ALAN: So, you know, when I was growing up, there was this mechanical coffee machine in my dad's office that was just fascinating to me. You would drop in a coin and listen for it to clink into the bucket, and then you could push the buttons on the machine to get it to do what you wanted. Do you want light blend, or dark blend, or whatever?

And then, it would just, like, spring to life. It'd be hissing and clicking and whirring, and I didn't know what was happening behind the scenes, but it was fascinating to listen to. And then, at the end, it would like...this glorious aromatic black liquid would just splash into the cup. That was so cool.

JARED: Alan, we're talking about software today.

ALAN: Oh, right, right, right.

JARED: Welcome to Dead Code. I'm Jared. And just a little housekeeping reminder, I guess, I have a mostly weekly newsletter, where I often talk about the latest episode of Dead Code once it's had a little bit of time to sit and percolate in my brain. Usually, we're recording them at least a week or two in advance, and by the time they come out, I've got thoughts, things I wish I'd said, elaborations, something fell out of my brain.

And my newsletter is always broken up into three little pieces. One is usually just me rambling about some technical thing or programming thing. The middle section is about that week's episode of Dead Code. And then, I always finish it off with a little music recommendation. I spend a lot of time listening to new music, and so I try to find something I'm excited about to recommend to people. So, you can go to jardo.dev and subscribe to that. The subscribe form is just in the footer of the website.

Today I am joined by Alan Ridlehoover, who I've seen speak a couple of times now and really enjoyed every time. He joins us to talk about coffee machines. Alan, welcome to the podcast.

ALAN: Hi. Thanks for having me.

JARED: Could you introduce yourself to our audience?

ALAN: Sure. My name is Alan. I am a Senior Engineering Manager at Cisco Meraki. I lead the platform engineering team for the Meraki dashboard. I've been in the software industry for about 35 years, almost exactly. Graduated in June of 1990, so it's been 35 years.

JARED: So, I was one year old when you started in the industry.

ALAN: [laughs] Well, I won't hold that against you [laughs]. Let's see, what else...I am primarily a web developer these days. I mean, I'm primarily a manager now, but prior to that, I was a web developer — Ruby and Rails mostly. That dates back to about 2010. Prior to that, it was a lot of ASP.NET. Prior to that, it was C and on the Windows desktop. Been around for a long time.

JARED: And I think I first became aware of you at...did you speak at RubyConf Mini?

ALAN: Yes, I did.

JARED: Yes, and you were talking about coffee machines then.

ALAN: Yeah, that's right.

JARED: If I remember correctly. Yeah, so I was there for that. And just recently, I was reading some blogs. I noticed you had blog posts about coffee machines, too, and perhaps maybe about...more important, programming ideas. Coffee machines might not be the most important takeaway, but you've been talking about complexity in software. And you describe it at the start of your blog post series about the coffee machines as...it's about how hard code is to understand. Why did you pick that angle for talking about complexity, rather than algorithmic complexity or maybe one of the sort of harder descriptions of complexity?

ALAN: Yeah. So, like I said, I'm a web developer. I don't deal with a whole lot of algorithmic complexity, per se. And the domain complexity that I deal with, I think, comes out...it falls out of abstraction. Like, how much abstraction is there in the application? I spend most of my time working on a method that has business logic in it. Business logic is not ordinarily algorithmically complex. It can be, I suppose, but it's not usually.

So, I'm most interested in, how hard is it to understand this bit of code that's right here in front of me, right now, in the editor? That could be a method, that could be a service object or something, but it's generally not bigger than that. I'm trying not to hold too much context in my head at one time. I'm old. It falls out really easily [laughs]. So, the key there, for me, is how much complexity am I dealing with in the minute, in the editor right now? And, like I said, it's not usually domain or algorithmic complexity.

Where you get into trouble is if you get too cute and you try to over-abstract something and, usually, you're trying to reuse something somewhere, and you're like, "Oh, if I could just add this interface here, I can reuse that." That's when you start to get into trouble with domain complexity. But as long as you limit the number of layers of abstraction that you're dealing with, and you focus on the method that's in front of you right now, you can simplify your life significantly by paying attention to the complexity.

JARED: So, in the coffee machine series, which we'll have links to that in the show notes, you start with a simple vending machine, and you keep adding new features to it. Was that based on a real-world experience with a feature, or was that just a good sort of example test, system for you to talk about complexity?

ALAN: You know, I don't remember the genesis of where it came...it's been years ago now. I think we did the first version of this talk at RubyConf Mini, which was, what, 2022? So, I don't

remember where we came up with the coffee machine as the central focus for that, but it worked out really well. Obviously, I've never built a coffee machine, so I don't know if it's [laughter] actually true to what a real coffee machine looks like, in fact, I doubt it is. But it's a simple heuristic that most everybody is kind of familiar with, and it allowed us to wrap that complexity discussion in a pretty simple, little application with an easy-to-demonstrate thing.

Now, I have witnessed code do that, that particular application, no. But most of the time, though, I've joined teams that were already in flight. So, I'm dealing with the aftermath of that, where they've added feature after feature with conditional after conditional, and everything ends up in one place. And you end up with a method that's got an astronomically high ABC score, and it's just hard to wrap your head around everything that's going on inside that method all at once, and, honestly, you probably don't need to. You can probably add at least one layer of abstraction there and make it simpler to understand.

JARED: Yeah, for our listeners that aren't familiar with the ABC metric, what's that?

ALAN: So, ABC stands for Assignments, Branches, and Conditionals. It is a quantitative measurement of the complexity of a method. And I think the actual...let's see if I can remember the algorithm. It's A squared plus B squared plus C squared, and then you take the square root of that. You don't really need to know that. There are tools that'll do that for you. So, you just count the number of assignments, the number of branches, which, branches, in this case, calls out to other methods. And then, conditionals is conditionals. It's quantitative, which means it's not biased in any way, and you can compare, like, language to language you can compare.

Generally, you'll end up with different scores in different languages because the heuristics are different for each language, but you can come up with ranges of “this is what a reasonable score is,” “this is what a borderline score is,” and “this is what an unacceptable score is” for each language.

JARED: Yeah. I noticed you use flog in the blog posts as well. And do you frequently sort of rely on these kinds of hard metrics to guide your decisions about when to change designs or refactor, that kind of thing?

ALAN: I have really three different ways that I think about that. The hard metrics, the quantitative metrics are the easiest to argue. I can point to the metric and say, “Look, it's quantitative. It's not biased. It's just doing its thing.” But there are a couple of other ways that I think about it as well.

I am a huge fan of Sandi Metz and her work, and she has some rules around things you should and shouldn't do. Her rules include things like: a method can be no more than five lines long. That’s a good heuristic for keeping it simple. It's not looking at the complexity of those five lines, I mean, they could be Perl-esque. They [chuckles] could be super complex. But keeping it short is one way to manage things.

And the other one is, I pay attention to my gut. Like, does it feel like I'm getting slower as I'm adding new features to the code? And if it is, maybe I should stop and reflect on the design a little bit.

JARED: Yeah, that's interesting that you bring that up because you also talk about co-locating test setup within tests themselves, even if that means duplication. And I was recently talking with Carson Gross, who's the creator of the HTMX framework, about that idea of...sort of counterpoint to breaking things down into little pieces, where you often want to see certain details alongside others, and co-locating them has a lot of benefits. How do you sort of balance those sort of extreme small method styles of breaking things down with forcing someone reading through some code to dive down into these little methods to understand what's going on?

ALAN: Let me use a concrete example. Say you're working in a controller, like I said, I'm a Ruby guy. I'm a Rails engineer. I'm working in a controller, and there's enough business logic there to make it significantly complex. Say, I have a complexity score around 50 or 60. For this method, it might be 20 to 30 lines long. It's not so complex that a human can't grok it. It can still fit on one screen, right? Maybe you leave that code there.

I have seen methods that are 600 lines long. I cannot get that on my screen, not only that, I can't hold context of the top 50 lines when I'm reading the bottom 50 lines. At that point, absolutely it makes sense to start splitting that thing up. By the way, the method in particular I'm talking about has a flog score of, I think, 1800.

JARED: Whoa [laughs].

ALAN: So, that's some scary code. And if I can find a way to pull abstractions out of that code, not only does it make those pieces that I pulled out much simpler, but if I can get to the point where everything is at the same level of abstraction in that top-level method, it'll make that thing a whole lot easier to understand.

So, if I can just have calls out to specifically well-named, like, service objects or some other pattern, I'll end up with a much more readable controller action. And each one of the pieces that I pulled out will also be much easier to grok. So, it's, like, relative. Personally, my preference, I want to get things down to Sandi's rule. I want to get all the way down to five lines if I can. Can't always do it. There are reasons why I wouldn't, but that's my preference. Now, I am particularly comfortable with abstraction, not everybody is, so, you know, to each their own.

JARED: Yeah. You refactored the coffee machine's event method to use private methods like heat water and grind coffee beans and that kind of thing. You are almost creating a domain-specific language in there. How do you decide what, you know, what abstractions are worth creating?

ALAN: For this, I'd like to think about it in terms of a ubiquitous language for that particular domain. So, we're dealing with a coffee machine in that case. So, grinding beans and heating water, those are definitely parts of that domain. And you're right; it's a domain-specific language for coffee machines. I think another one of the methods in that machine is, like, drop cup or dispense cup. You know, it's just something that a coffee machine...like, this is a vending machine. It's not like your Mr. Coffee or your espresso machine. This is specifically meant to be a vending machine, and so dropping a cup is a part of that vernacular.

At work at Meraki, there's a whole lot of networking concepts built into the codebase. I mean, you want something that is ubiquitous to that domain, basically. So, the ubiquitous language concept comes from domain-driven design, Eric Evan's work.

JARED: Yeah, and you mentioned that these abstractions are sort of the heart of simple code, and you use, I think, hardware drivers as an example. What makes an abstraction trustworthy versus one that might leak or break down?

ALAN: I view a trustworthy abstraction as something that does what it says it does. There's no side effects. There's no extracurricular stuff going on inside that API. And one way I like to think about it is, if I walk up to your house and there's a button next to the door, when I press it, I expect to hear the doorbell ring. I do not expect to hear the toilet flush. There's a certain expectation given the presentation of that, the affordance that particular method is giving me. So, for example, dispense cup, I think that's all it should do is drop the cup into the spot where it can receive the coffee. It would be unreliable if it also did something else.

JARED: One topic that I have strong opinions about that I maybe might differ from you on is you talked about comments, and you briefly mentioned they will always lie to you because they're not executable. How do you balance this view with, you know, the need to document, for example, why a decision was made around some business logic in your code?

ALAN: Yeah, I don't know what your opinion is. You hinted at it, but I'll share mine. I think that comments will always eventually lie to you, even comments about why a decision was made. And I'll give you some examples. I have seen a lot of code over the years, and I've seen some comments that just stand out as hideous comments. I saw one comment one time that said, "The previous comment lies." [laughter] Well, why didn't they just delete them both? Come on.

I've seen comments that say things like, "I'm not sure how this works. Figure it out." I mean, "Write a test that tells you how it's working." I've seen comments that say, "Fix this bug number 1, 2, 3, 4." Like, okay, at this point, I do have a reference I can go back and check, but what's going to keep that comment next to the line that fixes the bug? Over time, it may move through either an automatic code mod or some accident that the engineer makes.

And so, I think that comments generally are going to lie eventually. And the way that I solve that is...there are a couple of things. The first thing is I want to name things really well. Now, I can name things well in the sense of what that thing is doing. I'm not going to be able to tell you why

in the name of the thing, like, that's too much to ask of the method name. But that goes back to, what am I trying to communicate with the code?

I think code has a couple of different reasons to exist. One is to tell the computer what to do. I can do that with Assembly language. Why don't I just use Assembly language? Well, because I can't solve the second need, which is to tell other humans what the computer's supposed to be doing. So, we use high-level languages, Java, Ruby, whatever, to get to the point where other humans can understand what we're doing easily.

Now, we want to communicate what the computer is doing with the code. I think tests can fill in some of that. Tests tell you what the computer should be doing. They also have an ancillary benefit in that they can verify that the computer is doing what you think it should be doing. But code and tests I don't think are a great place to communicate “why?” I think why belongs in the commit message. Why is we made this decision, and, therefore, we're changing this piece of the code.

That's, you know, the why behind the change makes more sense than the why behind this particular line of code. This particular line of code is only communicating what the computer should do or what we're telling the computer to do, and the test should tell you what it should be doing. So, that's my take on it. I'd love to hear yours.

JARED: Yeah, I mean, I have a few thoughts. I mean, generally, I think there's a lot more under-commented code out there, you know, code that is difficult to understand and doesn't do anything or maybe doesn't do enough, whether that's better naming or information in the commit messages or comments. It doesn't do any of these things well enough to really communicate the intent. And I think that while I agree that comments can get out of date, I found that, in practice, good comments at least don't get out of date that often.

I think, you know, we often write code that, let's say, could be changed with side effects and have side effects we don't like in the same way that the code can diverge from a comment, and we just live with that. We just fix it when it becomes a problem. So, I don't know, I'm torn.

I think I agree, yeah, comments will eventually come back to bite you. But I'd rather just see more comments. They're right there with the code. It's, like, the most immediate way you can communicate an idea to someone besides the naming in the code, which I fully agree, if you could just communicate everything you need in the naming, then go for that. Don't put extraneous comments in there.

ALAN: What I try to do when I find a comment, and I'm specifically talking about inline comments here, when I find an inline comment that explains the next four or five lines of code, I just take the comment and make it a method name and stuff those four or five lines in that method. It could be a private method. It could be something I extract into another class. But the goal there is to communicate that thing in a way that if it changes, it'll break a test. It'll actually

tell me that it's broken, versus the comment staying there while somebody modifies those five lines of code later, and now it no longer does what the comment says it does.

So, I mean, the danger's still there, right? You could go into that method, and you could change those things. But, in theory, I've got a test somewhere that tells me that, that when I call that method, these are the things that are supposed to happen. So, I will say this: I think there is a place for comments, particularly when you're building a public API. If you have an API and you're trying to define, and this is really more of a tooling thing, like, you're trying to communicate to the outside world how to use this API, you can use comments to document that. And then, you can use tooling to generate a website that puts those documents on the web.

And I do think that's a good co-location thing, like, you want to co-locate that documentation with the method that it is referencing. That would be easier, to me, than trying to maintain two separate pieces of information like a...what are those systems called? Content management system. Like, instead of having a content management system that you manage all the docs in and then separately [inaudible 21:12] on the code, I think if you can program the CMS with the comments, then you're ahead of the game.

JARED: Yeah, I mean, I will admit that one of the most useful places I found comments is for code that is complex due to the optimizations or performance work that was done on it. There is, in fact, a much simpler way of doing this thing. However, due to the fact that the simpler way is unfortunately quite slow, we had to do weird stuff. And the comments explain the weird stuff, and it's usually in code that doesn't change frequently as well. If you're, you know, doing that kind of performance optimization on it, it's probably because the code itself is, you know, not changing frequently, or you wouldn't be locking it in into some particular --

ALAN: Yeah. In fact, you might even want to...I can see that case, and you might even want to, at that point, document it. For example, maybe you found a solution on the web that solves the performance problem, and you just want to reference the document that you found, so including the hyperlink to that document inside a comment with that method, I think, that's reasonable. And explaining maybe the algorithmic complexity might also be a useful thing. That's not why we made that decision necessarily. And, in theory, you could put that in the commit message as well, and maybe it belongs in both places. Like, here's the story of why this method looks like this.

JARED: So, we've actually kind of talked about where we put information, whether that's in method names or in, ideally, not comments or commit messages. We haven't really talked about, for example, pull requests as a spot where you communicate information. Do you have an overarching idea of, someone needs to know about something about this code. How do I decide where to put that information?

ALAN: Yeah, I mean, we kind of talked about it a little bit. I want the code to communicate what the computer should do. I want the tests to communicate what the computer should do. I want

both of them to be human-readable. And I actually have some thoughts on what that means for code versus what that means for tests. As far as the pull request goes, I think there are several things you can cover there.

It's always handy to have a link back to the ticket, you know, some kind of reference that this is the thing that we were implementing. It's useful also to explain, like we talked about, decisions around the code that describe why we're doing this. A lot of the time, here's the user story, like, the customer wants this so that they can do that. So, having the user story in there is, I think, either a link to a Jira ticket or whatever tracking system you're using. Or if you're using three-by-five cards, just type the story into the commit message.

One format that I've used that I personally like is I'll use all caps, I'll say, STORY: and then type the story, and then any notes about what, you know, anything special that's going on in that code, how I tested the code. And then, finally, for other kinds of tasks like migration, I might have a separate migration commit that just says MIGRATION and what it does.

It might have some kind of chore where I'm cleaning something up or it's a refactor maybe. And so, I'll use those keywords at the beginning of the commit message and then describe that story or refactor or whatever briefly. And then, in a separate paragraph, like a follow one, have the...I really want that first line to fit in a one-line git log so that I can quickly see what I'm doing.

But the goal of that is to be able to quickly suss out, okay, this is where we added that functionality. This is where we refactored it into, I don't know, refactored an if statement into polymorphism or something. And so, being able to clearly see that quickly that's the kind of thing I want to see in a commit message.

And then, if there's some reason, like, something that's not explained by that, for example, we refactored this to polymorphism, maybe that's not clear to somebody why we did that, and so maybe having some explanation of that would also go in that commit message. And, to your point, the algorithmic complexity, maybe there's some like, here's why we had to implement the complex version of this algorithm. It was so that, you know, we're not running an n+1 or, you know, some other performance issue.

JARED: Yeah. It's interesting. I feel like a lot of it comes down to, for any given sort of aspect of the code that you want documented, you have to think about when someone might need to know this. And are they going to need to know it to review this code? Well, then it should probably be in the PR description or potentially, at minimum, it needs to be in the commits. But, you know, some information is really only relevant to reviewers. It can be ephemeral. It's not going to be especially useful in a commit message.

But yeah, it's something, you know, we work...I run an agency, and we work on a lot of long-lived projects, projects that have been around for quite a long time and have often changed hands many times, and those, even the ticketing systems sometimes are gone. And we open up

commit messages and it says, “Oh, this solves ticket number this.” And it's like well, huh, wonder what that was.

ALAN: [laughs]

JARED: Too bad they don't use that tool anymore. So, I have this sort of...because of that, I try to put so much in commit messages just because, like, at this point, everyone's just using Git. The repositories don't go away, even if they change hands. So, that information is, you know, about as permanent as the codebase itself for most organizations.

ALAN: Yeah, that's a really good point. I'm working on an application that's been around for 18 years, 19 years now. It started in 2006. And you can see in the early commit messages a lot of, you know, fix this bug, 1, 2, 3, 4, or [chuckles] WTF. I've seen commit messages that just say, “WTF,” and that cracks me up when I see it, but I'm also really disappointed in the engineer [laughs]. Okay, you were confused, but explain to us why you were confused. And then, at some point, and I don't know exactly when this transition happens, somebody brought in a very rigid, not rigid, a more robust commit message culture.

When I first got here, and I was looking at the commits that my team was submitting, I was just super, super impressed with how much context they were sharing in the commit messages. But I've done some spelunking through the codebase, and it was not always like that. But you can instill that kind of culture into an organization, because once you see a good commit message, it's, like, hard to unsee. Like, oh, that's what they're supposed to look like. But if all you've ever seen are the one-liners, then that's what you're going to do.

JARED: Yeah, I was lucky enough...well, I don't know if lucky...lucky in the long run certainly, to very early on in my career work on a project that used Gerrit, the code review tool. Are you familiar with Gerrit?

ALAN: Yes, I am.

JARED: I wouldn't say it's pleasant to use, but the...for those that are unfamiliar with Gerrit, it's a project out of Google, I believe. And you push your commits to it, and it identifies each commit as what it calls a change or a change set or something. I can’t remember.

ALAN: Change set, yeah.

JARED: Yeah. And gives it an ID. And then, when you amend that commit in the future, you can see the changes to that particular commit over time. And the way we were using it is, you know, we were pushing individual commits, and they had to be, you know, atomic commits that passed all the CI, and you had to, you know, explain them fully. And they were going to get reviewed individually and approved individually and merged into the repository individually, though it does support branches and stuff.

And that just sort of enforced a very rigorous commit by commit. All the information has to be there, and it has to be atomic. It has to all work. It has to pass tests. You have to move the project forward incrementally and without breaking things. And while I don't think anyone wants to use Gerrit anymore, I did find that workflow very sort of formative and helpful in how I work.

ALAN: Yeah. I mean, a little secret; we use Gerrit at Cisco Meraki.

JARED: [laughs]

ALAN: And it's not my favorite tool, but we use it very similarly to that. Commits must be atomic. They must pass tests. There's no reason to make the, you know, the master branch or the main branch red. It should always be green. As with anything human, it's not always going to be the case, but the goal is to keep that thing green.

And yeah, there's also, I think, a discipline in telling a story with your commits. If you're using small commits and you've got a large project that you're trying to do, being able to organize the commits in a way that tells the story of that feature, or that bug fix, or whatever it is. Years and years ago, I was living in Seattle. I was going to the Seattle XP Users Group, and, occasionally, Ward Cunningham of Wiki fame would join us. He lived in Portland at the time, and so he would come up occasionally.

And somebody was asking him one time, “How do you convince management to let you do the refactor that you know you need to do before you add the feature?” And his answer was so beautiful. He said, “I don't. I just tell them, in order to make room for that feature in the code, I need this much time, and then it'll take me this much time to add the feature. So, I give them the two estimates.” But then he would then also structure his commits in a way that told that story.

So, if there was refactoring upfront, he would talk about, I'm doing this in order to make room for this other thing. And those commits would be separate from the commit that actually adds the feature later. And, you know, to tie this back to the coffee machine, when we're introducing that complexity upfront, those are individual commits that add a user story. We're going to add coffee to the machine. We're going to add tea. We’re going to add cocoa. And there's some refactoring that goes on in there to remove duplication in a way that we probably shouldn't. We're, like, over-drying the code.

And Dave Thomas would shoot me for using that definition of DRY because it is not his definition of DRY. But that seems to be the colloquial definition of DRY, like, just remove all duplication. So, anyway, we do that, and we treat each one of those things as a separate commit. And we actually built the code for this thing. In the codebase, we have all those separate commits laid out. And then, when we go to start the process of untangling it, each one of those refactor commits are a separate commit that move us forward incrementally. And also, they're atomic, and we can ship them, and the system's still working. Everything's fine, so yeah.

JARED: So, as projects grow and complexity sneaks in, changes become painful. What's your key message for developers who want to, you know, fight back against the tide of complexity in their codebases?

ALAN: Complexity is inevitable, but you can address it. You don't have to live with it, even when you're working under tight deadlines. To reference Ward again, Ward's the guy who came up with the technical debt metaphor. And, you know, you think about it in terms of we are purposefully taking on debt in order to ship something on time or to get a feature out quickly so that we can win in the market. Great. But there are costs to that debt. There's interest that you're paying. Every time you touch that code, it slows you down.

So, as you're paying that debt, every time you pay it, think about, is now the time? Is now the right time to just reduce that principal a little bit so that I can not pay as much interest the next time I touch this code? In the talk, and I don't think we actually got this far in the blog series, but in the talk, which is called “A Brewer's Guide to Filtering Out Complexity and Churn”...we've given it several times. The most recent recorded version of it, I think, was at Rocky Mountain Ruby last year. So in that talk, we introduced this concept we call rehydration because a lot of times the problem is code that is overly deduplicated.

So, what we want to do is we want to add back the duplication, that's the concept we call rehydration, so that we can see the missing abstractions. Because a lot of times, that's what the problem is, is that you didn't take time to introduce the abstraction that would have simplified the code, made it easier to understand and easier to change the next time you had to touch it. And that's really the issue. If you find yourself struggling to introduce a change, there's probably a missing abstraction. And one way you can find it is rehydration, if the problem is overly DRY code.

And so, my main message there is know that you can have an effect on the code. You don't have to live with code that is hard to change. In that particular situation with that coffee machine, where we've got the conditionals and the overly DRY code, first thing we do is rehydrate the code. Now we've got this big, giant long conditional with a bunch of duplication in it. But now it's clear that there's a recipe for brewing coffee, and there's a recipe for steeping tea, and there's a recipe for mixing cocoa. And we can pull those things out into polymorphic classes. And now, when we go to add apple cider...or, I'm not going to spoil it. The talk, there's a joke in the talk. I'm not going to spoil it.

JARED: [chuckles]

ALAN: Go watch it. When we go to add apple cider, it's just a matter of adding another polymorphic class. You don't have to change the main algorithm at all. So, it makes adding and extending your code really easy. And, in that particular case, it separates those recipes, those brewing algorithms in a way that makes them easy to change if you find a bug in one of them. And you know you're not going to introduce another bug inadvertently in a different recipe because you've extracted that recipe into a polymorphic class.

JARED: Well, it has been a pleasure having you on the podcast. Where can people go to follow you online?

ALAN: Let's see. I am...the.codegardener.com is our blog. I am alan@codegardener...is it @code? Yeah, I think it's alan@codegardener.com on Bluesky. I'm on LinkedIn. Yeah, that's about it.

JARED: Awesome. Well, thanks for coming on.

ALAN: Thanks, Jared. Thanks for having me.

JARED: I am very interested that Cisco Meraki uses Gerrit. I'm torn on it. Like, it forces a way of working that I think is really, really good, but it also is horrible to use. So, those, like, I would pay money for and, honestly, have thought about trying to build a good alternative to Gerrit.

But it was awesome talking with Alan. You know, we covered a ton of different topics: complexity, commit messages, a whole bunch of stuff in there. Really, really wise words from him. We'll definitely have to have him on again in the future to talk about the other things.

I think, spoilers: we are thinking of doing an episode about flaky tests with a bunch of different people, including him. So, stay tuned for that one.

This episode was produced and edited by Mandy Moore.

Now go delete some...