Why are Big Balls of Mud Easy and Microservice Architectures Hard?

Rebecca Wirfs-Brock

"In many ways, having an existing [largely monolithic] codebase you want to decompose into microservices is much easier than trying to go to microservices from the beginning."—Sam Newman, *Building Microservices*

Every couple of years, shiny new software architecture ideas crop up and gain traction because they push on the boundaries of what's possible. That's largely a good thing. But they are also attractive nuisances.

So what's the attraction of a microservice-based architecture? Instead of building monolithic, hard to comprehend, change, or deploy big balls of mud (no one wants to create unmaintainble piles of #*&), why not bust up complex software systems into smaller, easily deployable components (e.g. microservices) using lightweight containers and relatively conventional communications protocols?

And then, let's push on the boundaries of system dependencies and make microservices as independent from each other as they can possibly be. In practice as well as theory, microservices can be redeployed at the drop of a hat. What joy! Anyone working on the inner guts of a microservice is free to make technology implementation choices, fixes, enhancements, or whatever, as long as they provide a relatively stable service interface and don't degrade the current performance characteristics of that service.

Sounds good. So what's wrong with this picture?

Regardless of how you cut it, dependencies between pieces and parts of complex systems exist. There's no getting around that. Whether you implement an event-style communication protocol between services or a more orchestrated workflow (that explicitly couples services), complex systems have to coordinate work between their parts.

No service exists in isolation.

Whenever I update a service, I have to either preserve its interface (or try to make a new version available to new clients, while preserving backwards compatibility, perhaps by deploying multiple versions of my service until everyone who depends on my interface rolls forward to a newer version). And who's to say that what I think is preserving my interface is the same as any consumer of my service thinks. Or what a consumer's test code has revealed about my service?

What can I safetly change without screwing up clients of my service? Does data ordering, defaults, or acceptable values matter? Even if I didn't think they should (I'm using JSON and by golly I'm declaring that should never depend on ordering of parameters or existence of data!) some consumer might have made assumptions about my interface. What I had considered a non-constraint they might have depended upon. So even the slighted change to my interface can break code in other services.

I also have to be aware of the assumptions I make about the staleness or liveness of the information my service consumes or produces, lest I change those unstated assumptions and as a consequence inadvertently change the behaviors of other services I depend on and/or interact with. And the number of services that use my service can expand and change over time.

These are no small things to consider. And comprehending how such a system actually accomplishes work is no small matter, either.

So let's get around to answering why it is that big balls of mud are easy, while successfully implementing a complex system with hundreds of microservices is hard: With a big ball of mud, you can hack and craft the necessary communications between pieces and parts of your system as you may. Sure, eventually that big ball of mud will get unwieldy and unmaintainable, but not all parts are equally muddy. Mud isn't homogeneous. There are bits and pieces that shine, while other parts are more complex, tangled, and well, muddy.

And sometimes the friction of having to figure out and untangle that ball of mud prevents you from making changes too rapidly. While a big ball of mud may have a lot of inertia, it can be "fractured" along the natural paths of communications between parts, and made smaller. As long as you are willing to muck in the mud. And as long as there aren't deep, tangled communications between parts or excessive data dependencies, we have good instincts on how to approach decomposing a big ball of mud, because, well, it is working, even if ineffeciently. We can observe it and refactor, rework, and slice up related parts in a semi-rigorous fashion using controlled experiments.

On the other hand, starting a microservices architecture may be easy. It is predictably and steadily growing it that is hard. To start, it is guaranteed that we won't get the right interfaces or eliminate data dependencies between microservices. Interfaces *should* be stable, but until we understand how clients of microservices actually want to use our data, we won't get things right. Services are designed to be useful (and as a consequence as they are being used our notion of what makes them useful should also naturally evolve).

Nor will we understand all temporal couplings and “desired behaviors” given distributed interacting microservices. The behavior of such a system isn’t predictable. It’s emergent.

We can constantly tweak things (as long as we comprehend them). It’s when our systems get larger and more complex that they become challenging to comprehend.

So while I might initially be happily lulled into believing that my microservice is mine to tinker with, if it exists in a large ecosystem of other microservices, I am setting myself up for inevitable surprises. The behavior of a complex microservice architecture is emergent and always in flux. It isn’t exactly architected, so much as tended, monitored, and sustained. That’s what makes growing a microservice architecture hard: it requires ongoing attention, dilligence, and architectural stewardship.