FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

QCon

2018 Retrospective

LONDON 4-8 MAR

9-11 APR **QCON.AI**

NEW YORK 25-29 JUN

5-9 NOV **SAN FRANCISCO**

InfoQ

Browse by Event

LONDON

QCON.AI

NY

SF

FOLLOW US

facebook.com /InfoQ

@InfoQ

linkedin.com company/infoq

youtube.com /MarakanaTechTV

CONTACT US

GENERAL FEEDBACK feedback@infoq.com
ADVERTISING sales@infoq.com
EDITORIAL editors@infoq.com

Browse by Topic

ARCHITECTURE & DESIGN

AI, ML & DE

CULTURE & METHODS

DEVOPS

CONTRIBUTORS

Charles Humble

took over as editor-in-chief at InfoQ.com in March 2014, guiding our content creation including news, articles, books, video presentations and interviews. Prior to taking on the full-time role at InfoQ, Charles led our Java coverage, and was CTO for PRPi Consulting, a renumeration research firm that was acquired by PwC in July 2012. He has worked in enterprise software for around 20 years as a developer, architect and development manager.

Ben Linders

is an Independent Consultant in Agile, Lean, Quality and Continuous Improvement, based in The Netherlands. Author of Getting Value out of Agile Retrospectives, Waardevolle Agile Retrospectives, What Drives Quality and Continuous Improvement. Creator of the Agile Self-assessment Game. Ben is an active member of networks on Agile, Lean and Quality, and a frequent speaker and writer. He shares his experience in a bilingual blog (Dutch and English) and as an editor for Agile at InfoQ.

Manuel Pais

is a DevOps and Delivery Consultant, focused on teams and flow. Manuel helps organizations adopt test automation and continuous delivery, as well as understand DevOps from both technical and human perspectives. Co-curator of DevOpsTopologies.com. DevOps lead editor for InfoQ. Co-founder of DevOps Lisbon meetup. Co-author of the upcoming book "Team Guide to Software Releasability". Tweets @manupaisable

Rafiq Gemmail

is currently Technical Lead with Bank of New Zealand. He is a passionate advocate for mob programming, having supported cross-functional teams through over a year mobbing at New Zealand's largest news site. He shared his learnings on this at JSCon NZ in 2017. Raf is also a champion for DevOps culture and one of the organisers of New Zealand's DevOps days. He is also an ICAgile certified coach.

Wesley Reisz

is the QCon Chair & Community Advocate for London, San Francisco, and New York. Before joining C4Media, Wes spent 14 years with HP and was an Enterprise Systems Design Lead for HP Enterprise Systems. As an HP Solution Architect, Wes' primary roles supported the US Army's Human Resources Command (HRC), Army Recruiting Command, and Army Cadet Support Program based at Fort Knox, Kentucky.

Daniel Bryant

is leading change within organisations and technology. His current work includes enabling agility within organisations by introducing better requirement gathering and planning techniques, focusing on the relevance of architecture within agile development, and facilitating continuous integration/delivery. Daniel's current technical expertise focuses on 'DevOps' tooling, cloud/container platforms and microservice implementations.

Jan Stenberg

is working as an IT consultant since more than 25 years in northern Sweden, experienced in building systems on both .Net/C# and JVM/Java platforms. His experiences range from large distributed and service based systems through web based and rich client applications down to hardware related software.

Srini Penchikala

Srini Penchikala currently works as a senior software architect in Austin, Tex. Penchikala has over 22 years of experience in software architecture, design, and development. He is also the lead editor for AI, ML & Data Engineering community at InfoQ, which recently published his mini-book Big Data Processing with Apache Spark. He has published articles on software architecture, securiaty, risk management, NoSQL, and big data at websites like InfoQ, TheServerSide, the O'Reilly Network (OnJava), DevX's Java Zone, Java.net, and JavaWorld.

LONDON 4-8 MAR


QCon

Introduction

by **Charles Humble**

QCon celebrated its 12th year in London during 2018. Located at the Queen Elizabeth II Centre just steps from Westminster Abbey and Big Ben, this year's event attracted 1,350 tech leaders in software and offered 136 technical deep dives, open spaces, and AMAs (Ask Me Anything) to attendees.

While the conference featured 18 curated tracks that covered nearly all of the major trends in software today, there were three that seemed to thread throughout the conference: Artificial Intelligence / Machine Learning, Microservices, and Ethics.

Andrew Ng famously called AI "the new electricity". Dave Snowdon discussed building neural networks with DL4J, Tim Kadlec asked us to consider the role our algorithms play in important life decisions for our users, and Eric Horesny led a full-day track that covered the full spectrum of how AI is affecting our lives.

Microservices has moved from the stage of figuring out how to decompose the monolith to actually operating them. Sam Newman's day-one track on Microservices covered scaling, debugging, observing, and security with a microservice architecture. In the Architecture's You've Always Wondered About track, Expedia's Mariano Albera discussed how the company re-platformed

their B2B APIs while continuing to support their $5 billion business.

Everywhere we look in software, we are starting to see larger and more pressing ethical concerns. Anne Currie (chief strategist at Container Solutions) and Gareth Rushgrove (product manager at Docker) led one of the first ethics tracks we've seen in software. This all-star lineup featured talks like "A Young Profession Coping With Ethical Debt" from editorial-board member of *ACM Queue* Theo Schlossnagle and Tim Kadlec's "Focusing on What Matters" (where Tim challenged the audience with "it's up to us to build a web that truly is for everyone").

QCon London 2018 featured over 140 speakers from innovative software shops like Google, Sky Bet, Honeycomb.io, Data Artisans, Mozilla, UBA, Uber, and more, and the keynote lineup this year was especially noteworthy. @RealMrGloverman tweeted: "I have attended the last 4 @qconlondon and I can honestly say the closing keynote from @RichardWiseman is the highlight of them all. Outstanding. Thanks for another great year #qconlondon team."

Opening the 2018 conference, Rob Harrop (CEO at SKIPJAQ and co-founder of SpringSource) delivered a motivational talk to push developers to open the black box of machine learning. Harrop's talk illustrated that artificial intelligence and machine learning are approachable by all of us (it even spawned an infographic on the topic). Additionally, Laura Bell (founder of New Zealand's SafeStack), Randy Shoup (VP engineering at WeWork), and Richard Wiseman rounded out the 2018 keynote lineup with talks on culture and security, imposter syndrome, and a recipe book for changing your luck surface.

Here we present some of our favorite writing from InfoQ's reporters at the show.

Read online on InfoQ

Ensuring Data Consistency in Distributed Systems Using CRDTs

by **Jan Stenberg**

Conflict-free replicated data types (CRDTs) is a family of algorithms for ensuring strong eventual consistency in distributed systems without the use of a centralized server, which has now been theoretically proven to work, Martin Kleppmann claimed in a presentation at QCon London, where he explored algorithms allowing people to collaborate on shared documents and how they can be used to synchronize data.

Kleppmann, researcher at University of Cambrige, noted that the problem with people editing the same document simultaneously has been studied since the late 1980s. The goal is to find algorithms that can achieve convergence — at the end of editing everybody has the same document on the screen.

One family of algorithms are operational transformations (OT). Several algorithms have been developed but most of them have failed, leaving documents out of sync with each other. The only ones that have succeeded are the ones using a centralized server, which is used in, for instance, Google Docs. Kleppmann thinks

this is a severe restriction — the data must go through a datacenter somewhere even when you are syncing some data between a mobile phone and laptop that are connected to the same local area network.

CRDTs make up another family of algorithms, used for, instance, in the Atom text editor. They resolve conflicts in a peer-to-peer fashion without a centralized server, but still with a convergence guarantee. Given the problems with operation transformation algorithms, Kleppmann and his colleagues wanted to formally prove that CRDTs converge in all circumstances, even in odd edge cases.

To prove convergence, they used Isabelle, a theorem-proving software, and were able to show that the CRDT algorithms they tested satisfied a consistency property called strong eventual consistency. They also had a second layer of proof using a model of a very unreliable network and could prove that the algorithms satisfied convergence guarantees in all possible executions.

An implementation of their ideas is Automerge, which is a data structure library on top of which you can build collaborative applications. The library provides a JSON-like data structure or abstraction that can be modified as a normal JSON document. Automerge is written in JavaScript and is open source, but Kleppmann emphasizes that this is research code, it is not production-ready.

Two theoretical issues that are still not solved are how to handle an undo of a change, and moving subtrees in an atomic way. Kleppmann notes that moving subtrees is not the same as deleting and reinserting it because that could lead to duplicates in some scenarios.

A significant problem is the overhead in memory space, and this is one of the areas where Kleppmann currently works. Documents with up to about 100k characters can normally be handled, but over this size, you may run into heap-size issues. By representing the data in an efficiently binary-packed way, Kleppmann has decreased the overhead to less three times the size of the plain text, which he believes solves the problem for most cases.

Jonas Bonér

Read online on InfoQ

Jonas Bonér at QCon London

Events Are Reshaping the Future of Distributed Systems

by **Jan Stenberg**

There are many reasons why we should care about events; they drive autonomy, increase stability, help us move faster and allow for "time travel", Jonas Bonér noted in a presentation where he explored how events are reshaping modern systems.

Bonér, inventor of Akka and founder of Lightbend, has been using events for 20 years but noted that there has recently been increasing interest, which he believes depends on four things:

- cloud and multicore architectures,
- microservices and distributed systems,
- data-centric applications, and
- customer demand — "we want more, and we want it now".

Bonér started by describing the nature of events. They are immutable facts of information. We can only add events or facts, which means that knowledge only can grow. Events can be disregarded, but once accepted, they can't be retracted. Events can, however, be invalidated by later events.

Event-driven services receive and react to facts (events) that they receive. Within a service, we can have mutable state, but that must be fully contained and non-ob-

servable. Instead, the service reacts by publishing new facts to the outside world.

In an event-driven world, everything is eventually consistent, with all events moving asynchronously. There are no transactions or strong consistency, which Bonér believes should be the default. For him, strong consistency should be a rare thing we use when there are no other options. He emphasizes that we must rely on eventual consistency, as this is how the real world works.

As soon as we exit the boundary of a service, we enter a nondeterministic world, which we call distributed systems or the network. In the space between services, spectacular things can happen: messages can be dropped or reordered, and so on. However, the network also gives us solutions; we just have to model the uncertainty. Bonér refers to Pat Helland and his paper "Life Beyond Distributed Transactions":

In a system which cannot count on distributed transactions, the management of uncertainty must be implemented in the business logic.

Bonér claims that we need to fundamentally change how we think about failures: they are natural and will happen. For him, it is ironic that we in many languages call failures "exceptions" even though there is nothing exceptional about them, as they are expected. He thinks we should manage failures instead of preventing them, and we can use events to help us with that.

In a sane failure model, failures need to be:

- contained to avoid cascading failures — for instance, by building fully autonomous services;
- reified as events;
- signaled asynchronously;
- observed, by one or many; and
- managed, but outside of the failed context.

Bonér agrees that all this looks hard. Something he thinks can help is events-first domain-driven design. He notes though that we should not focus on the structure too early. We should start by focusing on behavior, on the verbs and events. A way to do that is to work with intents and facts. Intents are mapped commands and represent an intent to do something, and facts are our immutable events.

The ability to update data is a main problem for Bonér, because that means you can mutate data, which for him is just wrong. He refers to Jim Gray and his paper "The Transaction Concept" from 1981:

Updates-in-place strike systems designers as cardinal sin: it violates traditional accounting practices that have been observed for hundreds of years.

Bonér therefore thinks we should learn the basic principles of accounting or bookkeeping and apply them to the systems we build today. The way to do this is through event logging, which is the way many relational databases have worked for years — the transaction log is event logging.

Event sourcing is a pattern on top of event logging that helps us in applying these principles to our systems. Here we store all the events in the order they arrive, which gives us access to the full history of everything that has happened in a system. To create a view of current state, command-query responsibility segregation (CQRS) is one way where you can subscribe to events and create different views according to the needs. A disadvantage of event sourcing is that it's an unfamiliar model for most software engineers. The versioning of events can also be a challenge.

Bonér concludes by emphasizing that events-first design helps us:

move faster towards a resilient architecture,

- design autonomous services,
- balance certainty and uncertainty, and
- reduce risk when modernizing applications.

Event logging allows us to:

- avoid CRUD and the use of ORMs,
- take control of your system's history,
- time-travel by going back in time via replaying events, and
- balance strong and eventual consistency.

Read online on InfoQ

Asynchronous Event Architectures with or without Actors

by **Jan Stenberg**

Synchronous request-response communication in microservices systems can be very complicated. Fortunately, asynchronous event-based architectures can be used to avoid this, Yaroslav Tkachenko claimed in a presentation in which he described his experiences with event architectures and how actors can be used in systems built on this architecture.

Tkachenko, senior software engineer at Demonware, started by emphasizing that you cannot make synchronous requests over the network behave like local ones. There are a lot of challenges related to network communication that can be hard to overcome, especially if you lack experience working with microservices and distributed systems. For him, an event-driven architecture is a more straightforward approach to solve the problems that may occur.

At Demonware, they still have a big application monolith but are slowly migrating towards independent services. Historically, they have used a lot of synchronous request-response communication, but are currently working towards a more asynchronous communication and have decided to use domain events between services.

A typical service at Demonware is based on domain-driven design (DDD) and the hexagonal architecture, and with Kafka as a message-transport layer. Events received are transformed to commands in an event adapter before being sent to the business-logic core. Correspondingly, events created by the service are added to Kafka, using fire-and-forget or a guaranteed at-least-once delivery, backed by a remote or local event store.

In short, Demonware are using an event framework with:

- decorator-driven event consumers using callbacks,
- reliable producers,
- non-blocking IO, and
- Apache Kafka for transport.

A way of simplifying this design is to use actors, as they natively support message passing, both for consumers and producers. Briefly, actors:

- communicate with each other using messages in an asynchronous and non-blocking way;
- manage their own state; and
- can create child actors when responding to a message, send messages to other actors, and stop child actors or themselves.

When Tkachenko started to work with the actor model, he realized that the whole system started to look like a messaging or event-driven system. For patterns in this area, he recommends the *Enterprise Integration Patterns* book and Vaughn Vernon's book *Reactive Messaging Patterns with the Actor Model.*

For an implementation based on the actor model, Tkachenko referred to Bench Accounting, a company that transformed a large Java enterprise monolith into a series of Scala microservices and actors using Akka. Key components in this implementation were ActiveMQ to handle message queues; Apache Camel, which is an integration framework; and akka-camel, an official Akka library, although it's now deprecated and replaced by Alpakka.

In a typical event listener in this implementation, akka-camel gets messages from the queue and deserializes, translates, and does other work needed before sending them to the actor. A message sender is similar; the actor uses akka-camel for serialization and other work before the message is sent to the queue.

In short, this implementation used an event framework with:

- actor-based consumers and producers using Apache Camel,
- producers with ACKs,
- non-blocking IO, and
- ActiveMQ as transport.

Tkachenko learned these lessons working with actors:

- Semantics is important. Natural message passing is a huge advantage.
- Asynchronous communication and location transparency by default make it easy to move actors between service boundaries.
- More advanced advantages include supervision hierarchies, the cultivation of a "let it crash" philosophy, and excellent concurrency.

He had recommendations:

- Domain-driven design (DDD) and the book *Enterprise Integration Patterns* are useful for bootstrapping your understanding of this space.
- Understand your domain space and choose the concepts you need among events, commands, and documents.
- Explicitly handle all possible failures, as they will happen eventually.
- Avoid exactly-once semantics unless absolutely necessary, because this is very expensive.
- Message formats and schemas are extremely important. Use binary formats like Protobuf or Avro. If using JSON, it's important to design a schema-evolution strategy early on.

Tkachenko related some of the challenges he has experienced:

- This is difficult work, and it takes time to move from a synchronous paradigm to an asynchronous world.
- High coupling will kill you. Avoid coupling by using events and minimize the use of commands.
- It's straightforward to implement event-based communication for writes, but really challenging for reads. DB denormalization, in-memory data structure, or stream processing are common solutions to explore.
- Managing a message-broker cluster is still a non-trivial problem.

The question of whether you should use actors depends on your requirements. Tkachenko notes that it's possible to build asynchronous, non-blocking event frameworks in many languages, but actor-based frameworks are asynchronous and message-based by default, which is an advantage when you need it.

Read online on InfoQ

Bernd Rücker at QCon London

Common Pitfalls in Microservice Integration

by **Jan Stenberg**

In a microservices architecture, every microservice is a separate application, typically with its own data storage, and communicates with other microservices over a network. This creates an environment that is highly distributed, and with that come challenges, Bernd Rücker explained in his presentation at QCon London 2018.

The talk further explored common pitfalls in microservice integration and solutions that included workflow engines, either running as embedded or available as a service. Please note that this presentation was not recorded, at the request of Rücker.

Communication

Communication is complex, and this complexity should not be hidden; instead, a service should be designed to handle failures internally. Rücker, co-founder of Camunda, uses an example from his own experience where he tried to check in for a flight and expected a boarding pass in return. Instead, he got an error message that they had technical issues and a request to try again later on. For him, this is bad design and a typical problem where the service provider, instead of requesting

that the customer retries, should handle the error themselves and send the boarding pass when they have managed to create it.

The same behavior can be applied in service-to-service communication. If a service can resolve failures internally, it should retry and asynchronously return a response when it is available. This encapsulates the error handling, thus making the API both cleaner and simpler.

Rücker calls this "stateful retries" and in his experience a common reason for not using this pattern is the perceived complexity of the state handling needed. Often, state is stored as a persistent entity or document, but a scheduler and other components are also needed to handle the retries, for instance. His recommendation is to use a workflow engine to take care of all these details. He also points out that even if retries often should be used, there may be use cases in a domain where error handling should be left to the client.

Asynchronicity

For Rücker, there are many advantages using asynchronous communication, and often this implies using messaging. A problem that then arises is timeouts. In his boarding-pass example, had this process completed using messaging yet the boarding-pass message never was created or somehow got lost, it would again be up to the customer to handle the failure. What is needed on the server side is some form of monitoring that discovers messages lost or arriving late. This commonly is done using messaging middleware, but Rücker has met some customers that have implemented this with a workflow engine. However, this requires an engine that behaves like a message queue by using the pull principle.

Distributed transactions

In distributed systems, ACID transactions don't work unless you try two-phase commits, which currently are generally seen as too complicated — as evidence, Rücker referred to Pat Helland's paper "Life Beyond Distributed Transactions: An Apostate's Opinion". Instead, Rücker prefers long-running business transactions in cases where you must do several activities in "all or nothing" semantics. One solution for this is the saga pattern, where you work with multiple steps, eventual consistency, and compensations if something fails.

To use sagas, every involved service provider must offer compensation activities and Rücker strongly recommends that they also be idempotent. In network communication, there are three failure scenarios that you cannot differentiate among:

- The request didn't reach the service provider.
- The request did reach the provider, but it failed during processing.
- The request was processed, but the response from the provider was lost.

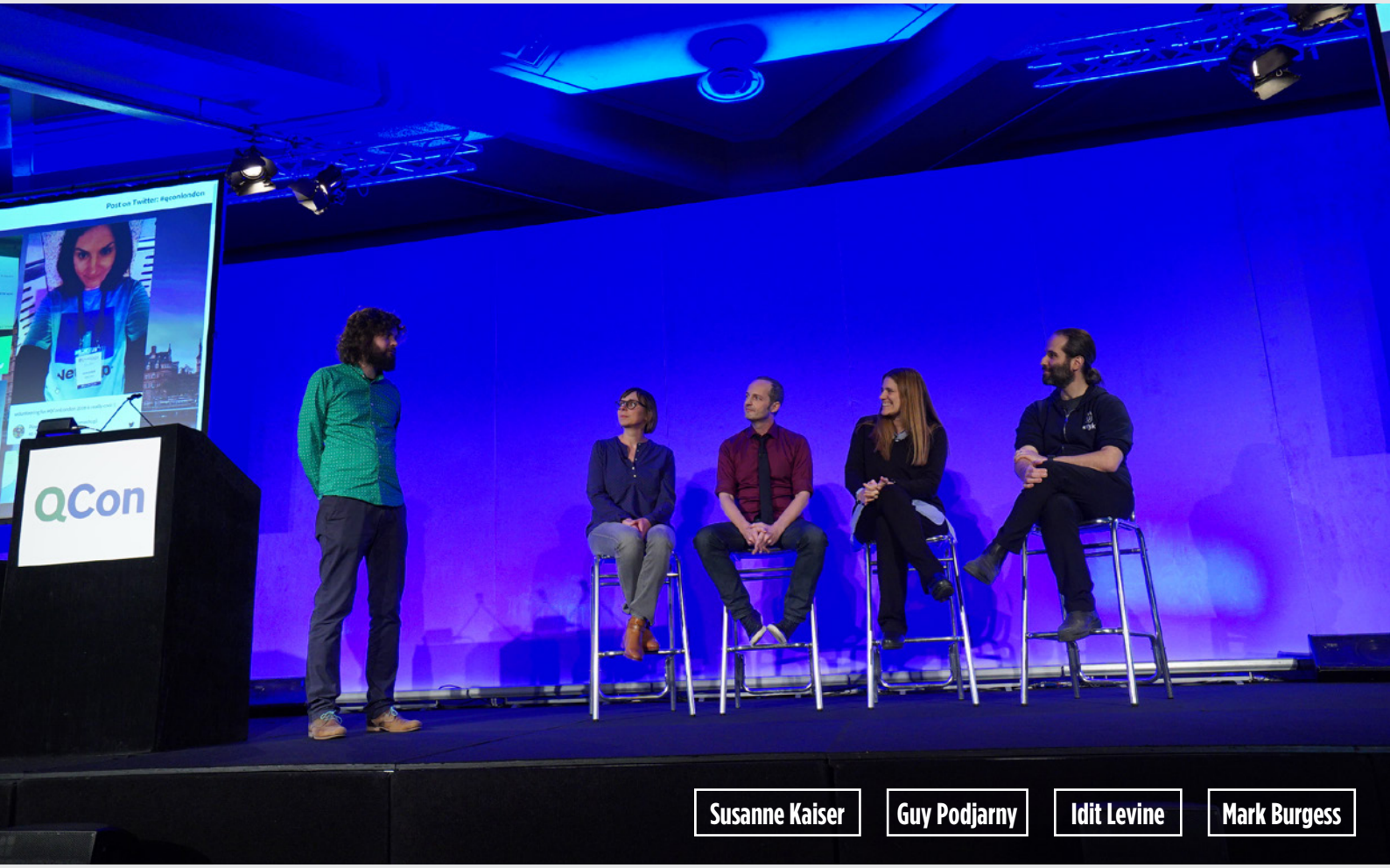
One solution when an error is detected is to ask the service provider about the request, but this means it must be possible to distinguish it from other requests. The common approach is to just retry the request, but this means it must be idempotent, and Rücker mentions four types of idempotency:

- natural, for instance when setting a specific state;
- business, where you have a business identifier, like an email address;
- unique ID, generated by the client; and
- request hash, where the service recognizes a request by a hash of the message.

Rücker notes that an embedded workflow engine can implement the saga pattern, and points out that in a microservice-based system there are commonly multiple engines within the different microservices, each handling different workflows. He emphasizes that the engines are embedded; there is not a central engine that every workflow has to pass through.

Looking into the space of workflow engines and state machines, he notes that there are several open-source frameworks, and that new frameworks have emerged during the last year or two. In the serverless space, AWS has created Step Functions, and other cloud vendors are at least thinking in this direction.

Rücker has published sample code that implements his ideas.



Read online on InfoQ

Microservices Panel Discussion

The Future of Microservices and Distributed Systems

by **Jan Stenberg**

In the microservices panel, participants discussed how service technology will change, and how we will build systems in the future.

Microservices track host Sam Newman asked if microservices will fade away and become an implementation detail, and the panel — consisting of Susanne Kaiser, Guy Podjarny, Idit Levine, and Mark Burgess — believe that microservices will continue to exist, but will evolve into becoming a base for other techniques like serverless architectures when building distributed systems.

- Kaiser, ex-CTO at Just Software, believes microservices will be the basis for more distributed architectures in the future, with serverless being one example, and will be a good foundation when building distributed systems.
- Podjarny, co-founder at Snyk.io, believes microservices will be embedded in the way we build software, but that will also include operational concepts.
- Levine, founder and CEO of Solo.io, noted that when we build systems, we later always extend them and microservice architectures is the key to be able to do that.

- Burgess, principal founder of CFEngine, claimed that microservices are just too hard for ordinary people to do. He believes that service-mesh technology is a start to something that eventually will become some sort of a «mega compiler» that will take away the necessity of dealing with all the smaller services and functions that we work with today.

Newman, author of *Building Microservices*, then asked if serverless technology will change how we build systems in the future — is it the next evolutionary step?

- Levine noted that the purpose of a function is to glue services together. For her, serverless is an interesting ecosystem, but she is not sure how it will evolve.

- For Kaiser, serverless takes care of a lot of commodities, but right now will easily lead to vendor lock-in. With more varieties coming that will abstract away the vendors, it will then be a more interesting path to follow.

- Burgess sees this as an evolution to function as a service (FaaS). He noted that we still have very primitive interfaces where we manually couple things together. In the future, he believes it will be much more user-friendly and compiler-like, and more integrated.

- For Podjarny, FaaS aspires to be a back end, or software as a service (SaaS), within which we also consume third-party services, so we write less of the code ourselves. But he doesn't think serverless is the end of the road; in the next steps, we will be creating diagrams describing how functions are connected in an application.

When asked if the characteristics of a microservice have changed during the seven years since the term was coined, the panel agreed that domain-driven design (DDD) is still an important concept. Kaiser noted that microservices have forced them even more towards DDD — for instance, when finding candidates for a microservice. She also noted that all the software architecture principles are still applicable, with DDD having a huge impact.

Regarding vendor lock-in when starting a new development initiative at a company, the panel agreed that they would try to avoid it. Kaiser would definitely go for a cloud-native approach — allowing a focus on the core business. Burgess noted that the standardization of the industry is still immature, but thinks this will change in the coming years.

If starting a next venture or product, Kaiser would probably start with microservices if the organization has the capability; otherwise, she would start with a monolith. Levine warned about creating technical debt and would start with microservices, which she feels very comfortable with. Podjarny's experience is that microservices is more of a pain from the beginning, but you are also set up for the future.

Read online on InfoQ

Demystifying Machine Learning for Development Teams and Children

by **Rafiq Gemmail**

QCon London 2018 opened on March 5 with a keynote by Rob Harrop, titled "Artificial Intelligence and Machine Learning for Software Engineers". Machine-learning (ML) expertise, according to Harrop, often sits behind a siloed wall between development and data-science teams.

These divisions can lead to the development of models that are divorced from an understanding of the data and its underlying domain. In addition, software teams are often removed from being able to develop their own competence due to such divisions and the aura of mysticism around ML. Complementing this, Dale Lane spoke in the sponsor stream, and demonstrated how he has been making ML accessible to children through declarative and accessible tools in combination with practical coaching around the edge cases of ML.

Harrop is the CTO of SKIPJAQ and an original founder of SpringSource. His keynote highlighted the dangers of reintroducing handovers between specialist silos; this time between data specialists and teams wanting to apply ML capabilities. Harrop spoke of the need to avoid introduction of bias when working with data specialists who lacked a contextual business understanding of

the development team's bounded context.

Lane, a developer at IBM, one of the conference's sponsors, demonstrated Machine Learning for Kids, which offers web-based tools targeted at educating children in machine learning. The website builds on MIT's Scratch, a visual platform used for teaching programming. It offers intuitive interfaces that enable children to create programmable flows with ML capabilities. A simple interface allows users to train models used for image recognition, natural-language processing (NLP), sentiment analysis, and detecting other patterns.

Lane discussed how, through working on practical examples, he has been able to make children conscious of data quality issues such as overfitting or introducing a data bias. Using the example of training a recommendation model for theme parks or funfairs, Lane presented his class with a dataset bias towards funfairs. With the resulting model, which preferred funfairs, Lane was able to challenge children to think about the moral consequences and impact on individual livelihoods and success of both businesses. Encouraging children to consider the recommendation of life-saving medication, he told the story of how they became better aware of the moral significance of such data bias.

Harrop brought up the issue of the EU's General Data Protection Regulation (GDPR), which would soon make it illegal for organizations to use data that discriminates on the basis of personal beliefs, religious background, ethnicity, sexual orientation, or political affiliation. He highlighted the dangers that a model may still be able to learn inherent patterns and apply similar bias, even with pre-filtering of data. Addressing this, Harrop recommends that developers engineer and test a solution so that it doesn't have an unwanted bias. He felt that of "all the sociological issues around machine learning, bias is the most important one".

Both Harrop and Lane talked about the mystification of machine learning in society. Harrop reminded the audience that while there is often a focus on the need for data specialists who understand the underlying theories; for most use cases, using ML is just another software engineering activity. Reducing this barrier to entry even further, Lane talked about how less-technical teachers were able to start to understand, teach and effectively use applied ML through the Machine Learning for Kids framework.

Harrop's summarized that ML is a key competitive advantage, but at its heart it is mostly software engineering. He warned the audience to avoid the mistake of old, stating:

Don't try to have a data science team and a software team. Co-locate those things together. Make sure everyone has an understanding of what everybody else is doing.

Lane shared a number of immediately accessible web-based ML platforms which children and adults can start experimenting with:

- Machine Learning for Kids,
- Teachable Machine, and
- Moral Machine.

Read online on InfoQ

Q&A with Laura Bell

Continuous Security

by **Rafiq Gemmail**

The practices of continuous and shift-left security attempt to reframe security away from being a gate after development, making teams responsible and informed guardians of their systems and user experience.

InfoQ talked to Laura Bell, a leading advocate and teacher of continuous security, to discuss her recent talks at QCon London, her professional security journey, and how she continues to champion a consciousness for team-owned security.

Bell is a founder of SafeStack and co-author of the O›Reilly book *Agile Application Security*. She has been hosting QCon London›s Security track and delivered both a second-day keynote and a talk on how unqualified fear can have negative repercussions on architecture choices. Bell›s keynote, "Guardians of the Galaxy: Architecting a Culture of Secure Software", further discussed the need to weave security into a product-development team of guardians with the tools to enable rapid and safe innovation.

In her talk, Bell advocates moving security away from being driven by fear and toward being a considered and continuous part of rapid innovation. She discussed a tendency to introduce visible and experience-jarring gatekeepers, such as captcha, rather than building seamless validation on

the server side. She points out that in a microservices world we need to move away from gatekeeping on less predictable boundaries to using a pattern of guardians, which monitor and respond to our services.

Intentionally expressing security concepts in generally accessible terms, Bell explained during her talk that the important thing about being a security person today is being able to tell stories and collaborate, "to go into a team using an analogy to bring about an insight".

InfoQ's Engineering Culture and Methods Trends Report for 2018 classified shift-left security in the innovator phase of our adoption curve. Consistent with this, Bell described continuous security as blending appropriate safety into the development process, making it repeatable, automated, team-owned, integrated, and scalable.

InfoQ: What were the main messages you wanted people to take away from your keynote?

Laura Bell: Make do with what you have. Don't aim for perfect. Aim for bringing security into things you're already doing.

The keynote's theme was about being conscious that when we're fixing security issues and trying to integrate security into everything we do, this requires us to architect a culture, the same way we'd architect everything else.

We talked about the principles of security at speed, our operating contract, and how we find new people and work together.

InfoQ: You also talked about data-driven security. How can data be used effectively for security?

Bell: It's all about knowing more about what's happening security-wise. If a client finds a bug in their mobile apps, what information do we need to know to find out how many people this is affecting and how quickly those people are no longer vulnerable? Can we detect that it is being misused by looking at changing behavior?

For instance, in Android, we are always asking what versions and what devices? We look at who is running the new version and who is running the old version. After we announce there is a new version available, how quickly does that curve go up? iOS released some interesting stats about the adoption of new versions, revealing that if they released new emojis, people would upgrade.

Security is never the reason most people update. They aren't that interested. Historically, we find a bug, release a fix, and say our job is done. From my point of view, our job is not done, not until that fix is in front of as many of our users as possible.

InfoQ: Your talks discussed how unqualified fear can skew the perception of risk. What can teams do to ground their fear and meaningfully prioritize based on risk?

Bell: Michael Brunton-Spall's talk on attack trees presented a good tool for this.

It's about taking the conversation much further than "X is going to happen and it's going to compromise our database." Take it into the specifics of how would they do this? How hard is it? Is it possible? What would they do with that information?

When I teach, I ask a group of people what the most valuable thing in this room is. They always answer that it is themselves. We move on to company secrets. If you are a malicious person, what are you going to do with these? Sell it to the competition. What happens if you contact them?

They might say they'd sell it on the Dark Web. Which may be right, but the Dark Web is full of all sorts of terrors and you don't just walk in there with a dataset and sell it. Getting them to go down this pathway, it's about really understanding the realities of what you do and don't know. It becomes more concrete. There isn't just a villain in another company, who you can call on the villain number.

InfoQ: How can security specialists and UX work effectively together to build really good products?

Bell: Firstly, you go listen to UX. Don't talk at them. Understand their world a bit. I've worked with UX teams and asked, "how would you write a phishing attack that looks genuinely like it's from your company?" They get it. UX people also love data. They love knowing where on the page people have clicked and the conversion rates between features. They measure everything; it's fantastic. If you add extra measurements and surface them, you can make them more aware of the impacts of their decisions. What was the impact on average quality of passwords when we made some change to the password interface? Do we see behaviors changing?

I really like what Slack does right now for their (login) challenge. If you log in on your mobile app, it says, "We assume your password is hard. If you don't want to type that in right now, we'll send you a magic link." So, you don't have to sign in with a password at any time in Slack. When they have an outage, their spare developers go on social media and handle calls. It's a very human-centric security model. It's about understanding the motivation of users.

Microsoft put out a post a few weeks ago saying they were going to get rid of passwords from Windows. I'm hoping they are going to be more human-centric in that.

InfoQ: How are such moves received by security professionals?

Bell: There is a familiarity with the way we do things. If you tell a security person "we're going to get rid of passwords," they get butterflies at that point.

I do think that progress is being made on the server side, away from the users. With monitoring and response, signal sciences, heuristics, and machine learning to spot patterns, we can be more user-focused on the customer side. We aren't just a username and password. We can assume that's been compromised by now. What are the times of day we connect at? What order do we normally do things in the application? What music are you listening to when you do this action? Is there a pattern? If we can tell what a normal version of Laura is, why don't we respond to it not being a normal version?

The funny thing is that in security we need to justify that we're doing this. We have all this data and are going to put it all together for your safety. In marketing, they have been doing it for years.

InfoQ: In your keynote, you described an approach to hiring where candidates would be asked to plan a burglary, to establish cultural fit. How well does this work in practice?

Bell: We tend to get the right fit. We've learned a lot about how people behave in the interview processes. If you bring an introverted person into an interview and say "rob a bank," that does not sit well. It tends to favor extroverted people.

The first few times, we didn't account for people not wanting to fail. It's an interview and you want a job. How do you prove you are awesome when you're making stuff up on the fly and there's no engineering, and you've got no text book to help you? It's not a test, so we coach them. Once they relax a bit, they realize it's OK to fail. It's exciting and energetic. You can see the person underneath it all. If they focus on the physical things, that tells you something. If they focus on the computers, that tells you something else.

InfoQ: In acting as host for the Security Track at QCon London, have you recognized any particular themes arising from the audience and speakers?

Bell: There is a definitely a lot of curiosity and a lot of good practical "how do I do this?" questions coming from the audience. We've gone past the stage of 'what does this even mean?" to "I'm trying to do this; help me." I think we've reached a maturity space.

InfoQ: Do you have any positive examples of industries where you've seen a good balance of innovation with a risk-based understanding of security?

Bell: The payments industry has some really great teams. I've seen full unit-test suites in Ruby using immutable architecture. Not on the code but on infrastructure as code. They've merged this with predictable security testing.

There are giant telcos rolling out security-champion programs to 2,000 developers. It's all well and good to run such a program across 50 developers. How do you scale that to seven countries across 2,000 devs? It's not all in the countries you'd expect.

InfoQ: What can organizations do to emulate these patterns?

Bell: We've all come into security from different places. Some of us were firewall engineers, software developers, or risk and compliance people. We're from all sorts of places and don't always have an understanding as to how teams are working now. We need to build some bridges internally to make sure security is pulled in and not pushed out. That the people in security teams are given as much education and training as our development teams. It has to come top down and there has to be management buy-in.

There have to be baby steps. No one wakes up one morning and has a Spotify tribe model. That is the result of doing lots of little bits of work. Go do the little bits of work. Build some bridges. Make some friends. Figure out where your pain points are. Measure all this stuff.

InfoQ: What are some of the other projects you're currently working on?

Bell: It's really easy to do a risk assessment when your system is deterministic. In the AI talks, there is the notion that you can introduce bias if you put bad data in. How can you rationalize and predict risk when you can't predict what it's going to do at all? Here's my machine-learning, black-box component. I want to see what tools we can build to help do the threat assessment in that space and understand how you can measure how badly it can go.

Also, I'm a people person. I started being very technical early in my career. Now a lot of what I do is culture and communication. I would love to do some more work with people like UX people and lawyers. To try and understand where the crossovers are. The more we cross-pollinate with the other industries the richer we become. I essentially want to make friends with all of these fields and say, "here's my world, tell me about yours!"

InfoQ: What advice would you give to individuals who want to begin their own journeys into security?

Bell: Security is a massive field. Watch a ton of conference talks. From QCon, Black Hat, and all the many which have been over the years. If you're bored after 10 minutes, stop it there.

Find out which of the vast array of these technologies and techniques motivates you. Ask "what if?" Go deep on that. Passions drives effort and motivation. If you've got passion, passion is contagious. You can share that interesting talk with people.

There are some famous talks. Barnaby Jack in 2008, when he exploited cash machines live on stage in Vegas. Charlie Miller in 2015 when he stopped a jeep on a highway. Apollo Robbins's TED talk where he shows you misdirection on change. You›ll watch the video twice, I promise you.

All of those things have lessons you can bring to your engineering role. Go find passion and excitement.

Kevin Goldsmith

Read online on InfoQ

Q&A with Kevin Goldsmith

Data-Driven Thinking for Continuous Improvement

by **Ben Linders**

Kevin Goldsmith, chief technology officer at Avvo, spoke at QCon London about building a culture of continuous improvement.

Organizations need an objective way to measure performance and tie actions back to business outcomes to improve continuously, Goldsmith said.

Avvo uses a data-driven decision framework with an autonomous-team model and a practice of retrospectives to help people make better decisions and proposals for continuous improvement.

InfoQ interviewed Goldsmith about what a culture of continuous improvement looks like, what he did to build a foundation for continuous improvement, how agile retrospectives and data-driven thinking fit in, and how autonomous teams and accountability go together.

InfoQ: What made you decide to start working on the culture? What triggered it?

Kevin Goldsmith: My experiences at Spotify taught me the value of building an engineering culture deliberately. When I came to Avvo, the engineering culture had evolved organically. It had

some valuable elements, but also some weaknesses. There was a lack of focus on delivery and poor personal-interaction models. It was clear to me that we needed to invest in creating a better culture to improve the delivery and morale of the organization.

InfoQ: What does a culture of continuous improvement look like?

Goldsmith: It's a culture where there is no status quo or "because that is how we do things here". People are encouraged to find better ways of doing their jobs. We allow trials of new approaches. If they work well, we adopt them. We not only give people ownership over their work but also over the organization and its processes. One example of this is our Journey Team structure. We knew that our existing team model wasn't working well for the individuals on the teams, but they never got "permission" to fix it. After moving towards our new culture, an ad hoc group of product and engineering team members came together to design a new team structure. Journey Teams are now our organizational model.

InfoQ: What did you do to build a foundation for continuous improvement?

Goldsmith: It was an extended effort that involved creating a solid foundation of multiple elements. First, we created a framework to bring clarity to the company strategy and priorities. For people to make good suggestions, they need to understand the company's business context. We then created a data-driven decision framework to help people make better decisions and proposals. We devised an autonomous-team model, which allowed (among other things) a narrower scope for experimentation. Finally, we built a practice of retrospectives, beyond the normal agile team practices, encouraging and demonstrating how to examine the results of experiments in a non-judgmental way. The retrospectives help us learn from our failures and successes.

InfoQ: How do agile retrospectives fit in?

Goldsmith: Retrospectives are critical in a culture of continuous improvement. They collect the lessons from any change. Without a blameless evaluation and an understanding of the results of an experiment, you are bound to repeat your failures and will be unable to recreate your successes.

InfoQ: How did you apply data-driven thinking for improvement?

Goldsmith: In a creative environment, ideas flow like water in a mighty river. With many choices, how can you determine the best course of action? If you only have opinions to base decisions on, the loudest (or most highly paid) person chooses. Data democratizes the decision-making; it clarifies the options and potential outcomes. We created our own data-driven decision-making worksheet influenced by the Toyota A3 form and Spotify's DIBBs.

Recently, I was talking to Avvo's chief product officer about a change I wanted to make to our data infrastructure, which would require some effort from our data engineers and would take them away from supporting the product team. He asked me where the decision worksheet was. I hadn't made one yet. In the course of filling out that worksheet, I found data that made me realize the change I wanted to make wasn't going to give the company a good ROI. I recognized that it was a bad idea.

(The concept of DIBBs — data insights, beliefs, bets — and how Spotify uses them for continuous improvement is described in the article "Spotify want to be good at failing".)

InfoQ: How do autonomous teams and accountability go together?

Goldsmith: Autonomous teams and accountability don't go together automatically. You need to build a structure so that teams feel responsible for their outcomes. Organizations need an objective way of measuring their performance and tying their actions back to business outcomes. Without this objective measure, an autonomous team can lose their sense of accountability. The team doesn't understand the value of their work for the business. An objective measure, agreed upon with the larger organization, also gives the company leadership visibility into the value that the team is producing, without having to disrupt their work.

Read online on InfoQ

Finding Talented People and Building Sustainable Teams

by **Ben Linders**

Vlad Galu, VP of engineering at GlobalSign, presented "Building and Growing Sustainable Teams" at QCon London 2018. Meetups, hackathons and conferences are fantastic opportunities to promote your company's work and ethos and meet talented people.

You can learn a lot more about a person if you let them drive the conversation initially in a job interview, argued Galu. Having room to grow professionally and psychological safety are key to building sustainable teams and establishing a collaborative and cohesive engineering culture.

InfoQ interviewed Galu about finding talented people, creating sustainable teams, dealing with differences in culture in different parts of organizations or on different continents, and how the engineering culture has evolved over time at GlobalSign.

InfoQ: What works for you to find talented people who want to work in your company?

Vlad Galu: There are several factors that contribute to talent attraction, but a few stand out to me:

1. Companies make sure their top talent is happy where they are, usually, so it is hard to reach good people through conventional recruitment channels. I find direct networking to work really well. Meetups, hackathons and conferences are fantastic opportunities to not only meet talent, but also promote your work and your ethos. Everyone wants to be part of something great and meaningful; learning about your mission straight from the source offers people a more genuine experience than hearing about it from a third party.

2. I often see job ads asking for least X years in this or that industry, irrespective of the specific set of skills required. That is usually another way of saying "we do not have the time or inclination to explain our processes and tools; we'd rather you knew them already."

I currently work for a digital-identity/PKI company with a primarily B2B business model. If we narrowed our requirements that much, the talent pool we could tap into would have been astronomically small. A lot of the challenges we face can easily be found in other industries. Our people come from areas such as airline ticketing, cloud storage, telecoms… all places with interesting, hard problems that we also solve.

Other companies explicitly prefer people proficient in specific tools and processes — such as programming languages, frameworks, databases, SaaS platforms — instead of looking for problem-solving skills. That may work well for short-term gigs with a very narrow scope, but that is not necessarily conducive to a long-lasting relationship, which would require more flexibility. We have worded our job description around concepts and approaches instead of tools.

InfoQ: What have you learned and what have you stopped doing when recruiting people?

Galu: I have shifted the emphasis completely on what people know, rather than what they do not, which reflects on all stages of the hiring process.

Online/timed coding tests were the first to go. While suitable for situations where you need to hire lots of people quickly, I find them very one-dimensional and narrow-scoped. They will tell you what people know but they will not tell you how they think. The people you screen may be able to answer your questions but are they able to ask the right questions themselves?

My approach is devising a take-home test that covers several

bases (in our case algorithms, systems-level programming, and code readability), allows access to any documentation, encourages research before implementation, and is subject to a more generous time boundary that does not employ a stopwatch. We mixed in some intentional tradeoffs, which give us a preview of a person's thought process and strengths, as well as great solution variety. In the three and a half years I have been with my current company and after more than a hundred interviews, we have not received two identical solutions, which goes to show how differently people think.

The second thing I changed was the format of the on-site technical interview. Many companies I worked for or interviewed with opened with a static list of questions that was sometimes relevant to the role but less often relevant to the candidate's background. To emphasize what people do know over what they do not, we first ask them to tell us more about past achievements, projects, and challenges to make them feel more comfortable. We go further down that path, slowly raising the question difficulty level, crossing into different areas and taking them out of the comfort zone. How they handle reaching the breaking point is a quite-good indicator of their personality and professional maturity.

InfoQ: What is it that makes teams sustainable?

Galu: To me, a couple of things:

The opportunity of bettering oneself, day in and day out. Having room to grow professionally, learning to address the "what", "how", and "why" challenges of the business. You can cover the "what" and "how" by building teams around a common core of skills while aiming for a good mix of individual strengths; this way, everyone can learn from everyone else. Creating space to shift from one team, product, or technology to the next when possible is a huge loyalty booster. The "why" depends on the collective drive to move up the career ladder and is different from one place to the next. But it's still perfectly possible by proactively monitoring your reports' progress and their reports' progress to identify leadership strengths and aspirations and creating space for those to manifest. Aiming for a good mixture of junior and senior people helps ensure they all have room to evolve.

Psychological safety is key — not being afraid to make mistakes or share ideas. Evolution is a very long stream of mistakes; we fundamentally learn through trial and error. Smart and kind people make up fantastic teams in this regard.

InfoQ: How do you deal with the differences in culture in different parts of your organization or on different continents?

Galu: In my experience, joint projects where people work towards a common goal in the same room are great at bridging gaps in work and communication styles. Depending on the company and the product, it is possible to introduce short milestone-aligned sprints that bring together people who otherwise work together but in separate offices. Communication between product management and engineering is critical — the more face time, the better.

InfoQ: How has your engineering culture evolved?

Galu: It has certainly become more collaborative and cohesive. Before my time, engineers were clustered around products siloed in different markets, with little to no cross-team interaction, but we have since changed our approach and built a common platform supporting all products that is jointly owned and operated by several teams around the world. Working together has certainly helped overcome cultural barriers and made our people more comfortable with offering and receiving feedback. There is still room for improvement, as our product management is not as spread out as we would like, but we are making great strides towards that as well.

Read online on InfoQ

Software Engineering for Creativity, Collaboration, and Inventiveness

by **Ben Linders**

Dave Farley, independent software developer and consultant, presented "Taking Back 'Software Engineering'" at QCon London. Farley argued that a software engineering discipline must be iterative, based on feedback, incremental, experimental, and empirical. Craftsmanship is not sufficient.

Engineering is an amplifier; it enhances creativity, collaboration, and inventiveness. Continuous delivery is grounded in engineering principles. If you are more rigorous at the start of a project then you can create better, more innovative solutions, spend less time fixing bugs in production or hacking workarounds to deployments and configurations.

The term "software engineering" was first used in 1968 at a NATO conference. Although progress has been made towards an engineering discipline, software isn't engineering yet, according to Mary Shaw:

There has been a lot of effort over the years in software development methods. This established the production methods that support commercial practice. However, it did not establish the codified basis for technology that is required for engineering practice.

InfoQ spoke with Farley about what defines software engineering and why it is important, how craftsmanship relates to engineering, and what can be done to develop an engineering mindset.

InfoQ: How do you define "software engineering"?

Dave Farley: I think that there are a lot of misconceptions amongst software developers about what "engineering" is, let alone what "software engineering" is. I seem to get myself into a surprising number of conversations about bridge building for some reason. So, the first thing that I would say is "engineering" is NOT the same as "production engineering". Doing something for the first time is very different to doing something for the thousandth. Engineering is an intensively creative discipline! Think of the engineering that goes into the creation of something like the Curiosity Rover or the Falcon Heavy or the first iPhone!

My definition of engineering in this context is: Engineering is the application of an empirical, scientific approach to finding efficient solutions to practical problems.

For software engineering, I think that there are some specifics that MUST be in place for us to achieve this empirical, scientific approach. For me, any software engineering discipline worth the name must be:

- Iterative,
- based on feedback,
- incremental,
- experimental, and
- empirical.

My talk at QCon London explored each of these ideas in more detail.

InfoQ: What makes software engineering important?

Farley: Software is important in the world! Strangely, we software developers are the people that are probably changing the world the most at the moment. Ours is still a young discipline and we have been, and continue to be, growing explosively.

My perception is that maybe the majority of software developers have never seen something that we could honestly call "engineering" applied to software development. When we do apply these principles, what we see is fairly dramatically more efficient, in terms of speed and quality.

This is not surprising as science, in which engineering is rooted, is humankind's most effective problem-solving technique. I think that we should apply it to the very hard problem of software development.

Another angle on this is the evolution of software development as a profession. The VW emissions scandal has led to several people going to jail, including software developers. We are responsible for the decisions that we make and the code that we create, morally as well as economically. My view of our industry has been that often we ignore these responsibilities. Unless we grasp them, and take control, these things will be imposed upon us.

An engineer that builds a bridge or a car or an aeroplane that kills people will be held professionally responsible. We too will face that challenge. What kind of defense for our actions will we have if we have not followed effective disciplines for testing, measuring, and evaluating our ideas?

My last point though is more personal. Working with some engineering discipline is a LOT more fun. I can create better, more innovative solutions to problems. I can tackle bigger problems and spend less time fixing bugs in production or hacking workarounds to deployments and configurations if I am a bit more disciplined in the way in which I approach my work. In my opinion, engineering leverages and amplifies my capabilities without detracting in any way from the creative aspects of software development.

InfoQ: How does craftsmanship relate to engineering?

Farley: I believe that the software-craftsmanship movement was created in response to the horrors of big-ceremony, waterfall-style processes in the 1990s and early 2000s. Craftsmanship is a step forward to these planned approaches, which are fundamentally the wrong model for processes that are variable and include a significant amount of discovery.

Craftsmanship was the correct step, but if we look at the history of production, craftsmanship is a low-quality process. Think for a moment of a craftsperson-built iPhone or jet fighter. What about craftsman-led space program?

Craftsmanship is not sufficient. Engineering is an amplifier; it doesn't prevent creativity, collaboration, and inventiveness, it enhances it.

I agree with many of the ideas from the software-craftsmanship movement, in particular the ideas of apprenticeship-style training and continual learning and improvement, but these ideas are equally applicable to a software engineering discipline and such

engineering discipline takes us further, giving us routes forward when we are stuck and amplifying the quality and productivity of our work.

InfoQ: What can be done to develop an engineering mindset?

Farley: Ultimately, I think that this may be a generational change. If we could agree some principles for software engineering then it should be taught, it should be common practice, it should be expected!

I think that my principles are a good starting point: iterative, based on fast feedback, incremental, experimental, and empirical.

I am a bit biased in that I also think that a continuous-delivery approach to development is grounded in these engineering principles, but it must go deeper than that.

I think that there is a greater consensus about what really works than I have seen before, at least amongst the people that we may consider thought leaders in our profession. So now may be a good opportunity to, once again, try to define what software engineering really means.

If we could gain broad agreement on such a definition then maybe we could start advising educational establishments and professional bodies to teach the right things. A trivial example: how many people who studied computer science learned about the scientific method and the importance of experimentation on that course? How many people learned ethics as part of such a professional course? These things are pretty normal in most engineering disciplines.

InfoQ: How about the current generation of developers, juniors and senior tech people? What can they do to catch up with software engineering?

Farley: My number-one piece of advice for any software developer, junior or senior, is to adopt a more scientific approach to problem solving and take the scientific method seriously.

Think in terms of experiments, gather data to make decisions, try ideas out. Don't assume that your first guess is correct. In fact, assume that all of your guesses are incorrect and work in ways that 1) will help you discover how you are incorrect quickly and 2) won't kill you when you get things wrong.

InfoQ: What is the difference between a craftsperson and an engineer?

Farley: A craftsperson guesses what may work. An engineer guesses and then measures their guess to see where they got it wrong.

Read online on InfoQ

Q&A with Andy Walker

Dealing with the Broken Human Machine: How to Create High-Performing Teams

by **Ben Linders**

To really progress in developing software and build anything at a scale, you have to examine your blind spots and learn to deal with people (you are people so this includes you).

The culture we build is as important as the technology we use; the difference between a high-performing engineering team and a low-performing one is orders of magnitude in terms of productivity and quality. Focusing on how we do things is as important as what we're doing.

Andy Walker, engineering manager at Google, spoke about his experience with developing and coaching teams at Google at QCon London, and InfoQ interviewed him about building high-performing teams and establishing conditions that make engineering thrive.

InfoQ: What's the main challenge in building high-performing teams?

Andy Walker: We have this whole thing in a world where everyone wants to be a hero. And in reality, successful working code won't get you far; you can only write so much code before it stops being the best use of your time. The

more people involved, the more attention you need to put into HOW you do things rather than just WHAT you're doing.

If you're really going to progress, particularly if you're working in a larger company with other people, then learn to deal with people. Nobody ever teaches you how to deal with other people. We just kind of expected to figure out how as we go along. If you're going to invest time in something then — I think it was Dale Carnegie who said that — in any technical career 85 percent of your success is going to be down to your ability to deal with people and only 15 percent is going to be down to the technical skills. Boy, that's kind of scary.

I realized that I actually have to learn all the skills and I have to figure out systems that are working for me. Otherwise, I'm not really going to be able to do a job with it. It's just not enough anymore. I spend a pretty large amount of my time with Google just getting people to talk to each other.

There are some things and some behaviors that we do because it's convenient, but which really aren't good for us in the long run — things like the fact that we would rather send somebody a text message or e-mail than actually talk to them face to face, which is not a good way of actually communicating. Language is not very effective when it comes to getting ideas across, particularly in a world where we communicate via text rather than face to face.

Then there's how our fight/flight can be triggered, which frequently comes down to scripting, which is how behavior is habituated throughout our lives. These are the hardest behavioral loops to unpick because we're not even aware we're following scripts.

There are blind spots where the wiring in your head that you think works really well just doesn't. And once you start examining where your blind spots are, you can actually come up with much better responses to other people and also understand people's responses back to you and how to get the better of them.

InfoQ: How can we deal with this broken human machine?

Walker: Begin to learn about some blind spots. Learn models for understanding why we have those blind spots and become aware that there are parts of your brain where they have a very strong ability to influence your thinking, but you've got no conscious control there.

One useful model is called the triune brain, which is a model for how our consciousness might have evolved. Below the conscious part of our brain sit two animal layers which are roughly equivalent to a crocodile and a horse. The animal parts of our brains have particular motivations. For example, the crocodile brain cares about things like food, safety, warmth, and making little crocodiles. The horse cares about things like social acceptance and popularity. Blood flows through the animal parts of our brain to our conscious part. And the animal parts of the brain have the power of veto over unsatisfactory thoughts as well as the ability to boost ones that line up with their objectives. Therefore, we wind up contextualizing things in terms of our animal brains without realizing it. Advertisers also know this.

For example, as a 45-year-old man who is in a Ferrari dealership, it doesn't matter how beautiful the car is if I tell the crocodile that if we buy this car we have the opportunity to make many more little crocodiles. Then suddenly I have lots of approval for that chain of thought. If, on the other hand, I walk up to the edge of a tall building and think about jumping off, the crocodile objects and cuts off the flow of blood to that part of the brain. This also happens during a fight/flight reaction as the crocodile diverts blood to muscles that it considers more pertinent in a survival situation. This, in effect, stops your rational brain in its tracks. If you want healthy relationships then you need to not put people into situations where they go into fight/flight.

InfoQ: How can we establish conditions that make engineering thrive?

Walker: We need to accept that the culture we build is as important as the technology we use. Everyone needs to be able to contribute so we can take the best solutions from a wide diversity of ideas and perspectives. To do that we need to understand the big picture — to seek out the skills and knowledge we don't come pre-equipped with. Starting with ourselves and understanding of our own fallibility is a gateway to understanding both our peers and our users. Then comes understanding how we can make collections of people be a multiple of the sum of their parts rather than a fraction. This is where the role of a leader becomes important, as you adopt a servant-leader model where you are facilitating your teams rather than telling them what to do. Your job is to help people understand why they're doing things. Then we can help everyone improve how we operate. It's rare that I tell someone how to write a piece of code.

And it's more than just writing software at play here. It's the environment we work in, our ability to make decisions, to delegate, to learn from our mistakes, and to share and develop our ideas. The difference between a high-performing engineering team and a low-performing one is orders of magnitude in terms of productivity and quality. If we want to build anything at a scale beyond ourselves, this is a journey we need to take.

Examples:

- One thing that worked really well was helping people understand how poor a tool e-mail is. Having a rule to never respond when you were tired, grumpy, or felt an emotional response to an e-mail is tremendously powerful. We can write very abrupt prose, which has the potential to be misinterpreted. Better to go talk to the person and understand what they meant than responding straight away. It's unlikely they meant what you perceived, anyway.

- Encourage people to have more face to face interactions. The rapport you build when you're in the same room makes it easier to emphasize with the person on the other end of the e-mail.

- Encourage people to publicly recognize positive behaviors. Research shows people are twice as likely to help you in future if you say thank you or appreciate the effort. The more you recognize positive behaviors, the more positive behavior you see.

- Tell people to disagree with me and each other. Everyone should know why we're doing things and what problems we're trying to solve. And if I can't explain why then we're probably doing the wrong things. Also, I can be wrong. Be willing to admit that in front of people.

- Focus on what's important. I've told people that they choose how they spend their own time. If a meeting brings no value, don't attend it. Frequently, the number of meetings grows to become the dominant drain on resources. Have fewer meetings — do more useful things.

- Reduce interruptions so you can get into a state of flow. Inspired many, many years ago by the Joel Spolsky post on context switching.

- Everyone has to be able to succeed/grow. I use Dan Pink's talk on motivation here as a simple litmus test for people — if people don't feel a sense of autonomy, opportunity to learn/grow, or purpose then they're not going to be engaged.

- Focus on outcomes, not deliverables. Frequently, we get lost in a world of feature lists and bugs, which stops people looking at the big picture. You want to create an environment where anyone in the team can have the epiphany that enables you to build something remarkable. If success looks like "ship feature X" then you lose sight of the problem you're trying to solve.

- Focus on outcomes, not the one true way. There are many ways to skin a cat. We get caught up in holy wars on frameworks, languages, design patterns… you name it. But the energy we spend arguing about that detracts from the actual real-world problems we're trying to solve. At the end of the day, software engineering should be useful and if it's useful then people will find it easy to have a sense of purpose.

DELIVERY MUST CHANGE

PIPELINE SPRAWL.
CONFIG CHAOS.
MANUAL TOIL.

THERE IS A BETTER WAY.

GET THE DELIVERY EXPERIENCE YOU NEED

DO IT WELL.
DO IT ONCE.
DO IT IN CODE.

#DOITINCODE

ATOMIST

Read online on InfoQ

What Resiliency Means at Sportradar

by **Manuel Pais**

At this year's QCon London conference, Pablo Jensen, CTO at Sportradar, a sports-data service provider, talked about practices and procedures in place at the company to ensure their systems meet expected resiliency levels.

Jensen mentioned how reliability is influenced not only by technical concerns but also organizational structure and governance and client support, and requires on-going effort to continuously improve.

One of the technical practices Sportradar employs is regular failover testing in production (a kind of chaos engineering). Their fail-fast strategy is tested at an individual service level as well as at cluster level and even for an entire datacenter. The latter is possible because, as Jensen stressed, production environments are created and run exactly the same way across all datacenters. From a client point of view, they act as a single point of contact, whereas internally workloads can be allocated (or moved) to any live datacenter. Applications know as little as needed about the infrastructure where they are running, and they can be deployed the same way across on-premise and cloud (AWS) but the bulk of the work is done in Sportradar's own datacenters, for cost purposes.

Other common resiliency strategies employed at Sportradar include circuit breakers and request throttling, handled by Netflix's Hystrix tool. Jensen also mentioned decoupling of live da-

tabases from data warehousing to avoid potential impact of reporting and data analysis on live customers.

In terms of governance, Sportradar puts strong emphasis on managing dependencies (and their impact, because issues with third-party providers is still their number-one source of incidents), for example by classifying external service providers into three categories of accepted risk:

- "single-served" for non-critical services provided by a single vendor;
- "multi-regional" for a single vendor that offers some levels of redundancy (such as AWS availability zones); and
- "multi-vendor" for critical services that require strong redundancy, for which single-vendor dependency is not acceptable.

According to Jensen, expanding infrastructure to Google Cloud Platform has been in the cards in order to further reduce risk (thus moving the cloud infrastructure service from "multi-regional" to "multi-vendor"). Further, accepting that single-vendor services might fail means that dependent internal services must be designed and tested to cope with those failures. This focus on risk management also manifests internally, as each business area is served via independent technical stacks hosted independently on redundant services.

With more than 40 IT teams allocated to specific business areas, Sportradar also faced the need to set up some governance around the software architecture and lifecycle. Before a new development starts, it must pass a "fit for development" gate with agreed-upon architecture, security, and hosting guidelines in place. Perhaps more importantly, deployment to production must pass a "fit for launch" gate to ensure marketing and client-support teams are aware of the changes and ready.

Services are still improved after launch, as IT teams must follow a 30% rule whereby 30% of their time is allocated to improving current services' stability and operability, as well as improving existing procedures (such as on-call or incident procedures). Jensen highlighted the importance of iterating over established procedures, improving them and regularly communicating and clarifying them (not correctly following procedures is still their fourth largest contributor to incidents).

In terms of organizational structure, aligning IT (product) teams with business areas has worked well, with centralized IT and security teams providing guidance and oversight rather than executing the work themselves. For example, the centralized team defined the security development guidelines and iterated them for three months as the first product teams to follow them provided feedback on what worked and what not. Only then were the guidelines rolled out to all the product teams.

Finally, each service must have an on-duty team assigned before being launched. On-duty teams provide second-level technical support — roughly 0.5% of all 110,000+ client requests per year escalate to this level — throughout the service's lifetime. As Jensen stressed, only the best (and higher paid) engineers in the organization work in these teams, promoting a culture of client focus and service ownership. Clients are kept in the loop of any open non-trivial incident, which is followed by a post-mortem once closed. Jensen added that clients appreciate this level of transparency.

Read online on InfoQ

How Booking.com Uses Kubernetes for Machine Learning

by **Manuel Pais**

Sahil Dua, a developer at Booking.com, explained at this year's QCon London conference how they have been able to scale machine learning (ML) models for recommending destinations and accommodation to their customers using Kubernetes.

In particular, he stressed how the properties of a Kubernetes cluster — elasticity and resource-starvation avoidance on containers -— helps them run computationally (and data) intensive, hard to parallelize, ML models.

Dua detailed how the properties of the Kubernetes platform benefit his team and are key for Booking.com to use many ML models at large scale: customers daily book around 1.5 million room-nights and the site receives 400 million monthly visitors:

- **Kubernetes isolation** — Processes that run within Linux containers (and Kubernetes pods) can be isolated at the operating system level, and therefore can be orchestrated to not directly compete for resources.
- **Elasticity** — Pods running ML models can auto-scale up or down based on resource consumption.

- **Flexibility** — The self-service nature of Kubernetes and the rapid deployment of containers allows the team to quickly try out new libraries or frameworks.

- **GPU support** — Kubernetes offers support for NVIDIA GPUs (albeit this is still in alpha), which allows 20x to 50x speed improvement.

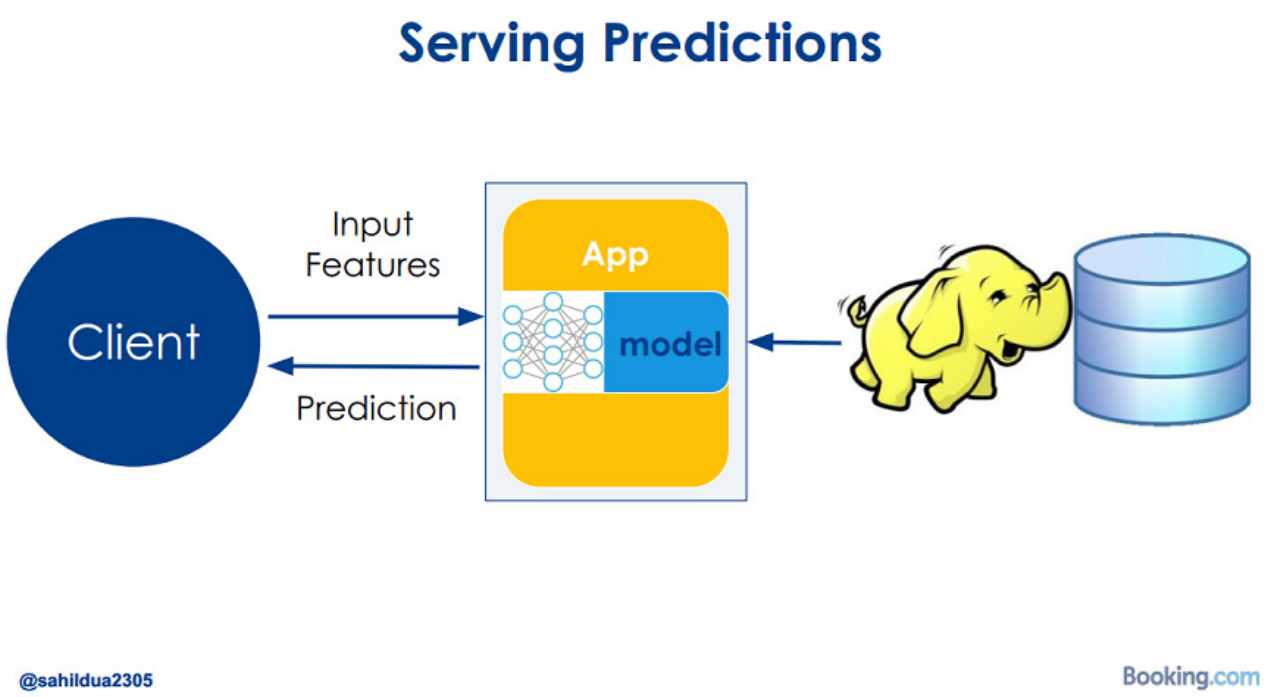
The declarative syntax of Kubernetes deployment descriptors is easy for non-operationally focused engineers to understand. Specifying that a pod requires a GPU resource tells Kubernetes to schedule it in a node with a GPU unit:

```
resources:
  limits:
    alpha.kubernetes.io/
nvidia-gpu: 1
```

Each pretrained ML model runs as a stateless app inside a container. The container image does not include the model itself, which is instead retrieved at startup from Hadoop. This keeps image sizes small and avoids having to create a new image every time there is a new model, thus speeding up deployments. Once deployed, the model is exposed via a REST API, and Kubernetes will start probing the container for readiness to receive requests for predictions, until finally traffic will start to be directed to the new container.

Besides Kubernetes's auto-scaling and load balancing, Dua revealed some other techniques used at Booking.com for optimizing latency of the models, namely keeping the model loaded in the container's memory, and warming it up after startup (by issuing an initial request to TensorFlow, Google's ML framework, where the first run is typically slower than the rest). Not all requests come from a live system; in some cases, predictions can be precomputed and stored for later usage. Optimizing for throughput (amount of work done per unit of time) is more important for the latter. Batching requests and parallelizing those that are issued asynchronous helped reduce the networking overhead and improve throughput, said Dua.

ML models need to be trained with preselected data sets before they are ready to provide the kind of predictions Booking.com needs. The training part of the process is also run on Kubernetes infrastructure. Base images for the containers where training takes place contain only the required frameworks (such as TensorFlow and Torch) and fetch the actual training code from a Git repository. Again, this keeps container images small and avoids proliferation of new images for each new version of the code. Training data is fetched from Hadoop clusters. Once the model is ready (training workload finished), it gets exported back to Hadoop.



Read online on InfoQ

Q&A with James Munnelly and Matt Bates

Kubernetes Stateful Services and Navigator

by **Manuel Pais**

The advantages of Kubernetes for stateless services are well documented. However, stateful workloads have particular requirements that have not been fully addressed yet by the Kubernetes ecosystem, according to James Munnelly, solutions engineer, and Matt Bates, co-founder of Jetstack, who presented at QCon London this year.

InfoQ asked Munnelly and Bates about their views and ongoing work to configure, deploy, monitor, scale, and auto-heal stateful services in Kubernetes in the same way as stateless services. In particular, they discussed in detail the approach and implementation of Navigator, an open-source Kubernetes extension they have been developing.

InfoQ: Why did you decide on a Kubernetes-only platform instead of a hybrid such as Kubernetes for stateless services and cloud-provider services for data storage?

Matt Bates and James Munnelly: Where managed cloud-provider services exist, such as Cloud SQL, there is a good case to use these in the pattern you describe. However, as more enterprises look to use Kubernetes in multiple environments, there is a desire for deployment and operational consistency and this is

not always achievable with different flavors of managed service. It is also the case that in some environments, especially on premises, such managed services simply do not exist. This is a situation for many of our enterprise customers.

There are already some efforts to integrate cloud-provider managed services into cloud-native applications with the Service Catalog project. We see Navigator integrating with Service Catalog to provider higher-level layers of abstraction than we offer today.

InfoQ: What are the fundamental difficulties you have faced in managing stateful workloads on Kubernetes?

Bates and Munnelly: Kubernetes provides many benefits to industry in terms of development velocity, resource utilization, and automated operations however it's fair to say that this has not translated across to stateful workloads.

Many common database systems make assumptions that they will be run on machines with fixed software versions, persistent disks, and network identity — pets, essentially. Few systems are designed for highly dynamic environments like Kubernetes, where pods can come and go and change identity, and services are round-robin load balanced, for instance.

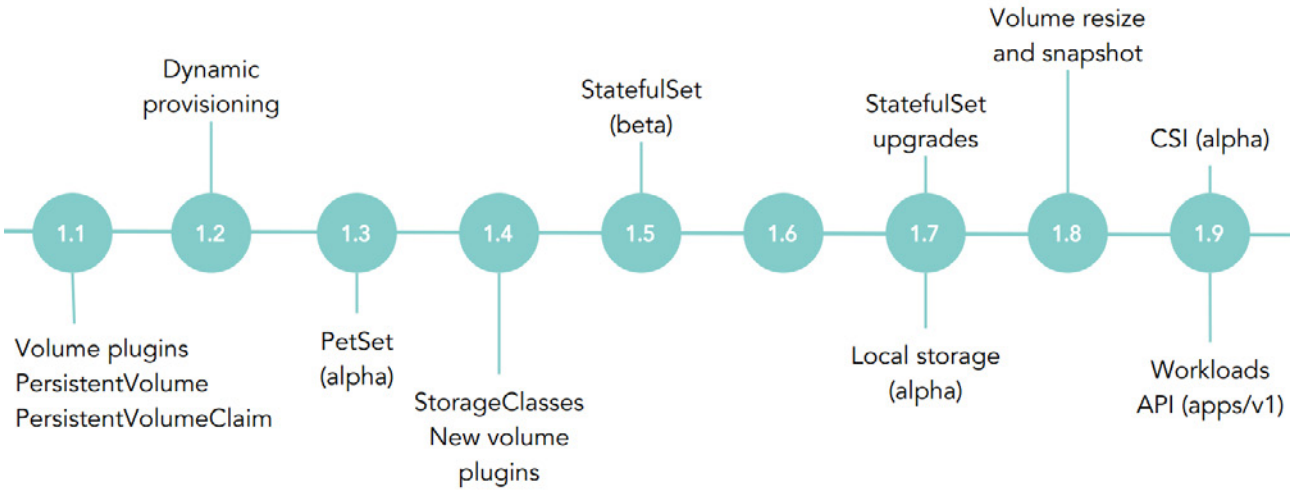
Moving database systems to Kubernetes is also problematic because it does not have the complex and application-specific operational awareness required to respond appropriately for all the various types of failure. So, during these events, human interaction is often still required to operate the database in question, and benefits of the time and efficiency savings anticipated with the automation can diminish.

InfoQ: In your talk, you highlighted how Kubernetes has been adding features over time that help manage stateful workloads. Could you expand on that?

Bates and Munnelly: Since the very early days of the project, there have been efforts to introduce and mature features to enable workloads with state. PersistentVolumes, dynamic volume provisioning, and StatefulSet provide **building blocks** that help run applications that require persistent disks or stable network identity, such as databases. These are brilliant tools, but on their own can be problematic to use and understand, and cannot do everything you need to automate the operations of the many flavors of distributed database systems.

InfoQ: Despite that evolution, you've opted to develop a Kubernetes extension called Navigator. Could you tell us what features Navigator provides and how do you see the tool fitting in the existing Kubernetes ecosystem?

Bates and Munnelly: We're very much building on this evolution. Resources such as StatefulSet and PersistentVolume, and their controllers, have brought about the building blocks for distributed stateful systems on Kubernetes. But by themselves, these primitives are not quite enough as they do not take account of the application-specific behavior for bootstrap, scale-up/down, backup and restore, and more. We are building extensions to Kubernetes in order to fill these missing gaps between platform functionality and user experience.



InfoQ: In your talk you mentioned the operator pattern. Could you summarize how this pattern can help or hinder the operation of Kubernetes stateful workloads?

Bates and Munnelly: The operator pattern was introduced by the folks at CoreOS, and they have led the way in adopting this pattern to orchestrate and manage the likes of etcd and Prometheus. In Navigator, we follow a similar pattern, but we also add a co-located binary (a "Pilot") that wraps each deployed database process. It's our eyes and ears to determine the database node's state, and this is reported back to the Pilot resource status in the Navigator API server (built on Kubernetes API machinery).

InfoQ: How does Helm-native Kubernetes application management fit in? What are its main shortcomings when it comes to applications with strong data-storage requirements?

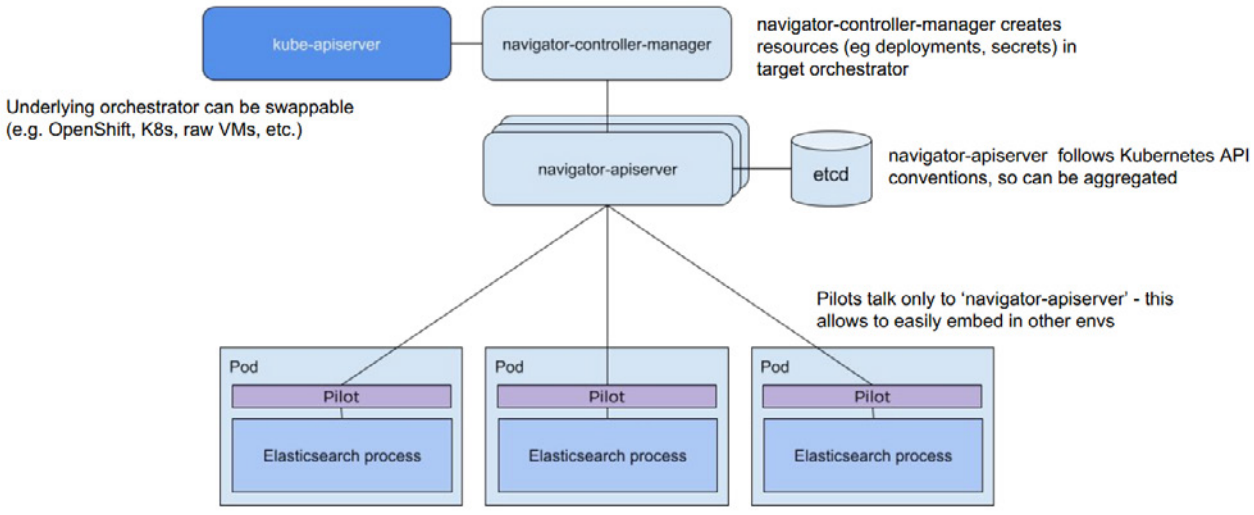
Bates and Munnelly: It's great to see such an extensive and ever-growing library of Helm charts. Most applications can now be easily deployed from a ready-made chart, and that includes stateful systems such as MySQL, MongoDB, and Elasticsearch, and the list goes on. However, many of these charts still require point-in-time management and lack the operational knowledge for proactive management and failure recovery. A chart will spin you up an Elasticsearch cluster, say, but it won't be able to handle scale-down gracefully.

InfoQ: How does the Navigator extension work, in a nutshell?

Bates and Munnelly: Navigator introduces new API types (such as ElasticsearchCluster and CassandraCluster) which represent higher-level constructs for users to interact with.

We have then created an operator that is responsible for manipulating and creating **other** Kubernetes resources in order to realize the desired state (e.g., a valid Cassandra deployment). This controller continually watches the deployment, and takes corrective action in response to failures, as well to drive operational tasks such as upgrade and scale-up/down.

In order to facilitate data collection from the databases being



deployed, Pilots, small applications that run alongside your database processes, collect information and store it back in the Navigator API in order to inform decisions made by the controller.

This separation of collection from action has been a key success for the project so far.

InfoQ: Navigator makes use of Kubernetes CustomResourceDefinition (CRD), correct? But it also extends the Kubernetes API. Why? Could you provide an example?

Bates and Munnelly: Navigator extends the API by including its own API server that can run alongside an existing Kubernetes control plane in order to provide the API extensions.

This is a new pattern to Kubernetes, and it's being used to break up the monolith and allow external collaborators to add native resource types to their cluster. A couple of examples of this would be Metrics Server and Service Catalog.

InfoQ: Do you think Navigator's API could become part of Kubernetes standard API?

Bates and Munnelly: This is very unlikely to be the case. The Kubernetes API and the powerful API machinery provide the foundations to build extensions such as Navigator. Building on Kubernetes is very much regarded as the preferred pattern for future application and environment-specific developments. The project maintainers' desire is to slow down development of the core and ensure stability and not overburden the existing API, which is already at hundreds of resources.

Looking into the future, we envisage a new generation of distributed systems built with Kubernetes primitives as a foundation: Kubernetes, container, and cloud-native from the get-go. The likes of CockroachDB offer us a glimpse into this future.

InfoQ: You've received a good amount of interest in the project since you gave the talk at QCon London. Any ideas on the roadmap and maturity level in the near future for the tool?

Bates and Munnelly: We were really pleased to present the project and our developments to date at QCon and, as you say, there has been a really positive reaction. We're fortunate to work closely with customers that are driving requirements, whilst maintaining the project in the open.

We are aiming to cut a v0.1 release in the coming weeks, which will represent the first supported API surface for Navigator. We will also include end-to-end tests for common disaster-recovery scenarios, and will be extending these further to ensure we can properly handle system failures. Looking further ahead, we're actively looking at supporting more database systems. Stay tuned!


Chris
SWAN
DXC Technology
aCon

Read online on InfoQ

Q&A with Jessica Kerr

Why DevOps is a Special Case of DevEx

by **Manuel Pais**

Jessica Kerr, lead engineer at Atomist, presented her perspective on DevEx (developer experience) and how it relates to DevOps, at this year's QCon London conference. Kerr stressed that embracing DevOps has led to teams accruing responsibilities (to be able to own and constantly improve their systems).

To reduce cognitive load on teams, we need better development tools that push down details on how systems get built and delivered, and also provide a seamless developer experience that is less dependent on how individual tools work.

InfoQ interviewed Kerr to find out more about her team's motivations, the tool (Atomist) that they've built to address these challenges, and what use cases and workflows it supports.

InfoQ: You have a particular definition of what a team consists of. Could you expand on that?

Jessica Kerr: A team is everyone it takes to be successful. As a developer, "success" means delivering useful software, so everyone who is required for that is part of the team (whatever the org chart says). That includes everyone who op-

erates the software; everyone involved in releasing it, designing it, testing it; and the people deciding what "useful" means.

InfoQ: Why do you say "DevOps is a special case of DevEx"?

Kerr: DevOps acknowledges that developers and operations people are on the same team. Then it says "let's improve the flow in this system", from packaging of artifacts, to rollout, and to production monitoring. With help from continuous-integration tools, we can push out changes more quickly and more safely — better for developers, better for users.

Building, testing, and releasing are part of the development experience. DevEx says there's more to our experience than that! How much of it can we automate?

As small teams accrue responsibility for more and more code, we need to scale the way we work. From code maintenance to coordination, what obstacles can we smooth out next? What tools will let us automate enough work to scale ourselves?

DevOps is a great way to start working on your development experience. Don't stop there.

InfoQ: What is your personal definition of a smooth developer experience? What are its main characteristics?

Kerr: Smoothness can be hard to see, because it's the absence of pain. Here are some aims:

- When there's an error, I can quickly see what caused it.
- I can find out where each piece of data came from.
- When it's time to make a decision, all the information I need is right there.
- As the system changes in ways that I care about, I learn what happened, without a lot of irrelevant news.
- When there's something important to remember, a computer remembers it for me.
- To change the system, I can express my intention concisely, and a computer implements it thoroughly.

InfoQ: What is Atomist? What gaps does it try to address and how does it work?

Kerr: Atomist is a tool for automating far more of your development experience than was practical before. Just as continuous-integration tools made it possible to run tests on every push to version control, Atomist makes it easy to respond to creation of an issue, to a request for a new repository, to test failures, to deployments, to pull request reviews — and, of course, to a code push. Our API for Software stores these events, relates them to each other, and triggers your automations to run.

Atomist offers a full software-delivery machine. Evaluate each commit and lay out its path to production: format, check policies, test, package, upload, run it, verify it, approve this in Slack, deploy to prod, and see where and when it's running. Atomist is not configured with YAML or JSON. Your automations are code: start with examples, change them, integrate your existing tools, add what you want. Your TypeScript (or JS) code runs where you choose, and registers with our API for triggering.

Published a new version of your internal library? Upgrade every project that uses it. Want a new project created to your specification and spun up in a test environment and then in production? That's five minutes, and it's part of our starting-point delivery machine.

When a person needs to decide, they can trigger automations from Slack or in a web UI. Have a useful script to clean up test data? Wrap it in a function and make it available to your whole team as a Slack command.

The Slack integration provides another unique feature of Atomist: when an automation reaches a decision point ("The tests failed! What should I do?"), it can contact a human. It can post in a channel or direct message: "Tests failed on your commit." It can provide the data: a link to the commit, the important piece of the log. It can provide some actions: a button to restart, a button to skip tests and build the artifact anyway. The automation becomes a collaborator, and the developer is in control.

InfoQ: Why did you decide to focus on Slack? What are some advantages and challenges of that decision?

Kerr: We want to bring all the pieces of the developer process together, to incorporate information that you'd otherwise have to collect from GitHub, Jenkins, JIRA, Kubernetes. We want to bring that information to you where you are, not add yet another place to go.

Atomist's built-in Slack notification replaces dozens of spammy messages: commits, build started and stopped, deploy, review request, etc. are all correlated and presented in one message that updates, along with useful buttons like "Merge" for pull requests or "Assign" and "Label" for issues. All of these correlations can also be queried by automations, so you can create similarly rich messages of your own.

Slack messages are targeted, clear, and interactive. They are useful on all devices — you can close an issue on your phone without logging in to GitHub. They let people trigger Atomist automations, and let Atomist ask people for decisions. The interactive messages are much harder to set up than a simple chatbot, so Atomist handles that for you.

Not everyone has Slack, of course. That's why Atomist also has a web workspace where you can see all these lovely correlated, actionable messages. Subscribe to the projects you're interested in.

InfoQ: How can development teams balance the abstraction level that Atomist provides with the need to understand and own their delivery toolchain in order to evolve it as delivery requirements change?

Kerr: Since all of the delivery automation in Atomist resides in code, you have full control. We provide libraries for cloning and committing to repositories, for registering with our API, and for understanding and modifying code. Our service triggers the automations, but they run in your process, and do whatever you code them to do. You're never dependent on the limited configuration options of our interface or some plugin.

One advantage of Atomist over tools like Jenkins is that you can automate every repository and team in your organization uniformly, with one code change — or limit it, if that's your choice. For instance, you can choose to run a linter using the rules file in each repository or global rules of your choosing. The creator of an automation has a lot of power.

InfoQ: What kind of workflows does Atomist facilitate? Are those purely on the development side or are there operational procedures that can benefit from Atomist as well?

Kerr: Oh, thanks for asking! As a developer I get very focused on my own experience, but mine is not the one that matters most.

Take a product owner or business person who has opened a ticket. You could create an automation to notify that person when a fix is deployed to the test environment, and give them link to where they can try it, and give them buttons to approve it or add more comments. The Slack interface is particularly good for this, since business people are also there.

For operations, we can choose to make Slack into a new command line, by wrapping scripts in automations and registering them with Atomist. This has a few advantages over running scripts locally: the automation developer has full control over who can run which commands; updating the automation updates everyone's use of it; and, in public channels, everyone can see what's going on. When there's trouble in production and people are investigating, it's great to see what others have already tried and seen.

InfoQ: What about facilitating business decisions and visibility into the status of the software systems and ongoing changes? Are there particular challenges bridging the gap between technical and business information?

Kerr: People in the larger organization do need visibility into what's going on in our teams. This is where tools like JIRA become useful: for tracking in a way that rolls up. While it's important to keep people informed, it's no fun to fill out required fields and update ticket status. If we can automate that, through automatic updates based on commits when possible and friendly Slack commands otherwise, it takes some irritating bumps out of my day.

In a sense, each team needs to implement an API to the wider organization. We can implement that manually or automate some of it.

My goal is to gather information with zero to little cost to the busy person who has it, and provide it in the most convenient form to the person who needs it. This isn't easy, so implement a little at a time, and iterate. We'll never be able to automate every case, because it's development: everything we do is new, every change is different. As a fallback, ask a person!

InfoQ: What other recommendations do you have in terms of reducing cognitive load for development and other teams?

Kerr: Offload everything you can from your memory into the computer, in the simplest way that works for you. Spreadsheets are great. Shell scripts are great; anything you can turn from five steps to one, you'll do that thing more often. Clipboard managers are great. Slack's "/remind" feature is great; your human brain is not an alarm clock.

Ben Hammersley called smartphones our "robot brain". It makes sense now why I don't like being separated from mine. I don't feel bad about depending on my phone anymore.

If it's important, automate it. Otherwise it's too much pressure on my human brain.

Read online on InfoQ

Ian Crosby Shares His Thoughts

Has Kubernetes Crossed the Chasm?

by **Manuel Pais**

Ian Crosby, senior engineer at Container Solutions, tried to answer the question of whether Kubernetes has crossed the chasm from early adopters to early majority.

Based on real-world examples from several organizations he has been working with, Crosby claimed that Kubernetes is indeed close to mainstream adoption as the remaining challenges in the enterprise world (namely highly secured environments, support for Windows, better support for stateful workloads, and integration with legacy software and hybrid clouds) are actively being addressed by the community. As Crosby put it, "the question is not if Kubernetes will cross the chasm, but when."

Ideal use cases like fashionTrade, a 100% cloud-based and microservices-driven system — where Kubernetes provided several benefits in terms of reducing operational workload (by natively providing fault tolerance, auto-scaling, and service discovery) at a low adoption cost — are an exception rather than the norm, Crosby realized over time.

In other use cases, a system might initially appear like a good overall fit for Kubernetes, but a particular issue arises with the current maturity of the platform that can put adoption at risk. Crosby referred to Student.com, a fully cloud-

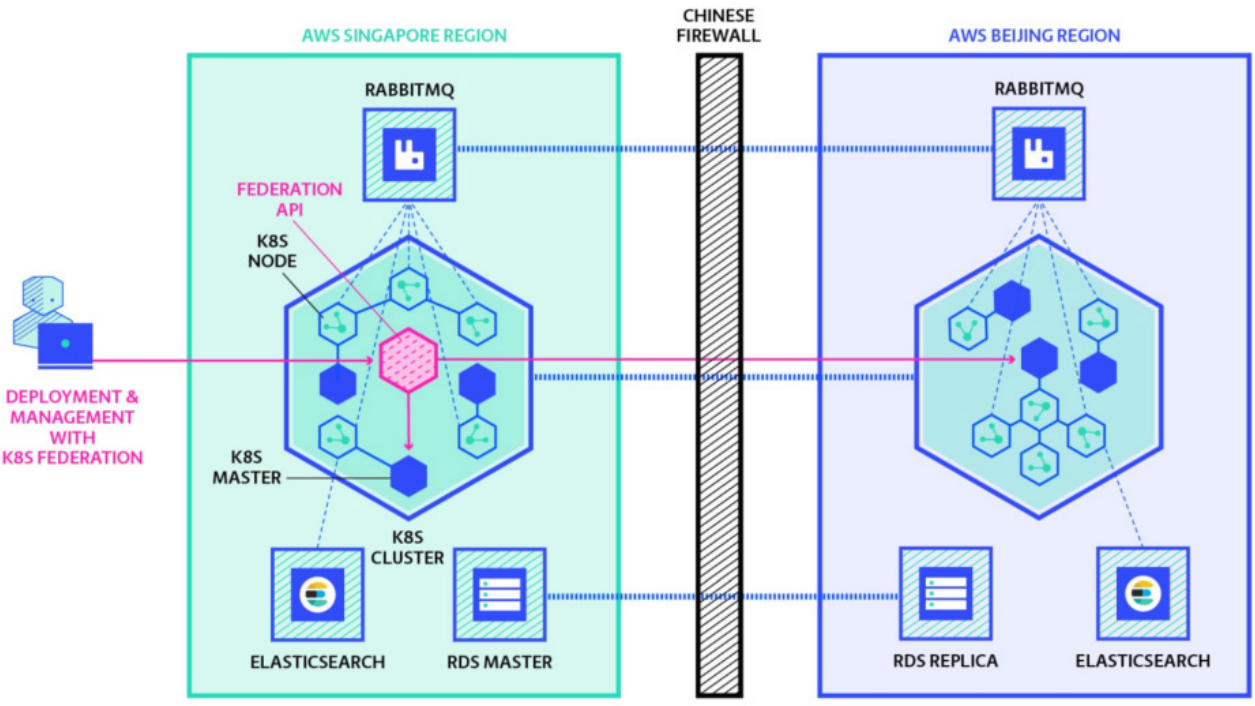
based system that was running in two different AWS regions, Singapore and Beijing, in order to better support their target markets. Initially, this appeared to be just a matter of running one cluster in each region and federating the two clusters. However, it turns out that the Great Firewall of China blocks all incoming traffic from Google, so the installation of Kubernetes within the AWS Beijing region itself became a challenge, as some of the components could not even be downloaded.

Fixing this installation problem required setting up mirrors to download the components, and customizing the Kubernetes deployment in the Beijing region via Terraform and Ansible scripts. Meanwhile, the community surrounding one of the most popular installation tools for Kubernetes — kops — added a readme on how to install in AWS China, highlighting the proactive evolution that is often critical for open-source software adoption in the enterprise.

The third, and most challenging, real-world example was Ericsson's move to a Kubernetes-native model for delivering one of their software applications that is run by clients themselves. With the help of Helm, an open-source package-management solution for installing and updating Kubernetes-native apps with a simple command line, Ericsson was able to package their software and get clients to update painlessly with a single command. Kubernetes-native rolling deployments also meant zero downtime for clients and transparent upgrades. This drastically reduced the number of clients running (and requiring support for) old versions of this software, while also giving them the chance to benefit from Kubernetes's operational features.

According to Crosby, this is a new model of delivering enterprise software altogether, and provides client ownership and fast upgrades, which is living proof of Kubernetes's landscape-altering potential.

In terms of Kubernetes adoption from scratch, Crosby recommends starting by using containers in the CI/CD pipeline then experimenting with Kubernetes clusters as the underlying infrastructure in development and test environments, and, once familiarized, rolling out Kubernetes to one non-critical product in production. Once everyone is on board and specific challenges have been addressed, rolling out to other, more critical systems in production becomes a natural step.



Deploying and running Kubernetes across multiple Asian cloud regions (credit: Ian Crosby)

QCON.AI 9-11 APR

QCon.ai
by InfoQ

The Inaugural QCon.ai

At the beginning of April, we ran our inaugural QCon.ai event conference at Parc 55 in San Francisco, with a focus on AI, machine learning, and data engineering for software engineers.

Our motivation for launching this event is that we believe that many of the techniques coming out of the various data-science disciples are fast becoming essential tools for software developers, and we wanted an event that would allow us to go much deeper into the topic than is possible at a typical QCon to help equip our attendees for this change.

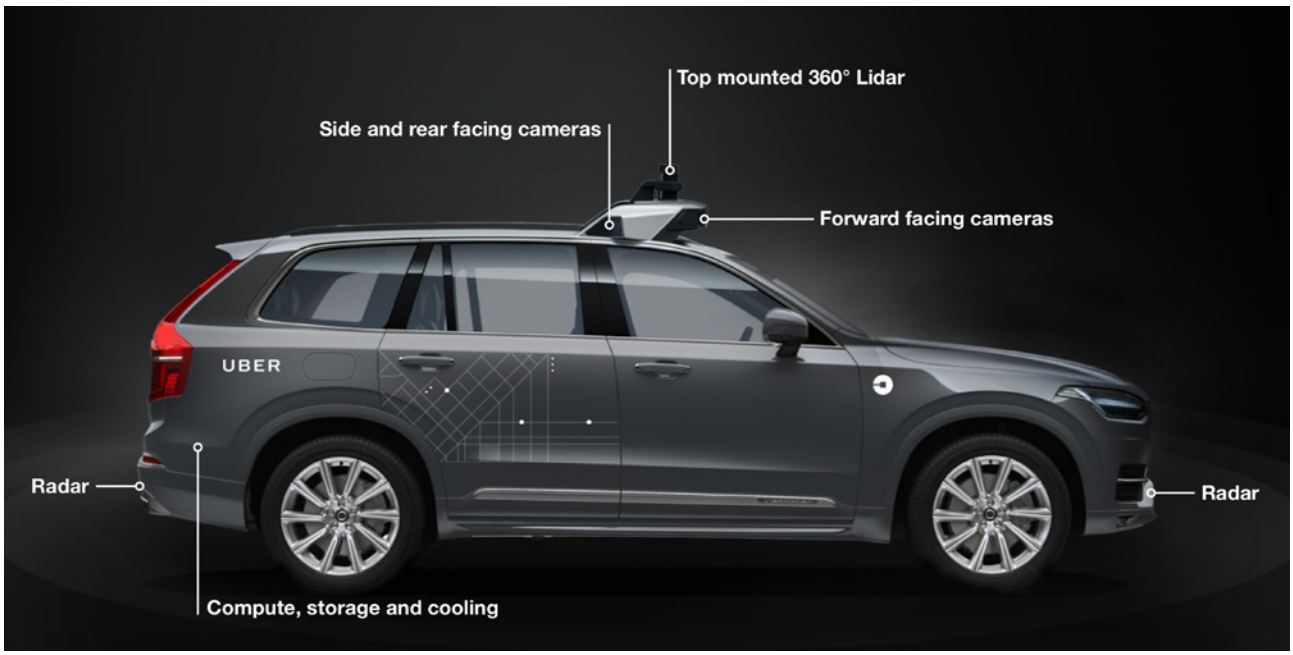
QCon.ai was a three-day show with four in-depth workshops covering TensorFlow, Apache Kafka, Apache Spark, and Python-based AI workflows, two days of three concurrent tracks on topics such as predictive data pipelines, AI in the physical world, deep-learning applications and practices, and real-world data engineering. We also had code labs on topics such as TensorBoard, R, and more.

One major theme for the show was autonomous vehicles.

"I want to start with a pretty staggering statistic, and that is that 1.3 million people die in car crashes every year. That, on average, is over 3,000 deaths a day," said Matt Ranney, senior staff engineer at Uber, as the conference opened. He went on:

[That statistic] doesn't even include the 20 million people every year that will be injured or disabled from cars. Cars are pretty lethal. If you live in the United States and you are a younger person, it is the most likely way that you will die….

And the reason for all of this is the drivers. It is the human beings…



The biggest thing is recognition error, which means you're not paying attention or you just couldn't pay attention, you can't see everywhere at once. It's just there's a lot going on out there on the road. Decision error number two, that means like you took a bad decision, like you decided to break the law. You're driving too fast or you just decided to do something that was unwise. And these are not the kind of decisions that software is going to make; software can fix these problems.

Uber is working on both self-driving trucks and self-driving cars as part of the firm's broader business model to improve urban transportation, Ranney stated. The hardware on the car is standard for self-driving vehicles at the moment.

On top of the car is a 360-degree LIDAR. "This spins around, paints the world with 64 lasers, and this gives us a 3-D image of the surroundings…. Under the bumper, cleverly concealed, are a bunch of radars, and in the back is a bunch of compute storage and cooling.

The vehicles also have a large number of cameras. All these sensors are hooked up to a compute blade, which is a standard X86 computer running Linux. This computer has local storage, which is used to hold maps, models and logs.

Ranney explained that:

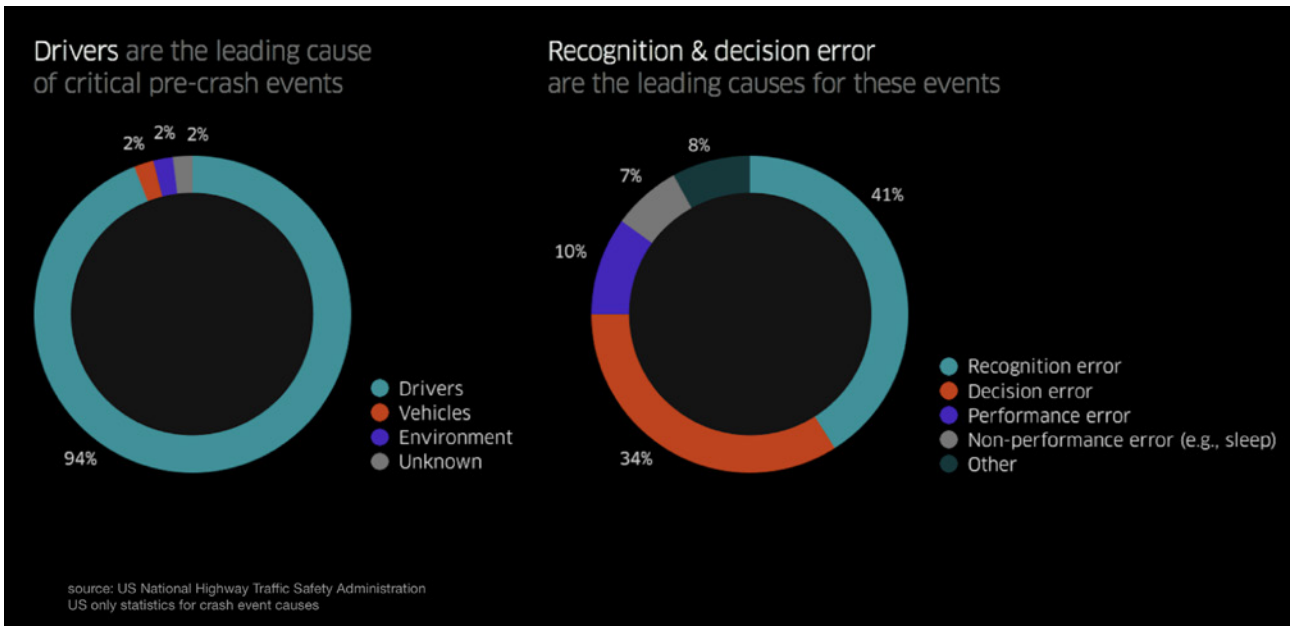
As we take sensor data and make decisions about it, that all gets logged. And running on this compute are a bunch of tasks. These tasks are Linux processes. They might have multiple threads, but there are different tasks that sort of drive the autonomy system. And the tasks talk to each other with an event bus. And, so, this event bus is not entirely unlike, say, Kafka. But in this case, it's optimized for exchanging events very efficiently for processes running on the same machine. So, the message format that they exchange is not unlike protocol buffers, but it's a different thing that we use for various legacy reasons.

There are multiple compute blades within the car that run different tasks, and the different blades can communicate via a network.

Ranney talked about three main problems that need to be solved for a self-driving vehicle to work:

1. **Perception** — This is how the software figures out what all the objects around the vehicle are. It takes data from multiple sensors and fuses these together to produce "a single output stream of objects, their type, a classification for those objects as well as their simple trajectory sort of over time." This is essentially building a set of learned models, and requires significant training data.

2. **Prediction** — This is how the software figures out how these objects are going to behave. "As a human driver, you mostly unconsciously make judgments about where you think the different actors in a scene are going to go and you make your own plan accordingly. And that's exactly what this system does as well."



3. **Planning** — This is working out where the vehicle wants to go, where the system enforces traffic laws, speed limits, turn restrictions, and so on.

To test the software, Uber uses multiple approaches. The firm has built a test track in Pittsburgh where it can set up a realistic but controlled environment with traffic lights, buses, emergency vehicles, and so on. They also have a simulator that allows them to do extensive offline testing, and, of course, they do road tests as well.

Ranney concluded with a plea:

We need people to build the data pipelines and to do everything from making websites so that people can look at the results down to the actual training of models and evaluating their performance; it is truly a full-stack problem in computing. And it's one that can really save people's lives. So, I hope if you're not working in this space, you will consider doing so.

Elsewhere, Jendrik Joerdening and Anthony Navarro talked about how a team of Udacity students used neural networks to teach in two days a car to drive by itself around a track, and we had a session on building computer-vision systems for self-flying drones from Alexander Harmsen, CEO and co-founder of Iris Automation.

The conference closed with a keynote from Rachel Thomas, co-founder of fast.ai, who spoke about "Analyzing and Preventing Unconscious Bias in Machine Learning":

Algorithms are increasingly being used to make life-impacting decisions or on hiring and firing, and in the criminal justice system….

Pro Publica did an investigation in 2016 on a recidivism algorithm that is used in making pretrial decisions. So, who is required to pay bail and who isn't. A lot of people can't afford bail, and so this really impacts their lives. It's also used in sentencing and determining parole.

And Pro Publica found that the false-positive rate for black defendants, people labeled "high-risk" who did not re-offend, was nearly twice as high as for white defendants. So, there was an error rate of 45% for black defendants and 24% for white defendants.

Race was not an explicit input variable to this algorithm, but race is "latently encoded in a lot of other variables of where we live and our social networks and education, and all sorts of things." In other words, blindness doesn't work.

Another example Thomas spoke about was computer vision, which often fails people of color:

In 2016, Beauty.AI were doing the first AI-judged beauty competition. It found that people with light skin were j udged much more attractive than people with dark skin. In 2017, FaceApp, which uses neural networks to create filters for photographs, created a "hotness filter" that lightened people's skin and gave them more European features.

Joy Buolamwini and Timnit Gebru evaluated a lot of commercial computer-vision classifiers from Microsoft, IBM, and Face++ (a giant Chinese company) and found that "the classifiers worked better on men than on women, and better on people with light skin than

people with dark skin," Thomas said.

The third example Thomas looked at was word embedding used by products like Google Translate.

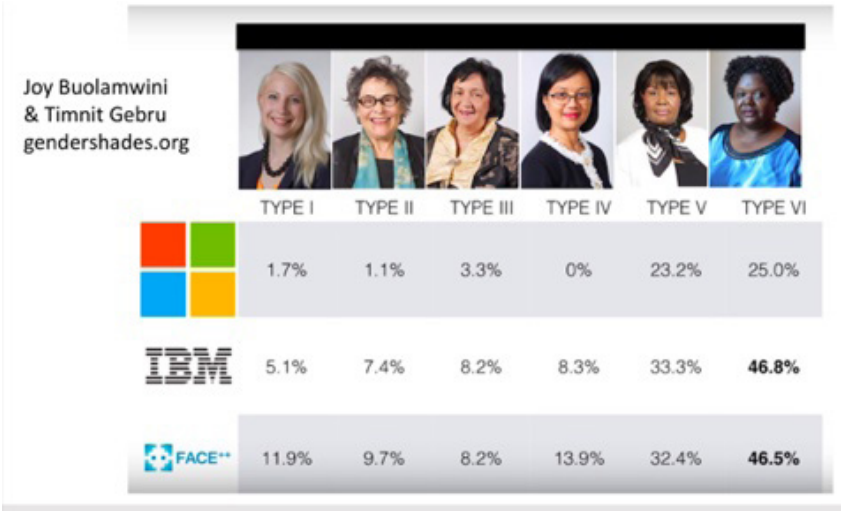
If you take the pair of sentences, "She is a doctor. He is a nurse," translate them to Turkish, and then translate them back to English, the genders have been flipped to fit the stereotype and it's now saying, "He is a doctor. She is a nurse." Turkish has a gender-neutral singular pronoun and you can see this in other languages with gender-neutral singular pronouns. People have documented this about a variety of words that women are lazy, women are unhappy, a lot of stereotypes.

Thomas made the point that machine learning not only reflects bias in society but it can amplify bias.

In the final section of her talk, Thomas offered practical ways to address some of the problems. For word embeddings, she notes that there are two different kind of schools of thought about this and one is to try to de-bias the word embeddings:

There's an academic paper by Bolukbasi that gives a technique for doing so. Rob Spear has released a de-biased set of word embeddings called ConceptNet.

Then the Caliskan-Islam paper, which is another academic paper, says that the de-biasing should happen at the point of decision or the point you're taking an action, that humans are able to perceive the world with bias and so computers should be able to as well, and that it's the point of action where you want to make sure there's no bias.



And they warn, and I think this is true that even if you remove bias earlier in your model, there's so many places that bias can seep in that you need to continue to be on the lookout for it, regardless of which route you take, but I wanted you to know these are two different schools of thought that researchers have about it.

For computer vision, more representative data sets are one possible solution. Work from Buolamwini and Gebru explores this in more detail, and includes a short video that showcases their work.

Thomas wrapped up her presentation with a list of questions to ask about AI:

1. What bias is in the data?
2. Can the code and data be audited?
3. What is the accuracy of a simple rule-based alternative?
4. What processes are in place to handle appeals or mistakes?
5. How diverse is the team that built it?

She also provided a number of recommended resources, including several videos from the Conference on Fairness, Accountability, and Transparency:

- Arvind Narayanan's "21 Fairness Definitions and Their Politics" talk;
- Latanya Sweeney's keynote, "Saving Humanity"; and
- a tutorial, "Understanding the Context and Consequences of Pre-trial Detention".

Thomas also cited *Weapons of Math Destruction* by Cathy O'Neil (Review, Podcast).

You can find the long talks from QCon.ai on InfoQ, and short talks on our YouTube channel. QCon.ai will return to San Francisco April 15-17, 2019.

NEW YORK 25-29 JUN

QCon

Introduction

by **Charles Humble**

Conferences often have a theme that emerges during their course that none of us had predicted. One year at QCon London it was, bizarrely, cat pictures; every presentation you went to had a cat picture in it.

Another year, as the microservices movement was just getting going, it seemed that we had mandated a Conway's law slide in every presentation — we hadn't, of course, but there certainly were plenty of them.

This year, at the seventh annual QCon New York, our second year in Time Square, it felt like the emerging theme was diversity and inclusion. The event had a particularly positive atmosphere that made for something truly special, and we got a huge amount of positive feedback from attendees and speakers about it, both during the event and afterwards. This is something the QCon team has worked on for several years, and it felt wonderful to see that work starting to pay dividends.

From a content perspective, attendees at the event got to see keynotes from Guy Podjarny, co-founder of Snyk, talking about "Developers as a Malware Distribution Vehicle"; Joshua Bloch giving a "Brief, Opinionated History of the API"; and Tanya Reilly, principal engineer of Squarespace, giving a thoroughly interesting and unusual talk about the history of fire escapes in New York City and what we as software engineers can learn from them.


QCon
QCon
QCon
QCon
QCon
Carpe Data
neo4j

Read online on InfoQ

Michael Bryzek's Lessons Learned at QCon NY

Designing Microservice Architectures the Right Way

by **Daniel Bryant**

At QCon New York 2018, Michael Bryzek discussed how to design microservice architectures "the right way".

Key takeaways included: engineers should design schema first for all APIs and events, as this allows the automated code generation of boilerplate code; event streams should be subscribed to in preference to directly calling APIs; invest in automation, such as deployment and dependency management; and the engineering teams must focus on writing simple and effective tests, as this drives quality, streamlines maintenance, and enables continuous delivery.

Bryzek, co-founder, chairman, and CTO at Flow (and previously co-founder of Gilt), began the talk by sharing a personal anecdote about how a request to change a URL within a system from the pattern of foo.com/latest/bar.js to foo.com/1.5.3/bar.js received the response that it would "take weeks to complete" and that the team did not have the resources to do this.

Puzzled as to why a simple URL format change would require this amount of work, he found out that the URL was implemented within a library, and this would require hundreds of dependent services to be updated, some of which had not been touched in years, and would require addi-

tional dependency updates in order to build and redeploy.

The anecdote was a clear example of a "not-so-great architecture", where near-term velocity had been traded against future paralysis. In contrast, a great architecture should actively help to scale development teams, delivering quality and enabling high performance and low cost while supporting future features naturally. The microservice architecture is a popular modern style of implementing systems, but it needs to be designed the correct way; engineers should strive not to design "spaghetti" systems and instead aim for a layered approach.

Bryzek presented a series of common misconceptions with microservices. The first misconception was "microservices enable teams to choose the best programming languages and frameworks for their tasks." The reality is that this can be expensive, and team size and investment are critical inputs into the decision to support another language within a technology stack.

If we look at Google as a generally great engineering company, they have about 20,000-30,000 engineers and, at last count, eight programming languages. So, I like to say [adopt] one programming language for every 4,000 engineers [within your organization].

The second misconception Bryzek discussed was "code generation is evil." However, the reality is that "creating a defined schema that is 100% trusted" is important for resources and events, from which code generation can be useful for scaling development efforts and maintenance. Bryzek compared the third misconception, "the event log must be the source of truth," against the reality that events are a critical part of an interface, but it is "okay for services to be the system of record for their resources". The final misconception, "developers can maintain no more than three services each," Bryzek countered with a discussion that this is in reality the wrong metric. When done correctly, he stated, this is where automation shines; developers at Flow maintain approximately five services each, and weekly maintenance takes less than 5% of their time.

Bryzek presented a tour of the Flow architecture. This included a look at the resource-oriented API definition schema, written in JSON, which allows everything from entities and corresponding properties to be defined, alongside appropriate metadata. The schema definition files are stored within Git DVCS, and the teams practice continuous integration. A series of tests enforces consistency across the entire set of APIs and effectively acts as an advanced linter. The goal with these tests is that it should result in engineers believing that "one person wrote the entire API".

The engineering team use the open-source API Builder tooling in combination with their tests to help prevent errors and verify potentially breaking changes during the API design phase. The API Builder CLI utilities allow the generation and update of code associated with resource APIs and routes (primarily written with the Play 2 framework at Flow). An API client›s code can also be auto-generated, which includes the generation of mock clients that enable high fidelity and fast testing. Bryzek stated that within the Flow architecture, the system of record is the API specification; code generation ensures that the team "actually adheres to the spec".

Moving on to discuss database architecture, Bryzek said that each microservice application owns its database. No other service is allowed to connect to this database directly; other services should use only the service interface, which consists of the provided API and an event stream.

The Flow engineering team has invested heavily in creating a single CLI development tool that is used for managing all infrastructure and common development tasks. A single command creates a database (using AWS RDS in the example shown). All storage requirements are defined using a JSON schema, which, although independent from the associated API schema, uses the same toolchain. Database DDL operations and configuration, such as table creation, are auto-generated as is the associated application-service client data-access-object (DAO) code. This normalizes access to the database and ensures that proper indexes exist from the start.

The next section of the presentation focused on deploying code, the trigger for which is the creation of an associated Git tag. These tags are created automatically by a change on the master (e.g., a pull-request merge) and are "100% automated and 100% reliable". Bryzek demonstrated Flow's continuous-delivery management system, Delta (the code is available on GitHub), and pointed out the importance of continuous delivery as he called it "a prerequisite to managing microservice architectures".

The definition of infrastructure required for each microservice is kept simple, and a single YAML file defines metadata such as the

compute instance type, ports open, and version of the OS configuration. All services within the system expose a standard health-check endpoint, which enables all of the deployment, observability, and alerting tooling to determine health.

Bryzek discussed the importance of events and event streaming. Many third parties ask for access to the Flow API, to which he responds, "we have an amazing API, but please subscribe to our event streams instead." The key principles for an event interface include: the provision of a first-class schema for all events, producers guarantee "at least once" delivery, and consumers implement idempotency. Within the Flow architecture, the end-to-end single-event latency is about 500 ms, and the implementation is based on PostgreSQL, which is scaled to about a billion events per day per service.

The approach to events means that producers create a journal of all operations on their corresponding database table, recording inserts, updates, deletes, etc. Upon creation of an event, a corresponding journal record is queued to be published. This happens in real time, is asynchronous, and a single event per journal record is published. Replay of events from a service is achieved simply by re-queueing its journal records. Consumers store new events in a local database that is partitioned for fast removal. Upon an event's arrival, the record is queued to be consumed. Events are processed in micro-batches, with a default period of 250 ms between each consumption attempt. Any failures are recorded locally. More information and code examples on this approach can be found in the Gilt Technology db-journaling repository. Interested readers can also explore the similar concept of change data capture (CDC), and open-source technology in this space includes Debezium.

Bryzek closed with a focus on dependency management and "keeping things up to date". The goal should be to regularly and automatically update all services to use the latest versions of dependencies; this is critical for security patches and bug fixes in core libraries. Referring back to his opening anecdote, Bryzek stated that "this should take hours, not weeks or months" and that the same process should be used for internally developed libraries as for external open source. The Flow engineering team upgrades all services every week to the latest dependencies, and the use their dependency tooling (released as open source) to track dependencies. The tooling initiates automated upgrading within a service's code base, along with the creation of an associated pull request. Each service is deployed as soon as the build passes.

Bryzek summarized that engineers should design schema first for all APIs and events, and events should be consumed in preference to directly calling APIs. Engineers should invest in appropriate and effective automation, focusing on tasks such as deployment, code generation, and dependency management. Teams should also be encouraged and enabled to write "amazing and simple" tests, as this drives quality, streamlines maintenance, and enables continuous delivery.


qcon
qcon
qcon
qcon

Read online on InfoQ

Susheel Aroskar at QCon NY

Scaling Push Messaging for Millions of Devices at Netflix

by **Srini Penchikala**

Susheel Aroskar from Netflix's engineering team spoke at QCON New York about Zuul Push, a scalable push notification service that asynchronously pushes data like personalized movie recommendations from cloud to devices.

The Netflix team uses Zuul Push to improve the user experience for their customers. A typical user on Netflix spends a significant amount of time picking a movie. This framework helps by providing better engagement, generating customized recommendations in the cloud that can be shown to the user. Also, if there is a better recommendation generated, it lets the client know in real time.

Zuul Push is a high-performance asynchronous service based on Netty, a non-blocking I/O (NIO) network application framework. Zuul Push can be used to scale to large number of persistent connections, and supports WebSocket and server-sent events (SSE) protocols for push notifications. The technology handles more than 5.5 million connected clients at peak.

Aroskar discussed the architecture of Zuul Push, which includes the following components:

- Zuul Push servers,
- a push registry,
- a push message queue,
- a push library, and
- a message processor.

The replicated client registry makes it possible to scale the concurrent persistent connections and deliver push notifications. The push registry, which is based on Redis and Cassandra, provides the features needed for this type of workload, which includes low read latency, record expiry, sharding, and replication. Data is stored using Dynomite, a generic Dynamo implementation for different key-value stores, which provides access to a Redis key-value backing store with additional capabilities like auto-sharding, read/write quorum, and cross-region replication.

Aroskar reviewed the design of the back-end message-routing infrastructure, which uses a Kafka server to decouple message senders from receivers. Message processing is responsible for queuing, routing, and delivering the messages. The solution uses the "fire and forget" approach to message delivery, and there are different queues for different priorities of message. Netflix engineering also runs multiple message processor instances in parallel to scale the message processing throughput.

Zuul Push supports custom authentication policies so you can authenticate using cookies, JSON Web Tokens, or some custom scheme. Aroskar discussed the operational aspects of Zuul Push. Netflix uses Amazon's Application Load Balancer to manage the push cluster and maintain long-lived, stable connections.

The Zuul Push framework is based on Zuul, which is Netflix's API gateway. All of the Netflix HTTP API traffic passes through Zuul. Readers keen to learn more about Zuul Push can consult the project's website and the sample Java application that shows how to use WebSockets as well as SSE to enable push messaging for Zuul.

Read online on InfoQ

Matt Klein on

Lyft Embracing Service Mesh Architecture

by **Srini Penchikala**

Matt Klein from the Lyft engineering team spoke at the QCon New York 2018 about the Envoy service mesh architecture. Facing the operational difficulties with their initial microservice deployment, Lyft migrated to using a service mesh.

Klein discussed the evolution of their architecture, which five years ago was based on Amazon Web Services Elastic Load Balancer, a PHP/Apache monolith, and MongoDB as the back-end database. When they started using microservices three years ago, they still had monolithic applications to maintain in multiple languages (PHP, Python, front-end code in Node.js, and Go services) and frameworks. They also had many protocols to connect to MongoDB, Redis, and caching servers. Lack of consistent observability in regards to metrics/stats, tracing, and logging was another challenge. Capabilities like service retry, circuit breaking, rate limiting, and timeout were not completely implemented.

Klein said that typically all problems come down to networking and observability, and so it's very important to have observability built into the solutions. At the same time, application development teams shouldn't be developing these capabilities from scratch in every project. Whenever the developers are not writ-

ing business logic or application code, they are wasting time.

Lyft's current application architecture is based on every service communicating with other services through Envoy. The idea behind a service-mesh architecture is that the network is fully abstracted from the services. It is also abstracted from the developers; a sidecar proxy is co-located with each service running on localhost. A service will talk to its sidecar, which in turn talks to the sidecar of another service (instead of directly calling the second service) in order to perform service discovery, fault tolerance, and implement tracing.

Envoy is an out-of-process architecture that supports:

- L3/L4 filter architecture like a TCP/IP proxy,
- HTTP/2-based L7 filter architecture,
- Service discovery and active/passive health checking,
- load balancing,
- authentication and authorization, and
- observability.

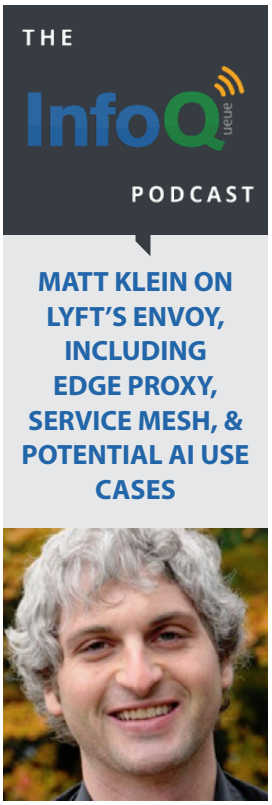
Envoy can used as a middle proxy, a service proxy, and as an edge proxy. An edge proxy is deployed on the internet gateway (point of external ingress) and takes care of concerns like service discovery and load balancing.

Klein discussed the developer experience after they started using Envoy. They provide per-service auto-generated consistent dashboards for all services to help with troubleshooting. Dashboards include clickable links to detailed data. Developers can click on a dashboard and navigate to a trace UI to see which parts of the application are generating longer response times. The Lyft team currently has 100% trace coverage without gaps between services. Each incoming client request gets a unique identifier that is used for correlation of logs. The dashboard also provides a service-to-service communication overview, with a drop-down interface for every service; developers can select the caller and called services from the template to see where errors are occurring. The global health dashboard that supports 20,000 hosts at Lyft is often the first stop to view the health of the system.

Klein noted that the Envoy-based configuration management solution is based on the xDS APIs for discovery services, including Listener Discovery Service (LDS) and Cluster Discovery Service (CDS) APIs. They support a split architecture where they use legacy service for service discovery and the new solution for the new services. Lyft's current Envoy deployment includes hundreds of microservices, 10,000 hosts, and five to ten million mesh requests per second. All edge traffic, all service-to-service traffic, and the vast majority of external partners are part of this deployment.

Envoy is a community project that was released as open source in September 2016. Klein suggested developers should use it because of its support for quality, velocity, extensibility, and an eventually consistent configuration API.

More about Envoy



Read online on InfoQ

Joe Emison on

Serverless Patterns and Anti-Patterns

by **Srini Penchikala**

Joe Emison, CTO at Branch, spoke about the design patterns and anti-patterns in serverless architecture.

Emison stated that for software we develop, especially software based on the software-as-a-service (SaaS) model, maintenance is critical because the software will last a long time.

Software applications with less custom code are easier to maintain. It is important for developers to know that the software is all about the value it delivers, not what's inside it. Serverless provides the pragmatic approach to architecture, which allows developers to focus on business logic and not infrastructure.

Emison talked about how serverless-architecture applications work. Serverless doesn't mean no servers, it means no server ops. Technologies like containers helped to abstract away infrastructure and server-level operations. Serverless takes away even the application-level operations.

The talk presented four real-world use cases: Fuel SQC, Spaceful, Commercial Search, and Branch.

The architecture of these applications included Angular and React on the web UI side, AuthO for user authentication, Cloudinary for image management, Netlify, Amazon CloudFront, Amazon S3, Lambda@Edge, Amazon Cognito for security, and Amazon DynamoDB as the database.

Emison discussed design patterns for serverless applications, anti-patterns for greenfield applications and migrations, and real-world consequences of running serverless apps in production.

The serverless patterns and anti-patterns he discussed included thick clients, functions, a service-centric approach, and custom research and code.

Emison advised having thick clients in serverless-based apps, but not thick middle tiers. Serverless doesn't work well on the client side (using frameworks like Ruby on Rails, Django, or Angular). Developers who are used to taking these frameworks and just moving them over to the serverless model are going to face challenges.

Functions, he said, are the glue in this architecture model, but avoid functions that call other functions. You'll have to pay for every function execution, which could become costly. Deploy each function monolithically to address this concern.

Develop your applications based on a service-centric approach and build on the back-end middle-tier services as opposed to writing and running them yourself. But beware of multiple single points of failure. Make sure the service you are using has the right uptime.

Custom research is a good practice in developing applications to leverage serverless architecture. But custom code can be an anti-pattern. It's OK to take some time to do the research and pick the right solution — Emison's example of this is to "prefer two weeks of planning and two days of development over two days of planning and two weeks of development".

More on Serverless

THE SERVERLESS SEA CHANGE

by Joe Emison

Golestan Radwan

Read online on InfoQ

Golestan Radwan at QCon NY

A Team's Transformation from Software Development to ML

by **Srini Penchikala**

As companies start to add big-data and machine-learning (ML) initiatives to their project portfolios, they face several challenges, including their teams' transition from software engineering to data engineering and ML. Golestan "Sally" Radwan spoke about her experience in leading a traditional software engineering team on a ML/AI journey.

(InfoQ does not store a video of this session at her request.)

Radwan talked about what went well in the transition from software architect to data architect. Techniques like frequent and visual communication helped, but they had to make some tough technology and people decisions. She discussed transformation at the level of individuals, teams, and the business.

Individual transformation included leaving behind languages like PHP and Python; changing the technology was not easy. The developers had to get comfortable with handling completely differ-

ent data structures, formats, and data sources.

They also had to learn to work much more closely with DevOps. She advised the developers to learn fast and be ready to compromise. It's important to lean on your team. Architects should learn about data and ML pipelines. People need to focus on the performance of each algorithm and its pros/cons and suitability for different types of data.

Some of the disciplines and resources they relied on for this transformation included:

- mathematics (linear algebra and multivariate calculus);
- probability and statistics;
- MOOCs;
- Books; and
- online resources like Kaggle competitions, Google Colaboratory, Machine Learning on AWS, and Azure Machine Learning Service.

At the team level, they focused on regular knowledge-sharing sessions: they held "Lunch and Learn" meetings every Thursday, in which team members could share what they learned. They also codified this knowledge and automated associated processes using CircleCI in order to make the most use of data scientists' valuable time.

Speaking of data science, Radwan suggested looking for some development skills (at least to prototype) when hiring data scientists. Probe their real-world understanding to learn how much they care about the big picture and company goals and how they fit in the organization.

At the organization level, if you have a podium, speak up and set expectations for your stakeholders. Educate and collaborate with other teams. It's important to be clear on targets, goals, and requirements. In the ML space, your responsibility includes privacy, provenance, bias, and quality. Take time to understand the context and implications of what you're doing.

Radwan mentioned the Rachel Thomas's presentation on analyzing the bias in machine learning at QCon.ai 2018. She concluded the discussion by suggesting that teams resist the urge to overcomplicate things so that they can be successful in their ML initiatives.

Read online on InfoQ

Randy Shoup Discusses High Performing Teams

Breaking Codes, Designing Jets and Building Teams

by **Daniel Bryant**

At QCon New York, Randy Shoup, VP engineering at WeWork, presented "Breaking Codes, Designing Jets, and Building Teams". He began by quoting Mark Twain — "History doesn't repeat itself, but it often rhymes" — and stated that he believes the most effective teams throughout history have focused on purpose, organizational culture, people, and engineering excellence.

The remainder of talk provided a compelling overview of three effective teams, examining how each approached these four focal points, and providing some subtle (and not so subtle) analogies to modern software development.

The first story of an effective team covered the World War Two code-breaking efforts carried out at Bletchley Park in the UK, where the world's first programmable electronic digital computer was built entirely in secret in 1943. It is estimated that the intelligence produced by Bletchley Park, codenamed "Ultra", ended the war two years early and saved 14 million lives. Ultra intercepts helped the Allies in the Battle of Britain, the Battle of the Atlantic, and the Operation Overlord Normandy landings.

Although the Bletchley Park work fell under the domain of the military, there was very little hierarchy, and the organizational style was open. The decryption process used a pipeline approach, with separate huts (physical buildings on the campus) performing each stage of interception, decryption, cataloguing and analysis, and dissemination. There was deep cross-functional collaboration within a hut, but extreme secrecy between each of them.

There was a constant need for iteration and refinement of techniques to respond to newer Enigma machines and procedures, and even though the work was conducted under an environment of constant pressure, the codebreakers were encouraged to take two-week research sabbaticals to improve methods and procedures. There was also a log book in which anyone could propose improvements, and potential improvements were discussed every two weeks.

Shoup discussed how diversity of experience is critical to code breaking, and Bletchley Park recruited linguists, mathematicians, bank clerks, crossword experts, and department-store managers. The team consisted of "boffins" and "debs", and at its peak employed 10,000 people, 75% of whom were women. Shoup called out the contributions of several people involved in the work, including Mavis Batey, Alan Turing, and Tommy Flowers.

The second story discussed the role of the Skunk Works, the Lockheed Advanced Developments Projects group founded in 1943, which produced generation after generation of the world's fastest, highest-flying, and stealthiest aircraft, such as the P-80 Shooting Star, U-2 Dragon Lady, SR-71 Blackbird, and F-117 Nighthawk. The design, development, and manufacture were conducted within a single cross-functional facility, which is not typical for modern aircraft.

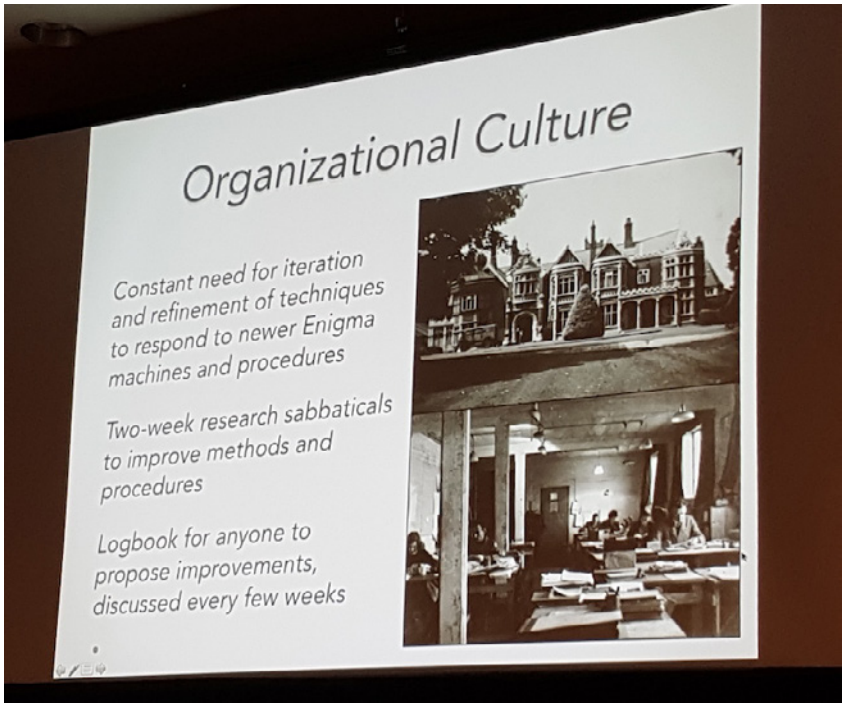
The aircraft were designed and built by the Skunk Works group in co-located teams, with designers and technical experts always available on site, and with intense collaboration with test pilots. There was extensive use of modelling, computational simulation, and wooden mockups for prototyping and verifying hypotheses. Clarence "Kelly" Johnson, the first team leader of the group, created an organizational culture focused on rapid iteration, flexibility, and collective ownership, and encouraged a direct relationship between design engineers, mechanics, and manufacturing. Johnson created 14 rules that captured this doctrine. Ben Rich, the second director of the group, also believed that everyone was responsible for quality:

We made every shop worker who designed or handled a part responsible for quality control. Any worker — not just a supervisor or a manager — could send back a part that didn't meet his or her standards.

Shoup also highlighted the work by Mary Ross, the first Native American female engineer, from the Cherokee Nation. She was part of the founding team of 40 Skunk Works engineers, and contributed to the P-38 Lightning, Agena rocket, ballistic missiles, and satellites.

The final story focused on Xerox PARC (Palo Alto Research Center), founded in 1970. The influence of this organization on the computing industry should not be underestimated, and it produced the first graphical user interface and overlapping windows on a screen, object-oriented programming with Smalltalk, WYSIWYG text editing with Bravo, networking with Ethernet, and modern printing via the laser printer.

Xerox PARC had a flat organization with no hierarchy, and was designed as a hybrid of academia and industry. There were regularly scheduled Dealer meetings, at which team members took turns presenting ideas and defending them against the ques-

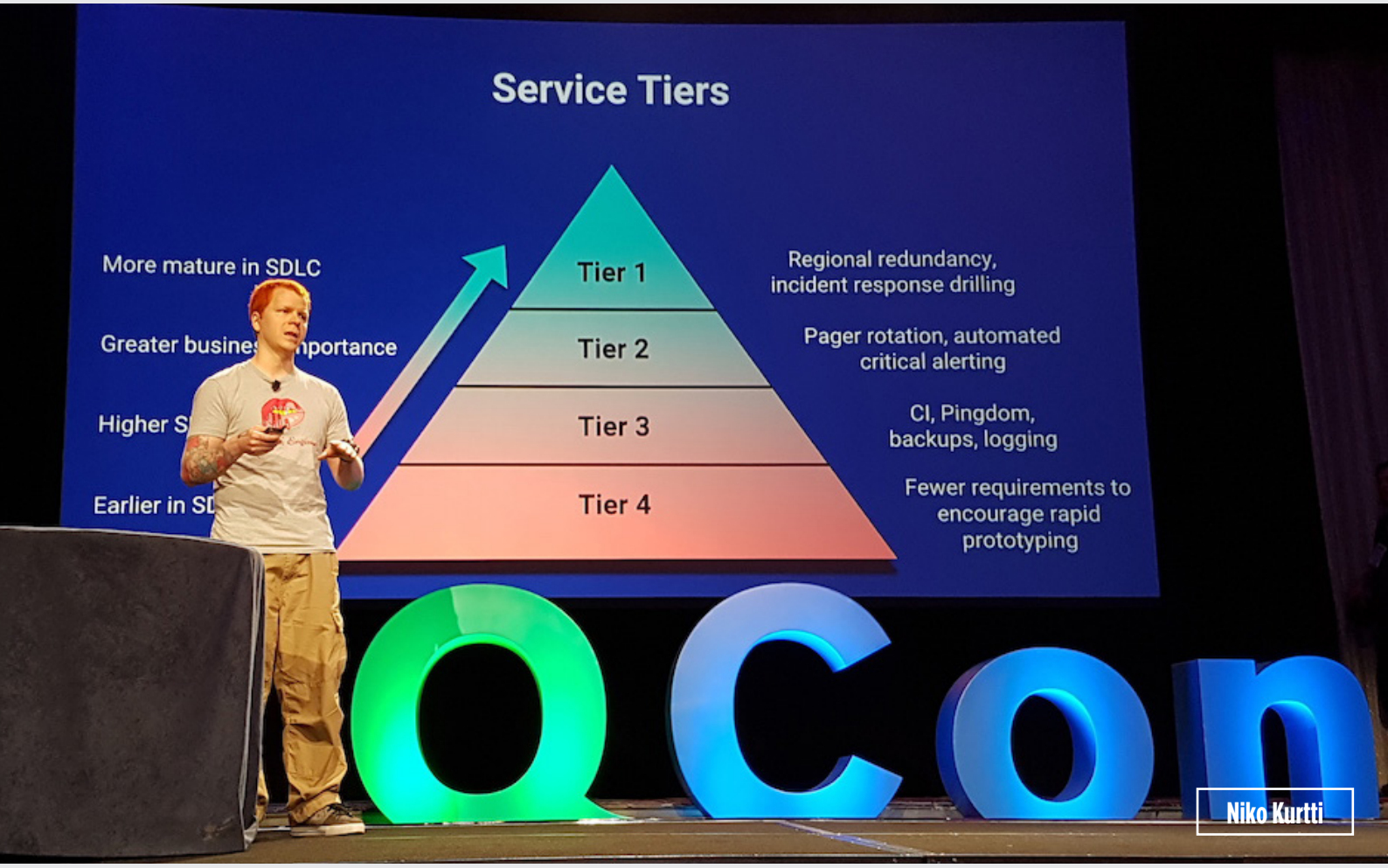


tions and inquiry from the rest of the group. The facilitators of the Dealer meetings were careful to make sure that only intellectual criticism of the merit of an idea received attention and consideration. These debates helped improve products under development and sometimes resulted in completely new ideas for future pursuit. The group also practiced what they called "Tom Sawyering" — informal and dynamic collaboration across groups and projects — which enabled a continuous form of peer review.

Alan Kay challenged his team at PARC to create the world's most powerful programming language in a single page of code. "Simple things should be simple," he said, and although the ultimate output, the Smalltalk language, was slightly longer than two pages, its elegance and simplicity inspired the creation of subsequent object-oriented languages, including Java. In addition to Kay's work, Shoup also highlighted the work of Adele Goldberg, the co-developer of Smalltalk-80 and design templates (later called "design patterns"), and his father, Richard Shoup, who in the early 1970s developed SuperPaint, the world's first digital graphics system. The elder Shoup inspired collaborated with Pixar co-founder Alvy Ray Smith.

Concluding the talk, Shoup revisited his earlier mention of the need for focus on purpose, organizational culture, people, and engineering excellence within high-performing organizations. Teams should be encouraged to think big, and they need to be laser-focused on an important and motivating goal for the entire organization. Cross-functional, full-stack teams are most effective for this type of work, and autonomy should be maximized amid minimal bureaucracy and central control. Collaboration is key, and so is cultivating a learning culture. The best people should be hired for the job, regardless of background — diversity of experience and perspective is highly valuable — and they should be treated well. Engineering excellence is established through systems thinking and looking for holistic solutions to problems. There must be a pragmatic focus on delivering, and constant iteration and feedback must be baked into all processes. Although many of the principles of agile, lean and DevOps may appear relatively modern, they might not be as new as many of us think.


Frank
GRECO
NY JavaSig



Read online on InfoQ

Niko Kurtti at QCon NY

Shopify's Journey to Kubernetes and PaaS

by **Daniel Bryant**

At QCon New York, Niko Kurtti presented "Forced Evolution: Shopify's Journey to Kubernetes", which described the Shopify engineering team's journey to building their own PaaS with Kubernetes as the foundation.

Key takeaways for other teams looking to build their own PaaS and associated developer workflow included targeting 80% of deployment and operational use cases; creating patterns and hiding the underlying platform complexity, educating people and getting them excited about the project; and being conscious of vendor lock-in.

Kurtti is a production engineer at Shopify, a rapidly growing Canadian e-commerce company that offers a proprietary e-commerce platform for online stores and retail point-of-sale systems. Shopify has more than 3,000 employees and processed $26 billion is transactions in 2017. The underlying e-commerce software platform sees more than 80,000 requests per second during peak demand.

At the start of 2016, the Shopify engineering team was "running services everywhere", including in their own data centers (using Chef and Docker), on AWS (using Chef and a variety of other tools), and with Heroku. Developers liked the developer experience

of Heroku, and Kurtti commented that this platform actually scales quite well, with simple UI sliders to increase the number of instances and associated CPU and RAM. Although the platform team had defined service tiers and appropriate service-level objectives (SLOs) based on criticality to the business, there were many processes that were not scalable, and accordingly these presented challenges as the company grew.

Kurtti continued by stating that manual or artisanal processes clearly did not scale well, and neither did slow processes that make people wait. Shopify teams encountered challenges with "rusty knobs that don't work when needed" within the platform and deployment operations, and also processes that did not work first time or reliably. Accordingly, the they recognized that they needed to increase their focus on tested infrastructure and automation that works as expected every time. Also critical to the ability to scale was giving developers the ability to safely self-serve in a consistent manner across the infrastructure/platform, and providing comprehensive training to enable them to become experts in the systems they operate. Alongside these new initiatives, the organization had also decided to embrace cloud computing, and were keen to promote migration to their chosen cloud vendor, Google Cloud Platform (GCP).

The Shopify engineering team recognized that they were effectively building an internal platform as a service (PaaS), and so decided that three principles were key to its success: 1) providing a "paved road", a platform that would by default meet a high percentage of use cases within Shopify but also allow customisation if required (the Netflix engineering team discussed a similar concept at 2017's QCon New York); 2) hidden complexity — there are advantages to knowing about the underlying platform but many developers do not want to be exposed to all of the details of the internals; and 3) self-service is a priority — developers should not have to wait for centralized operations or platform teams.

After analysis and experimentation, the Shopify team chose to build their PaaS on top of the Kubernetes container schedulers and orchestrator. Kubernetes had the best traction of the open-source projects within this space, it was platform agnostic, it could be extended via the APIs exposed, and Google offered it as a service in GCP (Kubernetes Engine), which allowed the team to focus on the value-adding components they could provide on top on this "strong foundation".

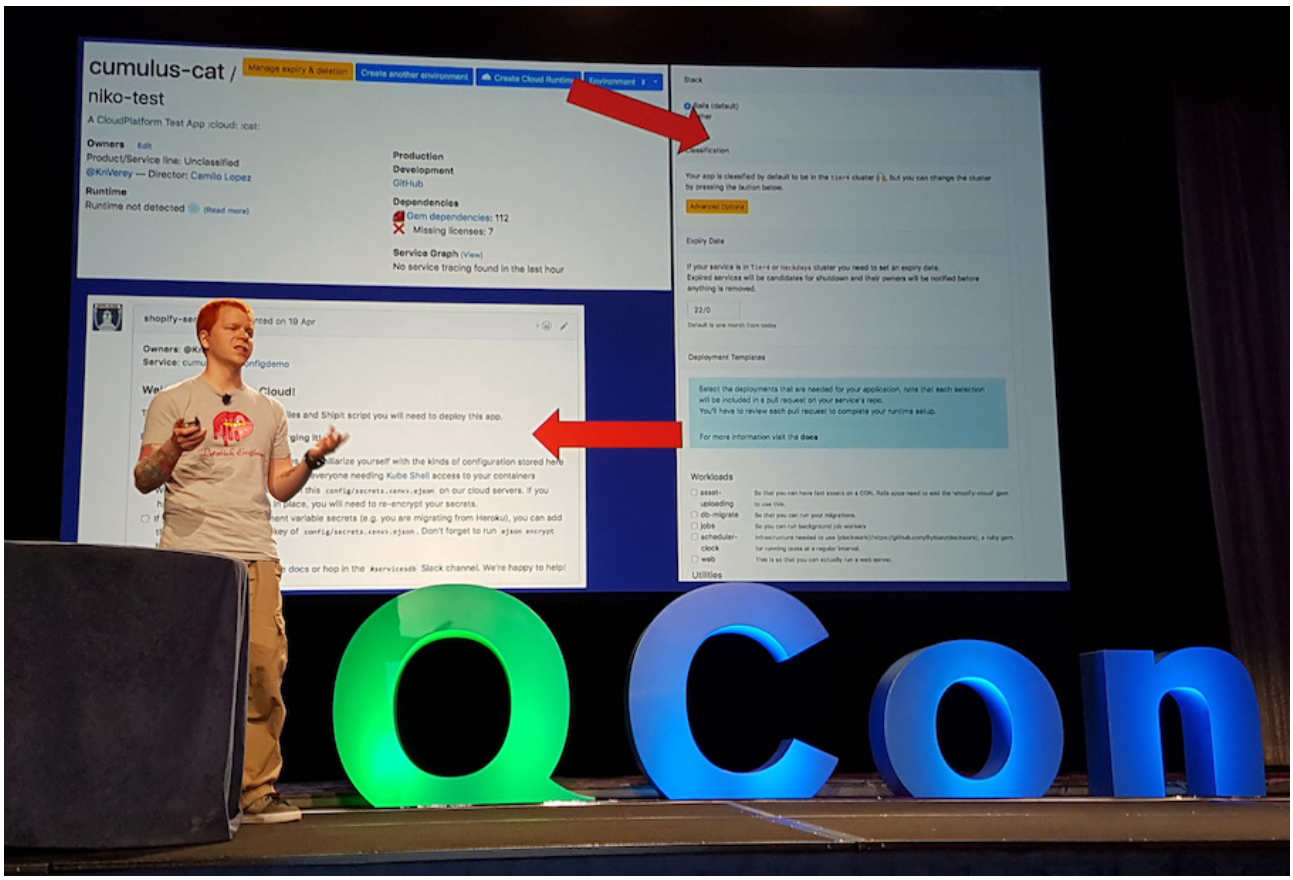
Kurtti listed four building blocks of running an application on the Shopify PaaS were:

- how to specify an application's runtime;
- how to build an application;
- how to deploy an application; and
- how to set up dependencies.

Accordingly, the engineering team created Services DB and the Groundcontrol tool. Services DB provided an interactive web UI for developers that included a catalogue of existing applications and a mechanism for the automated generation of associated Kubernetes manifests alongside build and continuous-integration configuration. The Groundcontrol system was a Go-based application that was deployed on the Kubernetes clusters to create namespaces and encryption keys and manage service accounts. An example of the web UI is shown below, demonstrating the flow from a developer initializing a project to its automatic deployment within a test namespace on a Kubernetes cluster.

The team created additional tooling that included PIPA, an agent that builds Docker images, and Buildkite, which acts as a coordinator for PIPA. The tools combine to provide a "Heroku-ish" workflow by default, and can also be used to specify bespoke Dockerfiles or create custom pipelines. They also created kubernetes-deploy (and released it as open source), which is a "command-line tool that helps you ship changes to a Kubernetes namespace and understand the result". The tool is pluggable and provides a simple pass/fail result on deploys. It also configures ConfigMaps and secrets, and protects Kubernetes namespaces. All of these tools integrated with Shopify's open-source Shipit deployment tool that is extensively used internally within the company.

The platform team has invested heavily in the creation of "cloudbuddies", which are effectively custom extensions on Kubernetes in much of the same style as CoreOS's Operator pattern. Cloudbuddies extend the Kubernetes API and manage processes such as creating DNS records, configuring cluster/user quotas, and setting security rules. The cloudbuddies have highly influenced the success of the new platform, and Kurtti discussed how extending Kubernetes has been generally a good experience. The Kubernetes APIs are well documented ("if not super stable") and the Go client libraries are of high quality. Current concepts (like Deployment and Endpoints objects) and custom entities (using custom resource definitions) can be extended. It provides distribut-



ed-systems primitives. As extensions are written in pure Go, they can be unit-tested and run and deployed as normal applications.

The platform team does not currently expose the Kubernetes control plane to developers, and instead of using tooling like kubectl, they deploy and operate applications via the provided web UI. This UI provides the majority of functionality that kubectl does, and in the future kubectl itself may be exposed to power users within the development team. Extensive documentation focuses on "how to drive the car" rather than "how to build the car", meaning that the developer experience and operation has taken priority over explaining the technical foundations of the platform. Kurtti also praised the efforts of his Shopify teammate Jenna Black, who provides on-call cloud help during the working day, and stated that working alongside people who have specialized expertise in (and value the importance of) the support function is extremely beneficial.

Kurtti said the development team has extensively praised the platform, focusing in particular on the ease and short time required to get an application running on the cloud platform. However, challenges remain. The platform team is currently focusing on providing insight into how everything works for the developers, and also is addressing common development issues such as scaling and debugging. The platform site-reliability team has had to embrace giving up control of the underlying infrastructure (in regard to the use of the fully managed Kubernetes Engine). Providing a single platform to meet all common use cases has been challenging.

For engineers looking to build their own, Kurtti and the Shopify team provided several key takeaways: target 80% of use cases, create patterns and hide complexity, educate and get people excited about the project, and be conscious of vendor lock-in.


QCon
NEW YORK 2018 by InfoQ
BLOCKCHAIN ENABLED
MODERN CS IN THE REAL WORLD
EMPOWERED TEAMS
MODERN JAVA RELOADED
ASK ME ANYTHING AND OPEN SPACE
SPONSORED SOLUTIONS TRACK II
BREAK
LUNCH
Shubert Complex
Foyer
Foyer
Liberty
HE FRONT DESK

Read online on InfoQ

Haley Tucker Discusses Chaos Engineering at QCon NY

Learning to Bend But Not Break at Netflix

by **Daniel Bryant**

At QCon New York, Haley Tucker presented "Unbreakable: Learning to Bend but Not Break at Netflix" and discussed her experience with chaos engineering while working across a number of roles at that company.

Key takeaways included: use functional sharding for fault isolation, continually tune RPC calls, run chaos experiments with small iterations, watch out for the environmental factors that may differ between test and production, use canary deployment, invest in observability, and apply the principles of chaos when implementing supporting tooling.

Tucker began by discussing that a key performance indicator (KPI) for the Netflix engineering team is playback starts per second (SPS). Netflix has 185 million customers, and the ability to start streaming content at any given moment is vital to the success of the company. The Netflix system is famously implemented as a microservice architecture. Individual services are classified as critical or non-critical, depending on whether or not they are essential for the basic operation of streaming content to customers. High availability is implemented within the Netflix system by functional sharding, remote-procedure-call (RPC) tuning, and bulkheads and fallbacks. The design and implementation of this is

verified through the use of chaos engineering, also referred to as resilience engineering.

Tucker has worked across a number of engineering roles during her five years at Netflix, and accordingly divided the remainder of her presentation into a collection of lessons learned from three key functions: non-critical-service owner, critical-service owner, and chaos engineer. The primary challenge as a non-critical-service owner is how to fail well; as a critical-service owner, the focus is instead how to stay up in spite of change and turmoil; and for chaos engineering, it is how to help every team build more resilient systems.

The first question to ask when owning a non-critical service is "how do we know the service is non-critical?" An example was presented using the movie-badging service, a seemingly non-essential service that provides data to allow audio/visual metadata, such as "HDR" and "Dolby 5.1", to be displayed as a badge on the UI. Although this non-critical service was called from a downstream service via the Hystrix fault-tolerance library and included a unit-tested and regression-tested fallback, an untested data set caused an exception to be thrown for a subset of user accounts, which bubbled up through the call stack and ultimately caused the critical API service to fail. This resulted in a series of playback start failures for the subset of customers affected.

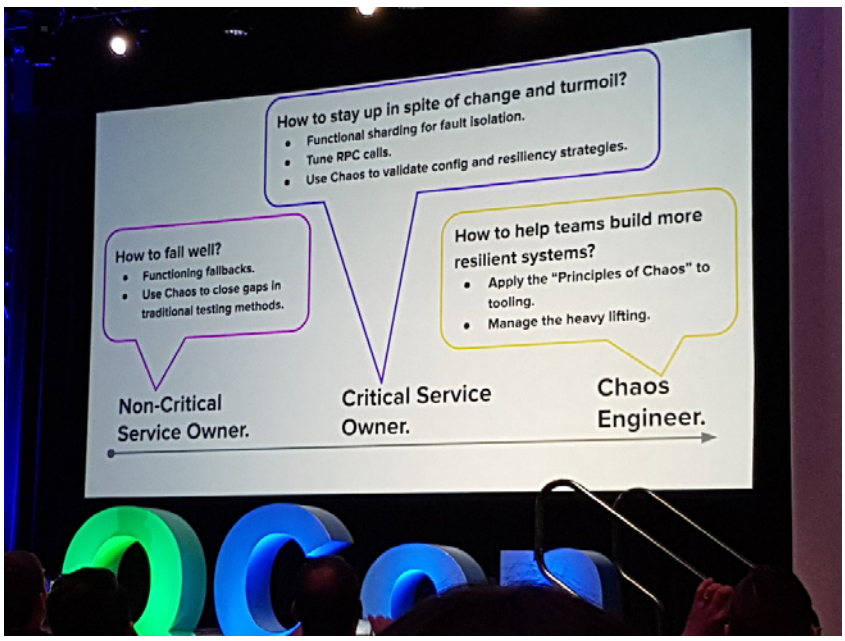
The owner of a non-critical service must consider that environmental factors may differ between test and production (e.g., configuration, data, etc.); that systems behave differently under load than they do in a single unit or integration test, and that users often react differently than expected. Tucker stressed that fallbacks must be verified to behave as expected (under real-world scenarios) and that chaos engineering can be used to close the gaps in traditional testing methods.

Critical-service owners essentially look to always decrease the blast radius of failures. Tucker explained that the Netflix engineering team experimented with sharding services into critical and non-critical cluster shards, and then failed (in a controlled manner) all of the non-critical shards. Although all the critical services remained operational, traffic increased by 25% on the critical-service shards. This additional traffic was the result of the critical services having to perform additional tasks, which were previously initiated by the non-critical services. For example, non-critical services typically pre-fetch or pre-cache certain content that is predicted to be required in the near future.

An additional consideration for critical service owners is to confirm that the RPC system is properly configured. For example, engineers must know that they have optimally configured retries, timeouts, load-balancing strategies, and concurrency limits. To reveal performance, the team runs chaos experiments and injects latency and failures into service calls. Tucker cautioned that this is especially challenging for a fast-paced organization like Netflix, which has a constantly changing code base and service-call graph and therefore must continually experiment.

The key takeaways for critical-service owners include the use of functional sharding for fault isolation, continually tuning RPC calls, chaos-testing with a few changes between experiments in order to make it easier to isolate any regressions, and fine-grained chaos experiments help to scope the investigation as opposed to outages with potentially lots of red herrings.

For the final section of the talk, Tucker discussed her role as a chaos engineer. The primary focus there is how to help teams build more resilient systems. The solution is to do more of the heavy lifting and provide tooling and guidance to service teams. After introducing the Principles



of Chaos Engineering, Tucker shifted the discussion to the importance of running chaos experiments in both test environments and, critically, in production. Netflix engineers always attempt to minimize the damage during experiments and have implemented a "stop all chaos" button that, when triggered, immediately terminates all chaos experiments at the company.

Extensive use is made of canary deployments and testing, with a small percentage of production traffic split off to both a control and experimental version of a service. Netflix allows a maximum of 5% of traffic to be exposed to chaos at any one time, and experiments are limited during holiday periods, when customer demand peaks. It is essential to address any failures identified during an experiment, and the chaos team's web UI will not allow experiments that resulted in issues to be re-run unless verified and acknowledged manually. Tucker recommended that chaos experiments should fail open for safety, and this should be triggered when the control-service errors are high, errors in the service are unrelated to the experiment, or platform components have crashed.

Chaos experiments should always start with a hypothesis, and an essential prerequisite of this is that the steady state of the system must be known. The best way to achieve this is through comprehensive observability, such as effective monitoring, analysis, and insights. Tucker discussed Netflix's open-source continuous-deployment tool Spinnaker, alongside the Kayenta for automated canary analysis. She also briefly mentioned the Netflix Chaos Automation Platform (ChAP) and the verifications the platform undertakes when running chaos experiments, including: validating

the fault injection; validating that KPIs are within expected values; checking for service failures, even if this did not impact KPIs; and checking the health of the service under test.

Real-world events can be automatically validated by carefully designing and prioritizing experiments. The system under test must be well understood and any potential ramifications of testing known beforehand. For example, the ChAP web UI warn engineers of services that have no fallbacks. When prioritizing which experiments to run, engineers should consider factors such as the percentage of traffic a service receives, the number of retries on a service call, the experiment type (failure or latency), and how long it's been since the previous experiment was run. However, safety (preventing a customer-impacting outage) is always the top priority.

Tucker offered a brief case study of running ChAP experiments in production, which found 14 vulnerabilities (identifying key tooling gaps) and produces no outages. Key takeaways for a chaos engineer include applying the principles of chaos to tooling and providing self-serve tooling so that service teams can avoid the heavy lifting of running chaos experiments themselves.

Tucker said that any organization can potentially derive value from starting a chaos practice. The company does not have to operate at the scale of Netflix in order to create hypotheses and design and run basic resilience experiments in test and production. Quoting one of her favorite Netflix shows, Unbreakable Kimmy Schmidt, she closed by stating that when dealing with the inevitable failure scenarios, "You can either curl up in a ball and die… or you can stand up and say 'We are different. We are the strong ones, and you cannot break us!'"


Adam
LEAR
Stack Overflow
QCon
QCon

SAN FRANCISCO 5-9 NOV


QCon

Introduction

by **Charles Humble**

In mid-November, around 1,600 attendees descended on the Hyatt Regency for the 12th annual QCon San Francisco.

QCon SF attendees — software engineers, architects, and project managers from a wide range of industries, including some prominent Bay Area companies — attended 99 technical sessions across six concurrent tracks, "ask me anything" sessions with speakers, 18 in-depth workshops, and eight facilitated open spaces.

We've already started publishing sessions from the conference, along with transcripts for the first time. The full publishing schedule for presentations can be found on the QCon SF website.

The conference opened with a presentation from Jez Humble and Nicole Forsgren, two of the authors of *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations* — one of InfoQ's recommended books for 2018.

As such, it seems appropriate to round out this eMag with a write-up of three of our favorite DevOps-themed talks from the conference.

Read online on InfoQ

Lessons Learned at LinkedIn

Building Production-Ready Applications

by **Daniel Bryant**

Michael Kehoe presentation of "Building Production-Ready Applications" drew on his experience with site-reliability engineering (SRE) and introduced the tenets of production readiness. He argued that all engineers working across the organization should continually focus on stability and reliability, scalability and performance, fault tolerance and disaster recovery, monitoring, and documentation.

Kehoe, staff site-reliability engineer on the production SRE team at LinkedIn, began by referencing Susan Fowler's book *Production-Ready Microservices* quoting it:

> A production-ready application or service is one that can be trusted to serve production traffic.…
>
> We trust it to behave reasonably, we trust it to perform reliably, we trust it to get the job done and to do its job well with very little downtime.

Changes in software architecture and the software delivery

lifecycle over the past decade have increased the challenges in asserting production readiness and have also provided new opportunities. For example, monolithic applications have been decomposed into multiple microservices, which allows rapid iteration and individual deployment and scaling but also means that more services need to be verified and operated. New techniques like continuous integration, combined with the emergence of different methods of working like those of DevOps and the SRE movement, mean that engineers can automate more and follow established best practices for working together across disciplines.

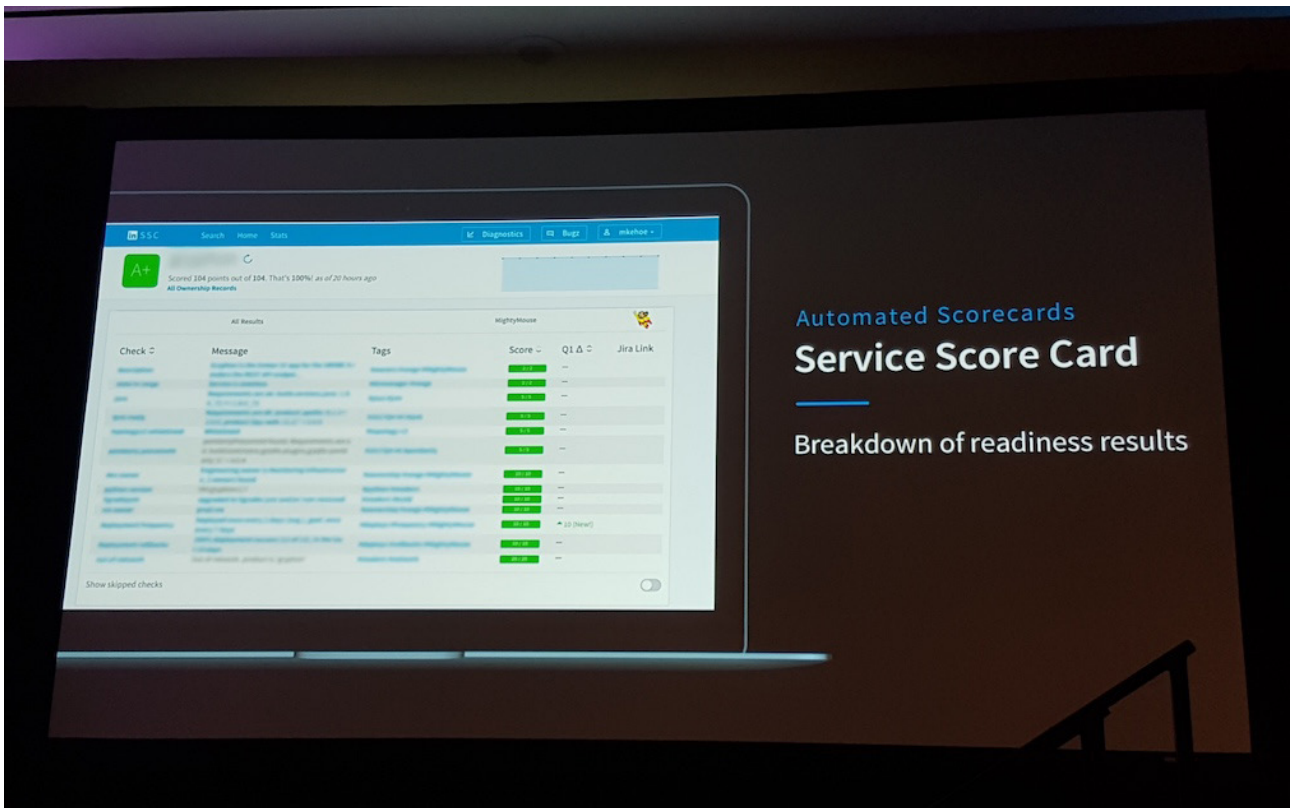
Kehoe stressed the need for stable and reliable development and deployment cycles. In this context, stability is all about "having a consistent pre-production experience". Within development, this should focus on well-established testing practices, code reviews, and continuous integration. For the deployment practice, engineers should ensure that builds are repeatable, that a staging environment (if required) is functioning "like production", and that a canary release process is available.

The value of a staging environment as a concept may be debatable, but if you do have one, you need to treat it like production in order to get value out of it

Unreliability in microservice-based systems usually comes from either change in inbound traffic or change in behavior from downstream services. Engineers should therefore understand production traffic routing, load balancing, and service discovery and associated health checks. Dependency management is key (both from service and code perspectives), and attention should be focused on the engineer onboarding process and the sharing of established best practices. An often overlooked, but important topic, is the service-deprecation procedure.

The discussion of scalability and performance began with a focus on understanding "growth scales": how each service scales with business goals and key performance indicators (KPIs). Engineers should be resource aware, knowing the bottlenecks that exist within the system and the elastic scaling options. Constant performance evaluation is required — ideally, testing this should be part of the continuous-integration process — and so is understanding the production traffic management and capacity planning that are in place.

In regard to fault tolerance and disaster recovery, engineers should be aware of and avoid single points of failure within design and operation. Understanding concepts within the disciplines of resilience engineering and chaos engineering is beneficial in the dynamic and ephemeral world of cloud computing. A team must



know how to react to failure and manage incidents — for example, by creating disaster plans and running game days — and constantly designing and running experiments to test assumptions with failure modes is vital.

Kehoe argued that for monitoring, dashboards and alerting should be curated at the service level, for resource allocation, and for infrastructure. All alerts should require human action, and ideally present pre-documented remediation procedures and links to associated runbooks. Logging is also an underrated aspect of software development; engineers should write log statements to assist debugging, and the value of these statements should be verified during chaos experiments and game days.

The final tenet Kehoe explored was documentation. There should be one centralized landing page for documentation for each service, and documentation should be regularly reviewed (at least every six months) by service engineers, site-reliability engineers, and related stakeholders. Service documentation should include key information like ports and hostnames, description, architecture diagram, API description, and on-call and onboarding information.

With respect to readiness, Kehoe discussed how to implement measurable guidelines to help implementation and to verify assertions. The challenge is that often the general concepts of readiness cannot be directly translated into "something that is true or false" and so cannot be directly scored or measured. One alternative is to focus instead on the outcomes of following specific readiness guidelines.

There is value in creating manual checklists for measuring guidelines and associated outcomes — especially when a team is just starting to explore the tenets of readiness — but the real value derives from the automated generation of readiness scorecards. Kehoe demonstrated several of LinkedIn's internal automated service-checking frameworks and the associated dashboards that are used to drive readiness implementation and verification.

The key conclusions are to create a set of guidelines for what it means for a service to be "ready", automate the checking and scoring of these guidelines, and set the expectations between product, engineering, and SRE teams that these guidelines have to be met as part of a definition of done.


QCon

Read online on InfoQ

Exploring the Changes, Limitations, and Opportunities Within Modern OSs

by **Daniel Bryant**

Justin Cormack explored "The Operating System in 2018" at QCon SF. The biggest changes in this space include performance-driven improvement, such as eBPF and userspace networking, the changing role of operations and how operators use and deploy operating systems, and emulation and portability.

There are also areas with little change so far but which show signs that this is on the horizon — for example, operating systems are effectively the "last monolith", there is a lack of diversity in OSs and OS programming languages and contributors, and security has not yet received the full attention it requires.

Cormack, a software engineer at Docker, began with an old quote of Ken Batcher: "a supercomputer is a device for turning compute-bound problems into I/O-bound problems." He noted that over the past several years "everything has changed". Modern computer storage and networking have become much faster; moving from 1-Gb Ethernet to 100 Gb over the past decade has resulted in a communication speed increase of two orders of magnitude, and SSDs can now

commit data at network-wire speed. Accordingly, modifications to operating systems have been required in order to keep up with these changes.

The first approach to mitigate the issue of increasingly fast I/O is to avoid the kernel/userspace switch latency. System calls are relatively slow and can be avoided by writing hardware device drivers in userspace. For example, DPDK is the most widely used framework in networking, and there is SPDK for non-volatile memory express (NVMe).

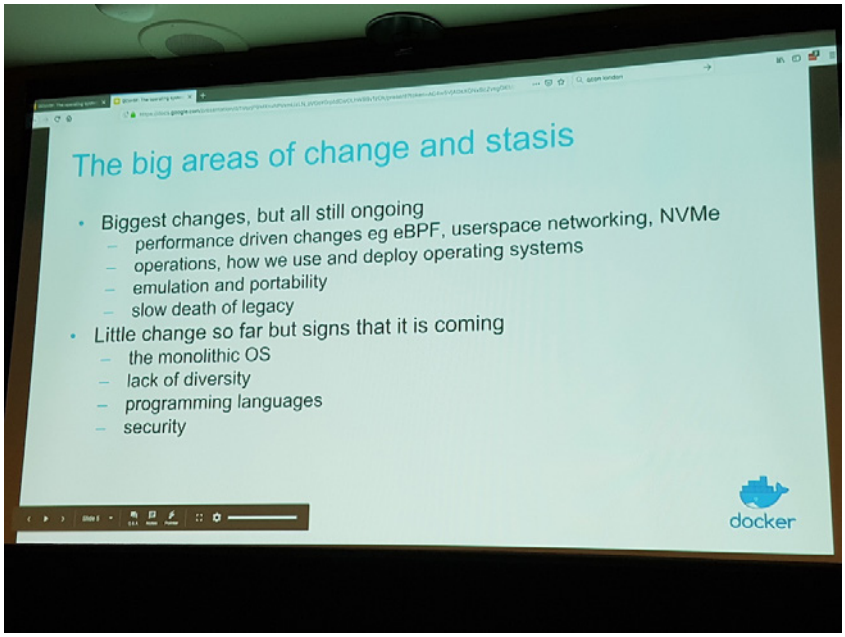
Another approach is to never leave the kernel. Because it is challenging to code for the kernel, eBPF has emerged as a safe in-kernel programming language. Cormack suggested that eBPF is effectively «AWS Lambda for the Linux kernel» as it provides a way to attach functions to kernel events. eBPF has a limited safe language that uses a LLVM toolchain and runs as a universal in-kernel virtual machine. The Cilium service-mesh project makes extensive use of eBPF to perform many networking functions within the kernel at very high speed.

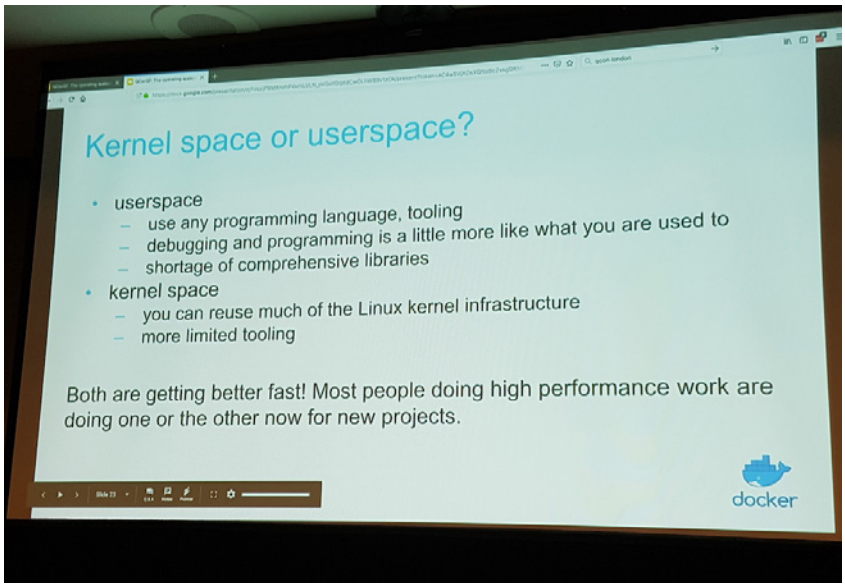
In summarizing this section of the talk, Cormack remarked that choosing between userspace and kernel space for implementing device I/O functionality involves several tradeoffs:

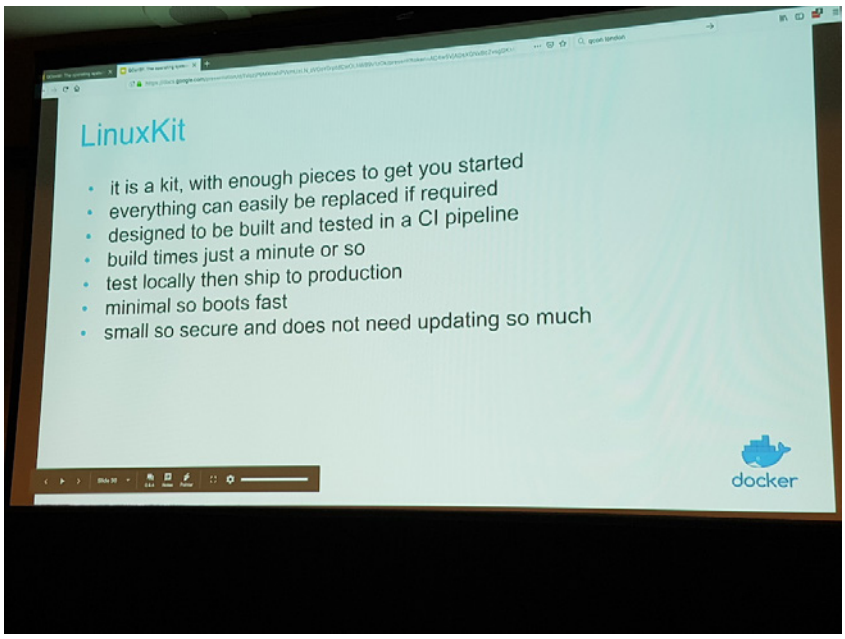
Moving on to the topic of operations, Cormack noted that although the development of UNIX started in 1969, modern (Linux-based) operating systems look little different than the original OSs, with the obvious exception of containing many more packages. However, the role of operations has changed radically over the past 10 years: the vast majority of (cloud-based) OSs never have a person log in to them, most are created via APIs and automation, and the notion of "immutable infrastructure" is now the norm.

As an alternative to existing OS distributions, Cormack discussed LinuxKit. This is an open-source toolkit, which began as a Docker project, for building "secure, lean, and portable" Linux subsystems. LinuxKit is designed to be built and tested within a continuous-integration pipeline, and images are configured using a YAML file. The LinuxKit design philosophy encourages engineers to build composable OSs that can boot very fast, require few resources, and require less security patching (due to the small number of modules included within a typical LinuxKit-built OS).

Cormack explored emulation, beginning with a discussion of how Linux creator Linus Torvalds, decided upfront that Linux would have a stable ABI. This introduced challenges to backwards compatibility but at the same time provided a stable emulation target. Recent work proves this legacy, as the Windows subsystem for Linux was launched in 2016 and







Google released the userspace emulator gVisor in 2018.

This stable emulation target also means that non-performance-critical software can be emulated elsewhere, which is a practice increasingly being used for security isolation — for example, Google safely runs all of the Google App Engine workloads on its multitenant cloud platform using gVisor — and allows existing applications to run on non-Linux-based OSs such as Windows. The longer-term implication is that Linux code, no longer needing to run on Linux, may migrate to new OSs in the future.

Moving on to what has not changed within OSs, Cormack highlighted the declining number of OSs. Only three OS variants currently have significant market share: Linux (and Android), Windows, and macOS (and iOS). Accordingly, Cormack said, this is leading to a monoculture where it is convenient that everything runs on Linux but that also limits new ideas.

With a subtle nod to Torvalds's recent apology e-mail, Cormack also lamented that OS contributor diversity is also poor. Arguably, the operating system is also "the last monolith", with an average Linux distribution weighing in at over 500-million lines of code and Windows is around 50 million, with this being the largest Git repository on the planet. The majority of OS code is also written using C, and other languages have not yet made much impact.

The final topic to be discussed was security. Linux has historically preferred agility over security, and "security as a driver for operating system change [has been] slow". The recent Meltdown and Spectre side-channel security attacks may have affected priorities but the influence of security requirements will take time to trickle down through the OS design process.

Drawing on his experience at Unikernel Systems (acquired by Docker in 2016), Cormack thinks that unikernels could be the radical answer to these challenges. Unikernels allow an engineer to build an OS as a library that is linked to their application, which is specialized to run only this application. A unikernel can be booted directly on hardware or a VM, and due to the specialization and limited number of components, the performance profile is typically very good and the security attack surface can be minimized. There are a number of successful unikernel projects, including Microsoft's SQL Server for Linux, and a growing community in the space — for example, around the OCaml and IncludeOS projects.

In closing, Cormack reflected that perhaps OSs have changed after all. Performance requirements and I/O improvements meant that engineers have created two new ways to run code in Linux: in userspace as self-contained systems that use little of the OS and as in-kernel eBPF, which can be thought of as "AWS Lambda for Linux". The changing role of operations over the past decade is forcing OSs to become more composable and API-driven (e.g. LinuxKit), and emulation is making code more portable. Security will drive the next changes, and unikernels are emerging as a strong area of interest. Attendees and readers must fight the monoculture and strive to make diversity and inclusion across technologies and communities the new norm.


QCon

Read online on InfoQ

Sangeeta Handa at QCon SF

Building Resilience in Netflix Production-Data Migrations

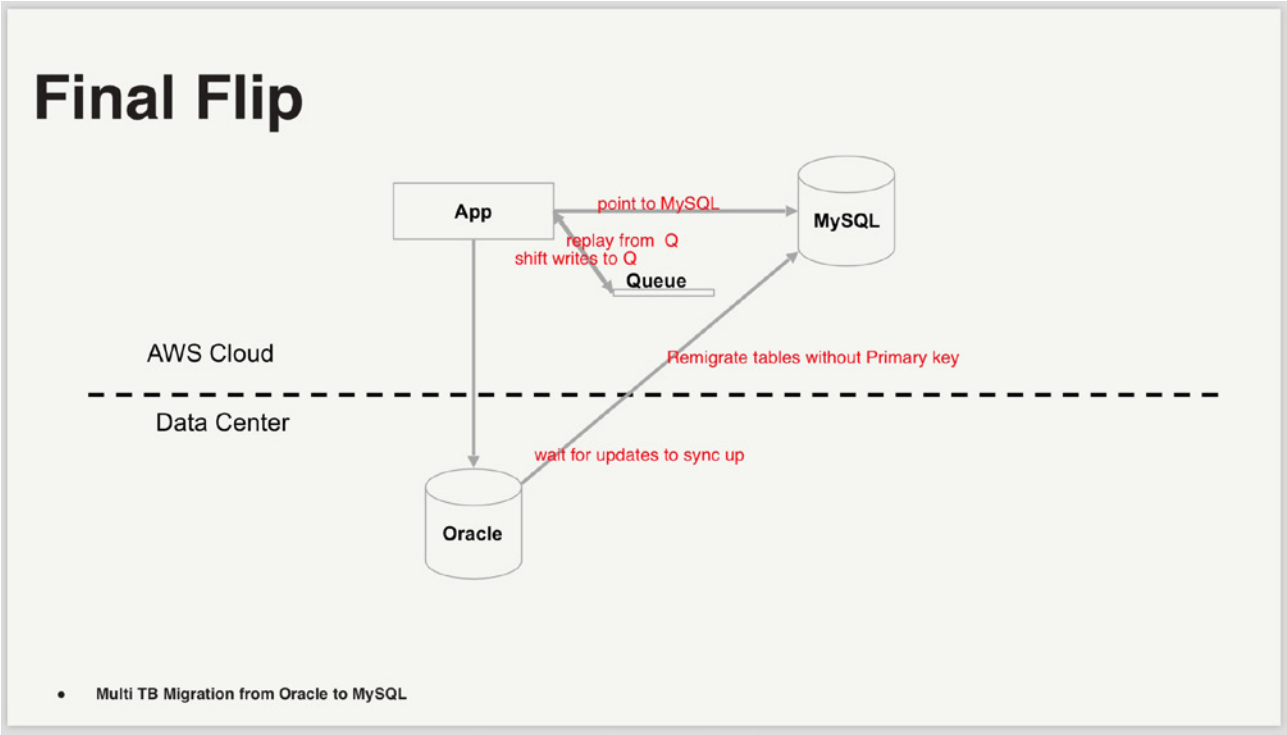
by **Daniel Bryant**

Sangeeta Handa discussed how Netflix has learned to build resilience into production migrations across a number of use cases.

Key lessons include: aim for perceived or actual zero downtime, even for data migrations; invest in early feedback loops to build confidence; find ways to decouple customer interactions from your services; and build independent checks and balances using automated data comparison and reconciliation.

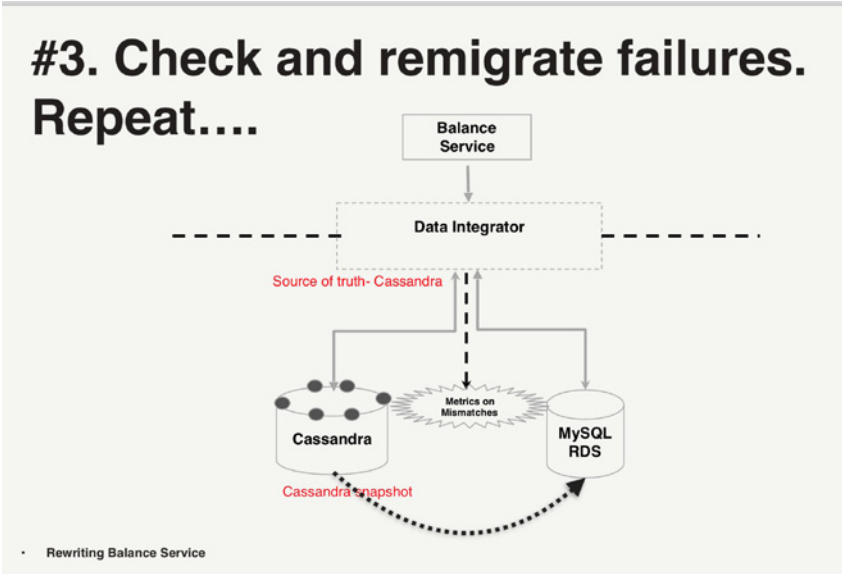
Handa, manager of billing-infrastructure engineering at Netflix, opened by discussing that there are several known risks with the migration of data that is associated with a change in functionality or operational requirements, including data loss, data duplication, and data corruption. These risks can lead to unexpected system behavior on the migrated state, performance degradation post-migration, or unexpected downtime. All of these outcomes have a negative customer impact, and any migration plan should explicitly attempt to mitigate these issues.

Organizations with a large-scale geographically distributed customer base, like Netflix, usually cannot afford to take systems fully offline during maintenance



procedures, so some variation of a zero-downtime migration is the only viable method of change. Variants of this approach that the Netflix team uses within a migration that involves data include "perceived" zero downtime, "actual" zero downtime, and no migration of state. Handa presented three use cases that explored each of these approaches.

The first use case was a migration within the billing-application infrastructure. This application handles recurring subscriptions, allows customers to view billing history, and provides many other billing-related functions. The billing infrastructure is busy 24x7 due to the fact that Netflix's 130+ million subscribers are distributed all over the globe — downtime during any migration is not an option. The plan with this first use case was to migrate the underlying data-storage operating system and database from an Oracle system running in a Netflix data center to a MySQL service running in the AWS cloud. The migration involved billions of rows and over 8 TB of constantly changing data.



Initial experiments with the snapshotting and reloading of data and using Apache Sqoop proved fruitless. Ultimately, the team opted for a "perceived zero downtime" migration approach, which involved copying data at the record level, table by table, alongside the operational read and writes undertaken on the table as part of the daily workloads. This approach allowed the team to pause migration at an arbitrary time and catch failures at the (business) record level. They established confidence in the resilience of the migration by comparing daily automated snapshots from the source and target databases, by checking row counts, and by running test workloads and reports on the target database before fully switching over to the new target database as the single source of truth.

During the final flip, the new target MySQL database had to be ful-

ly synchronized with the source database before application data writes could continue, which would result in actual downtime. However, the team implemented perceived zero downtime by allowing read-only access to the new MySQL database while the synchronization process completed and by buffering writes within an Amazon Simple Queue Service queue. As soon as the state of the flipped database was synchronized, the contents of the queue were replayed into the application, which in turn issued the appropriate writes against the new target database.

The second use case Handa presented was the rewriting of the balance service, which included a migration of a smaller amount of data with a simpler structure from a Cassandra database to a MySQL database. This was undertaken because MySQL was a better fit for the querying characteristics of the corresponding service.

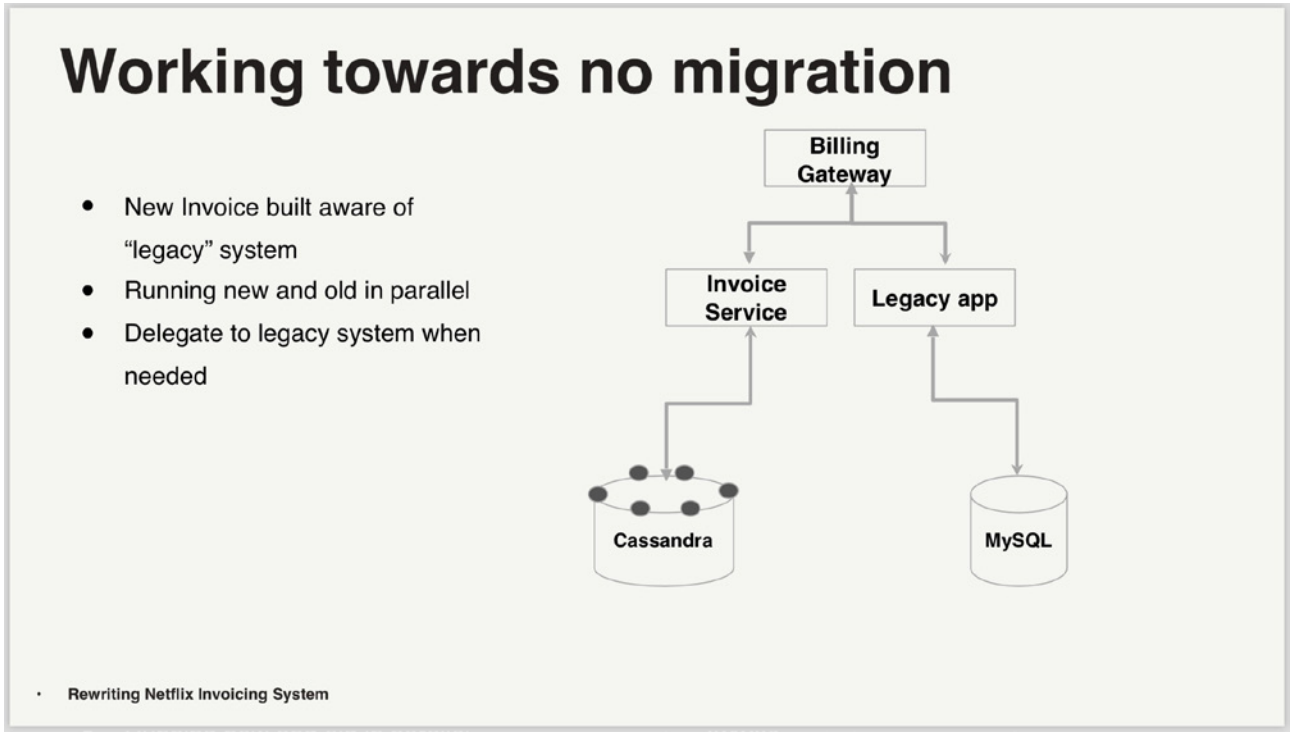
The team achieved the migration by first creating a data-integrator service as a data-access layer between the application service and the underlying databases. The data integrator could effectively shadow writes from the primary Cassandra instance across to the new MySQL data store. The data integrator was also responsible for reporting metrics based on any unexpected results or data mismatches between the two writes.

The second part of the migration process involved asynchronously "hydrating" data from regular Cassandra snapshots into MySQL. The data integrator would attempt to read data from MySQL, and fall back to Cassandra (which was still the source of truth) if data had not yet migrated. Writes continued to be shadowed to both datastores. Metrics of unexpected issues were again captured, and regular comparisons and reconciliations were made between the data stores. Any data with discrepancies was re-migrated from Cassandra to MySQL, with a complete rewrite of customer's data in MySQL even if there was only a single issue.

This hydrate, shadow, and compare process was continued until the required confidence of correctness had been achieved. The final flip in this migration simply involved making MySQL the single source of truth within the data integrator, and removing the underlying Cassandra data store.

Handa's final migration use case was the rewriting of the Netflix invoicing system. This required the creation of an entirely new path of functionality and corresponding service that would also require a new data store. This existing invoice data set was extremely large and had a high rate of change. The plan was to move the data from the existing MySQL database to a new Cassandra instance while not affecting a customer's ability to interact with the invoicing functionality.

After analysis and several design discussions, the team realized that for this migration the corresponding data did not have to be migrated per se. They decided to build the new service using the new Cassandra database and run



this in parallel with the legacy service and data store. By placing a billing gateway in front of the two services, a programmatic decision could be taken on which invoice service to call, based on whether a customer's invoicing data had been migrated or not.

Handa and her team canary-released the new service by incrementally migrating small subsets of data from the old system to the new (for example, copying the invoice data of all users from a small country). They built confidence in the resilience of the migration by testing in shadow mode, where writes were shadowed across the two services and the results compared and reconciled at the data-store level. They used extensive monitoring and alerting to catch the presence of unexpected behavior and state. The effective migration of the data took over a year to achieve as part of the operation of the normal invoicing functionality but there was zero downtime and no impact to the customer.

The key lessons that Handa's team had learned were to aim for zero downtime even for data migration, invest in early feedback loops to build confidence, find ways to decouple customer interactions from your services, build independent checks and balances through automated data reconciliation, and canary-release new functionality: "rollout small, learn, repeat".