Monolith to Microservices: Transforming a web-scale, real-world e-commerce platform using the Strangler Pattern — Runscope Blog

This is the third post in our Featured Guest Series! Kristen Womack shares the story of the transformation Best Buy went through when moving from a monolithic application to a microservices architecture, and the main secrets that made it successful.

If you're interested in being a part of our next series, fill out this short form and we'll get in touch with you for our next run.

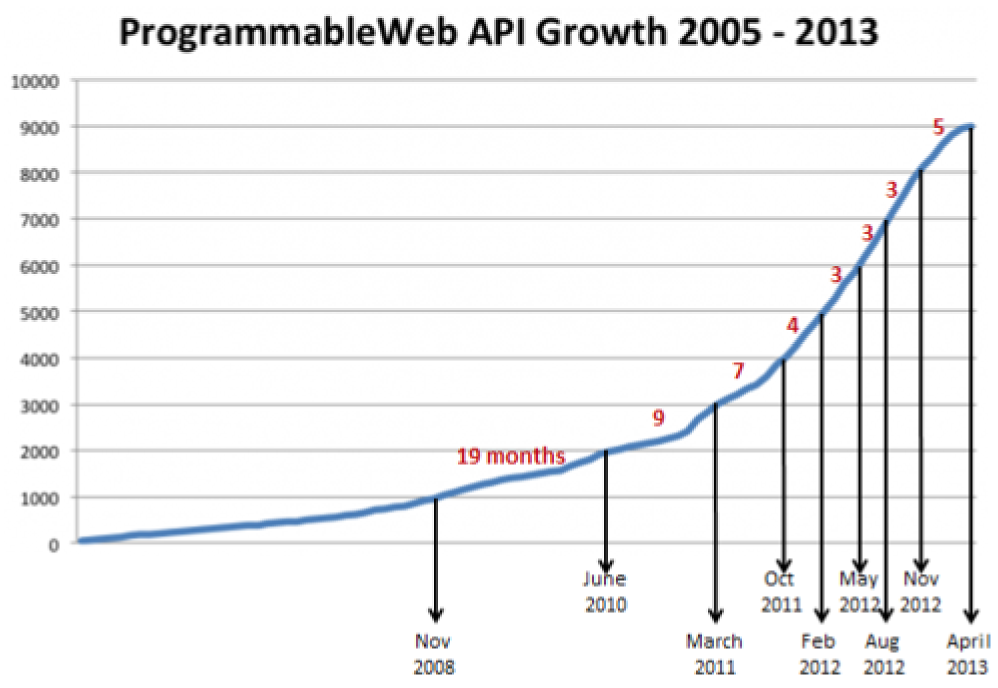
Microservices are hot these days. According to Google Trends, searches for "microservices" were almost non-existent five years ago. And here we are today:

Maybe your team is one of the many now moving toward a microservices architecture. And with good reason: breaking a monolithic application into smaller, simpler services increases your project's velocity, your ability to scale, and allows you to react more quickly to change.

In 2011, I was part of the Best Buy transformation that broke down the monolith of the website into separate web services. This is the story of that transition from monolith to microservices at one of the world's biggest e-commerce platforms—what we learned, what worked, and how you can learn from our experience—while highlighting two keys in our transformation success: focusing on culture, and using Martin Fowler's Strangler Pattern.

But first: How did we get here?

A monolithic code base is common, and often a byproduct of rapid product success. But as the application grows, there is a breaking point where it's difficult to keep up with the pace of market demand. A monolithic codebase becomes too cumbersome to be efficient in delivering new features and keeping ahead of the market. Integrations with other applications also become expensive. The need for companies to ease integration development with partners created a boom in public API growth from 2008 to 2013. Interestingly, the microservices trend seemed to follow shortly thereafter.



Transition and Transformation at a Major Retailer

In the 90's, Best Buy built their first website internally. In 2000, the company launched a new version built on IIS and SQL Server, then replaced that with a version built on ATG Dynamo and Oracle. From 2003-2010, most development was handled by outsource partners, primarily Accenture and Wipro - pretty standard for a retailer in the 2000s.

A decade later, the application ecosystem was overly complex, making simple changes and updates difficult. And due to the nature of outsourcing development, people with critical information and context about the systems walked out the door at the end of each project.

BEST BUY
Purchase gift card
MAKE INSIDE FEEL LIKE COURTSIDE
Shopping History
Your shopping history is empty. Let's fix that.

1996 2003 2017

In 2010, Best Buy kicked off an initiative to transform their e-commerce platform. The mission was to break the monolithic, tightly coupled application into microservices so that we could more quickly deploy new features and respond to market changes in the retail landscape.

Our approach to transformation was based on three pillars:

- People
- Process
- Technology

At first, that might seem counter-intuitive. **Developers often conclude that technology is the answer to everything, but people and processes are prerequisites to using technology effectively, especially software development**. (Don't believe me? Check out Conway's Law.)

One of the first things we did was to create a dependency graph of the interconnected modules within the monolithic platform to get an understanding of what we were working with. The graph was so congested and vast that in order to see the full shape of the drawing, you had to zoom out so far you could no longer read any text.

So, we embarked on a journey to rewrite Best Buy dot com—one of the largest e-commerce sites in the world. (At the time, Best Buy was number 10 on the Internet Retailer Top 500.) No big deal, right?

When rewriting an important software system, you can do one of two things:

- Rebuild the app from scratch in parallel

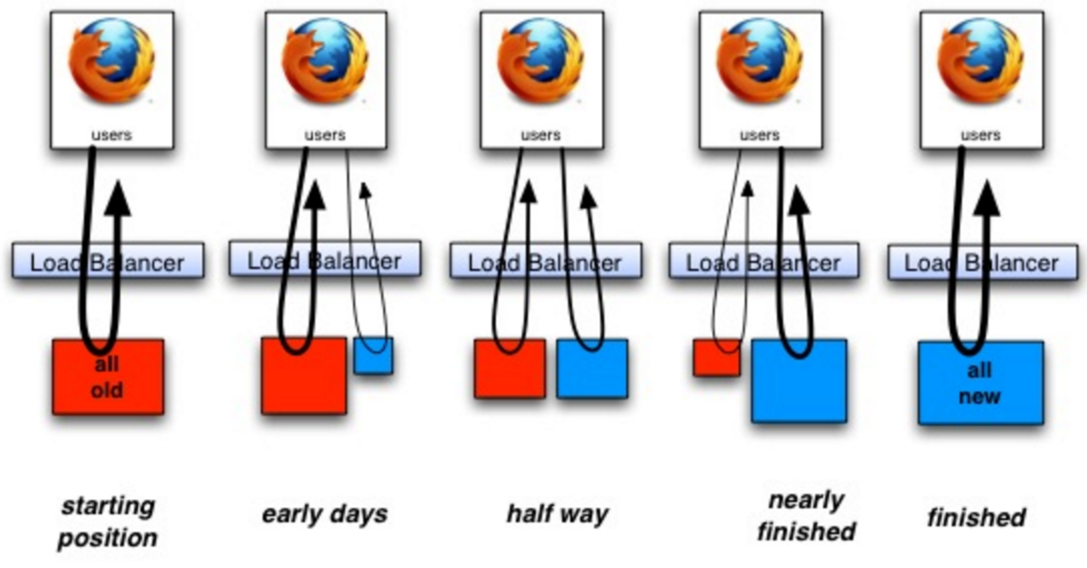
Or:

- Write around the old app and start to cut over to new services

You might think the first option is a no-brainer: "Just make a new one that does what the old one did." I've witnessed teams try to do this; it just doesn't work. When it does work, it takes a year or more, the customer/user cutover is difficult and noticeable, and revenue is lost in the process.

The (better) alternative to a complete rebuild is to slowly and systematically take over the old application. As you create new, independent services, you kill off the old corresponding services. This realizes benefits more quickly while reducing risk. This was dubbed the "Strangler Pattern" by Martin Fowler.

The process looks like this:



When we started the Best Buy project, our goal was to create microservices from the monolith.

The “before” state was infrequent “big bang” code releases, with deployments planned out **months in advance**. Releases were **regularly delayed** to finish work, and the actual deployment process was a massive undertaking that **required more than 60 people**, conducted in **all-night sessions** taking **more than 8 hours**. We had single points of failure in our application: spikes in traffic could bring the site down; contract teams spun up for projects and then left (with all the system knowledge) once the project was launched; the website didn’t support inventory complexities like buying open box products and shipping from stores; bugs lurked in every corner.

Deployments and Scaling

We looked to continuous deployment systems like Etsy and architectural icons like Netflix with the goal to deploy continuously, or —at a minimum—every two weeks. We were a long way from this, and the beginning was painful, but it was helpful to have a north star and others to look up to.

At the time, Best Buy was purchasing more and more infrastructure to support growing holiday sales each year. Holiday traffic can spike higher over the Black Friday weekend than the entire first quarter of the year, but the rest of the time that excess server capacity sat there, unused. And while we had more than one data center and a disaster recovery center, we didn’t have redundancy without heavy lifting.

To mitigate waste and risk, we moved our web services to the cloud, where we could scale new servers on demand and build an automated failover in different geographic regions. **Graceful failover was one of the main tenets of our transformation**. We didn’t want production failures or traffic scale to impact the business. This was important for elastic server capacity and graceful degradation of a service.

My team was in charge of building the product catalog APIs. As we methodically built around the old SQL database system with the new application, a distributed NoSQL key-value data store (Riak), we started moving over production traffic to the new services.

Early on in our development, we experienced a major failure where we lost a node in the Riak ring. Where the legacy database was previously a single point of failure, Riak handled the failure gracefully without a single customer noticing anything amiss. We only discovered the issue with monitoring. This was an exciting milestone!

Realizing Benefits Early

Within a few months, the product catalog rewrite had enough functionality for our first customer: the Enterprise Returns application. When customers come into the store with a broken TV and a Geek Squad plan, they would get a gift card to purchase a new television of the same value.

Calculating that value was complex because of how fast television technology developed. The returns app called the catalog web service to search comparable televisions by dimension, technology, and price. By delivering a valuable slice of the service, we immediately made a valuable impact for the company and customers.

Within the next six months we added more functionality, selecting the highest priority categories of the product catalog, to serve our second customer: the public APIs for external developers. Previous to the product catalog web service, the **public APIs were only able to get a refresh of the catalog changes every 24 hours**, which caused problematic data latency issues. After we had enough categories available in our service, they were able to switch over and realize **5-15 minute data latency** with deltas to keep the public APIs data fresh and as close to real-time change.

By having these two production clients early, we were able to learn a lot and improve our service and deployment pipeline before the entire website cutover from the tightly coupled SQL database, to the product catalog web service built on the Riak NoSQL data store.

Eventually, the end state was leveraging the product catalog across the enterprise, but if we had not used the Strangler Pattern, it would have taken us years. And by the time we delivered the rewrite, so much of the market and organizational needs would have changed. Using the Strangler Pattern allowed us to absorb change and learn as it was happening.

Feedback Loops and Learning Cycles

We created a development process that governed how we iterated through our work for **learning**, not for deadlines. That's not to say we didn't have deadlines—we did, and we took them quite seriously—but our **iteration goals** were to **learn how to continuously improve our work and delivery**, reduce noise, and detect important signals as early as possible.

Intentionally designed feedback loops were a big part of accelerating learning to reduce waste. This meant committing code early and often, and leaving no uncommitted code at the end of the night. This disciplined process, which included **developer-written acceptance tests and test-first driven development**, encouraged high quality. Running the test suite through Jenkins allowed for immediate feedback.

Pair programming was one of the things that made us most successful. Developers worked through thorny coding and design challenges more quickly than they could have alone, and caught mistakes that would have become pesky bugs later.

We documented everything (through the wiki as well as through our test-suite) so others could pick up where we left off, and also rotated through work to discourage siloing and single-threaded responsibility.

Key takeaway: Creating microservices is more about team organization and approach than it is about technology.

Through standardization, we focused on moving the culture from "fire-fighting" and "heroism" to a methodical, purposeful, calm process. With this discipline came **freedom from being reactionary and the ability to be proactive**. We were able to more clearly see the whole picture rather than the hot problem of the day.

We thought a lot about Conway's Law and purposefully architected organization into smaller teams by web service: one team for the product catalog, one team for commerce, one team for the product display UI, and so on. While maybe not true microservices, this was our first division of functional services. Each team was autonomous with individual goals of the overall platform mission.

The first team to have the most robust testing suite was the product catalog team—Magellan. They were also the first to bypass deployment checks, because we had automated the human function of validating the service. No more 3 AM validation checks after the deployment! Beautiful: humans can sleep while computers work.

Often in startups and in tech, development teams balk at process. But one of the biggest lessons I took from my time with the Platform Transformation team is that **collaboration, communication, and discipline will give you far more freedom and speed than an ad-hoc, no-process, do-what-you-want attitude and culture**.

Working hard on the culture allowed many of the teams to enjoy accelerated development and freedoms. Our focus was always on learning how to improve.

We used our retros to get critical feedback about what was working and what wasn't. We talked about what went well, what didn't go well and what we want to change in the next iteration. And we committed to regularly updating our team manifesto. If there was something there that we weren't doing, we would ask ourselves "do we want to start doing it or remove it from the manifesto?" We valued refactoring our processes as much we did our code.

But feedback loops are more than talking or retros—they're your test suites, logs, traffic patterns, demos, velocity, customer feedback and more.

Conclusion

Transitioning to microservices is tough but worthwhile. You want to design your organization in a way that will manifest the ecosystem architecture you want. **Success will depend more on people and process than technology**—so be ready to listen and learn everyday, emphasizing efficiency and discipline over dogma.

Focusing on building around the frays of the old system will reduce risk in an important system rewrite, whereas building in parallel will be high pressure, high risk, and often come with high opportunity costs. The Strangler Pattern is a proven approach to tackling legacy software.

There are now several books on microservices and production-ready software: