The InfoQ eMag / Issue #99 / November 2021

Architectures You've Always Wondered About

InfoQ

Building Tech at Presidential Scale

Using DevEx to Accelerate GraphQL Federation Adoption @Netflix

GitHub's Journey from Monolith to Microservices

InfoQ @ InfoQ InfoQ InfoQ

Architectures You've Always Wondered About

IN THIS ISSUE

PRODUCTION EDITOR Ana Ciobotaru / COPY EDITORS Susan Conant DESIGN Dragos Balasoiu / Ana Ciobotaru
GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

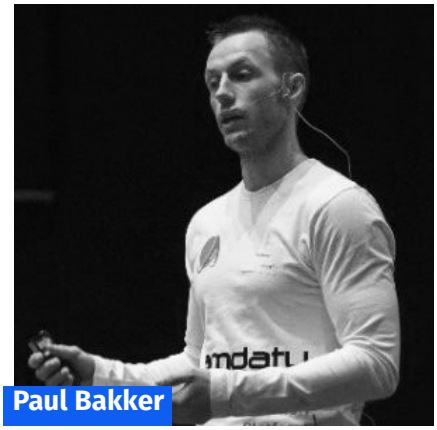
CONTRIBUTORS

Sha Ma

is currently Vice President and Head of Engineering at Catalyst.io, an industry-leading Customer Success Platform. Prior to Catalyst, Sha was the VP of Engineering at GitHub, where she was responsible for Core Platform and Ecosystem. In 2017, as VP of Engineering at SendGrid, Sha was part of the leadership team that took the company public. Sha cares passionately about diversity, equity, and inclusion in the workplace, and in 2018 was named winner of the Denver Business Journal's Outstanding Women in Business in Technology and Telecommunications.

Dan Woods

is CISO and VP of cybersecurity at Shipt. Before and after the working as the CTO of Biden for President 2020, he worked as a distinguished engineer at Target, where he focused on infrastructure and operations with an emphasis on building high-scale, reliable distributed systems. Prior to joining Target, he worked on the Hillary for America 2016 campaign's technology platform by way of The Groundwork, which was a campaign-adjacent technology company. Before his foray into presidential politics, Woods worked as a senior software engineer at Netflix in the operations engineering organization.



Paul Bakker

is a senior software engineer at Netflix, with a focus on developer experience. He has a long history in the Java community, is a Java Champion and author of "Java 9 Modularity" and "Modular Cloud Apps with OSGi", both published by O'Reilly. He has a passion for sharing knowledge and is a frequent conference speaker.

Kavitha Srinivasan

is a senior software engineer at Netflix, with more than 15 years of experience in distributed systems, currently working on developer experience on GraphQL.

A LETTER FROM THE EDITOR

Thomas Betts

is the Lead Editor for Architecture and Design at InfoQ, a co-host of the InfoQ Podcast, and a Senior Principal Software Architect at Blackbaud. For over two decades, his focus has always been on providing software solutions that delight his customers. He has worked in a variety of industries, including retail, finance, health care, defense and travel. Thomas lives in Denver with his wife and son, and they love hiking and otherwise exploring beautiful Colorado.

The “Architectures You’ve Always Wondered About” track at QCon is always filled with stories of innovative engineering solutions. Bringing those stories to our readers is at the core of InfoQ, and this eMag is a curated collection of some of the highlights over the past year. While some of the topics may sound familiar, no two journeys are the same, and every story brings something new. From GitHub to Netflix to the Joe Biden presidential campaign, the organizations discussed cover a wide range of business scenarios, all of which need outstanding software engineering.

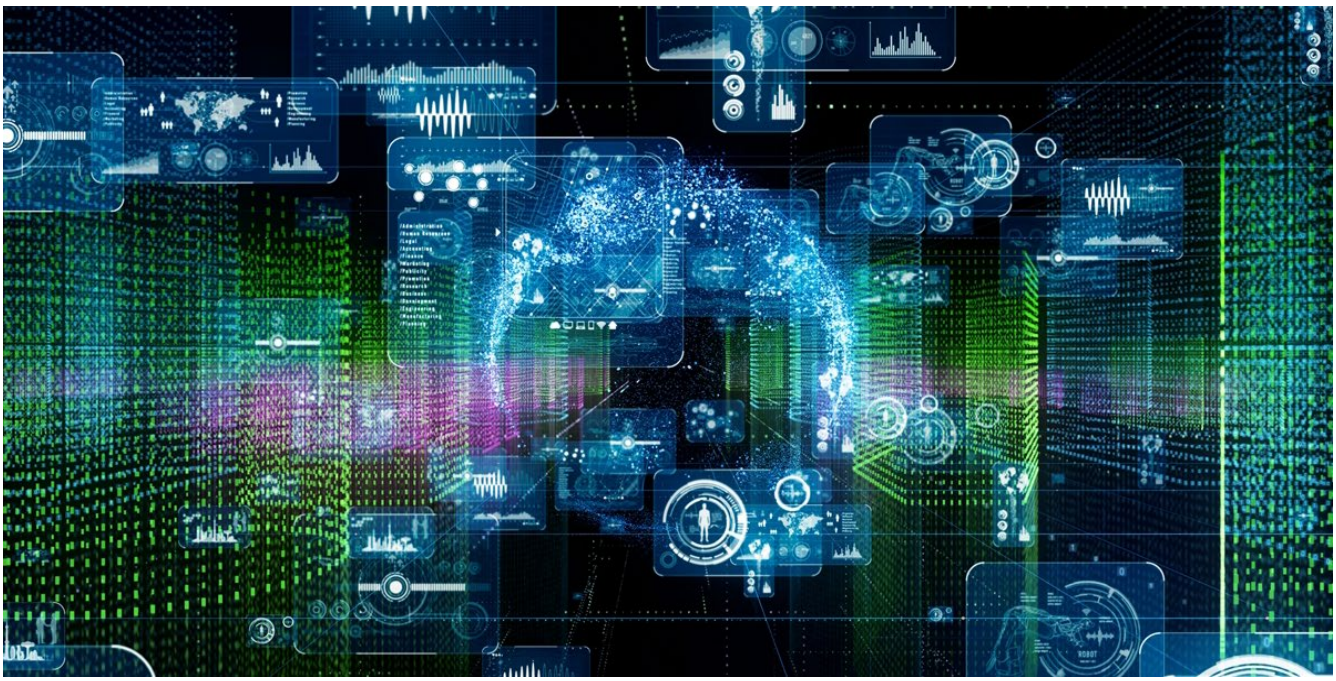
In a situation that only comes around once every four years, managing a US presidential campaign has a long list of technical challenges, coupled with a very demanding, unmovable schedule. While serving as CTO for Biden for President, Dan Woods was responsible for the overall technical operations of the campaign. His story describes the wide range of technology used to handle whatever was needed and to do everything as fast as possible. In the style that matches the fast pace of the campaign, his presentation highlights systems ranging from near-real-time vetting of donors to a peer-to-peer SMS platform to making audio work on Linux.

GraphQL is a great way to simplify querying multiple microservices. And when you’re working for Netflix, the graph gets quite large, and you have a next order problem. GraphQL Federation can be the solution for network bottlenecks, but it means additional work for dozens of development teams. Paul Bakker and Kavitha Srinivasan tell how a focus on the developer experience (DevEx) made the migration process easier for everyone involved. The paved path they created consisted of tools to take care of much of

the new GraphQL code while allowing developers to mostly remain focused on their business logic.

It would be an understatement to say that GitHub has experienced dramatic growth since its founding in 2008. As the number of hosted repositories surpassed 100 million, GitHub rapidly expanded its workforce to over 1,000 internal developers. To allow all those developers to be productive, Sha Ma tells how GitHub migrated from a Ruby on Rails monolith to a microservices architecture. She walks through the decision process on why microservices were the right approach, where they started, and how service-to-service communication patterns were considered for building a resilient system.

We hope you enjoy this edition of the InfoQ eMag. Please share your feedback via editors@infoq.com or on Twitter.



Building Tech at Presidential Scale

by **Dan Woods**, CISO and VP of cybersecurity at Shipt

I was the chief technical officer for US President Joe Biden's campaign during the 2020 election. At Qcon Plus in November 2020, I spoke about some of the elaborate architectures our tech team built and the specific tools we built to solve a variety of problems. This article is a distillation of that talk.

As CTO with Biden for President 2020, I led the technology organization within the campaign. I was responsible for the technology operations as a whole and I had a brilliant team that built the best tech stack in presidential politics.

We covered everything from software engineering to IT operations to cybersecurity and all pieces in between.

I joined the campaign as a distinguished engineer at Target, where I focused on infrastructure and operations with an emphasis on building high-scale, reliable distributed systems. I had previously worked on the Hillary for America 2016 technology platform by way of The Groundwork, which was a campaign adjacent technology company.

Campaign structure

The intricacies of a political campaign's moving parts influence all of the specific choices when considering tech in that environment.

Figure 1 lays out the different departments within the campaign and gives you a sense of the varying degrees of responsibility across the organization.

Each of the many teams across the campaign had its own specific focus. Although campaign tech gets a lot of attention in the press, tech is not the most important thing in politics. The goal for everyone is to reach voters, drive the candidate's message, and connect as many people as possible to the democratic process.

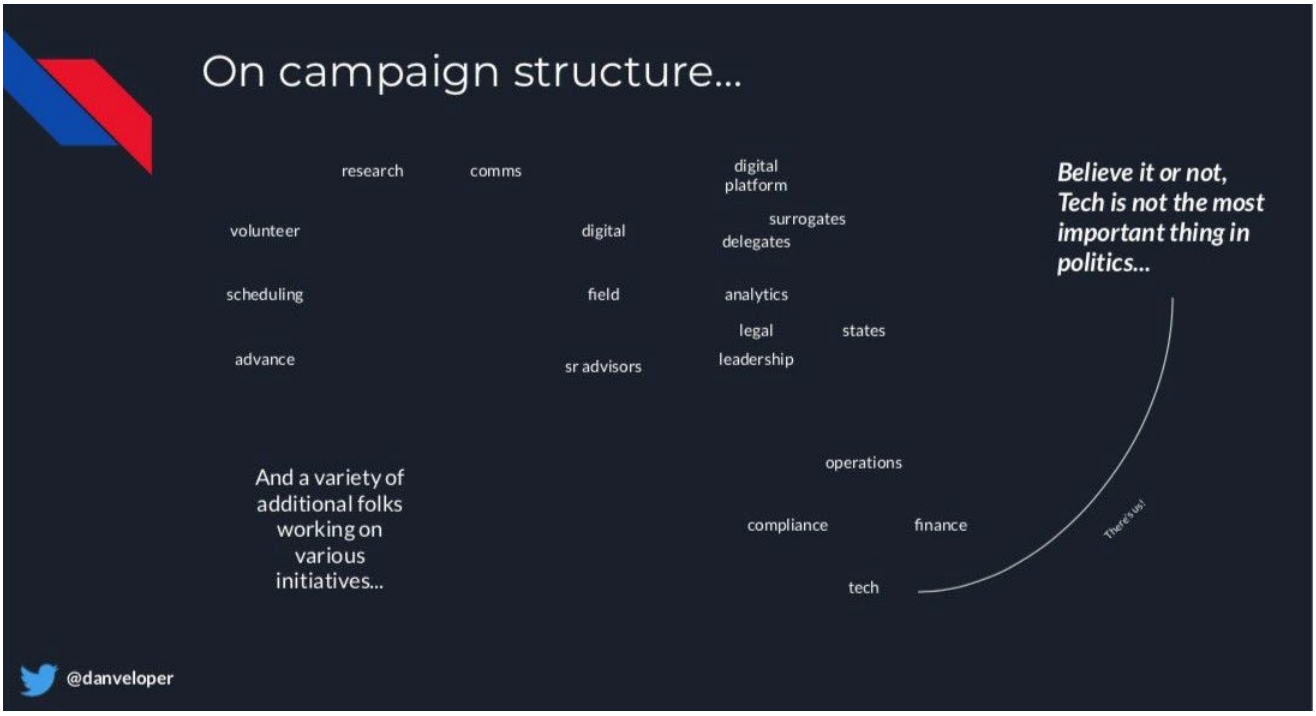


Figure 1: The organizational structure of a political campaign

Role of tech on a campaign

Tech on a campaign does quite literally whatever needs to be done. Nearly every vertical of the campaign needs tech in one form or another. In practical terms, my team was responsible for building and managing our cloud footprint, all IT operations, vendor onboarding, and so forth.

Our approach to building technology during the election cycle is simply to build the glue that ties vendors and systems together. This is not the right environment for delving into full-blown product development.

When a vendor or open-source tool we needed didn't exist or, frankly, when we didn't have the money to buy something, we would build tools and solutions. A huge portion of the work that needs to be done by a campaign tech team is simply getting data from point A to point B. Strictly speaking, this involves creating a lot of S3 buckets.

In addition, we ended up developing a wide range of technology over the year and a half of the campaign cycle. We built dozens of websites, some big and some small. We built a mobile app for our field organizing efforts. We built Chrome extensions to help simplify the workloads for dozens of campaign team members. Though unglamorous, we developed Word and Excel macros to drive better efficiency. On a presidential campaign, time is everything, and anything that we could do to simplify a process to save a minute here or there was well worth the investment.

Much of what we did boils down to automating tasks to reduce the load on the team in any way that we could. It's desirable at times to distill campaign tech down to one specific focus: data, digital, IT, or cybersecurity. In reality, the technology of the campaign is all of those things and more, and is a critical component of everything the campaign does.

What we did

We were a small and scrappy team of highly motivated technologists. We'd take on any request and do our best where we could. Over the course of the campaign, we built and delivered over 100 production services. We built more than 50 Lambda functions that delivered a variety of functionality. We built a best-in-class relational organizing mobile app for the primary cycle,

the Team Joe app, which gave the campaign the leverage to connect thousands of eager voters and volunteers directly with the people they knew.

We had more than 10,000 deployments in the short time we were in operation, all with zero downtime and a focus on stability and reliability.

We implemented a bespoke research platform with robust automation that built on cloud-based machine learning, which saved the campaign tens of thousands of hours of human work, and gave us incredible depth of insight. We built a number of services on top of powerful machine-learning infrastructure.

The campaign early on made a commitment to hold ourselves to a higher standard, and to make sure that we would not accept donations from organizations that harm our planet or from individuals who may have ulterior motives. To hold us to that promise, we built an automation framework that vetted our donors in near real time for FEC compliance and campaign commitments.

When we won the primary, we wanted new, fresh branding on the website for the general election so we designed and delivered a brand-new web experience for joebiden.com.

A huge aspect of every campaign is directly reaching voters to let them know of an event in their area or when it's time to get out to vote. For that, we built an SMS outreach platform that scaled nationwide and saved us millions of dollars in operational expense along the way. Beyond that, we operationalized IT for campaign headquarters, state headquarters, and ultimately facilitated a rapid transition to being a fully remote organization at the onset of quarantine.

Through all of this, we made sure we had world-class cybersecurity. That was a core focus for everything we did.

Ultimately, though, no one job on the campaign is single responsibility, which means everyone needs to help wherever and whenever help is needed, whether that's calling voters to ask them to go vote, sending text messages, or collecting signatures to get on the ballot. We did it all.

Infrastructure and platform

We relied on a fully cloud-based infrastructure for everything we did. It was paramount that we didn't spend precious hours reinventing any wheels for our core tech foundations. We used Pantheon for hosting the main website, joebiden.com. For non-website workloads, APIs, and services, we built entirely on top of AWS. In addition, we had a small Kubernetes deployment within AWS that helped us deliver faster simple workloads, mostly for cron jobs.

We still needed to deliver with massive scale and velocity overall. As a small team, it was critical that our build-and-deploy pipeline was reproducible. For continuous integration, we used Travis CI. For continuous delivery, we used Spinnaker.

Once we deployed services and they were up and running in the cloud, they all needed a core set of capabilities like being able to find other services and securely accessing config and secrets. For those, we used HashiCorp's Consul and Vault. This helped us build fully immutable and differentiated environments between development and production, with very few handcrafted servers along the way — not zero, but very few.

A huge part of the technology footprint was dedicated to the work being done by the analytics team. To ensure they had best-in-class access to tools and services, the analytics data was built on top of AWS Redshift. This afforded a highly scalable environment with granular control over resource utilization.

We used PostgreSQL via RDS as the back-end datastore for all of the services that we built and deployed.

From an operational perspective, we desired a centralized view into the logging activity for every application so that we could quickly troubleshoot and diagnose any problem to achieve the quickest possible recovery. For that, we deployed an ELK Stack and backed it with AWS Elasticsearch. The logs are very important to applications.

Metrics were the primary insight into the operational state of our services and were critical when integrating with our on-call rotation. For service-level metrics, we deployed Influx and Grafana, and wired them into PagerDuty to make sure we never experienced an unknown outage. Many of the automation workloads and tasks didn't fit the typical deployment model and for those, we'd favor AWS Lambda where possible. We also used Lambda for any workload that needed to integrate with the rest of the AWS ecosystem and for fan-out jobs based on the presence of data.

We created a truly polyglot environment. We had services and automation built in a wide variety of languages and frameworks, and we were able to do it with unparalleled resiliency and velocity.

We covered a lot of ground during the campaign cycle. It would take a lifetime to thoroughly detail everything we did, so I'm going to pick out a few of the more interesting architectures the software-engineering side of the tech team put together. By no means should my focus detract from the impressive array of work that the IT and cybersecurity sides accomplished. While I dive deep on the architectures, be aware that for each detail I discuss there are at least a dozen more conversations, covering the breadth of what we accomplished across the entire tech team.

Donor vetting

As noted, the campaign committed early on to reject donations from certain organizations and individuals. The only way to do this at scale is to have a group of people comb through the donations periodically, usually once a quarter, and flag donors who might fit a filtering criteria.

This process is difficult, time consuming, and prone to error. When done by hand, it usually involves setting a threshold for the amount of money an individual has given, and then researching whether or not the donor fits a category we flagged as problematic.

To make this more efficient, we built a highly scalable automation process that would correlate details about a donation against a set of criteria we wanted to flag. Every day, a process would kick off. It would go to NGP VAN, which is a source of truth for all contributions, export the contribution data to a CSV file, and dump that file into S3. The act of the CSV file appearing in S3 would trigger an SNS notification, which in turn would activate a sequence of Lambda functions. These Lambda functions would split the file into smaller chunks, re-export them to S3, and kick off the donor-vetting process. We could see a massive scale in Lambda while this workflow was executing, up to 1,000 concurrent executions of the donor-vetting code.

For example, we committed to not accepting contributions from lobbyists and executives in the gas and oil industry. It would take only minutes for the process to go through the comprehensive list of donors and check them against lobbyist registries, foreign-agent registries, and block lists for gas and oil executives.

Once the process completed, the flagged entries were collated into a single CSV file, which was then re-exported to S3 and made available for download. SES would then send an email to Danielle, the developer who built the system, indicating that the flags were ready for validation.

After validation, the results were forwarded to the campaign compliance team, who would review the findings and take the appropriate action, whether that be refunding a contribution or investigating it further.

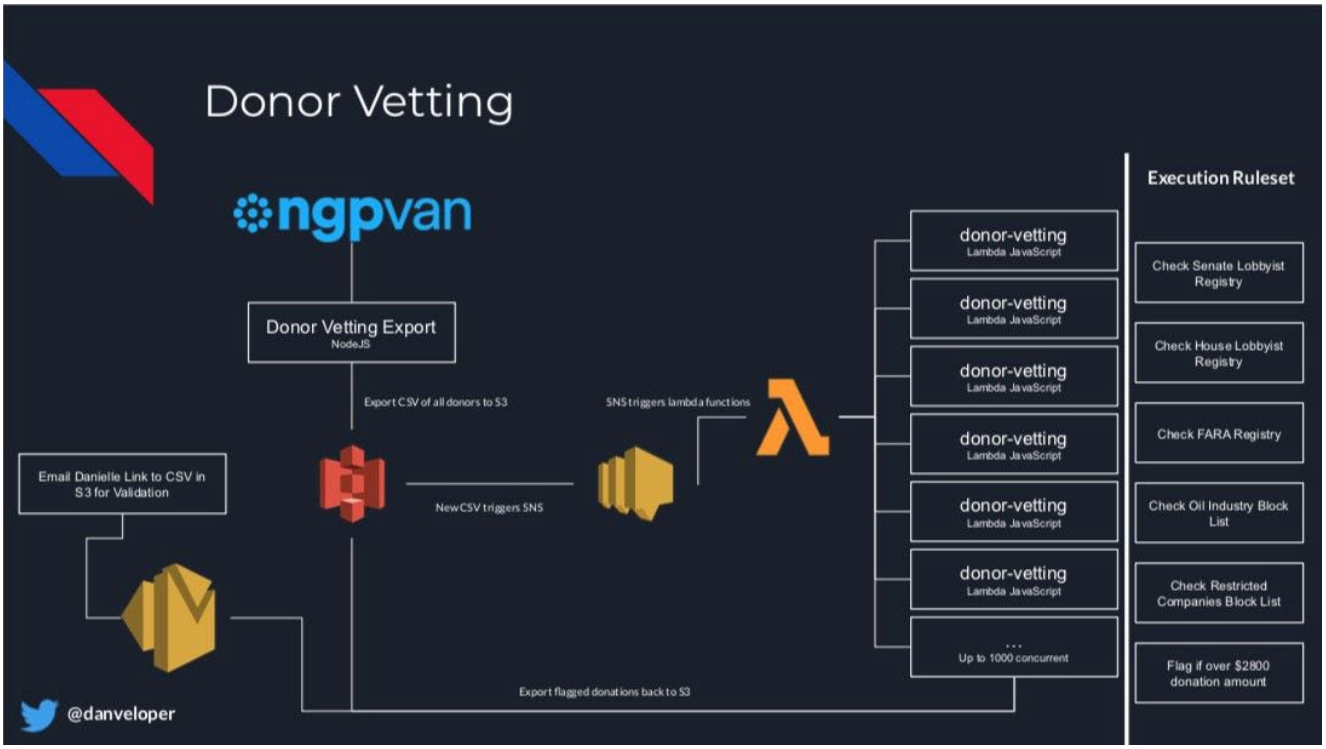


Figure 2: The automated donor-vetting process

To say this process saved a material amount of time for the campaign would be a dramatic understatement. The donor-vetting pipeline quickly became a core fixture in the technology platform and an important part of the campaign operations.

Tattletale

Early on, when we were a small and very scrappy team, we had so many vendors and so many cloud services and not nearly enough time or people to constantly check the security posture on each of them, we needed an easy way to develop rules against critical user-facing systems to ensure we were always following cybersecurity best practices.

Tattletale was developed as a framework for doing exactly that. Tattletale was one of the most important pieces of technology we built on the campaign.

We developed a set of tasks that would leverage vendor system APIs to ensure things like two-factor authentication were turned on or would notify us if a user account was active on the system but that user hadn't logged in in a while. Dormant accounts present a major security risk, so we wanted to be sure everything was configured toward least privilege.

Furthermore, the rules within Tattletale could check that load balancers weren't inadvertently exposed to the internet, that IAM permissions were not scoped too widely, and so forth. At the end of the audit ruleset, if Tattletale recorded a violation, it would drop a notification into a Slack channel to prompt someone to investigate further. It also could notify a user of a violation through email so they could take their own corrective action. If a violation breached a certain threshold, Tattletale would record a metric in Grafana that would trigger a PagerDuty escalation policy, notifying the on-call tech person immediately.

Tattletale became our cybersecurity eyes when we didn't have the time or resources to look for ourselves. It also made sure we were standardizing on a common set of cybersecurity practices, and holding ourselves to the highest possible standard.

Conductor and Turbotots

When we reached a certain point of complexity and span of internal tools, we knew we would need to better organize the API

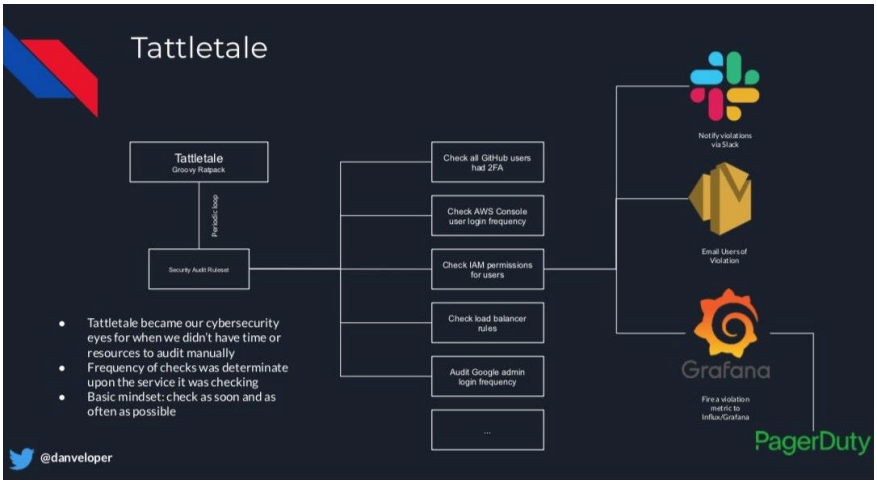


Figure 3: The Tattletale schema

footprint and consolidate the many UIs we had built to manage those systems. We also needed to standardize on a unified security model, so we wouldn't have bespoke authentication and authorization all over the place. And so, we created Conductor (named after Joe Biden's affinity for trains), our internal tools UI, and Turbotots (one of the many tools we named after potatoes), our platform API.

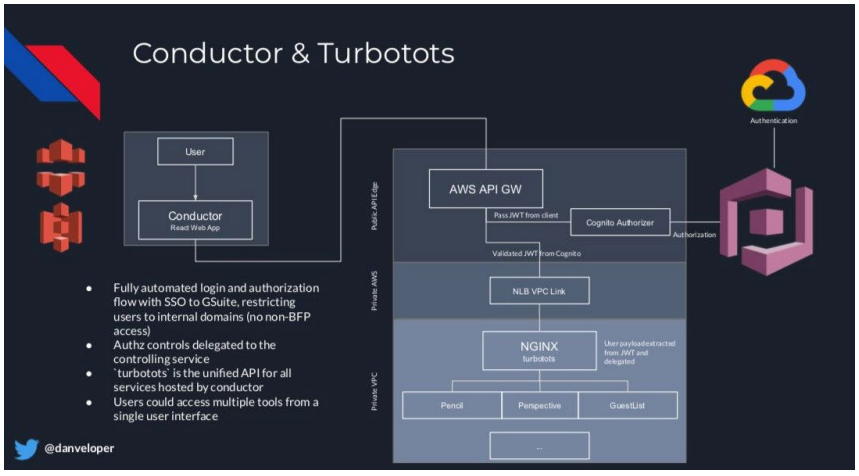


Figure 4: Conductor and Turbotots

Figure 4 is a dramatically simplified diagram of a complex architecture, but these broad strokes are enough to give you an idea of what Conductor and Turbotots did and represented.

Conductor became the single pane of glass for all the internal tools we were developing for use by people on the campaign. In other words, this was the one place anybody on the campaign needed to go to access the services we'd made. Conductor was a React web app that was deployed via S3 and distributed with CloudFront.Turbotots was a unified API that provided a common authentication and authorization model for everything we did, and which Conductor talked to. We built the AuthN and AuthZ portions of Turbotots on top of AWS Cognito, which saved loads of work and provided a simple single sign-on (SSO) through G Suite/Google Workspace, which we were able to configure for only our internal domains. Authentication was achieved by parsing JWT tokens through the workflow. For managing this on the front end, we used the React bindings for AWS Amplify, which integrated seamlessly into the app.

Things get a little bit trickier to understand on the right side of figure 4, but I'll try to simplify. The forward facing API was an API Gateway deployment with a full proxy resource. API Gateway easily integrates with Cognito, which gave us full API security at request time. The Cognito authorizer that integrates with API Gateway also performs JWT validation before a request makes it through the proxy. We could rely on that system to know that a request was fully validated before it made it to the back end.

When developing an API Gateway proxy resource, you can connect code running in your VPC by way of a VPC-link network load balancer. This is complicated but as I understand it, this effectively creates an elastic network interface that spans the internal zone in AWS between API

Gateway and your private VPC. The NLB, in turn, is attached to an autoscaling group that has a set of NGINX instances, which served as our unified API. This is the actual portion that we called Turbotots, and is essentially a reverse proxy to all of our internal services that were running inside the VPC. Following this model, we never needed to expose any of our VPC resources to the public internet. We were able to rely on the security fixtures that were baked into AWS and that made us all a lot more comfortable.

Once the requests made it through to Turbotots, a lightweight Lua script within Turbotots extracted the user portion of the JWT token and passed that data downstream to services as part of a modified request payload. Once the request reached the destination service, it could then inspect the user payload, and determine whether or not the user was validated for the request.

Users could be added to authorization groups within Cognito, which would then give them different levels of access in the downstream services. The principle of least privilege still applies here, and there are no default permissions.

Conductor and Turbotots represented a single user interface for the breadth of internal tools that was fully

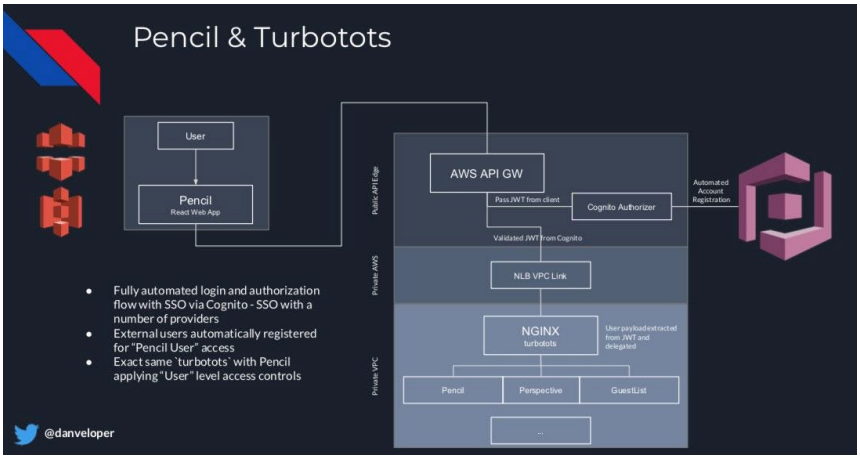


Figure 5: The Pencil architecture looks a lot like Conductor's

secured with seamless SSO to the user's campaign G Suite account. We use this same architecture in the next section to expose a limited subset of the API to non-internal users as well.

Pencil and Turbotots

Pencil was the campaign peer-to-peer SMS platform. Pencil grew from simple set of early requirements into a bit of a beast of an architecture to which we dedicated a significant amount of time.

The original motivation to build our own P2P SMS platform was cost savings. I knew that we could send text messages through Twilio for less money than any vendor would demand to resell us the same capabilities.

At the start, we didn't need the broad set of features that vendors were offering, so it was fairly easy to put together a simple texting system that met the needs of the moment.

The scope of the project grew dramatically as it became popular. Pencil came to be used by thousands of volunteers to reach millions of voters and became a critical component in our voter outreach workflow. If you received a text message from a volunteer on the campaign, it was likely sent through Pencil. Pencil's external architecture is going to look familiar because we were able to reuse everything that we did with Conductor and just change a few settings without having to change any code.

Pencil's user-facing component is a React web app deployed to S3 and distributed via CloudFront. The React app in turn talks to an API Gateway resource that is wired to a Cognito authorizer.

Users were invited to the Pencil platform via email, which was a separate process that registered their accounts in Cognito. At each user's first login, Pencil would automatically add them to the appropriate user group in Cognito,

which would give them access to the user portion of the Pencil API.

That's a little confusing to reason about — it's enough to say that from a user perspective, they clicked a link to sign up to send text messages. After a short onboarding process with the campaign digital field-organizing team, a user was all set up and able to get to work immediately.

It was very easy to do and worked great because it was built on top of the same Turbotots infrastructure we'd already developed.

Tots

Tots represents a different slice of the architecture than Turbotots, but was something of a preceding reference implementation for what Turbotots eventually became. T
Tots sat on a path directly downstream from Turbotots. Tots parsed a number of the services listed beneath Turbotots on prior figures, and those are a part of the Tots architecture.

Tots was an important platform. As we built more tools that leveraged machine learning, we quickly realized that we needed to consolidate logic and dry up the architecture.
Our biggest use case for machine learning in various projects was extracting entity embeddings from blocks of text. We were able to organize that data in an index in Elastic and make entire subjects of data available for rapid retrieval. We used AWS Comprehend to extract the entity embeddings from text. It's a great service — give it a block of text and it'll tell you about the people, places, subjects, and so forth discussed in that text. The texts came into the platform in a variety of forms, including multimedia, news articles, and document formats.

Much of the mechanics in the Tots platform in figure 6 involved figuring out what to do with something before sending it to Comprehend. The process codified in this platform saved the campaign quite literally tens of thousands of hours of human work on organizing and understanding these various documents. This includes time spent transcribing live events like debates and getting them into a text format that could be read later on by people on the campaign.

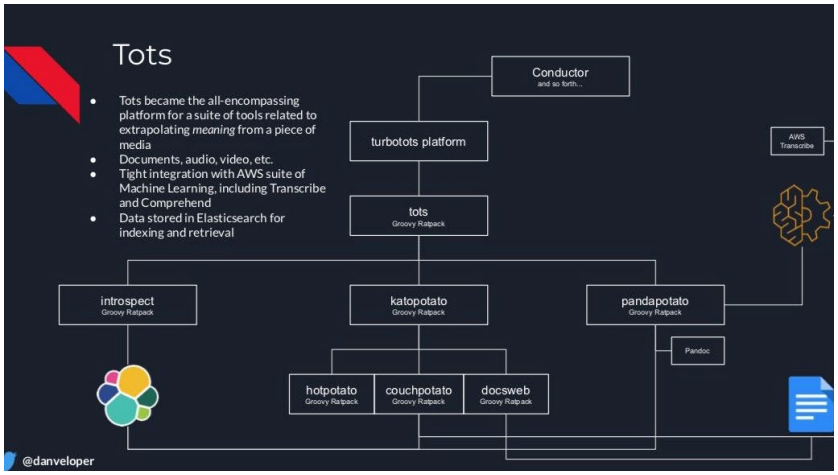


Figure 6: The Tots platform

CouchPotato

CouchPotato helped us to solve the hardest problem in computer science: making audio work on Linux. CouchPotato became very important to us because of the sheer amount of time it saved on working with audio and video material. It was also one of the most technically robust platforms that we built.

CouchPotato is the main actor in this architecture in figure 7 but the lines connecting the many independent services illustrate how it needed a lot of supporting characters along the way.

CouchPotato's main operating function was to take a URL or media file as input, open that media in an isolated X11 Virtual Frame Buffer, listen to the playback on a PulseAudio loopback device (FFmpeg), and then record the contents from the X11 session. This produced an MP3 file, which it then sent to AWS Transcribe for automated speech recognition (ASR).

Once the ASR process was completed, it would go through the transcript to correct some

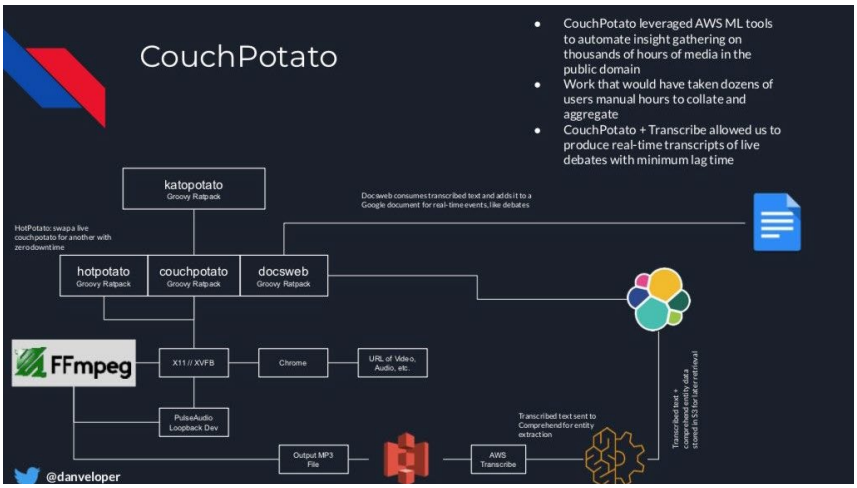


Figure 7: The campaign used CouchPotato to handle multimedia

common mistakes. For example, it could rarely get Mayor Pete's name (Pete Buttigieg) correct. We had a RegEx in place for some common replacements.

Once all that was done, the transcription would be shipped to Comprehend, which would extract the entity embeddings. Finally, the text would be indexed in Elasticsearch.
The real key to CouchPotato is that it was able to make use of FFmpeg's segmentation feature to produce smaller chunks of audio for analysis. It would run the whole process for each segment, and make sure that they all remained in uniform alignment as they were indexed in Elasticsearch.

The segmentation was one of CouchPotato's initial features, because we used it to transcribe debates in real time. Normally, a campaign would have an army of interns watching the debate and typing out what was being said. We didn't have an army of interns to do that, and so CouchPotato was born.

However, the segmentation created a real-world serialization problem. Sometimes Transcribe would finish a later segment's ASR before the prior segment had completed — meaning all the work needed to be done asynchronously and recompiled in the proper order after the processing. It sounds like a reactive programming problem to me.
We devoted a lot of time to solving the problem of serializing asynchronous events across stateless workers. That was complicated but we made it work.

That's also where DocsWeb came in handy for the debate. DocsWeb connected CouchPotato's transcription output for the segments to a shared Google Doc, allowing us to share with the rest of the campaign the transcribed text nearly in real time — except for the lag from Transcribe to Elastic, which wasn't too bad. We transcribed every debate and loads of other media that would have taken a fleet of interns a literal human lifetime to finish.

Part of solving the serialization problem for the segments was figuring out how we could swap a set of active CouchPotato instances for a new one — say, during a deploy or when a live Transcribe event was already running. There's a lot more to say on that subject alone, like how we made audio frame-stitching work so that we could phase out an old instance and phase in a new one. It's too much to go into just for this presentation. Just know that HotPotato a lot of work that allowed us to do hot deployments of CouchPotato with zero deployments and zero state loss.

KatoPotato was the entry point into the whole architecture. It's an orchestration engine, and it coordinated the movement of data between CouchPotato, DocsWeb, and Elastic, as well as acting as the main API for kicking off CouchPotato workloads. Furthermore, it was responsible for monitoring the state of CouchPotato and deciding if HotPotato needed to step in with a new instance.

Sometimes audio on Linux can be finicky. CouchPotato was able to report to KatoPotato whether it was actually capturing audio or not. If it was not capturing audio, KatoPotato could scale up a new

working instance of CouchPotato and swap it out via HotPotato.

The marketing line for CouchPotato is it's a platform that built on cloud machine learning that gave us rapid insight into the content of multimedia without having to spend valuable human time on such a task. It's a great piece of engineering, and I'm glad it was able to deliver the value it did.

Conclusion

There's so much more I could add about the technology we built, like the unmatched relational organizing app, the data pipelines we created to connect S3 and RDS to Redshift, or the dozens of microsites we built along the way. I could also explain how we managed to work collaboratively in an extremely fast-paced and dynamic environment.

I've covered some of the most interesting architectures we built for the Biden for President 2020 campaign. It's been a pleasure to share this and I look forward to sharing more in the future.

SPONSORED ARTICLE

What is observability and why should you care?

I've been asked several times about the difference between the terms 'observability' and 'monitoring', and why someone might prefer one approach vs. the other.

Since this is a bit of a false dichotomy, and the (many) articles written on this subject appear to be rather vague, I wanted to write up my thoughts and add some clarity.

Observability and monitoring do not have specific technical meanings and they delineate different eras in history. Additionally, corporate marketing has naturally muddied the water of their definitions. But don't worry about this.

Familiarize yourself with the new technology and techniques that are part of the movement to contextualize and standardize data as a way to improve quality, which the term 'observability' has become associated with.

The OpenTelemetry project has become the focal point for making these improvements a reality.

Control theory and other red herrings

If you look up the term 'observability', you will find a Wikipedia article focused on control theory. This is a concrete, mathematical definition of observability. It is also irrelevant. Observability in our context has no direct relation to control theory. Even if some of the words look similar, the similarity is superficial.

The V8 Javascript engine has nothing to do with cars – unless your car is running Javascript. Observing internet applications has nothing to do with control theory – unless you are applying control theory to the design of your internet application. Which I assure you, you are not. Nor should you. (You also shouldn't let your car run Javascript.)

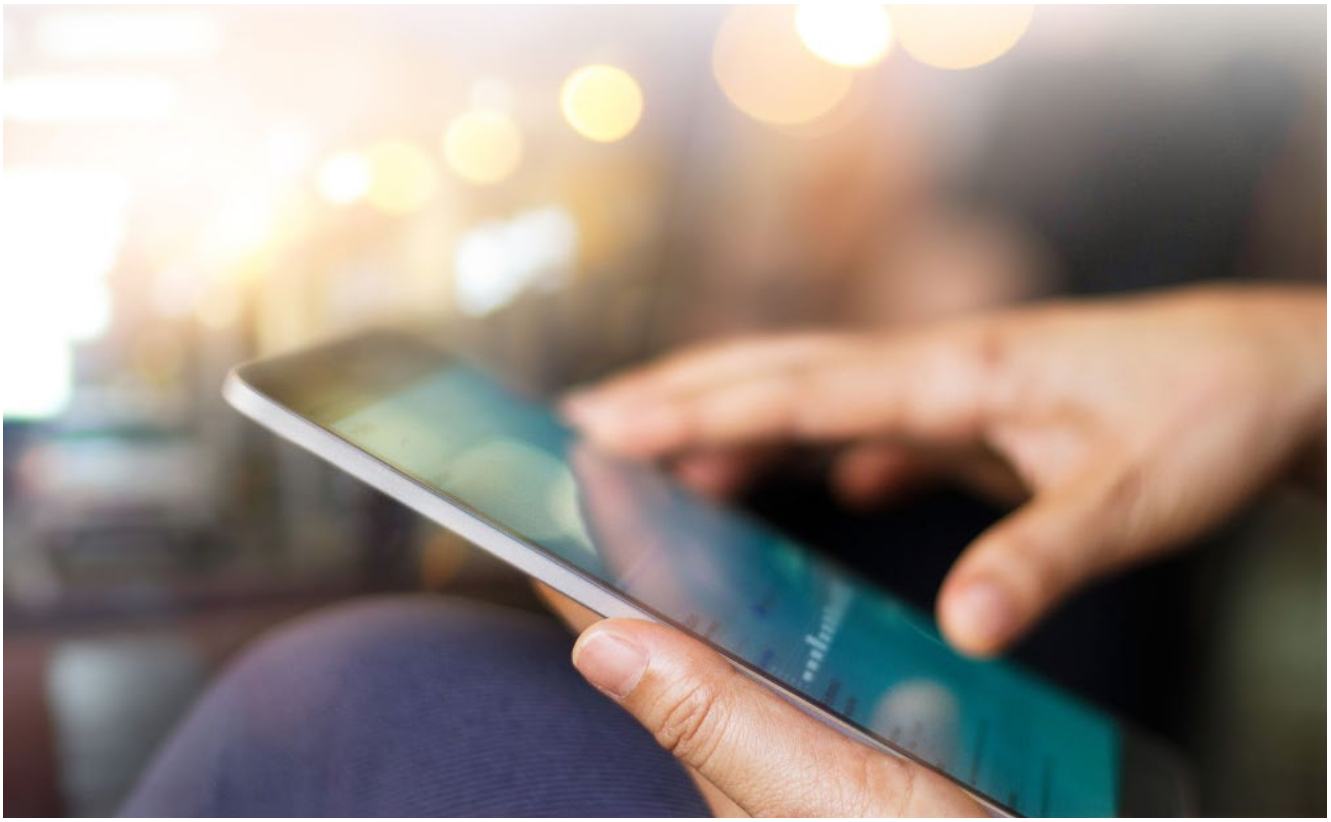
Why observability, and why now?

The observability movement is about saving time – saving so much time that it changes how you approach the problem.

If you've been operating systems for a while, and fought a number of fires, ask yourself: how much time, on average, do you spend collecting, correlating, and cleaning data before you can analyze it? Do you choose to limit your investigations due to the effort in collecting data? Has it gotten worse as your system has grown to include more instances of every component and have more components involved in every transaction? These are the issues that have pushed distributed tracing and a unified approach to observability into the limelight.

- In the end, it always comes down to this:
- Receive an alert that something changed;
- Look at a graph that went squiggly;
- Scroll up and down looking for other graphs that went squiggly at the same time – possibly putting a ruler or piece of paper up to the screen to help;
- Make a guess;
- Dig through a huge pile of logs, hoping to confirm your guess;
- Make another guess.

Read the full-length article here



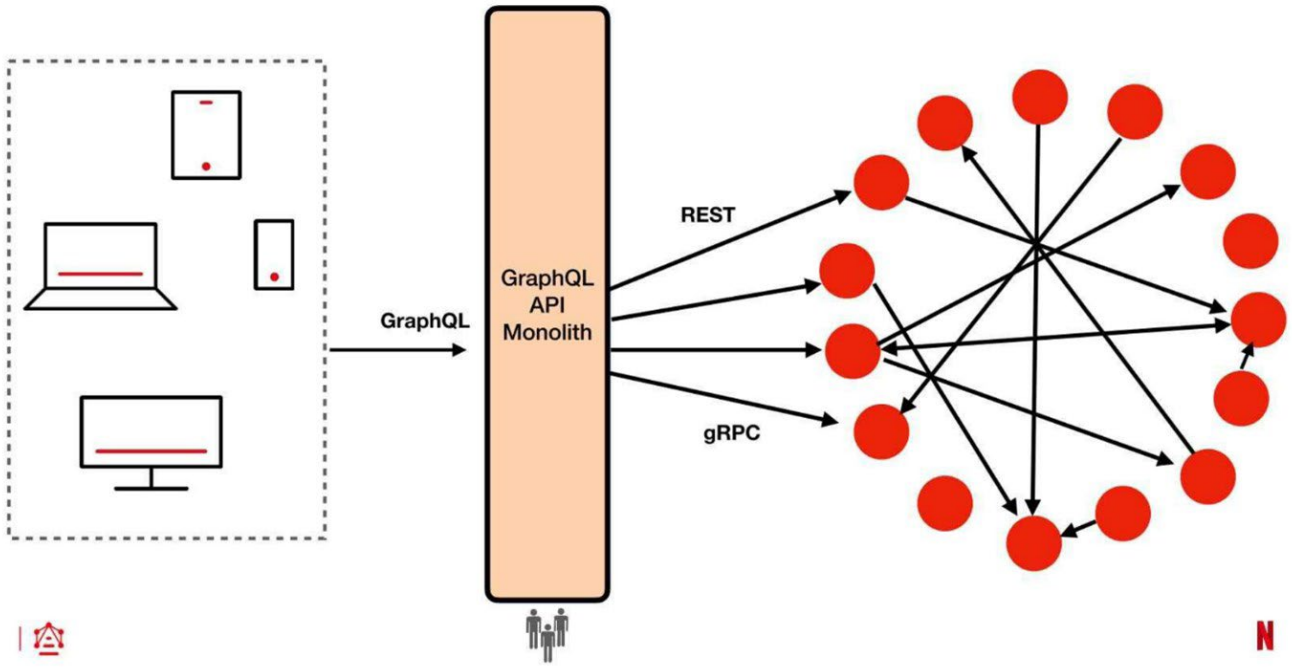
Using DevEx to Accelerate GraphQL Federation Adoption @Netflix

by **Paul Bakker** and **Kavitha Srinivasan,** senior software engineers at Netflix

At Q2020, Paul Bakker and Kavitha Srinivasan, both senior software engineers at Netflix, presented how they made specific Build vs. Buy (open source) trade-offs and the involving socio-technical aspects of working with many teams on a single shared schema.

In the space of show production, there's a lot of people involved in producing a show – and Netflix produces many shows. To support these efforts, we have many in-house built apps to keep track of all the things. At one point in time, the architecture we used for this was that we had many different microservices, each responsible for part of the data. These microservices would typically expose their data using a gRPC endpoint, and in some cases, REST. From the perspective of devices, we wanted to have one single API, specifically a single GraphQL API with a single schema. The way to accomplish that is that we added a GraphQL server API in front of those microservices.

It is an architecture that many companies use, and that is generally something that works out very well. Also, for us, it worked out quite well for quite a while. At some point, we did



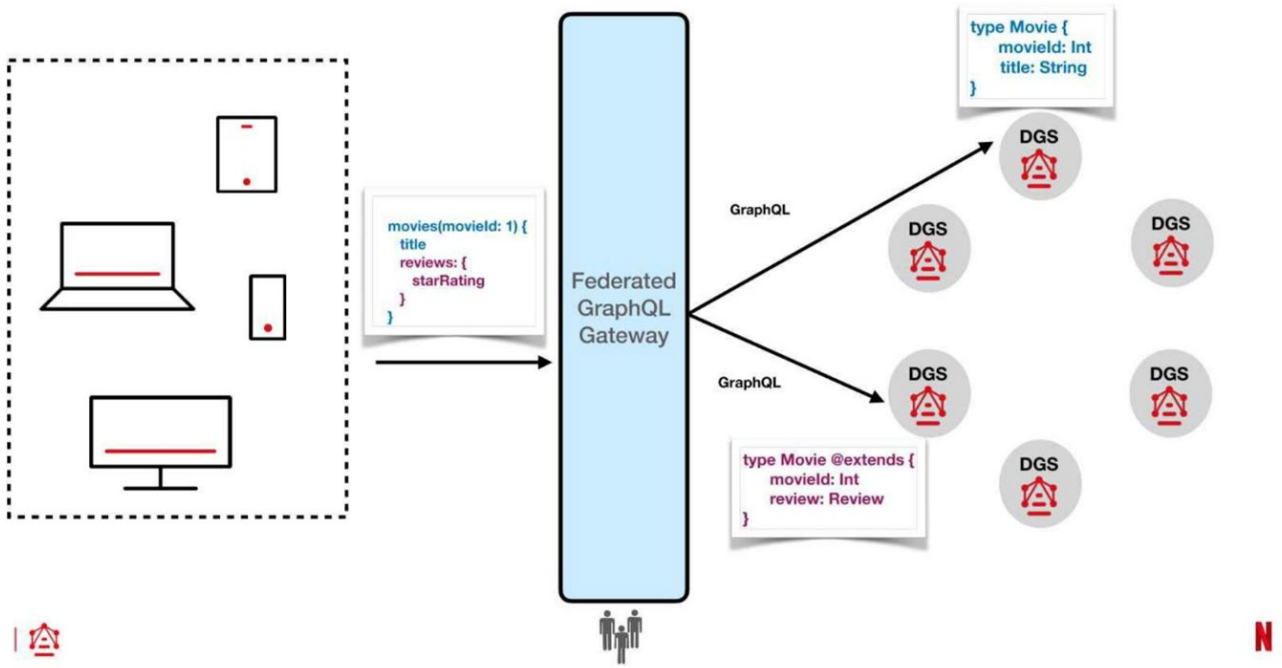
start to see limitations in this architecture. It had a lot to do with the scale we were doing things in and the number of teams involved. Every change made to these back-end services, like adding new data or fields to the schema, requires a code change from the team managing the GraphQL server API because they had to add code to do a new gRPC call to get the data. Then add it to the schema to be exposed in the GraphQL schema for the devices. The fact that this team had to do that manual work meant that they were a bit of a blocker, basically, for all the changes made. Just because of the sheer number of things involved, that just became a bottleneck by itself.

Another problem with this approach is that the device teams, the user interface developers, and the back-end teams responsible for the data are disconnected from each other - because we have this team sitting in the middle that actually takes care of the GraphQL schema. So, we didn't have a lot of natural collaboration between the device and back-end teams to really collaborate on a schema that works best for everyone.

Apollo Federation

Right along this time, Apollo came up with a Federation spec. It described how we can split a single shared graph across many different services, each owning a portion of the graph. We call these Domain Graph Services, or DGSs, in this architecture. For example, you have a type, movie that's owned by a movie DGS. It defines the movie ID and title field. Now you have another service called the review service that wants to extend this movie type by adding another field called reviews and providing the data. You can have an incoming GraphQL query that goes to the gateway. The gateway is configured, so it knows it needs to talk to the movies' DGS and the reviews' DGS to fulfill the query, get the data, and send the response back. That's how the Federation works at a very high level.

This solved the big problem with the bottleneck for us because now the gateway has no business logic. All it does is route requests and reach out to the appropriate services. All the schema collaboration is now being done by the back-end and the front-end teams, and any related changes. The gateway is entirely oblivious to this process.



How to Onboard 30 Teams on New technology

With this new architecture in mind, we envisioned increasing the rate of change that we could make and increasing the rate of innovation that we could go after. However, this new architecture also meant new challenges, and we introduced a completely new problem. Because of the back-end service teams that we had, and at the time, we had about 30 teams involved, they only had to expose a gRPC endpoint. It's something they were already very familiar with doing. They weren't too concerned about a GraphQL schema or how the data would look like in a GraphQL endpoint. We weren't worried about GraphQL in the first place. It wasn't really something they needed to think about because another team was taking care of that. With this new architecture, we did require those teams to be all on-board this new technology and this new architecture. They would now be responsible for owning part of the schema, which means the schema design that comes with it, and be responsible for writing the code to implement a GraphQL endpoint. On top of that, GraphQL was a very new technology when we look at back-end development because we didn't have a lot of GraphQL going on in the company in back-end microservices at the time. So the question that we were facing is, how do we get 30 teams excited about onboarding all this new technology and this new architecture, while it does imply a lot of extra, new work for them?

When You Give a Developer Carrots

The answer to that is providing developers with lots of incentives by way of a great developer experience. You want your developers to be happy, and you want to spark developer joy. How do you go about doing that? First and foremost is the ease of onboarding. In our case, we had a mix of developers who were already familiar with GraphQL and many who were not. Added to that, you have Federation, which is an entirely new concept to introduce. Finally, you want your users to focus on the business logic rather than the wire up and the setup details.

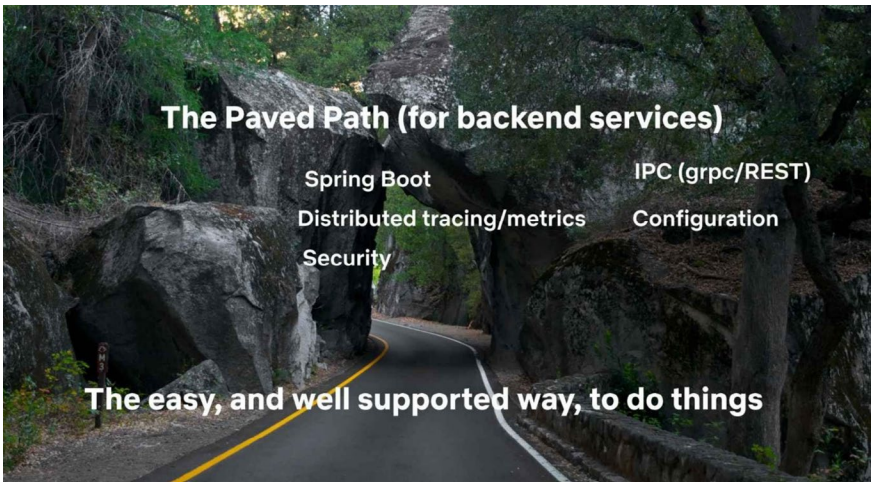
Next, you want to give your users a familiar ecosystem to work with. At Netflix, we have a slew of tools for debugging, tracing, and logging. We wanted to ensure we provide tight integration so users can use the tools they are already familiar with during this migration. The third aspect

is consistency. You want to try to make all the code look the same. You can easily enforce best practices as a result. It also makes it much easier to debug and share what you've learned with other teams. This also fosters more collective learning for patterns, and you're not reinventing the wheel so much. Finally, you want to build a strong sense of community, making the journey much more exciting for everyone involved.

The Paved Path for Back-end Services

Looking at how back-end development is typically done at Netflix, we have what we call the paved path. The paved path is the easiest and best-reported way to get things done. For example, looking at the picture below, it might be a bit more exciting and more fun to go off the road, climb these boulders, and find your way around them.

It's definitely more efficient to just follow the paved path and get to your destination. Very similar, we have the paved path for back-end development. Back-end development at Netflix is standardized on Java.

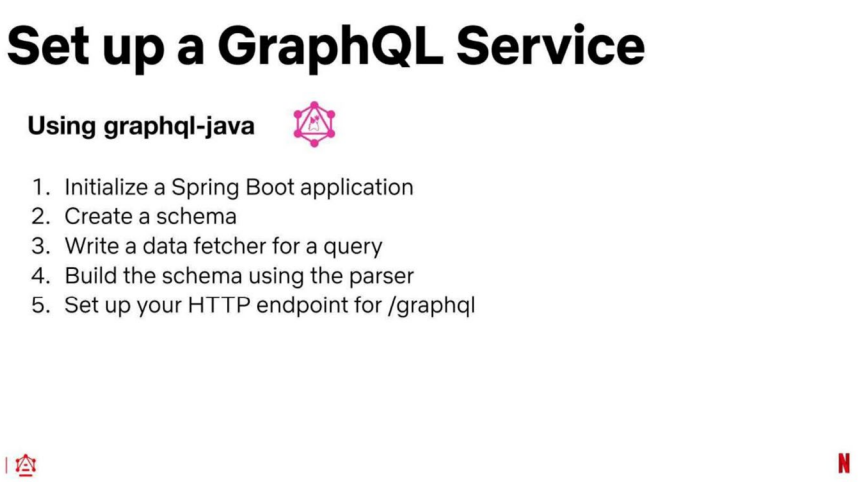


We use Spring Boot. On top of Spring Boot, we have built many integrations such as distributed tracing and distributed metrics and logging. We have integrations with security systems to do authorization and authentication. We have a host of different IPC clients to do gRPC and REST. Also, server implementations to do gRPC and REST. Note that there is no GraphQL here because GraphQL really wasn't something we were using a lot within the company when looking at back-end server development.

Set up a GraphQL Service

How do you go about setting up a GraphQL service? First of all, you want to initialize your Spring Boot app. Then you have to create a schema, which is the API you wish your service to expose. Then you write a data fetcher to return data in response to requests. Then you need to build your schema for the parser. Then lastly, you want to set up your HTTP endpoint to respond to GraphQL queries.

Let's look at an example and see how that actually would look like in code. The example that we will look at is implementing a schema. It's a straightforward schema. We have a query type defined, and on the query, we have one field that we can query for, which is hello. It has enabled arguments, and it will return as a message, as a string. If you want to implement this in Spring Boot using GraphQL Java, the



code looks like this. It's a working example, and it's quite a bit of code, so we are not going to look at the details of this code. The critical fact that I want to point out here is that there's only a single line of business logic in the picture below.

All the other code is just setup (boilerplate) code. Obviously, this would be code that every team would have to refigure out and write. An important detail here is that although this code is a working example, it is really a naive implementation. It's really not thinking about error handling. It is really just taking care of the more simplistic happy path. Doing this in a more production-ready way, there's quite a bit of code to write. Again, it will be code that every team would have to reinvent and get writes if it would not provide anything else.

You have the happy path, but now you want to add authorization. You want to add tracing, logging, metrics. You want to add error handling, custom exception handling, and so on. This is just repeated boilerplate code that users can easily do away with and not have to write each time in each of their services.

The Domain Graph Service Framework (DGS)

Very clearly, we could do much better here. Early in this architectural migration, we decided to invest in a Domain Graph Service framework or DGS. The DGS framework is really GraphQL integration for Spring Boot Netflix. The Netflix part in that is also essential. It's not just about general Spring Boot usage, but it is also about integrating with all the different components that we already have at Netflix. This framework uses GraphQL Java internally. GraphQL Java is a pre-built, maintained open-source library that takes care of the lower-level details for doing GraphQL in Java.

Let's take a look at how the DGS framework looks like. Using the DGS framework, the same code example that we have seen previously would just look like this code. You immediately notice it's a lot less code and looks a lot more declarative. In a familiar Spring Boot fashion, we use annotations for a lot of things. We use @DgsComponent to mark the component as being a class for the framework. We use @DgsData to say that this method provides a data fetcher for a specific field in our schema. In this example, it's the hello field that we had specified. We integrate things like @Secured to get authentication or authorization on a field level out of the box, integrating with the Netflix implementation of these security concerns. What's left is really just a business logic that the developer should be concerned about. Then there are all sorts of conveniences to, for example, quickly get input arguments using this @InputArgument annotation. It really makes your code really nice and concise, and most of all, that's really focused.

A huge benefit that we get from this is that there's a lot of consistency between codebases. There were initially about 30 teams that needed to be onboarded on this new architecture. We have less than a handful of developers in the developer experience team. That's the team helping out all these teams, and help them whenever problems come up,

```java
@RestController
public class GraphQLEndpoint {

    @Autowired
    ResourceLoader resourceLoader;

    private GraphQLSchema graphQLSchema;

    @PostConstruct
    public void setupSchema() {
        SchemaParser schemaParser = new SchemaParser();
        File schemaFile = loadSchema();

        TypeDefinitionRegistry typeDefinitionRegistry = schemaParser.parse(schemaFile);

        RuntimeWiring runtimeWiring = newRuntimeWiring()
                .type("Query", builder -> builder.dataFetcher("hello",
                        (DataFetchingEnvironment dfe) -> "Hello, " + dfe.getArgumentOrDefault("name", "Stranger")))
                .build();

        SchemaGenerator schemaGenerator = new SchemaGenerator();
        graphQLSchema = schemaGenerator.makeExecutableSchema(typeDefinitionRegistry, runtimeWiring);
    }

    @PostMapping("/graphql")
    public String query(@RequestBody String body) throws JsonProcessingException {
        GraphQL build = GraphQL.newGraphQL(graphQLSchema).build();

        ObjectMapper mapper = new ObjectMapper();
        Map<String, Object> parsedBody = mapper.readValue(body, new TypeReference<>() {
        });

        ExecutionInput executionInput = ExecutionInput.newExecutionInput()
                .query((String) parsedBody.get("query"))
                .variables((Map<String, Object>) parsedBody.get("variables"))
                .build();

        ExecutionResult executionResult = build.execute(executionInput);
        return mapper.writeValueAsString(executionResult);
    }

    private File loadSchema() {
        try {
            return resourceLoader.getResource("classpath:schema.graphqls").getFile();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
```

AuthZ

```java
@Component
public class AuthzFilter implements Filter {
    @Override
    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain) {
        //Authz code
        //But how to wire this to field level!?

        chain.doFilter(request, response);
    }
}
```

Error handling

```java
public class CustomExceptionHandler implements DataFetcherExceptionHandler {
    @Override
    public DataFetcherExceptionHandlerResult
onException(DataFetcherExceptionHandlerParameters handlerParameters) {
        if(handlerParameters.getException() instanceof AccessDeniedException)

            return DataFetcherExceptionHandlerResult.newResult()
                    .error(GraphqlErrorBuilder.newError()
                    .message("User doesn't have access to field").build())
                    .build();
    }

    // ...
}
```

Tracing

```java
public class TracingInstrumentation extends SimpleInstrumentation {
    @Override
    public DataFetcher<?> instrumentDataFetcher(
     DataFetcher<?> dataFetcher,
     InstrumentationFieldFetchParameters parameters) {
        // Write traces to distributed tracing system

        return dataFetcher;
    }
}
```

```java
@DgsComponent
public class HelloDataFetcher {

  @DgsData(parentType = "Query", field = “hello")
  @Secured("my-user-group")
  public String hello(@InputArgument("name") String name) {
      return "hello, " + name;
  }
}
```

```
@Test
void helloShouldIncludeName() {
    String message = queryExecutor.executeAndExtractJsonPath(
            "{ hello(name: \"DGS\")}","data.hello");
    assertThat(message).isEqualTo("hello, DGS!");
}
```

and getting things going. With that ratio, it is essential for us that if a developer comes to us with a question on, for example, our select channel, that we can go into their code repository and quickly understand how it's all fitting together and really focus on the code that might be problematic. We get that because of this consistency, because all our codebases are structured in a very similar way, we can really quickly point out the code that we actually need to look at.

Another benefit that we're not seeing in this code but that we are getting for free is that we get a lot of integrations with existing Netflix systems. For example, this code example integrates with distributed logging, distributed tracing, and distributed metrics. There's no setup required at all for a developer. There's all those working out of the box.

Testing

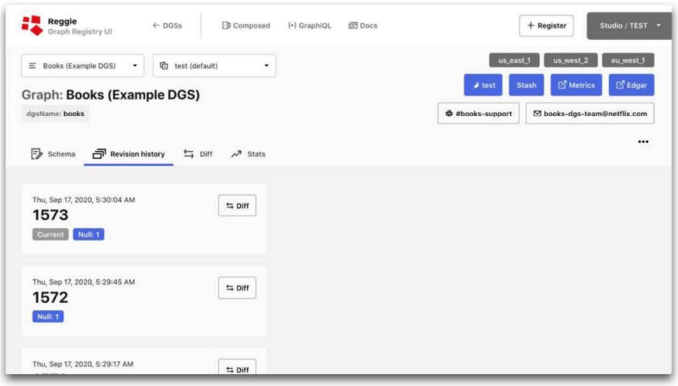
Next, let's talk about testing. Testing can have a significant impact on developer velocity.

We provided an excellent, easy way to eliminate any setup code and just focus on the business logic you want to test with the framework. In this case, the GraphQL query. The rest of the test setup purely involves any validation steps you wish to add. On top of this, we also make it really easy to construct your GraphQL query string. We will not discuss it here for the sake of simplicity, but we provide an excellent type-safe way to construct your GraphQL query string, as opposed to manually handcrafting it, which can be pretty painful. Not only this, this just talks about testing your local DGS. This has to operate in a federated system, which has many moving parts. We also offer our developers many easy ways to test manually, end-to-end, their queries in a federated setup.

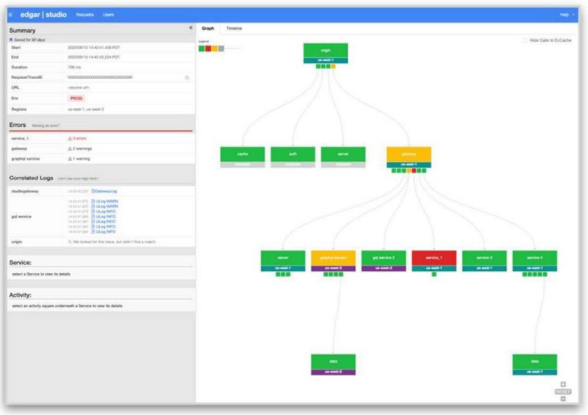
Schema Management

These are just some examples of what we did at the coding level, the way developers run their code. Furthermore, being in a federated architecture, there are many more parts that are required. One of the things to really think about is schema management because each team is managing their own part of the schema, analyzed to come together. Looking at the broader GraphQL community outside of Netflix, Apollo has some excellent tooling that helps with schema

Schema Management



Tracing & Logging



management. We actively collaborate with Apollo on both the Federation specification and the tooling around this to make the developer's life easier.

However, because we have so many existing systems within Netflix and wanted to deeply integrate with these systems, we decided to build this tooling ourselves. What you get with this tooling is really the one-stop-shop for everything around your DGS and your schema. You get schema efficiency. You get statistics on how your schema and the fields in your schema are used and used by who. There are integrations with build pipelines, distributed tracing, metrics, logging, and all these things. This really becomes the one place to find all your information and manage your schema and your DGS.

Tracing and Logging

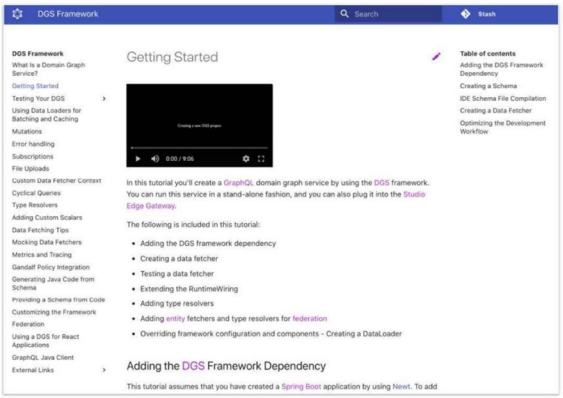
Then comes tracing and logging in this distributed architecture. As you can see below, we have our tool here. This is a custom in-house tool that developers have already been using. With the DGS framework, we integrated with this existing tool and made this available for all developers who have migrated onto this new architecture. This tool shows the request fanout pattern.

As you can see, it comes in; the gateway is now forwarding it to many different DGSs, which in turn can go ahead and talk to other back-end services. All of this is captured in this view. It's all color-coded, so you can see yellows for warnings and red for errors. It also has the correlated request logs on the left-hand panel. We're able to get correlated logs because, in the DGS framework, we ship all the logs to Elasticsearch clusters, and we're able to view all the relevant pieces of information in just this one view. Edgar also allows us to drill down and analyze performance. At every stage of the processing of the request, you can see how much time it took, all the way down to your data fetchers.

Documentation

Another critical aspect of developer experience, which so many of you spend a lot of energy on, is getting really great documentation. We have about 30 teams as part of this migration, and you have less than a handful of developer experienced developers. How can we scale supporting that many teams and developers?

Documentation



Documentation is really crucial there, we believe. It is really critical that developers can quickly get started themselves, can learn about the more advanced use cases by themselves, and for the most part, find solutions to problems they might run into themselves as well. So the documentation is just really the essential part there.

Learning – Which Path to Pave?

So far, we talked about the DGS framework that we built and the several tools that we developed to make a smooth developer experience. For the remainder of the talk, we'll focus on the learnings from this journey. Starting off with, which experience do you want to build out or which path do you want to pave? In our case, we started really small with a very narrow set of users and a minimal use case.

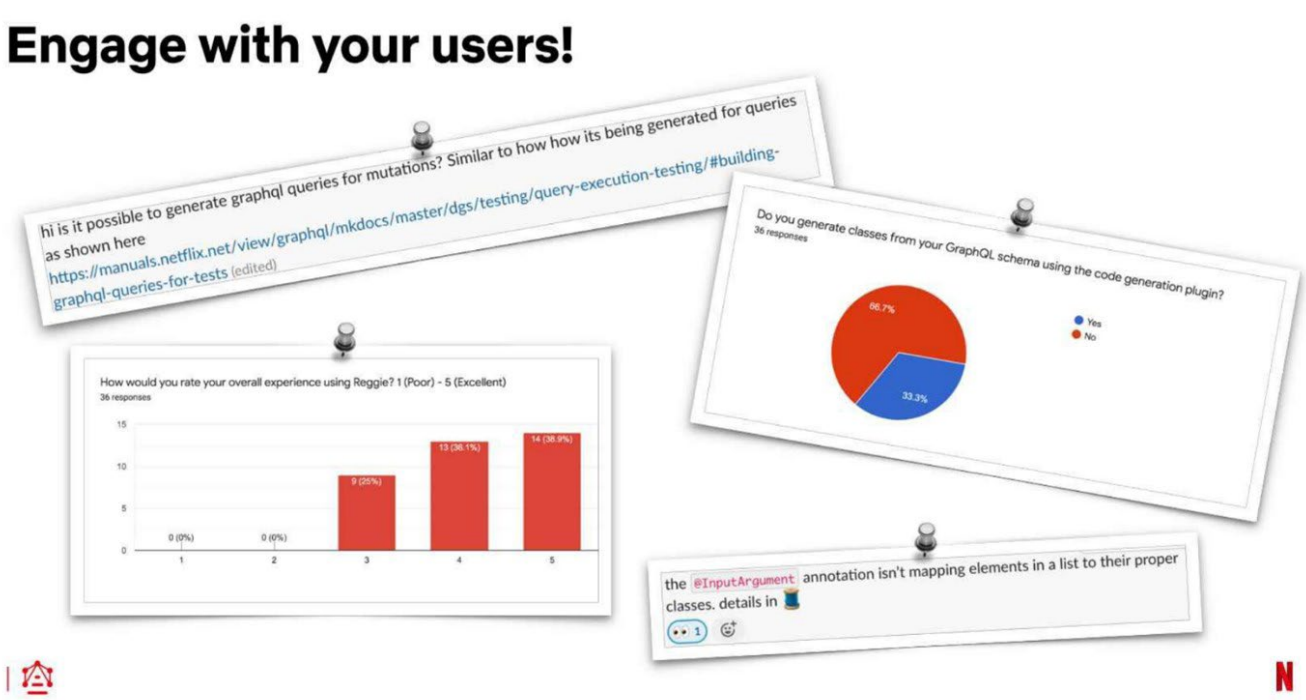
To begin with, all we wanted to do was reduce boilerplate code. Then, starting with GraphQL Java, we wanted to add more features to it, but make sure that we don't take away anything. To do that, we needed to provide escape hatches at the right places, so developers still have access to all the low-level features in GraphQL Java in the event that we didn't already support it in the framework, which was in the initial phases.

Then, as we moved along and the adoption started growing, we heard about more feature requests, and we incorporated a lot more features along the way. So now, gradually, we're up to a point where we have almost 30-plus teams who have onboarded onto this new architecture, all using our framework and tools. Not only that, but we also have users that have nothing to do with the Federation. They're not part of this migration, but they're also using our tools and the framework to just be able to quickly start up a GraphQL service.

Especially early on in this migration, we had to figure out what to show for and what features to add exactly, because the way this works is that the teams actually implementing these DGSs, started at the same time, as we as the developer experience team started talking about, maybe we should do a framework. We should have built all these tools. So prioritization was crucial, what to do first. That's early on with the teams that are actually doing the implementation work. The developer experience folks embedded with these teams were part of basically the first DGSs built. That way, we got a profound understanding of what problems to solve, what needed smoothening out. It also constructed just developer empathy.

Later on in the process, we adopted this philosophy that we never say no. Of course, there's a lot of subtlety to that. Essentially, there are many ways to say yes. What we mean by this is that we always prioritize the users first. Then, of course, we have the more significant features we are working on and things we want to get out in the framework. Whenever a developer comes to us with a question, that can be a bug report, but it can also be a feature request, and in many cases, they have some idea how to improve their developer experience. The easy thing to do would be to create a Jira, put it on your backlog, and come back to it a few weeks later whenever you have time.

Instead of doing that, what we do is we basically try to interact with these users right away. That means coming up with a design, basically, of whatever we need to do together with them. Then go and implement these features when possible, just the same day or the next day. With having that concise feedback cycle, we get a lot of benefits. First of all, our users feel like you're actually listening to them. Not just listening to them, but they get immediate benefits from being a user of the framework because they have this idea on how they can improve their own work. So instead of having to put in the time to actually implement those features in the infrastructure, they talk to us, and they basically get it for free. Obviously, the benefit for us is that we get really good feedback, and we know what to



focus on because that's what our users need. This does slow down the work on more significant features a little bit, but that is a trade-off that worked out really well for us.

Engage With Your Users

Finally, you want to engage with your users continuously. We have a support channel on Slack, where we almost hear from our users daily, whether it be feature requests, requests for bug fixes, and sometimes even design discussions with a narrow focus set of users.

We also do surveys to glean general patterns of how our tools are being used and how effective they are across different types of users. We also found user interviews to be highly effective, regardless of where you are in this process. Just targeted, small, focused groups of users to learn about the workflows and general needs can go a long way in informing your priorities and even figuring out the next new experience to build out.

The Risk Of Reinventing the Wheel

This migration has been an enormous success story for us. That is a success story that has been going on for about a year now. We also want to give you a little bit of a word of warning. That is before you dive into building all these custom tooling and frameworks yourself, there is definitely a risk of trying to reinvent the wheel. So the first thing that you really have to think about is that, can I actually do a better job than what is available in open source or the market? Because, especially at Netflix, where we have an influential culture of freedom and responsibility, and it also stretches to technology choices, a team is free to choose a technology they think they can do the best job fit. This means, if they feel they can do a better job using some open-source tool instead of using the tooling we provide, they're free to do so. They will actually go out and do that. So we have to make sure that the users see the benefits of using the tooling we provide and are doing a better job than they would otherwise do because they have these things available.

We put a lot of extra time and thought in; how do we contribute back to the broader GraphQL community? How do we also learn from the wider GraphQL

community? Concretely, what we're doing is we're collaborating very closely with Apollo. This is collaboration on the Federation spec, but also lots of in-depth discussions about how tooling should work and how tooling can help make the developer experience better. Hopefully, this way, others can learn from us, but we're also learning how other companies are solving some problems that we might be facing. So definitely make sure that you give these things thought as well.

GraphQL Federation and DGS - A year later

It's been a year since we first spoke about our federated architecture and DGS. A lot has happened since!

First and foremost, the DGS Framework has been open sourced and has built a great community. The framework provides the DGS programming model to anyone using Spring Boot. We are seeing a lot of adoption of DGS in the industry, and this is helping to further improve the framework.

Netflix is fully using the OSS version, and only adds some plugins internally for things like authz, tracing and metrics.

GraphQL federation and DGS within Netflix has also grown explosively over the last year. Federation is now used in all parts of the business and we're running multiple federated graphs, all powered by the same stack. In the presentation we mentioned onboarding 30 teams initially.

We currently have 190 DGSs in production, and adoption is still increasing at an incredible pace!

GitHub's Journey from Monolith to Microservices

by **Sha Ma**, Vice President and Head of Engineering at Catalyst.io

This article explores GitHub's recent journey toward a microservices architecture. It takes a deeper look at GitHub's historical and current state, goes over some internal and external factors, and discusses practical consideration points regarding how we started to tackle this migration. Then we will walk through some key concepts and best practices of implementing a microservices architecture that you can apply as you think about this transformation for your organization.

The Beginning

GitHub was founded in 2008 as a way of making it easier for developers to host and share their code. The founders of GitHub were open-source contributors and influencers in the Ruby community. Because of that, GitHub's architecture is deeply rooted in Ruby on Rails. Throughout the company's history, we have employed some of the world's best Ruby developers to help us scale and optimize our code base.

Today, we have over 50 million developers on our platform, over 80 million pull requests merged per year, and over 100 million repositories across every continent of the world. As you can see, a monolithic architecture got us pretty far. A code base that's over 12 years old coordinated deploy trains that handle multiple deployments per day, a highly scaled platform serving over a billion API calls on a daily basis, and a fairly performant user interface that focuses on getting the job done.

Rapid Internal Growth

Internally, GitHub went through a significant growth phase in the last 18 months. With over 2000 employees, we have more than doubled the number of engineers contributing to our code base. We've grown both organically and through acquisitions, such as Semmle, npm, Dependabot, and Pull Panda.

Additionally, GitHub is a highly distributed team, with over 70% of our employees working outside of our San Francisco headquarters prior to the pandemic. GitHub employees and contractors collaborate across six continents and work in all time zones.

With over 1000 internal developers bringing a diverse set of skills and operating in a wide range of technologies, it's become clear that we need to fundamentally rethink how we do software development at GitHub. Having everyone learn Ruby before they can be productive, and having everyone doing development in the same monolithic code base is no longer the most efficient and optimal way to scale GitHub.

According to Conway's Law, any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure. This also applies in reverse - having a monolithic environment will lead to bigger stakeholder meetings, and more complicated decision-making processes because of interwoven logic and shared data that impacts all the teams.

Monolith vs. Microservices

This got us thinking, is it finally time to start migrating out of the Ruby on Rails monolith towards a microservices architecture? If so, how should we go about doing it?

Both monolithic and microservices architectures have their advantages.

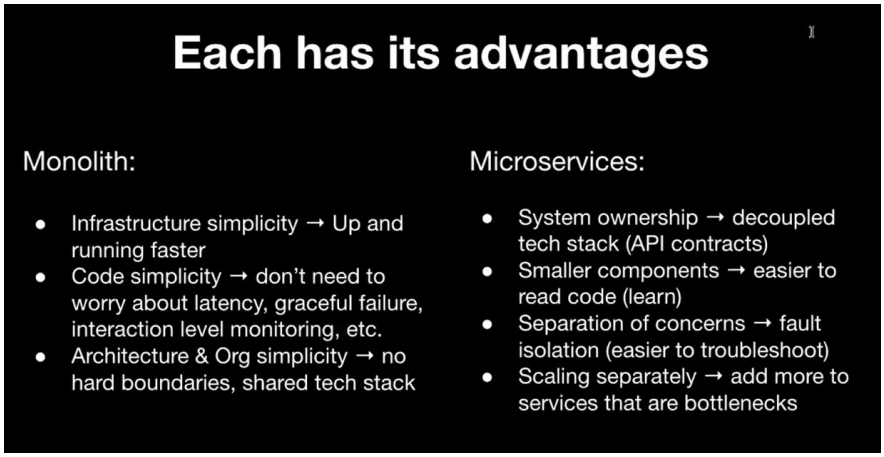


Figure 1

In a monolithic environment, it's easier to get up and running faster, without having to worry about complex dependencies and pulling in all the right pieces. A new Hubber can get GitHub up and running on their local machine within hours.

There is some code-level simplicity in a monolith as well. For example, you don't have to add extra logic to deal with timeouts or worry about failing gracefully due to network latency and outages.

Additionally, because everyone is working in a shared tech stack and has familiarity with the same code base, it's easier to move people and teams around to work on different features within the monolith and push towards a more global prioritization of features.

Because of the way GitHub has grown in the last 18 months, some of the advantages of a microservices environment are starting to look really appealing to us. For example, setting up feature teams with system-level ownerships and having functional boundaries through clearly defined API contracts. The teams have a lot of freedom to choose the tech stack that makes the most sense for them, as long as the API contracts are followed.

Smaller services also mean easier to read code, quicker ramp-up time, and easier troubleshooting within that code base. A developer no longer has to understand all the inner workings of a large monolithic code base to be productive. Most importantly, services can now be scaled separately based on their individual needs.

Be Pragmatic - It's About Enablement

Before we jumped into this transition at GitHub, we spent some time thinking about the why behind our decision and our goals for making this change. It's a huge shift for us from a cultural perspective and requires a lot of work.

We need to be intentional and think about what problems and pain points we're trying to solve. At GitHub, we're doing this so we can enable over half of our developer base, who joined us in the last 18 months, to be productive outside of the monolith.

The goal for us is enablement and not replacement. Because of that, we need to accept the fact that at GitHub, for the foreseeable future, we will be a hybrid monolith-microservices environment, which means it's still very important for us to maintain and improve the existing code base inside the monolith. A good example of this is our recent upgrade to Ruby 2.7. You can read more about what we did and how it made our overall systems better on the GitHub blog.

Good Architecture Starts With Modularity

Good architecture starts with modularity. The first step towards breaking up a monolith is to think about the separation of code and data based on feature functionalities. This can be done within the monolith before physically separating them in a microservices environment.

It is generally a good architectural practice to make the code base more manageable. Start with the data and pay close attention to how they're being accessed. Make sure each service owns and controls access to its own data, and that data access only happens through clearly defined API contracts.

I've seen a lot of cases where people start by pulling out the code logic but still rely on calls into a shared database inside the monolith. This often leads to a distributed monolith scenario where it ends up being the worst of both worlds - having to manage the complexities of microservices without any of the benefits. Benefits such as being able to quickly and independently deploy a subset of features into production.

Separating Data at GitHub

Getting data separation right is a cornerstone in migrating from a monolithic architecture to microservices. Let's take a closer look at how we approach this at GitHub (see **Figure 2**).

First, we identified the functional boundaries within existing database schemas and grouped the actual database tables along these boundaries. For example, we grouped everything related to repositories together, everything related to users together, and everything related to projects together.

These resulting functional groups are referred to as schema domains and are captured in a YAML definitions file. This is now our source of truth, and it is expected to be updated whenever tables are added or removed from our database schemas. We use a linter test to help remind developers to keep this file updated as they make those changes.

Next, we identified a partition key for each schema domain. This is a shared field that links all the information together for a functional group. For example, the repository schema domain, which holds all the data related to repos, (such as issues, pull requests, or review comments), uses repo ID as the partition key. Creating functional groups of

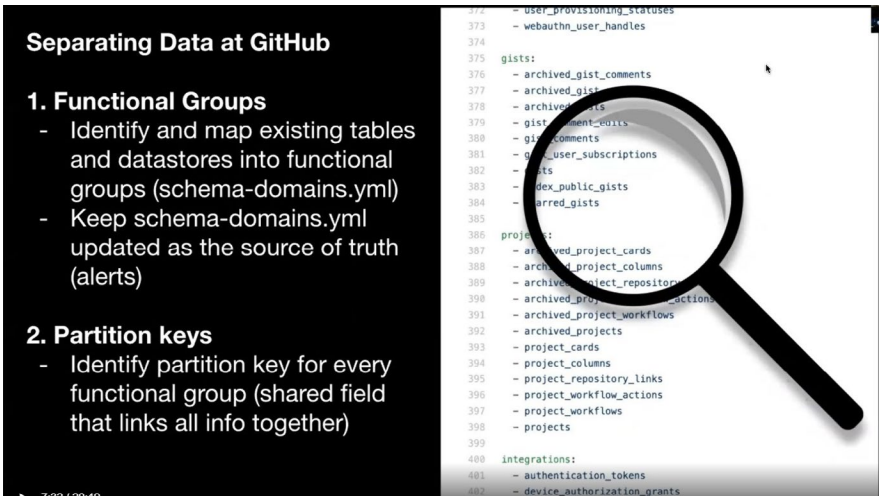


Figure 2

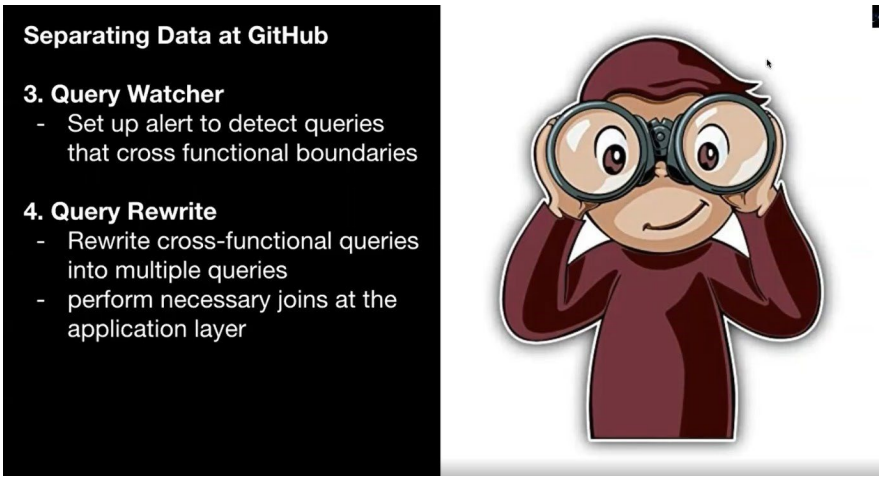


Figure 3

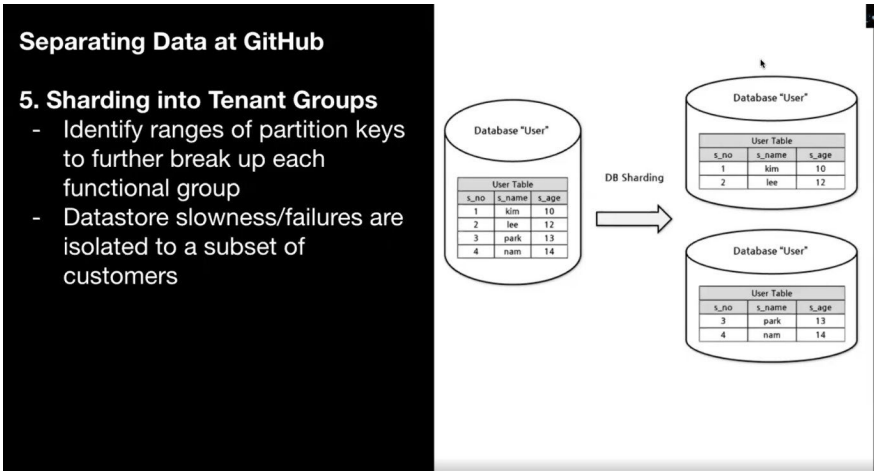


Figure 4

database schemas will eventually help us safely split the data onto different servers and clusters needed for a microservices architecture. First, we needed to fix current queries that go across domain boundaries, so that we don't end up breaking the product when data separation happens (see **Figure 3**).

At GitHub, we implemented a query watcher in the monolith to help detect and alert us any time a query crosses functional domains. We would then break up and rewrite these queries into multiple queries that respect the domain boundaries and perform any necessary joins at the application layer (see **Figure 4**).

Finally, after all the functional groups have been isolated, we can begin a similar process to further shard our data into tenant groups. With over 50 million users and 100 million repos, functional groups can grow pretty big at GitHub scale. This is where the partition keys come in handy. We can follow a similar process to identify ranges of partition keys to group together.

For example, an easy way is to simply assign different users to different datastores based on numeric ranges. There are probably more logical groupings based on the characteristics of each data set, such as region and size. Tenantizing is a great way to limit the blast radius of data storage failures to only a

subset of your customers versus impacting everyone all at once.

Start With Core Services and Shared Resources

After exploring the importance of data separation, let's switch gears and dive into how to lay the groundwork for extracting services out of the monolith. It's important to keep in mind that dependency direction should always go from inside of the monolith to outside of the monolith, and not the other way around, so we don't end up in that distributed monolith situation. This means when extracting services out of the monolith, start with the core services and work your way out to the feature level.

Next, look for gravitational pulls that keep developers working in the monolith. It's common for shared tooling to be built over time that makes development inside the monolith very convenient. For example, feature flags at GitHub provide monolith developers peace of mind for having control over who sees a new feature as it goes from staff shipped to beta to production. Make these shared resources available to developers outside of the monolith and start shifting that gravitational pull. Finally, make sure to remove old code paths once new services are up and running. Use a tool to understand who's calling this service and have a plan to move 100% of the traffic over to the new service, so you don't get stuck supporting two sets of code forever. At GitHub, we use an open-source tool called Scientist to help us with this type of rollout, where we can run and compare both the old and the new code paths side by side.

Extracting AuthN/AuthZ at GitHub

The core services that we decided to extract first at GitHub are authentication and authorization (see **Figure 5**).

Authentication is pretty complex because everything needs it.There's a ton of shared logic between the website and Git operations. This means that if github.com is down, then access to Git systems is also down, and Git operations, like pull and push, will no longer work even through a command-line interface. This is why it's so important for some of these fundamental pieces to be extracted to allow primary functions to still happen without having to be tied into the monolith.

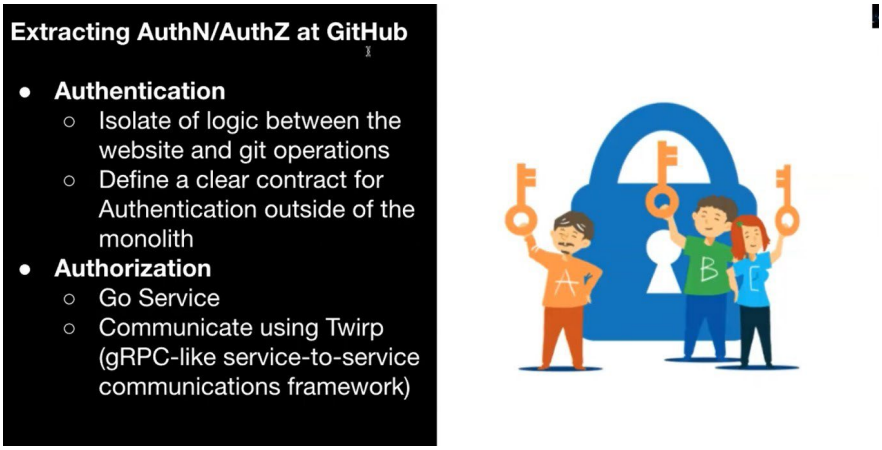


Figure 5

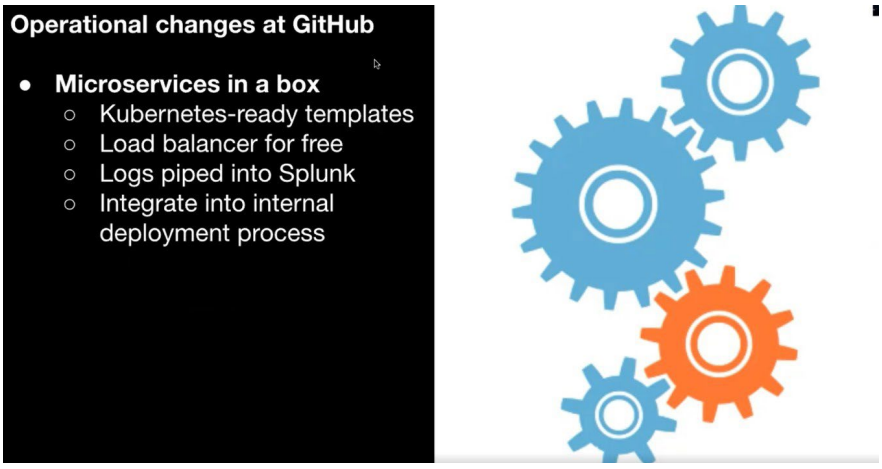


Figure 6

Authorization for us was much more straightforward and has already been rewritten as a ghost service outside of the monolith. The current Rails app, (aka our monolith), communicates to it using Twirp, which is a gRPC-like service-to-service communications framework, thus needing the inside to outside dependency direction.

Make Operational Changes

Monitoring, CI/CD, and containerization are not new concepts, but making the necessary operational changes to support the transformation from monolith to microservices can yield significant time savings and help expedite the transition towards microservices (see **Figure 6**).

Keep the main characteristics of microservices in mind when you make these workflow changes. Operationally supporting numerous, small, independently running services with diverse tech stacks is very different from running a single, highly customized pipeline for a large monolith. Update monitoring from functional call metrics to network metrics and contract interfaces. Push towards a more automated and reliable CI/CD pipeline that can be shared across services. Use containerization to support a variety of languages and tech stacks. Create workflow templates to enable reusability.

For example, at GitHub, we created a self-service runtime platform to deliver microservices in a box. The goal is to drastically reduce each team's operational overhead for creating microservices. It comes with Kubernetes-ready templates, free Ingress setup for load balancing, automatic piping of logs into Splunk, and integration into our internal deployment process. Thus, making it easier for any team that wants to experiment with or set up a new microservice to get started.

Start Small and Think About Product/Business Value

So far, we've covered a lot of ground on the structural changes and shared foundations needed for a successful transition from a monolith to a microservices architecture. From this point on, any new feature should be created as a microservice outside of the monolith. Next, look for a few, simple, minor features to move out of the monolith. For example, features that don't have a lot of complicated dependencies and shared logic. At GitHub, we started with webhook deliveries and syntax highlighting. Use this as an opportunity to look for common patterns and identify gaps before moving on to bigger and hairier functionalities in the monolith. Use product and business values to help determine the right size of microservices.

Look for code and data that are often changed and deployed together to determine features or functionalities that are more tightly coupled. Use these as your natural groupings for what can be iterated on and deployed independently from other areas. Focusing on product and business value also helps with the organizational alignment across engineering, product, and design. Keep in mind, breaking things up too small can often add unnecessary complexity and overhead. For example, maintaining separate deploy keys, more on-call responsibilities, and single points of failure due to the lack of shared knowledge.

Move Toward Asynchronicity and Code for Resiliency

Going from monolith to microservices is a major paradigm shift. Both the software development process and the actual code base will look significantly different going through this transition. To wrap up, we will quickly cover service-to-service communications and designing for failure, both of which are important concepts in microservices development.

There are two ways that services communicate with one another - synchronously and asynchronously. With synchronous communications, the client sends a request and waits for a response from the server. With asynchronous communications, the client sends

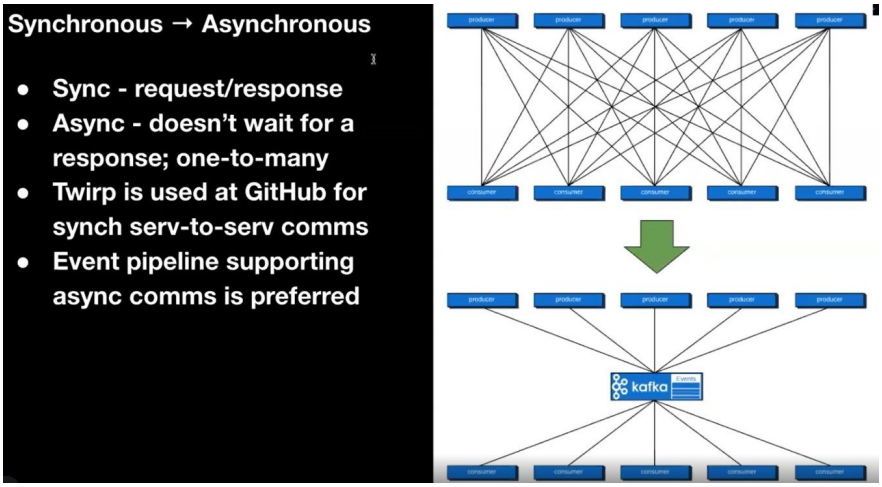


Figure 7

We use Twirp at GitHub to enable synchronous communications between the monolith and the core services outside of the monolith, like authorization.

As more services move outside of the monolith, however, synchronous communication starts to become wildly inefficient as the picture in the upper right demonstrates. It also creates tight coupling between all the services, which ends up defeating the purpose of moving to a microservices architecture. A better approach is to create a shared events pipeline that can broker messages across multiple producers and consumers. This is the architecture we used at SendGrid.

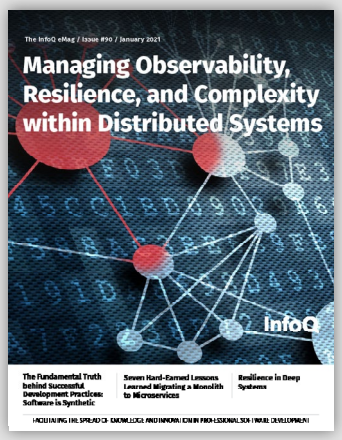
Because services are no longer hosted on a single server, it's important to account for latency and failure scenarios when communicating over the network. A simple retry logic with a clearly defined frequency of retries and a maximum retry count may be sufficient to handle most temporary network problems. Consider adding some intelligence to the retry logic using exponential backoff. Instead of retrying requests at a constant interval, exponential backoff will increase the amount of wait time in between retries and provides some relief to servers that are not responding because of overloads.

A circuit breaker can also be added as a medium between services as a self-protection and healing mechanism. For example, after a number of failed attempts, the circuit breaker will open and not allow additional requests to come through until the service is recovered. Set a timeout so your service doesn't end up waiting forever for an external service to respond. Try failing gracefully by presenting user-friendly messages or falling back to a last known good state in the cache. Be mindful of the user experience and do what makes sense for the business.

Conclusions

The first four sections focus on the foundational pieces that should be in place before you start down the journey of transitioning from monolith to microservices. Focus on the *why*. Think about modularity and data separation. Start with core services and shared resources and make the necessary operational changes. Getting these right will make the transition to microservices a much more enjoyable experience for your entire organization. Then, we talked about where to start and how to tie microservices back to product and business value. Finally, we covered two key concepts in microservices around service-to-service communications and building resilient systems.

Read recent issues

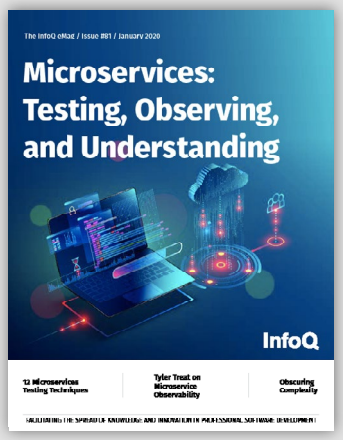


Managing Observability, Resilience, and Complexity within Distributed Systems

This eMag helps you reflect on the subject of reducing complexity within modern applications and distributed systems, and provides you with different perspectives and learned lessons from people who have already had to deal with challenges from the real world.

Re-Examining Microservices after the First Decade

We have prepared this eMag for you with content created by professional software developers who have been working with microservices for quite some time. If you are considering migrating to a microservices approach, be ready to take some notes about the lessons learned, mistakes made, and recommendations from those experts.



Microservices: Testing, Observing, and Understanding

This eMag takes a deep dive into the techniques and culture changes required to successfully test, observe, and understand microservices.

InfoQ

InfoQ @InfoQ InfoQ InfoQ