MICROSERVICES PATTERNS AND PRACTICES

eMag Issue 59 - Mar 2018

InfoQ

ARTICLE
Polyglot Persistence Powering Microservices

ARTICLE
Microservices Patterns and Practices Panel

ARTICLE
Managing Data in Microservices

IN THIS ISSUE

A LETTER FROM THE EDITOR

Thomas Betts

While the underlying technology and patterns are certainly interesting, microservices have always been about helping development teams be more productive. Whether used as a technique for architects to manage complexity or to make small teams more independent and responsible for supporting the software they create, the human aspect of microservices cannot be ignored.

Many of the experts who spoke about microservices patterns and practices at QCon San Francisco 2017 did not simply talk about the technical details of microservices. They included a focus on the business side and more human-oriented aspects of developing distributed software systems.

At Netflix, the cloud database engineering team is responsible for providing several flavors of data persistence as a service to microservice development teams. Roopa Tangirala explained how her team has created self-service tools that help developers easily implement the appropriate data store for each project's needs.

Drawing on his experience with developing a microservices application at Datawire in 2013, Rafael Schloming argued that one of the most important — although often ignored — questions a development lead should ask is "How do I break up my monolithic process?" as the development process is critical to establishing and maintaining velocity.

With microservices distributed across containers, how is a developer able to step into the code and debug what is happening? Idit Levine discussed the problem and introduced Squash, an open-source platform for debugging microservices applications.

Randy Shoup provided practical examples of how to manage data in microservices, with an emphasis on migrating from a monolithic database. He also strongly advocated for building a monolith first, and only migrating to microservices after you actually require the scaling and other benefits they provide.

The microservices track also included a panel discussion where several experts shared their experiences and advice for being successful with microservices. Questions from the audience highlighted common themes, such as dealing with deployments, communication between microservices, and looking at what future trends might follow microservices.

CONTRIBUTORS

Thomas Betts

is a principal software engineer at IHS Markit, with two decades of professional software development experience. His focus has always been on providing software solutions that delight his customers. He has worked in a variety of industries, including retail, finance, health care, defense and travel. Thomas lives in Denver with his wife and son, and they love hiking and otherwise exploring beautiful Colorado.

Rafael Schloming

is co-founder and chief architect of Datawire. He is a globally recognized expert on messaging and distributed systems and author of the AMQP specification. Previously, Schloming was a principal software engineer at Red Hat. Rafael has a B.S. in computer science from MIT.

Chris Richardson

is a developer and architect. He is a Java Champion and the author of POJOs in Action, which describes how to build enterprise Java applications with frameworks such as Spring and Hibernate. Richardson was also the founder of the original Cloud Foundry, an early Java PaaS for Amazon EC2. He consults with organizations to improve how they develop and deploy applications and is working on his third startup. He’s on Twitter as @crichardson.

Idit Levine
is Founder/Leader/Contributor on a variety of Cloud open source Projects. Expert in cluster management like: Kubernetes, Mesos & DockerSwam. Hybrid cloud: AWS, Google Cloud, OpenStack, Xen & vSphere Comfortable with Cloud Foundry and a laundry list of other frameworks and tools.

Roopa Tangirala
is an experienced engineering leader with extensive background in databases, be they distributed or relational. She leads the Cloud Database Engineering team at Netflix, responsible for cloud persistent run-time stores for Netflix, ensuring data availability, durability, and scalability to meet growing business needs. The team specializes in providing polyglot persistence as a service with Cassandra, Elasticsearch, Dynomite, MySQL, etc.

Louis Ryan
is a core contributor to Istio and gRPC and is a principal engineer at Google.

Randy Shoup
is a 25-year veteran of Silicon Valley, and has worked as a senior technology leader and executive at companies ranging from small startups to mid-sized places to eBay and Google. He is currently VP Engineering at Stitch Fix in San Francisco. He is particularly passionate about the nexus of culture, technology, and organization.

Daniel Bryant
is leading change within organisations and technology. His current work includes enabling agility within organisations by introducing better requirement gathering and planning techniques, focusing on the relevance of architecture within agile development, and facilitating continuous integration/delivery.

Watch presentation online on InfoQ

Polyglot Persistence Powering Microservices

Adapted from a presentation at QCon San Francisco 2017, by Roopa Tangirala, engineering manager at Netflix

KEY TAKEAWAYS

Choose the appropriate persistence store for your microservices.

By providing polyglot persistence as a service, developers can focus on building great applications and not worry about tuning, tweaking, and capacity of various back ends.

Operating various persistence stores at scale involves unique challenges, but common components can simplify the process.

Netflix's common platform drives operational excellence in managing, maintaining, and scaling persistence infrastructures (including building reliable systems on unreliable infrastructure).

We have all worked in companies that started small, and have a monolithic app built as a single unit. That app generates a lot of data for which we pick a data store. Very quickly, the database becomes the lifeline of the company.

Since we are doing such an amazing job, growth picks up and we need to scale the monolithic app. It starts to fail under high load and runs into scaling issues. Now, we must do the right thing. We break our monolithic app into multiple microservices that have better fallback and can scale well horizontally. But we don't worry about the back-end data store; we continue to fit the microservices to the originally chosen back end.

Soon, things become complicated at our back-end tier. Our data team feels overwhelmed because they're the ones who have to manage the up time of our data store. They are trying to support all kinds of antipatterns of which the database might not be capable.

Imagine that instead of trying to make all of our microservices fit one persistence store, we leverage the strengths and features of our back-end data tier to fit our application needs. No longer do we worry about fitting our graph usage into RDBMS or trying to fit ad hoc search queries into Cassandra. Our data team can work peacefully, in a state of Zen.

Polyglot persistence powering microservices

I manage the cloud database engineering team at Netflix. I have been with Netflix for almost a decade and I have seen the company transition from being monolithic in the data center to microservices and polyglot persistence in the cloud. Netflix has embraced polyglot persistence. I will cover five use cases for it, and discuss the reasons for choosing different back-end data stores.

Being a central platform team, my team faces many challenges in providing different flavors of database as a service across all of Netflix's microservice platforms.

About Netflix

Netflix has been leading the way for digital content since 1997. We have over 109 million subscribers in 190 countries and we are a global leader in streaming. Netflix delivers an amazing viewing experience across a wide variety of devices, and brings you great original content in the form of Stranger Things, Narcos, and many more titles.

All your interactions as a Netflix customer with the Netflix UI, all your data such as membership information or viewing history, all of the metadata that a title needs to move from script to screen, and so much more are stored in some form in one of the data stores we manage.

The Cloud Database Engineering (CDE) team at Netflix runs on the Amazon cloud, and we support a wide variety of polyglot persistence. We have Cassandra, Dynomite, EVCache, Elastic, Titan, ZooKeeper, MySQL, Amazon S3 for some datasets, and RDS.

Elasticsearch provides great search, analysis, and visualization of any dataset in any format in near real time. EVCache is a distributed in-memory caching solution based on Memcached that was open-sourced by Netflix in 2011. Cassandra is a distributed NoSQL data store that can handle large datasets and can provide high-availability, multi-region replication, and high scalability. Dynomite is a distributed Dynamo layer, again open-sourced by Netflix, that provides support for different storage engines. Currently, it supports Redis, Memcached, and RocksDB. Inspired by Cassandra, it adds sharding and replication to non-distributed datasets. Lastly, Titan is a scalable graph database that's optimized for storing and querying graph datasets.

Let's look at the architecture, the cloud deployment, and how the datasets are persisted in Amazon Web Services (AWS). We are running in three AWS regions, which take all of the traffic. User traffic is routed to the closest region: primarily, US West 2, US East 1, and EU West 1. If there's a problem with one region, our traffic team can shift the traffic in less than seven minutes to the other two regions with minimal or no downtime. So all of our data stores need to be distributed and highly scalable.

Use case 1: CDN URL

If, like me, you're a fan of Netflix (and love to binge-watch Stranger Things and other titles), you know you have to click the play button. From the moment you click to the time you see the video on the screen, many things happen in the background. Netflix has to look at the user authorization and licensing for the content. Netflix has a network of Open Connect Appliances (OCAs) spread all over the world. These OCAs are where Netflix stores the video bits, and the sole purpose of these appliances is to deliver the bits as quickly and efficiently as possible to your devices while we have an Amazon plane that handles the microservices and data-persistence store. This service is the one responsible for generating the URL, and from there, we can stream the movie to you.

The very first requirement for this service is to be highly available. We don't want any user experience to be compromised when you are trying to watch a movie, say, so high availability was priority number one. Next, we want tiny read and write latencies, less than one millisecond, because this service lies in the middle of the path of streaming, and we want the movie to play for you the moment you click play.

We also want high throughput per node. Although the files are pre-positioned in all of these caches, they can change based on the cache held or when Netflix introduces new movies — there are multiple dimensions along which these movie files can change. So this service receives high read as well as write throughputs. We want something where per-node throughput can be high so we can optimize.

For this particular service we used EVCache. It is a distributed caching solution that provides low latency because it is all in memory. The data model for this use case was simple: it was a simple key value, and you can easily get that data from the cache. EVCache is distributed, and we have multiple copies in different AWS Availability Zones, so we get better fault tolerance as well.

Use case 2: Playback error

Imagine that you click play to watch a movie but you get a playback error. The playback error happens whenever you click the title — it's just not playable.

Titles have multiple characteristics and metadata. It has ratings, the genre, and the description. It has the audio languages and the subtitle languages it supports. It has the Netflix Open Connect CDN URL, discussed in the first use case, which is the location from where the movie streams to you. We call all of this metadata the "playback manifest". And we need it to play the title for you.

There are hundreds of dimensions that can lead to a playback metadata error, and there are hundreds of dimensions that can alter the user's playback experience. For example, some content is licensed only in specific countries and we cannot play that to you if you cross a border. Maybe a user wants to watch Narcos in Spanish. We might have to change the bit rate at which we are streaming the movie depending on your use of Wi-Fi or a fixed network. Some devices do not support 4K or HD and we have to change the stream based on the device. Beyond these few examples, there are hundreds of dimensions on which your playback experience depends.

For this service, we wanted the ability to quickly resolve incidents. We want to have someplace where we can quickly look for the cause of an issue — which dimension is not in sync, which is causing your playback error. If we have ruled out a push, we want to see if we need to roll back, or roll forward, based on the scope of the error: is the error happening in all three regions, in only specific regions, or on only a particular device? There are multiple dimensions which we need to figure out the dataset.

Another requirement was interactive dashboards. We wanted the ability to slice and dice the dataset to see the root cause of that error. Near-real-time search is important because we want to figure out whether or not a recent push has caused the problem at hand. We need ad hoc queries because there are so many dimensions; we don't know our query patterns. There may be multiple ways for us to query the dataset to arrive at what is causing the error.

We used Elasticsearch for this service. It provides great search and analysis for data in any form, and it has interactive dashboards through Kibana. We use Elasticsearch a lot at Netflix, especially for debugging and logging use cases.

Kibana provides a great UI for interactive exploration that allows us to examine the dataset to find the error. We can determine that the error is in a specific region across multiple devices, in a specific device, or confined to a particular title. Elasticsearch also supports queries such as "What are the top 10 devices across Netflix?"

Before Elasticsearch, the incident-to-resolution time was more than two hours. The process involved looking at the logs, grepping the logs, and looking at the cause of error and where there's a mismatch between the manifest and what is being streamed to you. With Elasticsearch, the resolution time decreased to under 10 minutes. That has been a great thing.

Use case 3: Viewing history

As you watch Netflix, you build what we call a "viewing history", which is basically the titles you have been watching over the past few days. It keeps a bookmark of where you were, and you can click to resume from where you stopped. If you look at your account activity, you can see the date that you watched a particular title and you can report if there's a problem viewing a title.

For viewing history, we needed a data store that could store time series in a dataset. We needed to support a high number of writes. A lot of people are watching Netflix, which is great, so the viewing history service receives a lot of writes. Because we are deployed in three regions, we wanted cross-region replication so that if there's a problem within one region, we can shift the traffic and have the user's viewer history available in the other regions as well. Support of large datasets was important, since viewing history has been growing exponentially.

We used Cassandra for this. Cassandra is a great NoSQL distributed data store that offers multi-data-center, multi-directional replication. This works out great because Cassandra is doing the replication for us. It is highly available and highly scalable. It has great fault detection and multiple replicas, so that a node going down doesn't cause website downtime. We can define different consistency levels so that we never experience downtime, even though there are nodes that will always go down in our regions.

Data model

The data model for viewing history started simple. We have a row key, which is the customer or user ID. Each title a user watches is a column in that particular column family. When you watch, you are writing to the viewing history, and we just write a tiny payload: the latest title you watched. Viewing history grows over time, and Cassandra capably handles wide rows, so there is no problem. You can read your whole viewing history, and when you do so, you are paginating through your rows.

We quickly ran into issues with this model. The viewing history is quite popular, so the dataset is growing rapidly. A few customers have a huge viewing history, so the row becomes very wide. Even though Cassandra is great for wide rows, trying to read all of that data in memory causes heap pressures and compromises the 99th-percentile latencies.

New data model

So we have a new model, which we split into two column families. One is the live viewing history, with a similar pattern of each column being a title, so we can continue to write small payloads. And then we have a roll-up column family, which is a combination of all historical datasets that is rolled up into another, compressed column family. This means we have to do two reads, once from the compressed family and once from the live column family. This definitely helps with the size. We drastically reduced the size of the dataset because half of the data was compressed.

The roll-up happens in the path of read. When the user is trying to read from viewing history, the service knows how many columns they have read. And if the number of columns is more than whatever we think it should be, then we compress the historical data and move it to the other column family. This happens all the time based on your reads, which works out very nicely.

Use case 4: Digital-asset management

Our content platform engineering team at Netflix deals with tons of digital assets, and needed a tool to store the assets as well as the connections and relationships among these assets.

For example, we have lots of artwork, which is what you see on the website. The art can come in different formats, including JPEG, PNG, etc. We also have various categories of artwork: a movie can have art, a character can have art, and a person can have art, etc.

And each title is a combination of different things in a package. The package can include video elements, such as trailers and montages, and the video, audio, and subtitle combination. For example, we can have French in the video format with subtitles in French and Spanish. And then you have relationships, like a montage is a type of video.

We wanted a data store where we could store all of these entities as well as the relationships.

Our requirements for the digital-asset management service were one back-end plane to store the asset metadata, the relationships, and the connected datasets — and the ability to quickly search that. We used Titan, which is a distributed graph database. It's great for storing graph datasets, and it supports various storage back ends. Since we already support Cassandra and Elasticsearch, it was easy to integrate into our service.

Use case 5: Distributed delayed queues

The Netflix content platform engineering team runs a number of business processes. Rolling out a new movie, content ingestion and encoding, or uploading to the CDN are all business processes that require asynchronous orchestration between multiple microservices. Delayed queues form an integral part of this orchestration.

We want delayed queues that are distributed and highly concurrent because multiple microservices are accessing them. And we wanted at-least-once delivery semantics for the queue and a delayed queue, because there are relationships between all these microservices and we don't know when the queue will be consumed. A critical requirement was having priorities within the shard, so that we can pick up the queue with the highest priority.

For this particular service, we used Dynomite. Netflix open-sourced Dynomite some time ago. It is a pluggable data store that works with Redis, Memcached, and Rocks DB. It works for this use case because Redis has data structures

Netflix's Cloud Database Engineering team provides data stores as a service, with self-provisioning capabilities that allow application users to create clusters on their own.

that support queues very well. Early on, we tried to make queues work with Cassandra and failed miserably, running into all kinds of edge cases. Dynomite worked superbly for us in this case. And it provides multiple-data-center replication and sharding so we, as application owners, need not worry about data being replicated across regions or data centers.

Netflix maintains three sets of Redis structures for each queue. One is a sorted set that contains queue elements by score. The second is a hash set that contains the payload, and the key is the message ID. The third is a sorted set that contains messages consumed by the client, but which have yet to be acknowledged. So the third is the unacknowledged set.

Identifying the challenges

I love this quote, but I don't think my on-call team feels like this: "I expected times like this — but I never felt that they'd be so bad, so long, and so frequent."

The first challenge my team faces is the wide variety and the scale. We have so many different flavors of data store, and we have to manage and monitor all these different technologies. We need to build a team that is capable of doing all this while making sure the team has the skills to cater to all of these different technologies. Handling that variety, especially with a small team, becomes a challenge to manage.

The next challenge is predicting the future. With a combination of all of these technologies, we have thousands of clusters, tens and thousands of nodes, petabytes of data. We need to predict when our cluster risks running out of capacity. My central-platform team should know each cluster's head room so that if the application team says they are increasing capacity or throughput or adding a new feature that causes an increase in the back-end IOPS, we should be able to tell them that their cluster is sufficient or needs to scale up.

For maintenance and upgrades across all clusters, software or hardware, we need to know whether we can perform maintenance without impacting production services. Can we build our own solution or should we buy something that's out there?

Another challenge is monitoring. We have tens and thousands of instances, and all of these instances are sending metrics. When there's a problem, we should know which metrics make the most sense and which we should be looking at. We must maintain a high signal-to-noise ratio.

Overcoming challenges

The very first step in meeting these challenges is to have experts. We have two or three core people in our Cassandra cloud database engineering team that we call subject-matter experts. These people provide best practices and work closely with the microservice teams to understand their requirements and suggest a back-end data store. They are the ones who drive the features and best practices, as well as the product future and vision.

Everybody in the team goes on call for all of these technologies, so it's useful to have a core set of people that understand what's happening and how we can really fix the back end. Instead of building automation that applies patches on top of what is broken, we can contribute to the open

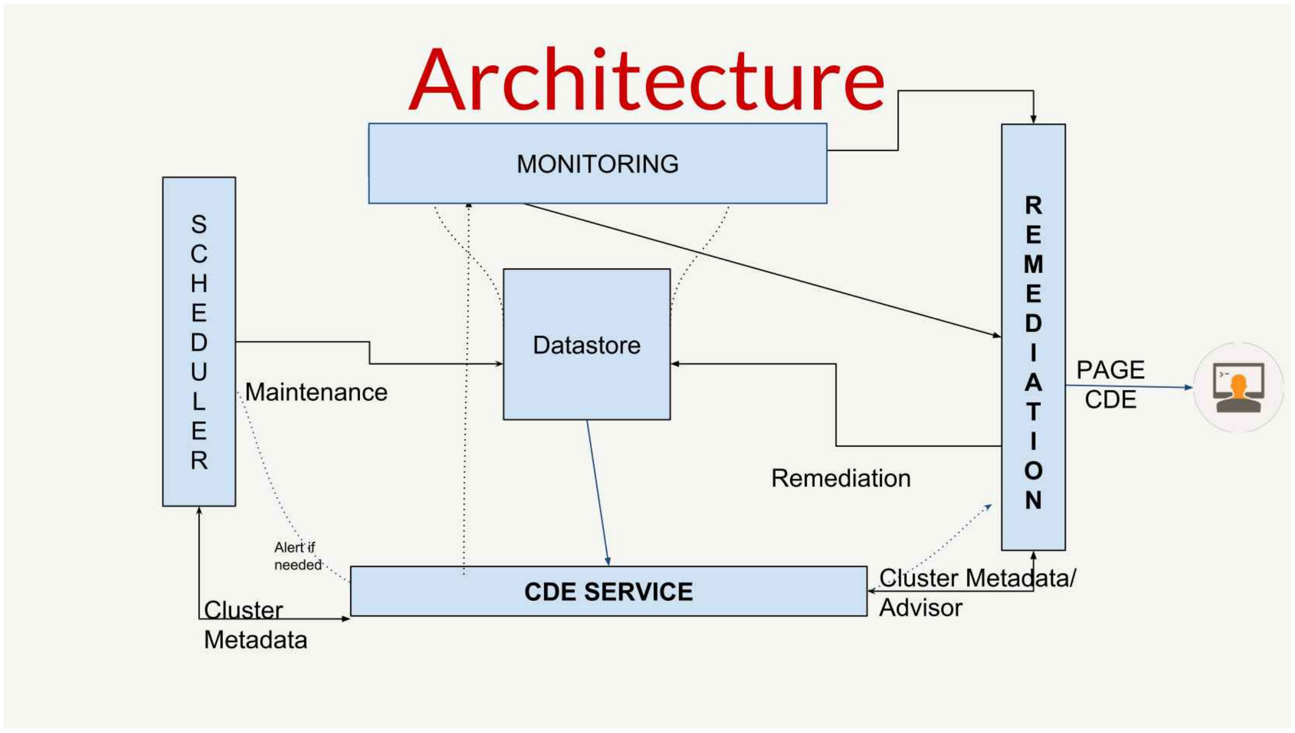


Figure 1: CDE architecture

source or to the back-end data tier — and produce a feature.

Next, we build intelligent systems to work for us. These systems take on all automation and remediation. They accept the alerts, look at the config, and use the latency thresholds we have for each application to make decisions, saving people from getting paged for each and every alert.

CDE Service

CDE Service helps the CDE team provide data stores as a service. Its first component captures the thresholds and SLAs. We have thousands of microservices; how do we know which service requires what 99th-percentile latency? We need a way to look at the clusters and see both the requirements and what have we promised so that we can tell if a cluster is sized effectively or needs to scale up.

Cluster metadata helps provide a global view of all the clusters: the software and kernel version each runs, its size, and the cost of managing it. The metadata helps the application team understand the cost associated with a particular back end and the data they are trying to store, and whether or not their approach makes sense.

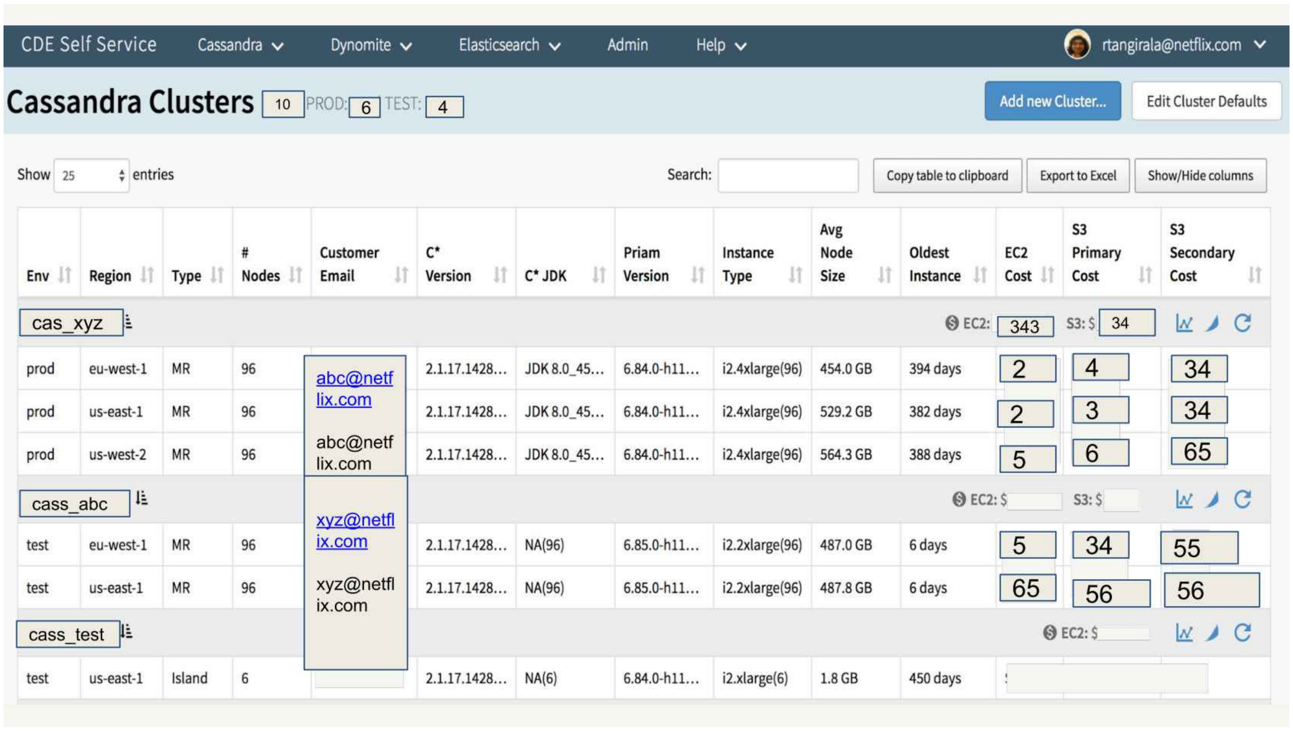
The self-service capability of CDE Service allows application users to create clusters on their own, without the CDE team getting in the way. The users don't need to understand all the nitty-gritty details of the back-end YAML; they only need to provide minimal information. We create the cluster and make sure that it is using the right settings, it has the right version, and it has the best practices built in.

Before CDE Service, contact information only sat outside the system. For each application, we'd need to know who to contact and which team to page. It becomes tricky when you're managing so many clusters, and having some central place to capture this metadata is crucial.

Lastly, we track maintenance windows. Some clusters can have maintenance windows at night, while others receive high traffic at the same time. We decide on an appropriate maintenance window for a cluster's use case and traffic pattern.

Architecture

Figure 1 shows the architecture, with the datastore in the center. For the scheduler on the left, we use Jenkins, which is based on cron and which allows us to click a button to do upgrades or node replacements. Under that is CDE Service, which captures the cluster metadata and is the source of all information like SLAs, PagerDuty information, and much more. On the top is the monitoring system. At Netflix, we use Atlas, an open-source telemetry system, to capture all of the metrics. Whenever there's a problem and we cannot meet the 99th-percen-



Cassandra Clusters 10 PROD: 6 TEST: 4

Show 25 entries Search:

Env	Region	Type	# Nodes	Customer Email	C* Version	C* JDK	Priam Version	Instance Type	Avg Node Size	Oldest Instance	EC2 Cost	S3 Primary Cost	S3 Secondary Cost
cas_xyz											EC2: 343	S3: $ 34	
prod	eu-west-1	MR	96	abc@netflix.com	2.1.17.1428…	JDK 8.0_45…	6.84.0-h11…	i2.4xlarge(96)	454.0 GB	394 days	2	4	34
prod	us-east-1	MR	96		2.1.17.1428…	JDK 8.0_45…	6.84.0-h11…	i2.4xlarge(96)	529.2 GB	382 days	2	3	34
prod	us-west-2	MR	96	abc@netflix.com	2.1.17.1428…	JDK 8.0_45…	6.84.0-h11…	i2.4xlarge(96)	564.3 GB	388 days	5	6	65
cass_abc											EC2: $	S3: $	
test	eu-west-1	MR	96	xyz@netflix.com	2.1.17.1428…	NA(96)	6.85.0-h11…	i2.2xlarge(96)	487.0 GB	6 days	5	34	55
test	us-east-1	MR	96	xyz@netflix.com	2.1.17.1428…	NA(96)	6.85.0-h11…	i2.2xlarge(96)	487.8 GB	6 days	65	56	56
cass_test											EC2: $		
test	us-east-1	Island	6		2.1.17.1428…	NA(6)	6.84.0-h11…	i2.xlarge(6)	1.8 GB	450 days			

Figure 2: CDE Self Service UI

tile latency, the alert will go off. On the very right is the remediation system, an execution framework that runs on containers and that can execute automation.

Anytime an alert fires, the monitoring system will send the alert to the remediation system. That system will perform automated remediation on the data store and won't even let the alert go to the CDE team. Only in situations for which we have not yet built automation will alerts come directly to us. It is in our team's best interest to build as much automation as possible, to limit the number of on-call pages we need to respond to.

SLA

Figure 2 shows the cluster view where I can look at all of my clusters. I can see what version they are running, which environment they are, which region they are in, and what are the number of nodes. This view also shows the customer email, the Cassandra version, the software version, the hardware version, the average node count, and various costs. I can also look at my oldest node, so I can see if the cluster has a very old node we need to replace, then we will just run remediations. There's a job that scans for old nodes and run terminations. In the interest of space, I have not shown many columns, but you can pick what information you want to see.

We have another UI for creating new clusters, specific to each data store. An application user needs to provide only a cluster name, email address, the amount of data they are planning to store, and the regions in which to create the cluster — then the automation kicks off the cluster creation in the background. This process makes it easy for a user to create clusters whenever they want, and since we own the infrastructure, we make sure that the cluster creation is using the right version of the data store with all of the best practices built in.

When an upgrade is running, it can be tricky to figure out what percentage of the test clusters and prod clusters have been upgraded across a fleet that numbers in the thousands. We have a self-service UI to which application teams can log in to see how far along we are in the upgrade process.

Machine learning

Earlier, I mentioned having to predict the future. Our telemetry system stores two weeks of metrics, and previous historical data is pushed to S3. We analyze this data using Kibana dashboards to predict when the cluster will run out of capacity.

We have a system called predictive analysis, which runs models to predict when a cluster will run out of capacity. The system runs in the background and pages us or notifies us on a Slack channel when it expects a cluster to exceed capacity in 90 days. With Cassandra, we only want to use a third of the storage allocation for the dataset, a third for the backups, and the last third for compactions. It is important to have monitoring in place and to have a system that warns us beforehand, not at the cusp of the problem because that leads to all kinds of issues.

Since we are dealing with stateful persistence stores, it is not easy to scale up. It's easier with stateless services; you can do red/black or scale up the clusters with auto-scaling groups and the clusters can increase in size. But it's tricky for persistence stores because it's all data on nodes, and the stores have to stream to multiple nodes. That's why we use predictive analysis.

Proactive maintenance

Things go down in the cloud and hardware is bound to fail. We registered to receive Amazon's notifications and we terminate the nodes in advance instead of waiting for Amazon to terminate them for us. Because we are proactive, we can do the maintenance in the window we like, as well as hardware replacements, terminations, or whatever we want to do.

For example, we don't rely on Cassandra's bootstrap ability to bring up nodes because that takes a lot of time. It takes hours and sometimes even days for clusters, like some of ours, with more than one terabyte of data per node. In those cases, we have built a process that copies the data from the node, puts it into a new node, then terminates the first node.

Upgrades

Software and hardware upgrades across all these different instances of polyglot persistence is an effort because any change to the back end can have a big impact. A problem, like a buggy version, can compromise all of your uptime. We have built a lot of confidence into our upgrades with Netflix Data Bench (NDBench), an open-sourced benchmarking tool. It is extensible so we can use it for Cassandra, Elasticsearch, or any store that we want. In the NDBench client, we specify the number of operations we want to throw at our cluster, the payload, and the data model we want. This allows application teams to test their own applications using NDBench.

When we upgrade, we look at four or five popular use cases. For example, we may try to capture 80 percent reads and 20 percent writes or 50 percent reads and 50 percent writes. We are trying, with only a few use cases, to capture the more common payloads people are using in the clusters. We run the benchmark before the upgrade, capturing the 99th-percentile and average latencies. We perform the upgrade and run the benchmark again. We compare the before and after benchmarks to see if the upgrade has introduced any regression or has caused problems that increased the latencies. This helps debug a lot of issues before they happen in production. We never upgrade when this particular comparison reveals a problem. That's the reason we are able to roll out all these upgrades behind the scenes without our application teams even realizing that we are upgrading their cluster.

Real-time health checks

We also handle health checks at the node level and cluster level. Node level is whether or not a data store is running and if we have any hardware failures. Cluster level is what one node thinks about the other nodes in the cluster.

The common approach is to use cron to poll all the nodes, then use that input to figure out whether or not the cluster is healthy. This is noisy, and will produce false positives if there are network problems from the cron system to the node or if the cron system goes down.

We moved from that poll-based system to continual, streaming health checks. We have a continual stream of fine-grained snapshots being pushed from all the instances to a central service we call Mantis, which aggregates all the data and creates a health score. If the score exceeds a certain threshold, the cluster is determined to be not healthy.

We have a few dashboards where we can see the real-time health. The macro view shows the relative sizes of the clusters with color coding to indicate if a cluster is healthy or not. Clicking on a unhealthy node will show a detailed view of the cluster and that node. Clicking on the bad instance shows details about what is causing trouble, which helps us easily debug and troubleshoot the problem.

Takeaway

The takeaway from all of this is that balance is the key to life. You cannot have all your microservices using one persistent store. At the same time, you don't want each and every microservice to use a distinct persistent store. There's always a balance, and I'm hoping with what I've covered you will find your own balance and build your own data store as a service.

Patterns for Microservice Developer Workflows and Deployment

Q&A with Rafael Schloming

InfoQ recently sat down with Rafael Schloming, CTO and chief architect at Datawire, and discussed the challenges that face modern software-driven organizations.

KEY TAKEAWAYS

People don't really care about moving to microservices per se. What they really care about is increasing feature velocity. In order to apply many people to a problem, you need to divide them up into teams, because people simply can't communicate effectively within very large groups.

You can organize your people as independent, cross-functional, and self-sufficient feature teams that own an entire feature from beginning to end. When you do this, you end up breaking up that monolithic process that was the gating factor for feature velocity.

A microservice system of any complexity cannot be instantiated fully locally, and therefore a hosted development platform must provide developer isolation and developer-driven real-time deployments

A service (mesh) proxy like Envoy is a good way to implement developer isolation through smart routing, and it can also provide developer-driven deployments using techniques like canary releasing.

Although the implementation of microservices is often simply a side effect of the desire to increase velocity through application decomposition and decoupling, there are inherent developer workflow and deployment requirements that must be met. Schloming here elaborates further on this and discusses how Kubernetes and the Envoy service proxy (with control planes like Istio and Ambassador) can meet this need.

InfoQ: A key premise of your recent QCon San Francisco presentation appeared to be that organizations that are moving from a monolithic application to a microservice-based architecture also need to break up their monolithic process. Can you explain a little more about this?

Rafael Schloming: This is actually based on the premise that people don't really care about moving to microservices per se — what they really care about is increasing feature velocity. Microservices simply happen to be a side effect of making the changes necessary to increase feature velocity.

It's pretty typical for organizations as they grow to get to a point where adding more people doesn't increase feature velocity. When this happens, it is often because the structure and/or process the organization uses to produce features have become the bottleneck, rather than the headcount.

When an organization hits this barrier and starts investigating why features seem to be taking much longer than seems reasonable given the resources available, the answer is often that every feature requires the coordination of too many different teams.

This can happen across two different dimensions. Your people can be divided into teams by function: product versus development versus QA versus operations. Your people can also be divided up by component: e.g., front end versus domain model versus search index versus notifications. When a single feature requires coordinating efforts across too many different teams, the gating factor for delivering the feature is how quickly and effectively those different teams can communicate. Organizations structured like these are effectively bottlenecked by a single monolithic process that requires each feature to be understood (at some level) by far too much of the organization.

InfoQ: So how do you fix this?

Schloming: In order to apply many people to a problem, you need to divide them up into teams somehow, because people simply can't communicate effectively in very large groups. When you do this you are making a set of tradeoffs. You are creating regions of high-fidelity communication and coordination within each team, and creating low-fidelity communication and relatively poorer coordination between teams.

To improve feature velocity in an organization, you can organize your people as independent, self-sufficient feature teams that own an entire feature from beginning to end. This will improve feature velocity in two ways. First, since the different functions (product, development, QA, and operations) are scoped to a single feature, you can customize the process to that feature area — e.g., your process doesn't need to prioritize stability for a new feature that nobody is using. Second, since all the components needed for that feature are owned by the same team, the communication and coordination necessary to get a feature out the door can happen much more quickly and effectively.

When you do this, you end up breaking up that monolithic process that was the gating factor for feature velocity, and you create many smaller processes owned by your independent feature teams. The side effect of this is that these independent teams deliver their features as microservices. The fact that this is a side effect is really important to understand. Organizations that look to gain benefit directly from microservices without understanding these principles can end up exacerbating their problems by creating many small component teams and worsening their communication problems.

InfoQ: Could you explain how this relates to the three development phases that, you mentioned, applications progress through: prototyping, production, and mission-critical?

Schloming: Each phase represents a different tradeoff between stability and velocity. This in turn impacts how you optimally go about the different kinds of activities necessary to deliver a feature: product, development, QA, and operations.

In the prototyping phase, there is a lot of emphasis on putting features in front of users quickly, and because there are no existing users, there is relatively little need for stability. In the production stage, you are generally trying to balance stability and velocity. You want to add enough features to grow your user base, but you also need things to be stable enough to keep your existing users happy. In the mission-critical phase, stability is your primary objective.

If the people in your organization are divided along these lines (product, development, QA, and operations), it becomes very difficult to adjust how many resources you apply to each activity for a single feature. This can show up as new features moving really slowing because they follow the same process as mission-critical features or it can show up as mission-critical features breaking too frequently in order to accommodate the faster release of new features.

By organizing your people into independent feature teams, you can enable each team to find the ideal stability versus velocity tradeoff to achieve its objective, without forcing a single global tradeoff for your whole organization.

InfoQ: Another key premise from the talk was that teams building microservices must be cross-functional and able to get self-service access to the deployment mechanisms and the corresponding platform properties like monitoring, logging, etc. Could you expand on this?

Schloming: There are really two different factors here. First, if your team owns an entire feature, then it needs expertise in all the components that go into that feature, from front end to back end and anything between. Second, if your team owns the entire lifecycle of a feature from product to operations, your team needs expertise in all these different engineering-related activities — it can't just be a dev team.

Of course, this can require a lot of expertise, so how do you keep the team small? You need to find a way for your feature teams to leverage the work of other teams in the organization without the communication pathways between teams getting in the critical path of feature development. This is where self-service infrastructure comes into play. By providing a self-service platform, a feature team can benefit from the work that a platform team does without having to file a ticket and wait for a human to act upon it.

InfoQ: What kind of tooling can help with self-service access for deployment, and also to the platform?

Schloming: Kubernetes provides some great primitives for this sort of thing — e.g., you can use namespaces and quotas to allow independent teams to safely coexist within a single cluster. However, one of the bigger challenges here comes with maintaining a productive development workflow as your system increases in complexity. As a developer, your productivity depends heavily on how quickly you can get feedback from running code.

A monolithic application will typically have few enough components that you can wire them all together by hand and run enough of the system locally that you have rapid feedback from running code as you develop. With microservices, you quickly get to the point where this is no longer feasible. This means that your platform, in addition to be able to run all your services in production, also needs to provide a productive development environment for your developers. This really boils down to two problems:

1. Developer isolation: With many services under active development, you can't have all your developers share a single dev cluster, or everything is broken all the time. Your platform needs to be able to provision isolated copies of some or all of your system purely for the purpose of development.

2. Developer/real-time deployments: Once you have access to an isolated copy of the system, you need a way to get the code from your fingertips running against the rest of the system as quickly as possible. Mechanically, this is a deployment because you are taking source code and running it in on a copy of prod.

This is pretty different though in some other important respects. When you deploy to production there is a big emphasis on strict policies and careful procedures: e.g., passing tests, canary deploys, etc. For these developer deployments, there is a huge productivity win from being able to dispense with the safety and procedure and focus on speed: e.g., running just the one failing test instead of the whole suite, not having to wait for a git commit and webhook, etc.

InfoQ: Could you explain these problems and how to solve them in a little more depth?

Schloming: For developer isolation, there are two basic strategies:

- Copy the whole Kubernetes cluster.
- Use a shared Kubernetes cluster, but copy individual resources (such as Kubernetes

services, deployments, etc.) for isolation, and then use request routing to access the desired code.

Almost any system will grow to the point of requiring both strategies.

To implement developer isolation, you need to ensure all your services are capable of multi-version deployments, and you need a layer-7 router, plus a fair amount of glue to wire it all into a safe and productive workflow on top of git. For multiversion deployments, I've seen people use everything from sed to envsubst to fancier tools like Helm, ksonnet, and Forge for templating their manifests. For a layer-7 router, Envoy is a great choice and super easy to use, and is available within projects like Istio and the Ambassador API gateway that add a more user-friendly control plane.

For developer/real-time deployments, there are two basic strategies:

- Run your code in the Kubernetes cluster, and optimize the build/deploy times.
- Compile and run your code locally and then route traffic from the remote Kubernetes cluster to your laptop, and from the code running on your laptop back to the your remote cluster.

Both these strategies can significantly improve developer productivity. Tools like Draft and Forge are both geared towards the first strategy, and there are tools like kube-openvpn and Telepresence for the second.

One thing is for sure, there is still a lot of DIY required to wire together a workable solution.

InfoQ: You mentioned the benefit that service-mesh technology, like Envoy, can provide for interservice communication ("east-west" traffic) in regard to observability and fault tolerance. What about ingress ("north-south" traffic)? Are there benefits to using similar technology here?

Schloming: Yes. In fact, in regards to bang for buck, this is the place I would look to deploy something like Envoy first. By placing Envoy at the edge of your network, you have a powerful tool to measure the quality of service that your users are seeing, and this is a key building block for adding canary releases into your dev workflow, something that is critical for any production or mission-critical services you have.

InfoQ: How do you think the Kubernetes ecosystem will evolve over the next year? Will some of the tools you mention become integrated within this platform?

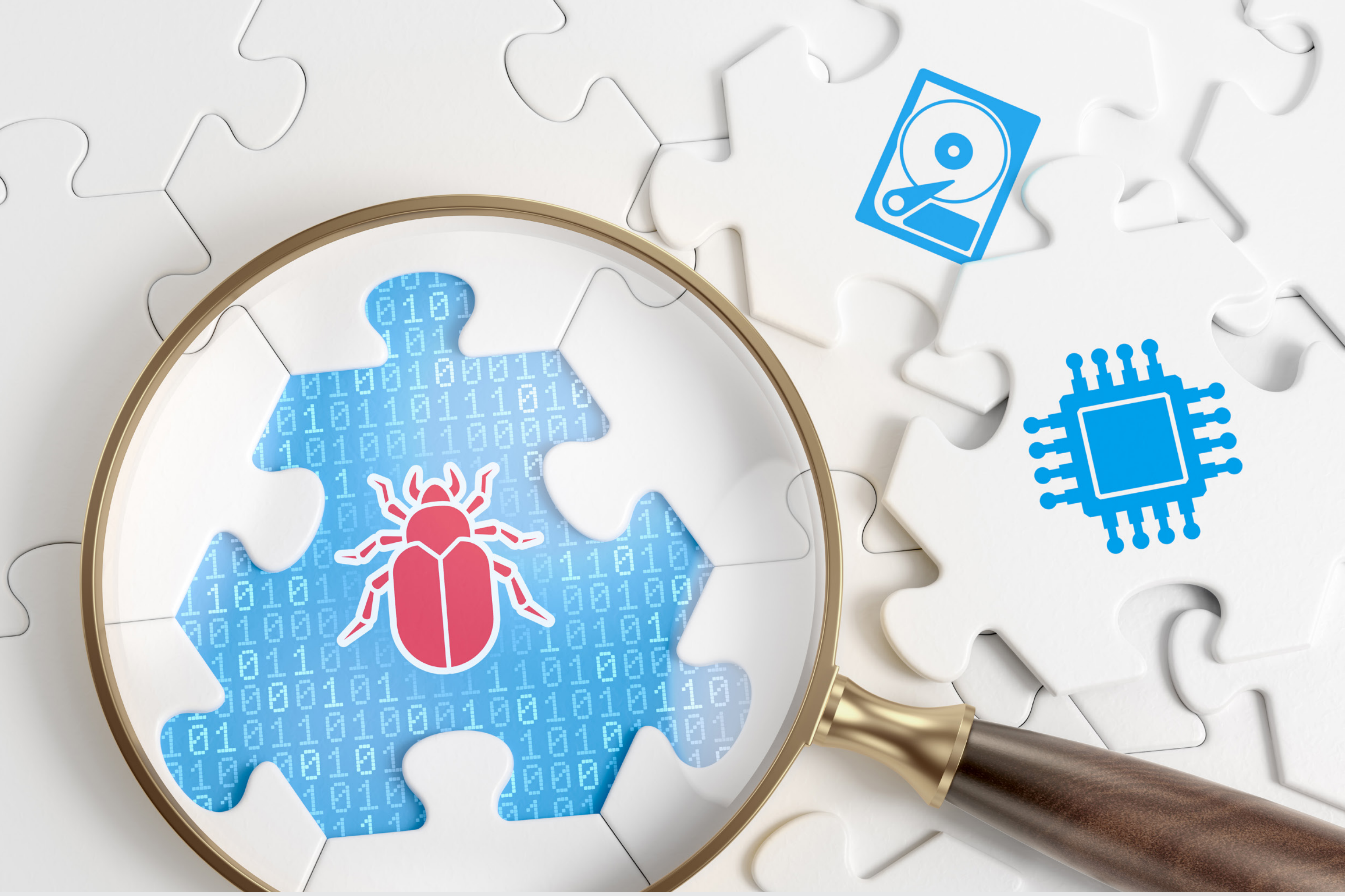
Schloming: I certainly wouldn't be surprised to see deeper integration between Envoy and Kubernetes. One thing I certainly hope to see is some stabilization. Kubernetes and Envoy are both foundational pieces of technology. Together they provide the core parts of an extremely flexible and powerful platform, but you really need to spend a while becoming an expert in order to leverage them.

I think in regards to the larger ecosystem, we'll see more projects geared at allowing non-experts to leverage some of the benefits these tools can offer.

InfoQ: Is there anything else you would like to share with InfoQ readers?

Schloming: The Datawire team is working on a range of open-source tooling for improving the Kubernetes developer experience, and so we are always keen to get feedback from the community. Readers can contact us through our website, Twitter, or Gitter, and you can often find us speaking at tech conferences.

The video from Schloming's QCon San Francisco 2017 talk "Microservices: Service-Oriented Development" can be found on InfoQ alongside a summary of the talk.



Watch presentation online on InfoQ

Debugging Distributed Systems

Idit Levine Discusses the Squash Microservices Debugger

KEY TAKEAWAYS

Debugging a microservice-based application is more challenging than debugging a monolithic application as it is difficult to attach a native debugger to multiple processes that communicate across a network.

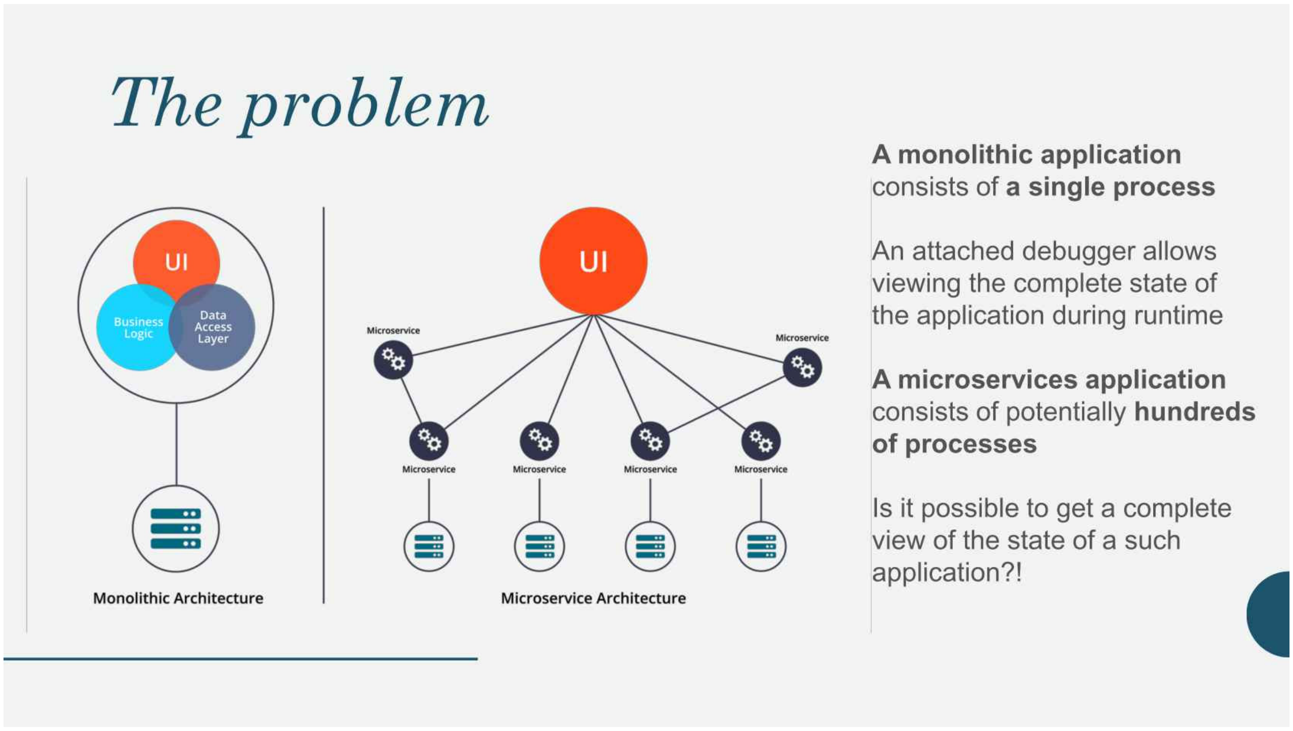
Currently, the best approach to debugging microservices relies on obtaining a trace of all transactions and dependencies using tools that, for example, implement the OpenTracing API standard. OpenTracing tools are powerful, but they have limitations and gaps, especially for ad hoc observation.

Squash is an open-source microservice debugging tool that orchestrates run-time debuggers attached to microservices (running within containers deployed onto IaaS or CIaaS), and provides familiar features like setting breakpoints, stepping through the code, viewing and modifying variables, etc.

A service mesh may be the future best point of integration for such observation and debugging, and Squash currently includes early integration work with Istio and the Envoy service proxy.

At QCon San Francisco 2017, Idit Levine, founder and CEO of solo.io, presented "Debugging Containerized Microservices" in which she outlined the challenges of debugging a distributed microservice-based system.

Levine began by comparing the debugging of monolithic and microservice-based applications. A monolithic application typically consists of a single process, and attaching a debugger to this process reveals the complete state and the flow of execution. Because a microservice-based application is



The problem

A monolithic application consists of **a single process**

An attached debugger allows viewing the complete state of the application during runtime

A microservices application consists of potentially **hundreds of processes**

Is it possible to get a complete view of the state of a such application?!

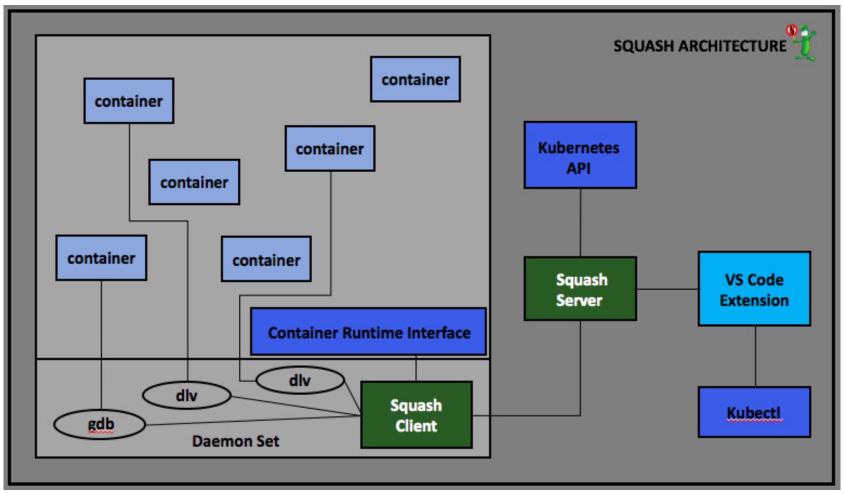
inherently a distributed system consisting of multiple processes communicating over a network, this adds significant complexity to the challenges of effective debugging.

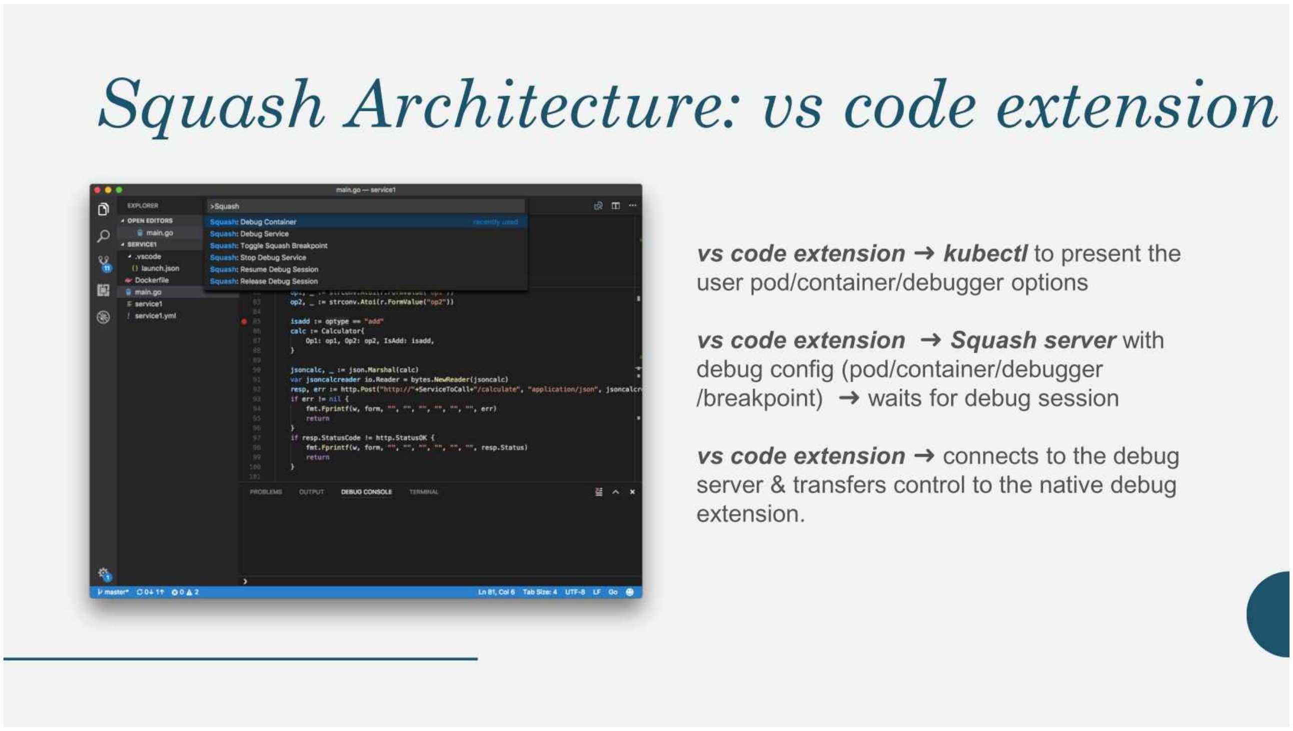
The remainder of the talk presented three approaches to debugging microservices: distributed tracing, using the open-source Squash microservices debugger that Levine has created, and exploiting the underlying capabilities of a service mesh.

Distributed tracing tools, such as Open Zipkin — which implements the OpenTracing API specification hosted by the Cloud Native Computing Foundation — can be used to monitor and understand the flow of execution through a microservices-based application. This approach has several advantages: it easily sends data to any logging tool, even from OSS components; it enables critical-path analysis; developers can drill down into request latency and other associated trace context metadata in very high fidelity; and operators can conduct system topology analysis and identify bottlenecks due to shared or contended resources.

Disadvantages to distributed tracing include: the inability to perform run-time debugging or modification of application state; the approach often requires wrapping/decorating and changing the code, which can incur a performance penalty at run time; and there is no holistic view of the application state — developers can only see what was output as part of the trace and associated baggage.

For the second approach, Levine presented her company's open-source Squash microservices debugger. Squash currently supports debugging within Visual Studio Code (VS Code) — or in IntelliJ for Java and Kubernetes only — of microservice applications that are written in a language that can be debugged by Delve (the Go language), GDB (C++, Objective C, Java, etc.), and its own language-specific debugging protocols for Java, Node.js, and Python. The services must be deployed to the Kubernetes container-orchestration platform or a platform that can use Istio, which itself is currently





Squash Architecture: vs code extension

```
    op2, _ := strconv.Atoi(r.FormValue("op2"))

    isadd := optype == "add"
    calc := Calculator{
        Op1: op1, Op2: op2, IsAdd: isadd,
    }

    jsoncalc, _ := json.Marshal(calc)
    var jsoncalcreader io.Reader = bytes.NewReader(jsoncalc)
    resp, err := http.Post("http://"+ServiceToCall+"/calculate", "application/json", jsoncalcr
    if err != nil {
        fmt.Fprintf(w, form, "", "", "", "", "", "", err)
        return
    }
    if resp.StatusCode != http.StatusOK {
        fmt.Fprintf(w, form, "", "", "", "", "", "", resp.Status)
        return
    }
```

vs code extension* → *kubectl to present the user pod/container/debugger options

vs code extension* → *Squash server with debug config (pod/container/debugger /breakpoint) → waits for debug session

vs code extension → connects to the debug server & transfers control to the native debug extension.

Kubernetes-focused. (Istio does offer limited support for Docker and HashiCorp's Nomad, but it is worth noting that all of Squash's Istio examples use Kubernetes as the underlying platform). Squash would like to add support for more IDE, language, and runtime platforms, and encourages community contributions.

The Squash architecture consists of a Squash server that is deployed and runs on the target platform (for example, as a DaemonSet on a Kubernetes node). The server holds the information about the breakpoints for each application, and orchestrates the Squash clients. The Squash clients also deploy on the target platform. Squash uses an IDE as its user interface — as mentioned, currently only VS Code and IntelliJ (for Java and Kubernetes). Squash commands are available in the IDE command palette after installing the Squash extension.

Levine's third approach to debugging microservices is to use the capabilities of a service mesh such as Istio or Envoy. A service-mesh data plane, such as Envoy, touches every packet/request in the system, and is responsible for service discovery, health checking, routing, load balancing, authentication/authorization, and observability. A service-mesh control plane, such as Istio, provides policy and configuration for all running data planes in the mesh. These properties provide ideal points of introspection and execution flow control. Istio currently integrates with the Open Zipkin and Jaeger distributed tracing systems and, as mentioned, with Squash.

She concluded by suggesting that the ultimate solution would be to integrate all three of these approaches to debugging, and encouraged the audience to get involved via the solo.io Slack channel and contribute to Squash.

InfoQ recently sat down with Levine to elaborate on the challenges of observing and debugging distributed systems and applications.

InfoQ: How has operational and infrastructure monitoring evolved over the last five years? How have cloud, containers, and new architectural styles like microservices impacted monitoring and debugging?

Idit Levine: Monitoring the state of an application is important during development and in production. With a monolithic application, this is rather straightforward, since one can attach a native debugger to the process and have the ability to get a complete picture of the state of the application and its evolution.

Monitoring a microservice-based application poses a greater challenge, particularly when the application is composed of tens or hundreds of microservices. Due to the fact that any request may involve being processed by many microservices running multiple times — potentially on different servers — it is exceptionally difficult to follow the "story" of the application and identify the causes of problems when they arise.

Currently, the main methodology relies on obtaining a trace of all transactions and dependencies using tools that, for example, implement the OpenTracing standard. These tools capture timing, events, and tags, and collect this data out of band (asynchronously). OpenTracing allows users to perform critical-path analysis, monitor request latency, perform topological analysis and identify bottlenecks due to shared resources. Users can also log what they think could be useful data, like the values of different variables, error messages, etc.

InfoQ: We've been eagerly watching the evolution of Squash and would be keen to hear the goals of the project and the rationale for creating this.

Levine: OpenTracing tools are very powerful, but they have limitations and gaps. Since logging the state of the application during run time can be expensive and result in performance overhead, one needs to limit the amount of collected information. One way to do this is to follow only a subset of the transactions, and not all of them. Tuning the size of this sample represents a tradeoff between the amount of information collected on one hand and the price in performance and costs on the other.

One consequence is that once a problem is identified, it is possible that some needed information is missing. Obtaining this information requires running the application again, and waiting for the data to be collected. Moreover, OpenTracing is not a run-time debugger and does not allow changing variables during run time to explore potential solutions to a problem. Any attempt to fix a problem requires wrapping the code, running the application, and waiting for the data again. Solving a problem may necessitate several such iterations, which can be both daunting and expansive.

Our vision for Squash is to complement the OpenTracing tools and close these gaps. The main goal of Squash is to provide an efficient tool for debugging microservices applications. Squash orchestrates run-time debuggers attached to microservices, providing familiar features like setting breakpoints, stepping through the code, viewing and modifying variables, etc. Importantly, Squash allows the developer to seamlessly follow the application and skip between microservices. Squash takes care of all the necessary piping, allowing developers to focus on their own code and solve the issues they actually care about. To make Squash accessible and easy to adopt, it integrates with existing popular IDEs.

Squash is designed to provide essential capabilities for monitoring the lifecycle of an application both in the development phase, allowing development of robust code, as well as during production, allowing fast adaptation of the code when new difficulties arise.

InfoQ: What other tools do you think future developers will need to understand and debug large-scale, rapidly evolving container-based applications?

Levine: As a community, we should aspire to provide distributed applications the same level of observability and control that is available for monolithic applications. A combination of existing tools already points us in the right direction. Log collection can be done by OpenTracing tools, metrics collected by Prometheus, and debugging by Squash. All of these methods should plug into a service mesh to achieve full efficiency.

InfoQ: What role do you think QA/testers have in relation to observability and debuggability of a system?

Levine: In one possible mode of action, I would expect the QA and testers to focus on the logs and provide context. With container-based applications, this should be done using OpenTracing. The developer will then be able to reproduce the bug and use Squash to attach a debugger, step through the code, and resolve the issue.

InfoQ: Is there anything else you would like to share with the InfoQ readers?

Levine: We at solo.io are working hard at building more open-source tools to facilitate microservices development and operation. In particular, we are focused on innovative and helpful tools to accelerate adoption of microservices in the enterprise. We are super excited about our plans for 2018 — please stay tuned!

Microservices Patterns and Practices Panel

Microservices almost seem to be the de facto way to build systems today, but are they always the answer?

KEY TAKEAWAYS

Defining the boundary contexts for microservicess is a particular challenge. In general, you need to really know your problem domain before you can get this right — in a field like banking, where the boundary cases are already well known and have been for many years, you can operate as a startup. That's less true in other problem domains, but you can start by organizing around business objects.

Decomposition applies at many levels. In a sense, you decompose methods, classes, packages, and modules, and so microservices is just another level in that kind of hierarchy. However, they also have a strong relationship to team structure.

Scale is one strong reason to consider microservices, and the most often cited perhaps along with team velocity, but another is security — if you have two things that shouldn't share a trust domain, for example.

If you do choose microservices, you'll face challenges at scale at both a technical and organizational level. What strategies should you use now that you are effectively building a distributed system? What's the one thing you wish you'd known before you got here?

This panel session brought together many of the most popular session speakers at QCon San Francisco 2017 for a frank discussion on microservices with the Microservices track host.

The panelists were:

- **Chris Richardson**, the author of *POJOs in Action* and founder of the original Cloud Foundry, an early Java PaaS for Amazon EC2
- **Randy Shoup**, a 25-year veteran of Silicon Valley, currently VP Engineering at Stitch Fix
- **Louis Ryan**, a core contributor to Istio and gRPC and a principal engineer at Google
- **Roopa Tangirala**, leader of the Cloud Database Engineering team at Netflix
- **Rafael Schloming**, co-founder and chief architect of Datawire

The session was recorded live as the panelists took questions from the audience. We've lightly edited the transcript.

Question: How do you manage your data when you are doing red/black deployments? For instance, you might have a version that writes new records, which the old version doesn't understand. How does the old version know it is not an error but actually real data?

Roopa Tangirala: In most red/black deployments at Netflix, whenever there are data changes, you can do it. It is stateless; it is not a problem. But when you have changes for Cassandra, it is schema-less. So when you are adding a new column or changing schema, you don't necessarily need to do a DDL to change the schema. You can keep directly inserting into the new data set with the new column. That is one way.

And the other way, if they help the migration from one column family to another, we help them build tools; we own the client libraries so we can help them write to the old and to the new. We have tools like Forklift, which helps move from the source to the destination. But not all red/black deployments need changes where we are moving data around, at least at Netflix.

Chris Richardson: For zero-downtime deployments, constrain the kinds of changes you can make at the database level. So you could add a nullable column, for instance, but you cannot just drop a column. So carefully make changes, and decouple database schema changes from your zero-downtime deployment.

Randy Shoup: Yeah, don't do what you just said. Not even kidding. What you did is you broke the interface. You made a non-backward-compatible data change and you exposed it to other people, and you did it, in a way, in between a minor release. To people familiar with semantic versioning, what you just described was a non-backward-compatible major version change: "We used to produce data in this form, and now we produce it in this non-backward-compatible other form."

And so, don't do that. What you can do is what Chris said. There are lots of ways to make backward-compatible changes. You can add an optional field — we'd need to talk about it in a little more detail, sadly. But the idea is that, as a service owner, your primary job is never to break the people that use your service. So you are never allowed to break clients, which are consumers of your events.

Q: Deciding boundary contexts for microservices could be as easy as having orders, and then there could be five types of orders, and then the microservices becomes a monolith after a while. How do you decide on a boundary context so that it is still good enough after a couple of years?

Louis Ryan: Mostly, it is probably going to be informed by your development practice in your development divisions, rather than any strict semantic thing you would try to guess from the get-go. I tend to think of microservices as emergent patterns that come out of the need for decoupling. Usually, the decoupling works at development-team boundaries pretty well, or at functional responsibilities within development teams. That's where I would start.

Richardson: I would sort of say this is one of the hardest problems, and it is really not specific to microservices. Another way of rephrasing it is "What are the boundaries of my module?" And I think picking module boundaries is difficult.

Unfortunately, there is no mechanical process that, if you apply it, will come up with a perfect set of module boundaries. In the case of services, most of them are organized around particular business objects, like order management and customer management and so on. But that is your first guess, and you go with that, and if later on, you find out that some services got too big, then hopefully at that point there is a clear-enough boundary between the two internal parts of that module to let you split it in a meaningful way.

The point of a service is to enable a small team of developers to deliver rapidly and safely. And so if a service gets too big, that really means the team that is developing it has gotten too big, and they are weighed down by communication overhead. And so you kind of want to split the team, and you want to split the code, so they can go back to being small, nimble teams again.

Shoup: Probably, if you are a five-person startup, you might not want to start with microservices. Part of the why is that it is still a little bit complicated, maybe a lot complicated, to build a distributed system and everyone's questions are around things that are complicated. When you are small, you want to start off doing something different. And another part of it is you want to understand your domain and be able decompose it in a reasonable way before you do microservices, because microservices are just a physical manifestation of a decomposition of your domain. So I have found, because I have tried and failed to do it many times, that my first cut at a new problem and figuring out the decomposition of the domain is messed up all the time. And I have gray hair/no hair; I have been doing this for a while.

There are two rare exceptions to the rule of maybe don't start with microservices when you are tiny. The first is if your MVP requires scale. So, if you are building the Heroku competitor, for example, you are building internet infrastructure, so you'd better scale from the beginning. That's a requirement. And the second is if you know your domain super well. One great example is people building new banks. Nubank from Brazil, who gave the first talk yesterday in the Architectures You Always Wondered About track, started with microservices. Why? Because the decomposition of the banking domain we have known for the last 50 years, the components of the bank, are really well understood. But the rest of us, we don't know the domain well enough, and that is why this is such an important problem.

Q: We know that we're a monolithic application, and we know that we want to get to business-context-type services. But where does that line get drawn? Is it a product level, an API level, a microservice level? Is it just what feels right?

Rafael Schloming: That is a hard question, but I think one of the ways to, sort of, think about it is actually something Randy said earlier, which is don't think about the size of a microservice in terms of its lines of code, think about the scope. And how do you define scope? Well, you need to understand what it is you are trying to achieve at a high level, in one or two sentences.

It is really a negotiation between the user of a service and the team that delivers that service. You need to track the usage; if your users are happy, then you're done. It really helps to think in terms of that framing, understanding who the user of the service is, and going from there. And, from that perspective, you can just try a lot of different kinds of APIs that will sort of serve the same mission and figure out what you need. And, again, you can track how successful you're making your users in order to measure your progress as you iterate through the difficult design space.

Shoup: So this is a little bit of a flip response, but I don't mean it in any aggressive way. Do you guys build one class, like one language class in Java or whatever? How do you know what the scope of the classes are? That's a design thing. The class is a single responsibility; we try to make the interface minimal and try not to be chatty. The reason we ask it that way is not to put you on the spot, and the people that are working for me are laughing right now: this is a thing that I have done many times with my team, where this is a legit thing to say.

That's the answer. You know more than you think about how to design services. If you know how to design classes, for the most part you know how to design services. The only part is recognizing that you cannot be as chatty with services as you can be with something that is in process.

Richardson: Decomposition applies at many levels. In a sense, you decompose methods, classes, packages, and modules, and so the microservices is just yet another level in that kind of hierarchy. One comment I would add is that I think microservices kind of have this important relationship with team structure as well. I think there are two models for microservices. There is this super fine-grain model, which is one service per developer, that seems to be happening at some companies. Or when you have thousands of services — that order of magnitude. Another way of thinking about services is as a small enough "application" that its team can remain nimble and agile. That is a much coarser-grain model of microservices. And so that impacts decomposition.

Ryan: I think it is probably a common problem for a lot of people in the room, that they have a monolith yak that they want to shave, and that is totally fine. Start shaving where you think shaving adds value, and stop shaving where you are not getting any more value. It is okay to have a monolith if it is doing what it is supposed to do. I know that might be heresy here, but if it is doing what it is supposed to do, why touch it? If it is not, shave it, and iterate.

Shoup: A related, excellent question is, more or less, are microservices worth it? And the answer, for most of us, is maybe not. As I tried to say in my talk, it is the 0.1 percent, or 0.01 percent that get really large, where you absolutely need them — there is no way Google, Amazon, Netflix, Stitch Fix work without microservices. But if you don't have a huge load, it is fine to stick with a monolith. When should you go with microservices? Well, when are you unable to scale things independently, when does it slow down, when do things evolve at different rates? That's the wall that you have to scale with microservices.

Richardson: And I want to add to that. If your development velocity is not where it needs to be, I would actually start to review your development practices before switching to microservices. So, for instance, if you are not doing automated testing thoroughly, and I think probably 70-plus percent of organizations, according to a SourceLab report, have not completely embraced automated testing. So if you are one of those, work on that first. And then, you know, once you have the hang of that and you really are able to automate as much as possible, then think about the microservice architecture. It is kind of like try walking before you run.

Schloming: That's a great point, and a great thing to do is just — it doesn't need to be super heavyweight — to track where you spend your time. If you are doing lots of manual testing and that is slowing you down, you don't necessarily think about that on a day-to-day basis. And, you know, if you are spending a lot of time wrestling with particular areas of your monolith, maybe that's the time you should start shaving that particular patch of yak.

Ryan: So I think Randy gave a couple examples of why you might want to do that, scale being one of the more obvious ones that is quoted in the industry. I think there are other reasons; security is a big reason why you might want to shave your monolith, because you have two things that should not be stuck together in the same trust domain. That's a big reason. The development experience is clearly one; release velocity is a big deal. So there's a variety of reasons out there. You know your domain, you know what is going on in your domain, you should be able to reason about those types of decisions.

Richardson: From my perspective. I think that microservices are primarily a way to tackle complexity rather than scale. Obviously, it is a way to scale, but complexity is first.

Q: Can you guys comment on what patterns teams are using to get to microservices? Do I start in the middle where it really matters with an important object or do I do it on the side where it doesn't make a big difference? Can I just slap a REST API on an existing app and call it a microservice?

Richardson: Well, you know, if your yearly bonus depends on having a microservice…. This term "microservice" really does get heavily abused, right? "Can we use a microservice for that" is just kind of the wrong notion, from my perspective. Microservice is shorthand for the microservice architecture, which is an architecture style for an application; it is all about having a system.

Say you have this massive monolith, and there is one part of it that is under very, very active development and another part that you never touch, and you want to extract them out. If you want to build a microservice or a service, then extract out the parts of your application that are frequently changing. Because that will give you the biggest bang for the buck.

Think about your monolith that is on the slow track of development, and everything that you extracted out of the microservice is on the fast track, the rapid deploys and all of that. So you want to invest the effort in those areas that really, really make a difference.

Shoup: I'm going to make some architectural change from the monolith to the microservices. So I want to prove that this fancy millennial way of doing a microservice is actually a thing and will work in our environment. So step zero is to do a pilot. And the way I would like to think about that is to take a vertical, end-to-end actual experience that matters to our business.

Let's imagine that you have something that actually matters to users and you want to build that in a new way. It could legitimately be a new thing you will build a new way or an old thing that exists that you will rebuild in a new way. Either way, take a vertical end-to-end thing and build it in a new way.

Why? We are building a pilot, we want to de-risk it and we want to learn all the things we don't know about the microservice thing. We are doing it as a pilot rather than building the entire infrastructure. We do a vertical end-to-end user experience because we want to be able to be focused on some particular thing and that tells us what we need to do and don't need to do. If we choose something that doesn't matter, we don't know what is in or out. If we choose something that is actually useful, then that will help us to focus on the minimal thing we need to do to get our job done. And the other reason we choose something useful is if that doesn't work, at the worst we have provided some value to our customers.

So that's the step zero, that pilot. And now that that pilot is successful, and we have learned all of these things about how to do things in a new way, then we will call it "microservices". The steps 1 to N are to take the things that have the highest return on investment — not the easiest things, not necessarily the hardest things, but the things that have the highest return on investment and we convert those to the new way first.

So think about the areas that are really fast-changing, maybe that have the highest ROI, or the part of the site with the highest revenue — that would be a place where it would be valuable to move faster to make more revenue. You did the pilot, you de-risked it, and then you do the highest ROI, and then the second and third highest, and you keep going until you run out of patience or resources. And if you run out before you are done, that is cool, because the monolith that still exists is something that you don't care too much about. There wasn't the ROI, it didn't go above the bar of what it would take for you to, you know, get motivated to convert it to microservices.

That is exactly what eBay did. eBay had this monster C++ monolith and they broke it into many applications written in Java. So it wasn't microservices, but the principle is the same. Once they did the pilot and they convinced themselves that Java could work in the eBay infrastructure with the skill set and people and all that kind of stuff, they basically reverse-sorted the site — they took the pages on the site and reverse-ordered them by revenue contribution. So they converted first the top-revenue pages, not because they desperately wanted to have the greatest risk but if and when they ran out of patience, money, or resources, they had the most valuable things done.

They had started the re-architecture in 2000 or 2002 or something like that, and they had mostly finished by, I want to say, 2007 or 2006. It took a while, and even after I left in 2011, there were still things that were on that v2 C++ monolith architecture, but they were things that nobody used; they were simple, they didn't change. So there was no ROI to convert them to the new way.

Q: My question is about the communication between microservices. We talk about having events, so service A talks to service B. For a business-critical service like credit-card processing, we see lot of patterns by Kafka or other brokers once the message is in the broker, and there are ways to recover or retry. But what's the recommendation to ensure that the credit-processing service does issue the event? Kafka now has Kafka Connect, which can publish every database commit or every database transaction straight to Kafka. What if the business object is not the same as what you have in the database?

Tangirala: In terms of services, each application service is the source of truth for the data it is serving. So, for payments processing, in Netflix's case, they don't use Kafka, they have different payments stores. They are using transactional data stores for that payment processing. But basically the idea is that each service is owning the piece of data it is responsible for, and it is the source of truth for that. That is how the interactions happen: other services will ask the service instead of directly either copying the dataset or having multiple copies in their back end.

Richardson: There are several parts to it. One is atomically publishing a message when the data changes. So, conceptually, there's a transaction involved in updating the database and publishing a message. There's a whole thing around transactional messaging, which is kind of a super interesting topic. And so, it ends up reliably being published to the message broker. That's step one. In step two, your message broker has to be reliable. That's what Randy was talking about with at-least-once delivery. And at the consumer end, you need idempotent event or message handling to ensure the correct semantics, and that includes keeping track of all of the message IDs you have seen. It is a whole complex topic, some of which I cover in my book, Microservice Patterns — shameless plug.

Schloming: This area of ownership is like designing classes — ownership and the whole area of communication and this whole event thing. That is where you are transferring responsibility for ownership of some data. And that is where microservices get the most different from designing classes, or one of the areas they get the most different from designing traditional class APIs, because you don't have this same locality of data in the context of a class hierarchy. So it is just something to keep an eye out for.

Q: Related to event-driven architecture, can you share your thoughts from the panel on the use of either pass by value or pass by reference on those messages, how the consumers work with that message, and maybe your thoughts on how to handle ordering those?

Ryan: I can give my opinion, which might also be slightly heretical here, but this is influenced by Google scale. We mostly don't do it. Most service-to-service communication is not reconciled through a broker. We use things like retries and network-level things to get scale by not hitting storage. So again, it is one of the scale questions. If persistent queues in storage give you reliability that you need at your application level at scale, then you should use it. And, I think, at certain scales, some of the patterns might become a little bit more limiting, particularly depending on the amount of work waiting for that. So it is not that we're anti that pattern, per se; we do use that pattern, and we use that pattern encapsulated behind an API with a clear segmentation of responsibility. But, for the most part, we don't do it. We don't do rendezvous or that type of thing.

Richardson: I can't believe you don't use Kafka.

Ryan: We have things that look like it.

Richardson: But Kafka seems fashionable.

Ryan: So I hear.

Richardson: Rightly or wrongly. So when we have been talking about events, in my brain I have translated that into domain events, which are a concept from domain-driven design (DDD). One of the DDD books, Implementing DDD by Vaughn Vernon, has a chapter on domain events that includes a discussion of how much data you you should have in an event, you have a choice. If an order is created, you can publish the order ID, but that is of no use to the consumer because they have to get the order. So there's the concept of event enrichment, which says to put data that is useful to the consumer in the event. And when you publish an order-created event, stuff the order details in there. And when you are using event sourcing, where your events are your storage mechanism for your domain objects, you have no choice except to put the necessary data in there.

And your other point was ordering. I think ordered, at least once, delivery of domain events is really, really important, because if they arrive out of order, then you are going to have pretty weird behavior. And I mean, there might be other situations where you don't care about delivery and you can just pub/sub an event, but ordering is usually quite important.

Shoup: You asked THE question, which is how I deal with event delivery when I might get the thing multiple times, and how do I deal with event ordering? So both of those things you don't have in-process but with messaging across a distributed system you have those problems. I keep threatening to do an event master class.

So, again, on delivery, you can have at most once or at least once. If you care about your event, you want at least once. So that is on failure; I deliver it two times, three times, N times. At most once is basically for logging data, things that on failure you want to drop. Domain events do not fall into that category, but logging things are. That's the first thing, and then you have this multiple times, and then you have to be idempotent; the consumer has to be able to correctly process the same event multiple times. CRDTs is something you should look into if you are kept up at night by these problems.

And there are several ways to deal with event ordering. You can deal with ordering in the bus — blecch, that is not so great. The other way to deal with it is to have events be the notification of a thing happening and then you go and read back to the service that produced the event for the current state of things. These are all legit ways of handling a problem, and think about these answers — there's Randy's way to do it versus Chris's way to do it versus Louis's. Think about that: there's a space of solutions to this problem, and don't take away from it that it's solved by reasoning with first principles

Martin Fowler of refactoring fame gave a wonderful keynote at GOTO Chicago 2017 on event-driven architecture. And he does, in his wonderful way, very clearly, discuss the pros and cons of events as notification, events that carry the data with them, event sourcing, etc.

Ryan: I want to throw in a cautionary tale, not necessarily a parable, I gave a talk earlier on superpowers: beware of superpowers. Event brokers are superpowers. Be careful when you put things into queues when you don't know where or how they are going to come out, or who they will come out to. If you don't know the answers to those questions, you shouldn't put those things into a queue until you can answer the questions. If you have data that you care about or your users care about, you need to reason about those things to some degree. And event brokers have been pitched as a way to give operators control so they can answer those questions or validate that.

Q: When the Web started, everyone was writing interesting apps. Then came Rails and MVC and Rust and people started writing those, and then we had monolithic scales slow us down. Now microservice is the buzzword. You cannot walk into a company and not hear the word "microservices". What are some things that you foresee after the microservice trend saturates? What is next? Microfunctions?

Ryan: Didn't that already happen?

Shoup: The meta answer is look at what Google, Amazon, and Netflix are doing now. Meaning no shame, I will be flip: if you are asking that question, you are years behind these people. And that's a good thing. You can look at what these larger architectures are sharing.

Richardson: At some level, there's a limit beyond which it doesn't meaningfully make sense to decompose a module. Go back to some of the classic work in object-oriented design like the common-closure principle: the things that change for the same reason should be packaged together. And that means, if you decompose a package into two packages — and, really, you have split this business concept across those two packages — then whenever that business concept changes, you are changing both of those packages. So you are going to see this lockstep.

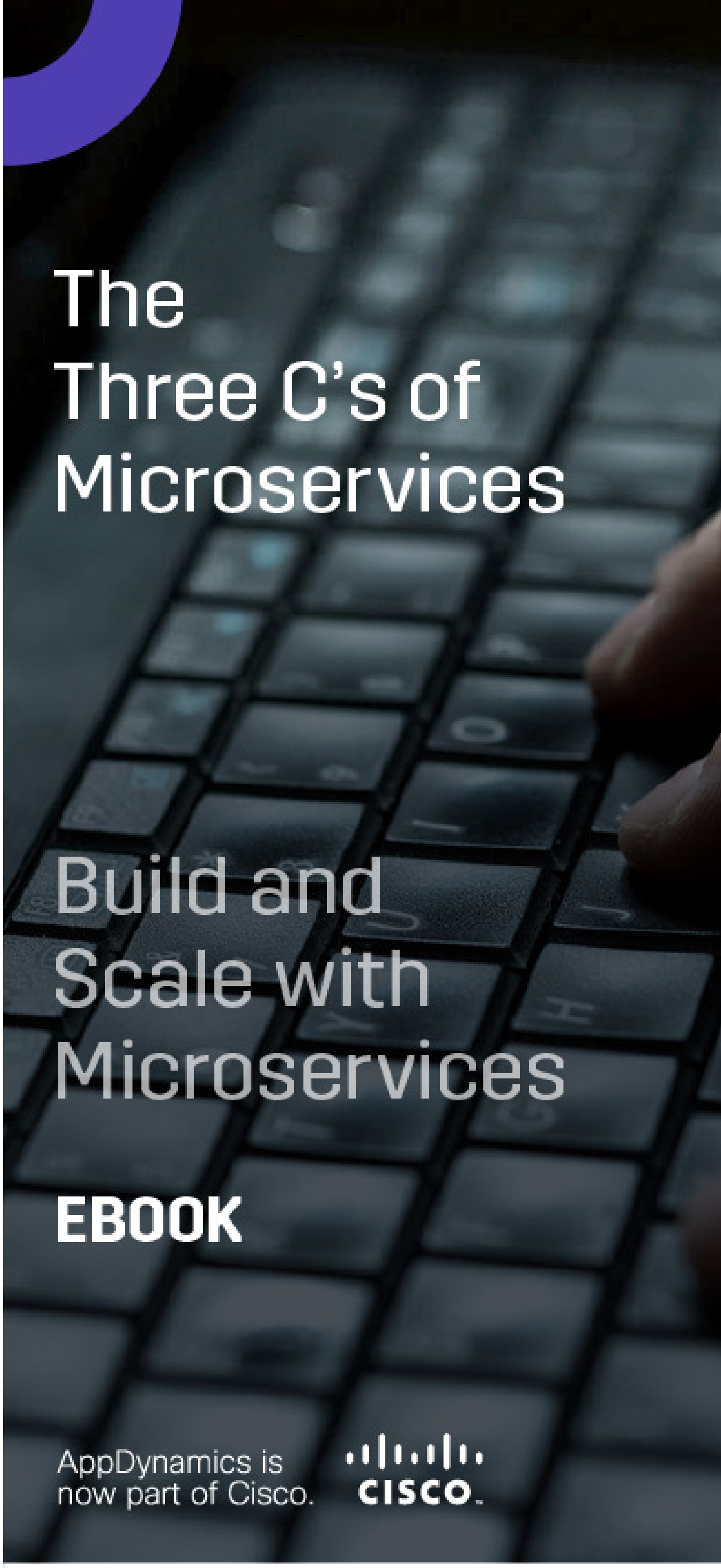
So certainly, to me, there's an anti-pattern in the microservice architecture, the distributed monolith, where you are really releasing multiple services simultaneously because of that. So that's one part which is, sort of, from a logical perspective.

And then, from just sort of nuts-and-bolts technical thing, you can certainly say that when it comes to deployment, our unit of deployment has been getting increasingly lighter and more ephemeral. So, 10 or 15 years ago, if you wanted to deploy something, you had to get a physical machine. And now you just deploy a lambda on AWS — and in such a short amount of time, that's been a radical transformation in how we deploy things. And so that, to me, is one kind of huge trend. And even from a design point of view, there's this common-closure principle that you have to keep in mind.

Schloming: There's a way I like to think about this question that is very complementary with this but from a different perspective, and that is thinking about the trends in terms of how many people you need to accomplish something. If you look at the transition from monolith to microservices through the organizational lens, it is a shift in the division of labor. You are taking the output of a team, an engineering team of thousands of people, and you are fundamentally assembling the output of that work in a different way into a single, coherent whole. Look at 10 years ago, the size of a team it took to deliver a given service. Today, a teenager could do the same thing out of his parents' basement in a weekend, at least close to that.

And so I think that the limit of this really comes down to the point where that team size can effectively stop shrinking. It is how much a single developer can absorb and accomplish, until you throw in something like AI, which I'm sure people are doing now.

Richardson: Can I just respond? One thing that is interesting is I don't know whether the productivity of an individual developer writing code has improved. Like, writing and creating brand-new code. So I look back and, some things have changed. Like machines have gotten faster and bigger, and if we are stuck, there is Google or Stack Overflow. And then there's all of this open-source stuff, so we can quickly assemble a bunch of libraries together, and if we get stuck, we Google the an-


The Three C's of Microservices
Build and Scale with Microservices
EBOOK
AppDynamics is now part of Cisco.
CISCO

swer. But, in terms of writing code from scratch, I feel like it is an individual developer muddling along somehow, scratching their head. And, if that hasn't changed, we have not had a Moore's law for software development in that regard.

Ryan: So if we are in the realm of predictions, I think some of the answers are sitting outside in the vendor booths. More and more of your code is running on the same network, and I'm not meaning only yours, I mean all of you at the same time. You are all putting your code into big cloud vendors; it is much more local with everyone else in this room than it was before. So we have this interesting networking effect. Microservices are not just a way for you to build services. It is also a way for you to consume services that other people have built for you.

When I look out there and I see vendors selling certain types of services, the thing that strikes me is that they're smaller versions of things that bigger vendors used to sell. I look at the APM space when I see that. And you will see the trend continue when there are more micro-vendors; there will be more marketplaces that help you acquire services that can do interesting things. Somebody asked about geolocation. You can buy that as a service. It is a tiny little service; it does very little in terms of an API and a huge amount in terms of the back end. So that is one thing that we might expect to see going forward.

Schloming: I think that those two answers spark a lot. I don't think a developer writes more lines of code, but they are way more productive because they figure out how to assemble a lot of things — and the other things or what Louis just mentioned are examples of that marketplace of other things to assemble.

Q: When I log into an application like Netflix, it is a pretty frictionless user experience. I log in once and I don't get a sense that I'm logging into the microservice for my user profile, customer history, etc. How do you maintain this frictionless UI in microservices architecture? Most of us are writing applications that span multiple services but it is really just one application users are trying to go to. How do I maintain the advantages of a share-nothing architecture where I can deploy independently without dependencies between services yet maintain a user experience that is frictionless, unified, and with a consistent look and feel?

Tangirala: So, there are different tiers in the microservice layer. There is a front-end tier, which takes all the user traffic, and then we have a middle tier and backend tier, which are your membership and all the core services that give that data set to you. And so, in terms of the UI integration, there is a lot of interaction between these services, but at any given time the source of truth is just one service.

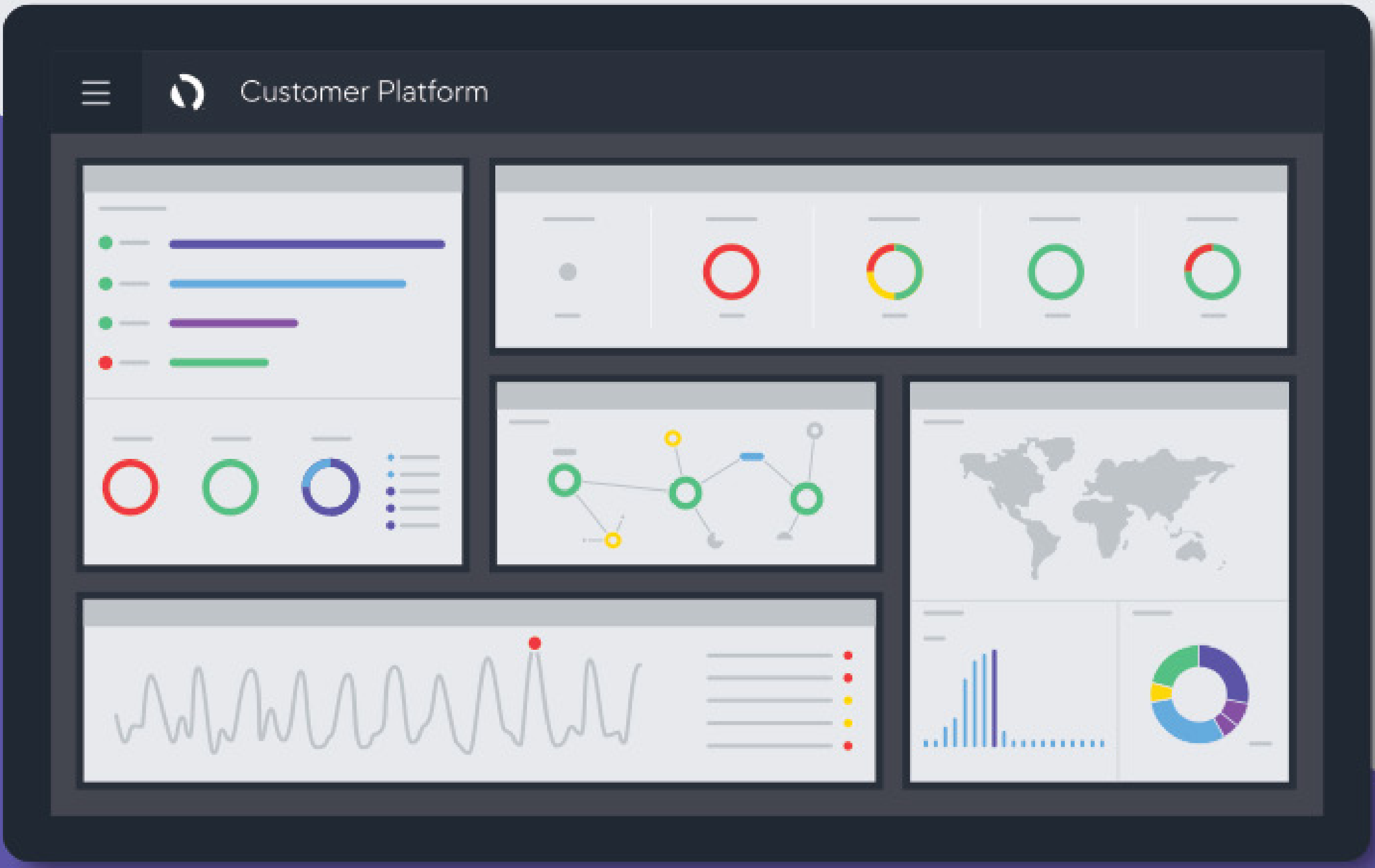
I don't have a lot of insights into the UI layer. But our UI team does a great job in making sure all the interactions between these microservices and the results that they are getting in the UI are seamless. There's a lot of work that goes on behind the scenes, but each microservice is not related to the other. That way, you know which service to call.

Though there's a lot of interaction, you have fallbacks as well. From the UI point of view, you don't see that you are having a degraded experience. If you are not able to get your personalized list of movies to watch from that service, if you cannot go to that service, then they may fall you back to a fallback page. So you might not experience degraded service; you do not think you are not seeing your active list of movies as the service is giving you the fallback experience.

Microservices IQ

Microservices correlates your applications, containers, and cloud infrastructure performance with end-user experience.

Free Trial



- Monitor microservices and cloud-native apps using a single dashboard.
- Utilize machine learning to avoid microservices and containers failure.
- Code level granularity to isolate and fix microservices performance issues.

APPDYNAMICS

Watch presentation online on InfoQ

Managing Data in Microservices

Adapted from a presentation at QCon San Francisco 2017, by Randy Shoup, VP of engineering at Stitch Fix

KEY TAKEAWAYS

Stitch Fix, a clothing retailer, employs nearly as many data scientists as engineers. The data scientists work on algorithms critical to the company's success, and require a substantial amount of data to succeed.

Although microservices may be necessary for achieving a highly scalable solution, do not start with the complexity of a highly distributed system until the company is successful enough that microservices become justified and necessary.

All major companies that are now using microservices, including eBay, Twitter, and Amazon.com, have gone through a migration that started with a monolithic system.

A true microservices platform requires each microservice to be responsible for its own data. Creating separate data stores can be the most challenging aspect of a microservices migration.

The process for separating out a monolithic database involves a repeatable process of isolating each service's data and preventing direct data access from other services.

I'm Randy Shoup, VP of engineering at Stitch Fix, and my background informs the following lessons about managing data in microservices.

Stitch Fix is a clothing retailer in the United States, and we use technology and data science to help customers find the clothing they like. Before Stitch Fix, I was a roving "CTO as a service", and I helped companies discuss technologies and these situations.

Earlier in my career, I was director of engineering at Google for Google App Engine. That is Google's platform as a service, like Heroku, or Cloud Foundry, or something like that. Earlier, I was chief engineer for about six-and-a-half years at eBay, where I helped our teams build multiple generations of search infrastructure. If you have ever gone to eBay and found something that you liked then, great, my team did a good job. And if you didn't find it, well, you know where to put the blame.

Let me start with a little bit about Stitch Fix, because that informs the lessons and the techniques of our breaking monoliths into microservices. Stitch Fix is the reverse of the standard clothing retailer. Rather than shop online or go to a store yourself, what if you had an expert do it for you?

We ask you to fill out a really detailed style profile about yourself, consisting of 60 to 70 questions, which might take you 20 to 30 minutes. We ask your size, height, weight, what styles you like, if you want to flaunt your arms, if you want to hide your hips… — we ask very detailed and personal things. Why? Anybody in your life who knows how to choose clothes for you must know about you. We explicitly ask those things, and use data science to make it happen. As a client, you have five items we deliver to your doorstep, hand-picked for you by one of 3,500 stylists around the country. You keep the things that you like, pay us for those, and return the rest for free.

A couple of things go on behind the scenes among both humans and machines. On the machine side, we look every night at every piece of inventory, reference that against every one of our clients, and compute a predicted probability of purchase. That is, what is the conditional probability that Randy will keep this shirt that we send him. Imagine that there's a 72 percent chance that Randy will keep this shirt, 54 percent chance for these pants, and 47 percent chance for the shoes — and for each of you in the room, the percentages are going to be different. We have machine-learned models that we layer in an ensemble to compute those percentages, which compose a set of personalized algorithmic recommendations for each customer that go to the stylists.

As the stylist is essentially shopping for you, choosing those five items on your behalf, he or she is looking at those algorithmic recommendations and figuring out what to put in the box.

We need the humans to put together an outfit, which the machines are currently not able to do. Sometimes, the human will answer a request such as "I'm going to Manhattan for an evening wedding, so send me something appropriate." The machine doesn't know what to do with that, but the humans know things that the machines don't.

All of this requires a ton of data. Interestingly and, I believe, uniquely, Stitch Fix has a one-to-one ratio between data science and engineering. We have more than a hundred software engineers in the team that I work on and roughly 80 data scientists and algorithm developers that are doing all the data science. To my knowledge, this is a unique ratio in the industry. I don't know any other company on the planet that has this kind of near one-to-one ratio.

What do we do with all of those data scientists? It turns out, if you are smart, it pays off.

We apply the same techniques to what clothes we're going to buy. We make algorithmic recommendations to the buyers and they figure out that, okay, next season, we're going to buy more white denim or cold shoulders are out or Capri pants are in next.

We use data analysis for inventory management: what do we keep in what warehouses and so on. We use it to optimize logistics and selection of shipping carriers so that the goods arrive on your doorstep on the date you want, at minimal cost to us. And we do some standard things, like demand prediction.

We are a physical business: we physically buy the clothes, put them in warehouses, and ship them to you. Unlike eBay and Google and a bunch of virtual businesses, if we guess wrong about demand, if demand is double what we expect, that is not a wonderful thing that we celebrate. That's a disaster because it means that we can only serve half of the people well. If we have double the number of clients, we should have double the number of warehouses, stylists, employees, and that kind of stuff. It is very important for us to get these things right.

Again, the general model here is that we use humans for what the humans do best and machines for what the machines do best.

When you design a system at this scale, as I hope you do, you have a bunch of goals. You want to make sure that the development teams can continue to move forward independently and at a quick pace — that's what I call "feature velocity". We want scalability, so that as our business grows, we want the infrastructure to grow with it. We want the components to scale to load, to scale to the demands that we put on them. Also, we want those components to be resilient, so we want the failures to be isolated and not cascade through the infrastructure.

High-performing organizations with these kinds of requirements have some things to do. *The DevOps Handbook* features research from Gene Kim, Nicole Forsgren, and others into the difference between high-performing organizations and lower-performing ones. Higher-performing organizations both move faster and are more stable. You don't have to make a choice between speed and stability — you can have both.

The higher-performing organizations are doing multiple deploys a day, versus maybe one per month, and have a latency of less than an hour between committing code to the source control and to deployment, while in other organizations that might take a week. That's the speed side.

On the stability side, high-performing organizations recover from failure in an hour, versus maybe a day in a lower-performing organization. And the rate of failures is lower. The frequency of a high-performing organization deploying, having it not go well, and having to roll back the deployment approaches zero, but slower organizations might have to do this half the time. This is a big difference.

It is not just the speed and the stability. It is not just the technical metrics. The higher-performing organizations are two-and-a-half times more likely to exceed business goals like profitability, market share, and productivity. So this stuff doesn't just matter to engineers, it matters to business people.

Evolving to microservices

One of the things that I got asked a lot when I was doing my roving CTO-as-a-service gig was "Hey, Randy, you worked at Google and eBay — tell us how you did it."

I would answer, "I promise to tell you, and you have to promise not to do those things, yet." I said that not because I wanted to hold onto the secrets of Google and eBay, but because a 15,000-person engineering team like Google's has a different set of problems than five people in a startup that sit around a conference table. That is three orders of magnitude different, and there will be different solutions at different scales for different companies.

That said, I love to tell how the companies we have heard of have evolved to microservices — not started with microservices, but evolved there over time.

eBay

eBay is now on its fifth complete rewrite of its infrastructure. It started out as a monolithic PERL application in 1995, when the founder wanted to play with this thing called the Web and so spent the three-day Labor Day weekend building this thing that ultimately became eBay.

The next generation was a monolithic C++ application that, at its worst, was 3.4 million lines of code in a single DLL. They were hitting compiler limits on the number of methods per class, which is 16,000. I'm sure many people think that they have a monolith, but few have one worse than that.

The third generation was a rewrite in Java — but we cannot call that microservices; it was mini-applications. They turned the site into 220 different Java applications. One was for the search part, one for the buying part… 220 applications. The current instance of eBay is fairly characterized as a polyglot set of microservices.

Twitter

Twitter has gone through a similar evolution, and is on roughly its third generation. It started as a Rails application, nicknamed the Monorail. The second generation pulled the front end out into

JavaScript and the back end into services written in Scala, because Twitter was an early adopter. We can currently characterize Twitter as a polyglot set of microservices.

Amazon.com

Amazon.com has gone through a similar evolution, although not as clean in the generations. It began as a monolithic C++ and Perl application, of which we can still see evidence in product pages. The "obidos" we sometimes see in an Amazon.com URL was the code name of the original Amazon.com application. Obidos is a place in Brazil, on the Amazon, which is why it was named that way.

Amazon.com rewrote everything from 2000 to 2005 in a service-oriented architecture. The services were mostly written in Java and Scala. During this period, Amazon.com was not doing particularly well as a business. But Jeff Bezos kept the faith and forced (or strongly encouraged) everyone in the company to rebuild everything in a service-oriented architecture. And now it's fair to categorize Amazon.com as a polyglot set of microservices.

These stories all follow a common pattern. No one starts with microservices. But, past a certain scale (a scale that maybe only .1 percent of us is going to get to), everybody ends up with something we can call microservices.

I like to say that if you don't end up regretting your early technology decisions, you probably over-engineered.

Why do I say that?

Imagine an eBay competitor or Amazon.com competitor in 1995. This company, instead of finding a product market fit, a business model, and things that people are going to pay for, has built a distributed system they are going to need in five years. There is a reason we have not heard of that company.

Again, think about where you are in your business, where you are in your team size. The solutions for Amazon.com, Google, and Netflix are not necessarily the solutions for you when you are a small startup.

Microservices

I like to define the micro in microservices as not about the number of lines of code but about the scope of the interface.

A microservice has a single purpose and a simple, well-defined interface, and it is modular and independent. The critical thing to focus on and explore the implications of is that effective microservices, as Amazon.com found, have isolated persistence. In other words, a microservice should not be sharing data with other services.

For a microservice to reliably execute business logic and to guarantee invariance, we cannot have people reading and writing the data behind its back. eBay discovered this the other way. eBay spent a lot of effort with some very smart people to build a service layer in 2008, but it was not successful. Although the services were extremely well built and the interfaces were quite good and orthogonal — they spent a lot of time thinking about it — underneath them was a sea of shared databases that were also directly available to the applications. Nobody had to use the service layer in order to do their job, so they didn't.

No one starts with microservices. But, past a certain scale, everyone ends up with microservices.

If you don't end up regretting your early technology decisions, you probably over-engineered.

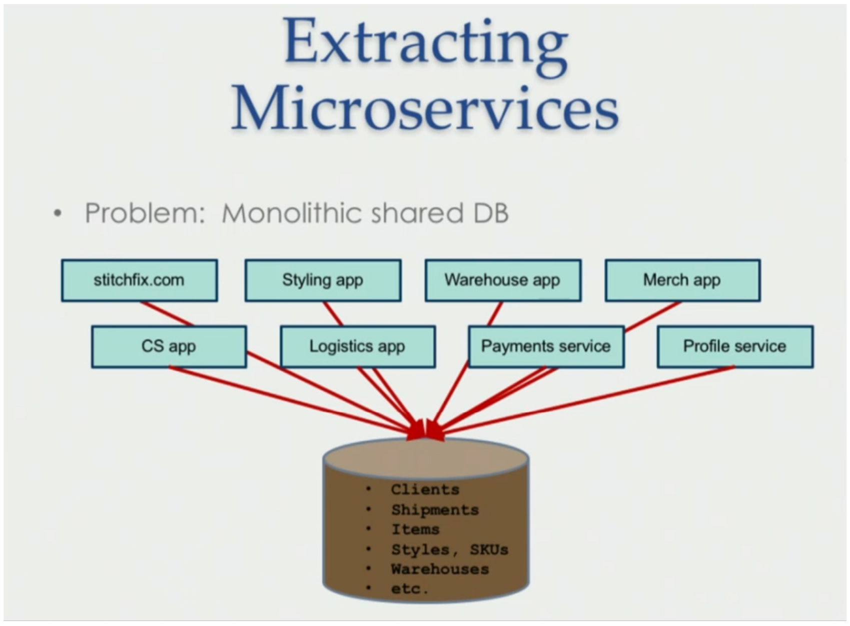


Figure 1: Stitch Fix's Monolithic, shared database.

At Stitch Fix, we are on our own journey. We did not build a monolithic application, but our version of the monolith problem is the monolithic database we built.

We are breaking up our monolithic database and extracting services from it but there are some great things that we would like to retain.

Figure 1 shows a simplified view of our situation. We have way more than this number of apps, but there are only so many things that fit in one image.

We essentially have a shared database that includes everything that is interesting about Stitch Fix. This includes clients, the boxes that we ship, the items that we put into the boxes, metadata about the inventory like styles and SKUs, information about the warehouses, times about 175 different tables. We have on the order of 70 or 80 different applications and services that use the same database for their work. That is the problem. That shared database is a coupling point for the teams, causing them to be interdependent as opposed to independent. It is a single point of failure and a performance bottleneck.

Our plan is to decouple applications and services from the shared database. There is a lot of work here.

Figure 2 shows the steps taken to break up shared database. Image A is the starting point. The real diagram would be too full of boxes and lines, so let's imagine that there are only three tables and two applications. The first thing that we're going to do is build a service that represents, in this example, client information (B). This will be one of the microservices, with a well-defined interface. We negotiated that interface with the consumers of that service before we created the service.

Next, we point the applications to read from the service instead of using the shared database to read from the table (C). The hardest part is moving the lines. I do not mean to trivialize, but an image simply cannot show how hard it is to do that. After we do that, callers no longer connect directly to the database but will instead go through the service. Then we move the table from the shared database and put it in an isolated private database that is only associated with the microservice (D). There's a lot of hard work involved, and this is the pattern.

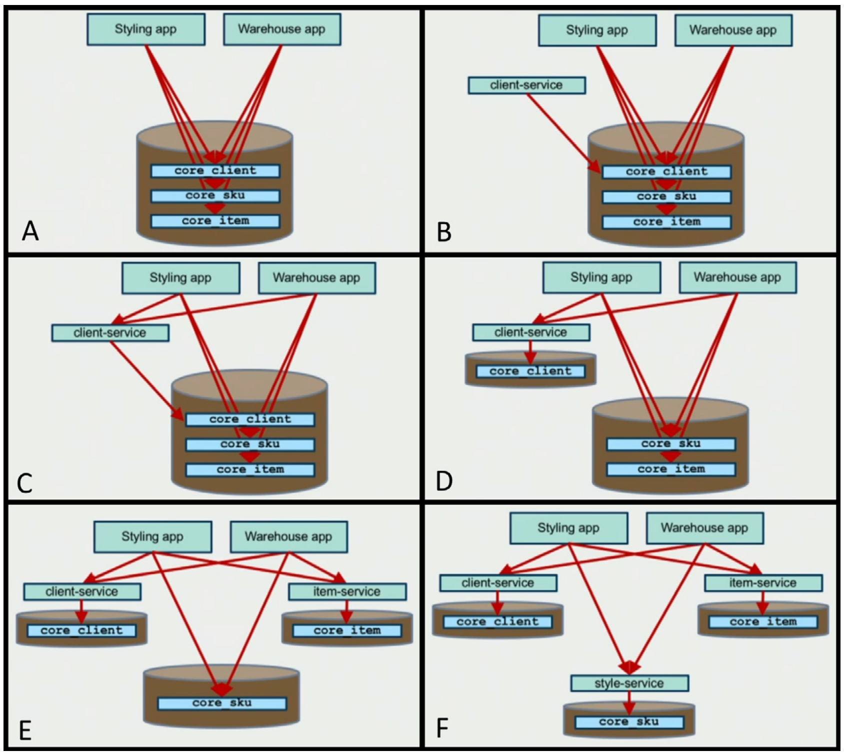


Figure 2: Breaking up the shared database.

The next task is to do the same thing for item information. We create an item service, and have the applications use the service instead of the table (E). Then we extract the table and make it a private database of the service. We then do the same thing for SKUs or styles, and we keep rinsing and repeating (F). By the end, the boundary of each microservice surrounds both its application box and its database, such as the paired client-service and "core client" database (F).

We have divided the monolithic database with everything in there so that each microservice has its own persistence. But there are a lot of things that we like about of the monolithic database, and I don't want to give them up. These include easily sharing data between different services and applications, being able to easily do joins across different tables, and transactions. I want to be able to do operations that span multiple entities as a single atomic unit. These are all common aspects of monolithic databases.

Events

There are various database features that we can and cannot keep through the next part of the migration, but there are workarounds for those we can't have. Before going into that, I need to point out an architectural building block that perhaps you know about but don't appreciate as much as you should — namely, events. Wikipedia defines an event as a significant change in state or a statement that something interesting has occurred.

In a traditional three-tier system, there's the presentation tier that the users or clients use, the application tier that represents stateless business logic, and the persistence tier that is backed by a relational database. But, as architects, we are missing a fundamental building block that represents a state change, and that is what I will call an event. Because events are typically asynchronous, maybe I will produce an event to which nobody is yet listening, maybe only one other consumer within the system is listening to it, or maybe many consumers are going to subscribe to it.

Having motivated events to a first-class construct in our architecture, we will now apply events to microservices.

A microservices interface includes the front door, right? It obviously includes the synchronous request and response. This is typically HTTP, maybe JSON, maybe gRPC or something like that, but it clearly includes an access point. What is less obvious — and I hope I can convince you that this is true — is that it includes all of the events that the service produces, all of the events that the service consumes, and any other way to get data into and out of that service. Doing bulk reads out of the service for analytic purposes or bulk writes into the service for uploads are all part of the interface of the service. Simply put, I assert that the interface of a service includes any mechanism that gets data into or out of it.

Now that we have events in our toolbox, we will start to use events as a tool in solving those problems of shared data, of joins, and of transactions. That brings us to the problem of shared data. In a monolithic database, it is easy to leverage shared data. We point the applications at this shared table and we are all good. But where does shared data go in a microservices world?

Well, we have a couple of different options — but I will first give you a tool or a phrase to use when you discuss this. The principle, or that phrase, is "single system of record". If there's data for a customer, an item, or a box that is of interest in your system, there should be one and only one service that is the canonical system of record for that data. There should be only one place in the system where that service owns the customer, owns the item, or owns the box. There are going to be many representations of customer/item/etc. around (there certainly are at Stitch Fix), but every other copy in the system must be a read-only, non-authoritative cache of that system of record.

Let that sink in: read only and non-authoritative. Don't modify the customer record anywhere and expect it to stick around in some other system. If we want to modify that data, we need to go to the system of record, which is the only place that can currently tell us, to the millisecond, what the customer is doing.

That's the idea of the system of record, and there are a couple of different techniques to use in this approach to sharing data. The first is the most obvious and most simple: synchronously look it up from that system of record.

Consider a fulfillment service at Stitch Fix. We are going to ship a thing to a customer's physical address. There's a customer service that owns the customer data, one piece of which is the customer's address. One solution is for the fulfillment service to call the customer service and look up the address. There's nothing wrong with this approach; this is a perfectly legitimate way to do it. But sometimes this isn't right. Maybe we do not want everything to be coupled on the customer service. Maybe the fulfillment service, or its equivalent, is pounding the customer service so often that it impedes performance.

Another solution involves the combination of an asynchronous event with a local cache. The customer service is still going to own that representation of the customer, but when the customer data changes (the customer address, say), the customer service is going to send an update event. When the address changes, the fulfillment service will listen to that event and locally cache the address, then the fulfillment center will send the box on its merry way.

The caching within the fulfillment service has other nice properties. If the customer service does not retain a history of address changes, we can remember that in the fulfillment service. This happens at scale: customers may change addresses between the time that they start an order and the time that we ship it. We want to make sure that we send it to the right place.

Joins

It is really easy to join tables in a monolithic database. We simply add another table to the FROM clause in a SQL statement and we're all good. This works great when everything sits in one big, monolithic database, but it does not work in a SQL statement if A and B are two separate services. Once we split the data across microservices, the joins, conceptually, are a lot harder to do.

We always have architecture choices, and there is more than one way to handle joins. The first option is to join in the client. Have whatever is interested in the A and the B do the join. In this particular example, let's imagine that we are producing an order history. When a customer comes to Stitch Fix to see the history of the boxes that we've sent them, we might be able to provide that page in this way. We might have the order-history page call the customer service to get the current version of the customer's information — maybe her name, her address, and how many things we have sent her. Then, it can go to the order service to get details about all of her orders. It gets a single customer from the customer service then will query for the orders that match that customer on the order service.

This is a pattern used on basically every webpage that does not get all of its data from one service. Again, this is a totally legitimate solution to this problem. We use it all the time at Stitch Fix, and I'm sure you use it all over the place in your applications as well.

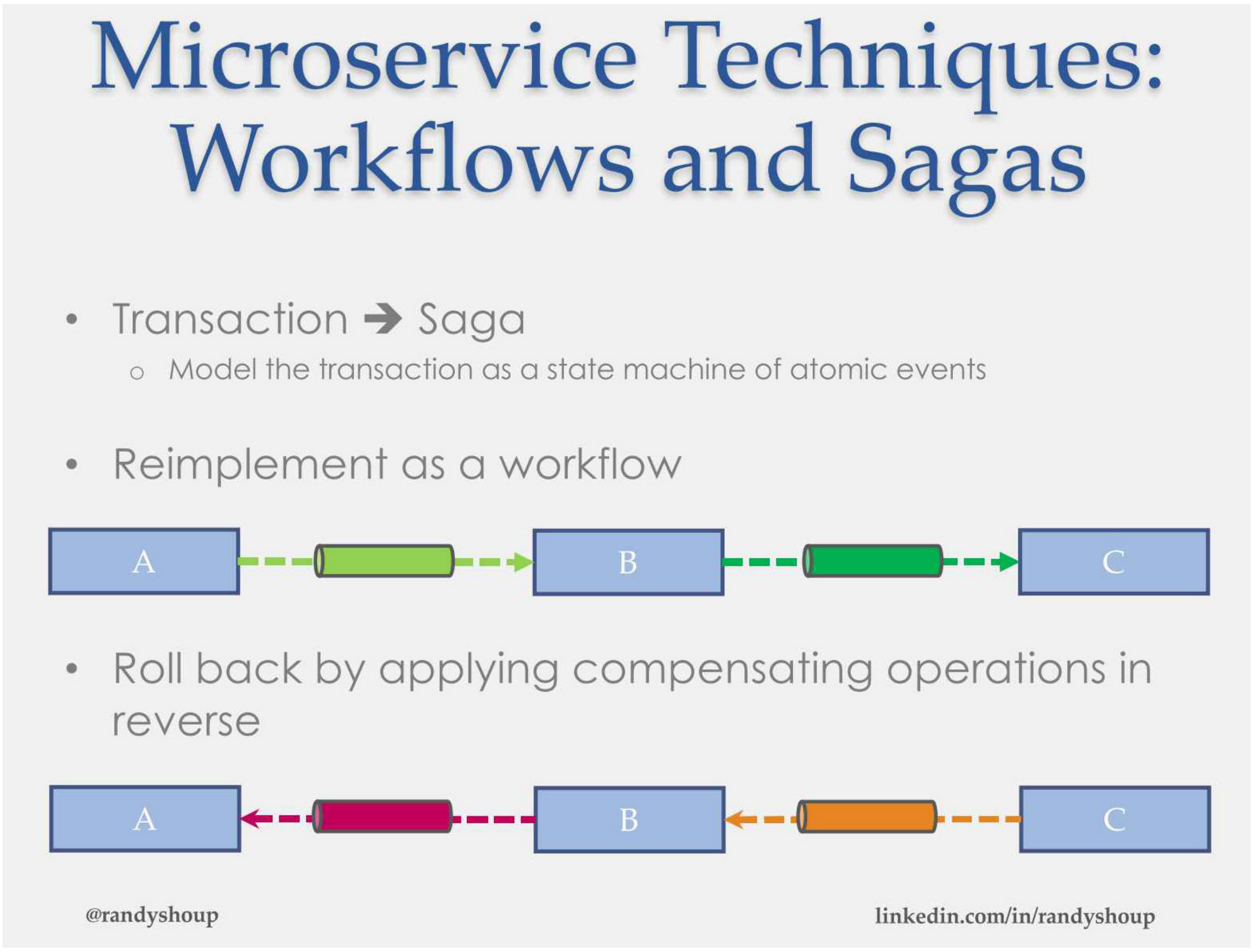
But let's imagine that this doesn't work, whether for reasons of performance or reliability or maybe we're querying the order service too much.

For approach number two, we create a service that does what I like to call, in database terminology, "materializing the view". Imagine we are trying to produce an item-feedback service. At Stitch Fix, we send boxes out, and people keep some of the things that we send and return some. We want to know why, and we want to remember which things are returned and which are kept. This is something that we want to remember using an item-feedback service. Maybe we have 1,000 or 10,000 units of a particular shirt and we want to remember all customer feedback about that shirt every time we sent it. Multiply that effort by the tens of thousands of pieces of inventory that we might have.

To do this, we are going to have an item service, which is going to represent the metadata about this shirt. The item-feedback service is going to listen to events from the item service, such as new items, items that are gone, and changes to the metadata if that is interesting. It will also listen to events from the order service. Every piece of feedback about an order should generate an event — or, since we send five items in a box, possibly five events. The item-feedback service is listening to those events and then materializing the join. In other words, it's remembering all the feedback that we get for every item in one cached place. A fancier way to say that is that it maintains a denormalized join of items and orders together in its own local storage.

Many common systems do this all the time, and we don't even think that they are doing it. For example, any sort of enterprise-grade (i.e., we pay for it) database system has a concept of a materialized view. Oracle has it, SQL Server has it, and a bunch of enterprise-class databases have a concept of materializing a view.

Most NoSQL systems work in this way. Any of the Dynamo-inspired data stores, like DynamoDB from Amazon, Cassandra, React, or Voldemort, all which come from a NoSQL tradition, force us to do it up front. Relational databases are optimized for easy writes — we write to individual records or to individual tables. On the read side, we put it all together. Most NoSQL systems are the other way around. The tables that we store are already the queries that we wanted to ask. Instead of writing to an individual sub-table at write time, we are writing five times to all of the stored queries that we want to read from. Every NoSQL system is forcing us up front to do this sort of materialized join.



Microservice Techniques: Workflows and Sagas

- Transaction ➔ Saga
  - Model the transaction as a state machine of atomic events
- Reimplement as a workflow
- Roll back by applying compensating operations in reverse

@randyshoup linkedin.com/in/randyshoup

Figure 3: Workflows and sagas.

Every search engine that we use almost certainly is doing some form of joining one particular entity with another particular entity. Every analytical system on the planet is joining lots of different pieces of data, because that is what analytical systems are about.

I hope this technique now sounds a little bit more familiar.

Transactions

The wonderful thing about relational databases is this concept of a transaction. In a relational database, a single transaction embodies the ACID properties: it is atomic, consistent, isolated, and durable. We can do that in a monolithic database. That's one of the wonderful things about having THE database in our system. It is easy to have a transaction cross multiple entities. In our SQL statement, we begin the transaction, do our inserts and updates, then commit and that either all happens or it doesn't happen at all.

Splitting data across services makes transactions hard. I will even replace "hard" with "impossible". How do I know it's impossible? There are techniques known in the database community for doing distributed transactions, like two-phased commit, but nobody does them in practice. As evidence of that fact, consider that no cloud service on the planet implements a distributed transaction. Why? Because it is a scalability killer.

So, we can't have a transaction — but here is what we can do. We turn a transaction where we want to update A, B, and C, all together as a unit or not at all, into a saga. To create a saga, we model the transaction as a state machine of individual atomic events. Figure 3 may help clarify this. We re-implement that idea of updating A, updating B, and updating C as a workflow. Updating the A side produces an event that is consumed by the B service. The B service does its thing and produces an event that is consumed by the C service. At the end of all of this, at the end of the state machine, we are in a terminal state where A and B and C are all updated.

Now, let's imagine something goes wrong. We roll back by applying compensating operations in the reverse order. We undo the things we were doing in C, which produces one or several events, and then we undo the set of things that we did in the B service, which produces one or several events, and then we undo the things that we did in A. This is the core concept of sagas, and there's a lot of great detail behind it. If you want to know more about sagas, I highly recommend Chris Richardson's QCon presentation, Data Consistency in Microservices Using Sagas.

As with materializing the view, many systems that we use every day work in exactly this way. Consider a payment-processing system. If you want to pay me with a credit card, I would like to see the money get sucked out of your account and magically end up in my wallet in one unit of work. But that is not what actually happens. There are tons of things behind the scenes that involve payment processors and talking to the different banks and all of this financial magic.

In the canonical example of when we would use transactions, we would debit something from Justin's account and add it to Randy's account. No financial system on the planet actually works like that. Instead, every financial system implements it as a workflow. First, money gets taken out of Justin's account, and it lives in the bank for several days. It lives in the bank longer than I would like, but it ultimately does end up in my account.

As another example, consider expense approvals. Probably everybody has to get expenses approved after a conference. And that does not happen immediately. You submit your expenses to your manager, and she approves it, and it goes to her boss, and she approves it... all the way up. And then your reimbursement follows a payment-processing workflow, where ultimately the money goes into your account or pocket. You would prefer this to be a single unit, but it actually happens as a workflow. Any multi-step workflow is like this.

If you write code for a living, consider as a final example what would happen if your code were deployed to production as soon as you hit return on your IDE. Nobody does that. That is not an atomic transaction, nor should it be. In a continuous-delivery pipeline, when I say commit, it does a bunch of stuff, the end result of which is, hopefully, deployed to production. That's what the high-performing organizations are doing. But it does not happen atomically. Again, it's a state machine: this step happens, then this happens, then this happens, and if something goes wrong along the way, we back it out. This should sound familiar. Stuff we use every day behaves like this, which means there is nothing wrong with using this technique in the services we build.

To wrap up, we have explored how to use events as tools in our architectural toolbox. We've shown how we can use events to share data between different components in our system. We have figured out how to use events to help us implement joins. And we have figured out how to use events to help us do transactions.

At Stitch Fix, we are on our own journey. We did not build a mono-lithic application, but our version of the monolith problem is the monolithic database we built.

We are breaking up our monolithic database and extracting services from it but there are some great things that we would like to retain.