nginx.com

Microservices at Netflix: Lessons for Architectural Design

Tony Mauro of F5Sr. Content Marketing ManagerFebruary 19, 2015

10-13 minutes

In some recent blog posts, we've explained why we believe it's crucial to adopt a four-tier application architecture in which applications are developed and deployed as sets of microservices. It's becoming increasingly clear that if you keep using development processes and application architectures that worked just fine ten years ago, you simply can't move fast enough to capture and hold the interest of mobile users who can choose from an ever-growing number of apps.

Switching to a microservices architecture creates exciting opportunities in the marketplace for companies. For system architects and developers, it promises an unprecedented level of control and speed as they deliver innovative new web experiences to customers. But at such a breathless pace, it can feel like there's not a lot of room for error. In the real world, you can't stop developing and deploying your apps as you retool the processes for doing so. You know that your future success depends on transitioning to a microservices architecture, but how do you actually do it?

Fortunately for us, several early adopters of microservices are now generously sharing their expertise in the spirit of open source, not only in the form of published code but in conference presentations and blog posts. Netflix is a leading example. As the Director of Web Engineering and then Cloud Architect, Adrian Cockcroft oversaw the company's transition from a traditional development model with 100 engineers producing a monolithic DVD-rental application to a microservices architecture with many small teams responsible for the end-to-end development of hundreds of microservices that work together to stream digital entertainment to millions of Netflix customers every day. Now a Technology Fellow at Battery Ventures, Cockcroft is a prominent evangelist for microservices and cloud-native architectures, and serves on the NGINX Technical Advisory Board.

In a two-part series of blog posts, we'll present top takeaways from two talks that Cockcroft delivered last year, at the first annual NGINX conference in October and at a Silicon Valley Microservices Meetup a couple months earlier. (The complete video recordings are also well worth watching.)

- This post defines microservices architecture and outlines some best practices for designing one.
- Adopting Microservices at Netflix: Lessons for Team and Process Design discusses why and how to adopt a new mindset for software development and reorganize your teams around it.

What is a Microservices Architecture?

Cockcroft defines a microservices architecture as a *service-oriented architecture composed of loosely coupled elements that have bounded contexts*.

Loosely coupled means that you can update the services independently; updating one service doesn't require changing any other services. If you have a bunch of small, specialized services but still have to update them together, they're not microservices because they're not loosely coupled. One kind of coupling that people tend to overlook as they transition to a microservices architecture is database coupling, where all services talk to the same database and updating a service means changing the schema. You need to split the database up and denormalize it.

The concept of *bounded contexts* comes from the book *Domain Driven Design* by Eric Evans. A microservice with correctly bounded context is self-contained for the purposes of software development. You can understand and update the microservice's code without knowing anything about the internals of its peers, because the microservice and its peers interact strictly through APIs and so don't share data structures, database schemata, or other internal representations of objects.

If you've developed applications for the Internet, you're already familiar with these concepts, in practice if not by name. Most mobile apps talk to quite a few backend services, to enable their users to do things like share on Facebook, get directions from Google Maps, and find restaurants on Foursquare, all within the context of the app. If your mobile app were tightly coupled with those services, then before you could release an update you would have to talk to all of their development teams to make sure that your changes aren't going to break anything.

When working with a microservices architecture, you think of other internal development teams like those Internet backends: as

external services that your microservice interacts with through APIs. The commonly understood “contract” between microservices is that their APIs are stable and forward compatible. Just as it’s unacceptable for the Google Maps API to change without warning and in such a way that it breaks its users, your API can evolve but must remain compatible with previous versions.

Best Practices for Designing a Microservices Architecture

Cockcroft describes his role as Cloud Architect at Netflix not in terms of controlling the architecture, but as discovering and formalizing the architecture that emerged as the Netflix engineers built it. The Netflix development team established several best practices for designing and implementing a microservices architecture.

Create a Separate Data Store for Each Microservice

Do not use the same backend data store across microservices. You want the team for each microservice to choose the database that best suits the service. Moreover, with a single data store it’s too easy for microservices written by different teams to share database structures, perhaps in the name of reducing duplication of work. You end up with the situation where if one team updates a database structure, other services that also use that structure have to be changed too.

Breaking apart the data can make data management more complicated, because the separate storage systems can more easily get out sync or become inconsistent, and foreign keys can change unexpectedly. You need to add a tool that performs master data management (MDM) by operating in the background to find and fix inconsistencies. For example, it might examine every database that stores subscriber IDs, to verify that the same IDs

exist in all of them (there aren't missing or extra IDs in any one database). You can write your own tool or buy one. Many commercial relational database management systems (RDBMSs) do these kinds of checks, but they usually impose too many requirements for coupling, and so don't scale.

Keep Code at a Similar Level of Maturity

Keep all code in a microservice at a similar level of maturity and stability. In other words, if you need to add or rewrite some of the code in a deployed microservice that's working well, the best approach is usually to create a new microservice for the new or changed code, leaving the existing microservice in place. *[Editor – This is sometimes referred to as the immutable infrastructure principle.]* This way you can iteratively deploy and test the new code until it is bug free and maximally efficient, without risking failure or performance degradation in the existing microservice. Once the new microservice is as stable as the original, you can merge them back together if they really perform a single function together, or if there are other efficiencies from combining them. However, in Cockcroft's experience it is much more common to realize you should split up a microservice because it's gotten too big.

Do a Separate Build for Each Microservice

Do a separate build for each microservice, so that it can pull in component files from the repository at the revision levels appropriate to it. This sometimes leads to the situation where various microservices pull in a similar set of files, but at different revision levels. That can make it more difficult to clean up your codebase by decommissioning old file versions (because you have to verify more carefully that a revision is no longer being used), but that's an acceptable trade-off for how easy it is to add new files as

you build new microservices. The asymmetry is intentional: you want introducing a new microservice, file, or function to be easy, not dangerous.

Deploy in Containers

Deploying microservices in containers is important because it means you just need just one tool to deploy everything. As long as the microservice is in a container, the tool knows how to deploy it. It doesn't matter what the container is. That said, Docker seems very quickly to have become the de facto standard for containers.

Treat Servers as Stateless

Treat servers, particularly those that run customer-facing code, as interchangeable members of a group. They all perform the same functions, so you don't need to be concerned about them individually. Your only concern is that there are enough of them to produce the amount of work you need, and you can use autoscaling to adjust the numbers up and down. If one stops working, it's automatically replaced by another one. Avoid "snowflake" systems in which you depend on individual servers to perform specialized functions.

Cockcroft's analogy is that you want to think of servers like cattle, not pets. If you have a machine in production that performs a specialized function, and you know it by name, and everyone gets sad when it goes down, it's a pet. Instead you should think of your servers like a herd of cows. What you care about is how many gallons of milk you get. If one day you notice you're getting less milk than usual, you find out which cows aren't producing well and replace them.

Netflix Delivery Architecture Is Built on NGINX

Netflix is a longtime user of NGINX Open Source and became the

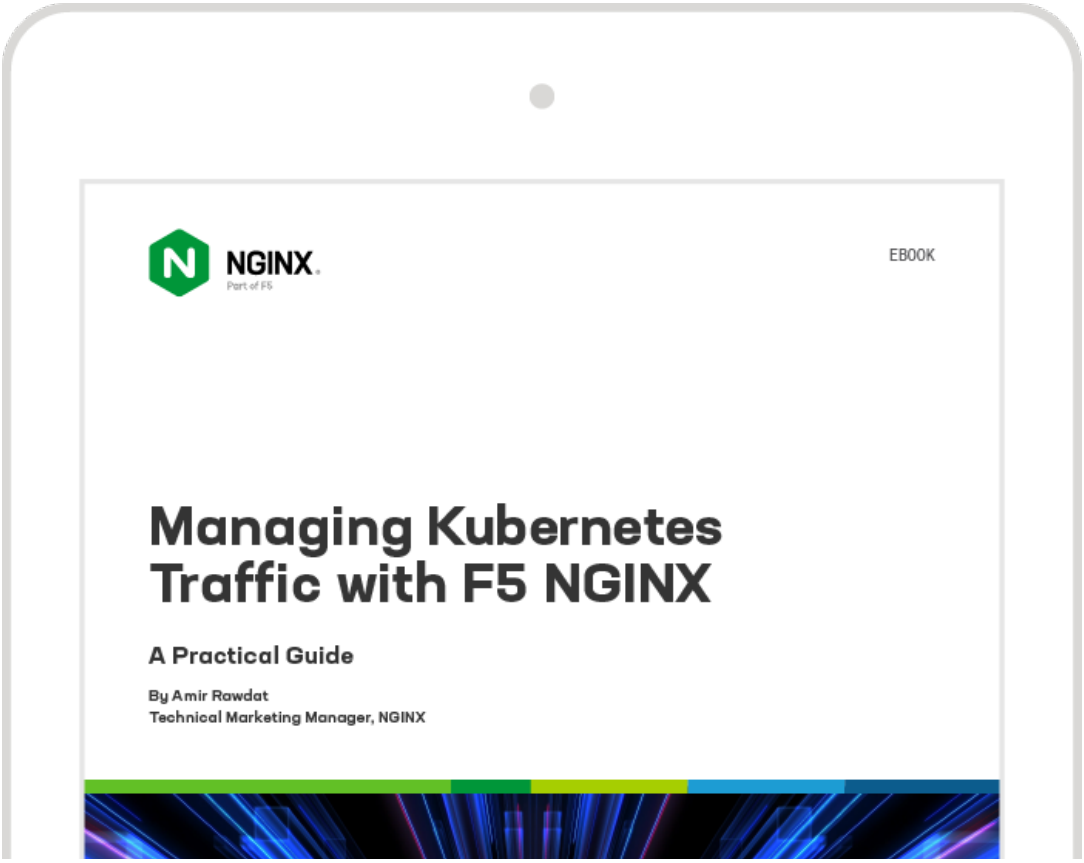
first customer of NGINX, Inc. after it incorporated in 2011. Indeed, Netflix chose NGINX as the heart of its delivery infrastructure, Open Connect, one of the largest content delivery networks (CDNs) in the world. With the ability to serve thousands, and sometimes millions, of requests per second, NGINX and NGINX Plus are optimal solutions for high-performance HTTP delivery and enable companies like Netflix to offer high-quality digital experiences to millions of customers every day.

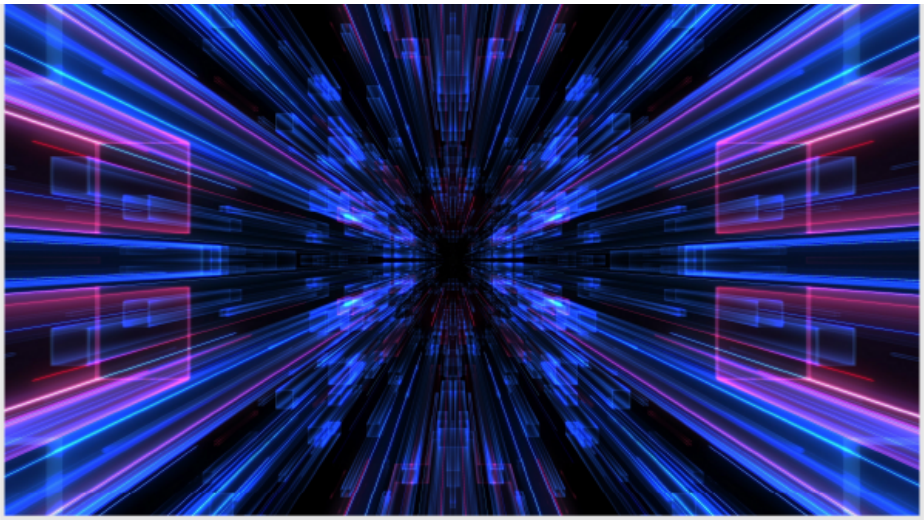
Video Recordings

Fast Delivery
nginx.conf 2014, October 2014

Migrating to Microservices, Part 1
Silicon Valley Microservices Meetup, August 2014

Migrating to Microservices, Part 2
Silicon Valley Microservices Meetup, August 2014





Managing Kubernetes Traffic with F5 NGINX: A Practical Guide

Learn how to manage Kubernetes traffic with F5 NGINX Ingress Controller and F5 NGINX Service Mesh and solve the complex challenges of running Kubernetes in production.