to adopt any special policies around branching and merging? When you've got a lot of contributors on one repo, I could see where some conflict could arise. Are there secrets to doing that successfully?

[00:20:33] DS: The biggest thing is I really believe strongly in branch protections. You want to have that – However you're doing your flow of code to make sure that you have a green branch that everyone's kind of basing on. And you keep it as green as possible by having really strong pull request and merge policies to say this cannot be merged unless you have a certain amount of builds. You have all these builds being green and a certain number of approvals from certain teams. And breaking that down even more by if you're in this region of the code base, you need to be having approval from this team is really helpful on that side. Code review is a human process that is not just for catching bugs. It's also for educating. So making sure people are aware of what's happening is really beneficial there.

I’ve seen things also that I in the Office in Windows monorepos that they don't even have just one trunk, that they kind of split into multiple trunks that all have branch policies. And so if you're working on, say, Word, you have a different trunk than Excel. But then those things at certain cadences merge into the big trunk that everyone has and merge back. So that way code is flowing but it also means that if somebody does have a catastrophic problem that was unexpected, wasn't caught by a PR review or if some wires got crossed in Word, then Excel is still moving forward without having to have any disruption. And finding ways to kind of make sure that flow works out pretty well is super valuable.

Especially at that scale, it's really hard to say I can't merge without running all of the tests. Because then developers would never be able to get done if you have tests that run for hours and hours and hours. But you can say the word can't merge into the main trunk unless it's gone through these hours and hours of tests and passed there. And so you can have extra levels of validation if you have these extra layers of checks.

[00:22:30] KP: Oh, that's interesting. I was going to ask you about that, yeah, that one criticism you could weigh on monorepo. So what about you know people already have enough problems with so many tests taking so long? Are there any tricks, techniques or tools to help organize that? I can definitely see where each subgroup, each office component needs to run all their own super long tests. And then there's probably a shared set that may or may not have some of its own super long tests that do something global across all of those. Is there any tooling that can help orchestrate that?

[00:23:03] DS: I know that there are lots of build tools. Microsoft's version is called BuildXL, used to be called Domino. And it has this idea of every single node of the build DAG has a fixed set of inputs and fixed set of outputs. And so we can use hashing to say, "Based on these inputs, do I already have a build artifact of that kind in the cloud somewhere?" And so that really accelerates the build to say, "Well, I only change this portion of the code. So I can get all the build artifacts for the rest of the code that have already been cached and only rebuild the parts that have changed." And that can apply later to testing as well. If the tests are checking these components, you don't have to run all the tests. You only throw in the tests based on the components that changed. And that can lead to really efficient DR checks. But it requires having this very advanced build system, which is a little bit also – This build system is also very rigorous. That means that it's really difficult to make sure you're staying in that system and getting yourself into that level of rigor.

[00:24:05] KP: I've seen it. I've read a lot on your blog about the work you've done with Git commit graphs. I have to say I don't spend too much time in my own development looking at my graphs. Maybe that's because I'm on a smaller team or I just don't know how to read them well. Do you have any advice for developers who are maybe looking to level up in their usage of Git? What opportunities do people have to get some insight from the metadata about their code and the teamwork they're doing?

[00:24:32] DS: Yeah. The reason I'm really excited with the commit graph is that, yes, the graph shape itself is something I like to think about as a former graph theorist. But that's not the reason you're looking at the history. The reason you're looking at the history is to say what has been happening recently? If I want to look at which pull requests have merged recently. One trick I like to do is Git log—first-parent to say go down the first parent of each merge and ignore the second parent. And what that does is it just shows you, for instance, if you're using a pull request flow, the pull requests that are completed. It just shows you those merge commits. It does not show you the commits that other people had in their topic branches that also got merged in. So you can see a linear sequence of merge, merge, merge, merge, merge. And that's a great way to see kind of what order are things coming in.

But the more valuable thing is saying I'm trying to figure out this code and I want to understand it better. How can I use the commit history to help me do that? So just doing Git log and then putting in the path can tell you which commits changed it, but usually you want to do something more involved like who changed this set of lines or this function? And Git log has a -L option that allows you to either say a range of lines or even a function name and say, "Give me just the commits that changed this region of the file." And you can go through all those commit messages and it'll be just that really, really small list of things. And hopefully the developers before you or even your past self left very careful commit messages to say, "Hey, I'm making this really isolated change, and here's why I'm doing it. Maybe these are even things that I tried before or I'm doing it because this special case happened." And you can go look at that commit to see which test cases got updated. But really you can say, "This is the part of code I care about. Why was this decision made?" And you can go and find that answer very quickly.

[00:26:25] KP: Do you have any hands-on examples of moments when that's been insightful maybe in your own development or helping a colleague figure something out?

[00:26:33] DS: I've been working in the Git codebase, and the Git codebase is over 15 years old. And one thing that you find out really quickly as a new Git contributor is that the standards for what your commit messages includes, especially its format, but also just the level of detail, is it's really high standard. It was something I had to really learn and think about really carefully as I was getting started, but now it's second nature to say I'm going to take this part of my code that I'm changing and I'm going to write a long commit message about it with all these details.

And when I was getting started I saw it just as here is something I have to do to convince people that this is a meaningful change, right? I have to go to the open source community. They don't know who I am. And I want to convince them this is actually moving the project in a positive direction. Now that I've been the codebase longer and I've been doing more of these history things, it's like, “Oh, it's also documentation for when I go back and see a problem.” And I'm working in the area of the index recently, which is kind of essentially the data structure that stores your staging environment. When you hit run Git add, that change is stored in the index. When you run Git commit, your index is essentially used to create a commit. That data structure is so old and it was built kind of in the wild west of Git days, back 2005, 2006, when they were just getting started. A lot of the code that I'm blaming or running this Git log -L stuff goes back to those days when they weren't so rigorous about commit messages. And I find things like why was this written this way? And it was literally the messages I rewrote this subsystem just. I rewrote it because reasons. And so I'm really appreciating that when I hit something essentially like 2008 or later, I've got a really robust commit message to kind of explain what's happening. Sometimes it's a platform-specific thing. Although at times it's just this really weird corner case. And, “Oh by the way, here's the test case that handles it.” So I've appreciated all these different things that I see especially in comparison to these really early commits that didn't have that.

[00:28:32] KP: Yeah, absolutely. You'd mention the relationship ideally between good commit messages and documentation. Do you have any vision for what an idealized team would be doing in that regard?

[00:28:45] DS: One thing that my team has been working on really closely partly because we work in Git and on a fork of Git, is that we are reviewing each other's commit messages as much as we're reviewing our code. If we see a pull request, we don't look at it for the these are all the files that changed. We look at commit by commit. What is this trying to tell me? What's the story? And we recommend things like, “Oh, here's a way to reorder your commits or split them up in a way that would tell the story better.” Or, “This piece that you change here in this commit doesn't need to be in this commit. You could split it out and justify it on its own.” That's a good way to teach these people. Especially some of them are newer contributors. How to prepare for sending things to upstream Git? But also just really double checks like do we need every single piece that's being here? Because every single change is adding risk to the product. So we want to make sure that they're meaningful that the value they present outweighs that risk. And viewing it at that meticulous level is going to pay benefits when we later need to assess, “, what was going on here? Why were you making those decisions?”

[00:29:48] KP: So I find some developers maybe it's bad habits or an attempt to get more done and take a shortcut, are quick to kind of let some of these things go to the side. And maybe it's, I don't know, personal pride, or professionalism, or the culture of the company that needs to be affected to get that to bubble up. If I’m an executive and I want to see that culture developed, how do I get a direct benefit out of getting everyone who's making commits to take a little bit more seriously what they're writing in the messages?

[00:30:18] DS: It's really difficult to measure productivity. And it's even worse how to measure good practices. If I were just to say, “Hey, I want to make sure you have enough commits per pull request.” That's not a really good measure because a lot of it could be just responding to PR feedback, responding to PR feedback, responding PR feedback, fixing up a bug. But really that's not what. We want we want to have high-quality messages.

So I’ve thought about a lot about how to measure it, and there's really one way to do it, is to do something like a developer satisfaction survey, which is to say you know, “What are your opinions of working in this codebase? Is it an enjoyable thing to do?” And a lot of that's going to boil down to is it easy to build? Is it easy to deploy? Can I see the actions of what happens when I make a change?

But one thing that can happen to that is does it make sense what's going on in the codebase? If I see something that's unfamiliar, am I able to determine why that was made? Or am I able to understand what's there? And that's one of the things where you really can only understand it with the benefit of hindsight, that I was making good decisions before in the last year or five years, and that's leading to me being in a good state now. And it's really hard to do that when you're in a greenfield project. You just want to move fast, right? I want to I want to just keep on going and build the next thing and build the next thing. And even to the point of I just need to get this out there. So even some tests are going to not necessarily be high priority for me. So why would quality commit messages be high priority for me?

But when you're in the process of actually having a product that's being supported and in the long haul of this is something we depend on as a company, that is where this is super important, to be able to understand what has happened a year from now or five years from now. Understand what you're doing today. And so putting in that effort is really valuable.

[00:32:09] KP: So let's put ourselves in the shoes of someone who's in that early stage under deadlines getting things done and they've gotten themselves into a little bit of a predicament with a messy, heavy, bloated repo. Obviously, it's going to be case by case. But do you have any thoughts on whether at that point we just start over and make something new? Or do we try and go and clean that repo up?

[00:32:30] DS: There are lots of ways to clean repositories up. Some people really advocate for rewriting history and saying, “Let's go and do a filter repo or something where we can remove all these large binaries but maintain our old commit history.” Other people can do things like, “Let's make a cut and point. Let's just start a new repo where what's currently in main is the root commit.” And that can be a great way to very easily map things over. But you have to then say, “All work in flight needs to be transferred over the new repo.” And so there's not that either way you have to do something where everyone currently working in the repository needs to pause what they're doing and transition it over to the new world. It could be a new commit history at the same repo URL, or it can be a new repo with a new history, with lack of history rather. And none of those options are good. So the better decisions we can make now, the better. The more we can make good decisions now, the better it'll work for us in the future.

[00:33:28] KP: Makes sense. Yeah.

[00:33:30] DS: But I've seen both work.

[00:33:32] KP: Well, do you see any anti-patterns that end up being really damaging in the long run? What should people be avoiding?

[00:33:39] DS: One anti-pattern I see that some people really believe in it, but I tend to not, is the idea of always squash merging. Or even making your history linear. I guess another way to saying it is you can do squash merge. So you have a linear history of this is a pull request merge, pull request merge, pull request merge. Or you can have even rebase and fast-forward, which means that you also see that person's topic commits in the main line of the history.

And I find that that can be problematic in multiple ways. One, just being a commit graph nerd and thinking about how file history works, is that if you keep those merge commits and those topic branches, it actually speeds up your file history because you can skip a lot of commits. Because actually if you see that, "Oh, this topic branch changed this path." I can actually follow that topic branch and skip a big chunk of history. And lots of other things can happen that can't happen when you have a linear history.

But also like that means that you're losing that idea of here's this topic branch and my commit story that is meaningful that either gets lost in the squash merge or it's present in every single history of you could possibly see. And neither of those really work from for my taste. And if I think about on the side of Github and which repositories are hard to host and which ones are hard to serve in terms of our file history queries and other things, we find that these linear histories are the ones that are more difficult to scale at this range.

[00:35:11] KP: I'm curious if there are any, I don't know, edge cases you're currently exploring in Git. Are there any projects that have pushed to some extreme limit in one direction and are starting to shape how you evolve the product?

[00:35:23] DS: I've been working on a thing this year. We talked a little bit about partial clone and how that reduces data. We talked about sparse checkout and how that changes your working directory. Those are really critical to how the Office monorepo scales, is to be able to say, “If I work on Word, I don't need the Excel code,” right? And so that's how it works.

The problem is that there's the – We also talked about the index, which is this data structure in Git that essentially is talking about what's your staged environment. And it lists every file at head even if they're not in the sparse checkout cone. And that's just a record of how it's been used in that, because sparse checkout used to always have a file-by-file path matching algorithm. It said, “Here's a set of regex patterns practically, and for each file in the repo I will match against those to say whether or not it's included or not.”

Recently, about two years ago, we built a cone mode sparse checkout matching which says we're going to use directory-based matching. So if you're in these directories, we care about all those things. But if you're outside that directory-based thing, you can say – As soon as I get to a certain directory, nothing's included. And that allows you to say, “Oh, if I go into the excel directory, nothing's included. I can skip doing things.” It makes that pattern matching a lot faster.

But it turns out that that could also give us a benefit for the index, because now the indexes need to store every path and to say is it included or not? We can create what's called sparse directory entries and say, “Oh, inside the index, I have a mixture of files and directories. And that directory can point to the tree it is representing in the Git database. And that allows us to shrink this index file from about 180 megabytes to like 10 megabytes. So things like its status have to load the index into memory, operate on it, and then sometimes write it out. And when you have 180 megabytes you have to do every single time, that's just a big amount of data, especially when you're not doing anything with most of that data. You're just saying, “Let me load all these records that I don't actually care about. Do some processing and then write it back out.”

So the sparse index is what we've been building to allow these sparse directory entries. But because there's so many things to the Git codebase that care about the index, and it's really old code for the Git codebase, we needed to very carefully integrate with Git built-ins one by one. So we started with Git status. Okay, can we make it status faster doing this? And we're getting good status from one and a half seconds to 300 milliseconds with this operation, which means that developers are feeling like Git is being a lot more interactive as opposed to something you run and then wait for.

And we've gotten probably about 13 of the most important built-ins integrated now and such that we're really close to releasing this to the Office developers and seeing how much they're going to improve their experience. But it's going to be pretty dramatic I think when they see this, especially complicated things like Git stash, which Git stash loads and parses indexes multiple times throughout the run. So it's going to go from the range of 10 to 12 seconds down to under a second. We're really excited to see how that goes.

[00:38:30] KP: So what's your expectation in terms of productivity? Will there be some sort of change point you can measure on this release? Or is it just an incremental improvement?

[00:38:39] DS: This one I think is going to be – You can think about it, so everything we've been doing so far is incremental. But this one, it feels different. It's going to be a bigger jump than we've seen before. And the most important thing is that when office moved from their old version control system on to Git, their developers were happier because Git was already working faster than their previous version control system. So they were already being more efficient. And now we're going to add this extra 3 to 10X performance boost on top. And so that way hopefully Git no longer is anything they have to wait for, even be aware of. It's just kind of things are happening, especially because a lot of them are using tools like VS Code that have Git integrations. Their UI is just going to be super responsive with these things running underneath the hood.

[00:39:25] KP: Very cool. Well, I’m wondering is there anything about your academic background in graph algorithms that informs the way you approach some of the problems you work on today.

[00:39:36] DS: There's one philosophy I’d like to use for my academic days and bring it over into code. And you know being a mathematician, everything was about writing proofs. And the idea was to like let's create these big theorems that are these really cool theoretical statements. In order to build those theorems, we prove things called lemmas, which are kind of sub-steps of proofs. But then those lemmas could be used maybe for multiple theorems. But then those theorems also had corollaries, which are things that follow quickly from the statement of the theorem.

And so I'm constantly thinking about when I'm working on a project, if I want to build something like the commit graph file, like that's going to be my theorem. But how do I get to it? Well, I need to build these sub-tools. I need to build a file format that's extendable so I can add things to it. And I'm going to make that be here. And I need to plug into this part of the subsystem. That's going to be another thing. And then they come together, "Okay, we've got this commit graph file." And it works really well because it's so much faster to parse that structured data than to go parse commit data. Well, what can follow from that is, well, now I've got this other place where I can store information and metadata about commits. And we can add, for instance, this thing called generation number, which is a metric to say essentially how high up are you in the history. And that extends to saying, "Once that's stored there, we can make certain algorithms much faster." It's essentially how that Git log graph works really quickly, is that I can scan a lot smaller portion of the graph to make certain reachability decisions. And that falls really nicely from the creation of the file. And it's something that people were working really hard, say, in 2012 to try to put – They wanted to change the commit format to include this data. And it was essentially saying we can't do that for compatibility reasons. And so it kind of died on the vine there. But once we had this other file format, that was something we could extend.

So I like to think about taking these big projects and say what can we do as these smaller pieces that build up to this final big bang, but then also once we've got that out there, what follows shortly after to get some what is now low-hanging fruit that this big effort was completed?

[00:41:36] KP: Derrick, I'm curious what's up and coming for you. We talked about some features and releases. Anything else you want to highlight?

[00:41:44] DS: Yeah. I was talking about sparse index earlier, and we're set for releasing that in our fork of Git, the Microsoft/Gitfork, which is what we use to service Windows and Office model repos. That's going to release in November along with Git 2.34. And it's got just some earlier integrations with the sparse index that aren't upstream yet as we continue to send them upstream. But most of these stuff is actually in core Git already.

In line with that, I'm actually co-presenting at universe in the late October with my co-worker, Leslie Dennington, and we're going to talk about a lot of these same things. But also a lot of other things and how they build together, talking about things as simple as make sure you upgrade your version of Git to have the latest features. To all the way to here's the latest and greatest sparse index thing and everything in between about how to really tell the journey of how you need to think about different features on your journey as your repository scale.

[00:42:41] KP: Do you have any other tips or tricks for practitioners? What should I be doing to upgrade my Git game?

[00:42:47] DS: Yeah. Well, first, yeah, upgrade is a great word. You should make sure your Git versions are updated, upgrade frequently. There are new Git releases about every 12 weeks. So keep an eye out for those. The GitHub blog usually does an announcement. But also you can just follow Twitter or you can watch the Git repository on GitHub.

If you have a recent version, for instance, in the last year or, there's a new feature called background maintenance. You can go to whatever repository you're using really regularly and run Git maintenance start. And that'll essentially kick off this background maintenance. It will set some config in your repository saying, "You know what? Don't ever do auto GC." That sometimes runs at the end of a command. And instead it will have a schedule that will do certain things. Like it will do some repacking every night around midnight to make sure that your repository data is organized really well.

It will also do things like fetch every hour from all your remotes. So that way if you run Git fetch in the foreground, it'll update your remote refs, but you're only getting the data that has been added to the repository in the last hour. So your fetches and pull times will get a lot faster. Even if you go on vacation for a week and you come back to your computer, it'll be good to go.

It'll also do things like that commit graph file I was talking about. It'll keep that up to date so that way things like Git log graph are constantly working really, really quickly. And there're some other things you can do. You can even go into the config and set your own maintenance schedule. For instance, if you really want to do GC every week, you could do that. So there's lots of ways to customize it. The default schedule is something that we've been using for our mono repos and we're really happy with. But if you feel like you have different ideas, it is extendable in that way.

[00:44:24] KP: Good advice. Derrick, where can listeners follow you online?

[00:44:28] DS: I'm on Twitter. I'm @stolee. You can also find my personal webpage, stolee.dev I'm at those places. And you can also you know take a look at my GitHub, github.com/derrickstolee, and drop a line. My DMs are open, and I love to hear from people.

[00:44:48] KP: A lot of good resources. Derrick, thanks for taking the time to come on Software Engineering Daily.

[00:44:54] DS: Thanks, Kyle. It's fun.

[END]