REST in Peace: Microservices vs monoliths in real-life examples

RDX Follow

Dec 13, 2016 · 8 min read

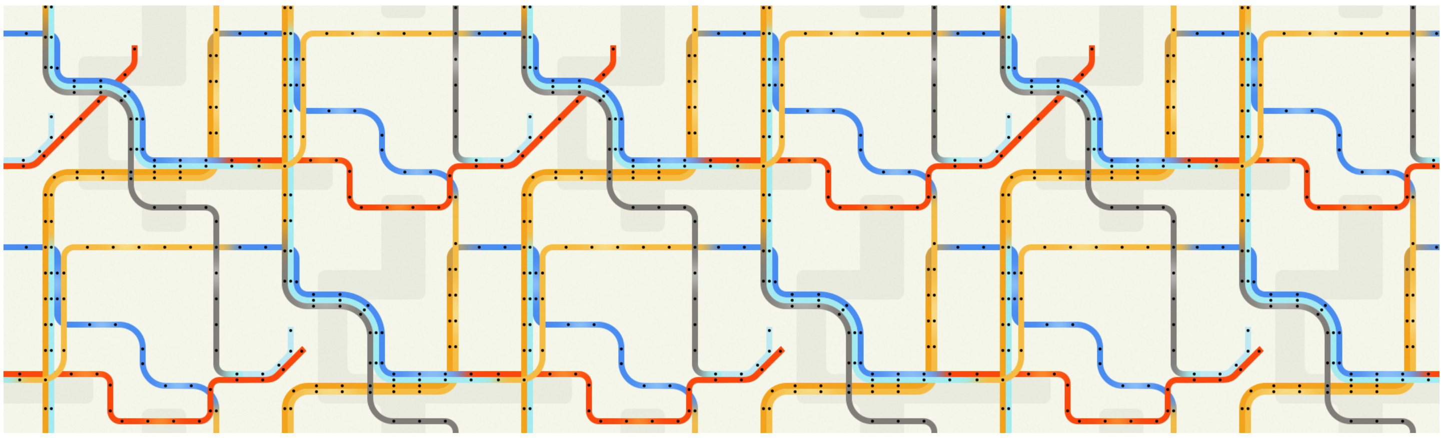

Image credit

I've consulted on a dozen microservice projects. Some were awesome *(this is the future!)* and some were equally frustrating *(who invented this crap?)*

It's the execution that matters, not the approach. You can succeed or fail with either. Don't just accept the love/hate propaganda out there.

Here are some experiences I've had that show how **both monoliths and microservices have their place.**

Picking the right tool for the right job

My team built a news reader app. It had to scrape articles, extract content, classify, serve an API, show admin panel, and manage users.

- Scrapy (python 2 at that time) is the leader when it comes to web scraping. So we created one microservice that scraped URLs. (Here's a bit more on my 15MB Docker image with Tor, Privoxy and a Process manager if you're curious.)
- Newspaper (python 3) is one of the best open source article extraction libraries. So we created one microservice that, given an

input URL, returns the extracted content.

- R has some good libraries for categorization. A colleague with good knowledge of R put together a service for categorizing articles using R's REST API.
- ActiveAdmin is one of the best and easiest Admin panel interfaces. For a long time, we have been leveraging it for admin screens, and we hooked it up in a few days here as well.
- Finally for the API gateway, we used Node.js with PassportJS (for multiple auth) and ElasticSearch.

Before this big re-write, our old app was a Rails monolith. It tried to reinvent everything mentioned before. You can imagine the trade-offs in code, effort, and time for quality, progress, and output.

Bad monoliths try to reinvent the world. They think that code and design patterns are the solution to everything. They strive to build reusable components, libraries, and also become a platform along the way. But they usually don't end up succeeding at any of these goals.

Takeaway #1: Microservices are a business solution, not just a technology solution. They save developers from having to waste time reinventing already solved technical problems.

I don't care if you call them "Services", "Service-oriented architecture", or "Microservices." Once we get past buzzwords, "monolith vs microservices" boils down to "1 service vs N services."

For our application, "N services" turned out to be the stack we wanted.

Monster CRUD apps

Some enterprise projects can simply be categorized as monster CRUD (Create-Read-Update-Delete) apps. They're a never ending stream of forms, data relationships, transformations, and hugely complicated "Business logic." The scope is huge, but the audience is tiny.

Nobody here cares about user experience best practices, and its common to have quick and ugly screens that get the job done.

We had one such enterprise transformation project. The old monolith was a database-view-based integration. It was pretty well designed, and only took a few people to maintain, including the ugly user interfaces.

The new backend services were mostly written in Java, so it was difficult to develop, integrate, share, and maintain compatibility. And new front-end services got pulled into the ever-changing world of single-page apps.

Deployment required a huge infrastructure automation setup. This was a few years ago, before the developer tooling ecosystem had matured. At the time, every contract change needed to be coordinated and updated by hand. It was the opposite of the previous scenario—the effort/output/overhead of microservices felt much bigger than that of the old monolith.

Takeaway #2: Be careful. Treating microservices as splitting code layers into docker-ized boxes can lead to “death by a thousand cuts.”

Taming a huge dependency tree

We had another legacy enterprise application transformation which did a lot of things like product scraping and parallel aggregation. It was a complicated, already troubled monolith.

When we’re doing a lot of orthogonal business features inside a single app, it causes a huge compile-time dependency tree, with tons of libraries and frameworks. Consequently, the runtime footprint, lifecycle, and build times were also long. This contributed to the real problem: developers couldn’t iterate quickly, and time-to-market of features suffered as a result.

Time taken to code a simple feature: several days.

Time to upgrade a library (guice) version: 1 week.

Time taken to upgrade a framework (Spring) version: forever.

Trivial stuff had the chance to rip apart every time estimate. Even a small refactor took long. It turns out that death by a thousand cuts is possible with monoliths as well.

We divided the project into some functional boundaries. We actively made sure to not share libraries and to avoid the dependency tree bottleneck.

For example, we used a modern client library for publishing messages over PubSub for the microservices. But the monolith’s big dependency tree did not allow us to use the same library. So we used a different HTTP-based PubSub client.

Microservices transitionists often make the classic mistake of sharing too many libraries and thereby re-create the same compile-time dependency trees (a “distributed monolith”).

But by avoiding sharing functionality, we could use different libraries to accomplish the same tasks without having to upgrade the world.

One of the services required a lot of concurrency (~1k lookups for every request). It initially leveraged RxJava. But it could be rewritten any day in Golang with the same API contract, and nobody would care about the dependency tree.

Takeaway #3: With microservices, you won’t hear the term “big rewrite” or “legacy system” ever again because there are no big systems.

The scalability myth

My team developed a code evaluation app. It was 90% CRUD, user interface, and reports, and 10% complex code evaluation for a dozen languages.

Before we came in, it was a series of microservices—one for each type of language, listening for different message queues. It had a separate front-end-as-a-service, admin-panel-as-a-service, and so on. Their original reasoning? Scalability.

We killed the whole thing and built a better monolith. It was completely done as a single Rails app—user interface, admin, back end, and candidate interface.

The code evaluation part ran as a background job (ActiveJob). We shelled this out across simple, stateless, one-off Docker containers. The core contract for evaluating code changed from REST/JSON into file/stdin/stdout. It scaled much better than the old system, because we just needed to ramp up the background workers to handle more code evaluations.

What looked like a sophisticated application from the outside—supporting 8 major programming languages with intelligent evaluation—was extremely simple on the inside.

Takeaway #4: You don't need microservices to run multiple instances of a service or worker. Good monoliths can also scale for some classes of problems.

It is quite possible to create scalable monoliths and unscalable microservices. It all comes down to how well you apply each's underlying principles.

Microservices as products

One of the most satisfying microservices projects I've worked on approached microservices as products. Our clients were extremely smart, techno-functional people, and they were clearly leveraging microservices as a business tool.

They modeled each service as a product, then released multiple products to different customers. They lined up the product releases in a way that each could leverage another's APIs. In turn, they created a brilliant ecosystem. It made them a market leader in their vertical.

The average enterprise today uses at least a dozen software products and integrations. The average cloud consumer uses multiple cloud products. I now see even non-technical people use micro-products and micro-apps. For example, one tool for interviewing, one for vacation tracking, one for payslips, etc. People are embracing smaller, more specialized tools that get the job done properly.

Takeaway #5: We are strongly and firmly into the micro-products, micro-apps and micro-services era. Better learn to do it well.

There is this constant fear of machine learning stealing programming jobs. Most programming jobs are becoming APIs today.

Distributed transactions

One of the most common arguments against using microservices is the risks associated with distributed transactions.

Are you calling an external Payment gateway system that deducts money, but could fail on your callback? Do you have multiple sign-on mechanisms (like email or OAuth)? Are you calling third-party SaaS products that don't have a rollback option? Are you leveraging Cloud APIs and storage buckets which don't respect your transaction boundary? Do you have workflows spanning multiple request lifecycles to the same service?

Then you already have distributed transactions in one way or another, whether you like it or not.

The whole idea that one system and one request can represent or control the entire transactional state of the business problem is a fantasy. If you can model your external integrations without distributed locks and transactions, then you can model your internal ones too.

Takeaway #6: Distributed locks and transactions aren't free with Monoliths, either.

Tooling vs People

Yes, more services means more tooling. This involves continuous integration and continuous deployment (CI/CD), infrastructure automation, developer tooling, the ability to design good APIs, contract sharing, documentation, client intelligence and libraries, processes, testing, and a lot of other tools.

You must be at least this tall to ride microservices.

If an organization does not have the engineering robustness and maturity to effortlessly run a handful of services (12factor, CI, CD, integration, testing, etc), it will be a disaster to switch to many of them.

Lots of people coming from a monolith mindset do Big Design Up Front. Microservices are best when they are **in-your-face**. Just throw away all the boilerplate, implement the API in a no-nonsense way and instead invest time in good quality unit/contract tests. Like with tooling VS people—microservices require a change in mindset, and a great deal of **unlearning**.

The good news is that many of these tooling problems have good engineering solutions. Docker, Kubernetes, REST, Swagger, Falcor, gRPC, CI/CD Pipeline tools, PaaS, Cloud, and so on. The ecosystem around microservices has already matured quite a bit, and its improving all the time.

The bad news is that microservices can't be learned like a framework or tool. They require a **holistic approach** that comes with experience. You need good people who are not only brute-force good coders, but also well-rounded engineers with a strong foundation in the whole software development lifecycle, from development to testing to deployment.

There are big enterprises that take months for every single integration. And then there are modern companies like Google, Facebook, and Netflix that run thousands of integrated services at a far greater quality and speed. The difference isn't just the tools—it's the people involved and their engineering approach.

Takeaway #7: Microservices are a culmination of multiple engineering practices. They have a steep curve of learning, unlearning, and transformation.

Conclusion

The microservices approach is just another tool in the solutionist's toolkit. And a tool is just a tool. It can end up being a powerful business asset, or an unproductive developer bottleneck. Whether we're right or wrong all comes down to how we use this tool.