EPISODE 1333

[INTRODUCTION]

[00:00:00] GM: Becoming a contributor to an existing software project can be a daunting task for an engineer. A common convention is to add a readme file to your repository to serve as a trailhead, which gives new visitors step by step instructions for running, exploring and understanding the structure of the code base. *The Missing README* is the recently published book which prepares new software engineers to both survive and succeed. Learning to code in school in a boot camp or independently can prepare you to write software. This book prepares you to do it effectively in a professional setting. I speak with authors Chris Riccomini and Dmitriy Ryaboy about *The Missing README: A Guide for New Software Engineers*.

[INTERVIEW]

[00:00:44] GM: Dmitriy and Chris, welcome to Software Engineering Daily.

[00:00:48] CR: Thanks for having us.

[00:00:50] GM: Pleasure to have you both here. Well, you guys have written a recent book called *The Missing README*. For senior engineers, I think the title will be very evocative of its contents. For a junior engineer who doesn't yet know about README, what is this reference manual?

[00:01:05] CR: So the manual covers kind of a few different subject areas, all of which we think are applicable to software engineers that are just entering the workforce and, probably, for engineers a few years beyond that. The book is kind of broken down into some technical stuff that you don't generally learn in like a new college curriculum. So this is like stuff around continuous integration and deployment, operations, on call, metrics and monitoring, that kind of stuff, as well as test and design. And then it also covers some of the – I'm hesitant to say softer skills, but maybe some of the stuff that's more about how to exist and work in an actual company. So working with your manager. How to learn? How to learn from more senior

engineers? How to get involved in different engineering programs? Stuff like that. So that's sort of a broad stroke at what's inside the book.

[00:01:58] GM: Makes sense. The book pretty much tells the story that a career is a journey. You're always learning on that journey. What's the best point at which someone should pick up this book? During their education? When they're looking for their first job? Or, yeah, I guess where do I begin picking up the missing README?

[00:02:14] DR: Yeah, I think we wrote it with a character in mind who would be just starting their first engineering job and experiencing that transition from sort of learning how to program at a boot camp or an undergrad program, and learning kind of the theory and being able to program by themselves. And then suddenly being part of a team and having to work on code that's owned by a team, and possibly even written by people who are no longer in the company. Now they have managers and their processes. And they're sort of encountering all of that and thinking, “Well, what is all this?” How do I engage with it most productively? What are the sort of unseen things that maybe I should be doing?” Part of the inspiration for writing the book was that both Chris and I independently have repeatedly had this experience with new folks joining our teams. And it turns out, there's a lot of knowledge that software engineers learn sort of by osmosis from each other just culturally. It isn't really taught in schools and properly so, because it's not really computer science. It's not theory. It's just kind of the practice, right? It's just when you work at a company, this is how things go. But because it's sort of a set of skills that's not really formalized, most people come in, and they don't know it, and then they aren’t really officially taught it and they just sort of learn it by stumbling about and somebody saying, “Oh, hey, don't do that. And did you remember to put in the logging?” And they go, “What's logging?” that sort of thing. So we're just trying to accelerate that process for people and have a book that explains why things are the way they are. We also encounter a lot of folks learn these things by, “Well, Joe Bob is doing it this way. So I'll do it what Joe Bob does.” But maybe Joe Bob has a reason for doing that. And there is kind of a theory behind it. But over time and over multiple generations of this sort of thing heading off, a generation being about two years in Silicon Valley.

The reason certain practices are in place gets lost and people start just doing them possibly slightly off and not knowing why and just kind of knowing by word of mouth that that's what you're supposed to do. We try to uncover some of that in the book. So that the natural reaction

of a person who goes, “What is all this and why is it happening?” Or I guess I just have to do it that way. And they don't understand why. And so they don't do it quite right. We want to address that core. And that's why I think also the book is fairly widely applicable, because we tried to explain the reasons behind different processes and different practices rather than specifically say, “This is how you do this thing and go. This is how you do this thing in JavaScript. This is how you do that thing in Python. Or if you follow this particular methodology, we're going to explain the thinking behind these processes.” So that wherever new engineers find themselves, they can read that and understand how it applies to whatever particular process of practice their team has.

[00:05:01] GM: I did a fairly traditional University program and got out of academia, got my first software engineering role. And one of the things I was surprised by, I had no exposure to version control. So I had to learn that on the job, probably a pretty common experience for a lot of people. And after being on it for a few days, I was thinking, “Why hadn't I started this way? This is this big gap in my knowledge.” Do either of you have any personal stories along those lines of things you were surprised you didn't know when you began your more professional aspects of your career?

[00:05:32] DR: Oh, I think both of us do. I'll tell mine really briefly. Chris has an extra fun one that's in the book. So one of my first tasks, possibly like the first real task at my first sort of real .com Silicon Valley company job was actually to figure out how to put my team's source code into some sort of version control down to, I'm aging myself, having to choose between like should we use SVN or CVS. So I definitely went through that transition, because when I graduated, source control existed, but it wasn't quite as ubiquitous as it is today.

I think it's less of a thing now. I know a lot of universities have people coming in to get a lot of the boot camp programs do. But there are other problems. So there's version controls and checking in your code. And then there is versioning your code and like dealing with version numbers of libraries and things like that. And we talk about that a lot in the book. But I'll let Chris talk about his source code story.

[00:06:33] CR: Yeah. Yeah. I was just going to dive in and say I had a very similar experience where, when I joined, I wasn't really familiar with version control at all. And this is back in the

day, and we were using a system called CVS. And so, much like most people still actually interact with their version control system. I was pretty much just like copying and pasting commands off of a document that described how to branch, and merge, and tag and all that kind of stuff, and managed to corrupt the entire code base for the company unwittingly, and ended up going home and coming back the next day. And a chunk of the senior engineers had been there all night trying to recover the code. So that was very early in my career. But a good lesson learned on the importance of understanding the tooling that you're working with.

And like Dmitriy said, we talk about a lot of that in the book. Not necessarily Git, versus CVS, versus SVN, versus Mercurial or whatever, but a little bit higher level understanding different merge and release strategies and stuff like that. So that engineers, when they enter the workforce, kind of get what's going on and why things are happening the way they are.

I mean, a similar story I had is when I joined at PayPal early on. I was a junior engineer. I was really – Like aside for like Make with C, I was not terribly familiar with build systems. Most of my experiences in like PHP, Python, some of these more dynamic languages, or server-side languages. And so for the good the first year or so, I spent my time building Java inside my IDE with Eclipse rather than using an actual build system. And it took me a while to like understand that at the time it was Ant. But whether it's Gradle, or Maven, or whatever, that there were actually like systems designed to build and manage dependencies and stuff. So these are the kinds of things that like they just happen. And if you're not aware, you're not really even aware of what's going on. It can cause a lot of friction for you.

[00:08:29] GM: Your example with taking down the repository is one I'm really glad you shared, because I feel like every engineer, or maybe I could say it this way, you're not a software engineer until you've broken everything at least once. But that's a terrifying experience, especially if you're new to a job. Do you have any advice for how a fledgling software engineer should best manage a situation like that?

[00:08:51] DR: I would say, first off, this stuff happens. And it's your team's job and especially your manager's job to make sure that they don't give you anything that you can break too badly. So things will happen, especially if you're in smaller companies where there just aren't the safety nets, but it's expected that new folks mess up. In fact, everybody messes up. Senior folks

mess up all the time too. And we build tools to try to mitigate the effects of that. So first off, try to avoid that, but also don't be so petrified of making a mistake that you don’t make a move. That's one.

And second, once you realize that something went wrong, the most critical thing is to immediately say what happened. Get people's help. Don't try to sort of work it out before somebody notices. That generally will only lead to badness. So immediately raise the flag. Say, “Help. Something went wrong.” And I think because all of us have done this at some point, you will find that your teammates are pretty understanding. What's the expression in politics? It's not the crime. It's the cover up, right? So yeah.

I remember at one of my jobs, which shall remain nameless for this particular anecdote, there's one time that I recall an actual email going out to all the engineers explaining why somebody just got fired. People got fired more than once during my tenure there. But generally, people try to keep things respectful. You don’t talk bad about people who being let go. That’s sort of thing. One time, I don't remember if it was like VP of engineering, or somebody pretty high up, sent an email to everybody saying, “Normally, we don't talk about that stuff. But this time I want to make an exception so that everybody's clear.” This story was essentially that somebody made a pretty significant mistake. And then for several weeks, tried to sort of cover it up and fix it up, scrambling to fix it while things were affecting customers in a bad way, etc. And this was a junior person who just did not tell anybody else. And that was the problem. The problem wasn't that they made a mistake. The mistake was fine. And that's why the engineering leader sent an email to explain that it's not the initial problem. It's the fact that then they sat on it for three weeks or something. And it kept getting worse and worse and worse. And it caused material damage to the company. It caused material damage to the customers. That's not acceptable thing, right? Get help immediately when you make a big mistake. Things will be fine.

[00:11:19] GM: Good advice. Well, another thing I think a lot of new software engineers may not be used to is collaboration and teamwork, and interacting with other roles. What sort of lessons lie in the book for someone who's going to have all those experiences and new with their first job?

[00:11:36] CR: Yeah. So I think we cover a few topics. Obviously, I think, sort of tactically, one of the most common places people interact with each other on the software engineering side is code reviews. So there's a whole chapter dedicated to that both on being a reviewer and a reviewee. And it's got practical tips on how to interact, get feedback, work in progress, nits, all that kind of stuff that you would – Again, after a few years in industry, you start to get a reflex to be able to do that stuff. But coming in, you don't even know like what nits are and that kind of stuff, or what a WIP PR might look like. So it's got some of that.

It's also got a lot of discussion around collaboration around design, which is something that I think new engineers really have to spend some time getting up to speed on. And so it talks about, with design, both working independently, and then also collaboratively, and we kind of discuss it in a process by which you kind of oscillate back and forth between doing some work yourself, trying to synthesize the feedback you've gotten. And then gathering feedback, talking with your team, talking across team, talking with trusted technical leads and managers and stuff. And so it talks a bit about collaborating and running meetings in that way, and as well as participating in other design discussions. And then at the beginning of the book, we have, I think, slightly more in our discussions around how to interact with the team and learn how to work with your manager. How to do one-on-ones? All the standard stuff that you would do with your manager.

On the team front, we talked about asking questions. How to time box and get a description of the problem that you have before you go off and ask someone and show a proof of work sort of for the investigation you've done so that you have a place to start when you engage in a discussion with another engineer, rather than just sort of throwing out random questions. So we do our best, I think, to try and equip the engineer with a set of practical tools like time boxing, and stuff like that, that they can use to ask questions productively, collaborate productively and sort of both get what they need, and also not get too much in the way and not feel like they're getting in the way, which oftentimes engineers will do. They'll feel so tentative about asking questions or timid about asking questions that they try and figure everything out themselves. And they might burn a whole week on something that the person sitting next to them just knows the answer to immediately. So that's the gist of it.

[00:14:04] DR: Yeah, we try to encourage healthy, productive collaboration throughout the book, because it's kind of hard. There can be a few different extremes that people gravitate to that aren't particularly productive. Like, for example, with asking questions, sometimes you get new folks who do not want to ask a question because they feel like they should know everything. And they get hired, because they're supposed to be a software engineer. And how could they not know how to do this thing with Git, or whatever? And like Chris described, they just go super deep and spend days on something that somebody could have answered for them and just isn't that important that they really understand deeply. That's one extreme, where they just waste a lot of time, a lot of their time, a lot of company time on something that could be answered easily.

And the other extreme is when like they just ask everything, right? How do you do this? How do you do that? Where's the code for this? Where's the code for that? How do I look something up? And it's just too much, right? At that point, whoever they're asking, possibly multiple people, are just not getting anything because they're just getting barraged with these very basic questions that could be answered by a simple Google search or looking things up in the wiki. So how do you know which one of those you're actually doing? Sometimes people aren't sure. I think maybe they're asking too many questions. And actually they're not asking enough. So we give some advice around that kind of stuff, and how to sort of find the right balance.

[00:15:22] GM: That makes sense. I mean, software engineering is a discipline you. You study it. And it's a craft to take on and a lot of ways. If I want to be great at that, maybe I can look at great habits. Like if you think of an Olympian participating in the Olympics, it's not just that they do their particular sport every day. They probably have diet involved and a lot of other factors and just their path towards that gold medal if they're going to get it. For people new who maybe are unrefined in certain ways, what are some of the aspects of the discipline of software engineering that you think people don't take as seriously as they should?

[00:15:57] DR: I'd say there's some amount of learning outside the job, practicing things outside the job. Again, with some people, they live and breathe coding. And like, actually, probably what they should do is pull back a little bit and hang out with people who don't do software so that they get a little bit more multidimensional. And with some people, they wind up going to the job and then do their job. And all the learning they do is whatever they expose at the job they

particularly happen to land in. And I've definitely found in my career to be extremely valuable when I've read academic white papers, read blogs by prominent software engineers. I just expose myself to the world of ideas out there. But also caution against sort of reading the latest, greatest thing on Hacker News and trying to implement at your company right away. That can be a bit of a recipe for disaster. But making sure you get kind of a very diet to go with your analogy. Just looking at multiple sources and reading without letting it completely overwhelm your life and just become everything you do.

[00:16:59] GM: You'd mentioned side projects, though, as being a part of most software engineers' lives. Could you comment, I guess, on your own side projects. And if you see success in collaborators you've had, how important is it that people should tinker on the side?

[00:17:15] CR: So I think to be clear, side projects are something that not everyone can afford to do, that not everyone has time to spare outside of work and stuff like that. But it is definitely a way to build a lot of great skills. So, for example, for me, I don't do much in the way of side projects these days. I've got three kids. And we're pretty busy. But when I was younger, I did spend some time on side projects.

[00:17:40] DR: You did write a book, Chris.

[00:17:46] CR: That's a good point. Anyway, I did spend some time doing side projects a while ago. What I found was that, oftentimes, the value of the side projects was not the project itself, but the learning whether it was some new stack that I was experimenting with. So I learned a lot of frontend stacks this way, or even new technologies or cloud. So it's a good way to get involved and learn quickly in something that you might not have easy access to at work. Plus, you don't really have to focus as much on test and production code. You can focus more on like learning how to interact with a technology and how it fits together with other stuff. So I find it really valuable for that.

I think the danger with that kind of thing is it can be very deceptive if you spend few hours a week hacking away on something, and then it works or appears to work for your side project. That doesn't necessarily mean it's a great idea or a great thing to use in a production environment that hundreds of thousands of people are interacting with on a daily basis if it's a

product that's got a lot of users or if you've got a large team with dozens and dozens of people. So I think it can – Again, sort of on a cautionary side, it can warp your reality of it. But it is a really good way, I find, to learn technology.

[00:19:00] DR: Yeah, I would just add that I think it's very good to do side projects if you have the passion and the time to do that. That can be great. Sometimes people sort of pick them up because they want to learn a particular technology, and the project they're doing isn't of particular interest to interest to them. And I think the more you're sort of interested in the problem, rather than the tech, the better the results will be. If you're just like, "I want to take this program in C and rewrite it into Rust," but you don't actually care about what the program does, you might run out of steam, or I might run out of steam if it's something where maybe home automation or whatever, like whatever kind of thing that somebody's actually into, or a website that you're trying to create for your hobby or something. Something an actual problem that you care about is much more motivating that just sort of abstractly practicing a thing because it's like a kata and you want to get better as an engineer, right? Do something that's going to be rewarding to you.

[00:20:00] GM: Absolutely. One of the points the book highlights is the importance of meet-ups and conferences is one of many educational resources, although I like the fact that it highlights to do those things sparingly. Can you comment a bit on that, and maybe if you have any anecdotes about value you've gotten out of certain conferences or meet-ups?

[00:20:19] CR: Yeah. For me, I kind of look at it as a return on investment, where a multiday conference is a pretty significant amount of time, for example, to put into doing something versus the other things you could be doing. And so the return should be pretty high. And so there are definitely conferences that are quite valuable. But there are also people that I think sort of become professional conference hoppers and spend a little too much time in that area to the point where the return starts to diminish a little bit.

I think, also, these days, with both virtual conferences and recordings, it's gotten a lot easier to get the content that you're really after at a conference without actually having to go. So a lot of the value, I think, with conferences is really on the social side and meeting people, getting connections, and that kind of stuff. And so that's something that is there. And that can be both

with your coworkers who you go to a conference with building bonds and stuff. And it can also be just within the industry, meeting speakers or meeting people that you used to work with, or they just share your interest. That's all super valuable. I think it's important to recognize there's a scale from multiday, very big, to a meet-up that might be once every month or two, which is obviously less investment. And, honestly, still pretty high return in terms of the learnings you can get out of it and the people that you meet.

And then even further down the spectrum is like consuming the talks on YouTube or after the fact asynchronously, where the investment in time is pretty much the length of the talk and the value is there, although you can't interact as much with a speaker. So I think what we try to encourage is just be thoughtful about where you spend your time and in which areas. In some cases, it's going to be great to go do a full-blown count conference. And then other times, you might just want to learn something very specific and you come across a talk or something. Then in that case, consuming the video or something is probably a better use of time, or just reaching out to the speaker directly, which is, again, something we encourage.

Oftentimes, these people love to talk about whatever it is that they're working on or presenting on. And if you simply get in contact with them, you might get them quite engaged. And so that's something that I think, again, new engineers can be a little bit timid about. But once you start down that path, it can be quite rewarding. And you can build really good relationships with people.

[00:22:32] DR: Yeah. I would add that there are a lot of different kinds of meet-ups and conferences, and just thinking about why that thing exists, and who's putting it on helps. So there are a lot of conferences that are put on by particular companies. And they can be technical conferences, but they're still in a big part a marketing exercise. There are a few that come to mind. But we'll save that. And then there are some that are sort of more focused on the technology. And if you're there to get exposed to the kinds of products that, say, AWS offers, and you go to AWS Reinvent, right? That's great. And they have some good technical talks. And a lot of it is about trying to sell you more AWS services, right? So you should just like know that that's what's going on and come into that with that mindset. And think about how much time out of your year you want to spend being sold to, right? Even if it comes with benefits, like getting technical information.

And then there are some conferences, there are academic conferences that are obviously all about kind of the latest and greatest ideas, but maybe a little bit less applicable to work. There are some conferences, like Strange Loop is a wonderful conference that is kind of very idea-focused, and it's much more community-driven and less corporate-sponsored. The more you can get to those types of things, the more probably you're going to get the return on the value on the time spent.

[00:23:49] GM: When I started as a software engineer, I want to have some way of measuring my contribution. What did I done? What have I accomplished for the team? And how could I improve upon that? The most naive thing I could do would be to measure the lines of code I wrote per day, which is sort of, I mean, not terrible, but obviously not a total picture of my contribution. Do you guys have any thoughts on that unit value of delivery that a software engineer contributes to the things they're working on?

[00:24:17] DR: That might be a whole other book. I don't know. And these days, I've been managing engineers for 10 years. Like if I could do that, I'd be probably making three times much money.

[00:24:28] GM: Well, how do you know which engineers that you've managed are on track and which might need a little bit more guidance?

[00:24:33] DR: Yeah, fair enough. Fundamentally, the engineers are there to solve problems for the company, right. And like you said, lines of code isn't a very good proxy for that. You can churn a lot of code that actually in the end doesn't really deliver value, right? It's satisfied customers. It's problem solved. It's features delivered, right? It’s in-houses done or enabled. Other engineers whose work is now faster or better because you put in some optimizations or improved some workflow, that sort of thing.

Broadly, I try to focus on impact. And when I talk to engineers about their career growth, it's how can you grow into making more impact? Sometimes it's by getting really deep on some set of software skills. Sometimes it's actually by understanding better the problem domain. Sometimes it's getting better at handling relationships, because it's a team sport writing software. But

fundamentally, it's about impact, right? So less about lines of code, or number of pull requests, or things like that. It's what did all those things accomplish? And I even recommend writing down sort of not just I implemented this feature that we should do X, but I implemented this feature, and then we can observe X effect, because that's the actual impact their work had. That said, there isn't like a unit of impact you can measure, right? There's isn't, “Oh, I made 17 impacts this month.”

[00:25:56] CR: Something I often caution people about when I'm talking to them, especially when they're considering like where to work, is impact is kind of a funny thing, because you kind of have to think about the impact of what. For example, you can go to work at Google and change a button from like red to blue. And that can impact a billion people, right? That can generate a massive amount of revenue in absolute terms. I can do a whole lot of things. But intrinsically, it doesn't feel like you've done a whole lot when you've changed the button from red to blue. Whereas you could work at a much smaller company and build a whole product that only 10 people use. And that can feel like a very big impact for that company. And so you kind of have to be thoughtful, I think, also with just about the kind of impact that you care about having and sort of what fits best with your expectations and hopes for your career.

Some people really like being in sort of a big fish in a small pond and are happy to work with 10 customers. And other people really want to have global scale civilization changing effect on stuff. And I think you can do both of those in both environments. But definitely having six of the world's population using your product, it gives you as a company a lot more possibility to have an impact, although individually, it might feel less so.

[00:27:17] GM: Absolutely. Well, the chapter I most wish I had had to read when I was starting up my career would be chapter four, writing operable code. Could you guys give a quick summary? And maybe we'll dive into some of these topics in greater detail?

[00:27:31] DR: Yeah. So this chapter, I think, when I left LinkedIn, I found that I had entered LinkedIn as like a fairly entry level software engineer. And I had left sort of a staff level engineer. And I felt that a lot of what I had learned was in this area around operability. And it was really metrics, logging, configuration, and things of that nature. How to write code that could withstand network outages and stuff like that.

And so I had actually written down sort of, as I left LinkedIn, just a lot of the stuff that I felt like, “Oh, this would be really – I wish more people knew this. This is good information.” And so it kind of sat for a few years. I kind of consider, “Oh, maybe you could do a class, or teach, or something.” But nothing like that ever happened.

And so this chapter contains a lot of that kind of information in it. So it's introduction to metrics, counters, engages, and histograms, and the gotchas with histograms and stuff like that. And then there're discussions about configuration. And we touched on like, “Everyone wants to do dynamic configuration and have etcd or something change things magically while the application runs. We can talk about tradeoffs with that.” We touch on logging and logging sensitive data and the impact logging has on your performance, your application, and string interpolation. So it’s a list or a collection, rather, of important stuff that you need to write real production code and something that's going to run for years that many people are going to have to maintain, and support, and debug, and how to do that in a robust way.

[00:29:07] DR: Yeah, I think there's a lot of – The main value of that chapter to me is actually sort of the introduction. The goal there being to get the new engineer to think about how actually just writing the code that solves a problem is different from writing code that runs for years and that other people have to maintain. There's a pretty significant difference. And that's the main difference between sort of knowing how to program and having gone to like a CS undergrad, or a boot camp, or something like that, or just learn from tutorials online and working at a company that treats software as a product, right? And that's one that is like it's a huge difference. And people don't really talk about it. And a lot of the time even, it's not covered in new hire orientation, things like that, right? A new hire gets some tickets. They implement some feature. It goes into code review. And the senior engineer goes, “That's great. But you didn't recommend any of the counters so that we don't know the deprivation happened.” And then the new engineer goes, “What does a counter? What is it supposed to do? And I didn't know I'm supposed to do that,” right?

And then like sort of the why of that also gets lost. So we want to cover the why and just like get people to start thinking about the operating about the software existing in an operational environment, and the fact that it's different than just getting the thing to work and to return the

right result, right? It's not about passing the test once. So that's kind of the main thing that this chapter tries to convey. And then there's a lot of the practical metrics and logging and all that.

The login section is pretty flat for me. I spent a depressing amount of time, a fraction of my life on logging. When we just go and learn to program, like logging is print foo, and you read the thing you printed out. And that's logging. And in an operational environment, that's not what it is. And you can just ship a program that spits out a bunch of garbage and other people will have to read, or deal with, or maybe it doesn't give them the right information. How do you format it? There's a whole ecosystem of tools now that process logging and like visualize log data, etc., just having sort of random English sentences or multi line messages. It's a total disaster. But it's a disaster that like you don't realize is a disaster until you try to process it on the other end, which most people don't get to. So they got to have to learn it the hard way.

Highlighting the fact that logs should be structured, and there should be a system to what's in there, and log levels are a thing that you should know about and understand what they mean. Versus just sort of, “Well, I saw in this other file they're printing to lock their info out, print this here.” There's a lot of that kind of information. None of it is particularly complex, but it's just stuff you should know.

[00:31:59] GM: Absolutely. When I was learning to code, I was pretty sloppy. Sort of, as you were describing, just print statements. And my code would write to standard out. And I'd look at it. And I didn't know there was much more to it. I, in fact, for a while kind of resisted log leveling because I didn't get it. Messages felt so ephemeral to me. What is –

[00:32:19] DR: Right. And that's the natural sort of way that that goes. Everybody does that. it's not you. It's everywhere.

[00:32:25] GM: Well, so for people who are still working under that assumption, can we maybe describe some of what logging looks like in a professional production environment? What's there besides standard out?

[00:32:36] DR: Well, there is actually a school of thought that you should only log to standard out. So there is that. But generally, logs are there to inform, and you want different levels of

information depending on what's going on, right? So that's where the log levels come from. If you're trying to debug some sort of really tricky production situation, you want all the information. What exactly is happening in the system? So that's one level.

But you don't want that happening all the time, because that's a lot of information that's you're spending CPU cycles printing that. Maybe you're spending space on the hard drive just kind of filling it up with tons and tons of info. When you're operating a website like PayPal, or Twitter, or LinkedIn, these are all companies we've worked at. Massive amounts of traffic means that just a few extra lines of logging is gigabytes and tens of gigabytes potentially per hour of data that's not very useful. So most of the time, you don't want to be printing that stuff, right? But you want to be printing the important stuff, because somebody should be able to look at the log and confirm that the process is actually running. It's doing the right thing. It didn't just freeze, and that's why it's not printing, right? So you don't want to be totally quiet. So there's kind of a level of logging, which is kind of general information. I booted up. I'm ready. I'm listening to requests. I processed some requests. Things are all going swimmingly. And then there is a sort of, “Oh my god, an error happened.” What do you do then? So that's the notion of log levels. Depending on what kind of situation you're in, you want to be able to turn on different levels of logging to give you more information, but you don't want more than you need because you get overwhelmed very easily as a human or even as a system.

Another thing with logging is that we very strongly advocate for structured logging. Structured logging, meaning it's a machine consumable, not just human consumable. Logging should be machine consumable. Because let's say that you're running a web service, right? Well, the logs are probably going to get consumed by some system like what's called ELK, Elasticsearch, Logstash and Kibana. It's a very common combination. LogStash takes your logs and slurps them up into a central cluster where they get parsed and stored in Elasticsearch. So the logs can be searched now. And operations team or the development team look at, “Does this rare event happen? I'm going to search for the string that I logged to see if it's happening on any of the instances of my service,” right?

To make that convenient and easy, you want to structure. Probably you want to log in JSON. That's a very common choice. But also not just like random JSON, where you just throw in whatever kind of keys and values, but something that can be planned for an expected because

then this Elasticsearch, Logstash, Kibana combo and another similar platform can be tuned to sort of create various dashboards or give you alerts when things happen, etc., right? That it needs to know what it's looking for. And the more it's kind of parsing freeform text, the more CPU cycles it's spending on something that isn't particularly valuable when you can just tell it where to look by having name fields.

[00:35:42] GM: Makes sense. Do you have any anecdotes about a time when logging either saved you or became some deeply insightful piece of your process?

[00:35:52] CR: I actually have a story about the reverse, which is what I when I worked in the early days at LinkedIn, we used a system called Splunk, which is loosely analogous to the ELK stack that Dmitriy was just talking about. And the thing, we didn't have enough resources to really run it in a way where it was responsive. So we effectively couldn't see our logs. The only way we could see your logs was we could go to the SREs and ask them to log into the individual machine that had a problem. And then like tail, the logs that we could see it, because, of course, they don't grant production access to all the engineers, which is again, a shocking thing that new engineers don't always expect. But like you most likely will not be able to log into the machines that are running your software. And so you need to think about how you would debug a system in that mode. And so I found myself spending just this inordinate amount of time like wandering over. There's probably like a path is worn in the rug between my desk and the SREs that had login access to the machines, because like I put a bunch of logging in, but then the system, that the Splunk system wasn't keeping up with it. So that eventually got fixed. And that was a whole other system and process to get going again. Until you go without logs in a production environment, you don't really know how important they are in debugging stuff.

[00:37:13] GM: Absolutely.

[00:37:14] DR: Yeah. I have a little bit of a story. When I joined Twitter, it was pretty small when I first joined that. It was like 100 people. And we had logs and we were running a system called Hadoop, which maybe the new generation doesn't know about, for collecting all those things. But we hadn't yet standardized on how logs get formatted. And so it was sort of whatever is the natural way for a particular language ecosystem or framework to log things. That's how things would get written.

So if you're running like the Apache HTTP server, it has the Apache log format, which is parsible, but it's got a plain text format that has a bit of delineation with like tabs and colons and things. And if you're running a Ruby process, well, Ruby on Rails logs in a different way, which is a really hilarious standard that like you name your fields, and then the fields get printed in alphabetical order by the field name. So if you add a new field, all your offsets are messed up. And you can imagine the hilarity that went on with that.

And then if you're running Scala, then maybe you're doing something completely different. And so we've had this collection of awful parsers and regular expressions that tried to parse all this stuff. And I used to give talks at conferences with our real production, like regular expression for parsing these logs, and just seeing how messed up that is with all the different captures and backtick. It was pretty impressive.

And then saying like, “Look, we could be doing that, and maintaining this regular expression for parsing your log.” Or you could log in JSON, or Thrift, or Avro, or what have you. And then it's just logmessage.getfieldfoo, and there it is, and nobody needs to think about how to parse it anymore, right. And that really shines a light on why that’s important.

[00:39:06] GM: Makes sense. Well, you both have worked at some very well-known brand name companies, the types of companies that a software engineer would aspire to work at, because they're known for being tech-forward. But every company seems to need a software engineer these days. And only 20% of the people can work in the top 20%, so to speak. So I think there'll be a lot of people finding themselves at companies that are, let's say, early in their tech journey, or maybe developing, aspiring to be more tech-forward. Can you comment on how some of the lessons from the book will translate into that more pioneering space that probably a lot of the listening audience will find themselves in at some point in their journey?

[00:39:45] CR: Yeah, I think a few things jumped to mind. One of them is, I think, just because a company isn't “tech-forward” it doesn't mean they don't have these problems and are not solving them. And so I think you have to be somewhat realistic about the environment you're in and what you try to change. And so we talk early on, for example, about this whole movement, use boring technology, which is sort of this pragmatic, practical philosophy that talks about the thing

you're trying to do is solve a business problem. And the way that you do that happens to be with software. So you need to be very selective about where you're trying to innovate with new software versus where you just try and use the tried and true thing. So I think we have some philosophical stuff like that that I think dovetails very well with more traditional companies or companies that are just getting started in the sort of software engineering journey.

[00:40:38] DR: I think, like Chris said, we put a lot of emphasis on, first, figure out what your team does, and why they do that. And we try to explain some of the ways they might be doing it in the book. But of course we can't cover all of them. And if they're doing something we're not covering, it doesn't mean they're wrong. Maybe that's the right thing for the company. So definitely would not want people to sort of brandish our book and say, “Everybody should be using this way to run their sprints or something.”

So picking what things need to change I think is important, and why, and kind of understanding what's there and for what reason. We happen to work at some brand name companies, also some non-brand companies that potentially later became brand name companies. There's always something to change or to improve. And a lot of the sort of leading technology now that was commonly adopted didn't come out of the Googles, and Facebooks, and then LinkedIns of the world. Django came out of, if I remember correctly, some sort of a publishing outfit, and they needed a web framework, and they created one for Python. A lot of these things actually come out of fairly small shops. So it's not necessarily, “Well, that's the way Google does it.” This is a really messed up answer. Let's cut all of that.

[00:41:50] CR: Let me take another crack at sort of the brand name part of it. But like, I think one of the things we've tried to highlight is like there are a set of problems. And regardless of whether you're working at some “forward-looking tech company”, like Google, or Facebook, or whatever, or whether you're working at a very traditional company, the problems by and large are the same. It’s collaborating with other engineers. It's committing your code and releasing your code. And this is true even whether you're doing web service development, or mobile development, or embedded systems. Now, the way that you get your software on to an embedded system is different. But build and release, there are sort of standard principles and practices that can translate.

And so I think a lot of what we cover in the book is best practices that apply to these set of problems, regardless of whether you find yourself in a very modern tech company or a more traditional one. And I think, to kind of couch that, we give some guidance on stuff like this, use boring technology movement, where we're somewhat cautious. I've been the young, very passionate engineer that tries to manifest change in an organization and write everything in Scala instead of Java and use this build system and that test system. And I think one of the things we just want to push is just pragmatism. So if you find yourself in a company that is still not as forward-looking or has frankly other concerns on their mind, you have to be somewhat thoughtful and practical and realize that you have only so many, as they use, boring technology people say, only so many innovation tokens that you can spend. And it may not be best for the business to spend them all on some monadic Haskell build system or something. So I think that's sort of how we try and balance it. But I think a lot of what we talk about applies to pretty much every company that's trying to work with software.

[00:43:35] GM: Absolutely.

[00:43:37] DR: Yeah, there are a few things like continuous integration, and even more early, continuous deployment to serve an ideal that has been bandied about a lot, and a lot of folks strive for that. And so if you sort of read in a book like ours about continuous integration, or like you read *Accelerate*, or one of those other books, it's a great idea. And then you're at a shop that releases their software once a month. Should you just say, “Oh, my God, I'm in the dark ages, and these people are doing everything wrong.” Maybe not. Continuous integration is great. And there's very good reasons for it.

Going zero to 60 on something like that isn't the right thing to do. So starting to look at, “Well, how can I improve my feedback loops and quickly understand whether the code is working or not? And how do I reduce the amount of risk?” We look at what they should deploy? That's important. And so you can start looking at that, but sort of immediately trying to, “Oh, my God, how do we release as soon as we commit so that we go from once every 30 days to once every five minutes?” That's probably going to break a lot more than you expect and not provide the value that you want to provide, right? So balanced with everything, I think.

I work at a company right now where we work with a lot of scientists. We can't just change the frontend on them for scientific tools as quickly as we develop it, right? They can't absorb change that fast. They would rather get a new training once a month or something about all the new tools and how things are changing. And there's kind of a rollout process with that. So do we just say they're doing it wrong and we're going to force change upon them? Right? Well, that's probably not the right thing to do. If they rightly want to focus on the science and not on relearning the tool every day, because it changed under them, right?

However, we can do things like push things out behind feature flags so that we can easily turn them on or off, or do things like dark reads and dark writes, where we see if the system will work under the covers even though we didn't actually change anything visible to the user. These are advanced techniques that can be put in place without sort of exposing all of the velocity of change to your end user. Without something like that, you can't just go, in our case, and just ship all these changes to people, right? So that's where maybe something technology-forward like continuous integration isn't – Or continuous deployment rather, isn't the right thing in a particular situation. So you kind of have to understand why the thing that you think is technology-forward isn't there? And what are the reasons? And then figure out the path forward. We're bringing the good things without forcing something that's unnatural to the organization.

[00:46:13] GM: Good advice. Absolutely.

[00:46:14] DR: Did that make any sense? Well, that was a run-on.

[00:46:18] GM: Well, I think *The Missing README* is a great reference guide for any software engineer. It could also make a great manager's gift for a new employee. Where can listeners pick up a copy?

[00:46:29] DR: Yeah. My thoughts with writing this book was that more so the tech leads and the software managers will be buying them by sort of the dozen to give to new engineers, because new engineers don't yet know that they need it, right? That's the whole problem. You can buy it on Amazon. You can buy it nostarch.com. That's our publisher. You can buy it everywhere books are sold, Barnes and Noble, what have you. It's distributed worldwide. So enjoy. Ebook or physical copy, whatever you want.

[00:46:59] GM: Very cool. We'll have some links in the show notes so people can follow up. And where can they follow up with you guys on social media?

[00:47:07] DR: So I am @squarecog on Twitter.

[00:47:11] CR: And I'm @criccomini on Twitter. C and then my last name, which is a long one.

[00:47:16] GM: Very cool. Well, Dimitriy, and Chris, thank you both so much for taking the time to come on Software Engineering Daily.

[00:47:23] CR: You bet. Appreciate you having us. Yeah.

[00:47:24] DR: My pleasure.

[END]