EPISODE 1383

[INTRODUCTION]

[00:00:00] KP: Thanks to the amazing books, blogs, videos, quick starts frameworks and other software related resources. Getting Started as a software engineer is easier than ever. Although you can get started in a day, it can take years to become a master of the craft. And most practitioners describe it as a profession of lifelong learning.

Titus Winters is a Senior Staff Software Engineer at Google and the author of the book *Software Engineering at Google*, often known as the flamingo book. This book is not just tips for structuring, writing and testing code, although that's there for sure. It's a resource that outlines all of the facets of software engineering practices that apply in professional settings through the lens of lessons learned at Google.

[INTERVIEW]

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

[00:00:49] TW: Thanks for having me.

[00:00:50] KP: So I think you might be most famous for being one of the authors of the flamingo book. Maybe there's another credit that shines higher sometimes. But for listeners who don't know it, what is the flamingo book?

[00:01:02] TW: This is software engineering at Google from O'Reilly. We put that out. Great timing. It was late March 2020. There was nothing else going on then, let me tell you. But, no. This was a joint project sort of following in the theme and style of the SRE books. But whereas SRE was itself fairly novel, software engineering isn't necessarily. But at the same time, Google has been. Let's say, we have some scale experience that isn't common for others. So we've solved a few problems and encountered some things that we thought would be useful to share with the rest of the world.

[00:01:43] KP: Absolutely. Software engineering at Google, as you said, there are problems that Google's tackled that I'm sure they were the first to tackle many times over. How has that shaped the overall process and the ways in which Google approaches software?

[00:01:58] TW: So I think that the commonalities that we keep finding are that it's really a matter of teamwork and communication, and making sure that our processes sort of scale, which really isn't a given. There's plenty of software engineering sort of norms that kind of don't scale once you get above a certain threshold. But then it's also a lot about planning for time, and maintenance, and change. I say fairly regularly, things like software engineering is programming integrated over time. Like these are different dimensionality. I don't believe that change for change's sake is good. But I do believe that over the expected lifespan of your code, of your project, you need to be capable of changing. And it's often wise to practice. And the places where we've, I think, had the most success have been in the places where we really kind of planned for how are we going to change this effectively in the future. And that's been a little novel, in my experience.

[00:03:11] KP: I think this is a point a lot of people even senior software engineers sometimes struggle with. It seems that something about source code feels like you're writing it in stone. Once you've got it working, it's this mathematical object that's just perfect. Why is that incorrect?

[00:03:27] TW: Because the complexity of everything that we're building is just so intense. And it's rarely the case that we have, a perfect mathematical proof of this is correct, much less optimal. And even in the cases where you have like an idea that, "Oh, this is theoretically optimal," there's still going to be changes in hardware, and languages, and dependencies underneath you. And over time, given enough time, something is going to change.

I lean a lot on examples from like security incidents and vulnerabilities because those are clearly very high stakes. Say, the speculative execution vulnerabilities, the Spectre and Meltdown that was making all the headlines a few years ago, and that are still very much a real thing. I think that for mitigating those sorts of things, by enlarge, you kind of just have to recompile. And I know that a lot of groups like aren't on a current compiler, or don't have access to the source code to recompile, right? And if you don't even have the ability to change the binary that you're operating on, then if we wind up with vulnerabilities in that binary blob that you're depending on,

now you have a choice, right? You either suffered the security vulnerability risks or you figure out how to change that thing.

And I think once you start like looking around at over 5 years or 10 years, most of the things you rely on are going to change in some fashion and probably in ways that you don't expect. That gives you a different sort of stance and footing for, like, what do you accept, and what are you going to try to plan for, and what are you going to practice?

[00:05:21] KP: Well, there was a time in my life when I was going to live somewhere for just a few months. So I said, “I'm not going to buy new furniture, decorate. This is temporary.” And maybe I've written some software like that where it's just to bridge a gap. But generally, I like working on software projects where I think I'm building for the next 1000 years or something like that, even though I know that can't be true. Do you have a good rule of thumb or way to think ahead? What am I building for when I take on a big project?

[00:05:48] TW: I don't have a rule of thumb for making that decision. Because I think it's very, very contextually dependent. I think it's very commonly the case. If you're working for a startup, you should probably assume that you're going to be around for six months or a year, right? You need to make it in the next round of funding. You don't need to plan a whole lot further out than that. If you become successful, then you can deal with changing the life expectancy of your project to that point.

I think it's sometimes the case that you can know ahead of time that the code you're about to produce is going to be years or decades. And a lot of the time, you're just writing like a throwaway little shell script or something, and you're going to delete it in 10 minutes, or never run it again, right?

And one of the ideas that I lean on a lot, and that's very present in the first chapter in the book, is that there's six orders of magnitude of reasonable answer to how long is this code going to live, right? Somewhere between seconds and decades. And it would be absolutely bananas if we actually thought that the same best practices applied on both ends of that spectrum. And the people that are really zealous about, “No, you have to write tests and have code review even for

those little shell scripts.” Like, I don't know if that's true. Like maybe, but like I wouldn't fight anyone over that.

Whereas on the very long end of the spectrum, like plan for change. Yes, write tests. Yes, have code reviews. Like you need a different perspective. And then between those points, you have this kind of gradient. I would recommend people probably plan for a little bit longer than you expect. Very much like your example of, I'm not going to buy furniture. It's really easy to underestimate how long something's going to live. But still, I think the most important thing is ask the question, how long is this going to live?

[00:07:53] KP: Yeah, makes sense. As an engineer, maybe even a recent hire, where do I look to? Obviously, I should get some direction from my manager. But are there any lessons or ways I should be thinking about my code and trying to figure out my role within the org?

[00:08:08] TW: I think that this is the thing that we are still very much trying to figure out. One of my side hustles right now is I'm sitting on an ACM IEEE task force for writing the next generation of computing curriculum guidelines. Specifically, I'm in charge of the software engineering parts, but I'm consulting on a few others. And this is really reinforced for me like what we teach our undergrads currently and what we can kind of assume from a new grad hire, for instance.

And I would argue that in almost all cases, our new grads are trained as programmers, right? They can write some code. They're probably not expert at that. But they can write some code. They're competent in that. And I don't think as an industry we do a really great job of telling them when we're interviewing, or hiring, or when they're onboarding. Hey, your job is not actually to sit down and write a bunch of code. Your job is to solve some problems. You might do that with code.

I like the comparison to aeronautical engineers, right? The job is not build things out of as much titanium as you can. The job is get people safely from point A to point B at a reasonable expense. You happen to do that with aluminum and titanium. And similarly, like the software engineer’s job is solve problems, right? All sorts of problems. Tech is good at addressing problems, sometimes causing problems too. But we don't tell people like, “Hey, your job is not actually measured in how much code did you churn out?” There're more important things. And it

will be I'm working on trying to make it clearer that our new grads, “Hey, that's not actually the job.”

[00:10:06] KP: I really appreciate your point about that gradient, from the way Google writes code, to some startup that is literally writing code to stay alive on a day-to-day basis. What are some of the most important lessons? If I'm in that role and I'm in a three-person team, and every day is a new fire to put out? Maybe I'm dreaming of Google scale for the venture I'm involved in, but I'm not there yet. What are some of the most important lessons I should take away and keep in mind even though I'm at the opposite end of the gradient right now?

[00:10:36] TW: I think it's important to still be aware of like which of your practices are going to work in the long term, and where your code has surprising sort of technical debts. And in both cases, keeping kind of an inventory of what you think is going to need work eventually is probably wise. So in the policy space, for instance, I still encounter plenty of groups that seem to think it's a really good idea to have dev branches that live a long time.

And I think there's been plenty of research from groups like the DevOps Research Association, as well as just tons of anecdote that long-lived dev branches are kind of counterproductive to good productive outcomes. And you can probably make it work when you have a small organization. But as you start growing, you have more teams and more people and you start having choices of what version of things to depend on. You're just adding choices, and collaboration points, and questions. And that's just a lot of overhead. And that doesn't seem to serve as well.

And similarly, things like who gets to merge next become kind of dominant and kind of annoying questions for an organization that started in that fashion and hasn't gotten themselves out of that habit. And that said, like it may be a reasonable place to start. I'm not really capable of judging the efficacy of that pattern for a three-person project or for a five-person project. But it certainly doesn't work for 50 people or 100 people, right?

So like keeping an inventory of, “I don't think this is going to be the right policy for us in the long term,” seems wise. And similarly, like your technical debts and dependencies of, “Hey, we

started writing this in Python several years ago. And we really should do the Python 2 to 3 upgrade, right?"

[00:12:42] KP: Yes. About time.

[00:12:44] TW: If your startup is facing bankruptcy or folding in the next couple weeks, I wouldn't worry about that, right? If it's six months out, maybe, right? If you're sure that you have a year's worth of runway, like maybe this is a really good time to start, like paying those things off. And I think all of that maybe summarizes as you just kind of need to have an awareness of what your exposures and hits to productivity are going to look like. The SRE's always say you can't run a reliable service in production without monitoring. And I think there's a reasonable corollary to that for the parts that are earlier in the workflow. You need to kind of be aware of what your risks are.

[00:13:35] KP: Well, to full stick with our hypothetical three-person startup for the moment, and imagine that you 18 months later they've grown. Tow they're 50 people and something has to change for sure. A first step, I could buy 50 copies of the book and hand it out to everybody and hope they kind of self-organized. But I imagine there's a little bit more to it than that. Can you talk a little bit about how to develop a good culture and good teams to make the system all work together?

[00:14:00] TW: So it is not an accident that the first chapters in the book are about culture, because I very firmly believe that environment team cultures that are supportive and collaborative are going to lead to better outcomes in the end. And I've seen this on my own teams. I work on very low-level C++ library infrastructure for the most part. So projects like Abseil were things that came out of my group. And what I saw there when we made a conscious effort in fostering positive and supportive team culture was that we got so, so, so much more done, because we specifically made it a norm that it is perfectly okay to not know everything, which should be obvious, because software is really complicated. It's perfectly okay to make a novel mistake. It's perfectly okay to ask for help.

And when we really focused on these ideas, we added a little ritual in our team meeting of kudos and the monkey. And the kudos part is just go around the room. Anyone that wants to

speak up can identify something good that other people on the team did for them, like helping out, answering questions, shouldering the pager, whatever. And then the monkey is sort of a jokey competition for who made the biggest most novel mistake this week.

And by normalizing it, by making it kind of a team shared experience, it makes it clear that it is okay. Like we're not aiming for perfection. We're aiming for learning from our mistakes. And by airing those publicly, you see that the rest of the team doesn't jump on you. They support you. They laugh. They learn from it. And we don't make the same mistake twice.

And by doing these two things together, we're like building bonds of respect amongst the team. And also fostering trust more aggressively in terms of, “Hey, it's going to be okay if I didn't get this right the first time, right? If I learned a new thing.” And none of this is the same as like I'm not saying that it's unacceptable to point out that someone made a mistake, right? It's just not about blame. It's about learning, right?

And the more that we have been able to foster and replicate that sort of experience, wow, those teams get an awful, awful lot done. Because you're not worried about people betraying you or stabbing you in the back. Everyone is there to catch you if you fall, and you move a lot faster in that environment.

[00:16:57] KP: One of the things that was interesting in Jeff Meyerson’s book on Facebook was their onboarding process. They hire people. And you kind of first become a Facebook engineer, and then you go to this kind of training camp. And from there, you are not applying for teams, but you end up joining some team through that process. Could you talk a little bit about the similar or compare and contrast the differences of what onboarding is like at Google?

[00:17:22] TW: In a lot of respect, I like the Facebook model. For Google it is, for software engineers, if you pass the interview bar, you get through a hiring committee. Then you'll probably be shopped around with some teams that have openings in whatever offices you are potentially going to join up in. And so there's a little bit of a matching process that goes on there. But by and large, it's mostly done before you actually join up. And so you're sort of making those decisions based on resume and describing what the project looks like, and a couple sort of fit calls, those sorts of things.

And so in some respects, I like the Facebook model better in terms of you get a more hands-on understanding of what that team might be like and what that work is going to be like. And I think that's maybe safer, healthier in the long run. In some respects, there certainly have been plenty of cases where an initial team fit at Google was not quite right. And that's hurtful for the new hire, and hurtful for the team, and hurtful for the organization. It would be much better if we had ways around that.

[00:18:40] KP: What's right point in my journey as a software engineer for me to pick up the book?

[00:18:46] TW: I would like to think that there's no wrong time for that. And honestly, I'm spotting more and more undergraduate courses on software engineering that are picking up the book as either primary text or additional reading. This probably helped by the fact that the book is now freely available. The PDF is just readily out there now. But I think presenting sort of a theoretical understanding of like why one version control policy is going to be more effective than another or why dependency management is not a hard problem for a programming project, but is an existentially difficult problem for a long-lived software engineering project. Those are great topics that I could easily imagine being discussed at length in an undergraduate lecture, those sorts of things.

I think for people that are new engineers, reading through the topics on culture, and teamwork, and even code review, those are very valuable sorts of things. And if you are a leader and making the technical decisions, making the policy choices for your team, I think that there's still a lot of wisdom hidden in the book on various topics. I don't think you have to read through it cover to cover. I think you can kind of poke at what topics feel most relevant. But it should hopefully be useful and informative at all of those points in the career.

[00:20:25] KP: Well, dependency management is one of those things. I'm sure there's someone out there, right? As you say, at scale, everyone's line of code is important to someone. But by and large, software engineers don't particularly like managing or fixing dependency things. We call it DLL hell, or library nightmare and stuff like that. Are there things I can do to maybe better

equip my code for the next person that's going to have to worry about it when I've moved along to something else?

[00:20:51] TW: Tests. Just write tests. Write good tests. That's by far the most important part. I wrote the dependency management chapter myself. And from my perspective, most of what we do is not actually working, certainly not at scale. And I think it's kind of wild that we're still relying on, say, SemVer, to suggest, “Hey, are these things going to be compatible together?” Instead of actually run the tests, “Are they compatible?” right?

I think from a from an abstract perspective, it is obvious that SemVer is a very lossy, like human attestation of how compatible do I think this is? Which is not in any way, shape, or form a proof. Running the tests is not proof, but it's a heck of a lot closer. And we would be – I suspect, we would have a whole lot less DLL hell, as you describe it, if we relied more on evidence and less on estimate.

[00:22:00] KP: I feel like tooling has come a good distance in this, the days when I was learning to programming, or learning to program and downloading stuff, and copying things out of magazines. Everything felt very nightmarish then. Although, today, I can sometimes go clone a repo, type npm i, and everything's up. It just kind of automatically works for me. So we've certainly made a lot of advancements as an industry as software engineers. What are some of the major gaps you think that could be closed in the future?

[00:22:30] TW: So I think that the ecosystems that are working like Rust and Node, like you just mentioned, both have some ability to query against the public like dependency tree of, “Hey, if I commit this change, isn't going to break people?” right? And that's kind of key.

When we focus entirely on backwards compatibility. In a theoretical sense, we're missing all of the nuance. And like it is entirely backwards compatible to remove a thing that you're sure no one is using. But that really points out the fact that compatibility is not a property. It's a property of a relationship, right? You can only really evaluate compatibility in any context of how it has been used. And we're getting better at that.

I'm thrilled to see legitimate language dependency ecosystems that are building up that actually, like, account for that sort of thing. But I think some of the places like C++'s lack of cogent package ecosystem is really challenging. We're making some progress. VC package is very nice, for instance. Conan is not bad.

But by and large, adding a dependency in C++ is still a thing that people will like get up in arms about. In Python, the PIP ecosystem scares me a ton. Because for pure Python packages, okay, it's probably fine. For packages that are built on top of anything lower level, they've fixed the tool chain. They fixed the libc version to something from basically 10 years ago. And so like a ton of energy and effort is being worked or is being devoted to trying to work around like version in compatibility things, because they didn't plan for change. Time, change, it's at the bottom of all of the worst problems.

[00:24:40] KP: Absolutely. One of the things scalability brings you is sometimes problems you didn't know you had. I'm trying to think of if I have a real world example, but like maybe in the backyard when I don't take care of the plants. I'm like, "Wow! This is really grown and gotten out of control in a way I didn't expect." Are there things like that that Google has learned its lesson on or maybe had some foresight on that are interesting talking points about what it is to be a software engineer?

[00:25:05] TW: Yeah, I think so. So two that come to mind, the very simple like early experience working on a team, you might put out a notice to everyone on the team saying, "Hey, I'm about to land this big refactoring." No one commit for the next few hours or days. That's fine when you're very small, but isn't actually going to work as you get bigger. And that starts nudging you towards you're going to probably want to find a different way to do that refactoring.

And then at our scale, one of the things that people often find surprising is you're certainly not going to rely on your in-IDE refactoring tools to make codebase-wide changes. That just doesn't scale that way. The public statistics on, say, the C++ code base here, I've got 12,000 developers that will commit a change in my codebase this month, and it's somewhere north of a quarter of a billion lines of code. Like I can't load that in my IDE and do a find-replace, right? That's not going to do it.

And even if I could, I wouldn't want to, because if I'm going to make changes to 50,000 files, it takes so long to sync that change like from upstream. That by the time I'm done with the sync operation, it's very likely that some file in that set has already been changed, because 10s of 1000s of engineers, right?

And so there comes a point where you get so large that the changes that you might want to make to your most common vocabulary sorts of interfaces, you can't make that in one step anymore. Because you literally can't sync to head to commit that change before someone has changed something out from under you.

And even if you could, the question of, “Can I test this? Can I roll it back if something goes wrong?” These things actually start to be very concerning. And so you wind up having to have entirely different approaches to how you do refactoring at scale. So the chapter in the book on large scale changes goes into kind of some of the theory and the practice around all of that. And people years ago would have been shocked at the amount of like just kind of background cleanup churn that we go through. But we find that it's been really important to shake out bugs and brittleness and clean up old technical debts. These days, changing 10,000 files is kind of a background task over the course of a week. It's just we got used to it. It's not a big deal anymore.

[00:27:50] KP: Well, knowing that you had these challenges, and also knowing that Google is a mono repo will seem kind of confusing to me and compatible. Why still be a mono repo?

[00:28:01] TW: Again, it becomes a question of choice, I think. And it's not so much that everything is checked into one repo that matters. So much as some of the properties of that that matter. And for instance, I would summarize it as you need to not have a choice of what version you depend on. You need to not have a choice of where you commit. And you need to not have a choice of what you are syncing to.

So as long as you have a consistent version of what is at head, and as long as like this is the version of this that I depend on, then you'll be fine. But you're effectively building the properties of like a virtual mono repo. And I know that it's surprising. But it turns out that when you have this sort of centralized and I have visibility across all of the users, yes, it is more work for me

when I want to make that change. But when I look at it as is it more work for Google? If I centrally discovered like, “Okay, this change needs to be made. I'm changing from foo to bar in some interface.”

If I look at all of the uses of that, and I find the common patterns, and I build tools to do that, because I'll drive myself crazy if I don't have tools, then through centralization and automation, I can probably just go into everybody's code and make that changed for them. If it's not changing behavior, then this is safe. And if it's a modest change to behavior that we can generally reason about, then you can usually do this with a little bit of static reasoning, and you go run the tests.

On the whole, this costs me a lot, but it's very cheap for Google as a whole. Whereas if we fractured everything into individual, smaller projects and smaller repos, then someone in each of those repos needs to be asking the question of, “Hey, which version of things am I depending on?” “Hey, for that version, when I do an upgrade, is there an incompatibility that I need to take into consideration? What is that change?” “Oh, you're changing foo to bar? How do I do that.” And then you can go track that down and you do it once kind of without experience and with a little hesitancy. And then you commit it and you move on. And you multiply that by hundreds or thousands of teams. I can tell you which of those is cheaper for an organization and in the aggregate, right? And that's, I think – To my mind, that's the actual argument for mono repo, is it's just fewer, less hidden, less unknown, fewer choices, and more opportunity to benefit from economies of scale.

[00:30:55] KP: When it comes to testing, I see a lot of advantages and that everything's in one place. And I'm not going to name names here. But I've seen an anti-pattern at a lot of companies where, initially, they have good commitment to unit testing. A couple of developers have really written some good unit tests, and maybe some good integration tests. They grow and grow and grow. And these tests start taking longer and longer. And at some point, everyone just slips into using this ignore all tests flag or something like that. And it's like everything's lost all at once because it outgrew itself. Google didn't have that problem. And you've grown bigger than, arguably, any other company. How can testing still be prominent when those challenges exist?

[00:31:32] TW: So we're still learning. I won't say that I have all of the answers here. But we've been pretty conscious, I think, about keeping a separation between things that we want to run pre-submit before your PR is merged, and things that we can run post-submit. And the post-submit work, for instance, is gated not by we're going to run all of the tests for every change. But we have kind of a fixed budget for how often are we going to run. Or, yeah, how often are we going to kick off the tests. And that may mean that there's a range of changes that landed that something in there broke, we'll have to figure it out. But that separation and that awareness that like, “No, we're not going to run all of the tests for every change.” I think that was bias.

The other thing that I'm increasingly seeing us move towards is I think we need to have a better and deeper understanding of how sort of non-binary tests fit into the continuous integration workflow. So I've been thinking a lot about like how does – How do we incorporate fuzzing? And how do we incorporate random tests, like actual tests of randomness, for instance, into the workflow? Because our whole model on CI systems is, “Hey, I ran all the tests. It's green. So I submit.” But at scale, that doesn't actually work out because some amount of tests are flaky. And some amount of things are just not actually a red-green signal, right? It's kind of a, “Well, I didn't spot anything. But if I run longer, maybe I would find something.”

And what I'm increasingly seeing is there's a pretty solid comparison to be made between the model that we use for thinking about continuous integration, versus the model that we use for like monitoring in production. You're going to have some, like, alerts that trigger when unusual events occur. And those probably require an SRE gets paged, they need to go investigate. Some of the time that is actually indicative of there're about to be a problem. And some of the time it's a, “No. The alert was wrong. I'm going to silence the alert and move on with my life.”

And I see a lot of parallels between that and the way that I think CI at scale probably needs to be conceptualized. And once we start doing that, then it becomes more clear that CI is actually a signal processing problem. And most of the decision points are really things that computers are going to be better at, right? We need to look at the historical state of this test as a signal. Did I change the passing rate on it substantively? Is that actually indicative of predicting a failure in production? Or is it just a bad test, right? If it's a bad test, then I can silence it, whatever, right? But we need to sort of advance from a model of tests are proof and it's impossible to skip everything and we need to run everything all the time to tests are a signal’s proxy sort of

problem. Run the subset of those that are going to be fast enough to give you some confidence. And let the computers do the rest of the signal, like gathering and processing, to decide whether or not your change was actually good. And if it wasn't, then roll it back.

[00:35:07] KP: I think most people intuitively know that some sort of code review is a good thing. And I think a lot of people's experience is they were just thrown to the wolves. Just go have a code review. You and two people figure it out. Of course, we should have some lessons learned and process around that. Can you share a few tips? I know there's a lot in the book, but just some ideas about how to organize and participate in a good code review?

[00:35:29] TW: Yeah, we did some interesting studies on this and even published academic papers on it. And it turns out that, for all that we take code review very seriously, the primary benefits of code review are not usually in bug finding. It might be in inefficiency finding. But it's mostly in education and knowledge sharing, and it's a communication activity with your team. A code review is a really good chance if you're on a team, for instance, that doesn't rely on pair programming, extensively. Code review is probably the only time that someone else on your team is going to look at the code that you are producing in detail.

And that is a really great opportunity for both parties to learn more about the code, more about the libraries that you might be using, more about the accepted coding style, more about the programming language. And it's even nice from a sort of pedagogical perspective, because any lesson that might be transmitted during a code review is actually like highly relevant, right? “I was working on this. I did this thing that is maybe a mistake. Now we have a perfect opportunity to introduce, “Hey, have you read this like best practice write up?” “Oh, yeah, that does apply. Okay, let me go update things.” So like the educational aspect of that I think is really valuable.

The other thing that is really important and not obvious to most people is code review. Because it is a communication activity with your team, really, really needs to be very polite, right? You need to focus an awful lot on clear professional communication, right? Be very careful about how your written communication comes across, right? It's easy to sound kind of like an ass, whether you meant it or not. And since it is someone on your team, you probably don't want to piss them off or upset them. And that's not the same as you don't want to point out what's

wrong. But you don't want to make it a finger pointing and blame sort of exercise. You want to use it as a learning opportunity again.

And so the more that we focus on professional communication, and clarity there, and take advantage of those learning opportunities, the more that we wind up with code that several people on the team understand and everyone on the team has learned from. And that's kind of a great win.

[00:38:15] KP: Well, in addition to your role at Google and being an author, you're also heavily involved in the C++ world. Can you tell me a little bit about some of the ways?

[00:38:23] TW: So I spent a few years on the C++ Standards Committee. And for about three years, I chaired the working group for the Library Evolution. So that's roughly the design of things that changed in the standard library for C++. Between C++ 17 and C++ 20, most of those design changes went through the room that I was chairing. And, yeah, I've spent a lot of time trying to get the committee to adopt a sort of different stance on adapting to change and being able to learn from or fix its mistakes. That has been challenging to say the least.

So I think my committee involvement has dwindled a little bit. But I still am fairly active on like C++ Twitter, and in the C++ conference space. And internally, we rely a whole lot more on things like Abseil than on the standard library at this point. Because from what we can tell, the standard library is just not going to really prioritize the ability to change things. And therefore, they're not really prioritizing performance. And at Google scale, those performance needs kind of dominate a lot.

[00:39:40] KP: And what's Abseil?

[00:39:44] TW: So this is a project that started back when Google split into Alphabet. And we asked teams, we asked other companies that we're about to be split apart, “Hey, what are you going to miss when you're ejected from the Google codebase?” And they identified a lot of things, but it was kind of the high-level crown jewels, like AI processing sorts of things. But because they're all computationally-intensive, they were all built against the common libraries that my teams had owned for years.

And we had never really had the funding to really aggressively clean those things up. But if we wanted to share the good parts, we would either be asking these other bets to adopt our technical debts in order to use those good parts. Or we have to figure out a way to share the infrastructure and clean up the technical debt.

And my leadership was wise enough and forward-thinking enough to fund that effort. And so we had quite a number of engineers that were cleaning things up, resolving old refactoring problems, centralizing things, and then open sourcing that. And that open source is what we call Abseil. It's a lot of infrastructure pieces. It's things like mutex, and command line flags, as well as very high-performance hash containers, utility code. And this is the utility code that underpins effectively everything at Google.

I kind of joke, I don't think it's actually a joke, you probably can't get more than six feet away from code that I've touched without making a conscious effort. Because everything that Google has built has the Abseil dependencies in it. Well, not everything, everything, but almost everything. **[inaudible 00:41:36]** somewhere. So it's our sort of nuts and bolts, as it were.

[00:41:44] KP: if I understood correctly, you highlighted, I guess, prioritizing some performance improvements as one of the motivations. As a user, I guess someone like myself would just benefit from those without even knowing it. Are there interesting things going on in the C++ language or has it just kind of solidified given its longevity?

[00:42:04] TW: No. I mean, that to me wrong. The languages preceding the C++ 20 release is a very, very large set of features. My concern, and I think Google's concern, has largely been that things like binary stability is dominating performance concerns.

And so, for example, because of the default assumption from compiler vendors that they can't ask their users to rebuild, this means that the hash containers in the standard, depending on your compiler vendor, probably are maybe 10 years old. There're design problems in them as well that kind of contributes to the inefficiency. But we've done a staggering amount of hashing research in the last 10 years. And it's worth a pretty significant fraction of fleet performance to be able to update your hashing, at least if you have compute profiles like we do.

And when it comes to, “Hey, we can use the standard types, or we can refactor everything and do it ourselves.” And it will be worth some percentage of fleet performance to run our own hashes and run around hash containers. If the standard is unwilling to ask people to rebuild to adopt such changes, then I think we're just going to go our own way. And that plays out kind of over and over again. Like we like our interface for mutex better, and I think it's more efficient on most platforms. The hashing is wildly more efficient. And you get various aspects of that. Like the standard regular expression libraries are terrible. There's a joke that it may be faster to fork, spin-up PHP, and do your regular expressions then to use standard regex.

But at the same time, like I don't know how much of a joke it is, and because we won't even ask people to rebuild, that problem is never going to get fixed. Like at most, there'll be a std regex 2. And that's not an entirely satisfying answer. So it's hard. I can't say that prioritizing stability is the wrong choice. It's very good for not frustrating users. But performance does matter.

[00:44:28] KP: Absolutely. Well, I found software engineering at Google, lessons learned from programming over time to be a great resource. I think there's lessons to be learned for people at all stages in their software engineering career. Remind listeners where they can get the PDF or print copy.

[00:44:44] TW: Print copies at any fine retailers, certainly Amazon. The PDF, if you search for software engineering at Google, there is a link on abseil.io that has a PDF to download from there. And that is fully licensed, totally legit. Our goal with the whole project is to try to give back to the community and to provide some foreshadowing of the lessons that we had to learn the hard way. And hopefully, if your companies grow and you're successful, that you don't have to learn everything in quite as painful fashion as we did.

[00:45:21] KP: Absolutely. And Titus, where can people follow you online?

[00:45:25] TW: I'm on Twitter @tituswinters.

[00:45:28] KP: Sounds good. Well, thank you so much for taking time to come on Software Engineering Daily.

[00:45:33] TW: Thank you for having me. It's been a pleasure.

[END]

