EPISODE 1390

[INTRODUCTION]

[00:00:00] KP: Many software projects run the risk of evolving over time to a complex state that is inhospitable for new contributors to join. This is a dangerous place for a company to be. Either software needs to remain more accessible, or faster paths must be created to help them get on board. Today's interview is with Kartik Agaram. We discussed several of his projects, including Mu. We explored some of these topics and software development in general. Kartik has some really interesting open source projects and we get into several of them.

[INTERVIEW]

[00:00:31] KP: Kartik, welcome to Software Engineering Daily.

[00:00:35] KA: Thank you. Thank you for having me.

[00:00:36] KP: To kick things off, can you tell me a little bit about how you first got interested in software?

[00:00:42] KA: Yeah, I think that was traditional in some ways, and not so traditional in other ways. I'd never been near a computer until I was 18 years old. And I discovered that I love computers in undergrad. I joined undergrad as a CS major. That was my first exposure to computers. I'd read about them before, but that was about it. But then I fell in love with computers. I ended up finishing my undergrad in India, moving to America, getting a PhD in CS, and it's been great.

[00:01:13] KP: What was the focus of your PhD?

[00:01:15] KA: So, my dissertation was sort of straddling microprocessor architecture and compilers. I was working on stuff like the memory wall, we used to call it in academia, back then. This is like 10 years ago. The fact that processors run so much faster than accesses to

ram or now even disk, and how do we keep programs running fast in the face of those latencies?

[00:01:38] KP: Is that still a modern problem? Or is computing evolved in new ways?

[00:01:42] KA: I think there's definitely still people working on it. I have grown less interested in it. Lately, I think of performance as something to satisfy for, rather than optimize for, and I care more about other things. For example, the way that I worked in my dissertation was that we would have a set of benchmark programs, and we wouldn't modify them. Whereas now, I'm all about how do I modify applications to get the properties I need? Whether it's readability or simplicity, or performance sometimes.

[00:02:15] KP: Well, as a developer, I'd like to think I can just ignore things like the microprocessor and the compiler that that's sorted out for me, maybe the compiler is going to fix a few of my mistakes, or my shorthand, optimize things for me, and I can really just focus on, I guess, my business logic. Is that a responsible way to look at it? Or should a good developer really be involved in the full stack?

[00:02:38] KA: I think there's two ways to look at it from the standpoint of a professional developer, I think you can get by pretty far while treating these as abstractions and not worrying too much about what's inside the black box. However, as you grow more senior, you end up having – you'll run into a few situations where it's helpful to know what's under the hood. And one thing I've noticed, which is a little bit of – nobody talks about it much, is that the best programmers I've seen are constantly asking what's behind the curtain, what's under the hood. And repeatedly doing that gives them, I think, resources for when they really need to do it at work.

[00:03:20] KP: Absolutely.

[00:03:21] KA: I'm thinking of people like John Carmack, right? It is pretty clear that he understands his computer on a very deep basis. It's not just on examples like that. It's I've seen programmers I've worked with at work, who were really, really great and they all had this habit.

[00:03:36] KP: When you're working with multiple programmers is, ultimately, I think every software project of any scale kind of tends to evolve to that and being a team effort. There become a lot of challenges around how do we collaborate? Do we have different coding styles? Things like that. I know you have a very strong thesis on this, I'd like to unpack in a couple of ways, but maybe just give you the floor for the moment. How do you look at the structure of programs?

[00:04:00] KA: So, a few years ago, Steve, yeah, he had this thesis that there's two kinds of programmers, conservatives and liberals. I tend to be on the batshit, liberal side of the spectrum. I strongly believe that we create better programs when we allow the people who build them to make mistakes. So, I tend to not like style guides and things like that. I used to teach programming for a while, and now when I'm with junior programmers, often they'll ask, “Should I do this or do that?” And I try as far as possible to say, “Well, try it try both ways. Which one do you like better and why?” So, having that conversation is, I think, much more valuable than defining decision.

[00:04:43] KP: There's something that can be intimidating about that a little bit. I think for a junior developer, I remember points as I was learning how to code where I might see two different ways and I wasn't sure I knew what the right metrics were like, how do I measure? What's your standard and having thought about that a little bit?

[00:05:00] KA: The thing is, most of the time, I don't know, either. And so, when I think that there's a really strong reason to choose one over the other, I will try to articulate it to someone else. But I'm very careful to try to articulate it not in terms of aesthetic choices, or you should do it this way. Instead, what I try to do is find a concrete example, a situation, where one is preferable to the other. I think the biggest one I can think of, for example, is lexical scope. Why should a programming language have lexical scope rather than dynamic scope? I think there's a very nice answer you can give there about how it prevents this action at a distance, and makes your programs easier to think about. We can even like spend – I spent time with my students collaborating on an example program, where if you use lexical scope, then you can make changes to this other part of the program, and it doesn't affect correctness, and things like that.

[00:05:57] KP: There's a challenge as many developers come onboard to a new project of just getting up to speed with it, learning the local style, if you will, or maybe there are aspects of the architecture, they need to get familiar with, the schema, things like this. Everyone kind of gets this. I've heard a whole lot of wisdom about how to prepare for a growing project and being welcoming to other developers. Do you have any advice along these lines?

[00:06:24] KA: Well, so first of all, like I said, it's a hard problem. The fact of life today seems to be that the way software is developed professionally, it tends to be very complex. I tend to think that's a bad idea. But at the same time, the reasons I think that our basic research, I can't really say that company should drop what they're doing and follow what I do. So, we'll set that aside for a moment. Within the frame of reference of the way we tend to develop software professionally today, there's a lot of complexity, and you have to pay people, because it takes weeks and months and years of full-time effort before people get up to speed on a new project. I think a lot of that challenge is fundamentally about learning a new domain. I find that if somebody is already working on, say, CI/CD systems, when they come to a new project, if it's using a completely new bespoke system, and a programming language they're not familiar with, it's easier for them to ramp up. Whereas someone who hasn't been in the domain takes a little bit longer. I think the role of the firm is once you hire someone, you support them as best you can, during that initial ramp up period, which can be multiple quarters.

[00:07:39] KP: Well, there's a challenge I've seen a number of technical professional's face, maybe someone has just started as a director level, CTO soMuhere, executive, and they're inheriting a whole big source code that is kind of hacked together over time for a growing business. And you know, maybe there's been a changing of the guard, even in some developers, and the new developers are saying, we have to start over. Maybe there's all developers saying, "No, the system is great. We just need to invest in it." And this is a challenging question. I don't know that we can solve it right here. But do you have any rules of thumb for how to approach such a challenge?

[00:08:14] KA: I think back in the early 2000s, Joel Spolsky, had this very influential blog post about why you should never ever rewrite a code base. He made a lot of good points. At the same time, the big example he gave then was Mozilla, which went open source and did this

huge rewrite and almost died. But then they came up with Firefox, and Firefox did really, really well, for a long time, is starting to go down now. So, it's complicated, right?

I think one thing is, if you have an open source project, you have more room for rewrites, because it's harder to kill you. The business pressures are less active. But I think, stepping back a little bit, the real issue here is that it's the wrong question to ask, should I rewrite or not, to me feels like the wrong question. Software is the most malleable medium known to man, and if you give up on rewrites, then what do you have left? I feel like you're throwing the baby out to the bathwater.

So, more fertile question to ask is, how can we develop software to preserve which rewrite friendliness and to keep it easy to rewrite over time? And that's still – there are lots of open questions there. But the two major things that I really, really look to when I try to keep my software rewrite friendly, are version control and tests. And so, if I ask, why did I write it this way? Well, to rewrite software, you have to first understand why it was written the way it was written. That's sort of the idea of Chesterton's fence. If you want to take this fence off the street, you have to first explain to me why it was there. If you don't understand why it was there. I'm not going to let you take it off. So, that's the idea of Chesterton's fence, from G.K. Chesterton.

I think that applies to software as well. Version control is really great for, if you have a team where there's a culture of writing detailed commit messages, then I really enjoy digging through the past history of a line of code, and understanding how it came to be, before I fixed the bug in it. And tests are helpful as well, because anytime you rewrite, the big cost of a rewrite is not the effort of typing in the characters. The big cost is the risk that you pay that pit in your stomach, when you're changing something and you're scared, you're going to cause a digression, and tests are great for avoiding that.

[00:10:38] KP: Well, I'd love to do like a zoom in on some of your open source projects. But before we jump right into that, could you maybe share an overview of what some of the projects you've launched are? And what really motivates you to open source these efforts?

[00:10:52] KA: I think my motivation structure is deeply ruled by feedback. So, I just constantly want people to see what I did, tell me what's right with it, what's wrong with it, to be seen. I think

that's probably fairly fundamental. So, I've always developed in the open, well, at least ever since GitHub came out. So, the projects I work on, at least for last 10 years, I've been wrestling with this problem, which we alluded to earlier of, how do we help people become more expert in a strange codebase. And at least on my side projects, I try to attack the boss level of this problem, the hard version. The easy version as you hired someone. Now, they paid to spend eight hours a day thinking about this one code base, and it takes them a few weeks or months to understand it.

The hard version is, it's just somebody driving by your repository on GitHub, and they use your software, and they want to make a change to it, and they have an afternoon. If you don't give them a sense of progress in one afternoon, they're probably not going to come back, which means you lost a potential contributor. So, the hard version of the problem is, how do I take people who use my software and gradually turn some economic fraction of them into contributors that helped me compete against others open source projects? That's how I view it.

[00:12:20] KP: And that drive by contributor, I mean, realistically, what's the expectation? What could they contribute in the best curated codebase?

[00:12:30] KA: Yeah, you're absolutely right. I think when you look at a contributing file, like a lot of open source repositories have a little markdown file, which describes how you can read to them, this will put the cart before the horse. In the early stages, I'm not really viewing someone as a contributor, so much as just trying to explain myself to them. Here's the software I wrote, here's why it's written the way it's written, and supporting them in understanding the design choices. So, in that first afternoon, I'm not expecting a PR of them. I'm just expecting them, I would like them to have a sense of satisfaction.

I had this niggle, maybe I had some curiosity about why, how does this work? I'm using this text editor. How does it know how to do this? How can I get it to highlight this parenthesis a different color or something like that? And I would like them to come away from an hour spent trying to answer the question with a sense of, "Oh, I answered that." Or "I learned this, and now I know enough to put a pin on it and I can come back next week." But if they just flail around for an hour, and they feel like they didn't make any progress, to me, that seems suboptimal.

I guess one way I would say it is learning software is hard. Writing software is hard. We're still in the stone age's there. We don't know how to do it well. So, I'm not trying to make it easy. That sounds too much like dumbing it down. Instead, what I'm trying to do is think about how to carve steps into that cliff of how difficult it is. So, make it easy to climb up one step at a time, rather than – well, one way I put it is, right now you have to put in 100 hours to learn something in software about a software project. And then you get possibly 100 hours' worth of return. I would like to be able to put in an hour of effort and get back, say 20 minutes' worth of return. But right now, most of the time, that first hour you get zero.

[00:14:26] KP: Is that some of the motivation for the Wart Project?

[00:14:30] KA: Oh, that's really going back. Yeah. So, the way I got into Wart, and I'm going to be revealing my two listeners here is I tried to do a startup. And my startup was about trying to make RSS feed readers accessible to people, because even then I could sort of sense that the direction we were going with walled gardens was not great. I wanted to provide some counter force. And I was completely a failure at that. Because as I worked on it more, I got more and more interested in the web stack that I was using to build that project, and I started going deeper and deeper.

In some ways, I was going back to my roots, because my grad school work was all-in low-level software. And I left grad school because I got sick of low-level software and I came to Silicon Valley and I started writing web apps where I could make a change and hit reload on the browser and get instant feedback. But then I started noticing all these problems with high-level software. And I started trying to use what little expertise I had in the low levels to try to think about how can we make things better in the high level?

[00:15:39] KP: Do you have any key takeaways from that effort?

[00:15:42] KA: I think the biggest one is, we make these problems harder for ourselves unnecessarily. Like people keep talking about simplicity, but if you look at the number of dependencies, even the simplest project has today, it's ridiculous. The inmates are running the asylum. So, I just try my projects to be keep things really, really simple. Because every new dependency you add to your project is a place where somebody's trying to compile it from

scratch, is going to run into a roadblock and go away a potential contributor. And so, I just try to keep my projects like very, very minimal in terms of dependencies. I think that's the biggest piece of it. If you have very little software that you use to build something, there's just less that can go wrong. Honestly, I don't have much more to contribute beyond that, like, on some level, everything I've done is trying to find ways to do more with less.

[00:16:35] KP: Well, if you went to Silicon Valley to get a little bit away from low level machine code, what inspired you to start the Mu project? Or maybe you could share what Mu is first?

[00:16:45] KA: Yeah, so Mu is a computing stack is sort of the best we have come up with to describe it. There are things that occupy a similar niche to Mu, which people call operating systems. But I try to think of it a little bit more holistically than that. Because when you say operating system, people have an expectation that the software that runs on a different OS can now be recompiled to this new OS. What I have is a computing stack. It sort of stands independent from all the software out there, and it builds without using any existing software. And so, it's built up from scratch from machine code, and it's built up in a sort of very disciplined way where I first built the lowest levels, the foundations, and then I built higher levels above them in terms of the low levels. And I sort of follow that discipline all the way through.

I very carefully avoided this idea in computer science called meta circularity where you use a C compiler to compile the next version of the C compiler. The reason I did that is that meta circularity I think, makes software less approachable to outsiders. Because if you probe inside the C compiler and say, “Well, what does it really mean in terms of the semantics of the compiler?” You quickly find yourself drilling down into C code, which contains ifs, which sounds suspiciously like circular reasoning. And if some in Mu trying to find the right place to cut that circle, that's very challenging.

So, the way I built Mu was A, it has very few dependencies. It has very few notations, very little that you need to learn. But on the other hand, everything in it is sort of new and alien, a slightly different way of looking at things. There's a notation for machine code for x86 machine code, which is much simpler than assembly. It had to be simpler because it's built in itself. So, that's the place where the bootstrapping happens. So, I call this notation sub x and sub x is implemented in sub x. But it's like so simple that it's feasible to audit that and understand exactly

what's happening. It's built in terms of the x86 machine code. So, there's a sort of underlying grounding there. It's not just some random notation that I picked. I'm trying to find the best notation to explain x86 machine.

And then I use this notation to implement a safe programming language on top of it, which is without using any C in the process. By safe programming language, I mean, that it manages its own memory, there's no garbage collection. So, it's pretty low level. It also manages its own registers. So unlike C, I expose the registers to the programmer, but at the same time, the tooling will protect you with good error messages if you make a mistake with memory allocation or register allocation. You don't end up in situations like in C where you forget to free something and that causes you problems far away, again, action at a distance in your code base. Or you see something twice all and then in C, you might as well just throw your hands up, right? You need tooling to fix that and to figure out what's going on. That tooling is incredibly complex and has its own dependencies.

So, here, I tried to avoid the need for that tooling by providing very strong safety guarantees upfront. It's kind of similar to Rust in terms of the amount of memory safety you get. It doesn't have all the functionality that Rust provides for concurrency. But it does provide similar guarantees for memory safety. But the way it's able to do that, by being really simple, is Rust has a lot of smarts to figure this stuff out ahead of time, without costing you any performance at runtime. Whereas I'm willing to take on the performance in runtime to keep the implementation really simple and easy to understand.

[00:20:33] KP: I was actually curious if you are aware of anyone who's rolled out Mu into their production system and use cases we can explore?

[00:20:41] KA: No. So, the current status of Mu, I think there's two major challenges. One is, it is a pretty alien way of looking at the world and I've had trouble convincing people to look at it. And I think with good reason, it's still there's a lot that it can't do. There's basically two ways I can deploy Mu today, one is on top of a bare bones Linux kernel, in which case I get access to the networking stack and the file system. You can write simple batch programs that read from a file or something like that, read from standard in, write to standard out. But there's no graphics.

The alternative is I can run Mu in what I call bare metal mode, where it runs without an operating system, and there have access to graphics, and I can draw pixels on the screen.

But on the other hand, I don't have a networking stack there. Because what Linux really gives you is like this really, really comprehensive support for lots and lots of device drivers for storage and networking. I support like one kind of hard disk, and that's it. That hard disk is I think the driver, that family of hard disk is probably like 20 years old, and no networking. At least with the hard disk, I can hope of attacking like 40% of the devices out there, with one device driver. With networking, things are much more fragmented and it's much harder to justify putting in the effort.

So, I sort of backed myself into this corner where I built something without making any compromises by my standards, like by the metrics that I cared about. But at the same time, I eventually ran into this brick wall where it was just too hard. I had to learn a whole lot to even get to this point. But beyond this, it was hard for me to see our progress. So yeah, to answer your question, nobody's used it. I'm not sure people should use it and I can't really justify asking people to look at it.

[00:22:35] KP: If nothing else, I imagine it's a teaching tool about how to interact with these lower level systems. I've always felt like I was very much in the dark about these technologies, you know, trust the library, trust the operating system, trust the DLL, whatever it is. Were there any interesting findings and learnings you had in doing it all from from scratch, really?

[00:22:54] KA: Yeah, thanks so much for bringing up teaching, because that's an intrinsic part of this whole trajectory with me. When I built the first prototype of Mu three years ago, or four years ago, it was just a simple tree walking interpreter. It didn't have all this low-level stuff with machine code, and so on. And I used it for teaching for several years. I got pretty far with that. I think a lot of that was just teaching someone one on one is so much more productive. It's so much more tolerant of error. I didn't know what I was doing with teaching. But just the fact that I was doing it interactively with a single person made it so much easier at a time. When I switched from that early prototype to the next version, part of the motivation was that I built a text editor, the environment for teaching in Mu itself, and it was really slow because it was interpreted.

So, the initial motivation then was to do Mu right, not just for teaching, but for building real software. I think I accomplished that goal. You can now build software in Mu, which will run really fast because it's built in – it's translating the straight up assembly machine code. One of the interesting properties of the Mu language now is that you get this memory safe programming language that is statement oriented. And every single statement translates to single instruction on machine code for the most part, which is really nice. I really enjoy programming in it. Given the problems that I described, the brick wall with networking, and graphics and so on. I think now I've fallen back to this idea of I have a really nice UI for teaching programming. So, I think that's still like a really nice way to learn programming. I tend to believe that we should teach programming from the ground up and add abstractions over time later, rather than start with a high-level language, which in some ways, almost guarantees that people will never learn the foundations, which are what you really need to be really successful and become a really great programmer.

[00:24:53] KP: Let's also get into Teliva, another, I guess, your most recent announcement. What is this project?

[00:25:00] KP: So, Teliva was sort of stemming from my frustrations with Mu. Basically, I spent four years programming Mu from the bottom up in hopes of eventually getting to a networking layer. And I never got to the network layer. And so, with Teliva, I spent all this time trying to avoid C, maybe that was the wrong idea. And so, it cost about looking for an existing stack that I could adopt, that would get me to a networking stack really, really fast. And at the same time, have very few dependencies. Obviously, it's not going to be quite as uncompromising as Mu. With Mu, I always had zero external dependencies. But I was willing to compromise a little bit there just to get to something that people can actually do use interesting things with, right?

So, I stayed with the Linux kernel, because I had some experience with that. I mean, I say Linux, but really, it's like any Unix like operating system. But I'm only using the kernel and the system libraries that come with it. And on top of that, I decided to go with Lua after casting about looking at a bunch of different languages and platforms. And Lua is this great programming language that comes out of Brazil, and it's been around for 30, 40 years, and it's had a lot of uptake in particularly the gaming industry, I gather. And the nice thing about it is that you clone the repository, it's 12,000 lines of C code, and you run make, and you're off to the races, it's just

so easy to build and test a little code. I've been looking at it for four months now and I already understand so much of it.

Come back to Mu, the big drawback is that it's not memory safe, because it's written in C. And already, I found myself running into cases where I was shooting myself in the foot when I make changes in Lua. But yeah, I'm starting with a really rock-solid base and I'm starting to now relearn these old skills I used to have a few years ago for programming with C.

[00:26:59] KP: What about Lua makes it so welcoming for people to learn?

[00:27:02] KA: Just the fact that there's so little code, so few libraries. It just uses Lip C, the standard library for C and that's pretty much it for the basic foundations. And then over the last month, I packaged up a Lua library for networking, and the Lua library for HTTPS for cryptography in the networking. And there's an ethos in this whole ecosystem where people are constantly trying to keep things simple. I really, really appreciate that.

So, it's been – I'm shocked at how easy it was for me to just bundle these things together into a copy of Lua and sort of make it that is included for my definition of batteries, which is basically trying to create text mode applications that run in a terminal. So, people would need to learn to open a terminal and stop being scared of it. But if you're willing to take that first step, I want to really try to support people in running, building their own applications, taking applications that other people have built and remixing them, and doing it in a way that's like really, really welcoming to outsiders.

[00:28:09] KP: Could you give an example of a text mode app?

[00:28:12] KA: Yeah. If you think about one of the classic GUI apps that people used to learn programming, I have a counter, and I press a button and it increments the counter. So, you can do that in the software today. I built a little networking app for testing out the networking where so I play a lot of chess, and I have a big user of this website called lighters, which has its own TV channel where you can watch really strong players playing chess. I find it exciting. So, they have an API. I built a little like program in Lua, which I can use to follow the latest TV Channel

Index mode, just shows you a little chessboard in text mode and shows you little icons for the pieces. It's not as nice as a web browser.

When I first launched it, I called deliver brutalist, like the architectural movement. And what I meant by that was that when you're working with raw pixels and graphics on a browser, you can make things really, really pretty and elegant, looking, at least superficially. But there's like a huge amount of software that goes under the hood to make that happen. Whereas with Lua, I focus on functionality, and the UI you get is sort of this blocky thing, every character looks like it's made out of cement or concrete. But you can do so much of what you can do in a web browser with so much less code, and so much less that can go wrong. One thing that I – so this is like the most important thing, and I haven't gotten to it yet, but the key distinguishing feature about Lua for me, compared to everything else out there is that when you run a program written into Lua, there's always on the screen, in the UI, a single consistent way to modify the app.

So, you modify that app from within the app, and the framework gives you the ability to do that. And so, you press this hotkey right now, it's Ctrl, E, for edit. You click edit, and you get thrown in this UI, which is you know, it has a built-in browser for looking at all the functions in the program. And then you can edit into functions. And when editing, then you're just editing raw Lua code, which could be running anywhere else and then you exit the editor and it restart your app, and you see if it works or not, and you can go back to editing it. At least that's the goal. It's still early days. I still find myself often having to come out of that built-in editor and do things in my daily driver editor. But hopefully, that'll go down over time. Lately, I've been trying to add version control into it. So, you can ask questions like, “What did I change recently that caused it to break that sort of thing?”

[00:30:48] KP: So, is Teliva a new language or a fork or a framework? How do you describe it?

[00:30:54] KA: I suppose it's fair to call it a fork. I'm trying very hard to keep people's muscle memory. So, if you know Lua, you should be able to program in Teliva very easily, and nothing's going to – I’m not going to change the syntax, I'm not going to change how functions work or introduce radical new concepts, that sort of thing. The big difference is the incompatibilities mostly come from the fact that I'm sticking to text mode for now, and it's indented for UIs. So, I'm

not sure that there's a good way to support standard in and standard out in Teliva. Though, that's kind of an open research problem. I'm still thinking about it.

So, there's like challenges like that, Oh, here's another one. So typically, when we program in any programming language, including Lua, we organize our code in files. And the files often double as modules like they do in Lua. And so, when you want to include code from some other file, you often have directive like require or import to make other functionality available within a given file of code. So, Teliva, unlike Lua doesn't have any notion of files or source code. All it knows about our top-level definitions. So, you can't use require anymore. Everything's in a sort of single global namespace. And that has some drawbacks. But for me, the single biggest thing that reason I went with that is, it makes the editing experience really easy to explain to a newcomer. I think not having namespaces actually is like a useful force to keep people from bloating their apps too much. So, I really want to give people lots of nudges to keep their app simple.

[00:32:32] KP: Are there any industries or use cases that you think are an especially good fit for Teliva?

[00:32:36] KA: Honestly, I'm not sure. In many ways, the trajectory I've been following like, particularly with text mode, is it sort of going against all of industry. The whole thing is MIT license, just like Lua is. So, in principle, anyone can use it. I think the whole market research side is still sort of open. I'm probably doing this all like exactly the wrong way, because I start out with these principles that I care about. And then I tried to find people who resonate with them. If I was trying to do a startup, I wouldn't be doing it this way.

[00:33:10] KP: Fair enough. But I mean, there are certain people that may be attracted to some of those features. I think we've talked a little bit about type safety. Are there other things that are novel that you think are really worth paying attention to?

[00:33:21] KA: Oh, I should mention that, unlike Mu, which was statically type safe, and also memory safe. Lua, is memory safe, because the high-level language is interpreted, but it's sort of dynamically typed. And so, it is different from Mu in many ways.

Honestly, I don't have a good answer for like, who should use Teliva. The way I see it is, I'm trying to foster a different way of doing software. There's been some echoes of it in the past. Clay Shirky had this article about what he called situated software, where instead of the way industry builds software today, where you have a few thousand programmers building things for that will be used by millions of people, you would instead build something that's just going to be used by you, or a small group of people around you like say a hundred or a thousand people. That kind of resonates with me. I think that's like a key way to avoid the market forces that cause complexity. So, I'm trying to do that.

Robin Sloan had a fantastic article last year, about how an app should be like a home cooked meal. And the analogy he made was that when you use Instagram, it's like going to a restaurant. But he wants to get that an app for photo sharing with just his family and he built it himself. And that's kind of like cooking at home. He talked about how hard that was and how the infrastructure we have doesn't make it easy. So, I think I'm trying to create infrastructure for home cooked software.

[00:34:48] KP: Makes sense. Text mode, I think has a lot of opportunities, obviously, not for graphical applications, but there are so many console-based things, developer tools, solutions along those lines. We're actually I think some of the tooling is laxed, could that be maybe an opportunity for things to gain momentum?

[00:35:06] KA: Yeah, definitely. I'm sort of keeping it very open. My goal is to just build as much of the software that I tend to build in Teliva as quickly as possible. And so, if I can get there in a couple of months, it's kind of open-ended what apps I will build. And I'm hoping that just by building a whole lot of apps on a constant basis, and cranking them out, I'll eventually have some interesting apps that people find interesting enough to start using. And if they start using them, then I think, hopefully, I've made it easy so that modifying them and forking them and remixing them and sharing them on the internet, they're all just little steps away from each other.

So, for me, I think I'm not trying to focus on a single kind of customer. I'm instead trying to focus on a new way of sharing software of collaborating, of appreciating each other's code. I think we sort of live in a pre-literate society when it comes to software, where if you write code, people just want to use it, nobody wants to read it or appreciate it. I'd like to have this analogy with like

regular literacy where, most of us don't write novels, but all of us can read novels. We can usually compare novels and talk about the differences between two similar block structure and things like that.

So, in my analogy, I don't expect people to be writing operating systems for themselves. But I would really like to live in a society where everybody can read an operating system, or some part of it, the part that they dip into for an hour and say something intelligent about it.

[00:36:44] KP: Do you have a vision for where the project should be in five years?

[00:36:49] KA: I would like a small subculture of people building and sharing apps using Teliva. That's sort of the goal. I think that was my goal with Mu as well and it didn't really happen. I think I had like, two or three people hack on Mu at different points in time. But it was always me and one other person at any given point in time. And I'm just trying to find the right sweet spot. Peter Drucker said that a business exists to discover a customer. I think I'm still trying to find a community that can organize around these principles.

[00:37:20] KP: Well, you'd mentioned the contributor's markdown file that a lot of repositories have. What does your say if you have one?

[00:37:27] KA: I don't have one. Actually, with Teliva, I don't have much of anything like most of the code that is just straight up. It's the Lua implementation and stuff like that. And basically, like stealing codes, left and right, but trying to be very careful about what I steal.

Let me step back to Mu to answer that question. So, the way I set things up in Mu was that I had a couple of started exercises for a while as open issues. These were not like regular issues in the sense that when somebody didn't finish them, I would close the issue. It was more an invitation to start with this branch and try to perform this exercise and in the process, hopefully learn more about Mu. I did that for a while. And lately, I have the sort of guided tour. So, there's a tutorial directory in the Mu repository, where instead of having to read all the different things I've written about Mu and also different places in sort of linearizes it a little bit and says, "Okay, go read the first. Task one is go read the first two paragraphs of the readme, and then try to answer this question. How would you build it?" That sort of thing.

So, it starts out really, really simple, and really hold your hand a lot early on. And then by the 16th exercise, I've sort of got it. I tried to keep it on an exponential scale. So, by the 16th exercise, you're doing, like, hopefully, really interesting things.

[00:38:45] KP: Where can people follow the projects online?

[00:38:48] KA: It's on GitHub. I guess there must be some way to share a link with the podcast.

[00:38:53] KP: Yeah, we can put it in the show notes.

[00:38:55] KA: Yeah, that sounds perfect.

[00:38:55] KP: And anywhere people can follow you and hear your thoughts and philosophies?

[00:38:58] KA: Yeah. So, I'm AK, and then my first name, Kartilk, and that's just where I write. I sort of have what they used to call a blog. I just think of it as my website. It's chronologically ordered. I tend to write like, once or twice a year, but yeah, that website is for like, when I have longer things to say, which tends to happen once or twice a year. And then I also have – lately, I'm actually most active on Mastodon, and I really encourage people to check it out. If they haven't already created a Mastodon account.

[00:39:31] KP: I'll have to get on that myself. We'll put that link to that in the show notes as well.

[00:39:35] KA: In the last couple of years, I spent so much time checking out all these fringe indie alternatives to mainstream software. So, I'm on Tilde.Club. I'm on Mastodon. I use something called TWtxt, where it's a social network like Twitter, except every tweet is on a single line of flat text file. And so, it's like this really simple feed reader where you just download text files from people you follow and then you have a little bit of software to like merge them together. It's pretty hilarious. I think it's pretty awesome.

So, I have a contact page on my website, and my contact page lists all the different ways you can reach out to me and I'm very open to being reached out to on any of them or new ones if

you suggest them to me, because I'm constantly trying to try out all the cool things that people are building that are you know, not trying to build for the mass, but have some soul to them, something they care about beyond just getting a lot of adoption.

[00:40:34] KP: Kartik, thank you so much for taking the time to come on Software Engineering Daily.

[00:40:36] KA: Thank you so much for the insightful questions. I really appreciated all the questions you asked.

[END]