EPISODE 388

[INTRODUCTION]

[0:00:00.3] JM: After raising $18 million, social networking startup Yubl made a series of costly mistakes. Yubl hired an army of expensive contractors to build out its iOS and android applications. Drama at the executive level hurt morale for the full-time employees. Most problematic, the company was bleeding cash due to a massive over-investment in cloud services.

This was the environment in which Yan Cui joined Yubl. The startup did have traction. There were social media stars who would go on Twitter and announce that they were about to jump on Yubl, and Yubl then would be hit by an avalanche of traffic. 50,000 users suddenly logging on to interact with their favorite celebrity, this was a significant traffic spike for Yubl.

How do you deal with a traffic pattern like that? You've got these spikes that just occur almost randomly that massively change the traffic on your website. One solution is serverless computing. AWS Lambda allowed the company to scale up quickly in a cost-efficient manner. Yen began refactoring the entire backend infrastructure to be more cost-efficiently heavy leveraging AWS Lambda.

Unfortunately, Yen's valiant effort was not enough to save the company, but there are some incredible engineering lessons from this episode; how to build cost-effective scalable infrastructure. How to migrate to effective infrastructure? Also, there're lots of lessons if you're just building a startup or if you're deciding whether to work at a company as an engineer. There're lots of business lessons here.

It was really fun talking to Yen and this is just a great story. I think this is one of the episodes that I've done over the 500+ episodes just because of how you unique and strange this story is. I really hope you enjoy it.

[SPONSOR MESSAGE]

[0:02:15.9] JM: In the information age, data is the new oil. Businesses need data and there's no better data than real time data, which is why Amazon Web Services built Amazon Kinesis; a powerful new way to collect, process and analyze streaming data so that you can get timely insights and react quickly to new information. Here's the thing, websites, mobile apps, IoT sensors and the like can generate a colossal amount of streaming data, sometimes terabytes an hour. That, if processed in real-time can help you learn about what your customers, applications, and products are doing right now and respond right away.

Amazon Kinesis from AWS lets you do that easily and at a low cost. With just a few clicks, you can start sending data from hundreds of thousands of data sources simultaneously. Loading in real-time, it lets you process and analyze the data and take actions promptly. All you need to know is SQL. Kinesis also gives you the ability to build your own custom applications using popular stream processing frameworks of your choice. With Kinesis, you only pay for the resources that you use. There are no minimums, no upfront commitments. To learn more about kinesis, just go to kinesis.aws, and let's get streaming.

[INTERVIEW]

[0:03:53.7] JM: Yan Cui is a cloud architect and an engineer who writes about software on theburningmonk.com. Yan, welcome to Software Engineering Daily.

[0:04:02.0] YC: Hi, Jeff. Nice to be here.

[0:04:04.3] JM: You've written about migrating Yubl, which is a social network that you worked at to a serverless architecture. This was a social network. It was originally written how you would expect a social network to be written. I think it was on Node.js and with some sort frontend JavaScript technology, but you made the effort of migrating it to Amazon AWS Lambda, which gives you a lot of great features that we'll talk about, but it's a bit of a usual migration. I think it's a technology that people are talking a lot about.

Before we get to talking about serverless and the migration process, give an overview for Yubl and what the technology stack was for the social network when you joined it.

[0:04:52.6] YC: Sure. Yubl — I will pronounce it Yubl.

[0:04:56.1] JM: It's Yubl. That's right. I'm sorry. I forgot the — Sorry for that pronunciation.

[0:04:59.9] YC: That's fine. It stands for your social bubble. It was a social networking startup and we were trying to be a social network that is — I guess, you can think of it as a mix of Twitter, Facebook and Instagram. Instead of posing 140 letters, you'll be posting something that's like Instagram but at the same time you can attach interactive elements on your post so that your followers can follow them directly or click on one of the buttons and go through a location page to see where perhaps you're hosting a party.

When I joined the company in April last year, we had a few monolithic systems all written in Node.js, Node.js application running on Express.js for the web API and then there are web sockets workers as well as a background worker that process task for AMQP. It was what you expect from a monolithic system. Everything is hosted on Amazon EC2, but it looks simple on paper, but once you start peeling away the covers, you find a lot of hidden complexities and dependencies that's not obvious from the, I guess, 30,000 meter view.

One of the problem that we ran into often is because being a small social network, our traffic is usually fairly minimal, but we have number of high-profile users on them. Some users would — I think we had a couple of users with about 50,000, 60,000 followers and whenever they would do something, they would drive a lot of traffic on to the app all at the same time. For example, one of our influencers, she had, I think, about 40,000 followers at a time and often she would run some campaigns. She would do giveaways and she'll write a post and say, "Hey, guys. I'm going to give away some designer handbags. Vote on these two pictures. Which one do you like better? Then I will randomly choose a winner from the people that voted on my post and I will announce winner at 10:00 tonight."

Pretty much at exactly 10:00 there will be a whole flood of people coming to the app right at 10:00 and because the way EC2 scales, it takes 10, 15 minutes to get a new server spawned and able to serve your request behind a load balancer. This kind of spike of traffic really didn't work very well. To accommodate the kind of spikes and traffic that we see, often, sometimes 70 times what the normal traffic would look like in one of these influencers running a campaign. We will run our servers at a much higher instance type than we really need to and we will leave a lot of headroom so that we can scale, we can handle many spikes. Even then, we still find that often that's not enough and when we do need to scale up it takes a longtime to get our new instances serving our request. By the time those instances are ready, most of the time the traffic was already there and the spike is already on its way down.

We really got a worst of both worlds, really. The move to Lambda and we faced it's a much — Almost instant scalability and its cost-effectiveness when you have a system that doesn't have high traffic most of the time, but doesn't need to spike very quickly. It's a great fit for the position that we found ourselves in.

[0:08:47.4] JM: Yeah, you're describing about as bursty a workload as it gets, where basically you've got long periods of inactivity and, essentially, unpredictable bursts of massive activity. It's kind of different than the classic problem of some company that has a nightly Hadoop job that they're running and like, okay, they're going to need to spin up a bunch of servers or allocate some servers that were in use high traffic hours for running Hadoop jobs. This is totally unpredictable.

You're not going to be like schedule this, and that's why it's important to have some sort of system like AWS lambda and just to set the takes for people who are still not familiar with this topic, this Amazon Lambda topic. We've done a bunch of shows on it, but basically the premise is you can run a function against some opaque blob of compute. In reality, it's a container that gets spun up in response to you calling that function or an event triggering that function on Amazon server somewhere or you could be talking about Google or Microsoft Azure. Everybody has a serverless thing at this point.

Basically, the core innovation is you just write a function and the massive amount of compute across AWS will accommodate that function, will schedule that function against a container somewhere and the advantage of this is that you're not paying for an addressable EC2 server instance. You're just paying for some machine that you don't know about or care about and it's fine. It will do the job for you. It will satisfy your requirements to execute that code. In addition to the cost advantages, you get scalability advantages. That what makes this such an appealing technology that's really driving a lot of adoption. Have I painted the picture correctly?

[0:10:59.6] YC: Yes, absolutely. Scalability and cost are two of the main reasons why you want to adopt a serverless technology such as Lambda or the equivalence on Azure and Google clouds. Besides the cost and the scalability, I think another interesting benefit with this new paradigm is how simple, how quickly you're able to move from heaving an idea to having something that's production ready, that can burst and handle a large amount of traffic without have that normal face of capacity planning and maybe bind reserve instances in order to cut down your cost. All of that goes away.

As a developer, once I've got used to writing, working with serverless technology, it really just doesn't make sense to go back because having an idea, "Okay, we need to build this feature," and then you go from that. What are the things you're going to need? Okay, I'm going to need to have an API. May be some stream processing happening in the background. From then on, how to go about actually making that happen.

Back in the days, you'll have to learn about which referent word to use, which languages, how to go about configuring a service and how to bake the AMIs and how to do continuous deployments, and don't forget to patch your service as well so that you have the latest security updates, and so on. All of things just go away. Now, have the idea to know what you need to delivery. You just go and do it without having to go to all the pervious steps that you have to do in order to get there.

[0:12:41.8] JM: Right. You discussed the way that the application was architected. It's a social network where you've got these certain power users that are spiking the traffic occasionally. In order to accommodate this, when you walked in or shortly after you walked in, the setup was you're just overpaying for large compute instances. They're going to sit dormant most of the time. Problematically, when you have this traffic spike you still can't even accommodate it even with the extra headroom that you've given it. Like you said, the worst of both worlds.

Operationally, what did that look like? When you had a traffic spike, is it just like you get a page? You're asleep and you get a page on your pager and it's like, "Wake up. Please spin up some more instances. Please open up your monitoring tools and start fighting this fire that occurs every time there's a spike in traffic." Is that what happened?

[0:13:47.3] YC: Fortunately, because the app was only live in the U.K., so we didn't have to get and working up at 3:00 a.m. in the morning kind of scenario, but most of the time when those spikes happen, we are still awake and we typically just get a page because the number of 500 errors has gone over some threshold and pretty much every single time you look at the dashboard because you have more traffic going through the system that you have enough server to handle them. There wasn't any easier way to deal with that than to just to let the EC2 auto-scaling kick in and hope for the best.

[0:14:24.7] JM: Yeah. Let's start to talk about this migration process, because I'm sure you looked at this technology and you're like, "Well, this could solve our problems." We've done a bunch of shows about how do you migrate to Docker, or how do you migrate to Kubernetes. The story that I always hear is you take maybe your least sensitive service and let's say the classic example was Netflix with its job board. If you're Netflix, the job board, yeah, it's important. People need to apply to jobs but it's not like this is going to — If you screw up a refactoring of the job board, it's not going to be the end of the company. They would start with the migration of something trivial, like the job board.

When you were looking at Lambda and you're like, "Okay, we need to migrate these production systems to Lambda. Was there a service that you picked out to start with to sort of say, "Okay, let's get some adoption going for this product."

[0:15:24.5] YC: When I joined the company and I sat down with the team at the time, we talked about what would a good architecture look like for us and we did that before we even made a decision to migrate to Lambda. Most of the things that we came up with are the things you would expect these days, the ability to deploy changes quickly and mentally and without any downtime and the ability to deploy features independently without them stuck in with each other's toes, also be cost efficient and reduce the amount of hand holding we have to do on our architecture.

From them on we kind of looked at, "Okay, Lambda seems a really good fit for what we're trying to do. How do we go about migrating to Lambda in a way," like I said, "that's not so risky and also allows us to continue to deliver value to our users preferably faster than we have been able to in the past." At that point, we are doing releases to production, maybe four, five times a month. By the November, so about five months after we wrote our first Lambda function production, we were doing releases to production something like 80 times a month. At peak, we did about 150 production releases in September. We're pushing out a new feature and that required a lot of tweaking and adjustment straight after it went out.

As part of the migrate process, what we typically do is we need to work on a feature. In this case, we started off with fairly, I guess, not — What's the right word here? Not important. Not trivial, as in less risky feature and we started to chip away at the monolithic API and we started, moved our feature into its own API in a way that you would do when you migrate from a monolithic system to a microservices with that service being self-contained with its own data stores. At that point, that API will be returned using Lambda.

Because the clients, they're coupled with the monolithic system, we also didn't want to change the client and force ourselves into a situation where we need to do lots of deployment between the client server. What we typically would do is we would create a new API that's micro-compatible with the legacy system and we also update the legacy update the legacy system to proxy is endpoint to just hit the new API instead. As the client [inaudible 0:18:05.6] catch up with the changes we're making. They would then start to use the new API directly and eventually we'll deprecate the legacy endpoint no the legacy system.

[SPONSOR MESSAGE]

[0:18:27.4] JM: Your application sits on layers of dynamic infrastructure and supporting services. Datadog brings you visibility into every part of your infrastructure, plus, APM for monitoring your application's performance. Dashboarding, collaboration tools, and alerts let you develop your own workflow for observability and incident response. Datadog integrates seamlessly with all of your apps and systems; from Slack, to Amazon web services, so you can get visibility in minutes.

Go to softwareengineeringdaily.com/datadog to get started with Datadog and get a free t-shirt. With observability, distributed tracing, and customizable visualizations, Datadog is loved and trusted by thousands of enterprises including Salesforce, PagerDuty, and Zendesk. If you haven't tried Datadog at your company or on your side project, go to softwareengineeringdaily.com/datadog to support Software Engineering Daily and get a free t-shirt.

Our deepest thanks to Datadog for being a new sponsor of Software Engineering Daily, it is only with the help of sponsors like you that this show is successful. Thanks again.

[INTERVIEW CONTINUED]

[0:19:53.2] JM: Now, I want to take a little side note here, because one of the things I really liked about your series of blog posts about this serverless refactoring is you told the story in the context of what was going on at Yubl, the company, as this refactoring was occurring. It was basically like the company raised — It's not completely clear what happened to me, but it's something, like the company raised some money. It hired too many people too quickly and kind of like went out of business. Reading it, I was like, "Well, this is like a classic — Just like classic startup story," and you as the engineer, you're just like — One day, you're like, "What? We're out of money."

Not exactly clear what happened, but you want to talk about that a little bit and how it pertains to the engineering team, because what was interesting about this is you had a fairly new service. It was a fairly new social network. It was not super old but you were in a unique position to refactor it. I'm just trying to understand the size of the team, because looking at your refactoring there was a lot of work that went on, but I'm just trying to understand if this was an army of one type of refactoring or if you were actually orchestrating a large team. What else was going on in terms of the climate of the company?

[0:21:15.4] YC: That's actually a very interesting story on its own. As I was watching Silicon Valley, it's amazing, so much crazy things that that does on this show and to witness firsthand at Yubl. The company started, I guess about three years ago now and they spent maybe the first two or three and a half years just in solid building mode. There's not much market penetration. The product was delayed and it was so bad at one point that both the CTO and the CEO was — The CTO was dismissed in a pretty public and I guess humiliating fashion and the CEO was pushed aside. Then a new CTO came in and just realized what a mess it was.

Between the old CTO [inaudible 0:22:10.3] and the new CTO join in, one of the previous engineers that took over as the head of development and he solution to we are behind, we can't deliver this system in three months. Was hiring an army of developers that I think at one point we had something like 20 android developers and 20 iOS developers all of them are expensive contractors. There's no [inaudible 0:22:39.4] in the company. They were just burning money. They did raise a lot of money at the start, but by this point I think a lot of money were already gone. A lot of the people that were there where on really big contracts, they won't deliver much. What's also funny is that they had a whole marketing product without a product.

The new CTO came in. He's a friend of a friend and one of the first thing he did was to just pretty much carried off anyone who he can verily easily see as just a push to someone who is maybe good at talking the talk but can't walk the walk.

From them on the team was down to about I think four iOS developers and four android developers, still all contractors. We have massive problem to hire permanent mobile developers. For the server team, I think we're down to the last two people by the time I joined the company. After that, he also bring in a few other guys that he knew from previous jobs who he knew were, A; good people they can work with, and B; also these guys that knows how to deliver stuff and they have done that. They have good track records.

One of the things that was interesting as well is that even though everything was running on EC2, on Amazon services, most of the guys that joined us after I started had no previous AWS experience. One of the things that we also had to do was to make sure that those guys are up to date with the basic AWS ecosystem and also what's this Lambda thing. How do you deploy software without servers?

At the peak, I think we had, during my time, we had the six server developers. We typically would break up into smaller teams of two on a per feature basis where we are working on new feature, there'll be two of us that bring down on that particular feature, get it done in a week or two and then we move on to the next thing. As we've changed features, sometimes we will regroup and we'll work in different pairs.

In my particular role, I'll also try to look at what we're doing as a whole so that I'll provide some, I guess, vision, for where we're going and also identify problems that we would likely face soon. As we go from having one Lambda function in production to having, in the end, about 170 Lambda function running in production, along the way I can see that, "Okay. Soon, we're going to get to the point where we have many services that depend on each other." Tracing is going to be an issue, so we started looking at how do we adjust the issue of distributed tracing and what do we do about centralize all our logs so there is easy searchable. We also would take some time to work on those kind of problems that we have with our growing pains.

All in all, it was a very small team that migrated — Not only migrated their existing features, but also worked on a lot of new features. Along the way, we also adjust many of the problems both security but also API design-wise as well. It might sound crazy, but this is an app that when I joined, had the user recommendation system that returned the first 30 users from the database.

[0:26:12.4] JM: I read that. I read that in your blog post. This is hilarious. User recommendation says if you log in for the first time, and it basically recommends the first 30 people that ever join the social network. It's like the founder of the company, the first employee of the company. It's just like, "What is going on here?"

[0:26:32.6] YC: Yeah. It's crazy. They also found a problem whereby, "Oh, whenever you try to see the followers for this popular user, the app would just freeze up." When I looked at it, the API had no pagination. We also found that a lot of APIs were just really poorly designed. As part of migration, often, we also had to redesign the API from the ground up and adjust many of these problems [inaudible 0:26:32.6] the client team to make sure that we have the right approach in place for retries. We started to put in place things like circuit breakers so that when this particular feature is struggling, don't just keep hammering away with infinite retries, which [inaudible 0:26:32.6] a couple of times when we have one of those spikes and the server was crumbling under but the client was just infinitely retrying the background.

There's a lot of problems that came with, I guess, not having that discipline, the maturity in the previous engineering team. Everything was going in a very good way and we started to delivery features. When I joined the company, the product guys were, from what I told, really good guys, but they were coming to us before these crazy solutions that once you talk to them a bit more you realize they are just so used to being told what they want to do can't be done. Instead, they had all these crazy workarounds that is not what they want but is what they think the server team can deliver.

After working with us for some time, I think that perception started to change. We were able to do things the way they want to and also do them quickly and deliver those features in a timely fashion. Everything was moving in a very good direction. Unfortunately, at that point, we started to run our money and even though there's been lots of background discussion with the investors, one of the investors follow through with the promise, the money they did promise us and our lead investor at this point, he came from a real estate background and he made a decision that, "I'm not going to put any more money in because you have been raised enough funds from other people so I'm just going to pull your funding and put you into an administration with immediate effect." At that point, I was actually in a conference in Sweden where my boss paid me on Slack, "Oh, we need to talk." Apparently, one of the guys in my team was doing a department to production. At that point where he turned around and he found a couple of guys dressed in suits and everyone was being gathered around the company to announce that, "Oh, we are into administration now." It's pretty crazy.

[0:29:27.8] JM: Did you say in administration or arbitration? What word did you use?

[0:29:31.8] YC: In an administration. That's when the administration has come in to start to take away what's left of the company to pay the creditors —

[0:29:41.5] JM: Oh, like liquidate. They were liquidating it.

[0:29:43.0] YC: Yes.

[0:29:45.5] JM: Okay. This story is so insane for so many reasons. I'm torn here, like, "Do I go further in the Machiavellian disaster of kind of the management problems or do I go into serverless stuff?" I think I'm going to go with the former just because it's a little more unique. As curious as I am with your serverless migration, I got to ask you a little bit more about this company. Okay, in the aftermath of this Machiavellian situation where you've got these CEO, or CEO gets pushed aside. CTO leaves the company or something and these investors that probably were not a good fit. You came in — It sounds like after there had been some crazy spending, 80 contractors across mobile teams. As you come in, there's some calm that starts to settle across the company. Was there still a decent user base around this time? Having spikes where you've got 50,000 active users in a given instance, that's not easy to create. That's a special type of application if you've got that kind of usage.

[0:30:58.4] YC: It wasn't 50,000 active users at the concurrent users, it was a popular user with, say, 50,000 follower.

[0:31:05.6] JM: Oh, I'm sorry. Okay. Right.

[0:31:07.2] YC: Who pretty much all come back and log in to the app at more or less the exact same time to check on these popular users new post.

In total, I think we ran live in February. Again, a lot of that was possible because a CTO came in and he was the one that brings a lot of the calm that was there when I started the company. At that point, I think we've been live for about 2-1/2 months. There's been quite a lot of a pretty aggressive marketing campaign, working with YouTubers and Instagram, people that are what you'll find as, I guess, social media influencers. Working with people at universities and campuses. They were doing pretty well. From talking to a lot of the VC funds, it sounds like we were much further ahead compared to other similar social networks that's come before us.

At this point in our lifetime, the problem that we had was that because of VCs were not engaged from the start, again, it's a business decision. I don't know why there was. At that point, we had about six months of live data. For VC to get engaged at this point, I guess this point in our life, they wanted to see at least 12 months of data to better understand what our projection looks like which is why we couldn't get the VCs to get involved at that particular round of funding.

[SPONSOR MESSAGE]

[0:32:51.8] JM: Look for a job more efficiently with Indeed Prime. Indeed Prime flips the typical model and lets you find a job more efficiently even while you're busy with other engineering work or coding your side project. You simply upload your resume and in one click you get immediate exposure to companies like Facebook, Uber and Dropbox. The employers that are interested will reach out to you within one week with salary, position and equity upfront.

Don't let applying for jobs become a full-time job itself. With Indeed Prime, the jobs come to you. The average software developer gets five employer contacts and an average salary offer of a $125,000 through Indeed Prime. It's 100% free for candidates. There are no strings attached.

Sign up now at indeed.com/sedaily. Thank you to Indeed Prime for being a repeat sponsor of Software Engineering Daily. If you want to support the show while looking for a new job, go to indeed.com/sedaily.

[INTERVIEW CONTINUED]

[0:34:06.7] JM: Right. Okay. I get that. you can't raise — You raised a series A, right? You would have been having to raise a series B with six months of data. Is that what would have happened?

[0:34:16.4] YC: Yes.

[0:34:17.6] JM: Right. Which is pretty tough. I think about the company and, like, was the infrastructure really that expensive? How expensive were after you got everything ported to Lambda. It did look like you had a lot of platform as a service products, or like database as a service stuff. I know that the cost to that adds up. Do you remember what the expenses were like when the company was shutting down?

[0:34:44.2] YC: When I first joined the company, the EC2 bill was pretty high. Besides that, there's also a lot of expenses associated with MongoDB. They were using Mongo Lab and they were using the biggest instant available even though they didn't have any data to justify that size. Again, a lot of that goes back to some of the early decisions made by, I guess, the technical team. The long-term expensive contracts were signed and there's even crazy things like moderation were done by a third-party and they were being paid tens of thousands a month with barely any post to moderate. You could have a paid intern to sit in a corner to do that.

There's also expensive contracts being signed for — I think it's a mixed panel, which is pretty good for something to get the guests started with, but we were paying something like 3,000 a month for a package that contains so much data. We had no hope of using it or ever filling it. One of the things we did early on was to put together our own analytics pipeline and stream everything, all events to Google BigQuery. I think by the time the company went down, we were running queries against BigQuery and our monthly bill was something like 3 cents. It was ridiculous, because if they give you such a big free tier and we didn't have really that much data and the query that we were running are mostly talent bound for using only data that's inserted to BigQuery for the last 24 hours. They were pretty cost efficient.

Compare that to what we were paying mixed panel for, and the mixed panel was good for certain things but it didn't understand the fact that we are social network. A lot of the questions we want to ask is about who follows who? Who are the most active users that are following this particular person? Those kind of questions we couldn't do with mixed panel, whereas with our own analytics pipeline, that was something that we were able to model, we were able to work with the BI guys who just writes some SQL query and we have the infrastructure in place for him or someone else who work with him to say, "Okay, I'm going to fund some marketing campaign with this influencer.

I'll work with the BI guy to identify this BigQuery query so that when we send push notifications to all his followers it would target only his followers and we can do A-B testing and there's a webpage for you to submit that request so that either the CTO or, I guess, the head of content can say, "Okay, yup. Sure. That looks fine. That's going to target everyone who follows [inaudible 0:37:29.6] and that's going to be 35,000 users and this is the message we're going to show them on android, on iOS. Okay. Approve." Then the background process would kick in, again, using Lambda functions to send out all those push notifications against the Google cloud messaging or the Apple equivalent.

It;d would be a lot of those two lane infrastructure. By doing that, we were getting ourselves into a position where we can just get away from mixed panel and as we move to breaking down a system and start to move core parts of the system into microservices with its own databases. Most of the time that would be done on DB.

We were also getting our self into a position where eventually, maybe hopefully in the few more months, we would be able to move away from having that really expensive MongoDB database. Because of the way we are using MongoDB, the crazy thing is we're using the biggest instance available that's much bigger than the amount of data we have but we were still running to performance issues because of the way the data modeled really badly. Because Mongo lets you do whatever the hell you want, so we were shooting ourselves in the left, front and center. It's crazy. It's super crazy.

[0:38:54.0] JM: It's crazy. Painful to listen to it. It's just like the most severe case of premature optimization I've ever heard. Maybe you answered my question there, and I've missed it. Do you know the number for how much the monthly bill was at the end? It just sounds like kind of a tragedy, because it's like, "Here is a service that some people actually were using," where there is some semblance of traction and it had to get shutdown kind of just because of repeated budgeting and hiring issues and it's just like — Yeah, avoidable mistakes. Do you know what that bill was? I guess it's not important.

[0:39:34.9] YC: I don't remember the exact number, but it was more than 10,000 a month for EC2 and that was more than —

[0:39:43.6] JM: For EC2 alone, wow! Okay.

[0:39:45.1] YC: Not just EC2, but Amazon web services and some of the other services that we are using as well. To put into perspective, that's more than what I paid — A previous team I was working in the paid for running a game with nearly a million daily active users. When I look at that bill, I was shocked how much we were spending for a system with such little number of users.

[0:40:14.9] JM: Yeah. I got plenty of questions that I could ask a CFO at this point, but it's not like finance daily. I guess we can't really go there. I guess let's talk a little bit more about the serverless refactoring you were able to do. Basically, it sounds like you could have gone in a million different directions of — Because you were looking at how do we improve scalability and reduce cost, and there was probably every access you could have looked at. You could have looked at analytics and done serverless stuff there and improve the cost. You looked at recommendations, user notifications, testing in CD, logging. Let's see. Where should we start? What was the most important — What was the most interesting or unusual serverless migration move that you made in that period where you are refactoring this kind of over engineered application into a lower cost serverless infrastructure?

[0:41:16.0] YC: I don't know if there's anything that's really unusual, because most of the things we were doing were pretty much by the book, migrating APIs, chipping away from a legacy monitoring system into microservices, do endpoints at a time and [inaudible 0:41:32.1] API gateway, Lambda functions. I don't think there was anything that was particularly exchanged, I guess.

[0:41:42.2] JM: Let's use the monitoring one as an example. Let's talk about logging and monitoring. Give me a picture of the logging and monitoring strategy before these migration efforts began and then a picture for how you migrated that to some serverless stuff.

[0:41:59.8] YC: All of our legacy systems that was running on EC2 instances. All of our logs were being sent to a self-hosted cluster of elastic search instances. Therefore, Lambda, [inaudible 0:42:14.1] go straight to cloud watch and that's okay when you have just a small handful of functions in there. Once you start to have more and more functions, cloud watch really is not very good solution because it doesn't make things easy searchable.

We did a bit of work to ship cloud watch logs, from cloud watch logs and into the same elastic search cluster so that we have all of our logs in one place. As our services expanded and we started to build APIs that depend on other intermediary services as well, we started to run into problem that trying to debug and trace problems are really difficult and we knew that we needed to have distributed tracing in there as well. We invested in some work into standardizing a way of capturing and forwarding correlation IDs from the first, I guess, the edge service all the way through to your second or third layer of services that you depend on making sure that same correlation ID as well as the, say, the user ID, as well as the ID for the post if that's relevant, or get recorded along with other relevant lock messages.

That made things a lot easier for us in terms of trying to find out — For example, if I post something and one of my followers didn't get my post, where could things have gone wrong and having the log, all the relevant logs as the services traverse from the first service to the relationship API where we query who are your followers and then all the background processing that happens over Kinesis and SNS and having the same correlation IDs and user IDs that flows through all of those systems. It made things a lot easier.

[0:44:05.3] JM: We actually just did a show about serverless continuous delivery, and I think otherwise I would ask you some about how continuous delivery works with the serverless architecture, but people who are curious about that can listen to that episode. I guess what I'd like to get from you, since you spent so much time migrating an entire architecture to some serverless stuff, do you have any general principles that you kind of learn from the experience related to the serverless?

I've done shows about migrating to Kubernetes and migrating to Docker, like I said. What's different? What's different here? What's different about migrating to AWS lambda stuff?

[0:44:49.2] YC: I think, for me, I've done that migration myself as well, migrating a system from EC2 hosted, your typical service setup to using Docker and then later on started looking at the Kubernetes or other schedulers. I'd say the difference in migrating to Lambda is that the barrier of entry is so much lower, sort of the learning curve to get started to trying things out is much lower. Also, I guess the thing that often people forget is that even when you have containerized all your application and you have Kubernetes or Mezos or whatever schedulers, doing all the scheduling for you. Amazon also have ECS as well. You are still responsible for paying for any headroom for a burst in traffic and also when you do need to scale up, you still need to scale up both in terms of the containers as well as the virtual machine, the EC2 instances that are required to host your containers. Some of the constraints around how much time it takes to scale EC2 instances, a cluster of EC2 instances. They still apply to you as scale at least.

For Lambda, a lot of that are now handled by Amazon. Sure [inaudible 0:46:11.3] I'm pretty sure it's still containers and schedulers, but Amazon is doing all the heavy lifting for me so I don't have to and they also, as part of their core competency, they're also making sure that all their OSs are readily patched so that you have the latest security updates and whatnot which again is something that we'll be responsible for doing if I was to manage my own cluster of containerized services.

[0:46:44.4] JM: All right. We've covered the architecture stuff pretty well, the engineering stuff pretty well. I'm sure we could go deeper. I have a lot of questions that I wanted to ask you about that. Since we're nearing the end of our time, I want to spend the rest of our time just talking a little bit about your experience at this company as an engineer. If you have any takeaways, like career-wise.

To give you some personal context, I've worked at a variety of companies. I worked at a number of different companies before I started Software Engineering Daily and all of these companies I didn't really have a great time for various reasons. There was nothing that was even comparable to kind of what you dealt with at Yubl and it just sounds like all of these work — I just imagine someone essentially painting the Sistine Chapel. You put so much work into re-engineering this thing and then somebody just like pulls the funding and basically says, "You know what? Screw this product. It's all disappearing." What are your takeaways from this as an engineer?

[0:47:48.3] YC: It's obviously very disheartening and very disappointing to see all of your effort, all of your hard work, all of your — I guess, the sleepless nights eventually just go away. At the same time, the experience I had there was probably the most intense and the most exciting I had in my entire career. I've met some really good people there, people that I still keep in touch now and we still meet every now and then for lunch and dinner and talk about what's happening, what are the new things that we have learned about Lambda or just how to do better software engineering. None of that — No one can take away from me and from any of us, the things that we've learned.

The things that has helped us with our respective careers. Sure, it mayhem at the time, but at the same time also learned a lot. I think [inaudible 0:48:46.7] that you learn the most when you put yourself in uncomfortable situations. I think Yubl, as crazy as it was, I was lucky enough to have not experienced some of the early craziness that other people may have had to deal with. When I was there, what I experienced was a challenge to do something really interesting, a lot of autonomy and also the chance to work with some very good people that I now call friends. As a person, I don't regret that experience one bit.

[0:49:25.1] JM: Do you regret not having a better perspective of what was going on financially, or did you understand what was going on? Did you understand kind of the risks of where the company was cap-table-wise and investment-cycle-wise?

[0:49:39.5] YC: I understand some of the risk it being a very early startup, it being funded by a small number of investors and it having essentially one major investor that is able to decide the life or death of the company. At the same time, the CTO, the second CTO that took over the company, he paint me a picture of some very interesting challenges. Unfortunately, we were just on the cusp of tiding up and making sure that the fundamental, the basic, the foundations for what we're doing is there and we were starting to get on that next phase of, I guess, our evolution when we started looking at using machine learning and AI to improve recommendations, to improve how we delivered contents and newsfeeds to our users. All of that were going to come and they were part of the package when I signed up. That interesting challenge that is to come once we have dealt with the basics.

I had an inkling of what's happening behind scenes in terms of all the negotiations happening, with the investors. I didn't realize how, I guess, the predicament that we were and I don't think anyone knew apart from, I guess, the people that's involved in negotiations. Even the CTO was under the impression that everything was going to happen. Everything was going to be fine, that we just needed to — I guess we just needed to have the money in our accounts. Unfortunately, that didn't happen and everything just fell apart very very quickly.

[0:51:17.2] JM: Yeah. One other area that I'd like to explore with you, after going deep on AWS serverless technologies and also playing with some Google stuff, what do you think about AWS versus Google, versus Azure, versus whatever other cloud service providers you've been playing around with? What are your predictions and what are your assessments of the current state of technologies?

[0:51:46.6] YC: In terms of comparison, I haven't done much with Azure. Perhaps I've done some bit work at Google in the past both with, I guess, app engine, Google BigQuery. My current view of, I guess, the big three cloud provider is that Amazon has got, I guess, by far the biggest, the market share and pretty much all of my cloud experience has been with Amazon and even though I can find comparable services on any of the other clouds, a lot of the operational knowledge to understand the caveat of different services, that would take time to translate if I want to more work with Google cloud. From where I can see right now, Amazon is unfortunately lagging behind it a bit in terms of the data solutions that they offer. I've been using Google BigQuery for I guess coming up to five years now, and only last year, Amazon announced something equivalent to Google BigQuery with Athena.

What I've seen so far, Athena looks quite interesting solution, but Google also has quite a few other, I guess, different database solutions for transactional systems as well. That's something that Amazon seemed to be lagging behind. Done it with DB, it was great and has been good. Has been pretty good for a long time but is no starting to show is age compared to some of the offers by Google certainly.

In terms of predictions, I don't have any, but I did know that in terms of understand landscape and playing that strategic game, Amazon is above everybody else as far as I can see. Just that move into Lambda is something that I think a lot of other companies by surprise. Sure, pretty soon after Amazon announced Lambda, Google and Azure started to announce some equivalent services, but the fact that Amazon had the foresight to see that opportunity to really step into and be the market leader in this new space tells you something about the strategic thinking behind Amazon.

I think as far as I can see in the serverless space, Lambda has by far the most, I guess, user base and also most of the people that are talking about serverless technologies are mostly about Lambda as well.

[0:54:26.2] JM: Yeah, no kidding. Yeah. A lot of interesting stuff you said there. We've been talking for a while. I really enjoyed the conversation and so many interesting lessons. I know you've been writing about serverless. It looks like you've been kind of consulting on serverless stuff. You've been, maybe, evangelizing it and speaking. Where are you going next? What are your kind of plans with where you're going to take — because I think you're starting to build something of a name for yourself as like at least a serverless refactoring expert. Do you have any plans?

[0:54:59.4] YC: Right now I've been working with O'Reilly on production online course for Safari books online. The first I'm going to run it is on the September 11th and 12th. It's a two-day course with three hour each where we will cover pretty much everything I've learned about serverless both from the operational side of things as well as the development. Also, one of the things I haven't really read much about, but I'll be spending quite a bit of time exploring is the security aspect of serverless and how that differs or how much it's still the same compared to where we build services that run on servers.

One of the key things, I guess, I haven't really talked about but has been on my mind even from the start is that even though we are now billing services on serverless platforms, architecturally, what was good about microservices running on servers, many of them they still apply to us now. Many of them are indeed microservices challenges. Even as we step into this world of serverless of architectures, we still want to strive for architectural parity with services, microservices that run on servers. A lot of the things that I've been writing about is around how do we adjust some of the gaps that we currently have.

[0:56:25.9] JM: Indeed. Yan, I look forward to seeing your material in the future and I will certainly link to the serverless O'Reilly courses in the show notes. Thanks again for coming on Software Engineering Daily. It's been a real pleasure.

[0:56:39.1] YC: It's been a pleasure. Thanks for having me.

[SPONSOR MESSAGE]

[0:56:43.0] 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.io/sedaily, you can find out how to bring continuous delivery to your teams. Say goodbye to deployment panic and hello to consistent, predictable deliveries. Visit gocd.io/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]