EPISODE 1493

[INTRODUCTION]

[00:00:00] SPEAKER: Profiling has been part of the software engineering toolbox since the 1970s. But it was long regarded to be too high in overhead. So, it was only done ad hoc. The problem with profiling that way is that you never catch the moment in time, where, for example, an incident happened. Continuous profiling is the systemic capturing of profiling at all times. Google has been doing it for over a decade. But it comes with both challenges of how to capture the data at low overhead, and also storing and queering the enormous amount of data that continuous profiling brings with it. Polar Signals created the open source project Parca to bring continuous profiling to the masses. Frederic Branczyk, founder and CEO of Polar Signals, joins the show to discuss the uses of continuous profiling, as well as the challenges in building everything from the low overhead profilers using eBPF to purpose-built storage engines for the high-volume data it entails.

[INTERVIEW]

[00:01:03] JM: Today, we're talking about profiling, which is, I would say, one class of observability. Can you describe how profiling for a program or a entire code base fits into an observability stack?

[00:01:24] FB: That's a great start. So essentially, profiling the way that we're seeing it with continuous profiling, and we can get into that in a second, how that kind of differentiates from the classical profiling. It kind of complements your existing observability tools. And when I say existing, I mean things like metrics, logs, and distributed tracing. In a way, it really shines a different light on your applications than those two. All those are still useful, but profiling allows you to understand where resources like CPU or memory are being spent down to the line number, right? We're looking at it on a per process level, as opposed to like with distributed tracing, we're not tracing a request through a system, for example.

So, we can down to the line number tell you where CPU resources are being spent, for example, and that, really, it's a completely different resolution of data that we're getting there. It's just a completely different kind of insight into the application. So yeah, I think the ability to just understand how our code is being executed, is I think, the way I would sum up profiling.

[00:02:42] JM: Why would that be an issue? Why would it be confusing? Why wouldn't I know how my code is executing?

[00:02:49] FB: Yeah, that's a great question. It turns out in reality is often very different from the way that we think and something that we, at Polar Signals and with our open source project, Parca also do is what we call whole system profiling. So, we don't only – the kind classical profiling, that maybe listeners are already familiar with, is, we look at our process for let's say, 10 seconds, and we record everything that it's executing within that period of time.

The kind of problem with that is that we're truly only getting a snapshot of that period of time, and we're only getting that snapshot of that one process. And so, with whole system profiling, as well as continuous profiling, we're actually looking at the entire infrastructure and all of the processes in the infrastructure and we're doing that throughout all of time, so that we're never missing anything in our infrastructure. So, I'm saying all of this, because not only in our code, does it tend to be very unexpected, where resources are being spent. But actually, that kind of translates to our entire infrastructure. Just recently, we were talking to an open source user who kind of was starting to use Parca, and immediately they could see that in one very specific sidecar in their Kubernetes infrastructure, they're spending 55% of their CPU time, right? They just didn't realize before that it was this one particular container that they have in every single part of their deployments. This tool kind of immediately shows it because it merges everything in your entire infrastructure into one report, basically.

[00:04:30] JM: Could you give me an example of, maybe talk application level that – an example of a code base that might be executing in a way that I don't expect or executing sub optimally?

[00:04:45] FB: Yeah, great question. So, I mean, it can be anything from like your database, for example, right? You're doing some requests against your database, and because we're profiling everything, we can also look into the database and we're seeing that it does a very expensive, like whole table scan or something like that. That's something that you would then fix by adding an index, for example, to your database. But often we see that, you know, we're not correctly pre-allocating memory, or things like that or we're doing expensive string concatenations, where they're completely unnecessary, very often is really small things that can make a huge impact.

[00:05:27] JM: When you talk about profiling a code base, are we talking about when that code base is executing? Or is it a pre-compilation analysis? Or maybe you could talk about how this works in practice?

[00:05:40] FB: Yeah, absolutely. So, the way that our tool works, and there are a variety of tools that basically do the whole spectrum, but the way that our tools work, is they're eBPF based. And so, what we essentially do is, we use the Linux perf subsystem, and we kind of in Kubernetes, for example, we find the particular C group that belongs to a container, and we attach a profiler to that C group. And so, when your code runs in your production infrastructure, that's where we attach the profiler and kind of just profiled the process throughout its entire lifetime.

[00:06:20] JM: What kind of introspection into that code base do you get from eBPF? We've done some shows on eBPF, maybe you could, since it's a fairly new topic, maybe you could talk through what that means.

[00:06:33] FB: Yeah, so eBPF, maybe I can take at least say one or two sentences about what eBPF is. Essentially, eBPF is, you can think of it as a virtual machine in the Linux kernel where you can safely execute code. So essentially, the verifier, the eBPF verifier, it only lets code through that it can verify will definitely halt. So essentially, it's kind of reduced what a program can do so that it can solve the halting problem for small set of programs essentially. What that means is that you can have things like potential endless loops, basically, all loops have to be able to be unrolled by the compiler. That then there's a maximum instruction number that can be executed. I believe it's something like 65,000, or something like that. But don't quote me on that. But it's limited, essentially, what you can execute.

Basically, the data that we're getting was always in the Linux kernel, and the way that we happen to use it, as I mentioned previously, we use the perf subsystem and the perf subsystem essentially allows us to create an event. And in our case, we attach that event to C groups, and we can configure certain parameters. In our case, we use something that is overflow based so that, essentially every x CPU cycles, our eBPF program is called. So, what that amounts to is that you can think of it as we're looking at the current stack trace a hundred times per second. But because it's overflow based, we actually are only actually recording data when there's actual CPU time happening, which is important, right? You don't want to do unnecessary measurements when nothing's happening.

And basically, from eBPF space, all we're getting is a stack that is just memory addresses, and we just kind of record these memory addresses. We say, we give it an ID, and we say we've seen this stack trace, and then we count how often have we seen the stack trace. And we do that over a 10-second period of time. And then we export this data to user space, because like I said, all of this is executed in kernel space through this virtual machine, basically.

And then in user space, we essentially start to make sense of this data. We as humans can't really as much as I like to pretend, I can make sense of just like memory addresses, right? They need to be kind of converted into something that humans can understand, and that's where our user space agent comes in, kind of looks at various places where it can find metadata to symbolize it so that we as humans can see our function names, line numbers, file names, and so on.

[00:09:24] JM: So, let's say I have a program that's executing, it's running across – they've got 20 instances of it running across containers with variable loads on them. You've given some description of how eBPF would introspect that. Maybe you can talk at a higher level about Polar Signals, which is your company and give a little more discussion about what you're doing with the data that EPF gathers.

[00:09:52] FB: Yeah, absolutely. So, the collection is really just one side of the equation or maybe even – yeah, like one of three sites. The collection obviously is really important. And through eBPF, we're able to collect this data at very low overhead, because we can truly only collect the data that we're interested in, at exactly the granularity, and exactly the format that we needed. So, by controlling that whole pipeline, essentially, we can really optimize it very much so that we can lower the overhead to just about 1% of overhead, so that you can truly run this in production all the time, right? That's really important for all of this to work, essentially, so that people are comfortable with running this in production all the time.

Because what we're finding is once you have this data, and once you can see where improvements can be made, it's kind of a classic data problem. If you didn't measure it before, if you didn't know, if you just didn't have the numbers before, you can really make informed decisions about improvements that you wanted to make. So, that's essentially where Polar Signals then comes in, you run the agent on your host, and you can send this data to Polar Signals' cloud, where it is then stored in a columnar database that we specifically wrote for this purpose. And then we also have a query engine and a front end to visualize all of this, with visualizations that if people are familiar with profiling are probably already familiar with things like flame graphs, or the upside down version essentially, are called icicle graphs, which was a happy little accident with the company name actually Polar Signals, right? Icicle graphs. I didn't actually know that they were called icicle graphs until I found out after I named the company. Turns out Brendan Gregg also coined that term. But that was funny.

Yeah, just tables, and sometimes people are familiar with something called call graphs, where as opposed to the flame graphs where you kind of see cumulative leaves, you kind of see just a cyclic graph of functions calling other functions and you're seeing where, like CPU time is added, and how much in cumulative that function is spending. So yeah, that's kind of what Polar Signals is. But we do also have an open source project called Parca, P-A-R-C-A, where people can kind of try this on their clusters if they can use a cloud product or something like that. Yeah, that's Polar Signals in a nutshell.

[00:12:26] JM: Got it. So, you store the data that you're gathering from these eBPF traces?

[00:12:33] FB: That's right. So essentially, the first thing that the agent does is it transforms it into like an open standard and profiling called pprof, which was created by Google. And once it's kind of transformed into that format, and sends it off to any server that can handle that format, essentially. And that can be either Polar Signals' cloud or our open source project.

[00:12:56] JM: Can you tell me about the data storage and what that looks like?

[00:13:00] FB: Yeah, absolutely. So, this is something that we iterated several times on until we in our most recent iteration, essentially, ended up settling on a columnar layout. So, columnar databases, as opposed to most other databases, like MySQL or Postgres. MySQL or Postgres store data in rows, so that when you access an entire row, it's essentially co-located on disk, and you can kind of read the entire row. In columnar databases, the unit at which data is organized are the columns and this is super useful for analytical workloads, so that you can quickly scan over everything that is in one column, and do aggregations or filtering over lots and lots of data. And more importantly, you only look at the columns that you're actually interested in, that the query actually asked for.

So, that's essentially what we ended up with, and we needed to build, actually our own database, because we wanted to have a very specific kind of query model. Our query model is very heavily inspired by Prometheus. So, in Prometheus, which is a very popular open source metrics project, you can kind of identify time series by their unique combination of labels. Let's say you have HTTP requests total, and then, let's say in Kubernetes, your namespace, your pod and your container name. And that unique combination identifies the time series. We did essentially exactly the same thing with profiling data.

Now, the problem by having such a model is that you can't just save all of the labels in a map, because then you're losing all of the properties that the columnar database gives you, where you kind of have all of the labels, ideally in its own column. But if you start at a map, again, you're storing all the labels in one place. So, we would be kind of back at square one, similar to a row-based database.

So, we built something that's kind of in between, where you have – we can define some columns in the database to be static. So, things that will always be there. For us, that's, for example, a timestamp. The timestamp is always going to be there. But the labels, the label columns, they will be dynamically created as we see label key for the first time. That way, we can kind of still exploit the columnar nature of a columnar database, the nature, that data is organized by columns, and still kind of – that's how we can exploit all of that and have that data model, that grid model that we desired.

[00:16:04] JM: So, you don't have like a database of choice for storage and retrieval? You're storing it in like a columnar format, and then the user chooses how to query it?

[00:16:14] FB: That's right. So, in a way, it's very, very similar to Prometheus. In the open source project, we bundle the database as part of the statically linked binary. So, it's all in one package, essentially, the like, query engine, is there the database is there, the API is there, the UI is there, everything is in one binary, so that it's extremely easy to try out. And then in Polar Signals, we have a distributed version of this database to kind of scale the product.

[00:16:43] JM: Gotcha. Are you building a database in house? Or what does that database going to look like?

[00:16:49] FB: Yeah, so the database is built in house. Parts of it, or basically, the single node version of it is open source. We call it FrostDB, you can check it out under our GitHub, under Polar Signals. And we didn't build the whole database from scratch, essentially. We did build on top of existing technologies and the like most two most prominent ones that kind of allowed us to build this in months as opposed to years is we build on top of Apache Parquet and Apache Arrow. These are kind of two well established columnar data projects. And Parquet is essentially the like persistent format. It has a lot of encoding so that you can compress things really well, that you can encode things very efficiently. And Apache Arrow is really in order to represent the data at query time in very efficient ways so that they can be processed extremely fast.

[00:17:42] JM: Gotcha. So, I run my program, it's a big distributed program or set of programs, they're running in containers, and eBPF is analyzing the network traffic and creating traces, you're storing those traces in Parquet files, you said? And then in your enterprise version, you have a database that essentially uses Arrow to load them into a memory efficient format for analysis.

[00:18:15] FB: That's right. That's right.

[00:18:17] JM: Gotcha. So, once I have those, I mean, like gathering and storing and acquiring the data is half the battle, displaying it and scrutinizing it and analyzing it is entirely other half of the battle. Are you getting into the kind of Datadog game of building complex visualizations and stuff?

[00:18:41] FB: Yeah, absolutely. Like I said, we already have kind of the most common visualizations for profiling data, and the kind of interesting thing about continuous profiling is also that you're not just getting that like 10-second view of your process, and you only see what happened in those 10 seconds. You can actually look at the profiling data throughout the entire lifetime of your process, or of your entire version of a software, or a region of your data center. Because we have all of this by labels, you can slice and dice it however, it is useful to you and however you organize your infrastructure, right?

That way you can kind of look at, like I said, the CPU time that your process used over its entire lifetime, as opposed to those 10 seconds. That can be much more representative, for example, when you want to reduce cost, right? If your aim of using all of this is to lower the resources that your system needs, using a report like that is extremely powerful, because all of a sudden, we're looking at something that is statistically representative for the lifetime of the process, as opposed to just this one glimpse that we happen to look at.

[00:19:56] JM: Gotcha. What's a typical day problem that I might identify with an eBPF based observability system?

[00:20:07] FB: Great question. So, yeah, I think the things we see most often is like, suddenly misconfigurations, actually, right? You completely over configure at some memory buffer somewhere. We see this with Fluentd or STO. We see often these pieces of software, just not that there's necessarily anything wrong with them, but that they were just configured in a way that is completely over provisioned, for example, for the infrastructure that people have. But we do also see, often that people use hashing algorithms that are not appropriate for the situation. We see garbage collection be extremely spiky or extremely heavy in CPU usage. And that means we need to figure out where allocations are being spent, and where maybe, objects can be reused or something like that. I think probably memory allocations is the one that we see most often, and we're just a little bit of pre-allocating the right memory or reusing an object can make like huge differences.

[00:21:14] JM: We've talked through the engineering stack, and in some detail, maybe you can talk about the – I guess one question I have is the choice programming language. I imagine you're using Go or Rust for the profiling?

[00:21:28] FB: That's right. So, the eBPF programs, they're written in Rust, and then our user space agent is written in Go. Basically, everything else is written in Go.

[00:21:37] JM: Gotcha. So, tell me about some of the engineering challenges you've run into when building Polar Signals?

[00:21:45] FB: Yeah. I mean, if I'm honest, actually, everything about this was hard. The collection of data is difficult, because we're talking about like pretty high-resolution data, as well as kind of being able to make sense of these memory addresses in some way. One thing that we still, there's always one more compiler optimization that exists that makes our life a living hell, where there's this one thing where compiler reorders something, and it's just makes all other assumptions incorrect. And then a stack trace all of a sudden doesn't make any sense anymore. Over time, we've gotten better with this, but it's kind of never ending. So, even at collection time, this is already really difficult. But then storing this data, we built our own database for it in order to handle the complexity of this workload, right?

We're talking about tons and tons of data, being able to store that, but then also to efficiently query this data is a very, very challenging task. And then lastly, visualizing this data is kind of everything deals with a lot of data and the entire stack, right? So visualizing tons and tons of data is also a difficult problem. Yeah really, everything in the stack needs to be hyper aware that we're dealing with a lot of data.

[00:23:12] JM: So, with that high volume of data, I imagine there's a lot of problems that emerge. Can you talk through some of them?

[00:23:19] FB: Yeah, I mean, I think the place where we really spend the most time is the database. And it's things like, there are just some fundamental engineering tradeoffs that you ultimately need to make a call on. Like, how fresh can you make data querying? How quickly do people see the data that is being written to the database, when they issue a query against the database? Essentially, the bigger you make that delay, the more optimizations you can do on the data. But ultimately, with observability tools like this, people want very, very fresh data, because oftentimes, it's something in the here and now that you're troubleshooting, and you want to understand. So, it's sometimes really just finding the right balance of tradeoffs and then also sizing the kind of buffers of how much data do we buffer up in memory until we flush it to disk in Parquet file format, things like that. It's just kind of finding the right tuning for all this, is very challenging.

[00:24:28] JM: Can you share a little bit more about your own infrastructure? So, for a cloud product, okay, so I can imagine the user installs some kind of agent across their infrastructure, the agent hits the C groups to get the observability data from the containers. And then I guess that data gets shuttled to your servers, and then periodically, as you said, you're going to flush that to disk, in a Parquet file. Okay, so you have the servers that are basically the servers that the agents are sending data to. Can you tell me about the choice of runtime for those servers, like what you're using to ingest and read and store all that eBPF data?

[00:25:15] FB: Yeah, absolutely. All of our infrastructure is on Google Cloud and we use GKE for all of our workloads. The nodes themselves are kind of the balanced nodes. So, they're relatively high in CPU and memory. But then, like I said, we kind of buffer up all of this data in memory, and that just by that nature is pretty fast. The whole ingestion pipeline is essentially a lock free data structure. So, we can buffer all of this data up extremely efficiently. And then once we kind of flush it into the Parquet format, we kind of flush all of this data off to object storage from where you can think of it as kind of memory mapping this data against object storage. So, pretend that this file is local, when it is actually in object storage. And essentially, we're building something very similar to Nmap, but against the file in object storage. Essentially, this is kind of a more sophisticated page cache you can think of.

[00:26:26] JM: So, the choice of GKE, any particular commentary on Google Cloud versus Amazon?

[00:26:36] FB: I don't know. I think we just kind of went with it and we've been super happy with it. I think I've seen a lot of people struggle with a lot of Kubernetes providers. But frankly, I just wanted something where I don't have to take any responsibility in like maintaining the Kubernetes cluster, even though actually, before I started this company, for many, many years, I worked very closely in the Kubernetes ecosystem. I was actually tech lead for all things instrumentation within the Kubernetes project, even though I'm very intimately familiar with Kubernetes. And even running Kubernetes, it’s something that I wanted to offload as much as possible and GKE was just kind of the service that has been around for longest so kind of naturally, I think, the most reliable one.

[00:27:23] JM: Gotcha. Yeah. I think I've heard similar things, particularly because, obviously, Kubernetes comes out at Google. So, you take in vast amounts of data into this server endpoint. And periodically, you're hitting the Parquet files with it. And then I imagine, you have kind of a similar question about how to read the Parquet files, right? I mean, don't you have kind of a decision around how to stream in the data into whatever observability system is sitting on top of the Parquet file system?

[00:28:02] FB: Again, essentially, we built like a query execution engine on top of all of this. Basically, all of this is completely custom, so that we can actually deal with the kind of shape of the data that we're talking about, as well as the volume of data that we're talking about. So we built like a query execution engine on top of all of this, that then kind of reads the data from object storage, as well as from the memory kind of buffers, and then kind of computes the query and kind of computes the result to the question that the user asked.

[00:28:40] JM: Gotcha. So, if you think about the different ways to display this observability data to the user, how do you – I mean, it's dense numerical log like data, how do you parse and present that data to the user? What does the user want to see? How can you visualize it to the user in a way that allows them to identify the problems?

[00:29:08] FB: Yeah, I mean, this is definitely a challenging problem, I think, and there's definitely also research that we're still doing if there are more visualizations that could be useful in order to interpret profiling data. But so far, we've really gotten with the classics in this space, which are flame graphs, just ordinary tables that tell us like, this is how much this function is in in total using in your system or kind of this graph visualization. We can see which function calls which other function, and kind of all of the resource resources that are being spent when that happens.

[00:29:51] JM: If I have a stack of different observability tools, and I'm looking for the problems in my infrastructure, when am I turning to a tool like Polar Signals? Is there a particular class of problem that would turn me towards the eBPF source?

[00:30:11] FB: Yeah, absolutely. So, we kind of see the three main use cases for this kind of product. The first one is, you want to optimize cost of your infrastructure. Basically, this is something that I kind of lightly touched on earlier already, once you have the ability to know which pieces of code are using the most CPU in your entire infrastructure, you can actually do something about that, and you could do it with statistical significance. So, cost saving is actually a really big aspect of all of this. But then the second one that we're seeing a lot is from companies that have some sort of competitive advantage, when their applications are faster.

Two very classical examples that I like to give here are high frequency trading companies, right? Every CPU cycle they can shave off of their code is a competitive advantage for their company. I think that one is extremely natural. And then the other one is essentially, all kinds of ecommerce companies. There's a lot of like research out there that shows that when ecommerce websites are fast, they feel instant to the user, then users are more likely to purchase things and convert on those websites. So, those kinds of companies are actually also incredibly interested in ensuring that their applications are as fast as they can possibly be. That's also a really interesting aspect.

Then the third one is kind of like, I guess you can sum it up as incident response, in a way. So, one really interesting report that we can do with Polar Signals is that you can say, what was different about my process at this point in time versus this other point in time that might have been a baseline in CPU versus CPU spike that happened. This can be maybe at the same time when you had a latency spike or something like that. Finally, as software engineers, we can finally tell what code was executing at that one point of time versus that other point in time, and where more CPU time was being spent. The same kind of thing works with memory, the same kind of thing works with any other resources, really.

[00:32:37] JM: Is it useful to look at eBPF profiling data to see if there are potential regressions? If I deploy my code, do I want to check the eBPF profiling data to see if it goes slower now? And then if it does maybe rollback or try to fix it, is it useful for that kind of active acceleration debugging process?

[00:33:06] FB: Yes, yes, absolutely. This is kind of, I would call the mixture of the two use cases, right? It's the I want to improve the performance of my application, and maybe I regressed or maybe it didn't have the performance improvement, didn't have the effect that I thought it was going to have. So, because we have this compare feature, where down to the line number, it'll tell you, where more CPU time was being spent, or less, if the improvement actually worked. You can also see that, right? So, absolutely. This is a really, really powerful workflow with this type of data.

[00:33:42] JM: If I look at a flame graph, and I see a spike in CPU usage, or a spike in memory usage, how exactly am I able to trace that back to a specific line of code?

[00:33:59] FB: Ideally, in order to see it quickly, you would have profiling data from a baseline as well as the spike. When I when I say, baseline versus spike, I'm saying, you're essentially looking at the cumulative. You're looking at the total CPU usage, let's say. When you have that baseline of I don't know, 10% of CPU being used, and then all of a sudden, you have 50% of CPU being used by one process. Then you can, in Polar Signals, you can select, you can kind of pull up the comparison mode, and you can select profiling data from that baseline, and you can say, like profiling data from that spike, and then it will color code where more CPU time is being spent, and the more it's being spent, the darker red color it will get in the flame graph, essentially. If it got better, it'll be colored in green. So, our flame graphs actually by default, they're not I colored in any way. Actually, the original flame graphs were only colored in that way, because that made it look like a flame. There was no kind of particular pattern to it, other than making it look that way. We try to use the coloring in the flame graphs to actually kind of communicate meaning of those flames.

[00:35:23] JM: Let's say I have an application that's been running for a very long time, maybe there's lots of inefficiencies in it, and I install a profiling tool like Polar Signals. I'm going to get a very detailed gratuitous readout of the things that are wrong with my infrastructure. Where do I start? Is there an easy visual way to guide me towards the most obvious problems?

[00:35:51] FB: Yeah. So, I think the most natural thing is to actually just ingest all of the data and look at a report where you are kind of merging all data ever collected, because that will give you a representation, especially if you only just started doing this, it'll kind of give you a representation of this is where all my CPU time in my entire infrastructure is being spent. This is merged across processes, shared libraries, and all of these. They're all kind of compressed into one report. If you have multiple times the same process, it'll be the same stack traces merged into one report, essentially. So, you'll actually see simply by the size of the frames, where the most CPU time is being spent in your entire infrastructure.

So, I always like to start with that. But if you have a specific application that you know you would like to improve, then I would actually recommend to filter the data down by the labels. Maybe if it's in Kubernetes, maybe your application is in a particular namespace in your Kubernetes cluster. So that's, for example, a popular label that people like to label their data with. But all of this is configurable. You can make it however you organize your infrastructure. But that's a very common one.

So, you could filter down by like, my namespace equals my application, right? And then you could do this merge, and then you see all of the CPU time of your entire application across all servers that you have deployed on. And then again, you can kind of go by the sizes of the stack frames. That kind of guide you in where the most CPU time and cumulative are being spent. And then when you kind of get to the point where, you know, maybe you're doing something that is unnecessary, or where you end up at a leaf and a flame graph, those are the places where you can have the biggest impact in actually doing an improvement. Basically, all performance improvements always boil down to is either not doing something that you're doing right now, or doing it more efficiently. And those are essentially the two tools that it will always boil down to.

[00:38:15] JM: I'd like to get a vision for what else you're building out. How do you anticipate the platform changing over the next five years?

[00:38:23] FB: Yeah, so there's a bunch of really interesting stuff that we can do with having this kind of data. So, one of the things that we're really excited about is something called profile guided optimizations. This is kind of a concept that has been around since the ‘70s. I think when I did my research, where essentially, because we have real life data about how this program is being executed, we can actually give this information to the compiler, and the compiler can make opinionated optimizations based on that information. It can make optimizations that are not generally good, but good for this particular case, because we know how this code is going to be executed.

Actually, some like just in time compilers, like dotnet runtime, they already do this by themselves. But applications or compilers like LLVM, or like GCC, they've had this kind of capability for a really long time. But it's just always been this UX problem where that nobody has profiling data from their production systems. But guess what, that's kind of exactly what we're building here. So, this is one of those features that we're extremely excited about, kind of bringing to the masses, because it's really magic to plug this into your like build infrastructure and all of the sudden your infrastructure is like five to 10% faster just by doing that.

It's things like that, where we can go much further with what we're able to achieve with this kind of data. But ultimately, I think what we're going to start to do is integrate much, much more heavily into the existing observability signals. One thing that we're really excited about also is kind of connecting this data with distributed tracing data. One thing that we're actually already doing and it's already possible to do this is to every stack trace, you can add labels. One of the ways that we're utilizing this is we're attaching the distributed tracing ID, to those stack traces. All of the sudden, there's a new dimension, essentially, that we're creating where we're seeing these stack traces, while they're similar, this one was using CPU time for this distributed trace, and this one was CPU time for this other distributed trace. So, that's extremely powerful, because all of a sudden, we can say this request use this much CPU time, as opposed to my entire process. So, I think there's a lot more capabilities that we can work out here. I think, ultimately, what it'll boil down to, though, with four Polar Signals, if you're asking me for the five-year vision, we’ll probably eventually move into those vertical, it's because ultimately, that's how we're going to be able to do the best integrations.

[00:41:24] JM: Is company defensibility at all an issue? Because there are some just gigantic observability companies already in the space. I wonder how you prevent just becoming a feature of one of these other giant observability companies?

[00:41:41] FB: Yeah, I think in a way, starting with continuous profiling was kind of intentional in that way, right? We use something that was kind of this new niche that wasn't very widely being done yet. And so, we use it as kind of the thing that gets us into this ecosystem. Like I said, we have a very strong Prometheus background. I myself, a Prometheus maintainer. We have a long, long history with the Kubernetes and monitoring space. So, I think we're capable now that we're in this space, and we're kind of starting to establish ourselves as a player in continuous profiling, ultimately, we're going to need to make that step. So, that we're not just a feature, but that we're actually kind of competing on the same level with these other companies. But yeah, it's, it's difficult, I'm not going to lie.

[00:42:31] JM: Do companies these days, choose, like a set of observability products? Just thinking from the perspective of the buyer. Is a company willing to buy Datadog plus Polar Signals to get more coverage?

[00:42:47] FB: So, what we're particularly seeing, because of how closely we engineered our product to kind of harmonize with Prometheus setups, we're actually seeing the overwhelming majority of users want to mix it with their existing Prometheus setup, or maybe it's Thanos, like one of the distributed Prometheus setups. But essentially anything that is very Prometheus heavy, those are the kinds of customers that are actually – they've gone through that story of working through building a distributed Prometheus in house, they don't want to do it again. And now, they're actually quite happy to pay a provider for this.

So, in a way, by engineering it to work really well with Prometheus, we've actually done part of that work, where even though we're not offering that product ourselves, people are still able to have that entire suite of observability tools, even though we're not necessarily the first, the provider of those.

[00:43:52] JM: As we draw to a close, maybe you could give any more perspective on the observability landscape, and any outstanding problems in the world of observability that you think are unaddressed?

[00:44:06] FB: Yeah, that's a great question. I think, and this is sort of Polar Signals' mission. We actually wrote this down. In a way we're not here necessarily to solve continuous profiling. At least, that's not the only thing we want to do. But part of our mission is essentially, we feel that the observability world has kind of lost itself a little bit, where we think it has kind of turned into this game of just collect all of the data you can and eventually you'll make sense of it. While that's not wrong, I think most of the tools, maybe it's because of how billing works. I'm not 100% sure. But I can see how that could be the reason. Basically, all the tools focus on just write more data into these products, and you'll figure it out later right.

But I think this figure it out later part, I think this is where observable We'll see tools fall short today. We have these giant piles of data. But most people are not capable, like most organizations are not capable to make to make sense of this data. I made this mistake essentially myself as well. In one of my previous jobs, I maintained a very, very popular Prometheus stack that is kind of used by tens of thousands of companies now, to run Prometheus on top of Kubernetes and we did the same thing. We put all of the exporters in there, and eventually it was just like, people were saying, “Hey, this uses so much memory and I don't even understand any of this data that is shipped with this.” I think this is kind of on a meta level, the problem that the observability landscape has to solve, even if it's not about more data, maybe more data makes it actually even a bigger problem.

But on the other hand, I do also think there are still more data sources that we're not utilizing today. And eBPF is kind of only like, starting to show us this, where there is much more data that is really, really useful to shine different lights on aspects of our running applications. But the very important thing about introducing new things like that, and this is what we wanted to show with continuous profiling, when we do that we need to kind of teach people and give people the right set of tools in order to truly make sense of this data, so that they can also understand it or at the very least, so that they can learn to understand it. So yeah, I hope that answers the question.

[00:46:39] JM: Cool. Well, Frederic, thanks so much for coming on the show.

[00:46:43] FB: Thanks for having me.

[END]