EPISODE 1424

[INTRODUCTION]

[00:00:00] KP: Francesco Cesarini founded Erlang Solutions in 1999 with a mission to help companies adopt Erlang. In this interview, I speak with Francesco and Gabor Olah from Erlang Solutions. We discuss the Erlang language, its ecosystem and features like concurrency, resilience and scalability, that motivated adoption. We use Java and the Java Virtual Machine as a comparison point for Erlang, and its virtual machine, the BEAM. Lastly, we explore where Erlang fits best and contemporary software engineering projects.

[INTERVIEW]

[00:00:35] KP: Francesco and Gabor, welcome to Software Engineering Daily.

[00:00:38] FC: Thank you for having us.

[00:00:41] KP: So you're both with Erlang Solutions. I know Erlang as a language. And I assume there's a bit more to it. Can you give listeners just a general overview of what's going on with Erlang?

[00:00:50] FC: Well, yeah. I kind of founded Erlang Solutions back in 1999 as with a mission of kind of helping companies with the adoption of Erlang and other kind of similar technologies. And, yeah, we've today expanded to offices all over the world, and probably around 120 employees.

[00:01:10] KP: And what makes Erlang special?

[00:01:13] GO: Yes. Erlang as an ecosystem is quite unique in the market nowadays. Some programming languages are catching up to some of the features. But in my opinion, what makes Erlang special is the runtime, because it comes with some really interesting features. For example, a preemptive scheduler that lets software time systems implementation. The Erlang VM, which we call the BEAM, because it doesn't just run Erlang, but other programming

languages. It has some unique features that enables software time systems to be implemented quite easily as a world-class scheduler, and some other really nice features.

[00:01:51] FC: So if you think of it, I think the Erlang ecosystem consists of three things. One is all of the programming languages, which run in this ecosystem. And the oldest, very similar properties, which come from Erlang, the focus on concurrency, resilience and scalability. And I think, today, there are about 45 languages in the ecosystem. From your regular programming language, such as Erlang, and Elixir, LFE, which is Lisp Flavored Erlang. They've actually taken Lisp to the BEAM **[inaudible 00:02:21]**, all the way to languages used for smart contracts with crypto currencies.

The second item, I think, which creates this ecosystem, as Gabor was saying, is the BEAM virtual machine itself. So highly optimized for your kind of concurrency, scaling on multi core, and has built-in distributions **[inaudible 00:02:45]** I'm aware of, I think, which is kind of used in Anger with built-in distribution.

And this allows your different VMs to be transparently clustered together. So orchestration was something we were doing back in the 90s. Adding and removing VMs to the cluster during runtime. Fast forward into today, I think the main – There's a whole team at Ericsson, which is maintaining Erlang and the BEAM. And I think a lot of the effort they're putting in is on improving the virtual machine for your modern-day architectures and applications. So they just recently added the JIT compiler, which has huge performance gains. They're focusing on multicore optimizations and kind of large-scale concurrency. So that's where the preemptive scheduler, for example, comes in. And the result is, when we started, you could have maybe 20,000 processes running in the VM. Today, we're talking about millions of processes, which kind of all run in parallel. The third kind of pillar in the ecosystem is something we call OTP. I don't know. Gabor, maybe you want to mention something about it.

[00:03:53] GO: Erlang doesn't just come as a programming language or the whole ecosystem, but comes with some design philosophies that enable us to write these highly concurrent systems. Back in the old days, and even modern languages, you have locks and mutexes to manage concurrent or parallel runtimes. And that's not easy, because the programmer, when rise to business logic, also needs to think about what to do about the data structures. What can

interfere with that business logic? In Erlang, you don't have that. But to build that kind of application, you need some common understanding how to build that. And Erlang comes with one of the standard libraries. It's called OTP. It comes from the telecom background, but nowadays, it's just a standard set of building blocks that you can use without thinking about how, in the background, it solves that difficult problem of synchronization.

[00:04:45] FC: You mentioned the word telecom. And I think people are going to start yawning and jumping out the windows. But if you go back – How fun is that? But yeah, if you go back to kind of the late 80s, early 90s, the only system which really had to scale and could never fail were telecom systems. And when a computer science lab said about to figure out how to program and how to develop, not even program, but how to develop the next generation of telecom switches, the solution they came up with was Erlang. So they set out to solve the problem of scalability and reliability. The solution happened to be a programming language.

But soon after, why are we speaking 25 years old? It's because the Internet came about. And now, today, pretty much any system connected to the Internet has to be not only scalable, but also resilient. And that's why we're speaking, I think, the properties, which they came up with back then, are just as relevant, if not even more relevant today.

[00:05:45] KP: So I absolutely understand why concurrency would be vital to telecom. And it worked well, right? Very rarely do you hear anyone get an error on their telephone system. That isn't because they went into a tunnel or something. It's pretty reliable. Why do those values translate well to other domains in other areas?

[00:06:03] GO: It's a very, very interesting topic. Because nowadays, when we have these packetized services, everybody accesses their favorite internet site via HTTP. There is no tunneling. No fix landlines. All of that is completely foreign to this world. But what we want is low latency. We don't really want a system where my request to a server interferes with somebody else's request.

Also, you can't really manage workload, because, for example, if something goes on to Hacker News, then your site will explode in popularity, most probably. How your site degrades is a big question.

So when we talk about this resiliency, we also want a nice predictable failure pattern in our applications, preferably low latency when there's nothing going wrong. And the latency shouldn't explode, but be kept manageable, for example, when the load gets a bit higher.

[00:07:03] KP: So, Kyle, I think you mentioned concurrency. I think concurrency and scalability go hand in hand, because – Concurrency, when you have processes, it makes reasoning of your problems much, much easier. What you do is you go in and you think about a single instance of what you're building. It could be a player in a massive multiplayer online game. It could be a WhatsApp message. It could be a bank transaction. You're transferring money from one account to another. And how do you scale WhatsApp? How do you scale a massive multi-user game, or financial switch, is you scale it by creating more processes each handling a single request?

And when you're using an ecosystem where you got a virtual machine, which can handle millions of these requests, that's really where you start talking about scale. Now, this will come at the cost of speed. And as Gabor say, I think what you need to make sure is that when you're handling a million users, that your latency doesn't go out the roof. And more often than that, it won't. It will stay fairly constant. So by having the virtual machine allowing you to scale out of the box, you as a programmer just need to focus on the business logic and not the underlying mechanisms, which let your business logic run.

[00:08:27] KP: My experience is a little bit more with Java, but I'm drawing some strong parallels to the JVM, and in particular, how it's matured and allowed other languages to be part of that same ecosystem. Is that a fair comparison?

[00:08:39] FC: Very fair. I mean, Java and Erlang came about more or less at the same time. When I was at the computer science laboratory, so back in the mid-90s, this was before Erlang was being used in Anger in any project. And I remember someone coming up and giving me the Java whitepaper, and, “Oh, a virtual machine. A garbage collector. Oh, automated memory management.” I actually got a sense of visual. And at the time, we also had green threads.

And the two languages, I think, are very similar, because they're more or less the same age. The big difference, I have to say, is the direction in which the JVM took versus the BEAM. I think the JVM focuses on parallelism. The BEAM focus on concurrency. The JVM focuses on speed. And speed, you get through shared memory, you get through threads. The BEAM focuses on scalability. So how do you scale your system? How do you manage 10s of 1000s of users or millions of users without affecting the property, the properties of the system? So there are differences, and there are also differences, I think, in the types of problems where in which you would use the JVM versus the BEAM.

[00:10:02] KP: So scalability is an interesting one. That's something I've had to put a lot of thought into in various projects I've worked on. What's it like to consider scalability as a user of the BEAM?

[00:10:12] GO: So, Kyle, you mentioned that you did work with the JVM, or your experience with the JVM. For example, when we talk about scalability, we can talk about two orthogonal directions. One is you can scale out or you can scale up. And both has its own challenges. Now, in Erlang, you have two different approaches that you can do. But the toolset that you need to work with is exactly the same. You need to isolate your individual requests to processes and let those processes spread out to either many schedulers, for example, that the Erlang, the BEAM gives you. Or you can spread those requests out, those processes, to multiple nodes when you have a clustered environment.

And that bigger thing functionality means that when you write your application, you don't really need to think about concurrency. As long as you find a natural way to cut your applications into small pieces, that is not a pipeline, but the individual requests that are isolated independent from each other can just coexist. And they each can progress at its own pace.

For example, think about web requests. If two different people creates a web request to a server, they are usually not dependent on each other. They could be served by two different web server instances. In Erlang, you can run this web server inside of an Erlang VM. I keep saying Erlang VM instead of the BEAM. Bad habits die hard.

So in the BEAM, you can start your web servers as a single process that will create a separate process for each web request. And depending on whether you have enough resources on a single machine, or you have a clustered environment, you can spread these processes out to multiple machines, if necessary. But when you write the application – This is what I mentioned. You write the business logic without caring about this. This is the underlying framework that takes care for you, or use a library that that makes it happen. So for me, scalability in Erlang is not about thinking about scalability. It's more like just using Erlang and let Erlang shine.

[00:12:28] FC: I think the best comparison here, and this is also a good comparison between the BEAM and the JVM, is think of – So I live in Rome right now. I've just moved to Rome. We often go out and have pizzas. Now, how would you scale a pizzeria using Erlang? Well, what you do is you get your baker, and everyone baker will do one pizza. So they'll flatten out the dough, then they'll put tomato sauce on it, and **[inaudible 00:12:59]** cheese, and then they'll put in all of the ingredients on top of it, put it in the oven, and then take it out when it's done.

And the way you would scale that is you would get many bakers each doing one single pizza. And so you need thousand pizzas, you have thousand bakers all doing them. Versus Java, what you would do is you would have one person maybe focusing on doing the base, a pizza base, and then they pass on the base to someone else, who puts tomato sauce on it. Then they pass it on to another person who puts a cheese on it. And they pass it on to a fourth person who puts all the ingredients on it and into the oven. So obviously, doing it the Java route, that's usually what we call parallelism. The parallel route is the fastest way of doing a pizza, because you're going to get a pizza very, very quickly. In the Erlang route instead, what you do is it will take longer for one person to go through the whole process of doing the pizza, but you throw more people at the problem. And that's how you scale.

[00:14:04] GO: And for those who are interested in a more technical example, take for example RabbitMQ.

[00:14:07] FC: There's nothing wrong with pizza.

[00:14:10] GO: I know that everybody's interested in pizza. I'm interested in pizza now. It's getting late. So I'm more interested in pizza than RabbitMQ now. But the actual example is that

take for example RabbitMQ, which is a message broker. You define queues and push messages to queues. Each queue can operate independently. Now, when we talk about, for example, scaling RabbitMQ, then since each queue is implemented as a single process, you can utilize that.

If you, for example, run out from the capacity of a single queue, then you can create another queue and let's say shard your messages. Or if you have completely independent applications who use different queues, then you can just define these queues in a single broker, and they naturally scale, because the processes are run the logic independently from how many cores you have. Let's say you have a single core machine with two queues, then the scheduler in the BEAM will take care of figuring out how these two processes should intellect processing the messages.

In this case, for example, in a single core environment, which is more and more common nowadays with shared resources, for example, in Kubernetes, then you don't really have parallelism, because that single core cannot do two things at the same time. But the scheduler can take one use processing capacity ready to run – Or processing load, ready to run for a set amount of time, and then take the other one and keep repeating it.

And when you write this kind of application, you don't need to worry about that, because the scheduler is preemptive, which means that you don't need to put, for example, predefined points where the scheduler can un-schedule your program. And this is a major difference, for example, from other languages out there. And to my knowledge, I'm not aware of any other major ecosystems with a preemptive scheduler.

For example, take the Java example. You have lightweight threads or actors in the Java ecosystem. Where they differ is that if you use a library, which is not developed with the concurrency in mind, you may end up with – Or parallelism in mind, you may end up in a situation that that library just takes too long and skews up this nice latency property of your system. But if use Erlang, you don't need to worry about that. Because if, let's say, calculating the length of a list takes one second, then the scheduler will stop processing that calculation when it runs out of allocated time, and then schedules in another thing. And when that finishes

or runs out of its time and its schedules is back the original task. And it continues on till all of them are finished, which gives a really nice property.

Now, if you take that single core system with two moving components to a chain that has four cores, for example, then each of these processes will roughly aligned to a core. It's a bit more complicated than that behind the scenes, but it measures in reality that way. And you can have proper parallelism and double the power. And you don't need to change the source code. You don't need to change the mindset how you program your application, as long as you cut it down to the concurrent or the problem.

[00:17:30] KP: Being able to set aside some of these concerns about currency and focus on my code is extremely appealing to me as a software engineer. That's kind of where I really want to spend most of my time. But if I'm exposed to this for the first time, I'm going to have some concerns, because that's not a typical experience. How do people looking at the language and the whole ecosystem get comfortable with this idea initially?

[00:17:51] FC: You really need to start thinking concurrently. And I think the point you make is the hard part in learning Erlang is not the syntax. **[inaudible 00:17:58].** It's not the syntax. It's not the semantics. Those are things which take a few days. The real hard part is actually unlearning all of the habits and approaches you've picked up with other programming languages.

And Erlang is a language which really makes you change the way you think and reason around programming. And the reason for this is that you need to actually start thinking concurrently. And that's how you design and you model your systems. You need to start thinking of having a process for each truly concurrent activity in your system. And the question, what takes a while to learn? Is what is a truly concurrent activity in your system?

If you think of instant messaging session, for example, a concurrent activity is not the session itself. The session itself, when a user logs on, is just some state. You store in an in-memory database or in a cache. The actual concurrent activity is a message being sent. It's a message being received. It's a status update. It's a login, or logging in, or logging out.

For these concurrent activities, that's where you go in and you create a new process for them. And it takes a while to start approaching the ecosystem with this mindset, because most people are coming from a background where they're used to using threads instead of lightweight processes, is they'd probably create a thread, or did create a process for a user when they log on, which goes in and handles all of the requests for this user.

[00:19:29] KP: Well, I know, in my experience with Java, I'm thinking of a particular incident where I wrote some code that was going to be run at really high scale and high velocity and was throwing some very rare exceptions that we eventually, with the help of another engineer who had some specialized tools I didn't know was able to figure out, it was some configuration of the JVM that they were able to do some black magic and fix. Is there a similar need in Erlang? If I write some Erlang code that scales up, will I have to go find some DevOps person who knows the secret tools to help me to find rare cases?

[00:20:02] FC: Incredibly rare. Yeah, it's incredibly rare. It does happen. But we're talking about extreme scalability right here.

[00:20:11] GO: And one of the nice features of the BEAM is that it's a VM that gives you a window into how the program runs and gives you capabilities of faking traces and log what the VM does for a specific function, for example, during the runtime. And this is a feature that not a lot of people uses. And I wouldn't encourage doing it in production unless you really, really, really know what you are doing. Especially during development or testing purposes, you can create another process inside the BEAM annotates what the VM is doing. And that, for example, logs into a file or gives you a snapshot of what is going on with a specific function, and gives a very unique way to debug applications.

For example, in the last 12 years of my Erlang career, I've never started the debugger, because I never needed one. Some people use printf's. I don't use printf's because I don't need one. When a function is called and I don't know what it's doing, instead of printing it out, I use tracing, or the tracing infrastructure that is usually shipped in every Erlang system, and including Elixir, because it's part of the standard library. You can use that and understand what the program is doing in a very reliable way.

[00:21:26] FC: I mean, the introspection is really second to none. To the point Gabor was talking about, tracing, what you can do is, in code, you actually run it. And I've done this in code in high-volume, high-throughput code in production looking for extreme edge and borderline cases, which you would never be able to reproduce in a lab, which we were seeing happening once a month, once every other month. What we did is you're able to go in and examine on variables parameters passed to local function calls. And if they follow some particular – If they have a particular value, or you can do comparisons, and simple logical operations on them. And if they trigger certain parameters, you can go in and fire off a trace message.

And so all of this internal introspection is done with minimal overheads over the traffic, which is going through your system. So this way, you're actually able to go in and find obscure bugs, which you would never find otherwise in a regular system. So amazing introspection. And cut it out here. It almost sounds like a sales pitch.

[00:22:41] GO: Exactly. It's one of those features that I never found anywhere. And whenever I program, for example, in Python, or Java, or basically any other language, I always feel that even if I get a shell, I just like to know what is going on behind the scenes. And while the debugger works very well, for example, a single threaded program for a concurrent programming, it's not a good solution. Because if you stop the execution of the program, then you might hit, for example, timeouts that are triggered. Or another machine that you haven't stopped just carries on operating. So you can't really rely on spending half an hour trying to figure out what the individual parameters are doing. Instead, what you do is take a snapshot of what's going on without breaking the current execution threads or the processes or their interactions, and then analyze it as a postmortem of the fact and see what's going on.

[00:23:36] KP: Well, we touched on the telecom example. I don't know if every listener is going to think that's boring. But just in case some do, could you provide a more contemporary example of a company for whom they determined it was the right ecosystem to adopt?

[00:23:49] GO: Some big companies out there who use Erlang who are not in the telecommunication industry. Some of these examples are, for example, WhatsApp, Klarna, Riot Games, Discord, which is definitely not in the telecom industry, or Forza Football.

[00:24:04] GO: The Bleacher Report – I don't think Bleacher Report. Bleacher Report migrated from Ruby to Elixir. And in doing so, they were able to reduce their, basically, server needs by around 90%. So just imagine what that is doing for the environment in reduced energy consumption.

[00:24:22] KP: Yeah, good point.

[00:24:23] FC: Amazon might not be happy about it. But yeah, it's –

[00:24:28] KP: Do you have any insight into what brought that 90% gain?

[00:24:31] FC: So what [**inaudible 00:24:32]** server needs is the scalability properties you have of the BEAM. Every request coming in, you just pull a new process. And a BEAM virtual machine can handle millions of processes in parallel. And so, all of a sudden, you've got a million users all sending in their request. You can handle them with much less server capacity, because all you do is you spawn new processes in each VM. You just need some large instances. And that's how you scale. How do you scale Ruby? Well, you fire off more instances, and you scale horizontally. With the BEAM, you scale both vertically as well as horizontally. And that's what we're seeing right here.

[00:25:12] KP: And in terms of deployment, what is the production environment look like? How do we get Erlang code out there?

[00:25:17] GO: Nowadays you can just create a release and upload it into your server, or very popular technology using Docker or any kind of containerized technology, and how you orchestrate it, for example, Kubernetes. That all works just as well. I don't see any real difference from how you deploy any other application. During the **[inaudible 00:25:37]** process, you create a release and move that release into your production environment and start it up.

[00:25:45] KP: But in terms of the way – If I understood correctly, you describe that when maybe one machine is running out of resources, it could fall over to another in the cluster. If I build a couple Docker containers, how do they synchronize and orchestrate?

[00:25:58] GO: So when you have multiple Erlang virtual machines, then you can set up a clustered environment. That's basically a configuration parameter. Once the Erlang virtual machines are clustered, then they can communicate with each other, read location transparency. So that part, when you design your application with multiple instances in mind, you need to set up some kind of named discovery service. There's built in ones in the Erlang ecosystem. There are some libraries that work really well. But the processes that Francesco is talking about, these small, lightweight processes, you can name them and communicate with them. It's completely transparent. So when you send a message from one processor or another, the virtual machine takes care of it and delivers that message to the other process regardless of where it is in a cluster.

[00:26:52] FC: We often get questions, “But why should we be using the service discovery mechanisms libraries in Erlang? They already exist in Kubernetes.” And usually, I think the answer there is, well, you remove a dependency towards Kubernetes. You still use Kubernetes for your orchestration. But you want to make sure that what you develop is actually not dependent on a particular tool. And that's why you use these mechanism Gabor is describing.

[00:27:21] GO: And the other part is that not everybody is using Kubernetes.

[00:27:24] FC: Exactly. Exactly.

[00:27:24] GO: Lots out there who don't depend on this quite big technology. It's a very useful one if it's done correctly. But there are still lots and lots of applications where you just don't need that kind of scale. If you run out of one server, you can create another one. And to be honest, not everybody needs hundreds of servers to do their workload. If you can stop somewhere between low 10s, then that is a really good candidate for some homegrown installment, for example, and orchestration.

[00:27:53] FC: I think I'll throw in Phoenix here, which is kind of a framework for your web API's written in Elixir. I think it's kind of the next generation, say, of Ruby on Rails. And back in 2016, if I remember correctly, they went in and managed to get 2 million TCP/IP connections running on a single large instance on Amazon.

So this is basically 2 million users, which are able to connect to one single instance. How many applications needs to handle 2 million simultaneous users? And if you got one, which instead of 4 million users, you fire up a new instance. And you're up at 4 million connected users actually using your system in parallel. So it's often adding Kubernetes and adding – Dockerizing, and everything, is adding an overhead, which for many use cases is actually not necessary.

[00:28:54] KP: Well, for many reasons, as soon as you get into some distributed computing, you have to accept you're going to live in a world where there will be failures. How do BEAM and Elixir perform when they catch on fire?

[00:29:06] FC: To quote Joe Armstrong, you need to build your system with full tolerance in mind from day one. So you need at least two computers. You say catch fire, Joe Armstrong says you're one of the two computers might be struck by lightning. And what you tend to do is you've got two of everything. That's how the telecom systems achieve five nines availability. You've got redundant power supplies. You've got redundant hardware. You've got redundant networks. And if one fail, you've got your fallback to the other.

It's a similar mindset when you're dealing with problems in the ecosystem which may never fail. But what you need to replicate right here is state. And you will copy your state and your data for fault tolerance. And then you'll go in and your replicated for scalability. So what that means is that if you're sending a request to one particular server, that server goes down. Requests get redirected to another server, which has a copy of the state, and can pick up where the first server left off here. So that's one approach, which is very normal.

But that's all on losing the whole server. You then scale it down. What happens if you lose just a single process? So think of telephony server where every phone call, which is ongoing, is a process, is represented by a process. If there's a bug in your code, or your state gets corrupted, what you do is you cause your process to crash. And what you do is you can go in – This crash is detected by a supervisor, which goes in and recreates a process and tries to recreate the state of this process before the crash. And by recreating that state, and hopefully that state had gotten corrupted somewhere along the line, by recreating it, retrieving the data from the original source, hopefully, this issue will have been addressed. And it's called a supervision structure,

where you create supervisors who monitor the workers and other supervisors. And the sole task is to go in and restart the process in case it fails.

So we usually refer to this as the ladder crash approach, where we let processes terminate or crash. But we don't ignore the crash. We actually go in, and we handle bugs, but we handle them differently than what you might be used to. And they're handled in a generic way by the supervisors. So again, it simplifies the code base. And it makes it much, much easier to grow and develop and manage in the long run.

[00:31:43] GO: And Francesco mentioned a lot of really useful techniques that are provided by Erlang, Elixir, the standard libraries around the programming ecosystem. But I'd like to highlight another one that is not unique to Erlang, and probably the most important one. Once you move away from a single instance model and you move into a concurrent model or something that involves networking, you need to think about delivery guarantees, because you can either have at least once or, at most, one delivery guarantees. But what you cannot have is exactly once delivery.

And when you design your application with that, you need to implement usually item-potent messaging, which means that if you retry a message or retry an operation, then it will have the same effect. So it will not cause any side effects outside of the expected part. And then you build your application with that in mind, which is your generic how to design an application that is not a single instance. Then you get an ecosystem that supports the tooling around, because it is not fair asking for Erlang to say that, “Okay, now we have messaging, we have processes, we have the supervisor tree, we have let it crash. But I cannot lose a message. And the message only leaves in-memory and let the whole ecosystem solve it.” You need to be able to retry messages. You need to have a cord that doesn't crash that knows about what is going on. Either you outsource it to the user. For example, most web applications do that. For example, if something happens with your request, the user is expected to press the refresh button. Or back in the telephony days, if something happened inside these big telephony switches, the user was expected to redial. So this is not a completely new idea. Or you can create a core. For example, what the most popular web framework in the Elixir and Phoenix does, or the Erlang library, that it uses cowboy, uses behind the scenes, is that it has a core part of the application that receives your HTTP request and then creates a new process for that, will handle that HTTP. If something

happens during that request, then that process can crash. And this core part will catch that and will serve your 500. That is a really stable model. Because when you write your application, you don't need to care about in the middle of, let's say, user credential lookups that what happens if you lose access to your user's database? You just handle it as a situation that you don't know what to do. You don't know what is the correct response at that time. So you let the core say that, "Okay, this is a 500. Let's try again later."

And obviously, as you try to move some of these requirements back to a business requirement that has a different function for these, you can move that into a handled case where you don't let it crash, but assign a specific user function to that case. And that's the ease of development in Erlang that you can do. Because in other languages, you really don't want to let a threat crash, for example. That is usually not ideal.

[00:34:54] FC: So basically, we take care of our bugs. But we just take care of them in a slightly different way to what you might be used to.

[00:35:01] KP: What's a typical learning curve around that? We've touched upon a couple important ideas, like thinking concurrently, or working with item potent messaging. How long does it take for a seasoned software engineer to adapt to these new ways of thinking?

[00:35:15] FC: Erickson did some studies. And I wish they had published the results. But they did some studies back in the early 90s. And they, for someone straight out of university, who's basically used to switching across programming languages, they felt it took about a month learning, first, Erlang, and then OTP. And your they were productive within about a month. Versus someone who had been working with a particular technology, and this particular technology, it would – So someone, which was much more seasoned programmer, it would have taken about three months to become fully productive.

Now, obviously, when we're talking about a junior versus a senior, we're talking about different levels of productivity, which also explains a longer time. But I think a lot of the time with the three months got spent not so much about learning to think currently, but unlearning approaches and paradigms in other programming languages. And I think the team, which the study was done on, is the team which built the GPRS, which was built in GPRS back in the late 90s. I think

it was ’97, ‘98. And this is still very much code, which is in production today in Erickson's 5G systems, because GPRS became 3G, 4G, and today, 5G.

[00:36:33] KP: You've mentioned OTP. I'm not sure if we've defined that yet. What is OTP?

[00:36:37] GO: So OTP stands for Open Telecom Platform. But this abbreviation is no longer used, because it doesn't just contain the telecommunication part of the platform, but all the necessary building blocks. So usually, I define it as the standard library for Erlang.

[00:36:54] FC: I think we should never ever, ever mention the T word in OTP. It was named at a time when Erlang was not released as open source. And it was used internally within Ericsson. You have to serve a particular purpose. But what I usually define OTP is – What OTP does is actually introduces a programming model, which makes your reasoning around systems, which is distributed, scalable, resilient, easy, and almost natural. And it does so by forcing you to architect a node. So a node is an instance of the BEAM running in a particular way and providing tools and middleware, which rely you to follow these design principles.

And when you do follow these design principles, different systems, which do very, very different things, are actually done – And solving this is done in a similar way. So they'll go in and hide the complexities you might find in the of system you're developing. They hide the complexity is by putting these tricky parts into reusable libraries. And these libraries will often handle all of the **[inaudible 00:37:58]** borderline cases, so you don't need to think about them. So what you as a developer need to think about and focus is the actual problem you're trying to solve and not the accidental difficulties, which are brought forward by using maybe tools, which are not adequate for the task you're trying to achieve

[00:38:17] GO: It’s not only the standard library. But I'd like to second what Francesco talked about the libraries built on top of this. For example, Phoenix. If you are writing a web server in Elixir, then if you don't think about the concurrency and you just write an implementation for each API call, for example, then you get it out of the box from Phoenix. So there will be a very long way before you really need to start building around concurrent model.

And by that time, probably will get quite confident in what you are doing and get familiar with this time. So when we are talking about couple of months to get into this system, you can read many success stories where people who were familiar with, for example, Ruby, which is more or less single-threaded, came into Elixir, and they could write applications, which were highly concurrent out of the box.

[00:39:11] KP: Interesting. Make sense. Well, to wind up, I'm wondering if we can talk a little bit about what you're seeing in your work at Erlang Solutions. For example, when you're onboarding new customers, I'm curious if that's because they've sought you out for your expertise after identifying that they want to move to beam in this ecosystem, or if you need to evangelize the platform at all?

[00:39:33] FC: So, yeah, I think most of our work doesn't so much consist in selling or expertise. It consists in convincing the customers that the ecosystem, Erlang or Elixir, are the right tools for the job. Most of our customer will come to us, because at some point or another, they felt the pain points which the ecosystem solves, and pain points of scale and of resilience. So I think evangelizing I think it's more kind of helping them assess if Erlang and Elixir are the right tools for the job. And we usually do that through a prototype or a proof of concept, which is then used for a go, no-go decision.

[00:40:13] GO: And another leg that we stand on is, for example, our RabbitMQ services. So we do RabbitMQ consultancy. RabbitMQ is one of these very popular staple Erlang-based applications out there. I'm always surprised when I see it in companies where they never heard about Erlang before. They just run it and it does what it does, and until it doesn't, and they come to us to help us solve it. And our expertise in Erlang come in really handy, because it's a complex system. Sometimes you really need to understand what's going on behind the scenes and how to fine-tune it. And this is what we do best.

[00:40:50] KP: And are there any good rules of thumb for an enterprise who's thinking this might be the right choice for them?

[00:40:56] FC: So the right approach is always start off small. Start off by going in and thinking of what your pain points are. Are you having your stability issues? Is your operational team

being woken up in the middle of the night? Are you having – Because the system's crashing and failing, are you having issues handling increase in requests and demands for your services? If those two are usually the cases, then it's well worth your spending some time working on a prototype. So we're talking 30%, 40% days in addressing one of the pain points you're feeling and seeing if it actually addresses and gives you the solution you're hoping for.

So yeah, this is usually how we approach it. And I think the rules of thumb is to think, “Is what I'm trying to do scalable? Does it have to be resilient?” And in many cases, even if it doesn't have to be scalable and resilient, are we dealing with endpoints here? Are we dealing with a truly kind of concurrent system, which we're implementing? And if so, it's very well worth exploring. I think there are many case studies out there, there are many conference talks available on YouTube which showcase how Erlang and Elixir have been used in, well, not only the telco space, but in FinTech, in healthcare, in IoT, messaging. Yeah, there are many, many verticals out there. And they will help you assess if it's the right tool for the job even before you go in and start your proof of concept.

[00:42:31] KP: And can we leave listeners with any good links or a place to go to learn more?

[00:42:35] FC: I would like to recommend going and look at all of the Elixir conference talks, all of the code BEAM conference talks. They've got YouTube channels. There's a lot of talks, which have come even during the pandemic, which have been done virtually. There is the Elixir website. It has some excellent tutorials and links to tutorials on how to get started. There's also the erlang.org website as well, which has been up and running now for almost 23, 24 years. And finally, also the Erlang Ecosystem Foundation, which is well worth mentioning, ErlEF Org, which is there to basically help companies adopt BEAM technologies and technologies in the Erlang ecosystem.

[00:43:22] KP: Very cool. We'll try and put some of those links in the show notes for listeners to follow up. Francesco and Gabor, thank you both so much for taking the time to come on Software Engineering Daily.

[00:43:30] FC: Thank you for having us.

[END]