EPISODE 463

[INTRODUCTION]

[0:00:00.6] JM: A popular software application serves billions of user requests. These requests could be for many different things, and the requests need to be routed to the correct destination, load balance to cross different instances of the service and queued for processing. Processing a request might require generating a detailed response to the user or making a write to a database or the creation of a file on a file system or maybe all three of these things.

As a software product grows in popularity, it will need to scale each of these different parts of infrastructure at different rates. You may not need to grow your database cluster at the same pace that you grow the number of load balancers at the front of your infrastructure. Your users might start making 70% of the requests to one specific part of your application and you might need to scale up the services that power only that portion of the infrastructure.

Today's episode is a case study of a high volume application, a monitoring platform called Raygun. Raygun's software runs on client applications and delivers monitoring data and crash reports back to Raygun's servers. If I have a podcast player application on my iPhone that runs the Raygun software and that application crashes, Raygun takes a snapshot of the system state and reports that information along with the exception so that the developer of that podcast player application can see the full picture of what was going on in the user's device along with the exception that triggered the application crash.

Throughout the day, applications all around the world are crashing and sending requests to Raygun's servers. Even when crashes are not occurring, Raygun is receiving monitoring and health data from those applications. Raygun's infrastructure routes those different types of requests to different services, queues them up and writes the data to multiple storage layers, elastic search, a relational SQL database, and a custom file server built on top of S3.

John-Daniel Trask is the CEO of Raygun and he joins the show to describe the end-to-end architecture of Raygun's request processing and storage system. We also explore the specific refactoring changes that were made to save costs at the worker layer of the architecture.

This is useful memory management strategy for anyone working in a garbage collected language. If you'd like to see diagrams that explain the architecture and other technical decisions, the show notes have a video that explains what we talk about in this show.
Full disclosure; Raygun is a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

[0:02:50.0] JM: Are you a Java developer, a full-stack engineer, a product manager or a data analyst? If so, maybe you'd a good fit at TransferWise. TransferWise makes it cheaper and easier to send money to other countries. It's a simple mission, but since it's about saving people their hard earned money, it's important.

TransferWise is looking for engineers to join their team. Check out transferwise.com/jobs to see their openings. We've reported on TransferWise in past episodes and I love the company because they make international payments more efficient.

Last year, TransferWise's VP of engineering, Harsh Sinha, came on Software Engineering Daily to discuss how TransferWise works, and it was a fascinating discussion. Every month, customers send about $1 billion in 45 currencies to 64 countries on TransferWise, and along the way there are many engineering challenges. There's plenty of opportunity for engineers to make their mark.

TransferWise is built by self-sufficient autonomous teams and each team picks the problems that they want to solve. There's no micromanagement, no one telling you what to do. You can find an autonomous, challenging, rewarding job by going to transferwise.com/jobs.

TransferWise has several open roles in engineering and has offices in London, New York, Tampa, Tallin, Cherkasy, Budapest and Singapore among other places. Find out more at transferwise.com/jobs.

Thanks to TransferWise for being a new sponsor of Software Engineering Daily, and you can check it out by going to transferwise.com/jobs.

[INTERVIEW]

[0:04:41.1] JM: John-Daniel Trask is the CEO of Raygun. John-Daniel, welcome to Software Engineering Daily.

[0:04:46.6] JDT: Thank you very much for having me, Jeff.

[0:04:48.2] JM: Yeah, it's great to have you. You work on a crash monitoring system that handles a high volume of exceptions. The way it works is whenever a user's application crashes and they're running your crash monitoring software, the exception that caused the crash needs to get recorded and indexed on your servers so that the developers of whatever application crash can understand what is causing the crashes.

If I'm a developer and I'm building an app that plays podcast episodes and one of the users is listening to a podcast and then the app suddenly crashes, I would like to know about that as the developer of the podcast app so that I can fix the bug. In order to know about the crash, I need it to be detected and reported, and that's what your software, Raygun, does. Since a lot of crashes are going on across the world at any given time, your infrastructure is constantly getting hit with requests.

Can you describe the requirements for what you have to do with each of those crash reporting requests?

[0:05:56.8] JDT: Yeah, absolutely. I'll just fill in a couple of extra bits there too.

[0:06:01.3] JM: Please. Yeah.

[0:06:01.9] JDT: Raygun does work with about 25 different programming languages and platforms, so mobile devices, through the desktop backend services, frontend JavaScript, and we also have what's called real use of monitoring, and that's tracking the performance story as well as well as high level analytics. We're collecting every single page load, the timing on every image load, which views inside a mobile application people are looking at, how long it takes to

be really — How long networks calls are taking for mobile devices? How long HX calls are taking. All of the stuff's going in there as well as the crash reporting. We actually have the two pieces in there, which dials up the data volume by a couple of extra orders of magnitude as well.

There is a lot of data flowing through there, and we've done a lot of work to try and keep the performance pretty workable. I won't lie, we typically run through a fairly consistent development cycle of six months where we feel like the system is doing really well and it's nice and fast and performant and then six months of, "Oh, goodness! It's all burning down. Let's rebuild the aircraft hallway through the flight," that sort of thing.

Yeah, there are a lot of interesting challenges. We're processing probably a couple of billion data points a day now through the platform. For the most part, I know we're going to dig in to this a little bit more. For the most part, a lot of the system is built using Microsoft technologies using the .NET framework, .NET Core. There's a little bit of Go in the mix there, and then there's a reasonable amount of Java as well with relation to some of the data storage technologies that we're using.

[0:07:29.6] JM: You're on AWS?

[0:07:30.7] JDT: We are on AWS at the moment. Yes.

[0:07:34.3] JM: Okay. You've got all these requests that are coming in and they're making their way through your service infrastructure. First, they're getting load balanced, then they're making their way through your service infrastructure and then they're getting put into these various data stores so that the users can understand the performance and the crashes and what's going on in their applications.

Let's start with that load balancing layer. As you described, you've got a whole lot of different types of requests. You've got a high volume of those requests. What are you doing at the load balancing layer?

[0:08:08.5] JDT: We're definitely making use of some of the load balancing technologies that AWS provides that are fairly common across most cloud providers. Then behind that, we have

effectively an auto-scale group of EC2 instances that are running our API layer code. They basically receive the data and then they verify a couple of quick things and making sure, for example, that the client API key is valid, kicking a couple of rate limiting things. If somebody's paying for a certain level and they're way, way over, we manage that.

Then otherwise they put it on to an internal queuing system to be lifted and processed. That API layer used to be in Node.JS. They're not particularly massive boxes. They're relatively small. We ported to that Microsoft's .NET Core framework about a year ago, and we managed to increase the performance of that API layer by more than a thousand percent on there. We've actually got 2,000% improvement and throughput, which alarmed a few people who thought we must just be using Node wrong, which I disagree with them on that view, but .NET Core has been a fantastic win for us on that API layer.

That meant that we could process more per API node, and therefore decrease the number. We didn't decrease it right down, but we brought it down enough that we could still have enough sort of service in the one pole if you will to auto-expand quickly if we need it to. That's the first layer.

[0:09:34.6] JM: When you say API layer, does that mean that these requests hit a load balancer first and then the load balancer spreads them through the API layer and the API layer routes the requests to different queues, or is there a queue in front of the API layer?

[0:09:53.2] JDT: No. It's the standard ELP related stuff going to an auto scale group, then what we do is we maintain a queue per application that we're tracking, and this allows us to solve for the noisy neighbor problems. You wouldn't want to be using a service where just because somebody else was using it heavily it impacted your performance, and so by doing it on a per app level, one; it gives finer grain control of rate limiting if needed, but it also means that if you're doing a striped work or approach where you're fetching X-number of items off a Y-number of queues at a time, you don't have to worry so much that one queue might be backing up a little bit. It's not going to impact your processing time performance.

[0:10:35.0] JM: What are you using for queuing?

[0:10:36.3] JDT: We're using Rabbit at the moment. We're not using like the AWS built-in queuing services. What we've typically found with a lot of the services that the cloud providers have is that they're wonderful if you are either not money conscious in the lease or not doing a heck of a lot of operations with them. Frequently we end up where we'll do something like run our own instances running things like Rabbit rather than be paying per thousand request to a queuing service, that sort of thing.

[0:11:07.0] JM: Any reason you went with Rabbit over Kafka? I imagine, the simplicity, I guess.

[0:11:13.1] JDT: It's the simplicity. There's also just the way that we then pluck things off the various queues there. We've had some internal conversations about moving some things to Kafka overtime, but it's just not been a sort of a burning need at this stage, because we don't do too much in the way of advanced routing the way that the worker processes are architected. They kind of pick off a piece of work and then I think there's only really a couple of extra layers around the persistence side, so it'd be a pretty short workflow even with that in there.

[0:11:44.7] JM: Okay. You got this high volume of events that are coming in. They're getting load balanced across API servers. The API servers are routing those requests to the appropriate queues based on maybe what the request is. Is this monitoring around image loading? Is this a crash report based off of those different event types? You will have routing to different API endpoints, and then which means different queues in different workers. Do I have the right —

[0:12:19.3] JDT: That's correct. Yeah. You can kind of imagine like in a software development process the two products came together separately. They do have their own processing models. They're very closely aligned. Yeah, basically, different API endpoint, different queues, different workers.

The data storage, however, it often gets unified, because one of the driving things that we wanted to improve in the space of monitoring and ops is that there's a lot of silos of data and we hate having silos, because we think that the real insight lies in being able to overlay this information together, bring it together in one place. The data storage is kind of fairly unified.

[0:12:57.6] JM: Got it. The workers that are pulling these events off of the queues, what are they doing exactly? Classify kind of what the workers are doing.

[0:13:08.3] JDT: With the pressure pointing side, what they'll do is they'll popup the records, which is typically a JSON Blob. They'll inspect that to see what the sender was sending from. Basically, the provider was a JavaScript, was a Java C# and whatnot. The reason that's important is because we then will pick a grouping or a classifying strategy based on what it came from.

The reason for that is that we want a group up errors. If you had 10,000 errors, but only four bugs, you'd actually want to see the four bugs with the counts next to how often that occurred. You wouldn't want to get 10,000 items to look at in the list. It wouldn't be very manageable.

We do this grouping, and those groups themselves, they have a lot of logic in them. For example, our JavaScript one is smart enough to identify that even if an error looks wildly different between, say, Firefox and Chrome, they're actually the same fundamental bug. They'll align things like stack trace lines and messages and be able to bundle them together to make sure that you've got a very actionable workload.

Then what they're going to do is it's going to update a couple of various counters in places and then lock to persist various bits of data to the different data stores that are needed. The major data store that we have is an internally built storage engine which overlays on S3, but does a whole bunch of smarts to make fetching of data more efficient, the rollups of data. It's kind of a little bit like the Hadoop file system in a way where it's bundling things up into content blocks to make it much more efficient both on cost basis and a computer performance basis to load data faster for our customers.

[0:14:43.8] JM: What exactly is getting stored in that Hadoop-like file system?

[0:14:48.1] JDT: The entire raw data. For everything that Raygun receives, we can give our customers 100% of the raw data they had in there, if they wanted to see it and go that level. A lot of players in this space will give you averages and charts and not really let you access the

raw data. We store just hundreds of terabytes with the data that our customers, specially larger customers frequently like to pull out and feed into other systems.

Lastly, in that working process, it will also trigger a couple of internal notifications and that will mean that depending on what the customers have configured, do they want alerts to Slack, do they want to get an email. What are the rules around that? That's handled by a notifications worker. That's the one time it pops up and back on to a sort of a pops up type system internally.

[0:15:30.9] JM: What would be an example of one of those raw data files and what would lead to the creation of that raw data file?

[0:15:38.1] JDT: The raw data file would be the JSON Blob that the Raygun service received. The entirety of the error context, everything we picked up at the time of the bug, occurring environment information, server information, depending on where it's from, iOS. Let's say it might be the device information. As much as we can collect at the time of the error, to try and help our customers more rapidly resolve an issue.

The plus side of that is it's because we make all of that information available, when we do have some very large customers who have very serious problem security requirements, and we can say, "Look. You can see literally everything we've collected." There's no hiding it behind things. Have a look at that.

Our SDKs include filtering. So if there is some piece of data that they don't want to provide, they can filter that out. We of course have some defaults on there around things like credit card numbers and social security and whatnot, but they can always add more.

[0:16:30.7] JM: When you're talking about the events that are doing monitoring, like when you're just monitoring the application, but a crash has not occurred, does the storage model differ?

[0:16:43.1] JDT: The storage model doesn't change a whole lot. It's a fairly consistent story in there. We don't have to do things quite so much around the grouping, like we might group

based on URLs, which is pretty straight forward to do, or a page and activity names inside of a mobile app, but it's fairly consistent.

[SPONSOR MESSAGE]

[0:17:09.5] JM: Dice helps you accelerate your tech career. Whether you're actively looking for a job or you need insights to grow in your current role, Dice has the resources that you need. Dice's mobile app is the fastest and easiest way to get ahead. Search thousands of tech jobs, from software engineering, to UI, to UX, to product management.

Discover your worth with Dice's salary predictor based on your unique skillset. Uncover new opportunities with Dice's new career-pathing tool, which can you give insights about the best types of roles to transition to and the skills that you'll need to get there.

Manage your tech career and download the Dice Careers App on android or iOS today. You can check out dice.com/sedaily and support Software Engineering Daily. That way you can find out more about Dice and their products and their services by going to dice.com/sedaily.

Thanks to Dice for being a continued sponsor, and let's get back to this episode.

[INTERVIEW CONTINUED]

[0:18:25.9] JM: Let's talk through the ingestion and the processing of a single crash event. A crash event occurs, you've got this JSON Blob that contains a ton of information about the application at the time of the crash, and it gets queued up and then it's getting processed by the worker. What's the unit of deployment for these workers? Is it a virtual machine? Is it a container?

[0:18:57.5] JDT: No. It's not containerized yet. We are still a little bit old school on that front, where we use Octopus Deploy and we deploy updated versions of these services to a cluster of different machines.

What we've found is that while there is some spikiness to the volume, because we're a very much a global business, there's only a short period of the day really where it's particularly quiet, because we've got a lot of customers in Europe and a lot of customers in North America and a lot of customers also in Australia.

For the most part, it spikes a little bit, but it's more — One customer spiking doesn't typically cause us too much stress. It's more that the overall daily sort of ebbs and flows. We have affixed it in there with the ability to rapidly add additional instances with that software on there and then that will pick up different sections of queue to manage that scaling there.

[0:19:48.7] JM: These workers that are processing the JSON Blob, what kind of worker are they doing in order to process that before they put it into the storage layer?

[0:19:59.6] JDT: That's where they're doing things like reading it and analyzing for the grouping, triggering notifications to customers, updating various counters for people, because obviously we don't always count everything up in real time. We do it for the most part, but they'll update various things. They update the customer usage information so that we understand how much customers are utilizing the service.

They don't have to do a heck of a lot, to be honest, because they're passed off to the storage sub-systems for people to then come in to the application and actually analyze what's going on.

[0:20:30.4] JM: Right. Okay. I got it. They're just kind of updating things that might be viewable in a dashboard, or just stats, and they're rapidly passing it off to the storage layer. These workers are written in .NET, and I think — Okay, you said the API layer you rewrote from node to .NET. Were the workers always in .NET?

[0:20:56.1] JDT: The workers have always been in .NET, full framework, not .NET Core yet. We probably will be moving them to .NET Core at some stage, but it's not a hugely high priority right now. We also do have a little bit of Go in there around how we manage the symbolification of crashes from iOS, and that's just a small service that's orchestrating the symbolification process.

[0:21:17.9] JM: That's symbolification?

[0:21:19.2] JDT: In iOS or in an unmanaged language, like C or C++, when you get a crash, you get a memory dump, and it's not readable to a human. With iOS and nodes and other languages, that's where you get the symbol file, so maybe a PDB file or a DSIM file for iOS or X-code generated outputs.

Basically, if Raygun has copies of those, which we have integrations to automatically pull them up for people, we will automatically go, "Okay. We've got this memory dump and we've got this symbol thing. Let's do some magic to turn it into a human readable stack trace so that the GIF can read it and go, "Oh! It was line 49 of this class and this function," rather it's a raw memory dump of hex values that mean nothing to me. That's what symbolification is about.

[0:22:05.0] JM: I see. Just talking about the .NET, the .NET workers that are getting these JSON Blobs. They're making some updates to counters and things and then they're storing the JSON Blob in your S3. I think you said S3, right?

[0:22:24.1] JDT: It's backed off of S3, but yeah, there's a layer in front of that that does a lot of smarts for how we roll up. Like I said, it's very similar model to the Hadoop file system. Let's say we've got 100,000 errors coming through. We might roll that up into a single block that we can then index and pull data out very, very efficiently, but we can pass that block around so we don't have to make 100,000 requests to S3. We don't have to worry that the latency per request would kill performance, because we've done a lot of analysis to find what the ideal size is for our infrastructure to make sure we can get a lot of data very quickly, cheaply and efficiently.

[0:23:03.2] JM: Wow! Maybe we could talk about the file system a little bit later, because that sounds like a pretty interesting conversation for implementing that. It took a lot of work.

[0:23:10.0] JDT: It did.

[0:23:12.9] JM: Yeah. We did a show recently about Keybase, which is kind of this new identity platform, but they have a file system — They wrote their own file system — Talking about writing your own file system is always a challenge, even if you're building it on top of S3, I'm sure. As

far as the workers, they are processing these large JSON Blobs. I saw a talk that you gave about the garbage collection and the memory management that you have to do in your .NET worker layer. Is the main reason for that garbage collection the work that you did around garbage collection? Is that because of those large JSON objects?

[0:23:56.5] JDT: Yeah. You're talking about the video where I talked about the source map processing worker. Is that correct?

[0:24:01.2] JM: Yeah. Yes.

[0:24:03.4] JDT: In that one — And we'll sidetrack a little bit into that story, because it was quite fascinating. Source maps are very similar to, effectively, symbolification process, where when you minify your JavaScript and you can combine files, you're used to seeing those errors that will say, "Something happened on line one at position 34,000." You're like, "Well, I don't typically write my programs as a single line, 34,000 characters long. That's clearly a post-minification process on my JavaScript."

A source map file can be generated by most minifying engines, and that allows us to also unwind it back to the original version of the code that you had. It would say, "Actually, it's in line 490, column 10. Here's the code." It makes it easier to debug. Without them, it's kind of a pain in the butt.

This was a few years ago now. I remember talking with our CTO, and he was like, "Oh, you have just put in there a third machine for our source map processing." I'm a nerdy guy and I was like, "Really? Three machines for sourcing? How many are we processing?" He's like, "Oh, we're doing millions and millions of them a day." I was like, "Oh, man! I don't want to keep adding machines. I want to make this faster."

At the time we were using Mozilla's Source Map processing engine, which is open-source. I looked at this thing and I was like, "Okay. I can probably port this over to .NET," and so I did. Then this was like a weekend project for me, and then I ran the profile. I was like crazy to get this down. Ultimately, we ended up where we moved the source map processing to a .NET

service where a single machine out of the three could process our entire day's volume in like one hour or something like that.

That talk that's on the Channel 9 website, basically, I was talking about the ways that I approach the performance handling for that, and there were some low level stuff. But whenever you're building on a language that has a garbage collector, anything like Java or .NET, the two most common one to say, but there are others. You really need to be aware that the benefit of having garbage collection is that you don't have to worry about the allocations and de-allocations and things like that, but there is a cost. Everything comes with a cost.

One of the things that's been fascinating is that because allocations are so cheap and easy to do without thinking is a lot of developers are allocating memory like crazy not realizing that there's a huge performance penalty in there. Of course, you're thinking, "Hang on a minute. Isn't memory fast? Isn't that how our database is getting fast, because we just put the data and memory and that makes it quick?" That's where you've got to remember that accessing the L1 cache on a CPU is about half a nanosecond, L2 cache, about two nanoseconds. Then, say, your main system RAM is more like a hundred nanoseconds.

If you're actually writing and creating data all the time and memory and going backwards and forwards, one; that's actually quite a bit slower. Secondly, of course, you've now got this additional process inline which is your garbage collector that's having to mark and sweep the memory or however it wants to do the handling of clearing that RAM from time to time, but it's kind of into that where us as developers can be very, very efficient at writing code in a garbage collected world at the downside that we often don't think too hard about it on how much memory we're actually allocating.

When I was working on that particular little service, I was king of going, "Well, I want to make sure I don't allocate much memory." I suspect you're referring to the fact that the JSON data — So the way that a source map is structured is in JSON, and there were many things that we looked at, which was like, "Hey, what if we didn't actually de-serialize the JSON data into a first class object? We actually only need to read it once and we can just use — We already have it written as a string. Why don't we just read it like it's a character array and just mark the points

we need and pull out the data as we need it." Things like that, which dramatically accelerated the rate of handling those source maps.

Then, also, if you watched that video, I think go through about five or six different things, like for example, "Hey, if we only need to parse the source map 50% of the way, the plus side is we don't even need to read the next 50% of it." The best way to make code fast is to do nothing at all. There're a lot of shortcuts. There were things like not serializing the object. There was all sorts of stuff that went in there.

I remember, actually, bugging my friend Nick on that weekend [inaudible 0:28:37.6], "Hey! I've just shaved off another 25% of the processing time," and all of these. The only thing that makes me sad reflecting on that time was that I don't know if it was out at that stage, but there's an amazing tool in the .NET community now called Benchark.NET, and you can do in a very similar structure to how you would write your unit tests. You could just augment things with Benchmark attributes and it would tell you, for example, how much memory is being allocated. What stage of the garbage collected? It does all the statistical handling for median time, to main time, standard deviation, what's the baseline of the machine. All sorts of cools stuff. I didn't have that at the time. I was just using a full desktop profile, which was still good enough, but I kind of need to go back and have a fresh set of eyes, look over it and probably put some of those two server to benchmark.NET.

Sorry. That was quite a segue, but yeah. That's the sort of thing we look at.

[0:29:30.8] JM: That's fine. It's a good example. To be clear, you've got this large crash report essentially and you're trying to figure out how much of this crash report do we actually need to pull into the worker or how much — First of all, how much do we need to de-serialize?

[0:29:51.9] JDT: No. Just to be clear. What happens is we always do everything with the crash report, but in this case you're saying, "And if it's a JavaScript, and if there's a source map available, then we need to do basically a post process on the original stack trace to make it more human readable." It's more like an F-state where the few variables on there get checked. If that's the case, then go and do this extra piece of work.

Similarly for iOS, for example, if somebody hasn't provided us a decent file where we can't symbolify it, right? It just goes through as is, but if we have it, we want to make sure that we enrich the stack trace to make it human readable with the symbolification process.

There's just a couple of little edge cases in there. It was quite involved building out Raygun. It's a fairly mature platform these days. There're a lot of things it can do.

[0:30:40.6] JM: Yeah. I worked at a place one time, this options trading company. Like any trading company has, a really high volume of stuff that comes through it. There was a lot of work that would go into tweaking the garbage collector, and this was my first job out of school, so I had no idea, like, "How do you do the garbage collection tweaking?"

I would see these people, these engineers and they would be pulling up these Java tools for like assessing how much memory is in the — I guess, these different, like the Eden space or these different garbage collection tiers in Java. But that's not what you're doing here. Before stuff gets pulled into the place where it's going to be managed by the garbage collector, you're saying, "Do we even need to have this data be pulled into memory? Do we even need to create this object in memory?" Is that right?

[0:31:41.2] JDT: That's absolutely correct. To your point there as well, Java seems to have a few more different garbage collection options out there. In the .NET land, there's primarily like a client garbage collector and a server garbage collector, and then there's obviously new version of the garbage collector that come with different versions of the framework.

It's not quite as rich as the Java community. You can also control in a, sometimes, nondeterministic way how some of the things, like giving hints the garbage collector encode. On the whole, and this has actually been a huge driver for a lot of the performance improvements that Microsoft made in the .NET Core framework, is that if you just don't allocate the memory, then you don't give work to the garbage collector. If we can reduce those allocations as much as possible, the code actually goes faster, because memory allocations; one, are expensive; and two, you have that somewhat hidden cost that the garbage collector now has more work to do as well.

By focusing on the number of bytes that get allocated by a piece of code and bringing that down, all the better. We did a lot of that sort of work early on and we're still doing that. Literally, one of our team members, Jason, is working on some of the way that we do grouping code for the real use of monitoring data around the pages, and like now it's fairly common where he is posting poll requests that actually includes the Benchmark.NET output of before and after to say, "Here's the time it took. Here's the bytes that were being allocated and here's where it is now," to demonstrate functions [inaudible 0:33:11.0].

[0:33:12.4] JM: How much money can you save on these kinds of performance improvements? What sort of savings does it translate to?

[0:33:19.6] JDT: We've had some fairly significant wins in the past around these improvements. At the time, years ago, when we did the source map one for example, I think that was a relatively quick saving of about a thousand dollars a month. We were really, really small at the time. Further to that, because it was only using a fraction of one of the servers, we could also put other things on there. There were additional savings there.

We find that, today, most of our costs are actually driven by the data storage side of things, not the compute intensive side. That's where we're starting to look at things, like how do we — That's partly why we started looking at like our storage subsystem a couple of years ago, and that was a huge one for us. That actually saved us, I think — It's probably now accumulatively saved us several hundred thousand dollars since that went into operation, because it cost — If we bring something down by, say, $10,000 a month, a year later that's $120,00 that's been saved. It's not like we can stop data storage next month. It just improves things.

We're always looking at that. I think that's one of the strengths that I can help bring to our business as being a CEO with a tech background, is I can sort of have these conversations with our CTOs, my business partner, and we can sort of weigh up, "Hey, is this a situation where we just want to throw service in it, or do we actually have —" I consider performance things a form of technical data as well, even if the code is recently fine, "Hey, maybe we could just make this more efficient and not need to go and get a bunch more service."

Usually, it's dependent on what else is going on in the business at the time. Maybe we're working on — When we're building out the real use of monitoring system. Obviously, a lot of engineers were busy working on that, and so we didn't have the bandwidth so we just had to say, "Look. We're going to throw more service at it for now, make a note. We'll go back and turn this stuff up," but it varies.

I do think that's a huge thing, which is evaluating when to look at performance improvements. I see a lot of engineers, one; they'll either think that the performance issue lies somewhere where it doesn't, or it's not actually a performance improvement that does make the business save money or it's not something that impacts the customer in a positive way. There are always important things to consider as well.

[SPONSOR MESSAGE]

[0:35:45.9] JM: Do you have a product that is sold to software engineers? Are you looking to hire software engineers? Become a sponsor of Software Engineering Daily and support the show while getting your company into the ears of 24,000 developers around the world. Developers listen to Software Engineering Daily to find out about the latest strategies and tools for building software. Send me an email to find out more, jeff@softwareengineeringdaily.com.

The sponsors of Software Engineering Daily make this show possible, and I have enjoyed advertising for some of the brands that I personally love using in my software projects. If you're curious about becoming a sponsor, send me an email, or email your marketing director and tell them that they should send me an email, jeff@softwareengineeringdaily.com.

Thanks as always for listening and supporting the show, and let's get on with the show.

[INTERVIEW CONTINUED]

[0:36:47.9] JM: Generally, I think from a strategic standpoint, it makes sense to look at saving money as a core tenant, because you're not exactly in a business where you have complete dominance over the field. You're not like Google or Facebook where you can just say, "Okay.

We don't really need to worry about competitors. We'll just buy all the stuff that we need, because we sort of have a monopoly. We're just trying to cement that monopoly."

There are some competitors in the field, and so keeping the price in a competitive tier is probably pretty important to you.

[0:37:25.8] JDT: Yeah. I look at it more from the — You're right. We're not a Google or a Microsoft. We're working our way there. One of the things that's a bit different about Raygun to a lot of players in this space is that we're not a hugely VC-backed company. We're not just burning cash for sake of it. We actually have a very strong business unlike most of our competitors that are bleeding cash trying to soak up customer count.

Raygun actually has a very stable and still very fast growing business, and the way, again, that I look at it is any dollar I'm not giving Jeff Bezos is another dollar I can spend on growing my business. That's sometimes — Our logic is gross margins in a business. You look at what's that cost to serve a customer. How much do we spend on support, infrastructure, critical processing, all that sort of stuff. That's going to be a fixed amount that we have to pay every single month to support our customers.

The smaller I can make that, the more I can spend on marketing, the more I can invest in getting more engineers to develop additional features that add value to our customers, all of that. I'm more driven by how can we grow faster and help our customers more because we're efficient. As supposed to just saying, "Hey, can I make it the cheapest product in the space?"

There are cheaper products in the space, but the first thing that happens when you dig under the covers is, "Oh, wait. You're not actually storing all the data." "Oh, you're heavily sampling." I couldn't tell you how many hundreds of customers have moved to our platform there were like, "Well, it was kind of like trying to explore a cave with a candle versus having a floodlighting." If you don't have all of that data, it's quite difficult.

It's kind of like this tool — I'm not picking on specific competitors, but in the wider monitoring space where all they talk about is averages. Because we have all the data, we'll often give people the P90s, P99, the distribution of data of what's going on. The great analogy that I've

always loved about why, if you are looking at averages, you are looking at lies, is let's say, Jeff, you and I, we go and fill a stadium with homeless people and then we invite Bill Gates in. On average, everybody in there is a multimillionaire, right? Averages are lies.

If you're using any tools that are monitoring things and are giving you like, "Here's our average response time." You can be sure, probably no one experiences that actual time. You need to see the wider distribution. That's where we kind of see it as, "Hey, we're at a point now where the costs of storing large amounts of data, especially if you're quite clever about it, is not a huge, huge number." It still has some cost, but yeah, you want to store it all.

[0:40:17.7] JM: Yeah. Another note on the whole competitive landscape of software, my perspective in reporting on these different things is even the products that look like they're in very competitive spaces, like one of the early narratives when I was starting Software Engineering Daily was like, "Who's the cloud provider of choice? Is it Azure? Is it Google? Is it AWS?

As you drill into it, you realize, "Oh, cloud computing is actually a tremendous blue ocean space where there's room for lots of different people to arch-out what they want their domain specificity to be." As I'm sure, you have come to realize and your business is pretty much the same thing when it comes to monitoring and crash reporting where this is pretty much a fundamental thing that people want in their applications and there are just different flavors of it that makes sense in different context.

[0:41:13.6] JDT: Yeah, absolutely. There's no one right thing for everybody. That's absolutely correct. There're plenty of free tools people can try and use and things like that. That would be another though, usually false economy. Let's go and try and set up something for free and manage it ourselves rather than spending a couple of dollars that I see. It's usually a bad idea.

[0:41:34.0] JM: This file system that you built on top of S3 to store all of these large files, like these large — I guess, basically dumps of what's going on in the system when a crash occurs, the reason you built a custom file system on S3 was — If I recall what you said, is to categorize similar crashes that occur, and then you want to put all these similar crashes in a block together so that that block is easy to manage when you're moving it around S3. Is that right?

[0:42:10.3] JDT: Yes and no. It's not so much that they have to be similar. It's stored in a chronological order. What we maintain is several layers off storage. We obviously have in-memory, then we have local EBS, SSD backed stuff. Then we have S3. We cache the hot data first and then it's rolled up into these blocks.

Let's say rather than having — I'll use some round numbers. Let's say it there was 100,000 crash reports and they stacked up to, let's say, 100 megabytes data. We put them into a single block. We attach our own metadata to that that allows us to quickly be able to pick things out of that block very, very fast and also reduce some of the space sizes as well. Then they get stored in various caged buckets all the way down so that way the warmest data is obviously loaded very, very quickly. It also means that there's a process there that can handle massive, massive spikes in data coming through and how they block things up. Then we can do that longer term archival storage.

A Raygun customer can come in and say, "Hey, I'd like to export the last six months' worth of our data," and they might have had 40 million errors a month occurring, which is not unheard of, and it might turn into a multi-download for them, but we can actually fetch all of that data rather very efficiently. You can kind of imagine even with what the cost of git request against S3 is we probably wouldn't want to be storing hundreds of millions of individual crash reports and having to pull them one by one. It would just take; one, too long; two, cost a fortune; and three, it's actually quite slow to hit S3 by default if you are just fetching customer data from there, which is why we'd like to keep some of the more recent data warm closer at the source. It's kind of a classic caching strategy in there that's layered.

[0:44:03.5] JM: What are you using for the caching?

[0:44:06.1] JDT: We manage that ourselves for the disc storage in S3, where it's just the blocks who get moved around based on the temperature if you will, that's recent data that's in there. For the caching, that's done in memory. We are effectively just using the .NET cache providers that are built into the framework, which to be honest with you, I haven't looked in recent times at what their various eviction strategies are around that, but it hasn't been an issue for us, so I haven't had to look very deeply at it.

[0:44:39.1] JM: I see. The workers will handoff the file to S3 as well as to an in-memory cache elsewhere?

[0:44:50.3] JDT: No. They just pass it straight through to their storage processor, and then what it will do is to make sure that it ultimately ends up in S3 as part of a block that's been processed as well as in various caches too. Let's say, Jeff, you get an email that says or Slack notification or HipChat or whatever that says, "Hey! This new error has occurred." You want to click on it and you want to go into the Raygun app and see all of that right there.

It may still not have made it through to S3 at that point, but it can be served out at one of those hotter caches leading up to it and that storage services as well where it's at. We want to get to the point where if you get a notification, it might be within two seconds, three seconds of the error actually occurring and you can see the entire report within the application.

We used to get more people, early adopters were like, "Holy cow! This is amazing. Sometimes I get the notification of the error before my error page has even loaded." That was always our goal.

[0:45:47.6] JM: I see. The reason you cache it before writing it to S3 is so that it's in a place where the customer can easily and quickly access it, or like just on the fly, like shortly after the crash has occurred.

[0:46:02.3] JDT: They can access it quickly. Secondly, because we do want to collect up and blocks of data, so maybe — Like, say, it could be 100,000, could be 10,000, depending on the size. We want to then bring them into a single writable chunk that then gets written to S3.

It may not actually get to S3 for a few minutes, and we don't want to have you wait until it gets to that stage. We want to be able to give it to you very quickly. That's, again, part of that whole idea that if this spiky ingestion stuff, that service, we basically don't really have any issues, whether it just kind of quietly sits there and pulls the data back through.

[0:46:38.3] JM: You can imagine a scenario where one of your customers has like 10 or 15 crashes that are super similar within a single minute and you would want to batch — As those crashes come in, you would want to just keep them in-memory and then batch writes of 10 to 15 crashes out to S3 in these blocks.

[0:46:58.2] JDT: Yeah, absolutely. Yeah, although the numbers are a lot more than 10.

[0:47:04.1] JM: Okay. I can imagine. Sorry to drill in this, because I find it interesting.

[0:47:09.8] JDT: That's all right.

[0:47:11.3] JM: What's the advantage to getting those in blocks together?

[0:47:14.9] JDT: Because S3 is going to; one, charge you on the number of writes you do. If I write a thousand 1k blocks and I write 1,000 k block, one of them is the thousand times more expensive than the other. It's efficiently reduces our writes.

Secondly, there's obviously a latency to any request, whether it's write or read or whatnot. It's a round trip. We did a bunch of analysis ourselves on finding what the optimal block size was that would give — Let's say you had a really old error and it had to come out of the S3 block thing. What was the timeframe that we thought was acceptable for you to have to wait for that crash report to be ready? It worked out. I mean, it was basically a couple of seconds. So we measured and built out an entire chart that said, "Okay. Based on file size across these three, what's our time to get this back?"

We use that to define the size of the block. Then, lastly, because, one; it reduced the S3 cost. It also reduced the latency overhead because we weren't having to pull things. It also enabled us to do things like allow our customers to export massive amounts of data, and then it also spit up the time at the frontend with those caching layers, because customers didn't have to wait for it to be written.

Way back when we first sourced the service, it wasn't uncommon that you would get a notification that would say, "Hey, Jeff. This error code," and you click on the link and it would say,

"Hey, well, you're really fast. Wait a few moments for us to have to get that data stored and it'll be really in a moment." We wanted to get rid of that, which we did.

It was basically a triple or quadruple one. Customers got fast service both in terms of day-to-day usage and time the first error visibility. We saved a bunch of money on the backend and it made it — And it unlocked a bunch of features that would probably have been cost prohibitive either on price or performance using the older model.

[0:49:19.7] JM: You're not only storing these crashes on S3, but you also want to index these — At least some of the metadata perhaps about the crash in Elastic Search or probably keeping them in other file systems or other databases also. Can you talk about the multi-model database situation?

[0:49:41.4] JDT: Yeah. For example, we have our search system, using Elastic Search. We have a time series data store and then we have a relational data store, because if you try and use one data store for all those different types of information, you can end up tying yourself a not. We have these different data stores where information is getting stored. We also have Redis in there for a little bit of live data to show people what's going on. It really just routes to how we want to present the data to the customer.

[0:50:13.3] JM: Do you have — What's your pattern for writing to these different data sources?

[0:50:21.1] JDT: Let's say it was a crash report, that worker is going to ensure that it fires off an indexing job to Elastic Search to index at the search. It's going to send the data off to the storage system for that long term storage and the raw error report. It's also — In the case of crash reporting, it's going to update some various counters in a relational database for the group level information. How many times this occurred? How many users were affected? That sort of thing.

We do a lot more time series data on the performance side of things. In the performance tracking system, that would also fire off to, we use that druid for our time series data.

[0:51:01.6] JM: Oh, cool. Okay. Just to refresh people on the end-to-end architecture of this system, because I think this has been a great lesson so far in how to handle higher volumes of events. I'm using my podcast app, or some user is using the podcast app that I developed. I'm the developer. I'm using your system, Raygun, to see the crashes that occur. A crash occurs on my user's podcast app. The crash report is sent in a JSON Blob to Raygun servers. First, it hits the load balances. The load balances sends that request to — Is load balancing requests across API servers. That request is routed to an API server. The API server routes it to a queue in front of workers that process crash reports. One of the workers pulls the crash report off of the queue and starts to process it, and it's basically figuring out some aspects about that event in order to figure out what to tell the storage worker, I guess, and then it hands it off to the storage worker. The storage worker puts it in Elastic Search, MySQL, Druid, and batches writes of this type of crash to S3. Then the user can access all of these data sources. Is that correct?

[0:52:26.5] JDT: More or less. Yup. You're 99%. There's a couple of those things happen in the actual crash worker, but for the most part, yes.

[0:52:36.0] JM: Okay.

[0:52:35.8] JDT: It's quite — It is a lot of moving parts.

[0:52:41.5] JM: Yeah. It sounds like a pretty interesting system to wake up every day and work on, because I'm sure there's lots of performance, tuning things that you can do throughout that process, but at the same time a lot of it seems like it's probably a stable architecture.

[0:52:59.1] JDT: Absolutely. It is relatively stable, and this goes almost full circle back to how we started, which is what we typically find is for six months the system is just absolutely rocking along. It's really good. Then our fantastic customers and sales team and things like that keep closing more and more business and some little crack start to appear and we kind of go, "Oh, goodness! It's time for the next round of work in there."

It comes and goes, but that's certainly really fun problems. I'll be honest. In fact, when I first got into the industry, I used to think of, "Working at Google and the scale of the problems." We're

definitely not at Google scale yet, but it's one of those times where you are starting to go, "How could I cost effectively juggle a few petabytes of data and how can I real-time query some of these to give our customers the visibility that they need and sub-second response times if they want to slice and dice." And things like that. That's the high quality problems they have to think about.

[0:54:04.5] JM: Totally. You talked earlier, like you don't necessarily have a bursty workload, because you have stable traffic throughout the day.

[0:54:15.0] JDT: It's more that we've got the point where the bursts are — When it our first hundred customers, right? Because we've got thousands of organizations now using us. When it was earlier on, you might get one customer and go, "Oh, my goodness!" But we have sort of evolved to the point now where there's probably at any one point two or three or ten customers that are having some massive meltdown. Now, it's still not that significant to us to have to deal with it.

[0:54:40.7] JM: It's more like steady customer growth is a bigger concern of yours than bursty customer usage.

[0:54:47.8] JDT: Yeah, absolutely.

[0:54:49.1] JM: Can you talk about, what are some of the things that are starting to break, because I mean the way that you've described the architecture so far, it seems like, "Yeah, just horizontally just scale that to the moon. Why not?"

[0:54:59.2] JDT: Yup. Absolutely. Examples of that would be, like I said, where I mentioned the worker layer, where we can reasonably quickly spin up an additional instance to add additional work processing. We'd actually really like to try and move it to more over containerized workload that auto-expands and does all of that and is a little bit smarter, that sort of thing.

We are in the process at the moment as well around some of those data stores which are horizontally scalable. Just making them be a little bit more programmatically-driven, if you will.

Using various obs tools to make sure that we can spin up and shut down instances bases on what's needed. We're doing a bunch of work around that.

To be honest, that's a little bit about cost optimization and more just about the majority of the infrastructure and taking it up a level in there. Those are the areas where we might say, "Hey, maybe we're adding one server every couple of days. We need to make sure that this is sort of falloff a lot easy and almost anyone on the team can do it," rather than saying, "Well, there's a couple of people who manage the AWS account and they can go on it manually run up a name off of it," things like that we are improving around.

We also continually just find various areas to improve. For example, that storage system that we've talked a lot about today. We had our initial data structure for those chunks. That's been in operation for a few years now. We actually have a V2 version of that data format for the chunks that basically gives us an extra order of magnitude improvement and response times for customers, because we're finding, "Hey, now that we actually have millions and million of these giant chunks, what can we do to improve upon this and how can we appendix our metadata to them to provide extra querying and filtering?"

There are things like that just sort of gets slow with time. We want to improve those. Yeah, range of stuff. There's also new features always add additional demand. When we first put in that storage system, we're like, "Oh, cool. Now, we can build an export feature really easily." We've put that in.

We do have APIs available for querying data, specially enterprise customers like to do that a lot. Hey, maybe there's a query that's just not very efficient, so we need to work on that. Just everyday there's always little extra bits, and that's what we're monitoring is so important for understanding where are the hot spots, where is that time going?

[0:57:20.3] JM: All right. I know we're nearing the end of our time. We've talked about pretty much the full — The high-level view of the architecture. Since you're the CEO, tell me a little bit about running the business. Just to close off, what are some non-obvious aspects of running Raygun?

[0:57:39.5] JDT: I mean — I guess there's a background. I obviously have learned how to build out a sales team, which is what we've got here in Seattle. I moved to America to build out the sales team in the U.S. We're still headquartered in Wellington, New Zealand. That's been interesting. Also, marketing. The marketing team reports through to me, and they've wonderful job at getting the name out there. Credit to — I think it was Freya, has been working with you as well.

[0:58:04.9] JM: That's right.

[0:58:05.5] JDT: Jeff, on getting on your radar.

[0:58:06.9] JM: Yeah, she's great.

[0:58:08.4] JDT: Yeah, understanding all that — Yeah, I think she's fantastic, which is good. Learning the different areas, and that's kind of the day-to-day thing. Also, managing, being on a board of directors, answering to them, understanding investment rounds. You basically end up becoming the ultimate generalist, really.

The thing that I found hard is they transition from being a software engineer. I think people naturally will always fallback to what they're most comfortable with. A few years ago, what I would find is that I would frequently end up writing about a code for the system, and it was like, "You know what? As much as I enjoy doing that, it doesn't have the biggest impact for the company." We've got a bit team of people far, far smarter than me to write code. Why am I doing this? I kind of realize that it was because I need to be putting off something that I needed to do that was more difficult.

I actually ended going out and I moved to a 13-inch laptop. It was bloody eloquent to write code on. I couldn't run visual studio instances all over the show, and it was a forcing function to be like, "You know what? If you're the CEO, generally speaking, you should be probably be fine with like a Chrome Book, possibly even an iPad. You just go send emails and use the internet."

I had to force myself out of that comfort zone a few years ago. Now, to be honest, I still absolutely love to code. I have a beefy as hell machine at home that I do various bits and pieces

on in the weekends and I've been exploring things like TensorFlow, because I want to make sure that I still have a good strong understanding of this, and I love talking with our tech team about this performance issues. Gets me off. I love high performance software.

That was a difficulty, is the business would grow a lot faster when I wasn't writing code, because I'd be focused on the growth. That was a big transition. I guess, lastly to that point, coming from engineering land, we tend to think as engineers that we're like the smartest people in the room. Bad news folks. We're not. There are smart people on every category of the business. They're all delivering value in different ways.

I think there's a certain argument to be made that perhaps the Dilbert cartoons for 20 years are perhaps over inflated our egos about our talents versus other people.

[1:00:30.8] JM: Yeah. I completely agree. Try putting the engineers in charge of enterprise sales and you'll see what happens. You'll see how smart they are.

[1:00:39.7] JDT: I still remember when we first got into some of the pay per click advertising and there was a couple of engineers who were like, "This would never work. I've never ever, ever clicked on an ad." I was like, "You think you've never clicked on an ad. You think you've never been —" it's like, "Guess what? It works really well."

Yeah. I'm not just saying that, because now I'm officially the pointy-head boss, just to be clear. I possibly am the dumbest person in the room.

[1:01:07.4] JM: All right, JD. Well, it's been great chatting. Really good show, and look forward to hearing more from Raygun in the future.

[1:01:14.5] JDT: Yeah, absolutely. It's been my pleasure, Jeff. I really appreciate it. If you can't tell I can [inaudible 1:01:19.4] lyrically on just about anything, so I'm happy to do another show some time. Thank you.

[1:01:23.7] JM: All right. Sounds good.

[END OF INTERVIEW]

[1:01:27.3] JM: Simplify continuous delivery GoCD, the on-premise open-source continuous delivery tool by ThoughtWorks. With GoCD, you can easily model complex deployment workflows using pipelines and you can visualize them end-to-end with its value stream map. You get complete visibility into and control of your company's deployments.

At gocd.org/sedaily, find out how to bring continuous delivery to your teams. Say goodbye to deployment panic and hello to consistent, predictable deliveries. Visit gocd.org/sedaily to learn more about GoCD. Commercial support and enterprise add-ons, including disaster recovery, are available.

Thanks to GoCD for being a continued sponsor of Software Engineering Daily.

[END]