A Second Conversation with Werner Vogels

**THE AMAZON
CTO SITS WITH
TOM KILLALEA
TO DISCUSS
DESIGNING
FOR EVOLUTION
AT SCALE.**

When I joined Amazon in 1998, the company had a single US-based website selling only books and running a monolithic C application on five servers, a handful of Berkeley DBs for key/value data, and a relational database. That database was called “ACB” which stood for “Amazon. Com Books,” a name that failed to reflect the range of our ambition. In 2006 *acmqueue* published a conversation between Jim Gray and Werner Vogels, Amazon’s CTO, in which Werner explained that Amazon should be viewed not just as an online bookstore but as a technology company. In the intervening 14 years, Amazon’s distributed systems, and the patterns used to build and operate them, have grown in influence. In this follow-up conversation, Werner and I pay particular attention to the lessons to be learned from the evolution of a single distributed system, S3, which was publicly launched close to the time of that 2006 conversation.

Tom Killalea In your keynote at the AWS re:Invent conference in December 2019, you said that in March 2006 when it launched, S3 [Simple Storage Service] was made

up of eight services, and by 2019 it was up to 262 services. As I sat there I thought that's a breathtaking number,

Principles of Distributed System Design

Amazon used the following principles of

distributed system design to meet Amazon S3 requirements:

Decentralization. Use fully decentralized techniques to remove scaling bottlenecks and single points of failure.

Asynchrony. The system makes progress under all circumstances.

Autonomy. The system is designed such that individual components can make decisions based on local information.

Local responsibility. Each individual component is responsible for achieving its consistency; this is never the burden of its peers.

Controlled concurrency. Operations are designed such that no or limited concurrency control is required.

and it struck me that very little has been written about how a large-scale, always-on service evolves over a very extended period of time. That's a journey that would be of great interest to our software practitioner community. This is evolution at a scale that is unseen and certainly hasn't been broadly discussed.

Werner Vogels I absolutely agree that this is unparalleled scale. Even today, even though there are Internet services these days that have reached incredible scale—I mean look at Zoom, for example [this interview took place over Zoom]—I think S3 is still two or three generations ahead of that. And why? Because we started earlier; it's just a matter of time, and at the same time having a strict feedback loop with your customers that continuously evolves the service. Believe me, when we were designing it, when we were building it, I don't think that anyone anticipated the complexity of it...eventually. I think what we did realize is that we would not be running the same architecture six months later, or a year later.

So, I think one of the tenets up front was don't lock yourself into your architecture, because two or three orders of magnitude of scale and you will have to rethink it. Some

Failure tolerant. The system considers the failure of components to be a normal mode of operation and continues operation with no or minimal interruption.

Controlled parallelism.

Abstractions used in the system are of such granularity that parallelism can be used to improve performance and robustness of recovery or the introduction of new nodes.

Decompose into small, well-understood building blocks.

Do not try to provide a single service that does everything for everyone, but instead build small components that can be used as building blocks for other services.

Symmetry. Nodes in the system are identical in terms of functionality, and require no or minimal node-specific configuration to function.

Simplicity. The system should be made as simple as possible, but no simpler.

of the things we did early on in thinking hard about what an evolvable architecture would be—something that we could build on in the future when we would be adding functionality to S3—were revolutionary. We had never done that before.

Even with Amazon the Retailer, we had unique capabilities that we wanted to deliver, but we were always quite certain where we wanted to go. With S3, nobody had done that before, and remember when we were in the room designing it, [AWS Distinguished Engineer] Al Vermeulen put a number on the board: the number of objects that we would be storing within six months.

TK I remember this conversation.

WV We put two additional zeroes at the end of it, just to be safe. We blew through it in the first two months.

A few things around S3 were unique. We launched with a set of ten distributed systems tenets in the press release. (See sidebar, “Principles of Distributed System Design.”)

That was quite unique, building a service that was fundamentally sound such that you could evolve on top of it. I think we surprised ourselves a bit.

The eight services were really just the fundamental pieces to get, put, and manage incoming traffic. Most importantly, there

From the March 14, 2006, press release for the S3 [Simple Storage Service] launch²

are so many different tenets that come with S3, but durability, of course, trumps everything. The eleven 9s (99.999999999%) that we promise our customers by replicating over three availability zones was unique. Most of our customers, if they have on-premises systems—if they're lucky—can store two objects in the same data center, which gives them four 9s. If they're really good, they may have two data centers and actually know how to replicate over two data centers, and that gives them five 9s. But eleven 9's, in terms of durability, is just unparalleled. And it trumps everything. The need for durability also means that for example, one of the eight microservices would be the one that continuously checks all the objects, all the CRCs (cyclic redundancy checks), and there are trillions and trillions of objects by now. There's a worker going around continuously checking in case an object had some bit rot or something like that.

One of the biggest things that we learned early on is—and there's this quote that I use—"Everything fails, all the time." Really, everything fails, all the time, in unexpected ways, things that I never knew. Bit flips in memory, yes. You need to protect individual data structures with a CRC or checksum on it because you can't trust the data in it anymore. TCP (Transmission Control Protocol) is supposed to be reliable and not have any flips in bits, but it turns out that's not the case.

TK Launching with distributed-systems tenets was unique. Fourteen years later, would the tenets be different? There's this expectation that tenets should be evergreen; would there be material changes?

WV Not these; these are truly fundamental concepts that

we use in distributed systems. The 10 tenets were separate from S3, stating that this is how you would want to build distributed systems at scale. We just demonstrated that S3 is a really good example of applying those skills.

Some of the other tech companies that were scaling at the same time, search engines and so on, in general had only one task, such as do search really well. In the case of Amazon the Retailer, we had to do everything: robotics, machine learning, high-volume transaction processing, rock-solid delivery of web pages, you name it. There isn't a technology in a computer science textbook that wasn't pushed to the edge at Amazon.com. We were operating at unparalleled scale, with really great engineers—but they were practical engineers—and we made a change before building S3 to go back to fundamentals, to make sure that what we were building was fundamentally sound because we had no idea what it was going to look like in a year. For that we needed to have a really solid foundation.

TK One of the keys to the success of S3 was that, at launch, it was as simple as it possibly could be, offering little more than `GetObject` and `PutObject`. At the time that was quite controversial, as the offering seemed almost too bare bones. With the benefit of hindsight, how do you reflect on that controversy, and how has that set up S3 to evolve since then? You mentioned *evolvable architecture*.

WV `List` is the other one that goes with `Get` and `Put`, prefixed `List`.

TK Right. Could it possibly have been simpler at launch?

WV It was slightly controversial, because most technology companies at the time were delivering everything and the kitchen sink, and it would come with a very thick book

We went down a path, one that Jeff [Bezos] described years before, as building tools instead of platforms.

and ten different partners that would tell you how to use the technology. We went down a path, one that Jeff [Bezos] described years before, as building *tools* instead of *platforms*. A platform was the old-style way that large software platform companies would use in serving their technology.

If you would go from Win32 to .NET, it was clear that the vendor would tell you exactly how to do it, and it would come with everything and the kitchen sink—not small building blocks but rather, “This is how you should build software.”

A little before we started S3, we began to realize that what we were doing might radically change the way that software was being built and services were being used. But we had no idea how that would evolve, so it was more important to build small, nimble tools that customers could build on (or we could build on ourselves) instead of having everything and the kitchen sink ready at that particular moment. It was not necessarily a timing issue; it was much more that we were convinced that whatever we would be adding to the interfaces of S3, to the functionality of S3, should be driven by our customers—and how the next generation of customers would start building their systems.

If you build everything and the kitchen sink as one big platform, you build with technology that is from five years before, because that’s how long it takes to design and build and give everything to your customers. We wanted to move much faster and have a really quick feedback cycle with our customers that asks, “How would you develop for 2025?”

Development has changed radically in the past five to ten years. We needed to build the right tools to support that rate of radical change in how you build software. And with that, you can't predict; you have to work with your customers, to wait to see how they are using your tools—especially if these are tools that have never been built before—and see what they do. So, we sat down and asked, “What is the minimum set?”

There's one other thing that I want to point out. One of the big differences between Amazon the Retailer and AWS in terms of technology is that in retail, you can experiment the hell out of things, and if customers don't like it, you can turn it off. In AWS you can't do that. Customers are going to build their businesses on top of you, and you can't just pull the plug on something because you don't like it anymore or think that something else is better.

You have to be really consciously careful about API design. APIs are forever. Once you put the API out there, maybe you can version it, but you can't take it away from your customers once you've built it like this. Being conservative and minimalistic in your API design helps you build fundamental tools on which you may be able to add more functionality, or which partners can build layers on top of, or where you can start putting different building blocks together. That was the idea from the beginning: to be so minimalistic that we could allow our customers to drive what's going to happen next instead of us sitting in the back room thinking, “This is what the world should look like.”

TK The idea of being minimalistic in defining an MVP (minimum viable product) has gained broad adoption now,

but S3 at launch pushed it to the extreme. In those early days there was some discussion around which persistence service the AWS team should bring to market first: an object store or a key-value store or a block store. There was a sense that eventually each would be out there, but there's a necessary sequencing in a small team. Launching S3 first was done very intentionally, with EBS (Elastic Block Store), for example, following in August 2008. Can you share with us the rationale?

WV Quite a bit of that is learning from how we'd built systems ourselves, where a key-value store was the most crucial. After our "mishap" with one of our database vendors in December 2004, we decided to take a deep look at how we were using storage, and it turned out that 70 percent of our usage of storage was key-value. Some of those values were large, and some were really small. One of those drove in the direction of Dynamo, in terms of small keys, a table interface, things like that, and the other one became S3, with S3 more as a blob and bigger value store, with some different attributes.

One of the big winners in the early days of S3 was direct HTTP access to your objects. That was such a winner for everyone because now suddenly on every web page, app, or whatever, you could pull your object in just by using HTTP. That was unheard of. Maybe there were things at the beginning that we thought would be more popular and that didn't turn out to be the case—for example, the BitTorrent interface. Did it get used? Yes, it did get used. But did it get used massively? No. But we launched FTP access, and that was something that people really wanted to have.

So, sometimes it seems like not very sexy things make

A complex system that works is invariably found to have evolved from a simple system that worked.

it, but that's really what our customers are used to using. Again, you build a minimalistic interface, and you can build it in a robust and solid manner, in a way that would be much harder if you started adding complexity from day one, even though you know you're adding something that customers want.

There were things that we didn't know on day one, but a better example here is when we launched DynamoDB and took a similar minimalistic approach. We knew on the day of the launch that customers already wanted secondary indices, but we decided to launch without it. It turned out that customers came back saying that they wanted IAM (Identity and Access Management)—access control on individual fields within the database—much more than they wanted secondary indices. Our approach allows us to reorient the roadmap and figure out the most important things for our customers. In the DynamoDB case it turned out to be very different from what we thought.

TK I think that much of this conversation is going to be about evolvability. As I listened to you at re:Invent, my mind turned to Gall's Law: "A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system." How do you think this applies to how S3 has evolved?

WV That was the fundamental thinking behind S3. Could we have built a complex system? Probably. But if you build a complex system, it's much harder to evolve and change, because you make a lot of long-term decisions in a complex system. It doesn't hurt that much if you make long-term

decisions on very simple interfaces, because you can build on top of them. Complex systems are much harder to evolve.

Let me give you an example. One of the services added to S3 was auditing capability—auditing for whether your objects are still fresh and alive and not touched, or whatever. That was the first version of auditing that we did. Then we started to develop CloudTrail (launched in November 2013), which had to be integrated into S3. If you've built a complex system with all of these things in a monolith or maybe in five monoliths, that integration would be a nightmare, and it would definitely not result in a design that you are comfortable with evolving over time.

Mai-Lan Tomsen Bukovec [vice president, AWS Storage] has talked about a culture of durability. For example, within S3, durability trumps everything, even availability. Imagine if the service were to go down: You cannot lose the objects. Your data cannot disappear; maybe it takes you five minutes to get access to it again, but your objects should always be there. Mai-Lan's team has a culture of durability, which means that they use tools such as TLA+ to evaluate their code to see whether its algorithms are doing exactly what they're supposed to be doing.

Now let's say, to make it simple, you have a 2,000-line algorithm. That's something you can evaluate with formal verification tools; with 50,000 lines, forget about it. Simple building blocks allow you to have a culture that focuses exactly on what you want to do, whether it's around auditing or whether it's around using TLA+ or durability reviews or whatever. Everything we change in S3 goes through a durability review, making sure that none of

these algorithms actually does anything other than what we want them to do.

TK With full-blown formal verification?

WV Here's a good example in the context of S3. If you look at libssl, it has a ridiculous number of lines of code, with something like 70,000 of those involved in processing TLS. If you want to create vulnerabilities, write hundreds of thousands of lines of code. It's one of the most vulnerable access points in our systems.

TK I know that there's a plug for S2N coming.

WV Yes, so we wrote S2N, which stands for signal-to-noise, in 5,000 lines. Formal verification of these 5,000 lines can tell exactly what it does. Now everything on S3 runs on S2N, because we have way more confidence in that library—not just because we built it ourselves but because we use all of these additional techniques to make sure we can protect our customers. There is end-to-end encryption over every transfer we do. In how you use encrypted storage, do you want us to create the keys? Do you want to bring the keys and give them to us? Do you want to bring the keys and put them in a KMS (key management service)? Or do you want to completely manage your keys? I'm pretty sure that we started off with one, and customers started saying, "But that, and that, and that." You need those too.

If you build this as an evolvable architecture with small microservices, you can still allow encryption at rest just to do its job, and then you can think about how to start adding other services that may do other things—like life-cycle management from S3 down to Glacier. If this object hasn't been touched in 30 days, move it to reduced instance storage; and if it then hasn't been touched for another two

months, automatically move it to Glacier.

TK You launched S3 Object Versioning in February 2010. How did that fit into the evolving expectations of the service, how customers wanted to use it, and the architectural demands that posed?

WV It was mostly a back-and-forth with our customer base about what would be the best interface—really listening to people’s requirements. And to be honest, immutability was a much bigger requirement than having a distributed lock manager, which is notoriously hard to build and operate. It requires a lot of coordination among different partners, and failure modes are not always well understood.

So, we decided to go for a simpler solution: object versioning, officially called S3 Object Lock. There are two things that you can do to a locked object. First, once you create it you can only change it, which in the world of blockchain and things like that is a very interesting concept. You can also set two attributes on it: one is the retention period (e.g., this cannot be deleted for the coming 30 days); and another is LegalHold, which is independent of retention periods and basically says that this object cannot be deleted until an authorized user explicitly takes an action on it.

It turns out that object versioning is extremely important in the context of regulatory requirements. You may need to be able to tell your hospital or regulatory examiners that this object is being kept in live storage for the coming six months, and then it is moved to cold storage for the 30 years after. But being able to prove to the regulator that you are actually using technology that will still be alive in 30 years is quite a challenge, and as such

we've built all of these additional capabilities in there.

TK The absence of traditional explicit locking has shifted responsibility to developers to work around that in their code, or to use versioning. That was a very intentional decision.

WV It was one of these techniques that we used automatically in the '80s and '90s and maybe in the early 2000s—the distributed lock managers that came with databases and things like that. You might have been using a relational database because that was the only tool you had, and it came with transactions, so you used transactions, whether you needed them or not. We wanted to think differently about an object store, about its requirements; and it turns out that our approach gave customers the right tools to do what they wanted to do, unless they really wanted lock and unlock, but that's not something that can scale easily, and it's hard for our customers to understand. We went this different way, and I've not heard a lot of complaints from customers.

TK S3 launched more than four years before the term *data lake* was first used in a late 2010 blog post by James Dixon.⁵ S3 is used by many enterprise data lakes today. If they had known what was coming, would it have been helpful or distracting for the 2006 S3 team to try to anticipate the needs of these data lakes?

WV We did a number of things in the early days of AWS in general—it has nothing to do necessarily with S3—where there are a few regrets. For example, I am never, ever going to combine account and identity at the same time again. This was something we did in the early days; we didn't really think that through with respect to how the

system would evolve. It took us quite a while actually to rip out accounts. An account is something you bill to; identity is something you use in building your systems. These are two very different things, but we didn't separate them in the early days; we had one concept there. It was an obvious choice in the moment but the wrong choice.

Here is another interesting example with S3. It's probably the only time we changed our pricing strategy. When we launched S3, we were charging only for data transfer and data storage. It turned out that we had quite a few customers who were storing millions and millions of thumbnails of products they were selling on eBay. There was not much storage because these thumbnails were really small, and there wasn't much data transfer either, but there were enormous numbers of requests. It made us learn, for example, when you design interfaces, and definitely those you charge for, you want to charge for what is driving your own cost. One of the costs that we didn't anticipate was the number of requests, and request handling. We added this later to the payment model in S3, but it was clearly something we didn't anticipate. Services that came after S3 have been able to learn the lessons from S3 itself.

Going back to the concept of a data lake, I wouldn't do anything else actually, except for those two things, mostly because I think we've created the basic building blocks since S3 serves so much more than data lakes. There are a few interesting parts, in terms of the concepts in data lakes, where I think S3 works well under the covers, but you need many more components to build a data lake. In terms of building a data lake, for example, Glue is an

equally important service that sits next to S3, discovers all of your data, manages your data, lets you decide who has access to which data, and whether you need to pull this from on-premises, or does it need to come out of a relational database, and all of these kinds of things.

It turns out that you need a whole lot of components if you really want to build a mature data lake. It's not just storing things in S3. That's why we built Lake Formation. One of the things you see happening both at AWS and with our partners is that now that we have this massive toolbox—175 different services—they've always been intended as small building blocks. This makes them sometimes hard to use because they're not really solutions, they're basic building blocks. So, to build a data lake, you need to put a whole bunch of these things together. What you see now is that people are building solutions to give you a data lake.

We have to remember that S3 is used for so much more than that, whether it's a content lake with massive video and audio files, or a data lake where people are storing small files, or maybe it's a data lake where people are doing genomics computation over it. One of our customers is sequencing 100 million human genomes. One human genome is 100 GB; that's just raw data—there's nothing else there, so a lot of things have to happen to it. In life sciences, files are getting huge. If I look at the past, or if I look at that set of customers that start to collect that set of data, whether it's structured or unstructured data, quite a few of them are starting to figure out how to apply machine learning to their data. They're not really there yet, but they may want to have the data stored there and start

making some use of Redshift or EMR or Kinesis or some other tools to do more traditional approaches to analytics. Then they might be prepared for a year from now when they've trained their engineers and are ready to apply machine learning to these data sets.

These data sets are getting so large—and I'm talking here about petabytes or hundreds of petabytes in a single data file—and requirements are shifting over time. When we designed S3, one of its powerful concepts was separating compute and storage. You can store the hell out of everything, but your compute doesn't need to scale with it, and can be quite nimble in terms of compute. If you store more, you don't need more EC2 (Elastic Compute Cloud) instances.

With data sets becoming larger and larger, it becomes more interesting to see what can be done inside S3 by bringing compute closer to the data for relatively simple operations. For example, we saw customers retrieving tens if not hundreds of petabytes from their S3 storage, then doing a filter on it and maybe using five percent of the data in their analytics. That's becoming an important concept, so we built S3 Select, which basically does the filter for you at the lowest level and then moves only the data that you really want to operate on.

Similarly, other things happened in our environment that allowed us to extend S3. In the Lambda and Serverless components, the first thing we did was fire up a Lambda function when a file arrives in S3. The ability to do event-driven triggering and extend S3 with your own functions and capabilities without having to run EC2 instances made it even more powerful, because it's not just our code

that runs there, it's your code. There's a whole range of examples where we go under the covers of S3 to run some compute for you in case you want that.

TK This concept of extensibility is really key in terms of the lessons that our readers could take away from this journey. I know there were some examples starting with bare bones in the case of S3 and learning from the requests of a few very early and demanding adopters such as Don MacAskill at SmugMug and Adrian Cockcroft, who at the time was at Netflix. Are there other examples of situations where customer requests made you pop open your eyes and say, "That's interesting; I didn't see that coming," and it became a key part of the journey?

WV There are other examples around massive high-scale data access. To get the performance they needed out of S3, some customers realized that they had to do some randomization in the names. They would pre-partition their data to get the performance they were looking for, especially the very high-volume access people.

It's now been three years since that we made significant changes in how partitioning happens in S3, so that this process is no longer needed. If customers don't do pre-partitioning themselves, we now have the opportunity to do partitioning for them through observability. We observe what's happening and may do very quick rereplication, or repartitioning, to get the right performance to our customers who in the past had to figure it out by themselves.

With our earliest customers, we looked at that particular behavior and realized we had to fix this for them. Indeed, people like Don [MacAskill] have been very vocal

but also very smart technologists. Such developers knew exactly what they wanted to build for their businesses, and I think we thrived on their feedback.

Today it may be, for example, telemedicine, which needs HIPAA (Health Insurance Portability and Accountability Act) compliance and other regulatory requirements; it needs to be sure about how the data is stored and so on. We've started to build on this so that we could easily use a microservices architecture to test for auditors whether we are meeting HIPAA or PCI DSS (Payment Card Industry Data Security Standard) or another compliance specification in realtime.

Amazon Macie, for example, is one of these services. The capabilities sit in S3 to review all of your data, discover which is personally identifiable information, intellectual property, or other things, and we can use machine learning to discover this. Why? Every customer is different. It's not just discovering what they tell you they have; it's discovering the access patterns to the data, and when we see a change in the access patterns to your data that may signal a bad actor coming in. We could build all of these things because we have this microservices architecture; otherwise, it would be a nightmare.

TK In a 2006 conversation for *acmqueue* with Jim Gray, you talked about how the team “is completely responsible for the service—from scoping out the functionality, to architecting it, to building it, and operating it. You build it, you run it. This brings developers into contact with the day-to-day operation of their software.”⁶ I know that you remember that conversation fondly. That continues to be among our most widely read articles even today. There's

You
build it,
you
run it.

universal relevance in so many of the concepts that came up in it.

WV That conversation with Jim was great. It wasn't so much about AWS. It was much more about retail, about experimentation, and making sure that your engineers who are building customer-facing technology are not sitting in the back room and handing it off to someone else, who is then in contact with the customers. If your number-one leadership principle is to be customer obsessed, then you need to have that come back into your operations as well. We want everybody to be customer obsessed, and for that you need to be in contact with customers. I do also think, and I always joke about it, that if your alarm goes off at 4 am, there's more motivation to fix your systems. But it allows us to be agile, and to be really fast.

I remember during an earlier period in Amazon Retail, we had a whole year where we focused on performance, especially at the 99.9 percentile, and we had a whole year where we focused on removing single points of failure, but then we had a whole year where we focused on efficiency. Well, that last one failed completely, because it's not a customer-facing opportunity. Our engineers are very well attuned to removing single points of failure because it's good for our customers, or to performance, and understanding our customers. Becoming more efficient is bottom-line driven, and all of the engineers go, "Yes, but we could be doing all of these other things that would be good for our customers."

TK Right. That was a tough program to lead.

WV That's the engineering culture you want. Of course, you also don't want to spend too much money, but... I remember

that bringing product search from 32 bits to 64 bits immediately resulted in needing only a third of the capacity, but, most importantly, Amazon engineers are attuned to what's important to our customers. That comes back to our technology operational model as well; of course, DevOps didn't exist before that. All of these things came after that.

Probably one of the reasons that that *acmqueue* article is popular is because it was one of the first times we talked about this. The reaction was similar when we wrote the Dynamo paper.⁴ The motivation for writing it was not really to promote Amazon but to let engineers know what an amazing environment we had to build the world's largest scalable distributed systems, and this was even before AWS. One of my hardest challenges, yours as well in those days, was hiring. You couldn't hire engineers because, "Why, you're a [expletive] bookshop!" I know from myself as an academic, I almost wouldn't give a talk at Amazon. Why? "A database and a web server, how hard can it be?"

It wasn't until we started talking about these kinds of things publicly that the tide started to shift in our ability not only to hire more senior engineers, but also to have people excited about it: "If you want to build really big stuff, you go to Amazon." I think now with AWS, it's much easier; everybody understands that. But in those days, it was much harder.

TK That initial S3 team was a single agile team in the canonical sense, and in fact was quite a trailblazer when it came to agile adoption across Amazon...

WV Yes.

TK ...and the "You build it, you run it" philosophy that you discussed with Jim applied to the developers on the S3

team in 2006. They all knew each of those initial eight services intimately and would have been in a position to take a first pass at debugging any issue. As the complexity of a system increases, it becomes harder for any individual engineer to have a full and accurate model of that system. Now that S3 is not a single team but a big organization, how can an engineer reason with and model the whole?

WV As always, there's technology, there's culture, and there's organization. The Amazon culture is well known, technology we don't need to talk that much about, so it's all about organization. Think about the kinds of things we did early on at Amazon, with the creation of the culture around principal engineers. These are people who span more than one service, people who have a bigger view on this, who are responsible for architecture coherence—not that they tell other people what to do, but at least they have the knowledge.

If you have a team in S3 that is responsible for S3 Select, that is what they do. That's what you want. They may need to have deep insight in storage technology, or in other technologies like that, or even in the evolution of storage technologies over time, because the way that we were storing objects in 2006 is not the same way we're storing objects now. But we haven't copied everything; you can't suddenly start copying exabytes of data because you feel that some new technology may be easier to use. There is some gravity with the decisions that you make as well, especially when it comes to storage.

Principal engineers, distinguished engineers—these are roles that have evolved over time. When I joined, we didn't have a distinguished engineer. Over time we started hiring

more senior leaders, purely with the goal not necessarily of coding on a day-to-day basis, but sort of being the advisor to these teams, to multiple teams. You can't expect the S3 Select team to have sufficient insight into exactly what the auditing capabilities of Macie are, but you do need to have people in your organization who are allowed to roam more freely on top of this.

Our decentralized nature makes it easy to move fast within the particular area of responsibility of your team; the downside of decentralization is coordination. Now suddenly, you need to invest in coordination because these teams are small and nimble and agile and fast-moving, and they don't have additional people to help with coordination.

In the past at Amazon we had a few of these cases; when Digital (e.g., Kindle or Amazon Video) wanted to add something to the order pipeline, a physical delivery address was required. There was no way around it. They would walk to the 80 different ordering teams and say, "We need to change this." The ordering teams would respond that they hadn't budgeted for it. One of the consequences was we allowed duplication to happen. We allowed the Digital team to build their own order pipeline for speed of execution. There are advantages; otherwise, we wouldn't be doing it. There are disadvantages as well.

Sharing knowledge, principal engineers and distinguished engineers help with this. But sometimes people go to the wiki and read about your old API, not knowing that it's the old API, and then start hammering your service through the old API that you thought you had deprecated.

Information sharing, coordination, oversight, knowing

what else is going on and what are the best practices that some of the other teams have developed—at our scale, these things become a challenge, and as such you need to change your organization, and you need to hire into the organization people who are really good at....I won't say oversight because that implies that they have decision control, but let's say they are the teachers.

TK Approachability is a key characteristic for a principal engineer. Even if junior engineers go through a design review and learn that their design is terrible, they should still come away confident that they could go back to that same principal engineer once they believe that their reworked design is ready for another look.

WV Yes.

TK Could you talk about partitioning of responsibility that enables evolution across team boundaries, where to draw the boundaries and to scope appropriately?

WV I still think there are two other angles to this topic of information sharing that we've always done well at Amazon. One, which predates AWS of course, is getting everybody in the room to review operational metrics, or to review the business metrics. The database teams may get together on a Tuesday morning to review everything that is going on in their services, and they'll show up on Wednesday morning at the big AWS meeting where, now that we have 175 services not every service presents anymore, but a die is being rolled.

TK Actually I believe that a wheel is being spun.

WV Yes. So, you need to be ready to talk about your operational results of the past week. An important part of that is there are senior people in the room, and there

are junior folks who have just launched their first service. A lot of learning goes on in those two hours in that room that is probably the highest-value learning I've ever seen. The same goes for the business meeting, whether it's Andy [Jassy, CEO of AWS] in the room, or Jeff [Bezos, CEO of Amazon] or [Jeff] Wilke [CEO of Amazon Worldwide Consumer]; there's a lot of learning going on at a business level. Why did S3 delete this much data? Well, it turns out that this file-storage service that serves to others deletes only once a month, and prior to that they mark all of their objects (for deletion).

You need to know; you need to understand; and you need to talk to others in the room and share this knowledge. These operational review and business review meetings have become extremely valuable as an educational tool where the most senior engineers can shine in showing this is how you build and operate a scalable service.

TK Fourteen years is such a long time in the life of a large-scale service with a continuously evolving architecture. Are there universal lessons that you could share for service owners who are much earlier in their journey? On S3's journey from 8 to 262 services over 14 years, any other learnings that would benefit our readers?

WV As always, security is number one. Otherwise, you have no business. Whatever you build, with whatever you architect, you should start with security. You cannot have a customer-facing service, or even an internally facing service, without making that your number-one priority.

Finally, I have advice that is twofold. There's the Well-Architected Framework, where basically over five different pillars we've collected 10 or 15 years of knowledge from

our customers as to what are the best practices.³ The Well-Architected Framework deals with operations, security, reliability, performance, and cost. For each of those pillars, you get 100 questions that you should be able to answer yourself for whatever you're building. For example, "Are you planning to do key rotation?" Originally our solutions architects would do this review for you, but with millions of customers, that doesn't really scale. We've built a tool for our customers so that they can do it in the console. Not only that, but they can also see across multiple of their projects what may be common deficiencies in each and every one of those projects. If you're not doing key rotation in any of them, maybe you need to put a policy in place or educate your engineers.

Second, and this is much more developer-oriented, is the Amazon Builders' Library.¹ That's a set of documents about how we built stuff at Amazon. One of the most important ones is cell-based architecture: How do you partition your service so that the blast radius is as minimal as possible? How do you do load shedding? All of these things that we struggled with at Amazon the Retailer—and came up with good solutions for—we now make available for everybody to look at.

TK Thank you Werner; it's been wonderful to catch up with you.

WV Tom, it's been a pleasure talking to you.

References

1. Amazon Builder's Library; <https://aws.amazon.com/builders-library/>.
2. Amazon Press Center. 2006. Amazon Web Services

- launches; <https://press.aboutamazon.com/news-releases/news-release-details/amazon-web-services-launches-amazon-s3-simple-storage-service>.
3. AWS Well-Architected; <https://aws.amazon.com/architecture/well-architected/>.
 4. DeCandia, G., et al. 2007. Dynamo: Amazon's highly available key-value store. *Proceedings of the 21st Annual Symposium on Operating Systems Principles*. (October) 205-220; <https://dl.acm.org/doi/10.1145/1294261.1294281>.
 5. Dixon, J. 2010. Pentaho, Hadoop, and data lakes; <https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/>.
 6. Gray, J. 2006. A conversation with Werner Vogels. *acmqueue* 4{4}; <https://queue.acm.org/detail.cfm?id=1142065>.

Copyright © 2020 held by owner/author. Publication rights licensed to ACM.

SHAPE THE FUTURE OF COMPUTING!

Join ACM today at acm.org/join

**BE CREATIVE. STAY CONNECTED.
KEEP INVENTING.**

Association for
Computing Machinery

Advancing Computing as a Science & Profession