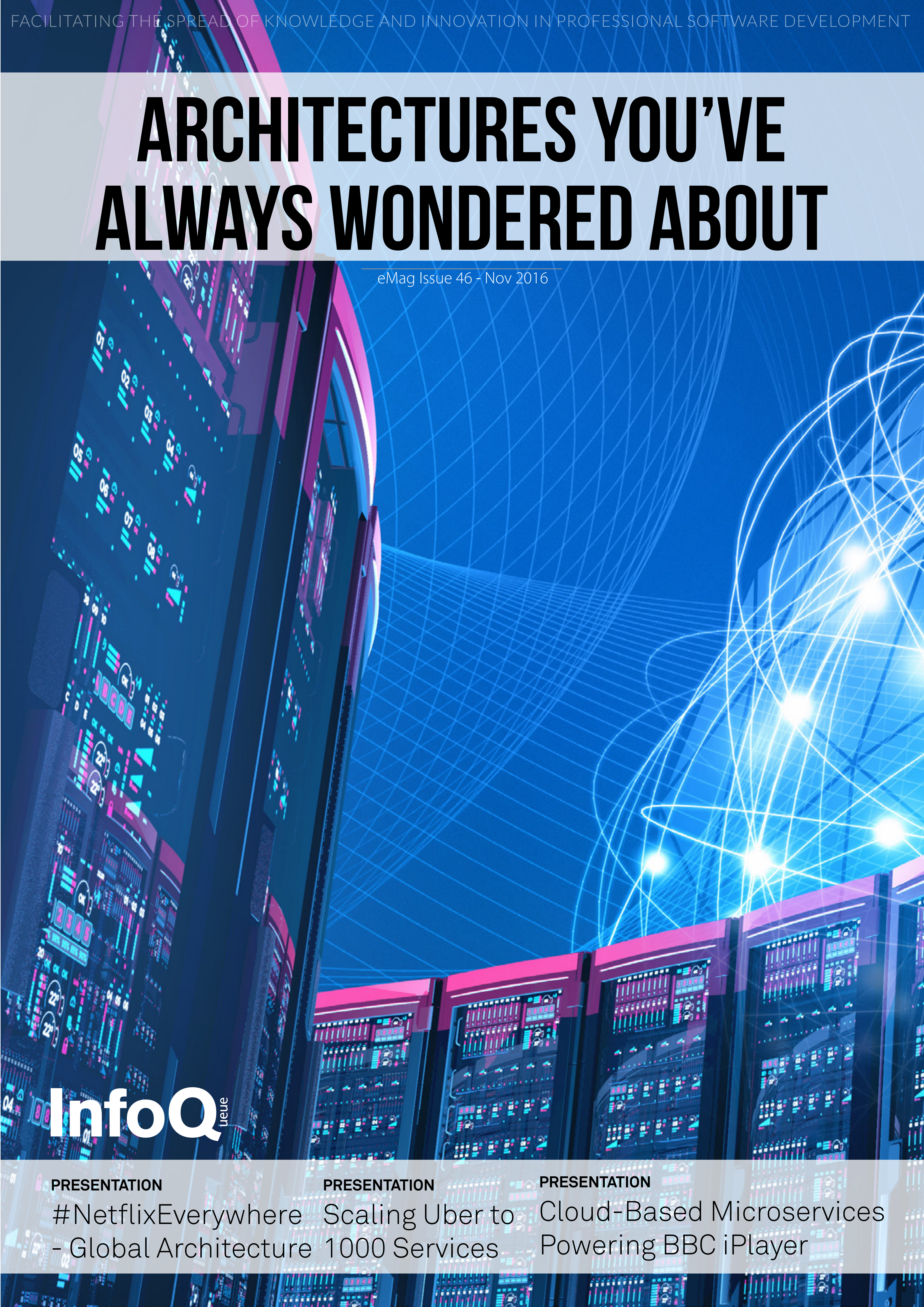
FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

ARCHITECTURES YOU'VE ALWAYS WONDERED ABOUT

eMag Issue 46 - Nov 2016



InfoQ queue

PRESENTATION
#NetflixEverywhere - Global Architecture

PRESENTATION
Scaling Uber to 1000 Services

PRESENTATION
Cloud-Based Microservices Powering BBC iPlayer

ARCHITECTURES YOU'VE ALWAYS WONDERED ABOUT
InfoQ
PRESENTATION
#NetflixEverywhere - Global Architecture
PRESENTATION
Scaling Uber to 1000 Services
PRESENTATION
Cloud-Based Microservices Powering BBC iPlayer

#NetflixEverywhere - Global Architecture

Josh Evans discusses architectural patterns used by Netflix to enable seamless, multi-region traffic management, reliable, fast data propagation, and efficient service infrastructure.

Cloud-Based Microservices Powering BBC iPlayer

Stephen Godwin describes how the BBC integrated its broadcast systems with AWS, how Video Factory is built around a microservices architecture that uses both REST and SQS.

Scaling Uber to 1000 Services

Matt Ranney talks about Uber's growth and how they've embraced microservices. This has led to an explosion of new services, crossing over 1,000 production services in early March 2016.

The Architecture That Helps Stripe Move Faster

Evan Broder talks about how Stripe has designed the systems to speed up the development process and how the software infrastructure in their API enables the next tech companies to build faster.

The Netflix API Platform for Server-Side Scripting

Katharina Probst talks about the situations in which server-side scripting is a good solution for applications. She describes Netflix's first approach, which uses Groovy scripts.

THOMAS BETTS is a senior software engineer at Nordstrom with almost 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 finance, health care, defense and travel. Thomas lives in Denver with his wife and son, and they love hiking and otherwise exploring beautiful Colorado

A LETTER FROM THE EDITOR

What lessons can be learned from the architects who work on successful, large-scale systems such as those at Netflix and Uber? How can Stripe and the BBC make major changes without disrupting their existing customers?

This eMag takes a look back at five of the most popular presentations from the Architectures You've Always Wondered About track at QCons in New York, London and San Francisco.

All the companies featured have large, cloud-based, microservice architectures, which probably comes as no surprise. While the stories told may sound similar, each presenter adds new insight into the biggest challenges they face, and how to achieve success.

One common theme is that adding capacity, functionality and resiliency are not free. The success of these systems depends on monitoring, tracing and logging tools tailored for a distributed system.

Josh Evans tells how Netflix always plans for failure, and tries to never fail the same way twice. This philosophy has helped them grow to a truly global infrastructure, supporting millions of customers and devices around the world.

Matt Ranney shares what he wishes he knew before scaling Uber to over 1,000 services. Chief among them is to look for the trade-offs which are everywhere. Microservices can provide agility and reliability, but with significant operational complexity.

The BBC needed to completely rewrite the content processing system behind their iPlayer. Stephen Goodwin describes the move from an on-premise monolith to microservices and cloud storage. The upgrade was successful, in part because the new architecture helped focus on building small, critical pieces of functionality.

Stripe has a similar approach, solving large problems by breaking them down and working incrementally. Evan Broder shares stories of three major projects, covering the evolution of the Stripe API, a rewrite of key PCI compliance software, and a migration between AWS data centers.

Coming back to Netflix, Katharina Probst discusses the importance of non-functional requirements when implementing a new system using containers to isolate server-side scripts used by hundreds of different client devices calling the Netflix API.

If you're currently using microservices successfully, and wondering what still lies ahead, or if you're just considering breaking up a monolith, these experts can provide valuable wisdom learned from maintaining and upgrading complex systems.

Watch online on InfoQ

#NetflixEverywhere - Global Architecture

Josh Evans is Director of Operations Engineering at Netflix, with experience in e-commerce, playback control services, infrastructure, tools, testing, and operations. Evans is a proponent of operational excellence - the continuous improvement of quality of customer experience and engineering velocity. For the past two years Evans has led Operations Engineering, an organization that creates, integrates, and evangelizes proven technical solutions and practices like continuous delivery, real-time analytics, and chaos engineering in order to achieve operational excellence at scale.

Adapted from a presentation by Josh Evans, Director of Operations Engineering at Netflix, at QCon London 2015

Achieving a global architecture platform, with extremely high levels of availability and performance, was an audacious goal for Netflix. Along the way, they suffered some very public outages. The team adopted a failure-driven approach for upgrading and migrating their platform, trying to ensure they never failed the same way twice.

One of the most notable outages occurred on December 24th, 2012, when the Netflix service was essentially down for almost 24 hours. In this case, the root cause was identified as "an ELB control plane issue" in Amazon's US-EAST-1 region where Netflix servers were located. A maintenance process was inadvertently run against the production ELB state data, thereby affecting Netflix and many other AWS customers. Another outage, on February 3rd, 2015, was caused when Netflix intentionally deployed configuration, but experienced some unexpected consequences.

The lesson is failure is inevitable, and whether self-induced or caused by an underlying platform you're running on, assigning blame is not helpful. The best way to deal with failure is to properly identify and address the root cause, so each failure mode only occurs once. This approach allowed Netflix to build a robust,

KEY TAKEAWAYS

Never fail the same way twice. Analyze root cause of failures and make strategic decisions to avoid them in the future.

Adding resiliency takes many forms. For Netflix, this meant adding redundancy from a second data center up through multiple global AWS regions.

Identify and invest in your architectural pillars.

Think globally, act locally. As small changes are made, always keep the final, larger goal in mind.

global platform, in a straightforward and reasonably fast manner.

Netflix is obsessed with striving for global ubiquity, in terms of devices as well as geography. When Netflix launched their streaming service in 2007, they chose the most ubiquitous platform available at the time, Windows. This has evolved to include set-top devices, smart TVs, video game consoles, and mobile devices. Since 2010, when Netflix became available in Canada, they've continued to expand across the globe to Latin America, Western Europe and APAC, currently reaching 75 million customers.

Early History

In August 2008, when Netflix was still primarily a DVD-by-mail service, a firmware update caused customer data corruption, and a three-day delay in service. At the time, Netflix had a single data center, and the failure was a clear indication that they needed a second data center.

Building a second data center was a challenge because Netflix has always been very focused on improving their core services, whether DVD or streaming. Racking and stacking servers in data centers was not adding differential value to the business. This led to embracing Amazon Web Services to achieve better scale and elasticity, as well as flexibility for engineers to experiment in the environment. Although located in only US-EAST-1 to begin with, Amazon's global footprint would provide long-term benefits for expansion.

The initial move to the cloud environment was only the first step in Netflix's global evolution. Over time, significant changes can be seen in four architectural pillars: microservices, database, cache, and traffic.

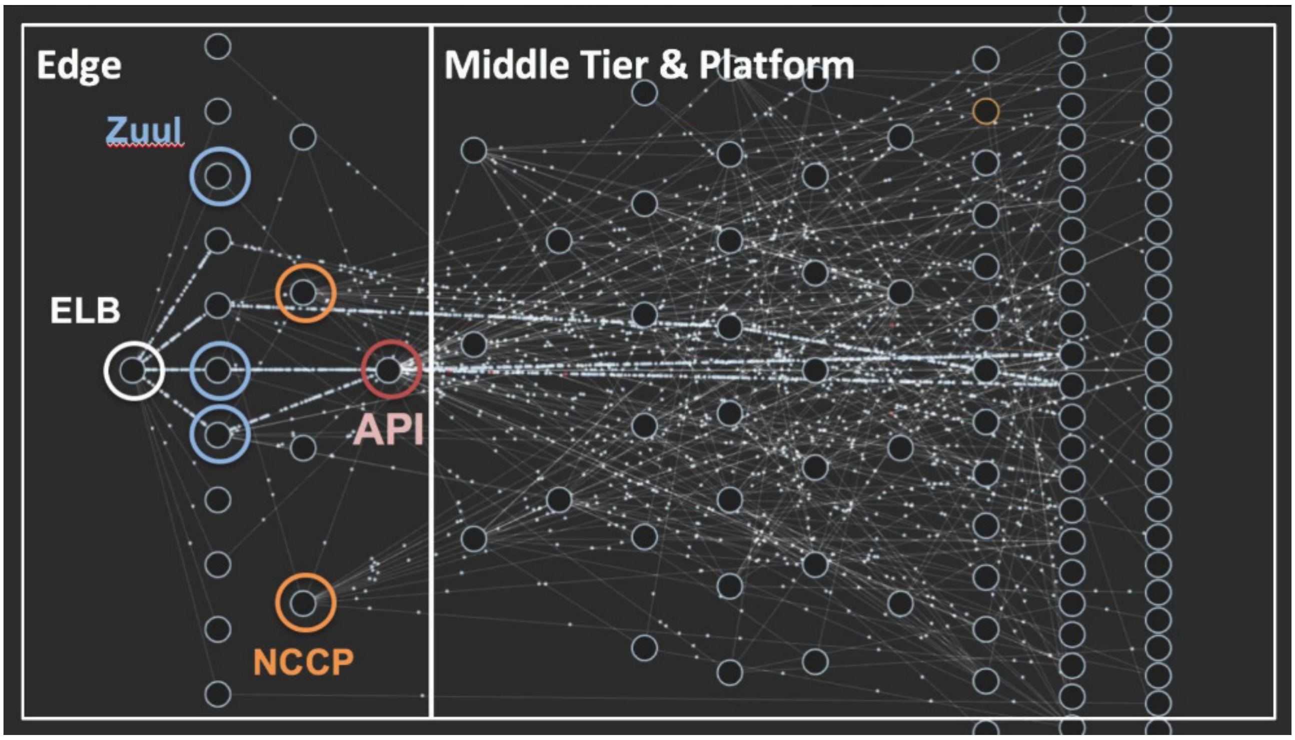
Microservices

With the move to AWS, Netflix moved to microservices, although they were simply referred to as distributed systems at the time. Departing from monolithic systems meant teams could work more independently, better understand their service, and more easily fix problems.

A visualization of the Netflix ecosystem shows the complexity of all the microservices which make up the middle tier and platform services.

The challenge of microservices is responding to failures. Most importantly, the failure of a non-critical microservice should not cause a catastrophic failure of the system. Netflix has seen this scenario occur many times, which led to the development of many technologies, including Hystrix, to provide structured fallbacks and timeouts. Fallbacks provide a graceful degradation in service rather than failing hard, such as providing most popular titles when recommendations are not available.

Chaos principles are necessary for testing and enforcing the isolation of failures throughout the system. Chaos Monkey ensures an individual microservice or auto-scaling instance failure does not take down the cluster. The Failure Injection Testing (FIT) framework performs the same function with entire microservices and clusters being removed from the ecosystem. Finally, Chaos Kong can move traffic between regions, and is used for both simulation purposes and in ▶



response to real-time availability issues.

Database & Cache

The need to scale globally meant not using relational databases. SimpleDB was a NoSQL solution that provided persistence and other benefits, but was quickly overwhelmed by Netflix's load. Caching was necessary to shield SimpleDB from that load. Experiments using Memcached led to the development of EVCache (Ephemeral Volatile memCache), a sharded, clustered implementation of Memcached, optimized for how Netflix uses the cloud.

An individual EC2 instance contains a Memcached instance, running Prana sidecar to connect to Netflix's discovery service, allowing applications to find and directly access a needed shard. Cache volatility is carefully managed, with TTLs on data helping with drift, and an LRU mechanism to evict least recently used records as the cache fills up.

Favoring local reads gives the lowest possible latency, around one to five milliseconds, and also reduces the cost of data transfer between availability zones. To facilitate the local reads, writes must be sent across the various zones, so some data transfer costs are necessary.

The EVCache layer effectively shielded SimpleDB, and exists in front of the microservices. By default, an application will call EVCache first, and only call the microservice directly when a cache miss occurs. The microservice calls into SimpleDB, returns the result and populates the cache. This ensures that a second request, even within a few milliseconds, will utilize the cache, leading to a 99% cache hit ratio.

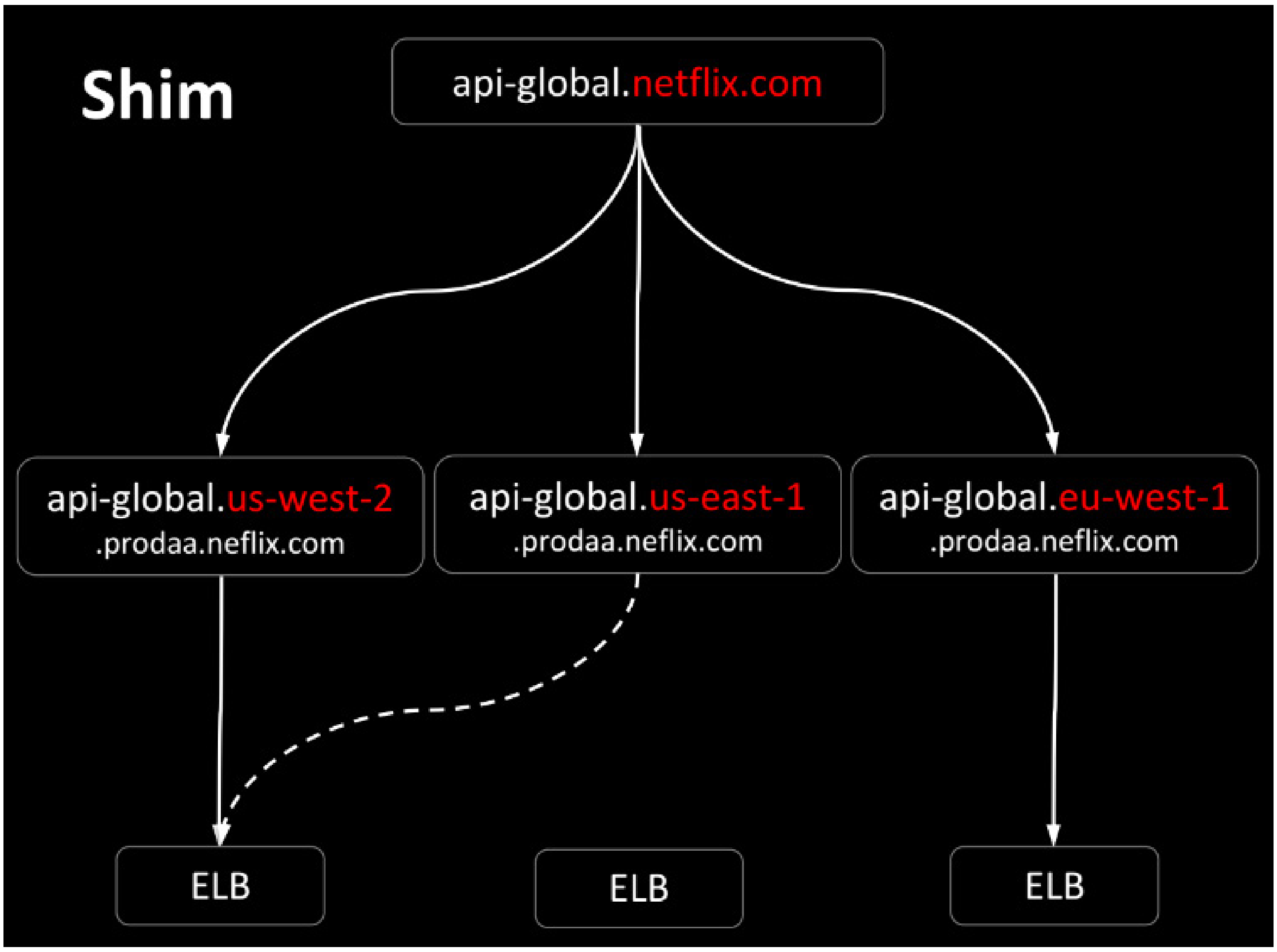
A robust caching strategy allows Netflix to handle over 30 million requests per second, or almost two trillion requests per day. Achieving millisecond responses relies on hundreds of billions of objects in Memcached, distributed across tens of thousands of instances.

Traffic

In 2011, during the migration to the cloud, Netflix was also expanding internationally as well as launching on several new devices. New traffic from Canada and Latin America was joining all US traffic in US-EAST-1, which put all of Netflix's eggs in one basket. With the launch in the UK, Netflix added EU-WEST-1, giving a second basket, and adding some resiliency.

Properly managing the traffic is primarily a function of Zuul, Netflix's open source gateway service that provides dynamic routing, and DNS geo-mapping via UltraDNS to map countries to the appropriate AWS region.

Global distribution coincided with a more scalable and durable database solution, Cassandra. In addition to being open source, Cassandra is multi-region capable and multi-directional, meaning no master node exists.


Shim
api-global.netflix.com
api-global.us-west-2 .prodaa.neflix.com
api-global.us-east-1 .prodaa.neflix.com
api-global.eu-west-1 .prodaa.neflix.com
ELB
ELB
ELB

Regarding the CAP theorem, Cassandra aligns well with Netflix's preference for availability and partition tolerance, with applications designed to accept eventual consistency.

Although Cassandra was a substantial improvement over SimpleDB in terms of scalability, it wasn't quite fast enough to abandon the benefits of EVCache. The tradeoff in complexity of maintaining a caching layer and a database layer is acceptable for the resulting performance of one to five millisecond responses.

Recalling the outage of December 24, which was caused by an ELB service event, Netflix wanted to survive a regional ELB outage. Adding the second region in the US was the first step in the survival plan. Primary DNS routing in the US and Canada accounts for state and province, allowing traffic to be split between US-EAST-1 and US-WEST-2. In the event of an outage at the ELB layer within one zone, all traffic can be routed to one region via DNS. The Zuul proxy can then route some traffic back to the other region, behind the ELB, bypassing the failure completely. This required Zuul to be updated with geo-location capabilities, and the Eureka discovery service became multi-region aware.

Although never used for an ELB failure, the upgrades were a necessary building block towards handling a large-scale, full regional outage. To survive such an event meant being able to evacuate one region and send all traffic to the other, stable region. This requires changes to the data architectural pillar, with data replication allowing any customer to be served by either region.

Revisiting Database & Cache

Cassandra already has a relatively straight-forward solution to this scenario. As with only a single region, a quorum process ensures writes have reached a sufficient number of nodes. Then, the coordinator node starts the replication process with a coordinator in the other region to update those nodes. The process is bidirectional, with a nightly job to clean up any drift or missed writes.

EVCache needed a similar replication process, but in this case, it wasn't built-in, so a fairly complex, custom solution needed to be created. In addition to an ▶

application updating EVCache, it now also added metadata to an SQS queue. An EVCache replicator would read the metadata for a set operation out of the queue, fetch the most recent cached data, and send it to a replication writer in the other region to update the cache for local reads in that region. As with Cassandra's replication, EVCache's is bidirectional.

With the database and cache fully replicated between regions, traffic management needed to evolve, with geolocation routing between regions. By default, DNS pointed directly to the ELB for each region. Because this proved too difficult to update DNS in real-time, the solution was to create a shim layer in front of the ELB. DNS then pointed to a statically addressed shim in front of each AWS region, which directed traffic down to the corresponding ELB. This allowed a single shim to be easily re-configured to direct its traffic to a different region as necessary, and then switched back after the outage.

Global Ubiquity

On January 6th, 2005, CEO Reed Hastings announced that Netflix was now (almost) everywhere, with a launch into 130 additional countries in one day. This included the addition of several languages with interesting challenges, from building a pictograph search keyboard in Japanese, to an inverted user interface for right-to-left Arabic. Content also became ubiquitous, with Daredevil, season 2, launching on the same day, on all devices and all countries.

This global ubiquity is only possible with global availability. The aptly named Netflix Global project ensures that if any region fails, traffic can be routed to another region, and more desirably, multiple regions; an outage in US-EAST-1 would send some traffic to US-WEST-2 and some to EU-WEST-1. The system is flexible enough to allow for cascading failovers. If EU-WEST-1 went down, all traffic from Europe would be sent to US-EAST-1, and some of its traffic would then be sent to US-WEST-2 to distribute the load.

Moving from the active-active setup in the US to full data replication in all three regions required another upgrade to EVCache. SQS was no longer adequate, both from a latency perspective, but also because SQS is a read-once queue. Kafka was chosen as a replacement to SQS, and provided better scalability and performance, as well as handling multiple readers accessing the same queue. This solution can currently handle over a million replications per second.

As with the prior work done replicating data in Cassandra, much of the needed functionality was built-in. Starting with the US ring, in two regions, and a completely separate EU ring, the goal was to create three, global rings. The first step was to extend the US Ring from US-EAST-1 into the EU region, then run repair operations on the new EU nodes. A final forklift operation, along with some data cleanup and configuration work, got to the desired state, with every cluster now having a global ring replicating to all regions, essentially simultaneously.

The final piece of global ubiquity again focused on traffic management. Virtual DNS regions were created, with default clustering of geographic regions into AWS zones. APAC and the western United States and Canada go into US-WEST-2. US-EAST-2 is split into two parts, the first with most of Latin America, and the second with Mexico joining the eastern US and Canada. EU-WEST-1 handles Europe, the Middle East and Africa.

The shim layer evolved to create a second layer, known as the origin layer, which is just CNAMEs, or aliases, that always point to certain ELBs. The magic happens between the virtual and origin level, with simple remapping of CNAMEs providing a variety of failover scenarios. For example, a split failover can take the two US East virtual regions, and send Latin America traffic to the US West region, and eastern North American traffic to Europe. A cascading failover out of US West can direct that traffic to US East, to join Latin America, while pushing eastern North America to Europe.

In a real failure scenario, or a Chaos Kong event, where error rates start climbing in one region, several layers of traffic management come into play. Initially, the Zuul proxy will be used to start re-routing some traffic to other regions, based on where more capacity currently exists. Auto-scaling kicks in to handle the additional load. Once fully scaled up, the DNS cutover occurs, and the Zuul proxy can take a break. After the recovery work is complete, the process is reversed, with the previously failing region being scaled back up, and Zuul and DNS being used to reroute traffic.

What Work Remains?

Although reaching such global ubiquity can seem like the work is complete, much is left to be done, both on the business and technical sides of Netflix. Decisions on where to invest in localization will be accompanied by corresponding development work. Global latency also becomes a bigger issue, especially in countries far from the AWS regions currently in use. This may lead to expanding into additional AWS regions, or experimenting with embedded caches within a CDN outside of AWS, which Netflix currently uses.

Monitoring tools based on machine learning will hopefully be able to replace manually configured alerts, which can be time-consuming to create and are rarely updated as needed. Similar work is being done with self-healing systems and automatic remediation.

Better utilization of spare capacity will help the bottom line. Some of this capacity is provisioned to handle failover, but there are also periods of time with less traffic, and auto-scaling can be tweaked, possibly with significant cost savings. Those down periods can also be identified as times to perform batch processes, rather than letting compute go unused, or needing to add additional capacity on top of high utilization periods.

Although all the necessary technology exists, a failover scenario currently takes about 30 minutes to implement. A goal is to get that down to about five minutes, which can have a significant impact on availability.

A final challenge is to remove the need to have both a caching layer and a database. Consolidating that behind an obstruction layer would make application development much more seamless and efficient.

Key Takeaways

Never fail the same way twice. For Netflix, this meant analyzing the root cause of failures, and making strategic decisions to overcome them. It also meant running regular chaos and Kong exercises, to ensure the solutions they created actually handled the failure scenario appropriately.

Adding resiliency follows many forms, but starts with moving away from a single data center and managing your own infrastructure. Moving to the cloud provides multiple data centers, but regional issues can still exist. An island model provides regional containment. An isthmus can bypass the ELB. Active-Active allows regional failover. Finally, going global provides true ubiquity, resiliency, and efficiency.

Invest in your architectural pillars. They may be slightly different from the four discussed here, microservices, database, cache, and traffic management, but the core ideas still apply. Going multi-region will always involve traffic management. Even without fully adopting microservices, a small number of services at a minimum will probably exist.

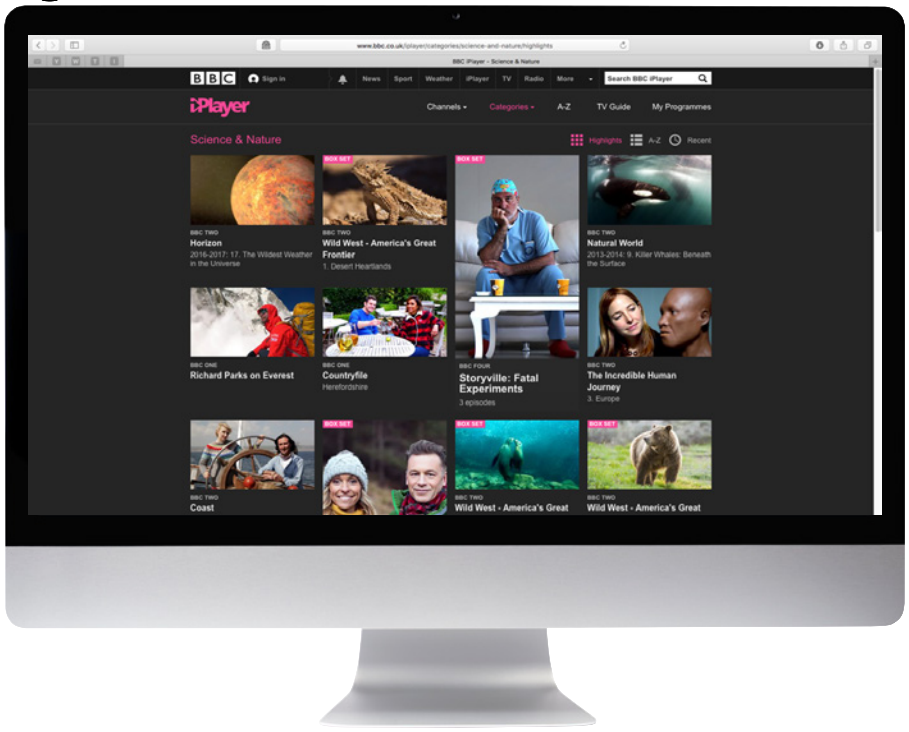
Lastly, think globally, act locally. As Netflix worked to solve each current problem, they always knew at some point they wanted to go global. By constantly looking ahead to that long-term strategy, the solutions they created always took them one step closer to the ability to go global. ■

"Think globally, act locally. As we were solving each problem in front of us, we knew, at some point, we wanted to go global. We were looking ahead to that strategy and so the solutions we created were always taking us a step closer to that ability to go global.

- Josh Evans

Watch online on InfoQ

Cloud-Based Microservices Powering BBC iPlayer



Stephen Godwin is a Senior Technical Architect at the BBC where he is responsible for designing the systems that provide audio and video to BBC iPlayer and iPlayer Radio. He joined the BBC in 2011 and designed the systems that controlled the 24 live streams the BBC made available online for the London 2012 Olympics. Since then he has migrated the systems that power iPlayer to a cloud based microservice architecture. Prior to joining the BBC, Godwin spent over a decade developing middleware at IBM Hursley Park.

Adapted from a presentation at QCon London 2016, by Stephen Godwin, Senior Technical Architect at the BBC

Each week, British public service broadcaster The **British Broadcasting Corporation** (BBC) publishes 10,000 hours of media online, available through iPlayer. A typical day sees around 10 million requests to playback video. During a nine month period, the BBC moved from a performance-constrained, monolithic system to a microservices architecture in the cloud, which has been able to handle all demands for over two years.

iPlayer History

The BBC iPlayer provides online access to content from the BBC's television radio stations, including live programming and 30 days of recent content. Some content is exclusively available in iPlayer, and BBC Three is now only accessible via iPlayer. iPlayer is regularly used by 31% of adults in the UK, and supports over a thousand different devices. The 2012 London Olympics required the development of new systems, including 24 live, online video channels covering all events as they occurred. Those new systems worked well, but the core systems for getting video into iPlayer were struggling.

Those core systems were designed about five years earlier, with fixed capacity. Adding support for mobile devices, tablets and HD content quickly reached the capacity limit. The only way to stay within the system's limitations was to be selective in the content made available online, and what to exclude. This was most noticeable with HD

KEY TAKEAWAYS

Elastic scaling is good, and can provide technical and business benefits

Common, central error handling and monitoring is a necessity for troubleshooting hundreds of microservices

Migrations can be accomplished using only configuration

Moving to a microservices architecture provided an improved customer experience, and increased the amount of content available online in iPlayer, and made it easier to add new features

content, which was capped at 20 hours per week.

Reliability was also a major problem, with any sudden influx of video causing the system to fall over. Harmonic problems existed, with issues in downstream systems causing work to backup, leading to another influx of video requests, and another system failure. Significant care and attention was necessary to nurse the system back to a healthy, steady state of operation.

Moving to the Cloud

In January 2013, a decision was made to rewrite the system. Recent success building systems in AWS had demonstrated the usefulness of the elastic model that is possible in the cloud. The ability to add extra storage and computing power, as needed, would solve the major limitations of the old system.

One small hitch affected the timeline for the migration to the cloud. The old system had been developed in tandem with a third party, and the contract with that third party was nearing its end. A decision to not renew the contract would mean the new system had to be in place in only nine months, by September 2013. A strong strategic plan was clearly necessary, and it began with decomposing the system into major functional units, then creating a solution for each unit.

When looking at a three-step process of creating content, processing it, and delivering it to viewers, two of those were not a concern. Broadcast video was available from the BBC's broadcast streams, and a CDN for distributing files to the audience was also in place. The big empty box in the middle, to create online video files, had to be developed from scratch. The plan was to start small, and unusually, to first solve a slightly different problem.

A related system was used for publishing short video clips on the BBC website, such as behind-the-scenes clips and trailers for popular shows. Like the system that powered iPlayer, the clip publishing application was having trouble, with a video having to be resubmitted three or four times before it appeared on the website. The system had a complex network topology and very synchronous connections that had a tendency of timing out and losing work. The first modification was to use Amazon's S3 storage, instead of writing to on-site NAS storage.

After gaining experience with S3, the team wrote their first microservice. This transcoding service takes a very large, high quality video file, and converts it to smaller files targeted at specific devices. The end result is many files to support mobile devices, tablets, smart TVs, PCs, etc.

The transcoding process lends itself to optimization, when similar types of videos are grouped together that can share parts of the transcoding. This requires less CPU power and runs faster, and therefore reduces costs. The primary function of the transcoding service was to group videos into batches, then delegate the real video processing to other back-end services, which were also newly written microservices.

The initial proof-of-concept for back-end processing was FFmpeg, but in production it was replaced with Elemental's PaaS offering, running in AWS. The ability to target multiple back-ends was a useful feature of the new system, and was initially used to support subtitles, converting subtitle files from various ▶

formats. Later, the addition of audio-only content required only a small change to the business logic to identify an incoming request as audio-only, and pass it to a new audio processing backend for transcoding.

Content Distribution

After generating the video files, and writing them to S3, the next challenge was getting them out to the audience. This meant getting the files onto the origin, then the content distribution networks could take the files and distribute them to the audience.

The previous transcode service also handled the distribution, at the end of the transcoding. Separating the transcoding and distribution would be the first real move towards a proper microservices architecture. Because multiple origins existed, a complicated process involved copying files into multiple places, verifying the distribution, then making them available to the audience.

By creating two separate services for transcoding and distribution, each could be focused on doing one thing, and doing it well. This led to a more reliable system, and happier customers, as the editorial staff could finally upload content in one attempt.

Integrating with the Broadcast Chain

In April 2013, with two major services reliably processing video clips, the next objective was to have full TV programs available on iPlayer. This required integrating with the Broadcast Chain, the physical cables in the ground and transmission towers spread across the country. Several locations were identified where the high quality TV feeds could be intercepted.

The bitrates for video are quite substantial, at 30Mb/s for each HD channel and 10Mb/s for SD. A Video Chunker takes a high-res stream and writes it to local disks, in 80MB chunks, approximately 20 seconds of HD or 60 seconds of SD video. A separate process running on the same box then takes those 80MB chunks and uploads them to S3.

The total amount of data is equally substantial: 21 TB per day written to S3. 5.2 TB of video files are created each day — 2.3 TB for SD channels, and 2.9 TB for fewer, higher bitrate HD channels. For resiliency, two copies of the infrastructure are run in two separate locations, creating four copies of the data. Although there were some initial concerns about the volume of data being continually written to S3, performance has not been an issue, at least in part to the use of (relatively) small 80MB chunks.

A distributed network of eight servers split the processing of all the TV channels, with each server handling up to 20 threads uploading chunks simultaneously. The parallel processing ensured that any one chunk having a problem doesn't slow down the other streams. Having a few hundred network connections into S3 allows Amazon to perform load balancing within AWS, which worked very well, except for one issue. Every few weeks, some of the connections would suddenly slow down dramatically, then basically stopped. Restarting the services fixed the issue, only to have it reappear weeks later.

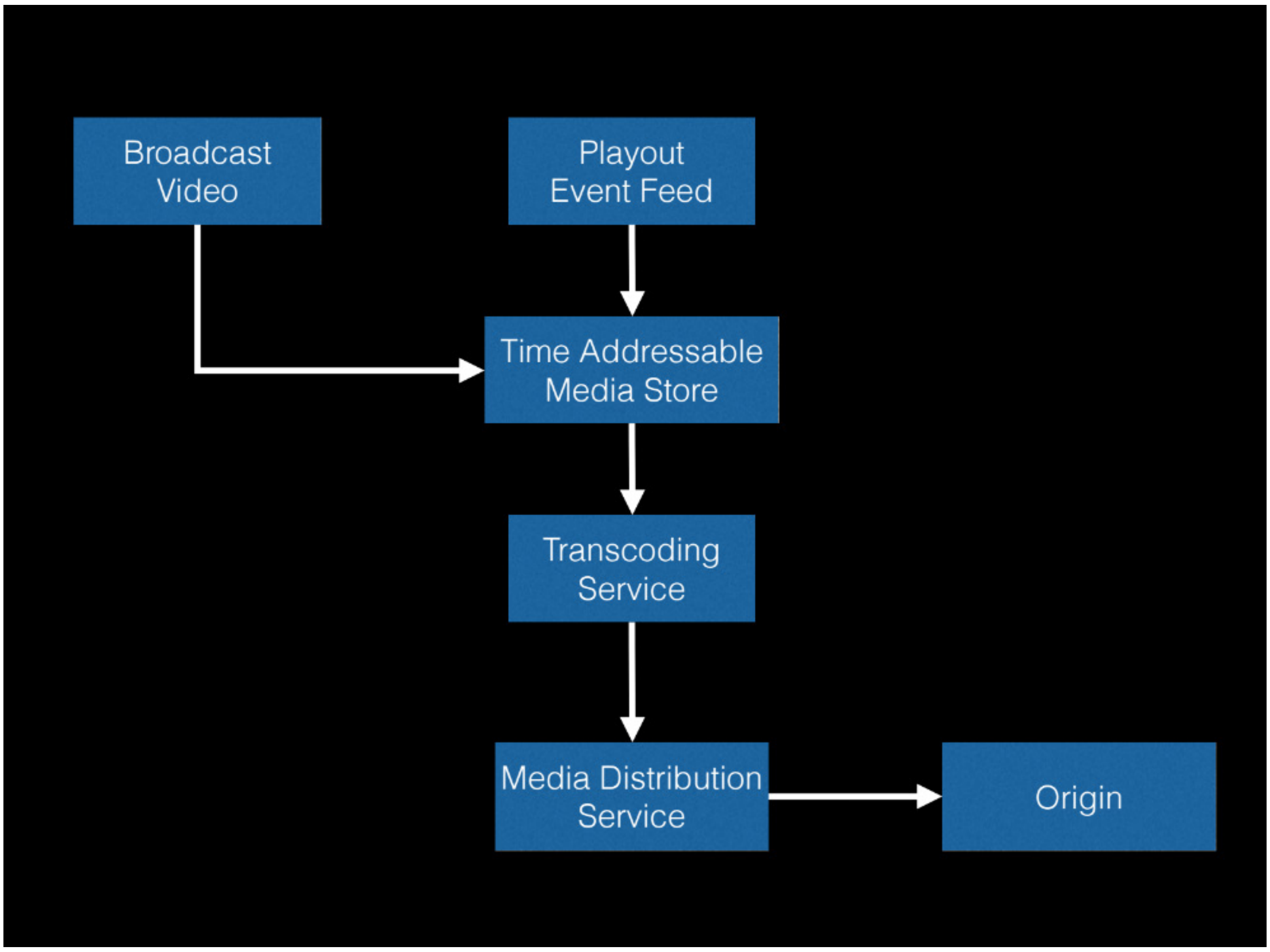
The nature of the broadcast streams meant continuously uploading video, all the time. Because Amazon's SDK is optimized for connection reuse, some connections would stay live for several days, or even weeks. If Amazon made networking or server changes, it would cause slowdowns or an effective outage in the upload pipeline. The solution was provided by Amazon adding an optional connection timeout. After setting a 15-minute timeout, causing a connection to be closed and reopened every 15 minutes, no problems have occurred.

The final task for moving the video data into S3 is reassembling the 80 MB chunks into TV programs. Each chunk represents about 20 seconds of HD data, and concatenating two chunks together stitches them into 40 seconds of video, with no noticeable join in between. S3 has a feature for creating a file from pieces that already exist, so all the concatenation happens on the S3 side. This allows an hour of source video to be made available in under a minute.

Time Addressable Media Store

The system described thus far was implemented in July 2013, and known as a Time Addressable Media Store. The main feature was the ability to query by channel and time, for example, "BBC One 9 PM to 10 PM, yesterday evening," and the result would be the corresponding broadcast video. In theory, this becomes the world's best DVR, with several days of very high-res recordings available. The only missing piece was knowing the start and end times for the TV programs coming into the system.

Integrating with the Playout Systems provided the needed data for every program. The Playout Systems control what is broadcast on BBC TV channels at any given time, and the associated



data feeds include frame-accurate timings for the starts and ends of every program being broadcast. The combined video files and timing data are then ready for distribution onto the Origin.

Microservices

Although a high-level diagram provides a simplified view of the major system components, most of the actual work is performed by about 20 microservices. Excluding the major integration points with the playout feeds and final distribution, most services are simple, message-driven services using Amazon SQS to provide input and output queues.

The BBC's primary concern for a queueing system was resiliency, and SQS has features which met this need. However, on rare occasions, SQS can repeat and resend a message, and this had to be accounted for during system design. In most scenarios, this was fine, as publishing the same thing twice wouldn't affect the final result.

Each microservice is a Java application, running inside the JVM, on an EC2 instance. The Apache Camel framework was used to integrate with SQS, and in some cases where SQS support needed to be improved, those changes have been pushed back to the Camel open source project.

The codebase for each service is typically in a separate SVN repository. While not done intentionally, analysis of these independent repos revealed each service consists of roughly 600 Java statements. This doesn't mean all microservices should be that size, but for the BBC iPlayer, it seems to be the right size for their purposes.

This also aligns well with the fact that each service only has one or two developers working on it at a time. More than a few developers usually indicates that a service is too big, or too many competing changes are occurring at once. Small services, with a very limited set of functionality, allows for very focused development effort on small changes.

EC2 Implementation

In most cases, at least three EC2 instances host each microservice. A Competing Consumers Pattern is used to read messages from the input queue and put them on the output queue. When an instance of the service reads a message off the input ▶

queue, it is locked and hidden from other instances of the service. If, after a 30 second timeout, the instance hasn't responded to SQS to renew, delete or mark the message as processed, the message becomes visible to the other instances of the service.

This pattern for reading from the queue improves both scalability and resiliency. If additional workers are needed, the Auto Scaling group can easily be modified to have a minimum of 30 instances, instead of three, resulting in messages being processed 10 times as fast. Alternatively, if one worker fails for any reason, the message will reappear on the queue after 30 seconds, and the service would be restarted and reappear a few minutes later. Chaos Monkey is used to randomly remove services and validate the resiliency of the system.

Furthermore, the error handling within Apache Camel aligns well with this system. When any unexpected error is encountered, it simply bubbles up to the top of Camel, and is let go, releasing the message back to the queue. This creates a retryer automatically built into the system.

In the case where a problematic message could cause repeated retries to fail, it needs to be identified as a "poisoned message" and handled appropriately. Each SQS message includes an Approximate Retry Count header indicating the number of attempts to process the message. At the beginning of each, each service is code to examine the header, and move messages with too many retries to a dead letter queue. Human review of the dead letter queue allows analysis to identify any significant problems. This functionality has now been implemented within SQS as a Redrive Policy.

Monitoring

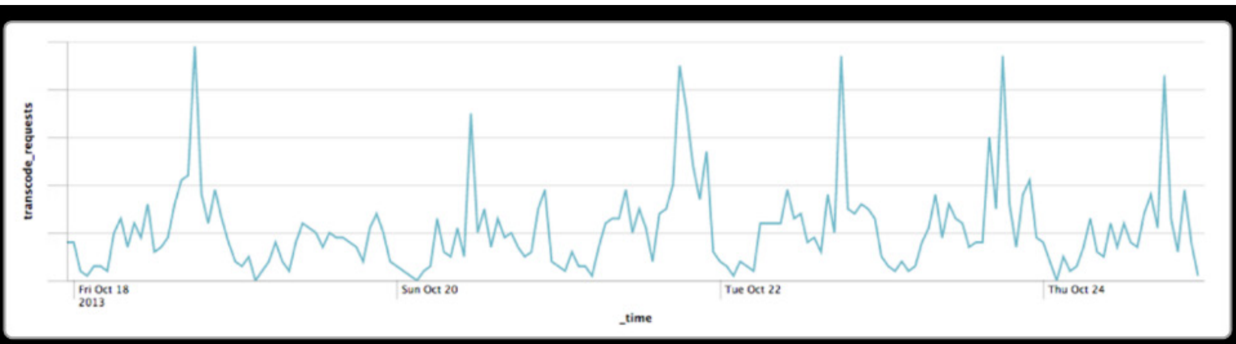
In addition to the dead letter queue, basic monitoring is configured on the EC2 instances hosting the services. Further monitoring is performed regarding the queues, with a focus on the depths of the queues. As a general rule, an empty queue is a happy queue, so alarms are configured when the number of messages in a single queue gets too high, with the threshold based on the nature of the service processing the queue. When a threshold is reached, it usually means something is wrong with the system, and processing has completely stopped, or there is simply a need to scale out to handle the load.

Along with monitoring, a consistent approach is used for logging and auditing the process within a service. Most of the auditing messages are business focused. These messages are sent via SNS to a log processing system, in this case Splunk, to provide a centralized view of what is happening across all the microservices. Each piece of work, for example, the 1:00 news ending at 1:30, is given a unique ID as it enters the system, and follows that work through the entire iPlayer workflow. This makes it very easy to see all the related pieces of work and debug any problems in the system.

Going Live

By August 2013, a complete system existed, including good approaches for operations and monitoring. The single major step remaining was actually going live. The goal was to avoid a "big bang" approach, switching entirely from the old system to the new, all at once. If that occurred, any problems or missed functionality could be catastrophic, and would be very publicly visible.

Achieving a gradual rollout was possible because of design decisions made very early in the project. Configuration options enabled the selection of what the system would handle, including which bits of videos to process, which devices to support, which TV stations to work with, down to the individual program level.



Off-Air Transcode Requests for 1 week

One interesting trend for the BBC iPlayer is a daily spike in incoming transcode requests, at 7:00 PM every weekday. From 6:30 to 7:00, regional news programs change BBC One from one TV channel to effectively 19 separate TV channels, all needing to be processed by iPlayer.

Resource constraints in the old system meant it took about 16 hours to get the last of the news programs out. Since news programs are usually taken down after 24 hours, this system was not terribly useful. Improving this scenario was the first test of the new system. Because the incoming load could be anticipated, the new, elastic system was scheduled to scale out every day, ready to handle the news programs when they arrived. The end result was reducing the 16-hour process down to 30 minutes.

With the news serving as a good test case for a few weeks, confidence in the system improved, and new devices and channels were gradually added. After several weeks successfully processing all the other channels the final move was the highest profile channel, BBC One.

After Go Live

When the September 2013 deadline arrived, the new system had been successfully built and implemented, completely replacing the old system. The fact that iPlayer was still working, and still had video, and not a test pattern, could be considered a major success unto itself. But the audience received added benefits because of the improved system. Previously, limited system capacity meant deciding which content would be made available online. Now, if the BBC had rights to the content, it could be, and usually was, published to iPlayer. This led to a doubling in the amount of content in iPlayer, and HD content increased by 700%. A few months after launch, the elastic capacity of the cloud was further leveraged to increase video availability from seven days to 30 days.

Developers also benefitted from the new platform. Changes can be deployed in 15 minutes, using immutable AMIs. A relatively small team, about 25 developers, performs 202 deployments per week, 34 of those to live. An average of more than one production deployment per developer, per week, comes with responsibility to perform testing and make reasonable changes. Developers are expected to spend 60% of their time writing tests. Working with very small changes both enables and requires frequent deployments.

An outside-in BDD/TDD approach starts with writing the acceptance test first. Although the services are written in Java, all tests are written in Ruby. This avoids the temptation for developers to reuse code between the test and implementation, such as a serialization library, which can lead to symmetric bugs which pass all tests, but fail in real use.

Even with considerable testing, issues can still appear. For example, when a two-line change was deployed, it included a security update, which led to the service failing every three hours. Using immutable AMIs, and making small changes, meant the previous version could safely and quickly be deployed until the new version could be tested and fixed, including the security patch. Because the deployment process noted all changes that occurred, identifying the new JVM as the root cause was much easier than with the previous monolith system.

Advanced Features

The new system, including the deployment process, allowed new features to be added quickly. The first major addition was integrating the Simulcast System for showing live copies of the BBC One and BBC Two TV channels into the microservices architecture. The Live Restart system was also migrated at the same time, allowing a viewer to jump back to the beginning of a live TV program while still being broadcast. By collecting the video as it goes past the Simulcast System, and writing it to disk, it allows live events to be made available within 10 minutes as a catch-up piece of content.

For the first time, a single system can now be used for both TV and radio content, instead of two separate monoliths. Microservices were a key factor in enabling this shared platform. A single monolith, with competing business logic, would have been a daunting system to build and maintain. Microservices allow clean separation of business logic, as well as reuse of code that can be common to both TV and radio. 60 different BBC radio stations, most with both international and domestic variants, are a major contributor to the 10,000 hours of content published every week.

Content from S4C, a public-service broadcaster for Welsh-language television in the UK, was also added to iPlayer. Small adapters were written to collect the content, and were connected to the existing microservices. S4C content appears alongside other channels in iPlayer. ►

Recently, BBC Worldwide, the commercial arm of the BBC, launched the BBC Store to make videos available to purchase and own online. Providing the transcoding for BBC Store, was only possible because of the new architecture. Additional capacity was required, incurring additional costs. The pricing model in AWS meant the expenses are easily quantified and communicated to what is effectively a third party for cost sharing.

Another change to iPlayer was moving from Adobe Flash towards HTML5. Previously, the CDNs would handle final video packaging, but this change meant the packaging needed to be added to the tail end of the transcoding and distribution service. The migration was again made in a gradual manner, with config options specifying devices and channels to utilize the new system. The old technique is still running, to accommodate devices which do not support the new video format, another benefit of using a microservice architecture.

Problems and Concerns

After a couple of years, some services grew too big, to around 1200-1400 Java statements, twice the average size. These were broken down into several small parts, making them much more manageable and easier to maintain.

Using AWS EU-West-1 is point of concern, since it is located in Dublin, across the Irish Sea from most of the BBC. When problems with network connectivity between the UK and Ireland caused an outage in Simulcast, it led to a second copy of the simulcast infrastructure running in a BBC data center, with a 50/50 split between the two systems. The long-term plan is to utilize a London AWS, when it is ready, and eliminate the need to maintain two different styles of data centers.

An interesting side effect of being able to move fast is sometimes being left behind. One planned change was delayed six months; by the time it was implemented, it was no longer relevant. The lesson is to expect the system to constantly change, and if significant time has elapsed, check the design again when finally working on a new feature.

Lessons Learned

Elastic scaling is good, and linear or better scaling is great. Designing the system around the idea of elastic scale, and the corresponding pricing model, meant it was easy to present options for adding system capacity to business owners to decide if the benefits justified the cost. It also led to reliable pipelines of microservices, breaking down complex problems into small pieces that are easy to change and redeploy.

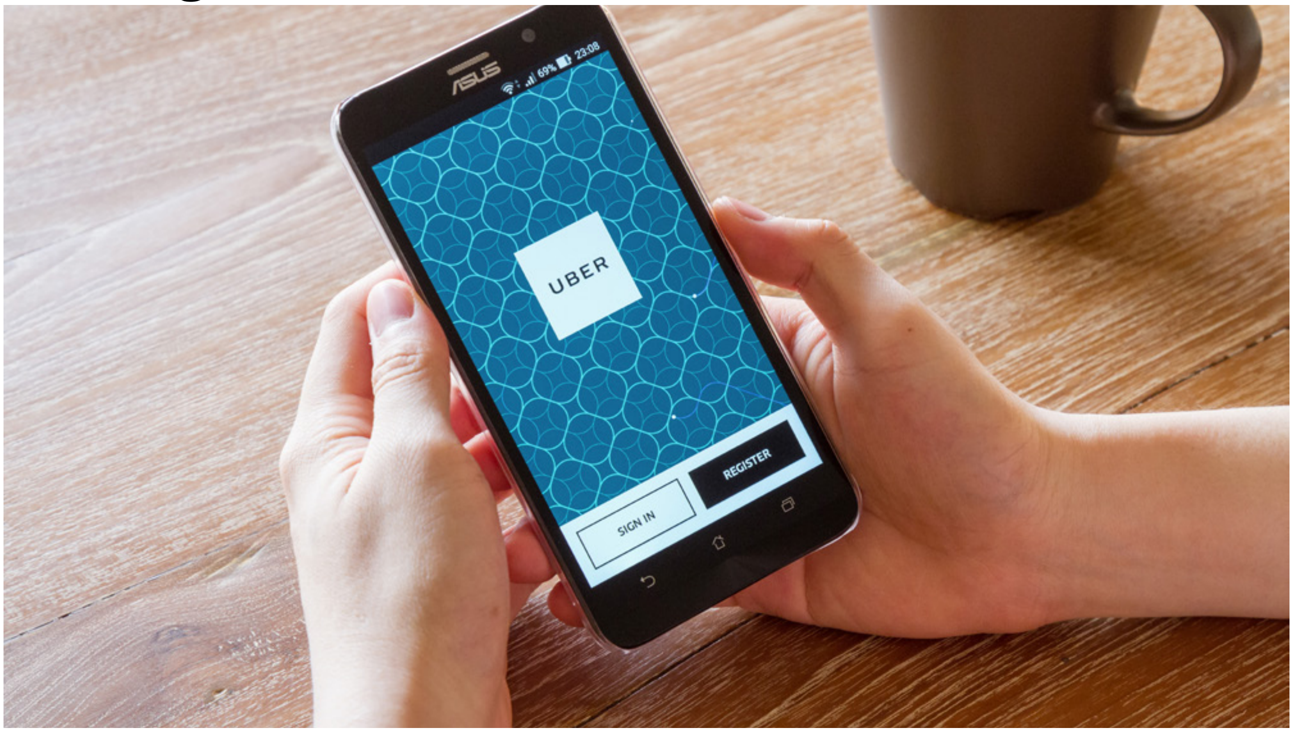
A common approach to error handling and monitoring, including the audit event system created at the very beginning, proved invaluable. With over 100 microservices spread across 300 instances, troubleshooting the system requires a central place to start looking for problems.

Migration can be done using only configuration, but it must be built in early. This allows the behavior of the system to change in small and large ways, up to adding major new functionality purely through config.

The BBC was able to move iPlayer onto AWS under significant time pressure. This massively increased the amount of content that can be made available online. The new system made it easier to add new features. Finally, using a microservice architecture provided a seamless audience experience, even while major changes were being made throughout the system. ■

Watch online on InfoQ

Scaling Uber to 1000 Services



Matt Ranney is Chief Systems Architect at Uber, where he's helping build and scale everything he can. Previously, Ranney was a founder and CTO of Voxer, probably the largest and busiest deployment of Node.js.

Adapted from a presentation by Matt Ranney, Chief Systems Architect at Uber, at QCon New York 2016

Uber has learned many lessons from dealing with incredible rates of growth in customer demand, company size, and the technology it relies on. In just two years, their platform scaled up from just 20 services to over 1,200, and was accompanied by a 10x growth in engineering headcount. Uber has the benefit of being a relatively new company, and is not burdened by legacy systems. Their architecture is, and has always been, microservices, which allows a strong perspective on all the benefits and shortcomings of a microservices architecture.

Benefits and Costs of Microservices

Agility is the major benefit realized by microservices. Components are isolated, and can be released independently, allowing for rapid changes and growth of the overall system. This, combined with a short ramp-up time, is crucially important when people are being added to engineering teams as rapidly as new services are being created.

Microservices also tend to be more reliable, as teams have more personal accountability. They own the uptime, availability and release schedules, and are therefore are more motivated to build in reliability as a feature. A closely correlated benefit of microservices is being able to choose the best tool for the job. Since there is no easy definition for what is "best" in each situation, a good understanding of the trade-offs is key. However, these trade-offs can lead to some of the less obvious costs of microservices. ▶

KEY TAKEAWAYS

Everything is a trade-off. Whenever possible, make decisions intentionally, rather than just accept them by default.

The major benefits of microservices are agility and reliability, but these must be balanced against increased complexity.

Microservices also have less obvious costs, including additional latency due to JSON and RPC calls.

The operational complexity of microservices must be managed using tools for tracing, logging and testing.

The benefits of microservices must be weighed against the costs, and the foremost concern is complexity. Instead of a simple system, the result is a distributed system which is complicated, hard to understand, and hard to debug. When outages occur, they are more difficult to troubleshoot, sometimes not making it clear where to start to fix the problem.

Choosing microservices means accepting increased operational complexity to achieve improved short-term developer velocity. This is probably a reasonable trade-off, especially for a company with exponential growth, but be aware that the level of operational complexity may be surprising.

One architectural pattern is to create immutable microservices, or an append-only microservice architecture. In other words, never turn anything off. If a system component is static and does not require modifications after becoming stable, why would you want to change it? The Uber service is most reliable on the weekends when engineers are not making changes, so sometimes increased complexity can actually result in increased stability. A hybrid solution is to consider making services immutable after they reach a certain age and maturity. Again, the architect needs to consider the cost-benefit analysis of leaving the old services running versus the cost to change them.

InfoQ recommends

The InfoQ Podcast
Uber's Chief Systems Architect on their Architecture and Rapid Growth

Less Obvious Costs

When designing and building a system, keep in mind that everything is a trade-off, even if it is an unconscious decision. This is more true with big microservices deployments, and it surfaces in subtle ways.

Optimization for developer velocity can lead to a temptation to build around problems instead of fixing the issue. If a dependent service doesn't work properly, why not just build a new, better service? Getting the developer unblocked, but adding to the complexity of the system is already the accepted trade-off. However, sometimes this decision is based on politics and relationships, rather than technology and algorithms.

Politics can surface in the form of developers being very productive, cranking out new services, just to avoid having hard conversations. It is important to identify and guard against this behavior, as it has an unexpected side-effect of being detrimental to the business, the entire development team, and a developer's career development. Avoiding hard conversations directly correlates to people to keeping personal biases. It is tempting to continue to do the same, comfortable work, rather than being forced to learn something new.

Languages

For Uber, this trend manifested through language specialization. Initially, Uber used Node.js and Python, and has been transitioning to new services written in Go and Java. On the surface, having multiple languages doesn't seem like an issue, because sharing code shouldn't be a concern given the ease of writing another service and talking to it over the network. However, every major system will always have some common, fundamental behavior that needs to be shared across components, resulting in expensive and difficult duplication of effort.

Moving among teams becomes more challenging when there is a greater chance that the other team uses a different language. The culture also becomes fragmented, with distinctions such as "Java people" which should not exist among a unified, cohesive team.

Remote Procedure Calls

Microservices rely on using Remote Procedure Calls (RPCs), trading the easy, fast, and intuitive world of inline function calls with a complicated, hard to debug, and sometimes impossible to understand environment where everything travels over the network. The most popular protocol for microservices is HTTP, and it certainly has a lot of benefits and support from various microservice tools and frameworks. However, widespread use of HTTP quickly showcases its intent as a protocol for communicating over the open internet, from browsers to web servers. Using HTTP inside the data center add unnecessary complexity. Instead of handling a simple function call with arguments, HTTP brings with it query strings and headers and response codes -- all features which don't benefit microservices.

JSON

Along with HTTP usually comes messages transmitted using JSON. Like HTTP, JSON has some ease-of-use benefits, including being human-readable, and being almost universally supported. However, the lack of a schema and strong typing means extra care is necessary when dealing with multiple languages. JSON is also slow, which could be said of any extra encoding and decoding process. This is compounded by the fact that RPCs will always be slower than local function calls.

An architecture of 100% microservices, using HTTP and JSON for RPCs, requires more computing power just to make the system work, compared to a monolithic architecture.

Repos

Source control for microservices brings with it yet another decision: whether to have one, very large monorepo, or to have hundreds of very small repositories. The tradeoff here is between the ability to make cross-cutting changes (and corresponding rollbacks) versus flexibility and faster checkouts. While Uber has gone down the path of thousands of repos, and the culture accepts the process, the recommendation is to make an explicit decision, rather than watching the process evolve organically.

Operational Complexity

Operational complexity is the core problem facing a large microservices implementation. Techniques used to understand how one big thing is broken may not be useful when the system is composed of hundreds of small parts. After all, the effort to create a system built of microservices, where pieces are isolated and unaware of the implementation details of other components, it would be really useful to have the exact opposite and be able to view the system as if it was just one big, integrated monolith.

Performance needs to be measured and understood at the system level, which seems obvious, but is often overlooked by microservices. There is a temptation to think that, since a service is just doing one, small task, and it's relatively fast, performance isn't important. This isn't an advocation for premature optimization. Rather, standard monitoring and dashboards need to be defined, implemented consistently for all languages, and provided automatically for new services. Teams can augment and add additional monitoring as they see fit, but standardization allows metrics to be consistently observed and aggregated at various levels within the system.

Being able to manage for performance is much easier if built in early, rather than bolted on later. Every new service should have an SLA, including acceptable levels for both availability and performance. Setting the initial performance threshold very high means it can be adjusted later, if needed. This is much easier to do than realizing there is no knob to adjust for performance, and having to add it later.

Good performance is not required, but *known* is. ►

Fan-out and Tracing

Identifying performance problems with microservices requires a good understanding of how fan-out affects the net performance of the system. The overall latency of any given request is at least as large as the latency of the slowest component involved in the fan-out. Simply stated, before the user can get their response, they have to at least wait for the slowest thing to complete. Even if a slow call only occurs rarely, the effect of the fan-out can magnify the number of slow requests and customers affected.

Tracing is essential for understanding system performance through fan-out, and various tools exist to accomplish tracing, from log stitching to Zipkin. For example, one call may always be really fast, but if it needs to be called hundreds of times, the net result of all those calls may be slower than desired. A batch method may be a good solution, even if it was not originally thought to be required. Tracing provides the insight into how the service was being used in production. Without tracing, it may be possible to find the root cause, but tracing certainly makes the process easier.

Although tracing is the crucial component for understanding fan-out performance, the overhead to create tracing data can exceed the core workload. For a production workloads, sampling just 1% or less of requests will provide enough data to satisfy most analysis needs.

Tracing becomes more difficult when multiple languages are involved, as there is no simple solution for cross-language context propagation. One technique is to have each service pass along properties on the context of the incoming request along to the outgoing request, with the understanding that, even if a given property is not applicable to your service, the properties will all be useful somewhere in the chain.

Logging

Similar to tracing is a need for consistent, structured logging. It can be tempting for individual developers and teams to add logging as they see fit. However, like the need for common dashboards, all services must generate logs which can easily be processed by off-the-shelf or custom log aggregation tools. Keep in mind these tools are the primary consumers of logs, not the humans, who first need a searchable index created.

Unfortunately, the costs associated with all this logging can become substantial, as the computing resources grow to handle the log processing load. Sometimes log messages may need to be dropped or an SLA will be missed. If possible, being able to tie the costs of heavy logging back to individual services can raise awareness among teams, and hopefully reduce extraneous logging.

Load Testing and Failure Testing

Planning for testing needs early can have significant long-term benefits for system performance as well as improving culture. If load testing in production is necessary, then processes need to exist to allow test traffic to create measurable load, while also being excluded from telemetry reports. Retrofitting all services to handle test traffic is extremely complicated and expensive compared to planning and building it in early.

Similarly, failure testing can be very beneficial to understand how the system responds to various failure modes. Not surprisingly, developers are reluctant to having their working software deliberately broken. This is less of a concern if failure testing was just another feature that is automatically included with every service.

Trade-offs are Everywhere

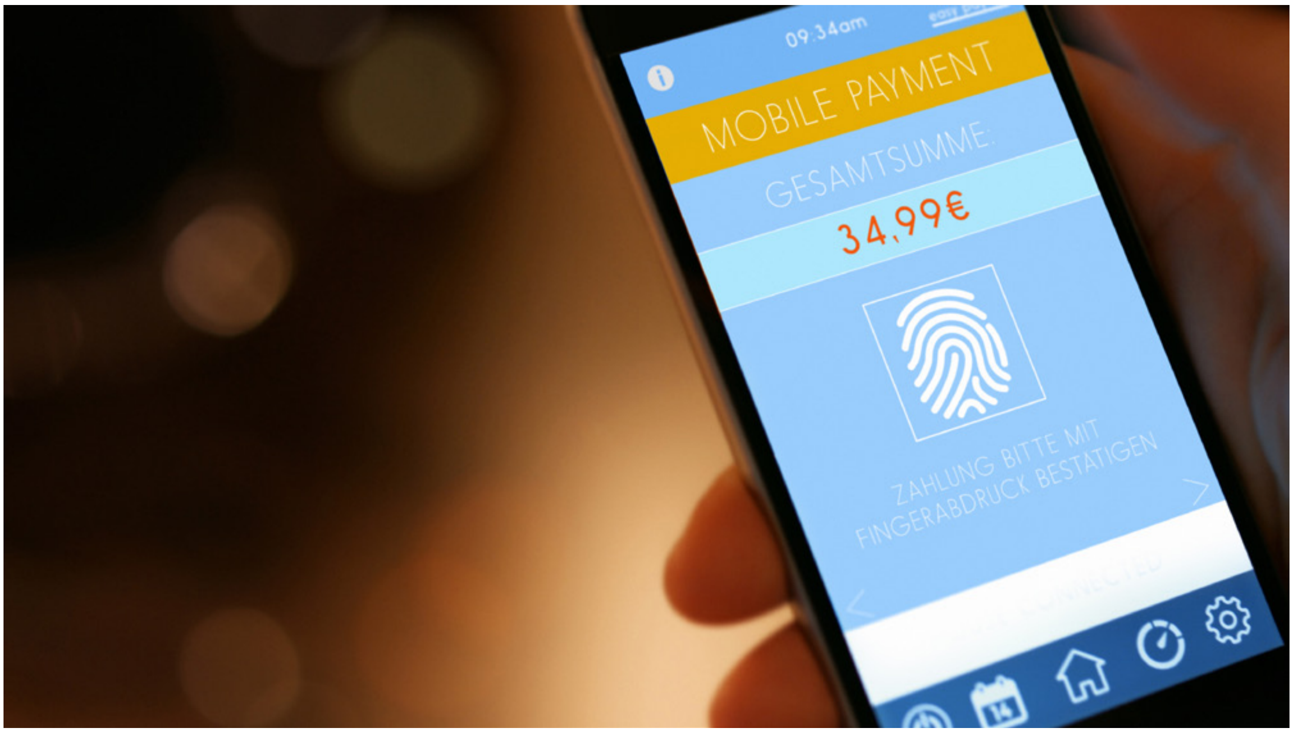
One of the most common trade-offs in software is build versus buy. It can be tempting to want to build really cool infrastructure projects that have a major impact on the company. Unfortunately, really useful platforms and infrastructure projects often become commoditized, either through open source or implemented as a service by a cloud provider. This can eventually become a disadvantage to own and support. Given the choice, most businesses would prefer to spend time and money on development and support of features that are market differentiators.

A final cautionary note regarding services is they will allow people to play politics. In this sense, politics are defined simply as any time an individual's priorities are valued above the team, or the team is placed above the company. Services allow a level of insulation that can make it easy to play politics along this spectrum. The tracing, logging and monitoring patterns which allow an understanding of the system as a whole, can serve as guidance for how to deal with political challenges that arise.

The fundamental truth is that everything is a trade-off. Sometimes, the trade-offs are not obvious, which can lead to decisions just being accepted by default. Whenever possible, identify and analyze the trade-offs, and make them intentionally. ■

Watch online on InfoQ

The Architecture That Helps Stripe Move Faster



Evan Broder has worked on systems and infrastructure at Stripe for four years, helping them stay online through several orders of magnitude of growth. Previously, he worked on virtualization management and the Linux desktop at MokaFive and helped build XVM at MIT, one of the earliest cloud computing environments.

Adapted from a presentation at QCon New York 2016, by Evan Broder, Principal Engineer at Stripe

Stripe has a proven history of being able to make big changes successfully. Unlike other startups, they cannot simply "move fast and break things," as Mark Zuckerberg famously described Facebook's practices. Stripe accepts payments made through online applications and by businesses, including Lyft, Kickstarter and Twitter. Although around 20% of Americans used Stripe in the past year, most were unaware of Stripe's involvement. Processing billions of dollars in payments means a higher expectation of reliability and stability than a typical startup of their size.

The business environment Stripe is in requires a careful balancing act of two different concerns. Being popular in a crowded, competitive industry means continually innovating and trying to improve the product and service they provide. On the other hand, it is equally important to provide a stable, reliable service, which usually means making as few changes as possible. This transition towards more stability is common to every successful company, but for some it occurs earlier than others.

Because stability is a fundamental need for Stripe, they tend to have a bias towards making slow, incremental changes. The big projects which have been successful, such as migrating infrastructure, all followed a deliberate plan. Engineers can be resistant to this approach, often thinking they can always write more code to solve problems after they arise. However, as three successful projects demonstrate, many factors contribute to Stripe ▶

KEY TAKEAWAYS

Find points of high leverage to solve a broad problem instead of trying to solve many small problems.

Have a plan to test early and test often.

Work incrementally in small steps and build on top of established layers.

For Stripe, incremental changes are the most successful, cause the fewest problems and are the most effective way to make big changes.

being able to move fast, while making big, safe changes.

Evolution of the Stripe API

Several approaches help Stripe change the API in incremental ways, providing continual improvement, while not impacting the stability for existing users. Over time, little modifications add up to big changes. If a current consumer were to look at the API from five years ago, it would be only vaguely recognizable. For example, the old technique was to have a parameter to specify the API method you wanted, which was not very RESTful, and has since been eliminated. Other changes have been made to support new products, features, and methods of payment, such as bitcoin.

Some updates to the API were for purely practical reasons, to ensure the Stripe platform could scale. Several features in the API were accidentally quadratic, which works fine with a relatively small user base, but can become a major problem as the product is adopted and demand grows. One sign of a healthy development environment is the ability to look back, recognize some past design decisions were not correct, and be able to fix them.

Whenever possible, API changes were made in a backwards compatible way. Relatively straightforward examples include adding new fields or new methods, or adding new parameters as optional. These are backwards compatible because there is no expectation for the users' code to change. However, the more interesting idea is how to handle backwards incompatible changes.

Engineering Values

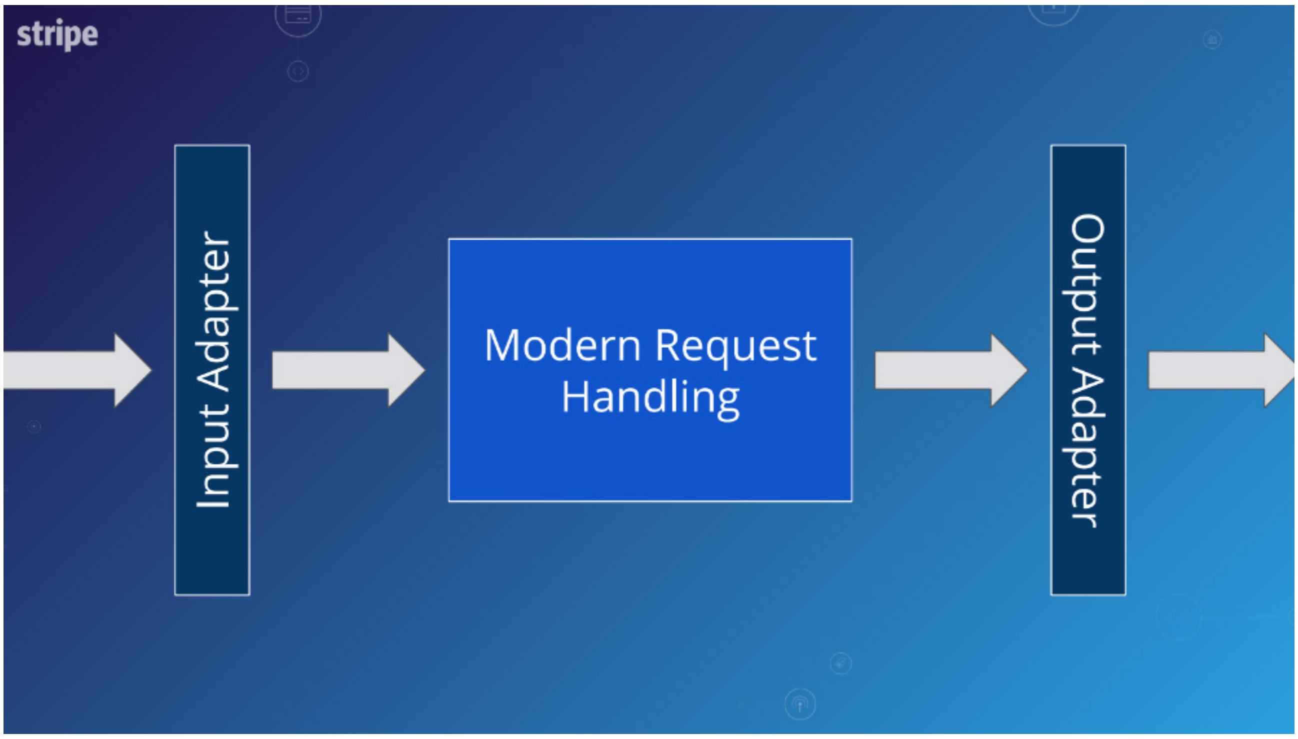
Before going into the details, it's useful to understand Stripe's engineering values, as they highlight what is most important to the organization. Stripe exists fundamentally to empower developers. They want to make developers more effective by solving broad classes of problems, allowing the developers to focus on the things that differentiate their business.

This philosophy is manifested in two significant ways. First, it should be as easy as possible to start writing code using Stripe. The process to sign up for an account and start writing a few lines of code is very easy. Second, once a developer writes a line of code using Stripe, that code should never have to be changed. Business requirements may require making modifications, but nothing Stripe does should break code running in production.

Backwards Incompatibility

While Stripe has made some complex changes to the API, a simple example works well to describe the process of making a backwards incompatible change. Historically, when details about a credit card are retrieved from Stripe, one of the attributes was named *type*, and could be Visa, MasterCard, American Express, etc. Over time, the team realized *type* was a really bad name for a field. (Naming Tip: never use the name *type* in your API. Find a better name.) In this case, the name *brand* more accurately describes the data. The challenge was to make an improvement to the API without having a negative impact on current users.

Being able to provide different behavior to different users required a concept called gates. Facebook has used this pattern by using a piece of infrastructure



called Gatekeeper to handle partial rollouts in different locations and other situations. Stripe's solution is similar, but the key difference is in controlling behavior for a specific user.

Each Stripe user has a list of enabled gates. Each gate enables some sort of legacy behavior. When new behavior is desired, a new name is chosen for the gate. That gate is added to every user in the database, so all the existing users now see the old functionality. Code in the API checks for the existence of the gate, and gives the old behavior if the gate is present.

In the field renaming example, the API grabs the *brand* field for old users and sets it to the *type* field. New users, who don't have the gate, will only ever see the *brand* field. This is a fairly simple example, which only affects the inputs and outputs, but does not alter the actual processing within the API. If the API logic has to know how to handle different types of requests, it can quickly become a tangled mess of spaghetti code.

Avoiding the mess within the core API code meant adding translation layers at the beginning and end of each call to explicitly handle the complexity. When a request arrives, the gate check for the user is performed, and parameters are adjusted so it looks like a modern request. The core code then only needs to handle requests that look like the current API. A similar, reverse process occurs to send the response back to the user, converting from the modern structure into what the user expects.

While the translation adapters can become arbitrarily complex, the complexity is very cleanly contained. The core API only needs to know about the current version of the API, making it much easier to test and greatly reducing the maintenance cost for introducing new gates. This effectively solved the first problem, by allowing incremental changes in the API. The second challenge was hiding complexity from the users, which came with additional issues.

Hiding Complexity

Keeping the API simple, at least from a user's perspective, relied on an abstraction. Like most abstractions, it was fundamentally leaky. Often, the documentation didn't line up with the either the expected or current functionality. When a user learned of and wanted to start using new functionality, they had to submit a request, then have their account modified with the new gates.

Unfortunately, adding new gates made it progressively harder to test interactions. This is a classic case of exponential growth, with two gates having four combinations, three gates having eight combinations, and so forth. The incremental changes were actually becoming quite substantial, with intermingled complexity. Eventually version numbers were introduced to help manage complexity. ▶

While versioning is certainly not revolutionary, Stripe initially thought they didn't need version numbers in their API. Hindsight proved that assumption wrong. Stripe's version numbers are based on dates, and correspond to a named set of gates. This meant a semantic naming could be used to determine if a specific version does or does not have corresponding behavior. It also meant a linear, rather than exponential, approach for handling backwards compatibility.

Stripe considered various options for handling version numbers, such as including the version in the URL. Stripe's core value of trying to make developers' lives easier led them to a different approach. When a new user registers for a Stripe account, the current version of the API is looked up, and associated with the user's account. From that point on, any requests coming in from that account are treated as being on the version from when the account was created. At any time, a developer can get a list of the differences between their version and the current version. The developer can then chose to upgrade to the latest version at any time.

This process meant the constant evolution of the API was transparent to users, and everything just works. It also puts the decision to utilize new features completely in the control of a user, when they are ready to make a change. On top of the 70 versions currently supported, a new version is released about once per month, with very low maintenance cost.

PCI Compliance

The API story demonstrates how Stripe makes incremental changes in their product offering. They use a similar approach to making changes to internal infrastructure. One of these was a rewrite of PCI sensitive infrastructure from Ruby to Go. A complete rewrite may not sound like an incremental approach, but the team focused their work on small pieces and was able to roll out the changes more effectively and with lower impact.

The biggest challenge with PCI compliance is that anything that touches credit card details is in scope for PCI. Any infrastructure that handles credit card data has to be PCI compliant, which can be burdensome and an impediment to development and moving quickly. In general, one of the main goals of going through the PCI compliance process is to make as few things in scope for PCI as possible.

Tokenization

Although many components do not need the actual card number, it can be useful to recognize a specific card as one that has been through the system before. One such case is fraud processing, and being able to spot multiple transactions on a single card. Tokenization is the common technique of taking something that is valuable, in this case, a credit card number, and replacing it with something that has limited value, and only within the context of the system.

Stripe's system for tokenization is called Apiori. (Naming Tip: don't get too clever.) Apiori is a thin veneer on top of api.stripe.com, and all API calls first pass through Apiori. The tokenization system looks for PCI-sensitive elements and replaces them with a corresponding token identifying the cardholder data.

Apiori deliberately knows almost nothing about requests, just where to find the PCI-sensitive information. Other than handling PCI data, it's basically just an HTTP proxy. This should have meant a rewrite would be fairly straightforward, since the scope was so limited.

Stripe is primarily a Ruby shop, and the PCI code was written in Ruby as part of the initial launch in 2010. The process of making a database call to fetch the token and other steps involved a lot of I/O operations. The EventMachine library was chosen to help handle asynchronous I/O. While this system worked well for four years, it became difficult to maintain and understand. Load tests to plan for future growth revealed Apiori was becoming the bottleneck.

Because the service is narrowly-scoped, horizontal scaling was used, but eventually became cost-prohibitive. The decision was made to rewrite Apiori, and to rewrite it using Go. Stripe had already been looking at Go as a future development language, especially for low-level infrastructure problems. Improved concurrency and performance were some of the goals which contributed to the choice of Go.

The initial development in Go took about a month, and the team believed they had achieved feature compatibility with the old Ruby code. After internal testing and code reviews, it was ready to be rolled out. As with other changes, the desire was to proceed slowly and incrementally. Using a new server with the Go code, and an extremely low setting on the load balancer, 10 requests were passed through the new system, then it was shut off. Detailed analysis of the logs revealed some problems occurred. This provided an opportunity to better understand unanticipated aspects of the old code's behavior.

Unexpected Behavior

One example of unexpected behavior involved encoding and parsing of parameters, which was handled by Ruby's Rack library. Rack has a very permissive idea of what nested parameters can look like. For example, from Rack's perspective all of the following are valid encodings of the exact same parameter structure:

```
• source[number]=4242424242424242
• [source]number=4242424242424242
• [source][number]=4242424242424242
• ]][[[][[]]]source]]]]]number]]=4242424242424242
```

Not all of the problems encountered were encoding problems, but they serve as a good example of unexpected functionality built into the old system which now needed to be understood then written in Go. In some cases, Go had internal behavior that disagreed with the RFC, which also had to be accounted for. The main lesson learned came down to Postel's Law, "Be conservative in what you do, be liberal in what you accept from others."

The team's understanding of the existing behavior was always limited, but learning new subtleties did not necessarily lead to being able to write tests to simulate the full variety of requests coming into the system. Go's HTTP library does not generate mal-formed requests, resulting in very narrow testing of only well-formed input. Stripe solved this problem with a technique they named The Zoo.

The Zoo

The Zoo is just a fun name for example-based testing. Every time a new exotic request is seen, it is added to The Zoo. The request and response are captured, and the expected behavior of the API can be determined for a given input. Tests were created to gain confidence that every time something unexpected was seen, it became a new animal for The Zoo. This also benefitted from the ability to send a very small number of requests through the system for analysis purposes.

For two months, more and more code was incrementally rolled out. First, ten requests over a minute, then 1% of requests for an hour, then 1% of all requests, all during normal working hours. When the team had confidence that enough bugs had been identified and squashed, they were able to send all traffic through the Go code, with no major incidents.

The primary goal of increasing performance was clearly achieved by migrating to Go. While all Go requests processed in under 150 microseconds, the Ruby infrastructure had a minimum latency of 500 microseconds, meaning half a millisecond elapsed before anything occurred. The new code has been running in production for about two years, and is much more maintainable. The Go routine model makes it much easier to reason about and add new features.

The wholesale rewrite of any system can be challenging. The narrow scope of Apiori was a benefit, but the major factor in the success of the rewrite was a very slow, deliberate and incremental rollout to production. The code took only one month to write, and two months to test and validate. That extra time going slow early has meant years of stability on the new system.

The Oregon Trail

The final Stripe case study covers the migration from one AWS region to another, effectively a complete move between data centers. It was probably the most complex infrastructure project in Stripe's history. Again, incremental work, building progressively closer to the end goal made the migration successful.

Like many companies, Stripe operates within multiple availability zones within a region. In 2010, when Stripe was first using AWS, the only two AWS regions in the US were on the East Coast and the West Coast. Stripe chose the West Coast because most of their early adopters were Bay Area companies.

Only ten months after being set up in the AWS data centers in Northern California, Amazon opened a new region in Oregon. Running infrastructure in Oregon is about 10% cheaper than in California, and there is much more room for expansion. It also became clear that Amazon was focusing West Coast AWS development in Oregon and not in Northern California. Around the same time, AWS released their second-generation networking stack called VPC, the Virtual Pri- ►

vate Cloud. VPC wasn't released initially in the Northern California region. Within a year or two of being in AWS, it was clear that Stripe needed to move to the Oregon region if they wanted to cut costs and benefit from new AWS features. However, other priorities kept postponing the migration until mid-2015.

Migration Goals

Three main goals helped guide the thinking and major decisions during the migration. Goal number one was no planned downtime. Stripe simply cannot have planned downtime, because their user base is global, in many time zones, with different patterns of traffic. The second goal was to minimize the amount of time spent in a vulnerable state. In any data center migration, there's a period of reduced resiliency, with one foot in each data center.

The third goal was to minimize the impact on other teams. While other teams within Stripe were incredibly supportive of the migration project, every engineering team has its own responsibilities and problems to deal with. Requesting help from another team could impact that team's projects. One pattern was to identify places where several problems could be solved at once and benefit multiple teams, such as core infrastructure. For more specific problems that didn't have a generalized answer, repeatable solutions were created that individual teams could quickly apply. The overall plan was to hide the fact of running infrastructures in two different regions.

The migration was very complicated, with thousands of servers running about 150 to 200 distinct services. There were also about ten different stateful data stores, databases or queuing systems, which required additional thought and careful planning. While many issues, large and small, were encountered, a few examples provide insight into the challenges faced during the migration.

Challenges During Migration

The first problem was the network, with traffic between AWS regions potentially going over the public internet. A VPN with independent IP addresses was setup to provide secure communication and a globally routable IP space. This simplified connecting to either region by using a standard IP address.

Security was another challenge. Normally, security groups are used with AWS's firewall implementation to restrict traffic between nodes, but security groups don't work across regions. Stripe had to work around the default security behavior that blocks traffic from random IP addresses. The solution involved a cron job to poll the AWS API, list all instances, then generate a set of IP rules that should be allowed and put them into iptables. The system was named Rays, after the Rays lighthouse outside San Francisco, which guides ships safely through the fog. Rays was run on every host, and worked, but also became the source of new problems.

A production incident in the AWS API caused incomplete results to be returned. Rays dutifully took the incomplete results, and then blocked legitimate traffic. Another issue was that iptables use an internal table to track connections, which has a low default size. When the table fills up, connections start getting rejected. Central services which normally received many connections would reject legitimate traffic. This is all reasonable behavior for a firewall service's failure mode, but it did cause some scrambling to fix the issue. Luckily, because the team was moving slowly, the problem was found before running in production.

Databases can be a significant source of problems in infrastructure, especially during a major migration. While some could be excluded, others had to have replicated copies. For MongoDB, which has built-in support for replication, the migration was fairly straightforward once the network infrastructure was in place. Mongo has primary and secondary nodes, and an application can connect to any node, then be directed to the primary. Because all the IP addresses in both regions were accessible, when nodes were brought up in the Oregon region they would appear in the standard service discovery. The MongoDB migration is a good example of finding a high-leverage problem that could be solved once so other people didn't have to.

The Oregon and California data centers are about 30 milliseconds away from each other, meaning every query is suddenly 30ms slower, about a 30x increase in query execution time. Considerable effort was spent to limit the number of queries whenever possible. Load balancing was used to send small amounts of traffic between data centers, to observe the results. An unfortunate interaction with the least connections load balancing strategy meant the delay in polling caused more traffic to be redirected than planned. Instead of 1% of traffic, 20% was diverted. Fortunately, the change could be rolled back quickly, and nothing broke. Lessons were learned, and additional, gradual testing was performed.

The final migration occurred only after considerable planning and confidence in the preparation work. With owners for specific monitoring of dashboards, and reasonable time estimates for every step, the last effort to move to Oregon was completed over a two-hour period, with no major issues.

Key Takeaways

The first key takeaway is to find points of high leverage. Whenever possible, solve a broad problem, instead of trying to solve lots of little problems. Common components and infrastructure, such as consistent data stores, made it easier to find points of leverage to solve problems once.

Second, test early and test often, because surprises will always appear. Having a plan is extremely important. Without a plan and early testing, some of the issues wouldn't have been found until the night of the actual migration.

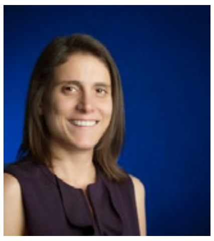
Finally, work incrementally in small steps and build on top of established layers. Being confident in one piece grants the ability to build on top of that. For Stripe, incremental changes are the most successful, cause the fewest problems and are the most effective way to make big changes. ■

"Work incrementally in small steps and build on top of sort of the established layers. Once you're confident in one piece, that gives you the ability to build on top of that—this has been really useful in general for us at Stripe; these incremental changes are the ones that are most successful for us."

- Evan Broder

Watch online on InfoQ

The Netflix API Platform for Server-Side Scripting



Katharina Probst is Engineering Manager at Netflix, where she leads the API team and helps bring Netflix streaming to millions of people around the world. Prior to joining Netflix, she was in the cloud computing team at Google, where she saw cloud computing from the provider side.

Adapted from a presentation by Katharina Probst, Engineering Manager at Netflix, at QCon New York 2016

Sometimes, it's a good idea for software architects and engineers to take a step back and think about what is really needed from the systems they develop and manage. What are the requirements and needs of the system? How are they changing? How is the surrounding ecosystem changing? Then, don't be afraid to realize that the system may be serving its users well right now, but may not be able to continue that service into the future.

The team responsible for the Netflix API has taken that step back and evaluated the API in the context of current and future needs. The analysis has led to major changes currently underway in the API platform for server-side scripting.

One way to think about the Netflix API is as the front door to the Netflix backend. All the microservices that Netflix runs have traffic flowing through the API. This includes new customer signup, billing, discovery, recommendations, ratings, movie metadata, and, of course, a lot of playback-related functionality.

The following diagram depicts how traffic from various client devices (PCs, TVs, phones, tablets, set-top boxes, etc.) flows through gateway systems, then the API, which fans out to the Netflix ecosystem of microservices. There are about 50 microservices that sit directly behind

KEY TAKEAWAYS

The Netflix API is transitioning from using server-side Groovy scripts compiled into the API to a layer of Node.js scripts in containers

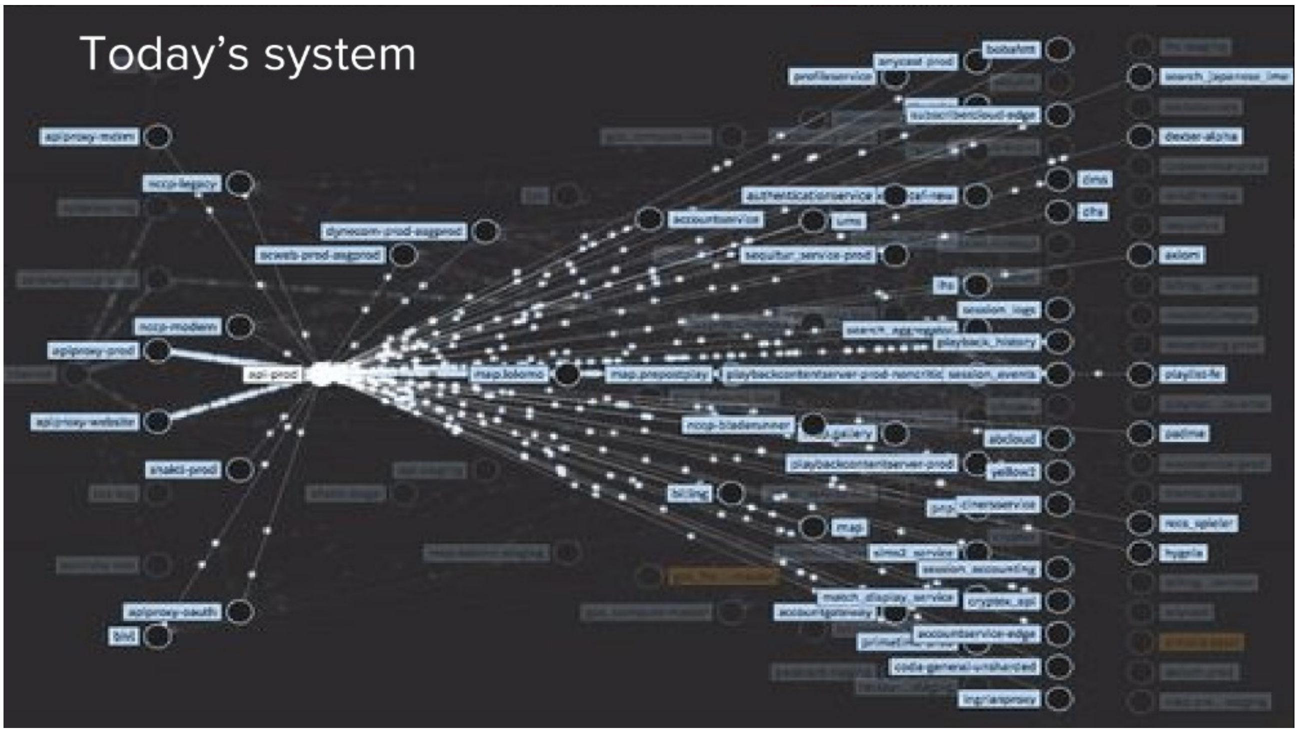
Non-functional requirements must be considered when making major design decisions

Four critical NFRs at Netflix are resiliency, great developer experience, flexible APIs, and velocity

Understand which non-functional requirements are most important in your environment

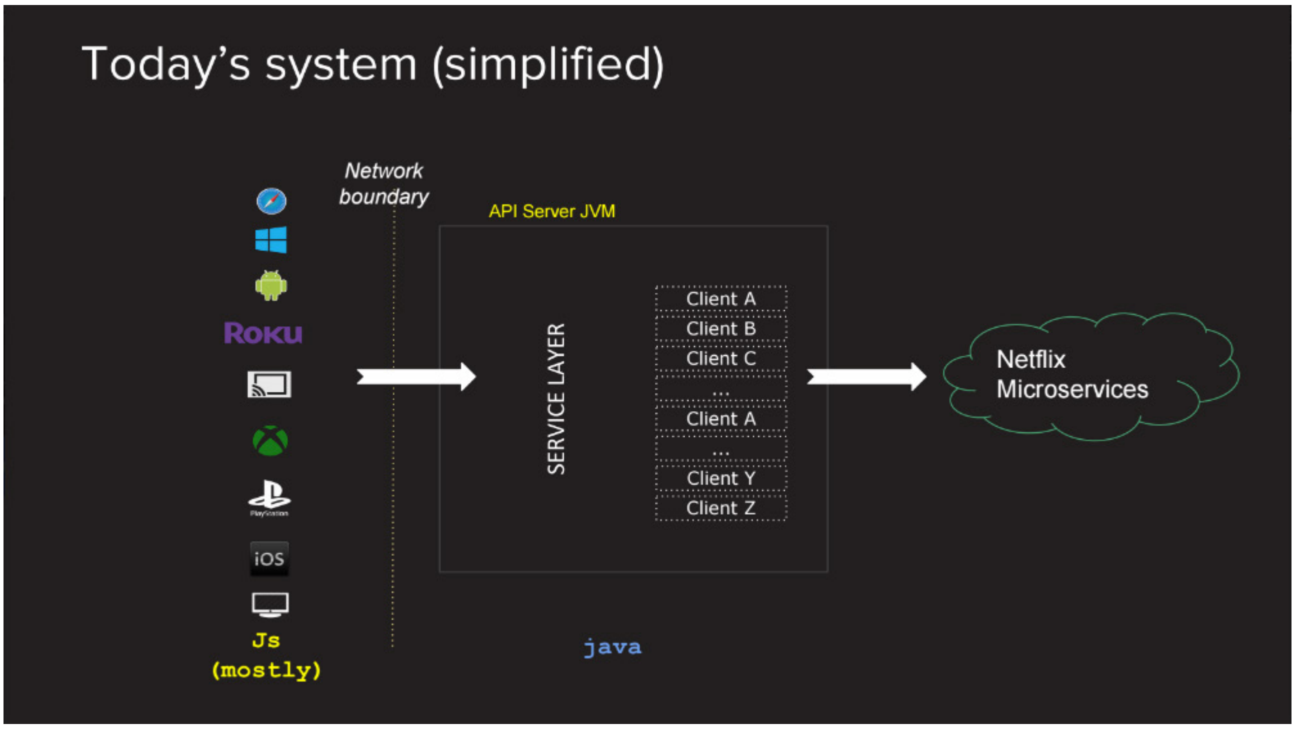
Don't be afraid to make big changes to key systems to ensure they can remain relevant and sustain future needs

the API, and then there are hundreds of them in the total ecosystem. The white dot at the center is the API, and will be our subject of focus.



►

For the purposes of this article, it will be more useful to talk about the system in terms of a simplified architecture diagram, to highlight how Netflix is making changes.



On the left are some of the device types Netflix runs on, representing more than a thousand different supported devices. The code that runs on these devices is mostly written in JavaScript, a fact that will become important later on. All these devices send traffic to the Netflix servers, mostly into the Java API.

The API exposes a service layer that is a unified API that all the device teams program against, and it provides access to all the Netflix microservices behind it. In practice, all the microservices expose client libraries, and the API uses those client libraries, embedded in the JVM, to access the services.

Non-Functional Requirements

With a basic context established, we need to understand what is really needed from the API. In addition to all the core features, the non-functional requirements (NFRs) must also be discussed and evaluated on an ongoing basis. Among many others, these can include low latency, low error rates, and great documentation. While many NFRs are important, four which are critical at Netflix are resiliency, great developer experience, flexible APIs, and velocity.

Flexible APIs

At Netflix, a flexible API means device teams can customize it for their own needs. This means the people who actually write the device code that runs on an iPhone or other device, also write the server-side logic. The server side logic, written as Groovy scripts, gets compiled and uploaded into the API and runs as part of the JVM. Today, Netflix has about 700 active scripts. This probably raises a question about what server-side logic do device teams write, and why is it needed?

The Groovy scripts are used for a variety of tasks, and some of the more obvious ones are formatting. Compare a mobile phone to a 50-inch TV and it's easy to see how the API needs for those devices can be very, very different. The screen real estate is different. The interaction models are different. This leads to writing very different formatting rules and rendering.

The server-side scripts are also used to implement A/B tests. Netflix is constantly evolving their systems and trying new features in the form of A/B tests. The Groovy scripts are a good place to implement new functionality for the tests. Netflix really cares about flexibility for devices, and the system in place supports that flexibility.

Velocity

The A/B tests are one way to maintain a high velocity. Constantly evaluating the system by writing new A/B tests is reliant upon teams being able to act and develop independently of one another. The practical implication of this is allowing new scripts to be uploaded at any point in time, and not having tight development cycles.

This also leads to completely decoupling deployments, which benefits velocity. The API has a regular cadence of deploying almost every day. Device teams have their own schedules, which could be once a week, or weeks with no changes, followed by daily publishing of multiple versions. Teams that support older devices may have scripts which have matured and don't need many updates. This is another aspect of the flexibility which is really important to the Netflix API team.

Resiliency

One of the challenges Netflix faces is resiliency. The current system actually works really well most of the time, with automated checks to prevent many issues. But, sometimes things go wrong.

The API sits in the middle of the devices and all the Netflix microservices. On the side facing the microservices, a lot of work has already been done to ensure resiliency. One of the tools to help is Hystrix, which detects failures or slowness of backend services. If Hystrix detects a problem, it will stop sending traffic to those services and serve fallback data. This adds a lot of resiliency. For example, if a personalization engine is down, the user may not get the most personalized experience when they log in to Netflix, but they can still explore, search and stream videos.

On the other side of the API, facing the devices, resiliency is more of a challenge. Obscure bugs, which aren't checked for, can cause unexpected failures. Scripts sometimes consume more memory or CPU resources than predicted or desired.

Many people may think it's like the Wild West to allow many teams to just upload new scripts to production, and that's a somewhat valid assessment. However, for several reasons, it works out really well most of the time. One reason is the Netflix culture of freedom and responsibility. The people uploading the scripts are internal developers, and understand the implications of their actions. They know what's at stake when they push new versions. Another reason is having protections in place to detect problems and recover very quickly.

This system has been around for three or four years, and as of one or two years ago, there were very few scripts, they were relatively small, and there were few uploads every day. Contrast that with today, and scripts have gotten a lot more complex as device teams have realized the power and flexibility of the system.

The production system now has about 700 scripts that run in the JVM, and dozens of uploads every day. This is great in terms of flexibility, creating a platform that people can really use and develop all kinds of complex application on. But it's also a good time to take a step back and evaluate if this is what the system was designed for, and if it will adequately serve Netflix well into the future.

Velocity vs. Resiliency

The complexity of the system is growing, and with it, the risk. Specifically, the lack of process isolation is a growing risk for the Netflix API. Several possible mitigation strategies have been considered, but all come with trade-offs.

One option is publishing scripts more slowly. When a new script is uploaded, it could be rolled out cell by cell, or region by region, but wouldn't be available globally immediately. This is already done to some extent. However, it flies in the face of developer velocity, which is really important at Netflix. Similarly, doing extra validation of memory needs before publishing flies in the face of developer velocity. Probst says, "In all these discussions, to me, it always feels like we're trading off velocity and resiliency."

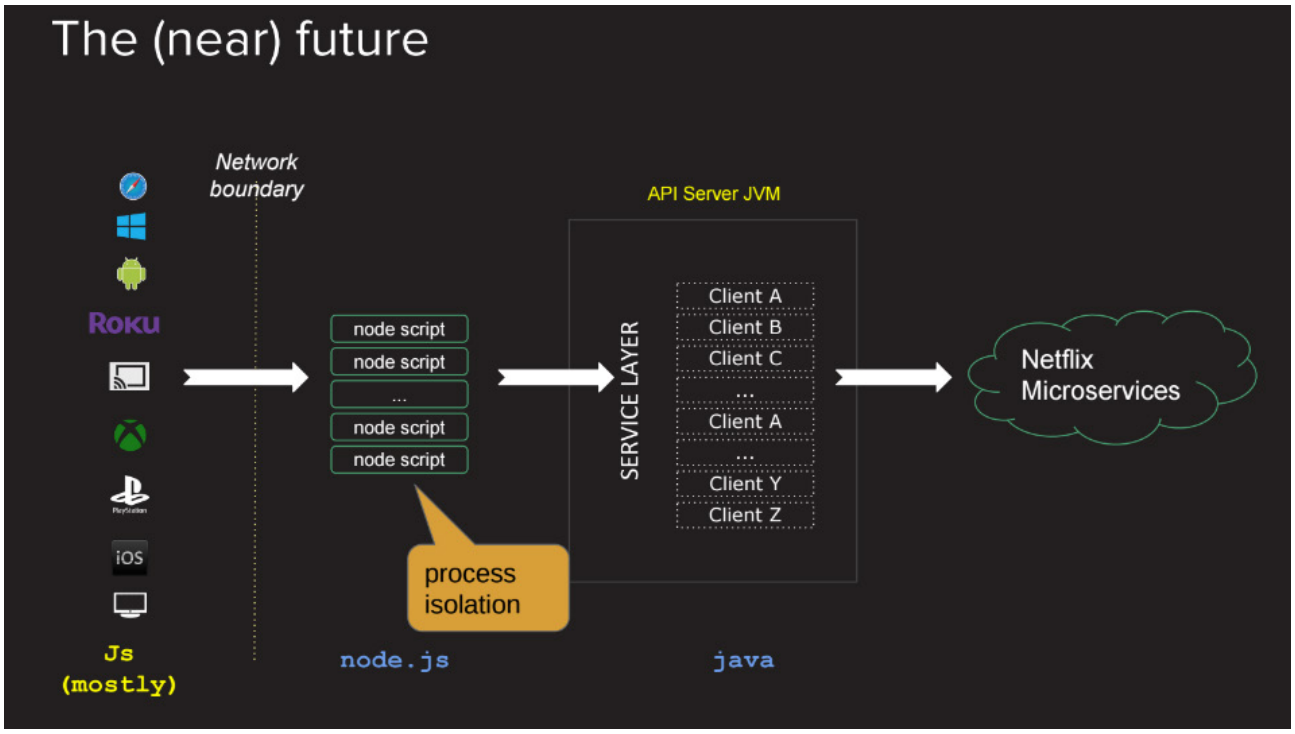
At Netflix, velocity and resiliency are both extremely important. In your systems, think about what's important to you, what trade-offs you face, and whether you actually have a system that solves all your needs.

Adding a Layer of Containers

Demanding both velocity and resiliency is challenging. Netflix is not afraid of making big changes, and is not afraid of putting a lot of work into creating their ideal API. They're currently working towards a system where scripts will run in containers, and call the API remotely. This creates a slightly modified version of the architecture diagram, with a fundamental difference (see image on next page)

Introducing a new layer for the scripts between the devices and the service layer has several benefits. First, the new scripts will be written in Node instead of Groovy. Because most of the device code is written in JavaScript, developers benefit from the move to Node.

Most importantly, the new layer achieves the desired process isolation, addressing the currently growing risk. But wanting process isolation doesn't necessarily mean using containers. Again, being willing to make major changes to the system meant the team could look for additional benefits they could achieve ▶



when writing something new. In addition to resiliency, the team wanted fast startup and consistent developer experience across environments.

Process Isolation
Anyone moving from a monolith to microservices will be familiar with some of the steps of breaking a very large system into multiple smaller ones. In the future, when a script for one device is unavailable, the problem will be isolated to that device, instead of becoming a problem for all devices.

Independent auto-scaling is another benefit of the future system. Some devices have more traffic in one region of the world, while other devices have different traffic patterns throughout the day. These are just two cases where being able to auto-scale independently for devices is a significant improvement over today, where everything sits in the API and must scale together.

Fast Startup
The API currently takes minutes to startup, whereas containers take only seconds, but why does that matter? Netflix wants the ability to roll out new versions quickly, and, more importantly, roll back quickly after a problem is detected.

One concrete example of this need is when a major problem is detected and traffic is failed over to another region. When this happens, the API needs to scale up in the other region. The time it currently takes to bring up new API servers has an impact on the ability to respond to problems. One hope with a new architecture is improved agility to respond to these scenarios.

Great Developer Experience
Just as resiliency and fast startup were important considerations that led to the new architecture, a great developer experience was an equally important goal. This means the developers have a good experience when they develop, they are productive, they can find problems quickly, and have a very quick turnaround cycle.

In today's system, step-through debugging requires a cumbersome local setup which is difficult to get working and takes time. Developers tend to rely on more rudimentary debugging techniques, including the use of print statements. The future state greatly improves on this setup.

A developer using the new environment will connect the local project with a local Docker container. A file watcher watches for changes in the local project and updates the container as needed. A Node Inspector is attached to the container and the debugger, providing the desired debugging experience. A network agent helps connect the local instance to the rest of the Netflix ecosystem in testing environments.

There's also room for improvements in optimization over the current system. Today, an uploaded script becomes part of a very big server, with a lot of shared dependencies. It becomes very difficult to optimize the performance of the scripts because of differences in calling behavior and resource utilization.

Running each script in its own container, then calling the API remotely means it becomes much easier to measure how each script behaves and how it can be optimized. This also provides the ability to throttle traffic if a sudden increase from one script is detected.

Self-Service Management

Currently, when device teams write new scripts, they get uploaded to the API and a dedicated API team actually operates the API service. Tools are being built in the new system to help teams deploy in seconds, and then self-manage the operation of their code. New UI and CLI tools should provide a lot of basic operations features already hooked up for the development teams. When a script is deployed to the cloud, it should come with life cycle management, dependency management, auto-scaling and tooling and insights.

A new deployment pipeline will handle the complete build, test and deployment process to roll out new versions of a script. Logging is another common feature that will be handled automatically, to ensure development teams don't have to be concerned with hooking up to the internal Netflix logging mechanism. New services become discoverable, and also have standard dashboards automatically created.

The CLI being developed is called NeWT, the Netflix Workflow Toolkit. NeWT is another example of Netflix having unique problems that require custom solutions to be developed from scratch. Other examples include the telemetry system ATLAS, and the container platform Titus.

Evaluating the Changes

Netflix doesn't just plan for the happy path, and works hard to anticipate what will inevitably go wrong and how to recover quickly from failures. One way to evaluate the new API for potential issues is by sending shadow traffic through the new system, using very specific use cases and a very specific set of devices.

When the system was first setup to handle the shadow traffic, the new API boxes died within hours. Luckily, the testing setup ensured no live impact occurred. The new system made it much easier to pinpoint the root cause, in this case a memory leak in the API server. A few days later, another issue appeared, a memory leak in the Node script, and was also easier to identify than in the old system.

Request tracing has also proved extremely valuable, being able to understand the fan-out behavior of the new system. As in most microservice architectures, fan-out can be quite complex. When a particular pattern is observed, it can be evaluated and identified as either expected or a potential point for optimization.

Bringing it Full Circle

The new API system being developed at Netflix will ensure legacy technology does not become a factor that limits the company's ability to respond to change. As the API evolves, the team always keeps the important non-functional requirements in perspective. The current system provides flexible APIs and velocity, and it's important to always check to make sure the new system still provides those both. Netflix also cares about resiliency and providing a great developer experience, and the new API makes measurable improvements in those areas.

Don't be afraid to change your system if you need to in order to meet the functional and non-functional requirements. Probst summarizes this very well, saying, "Progress is impossible without change, and those who cannot change their minds cannot change anything. If we get this right and we keep evolving our systems to the changing needs, then we won't actually have any legacy systems to deal with. And I think that's a competitive advantage, too." ■

PREVIOUS ISSUES