
Microservices Overview

What are Microservices?

Microservices Characteristics

Deconstructing a Monolith

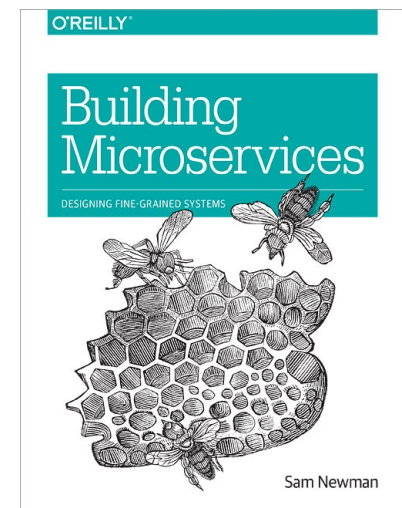
What are Microservices?

Fowler (2014):

Microservices are "an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."

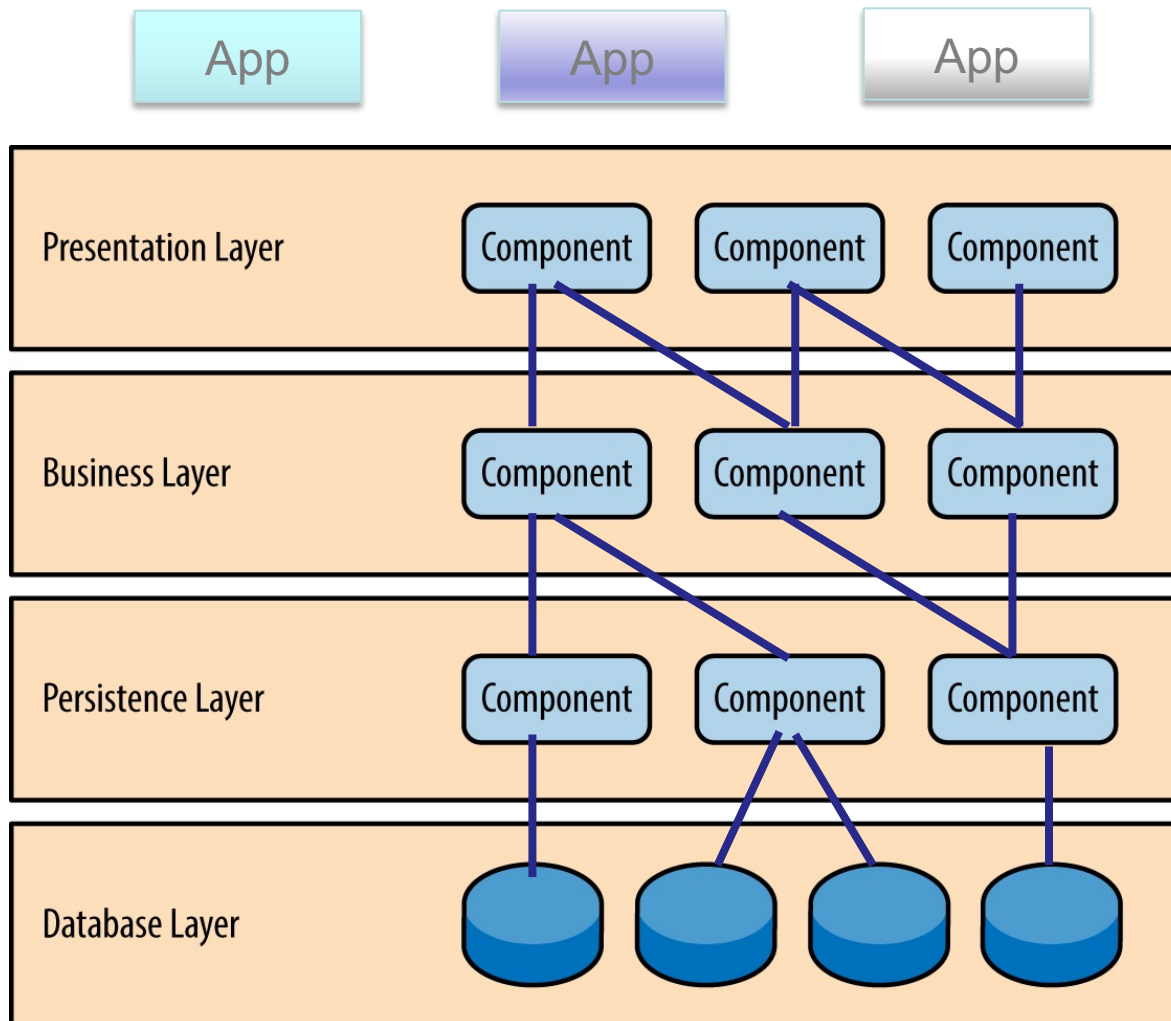
Norman (2015):

"Microservices are small, autonomous services that work together"



Microservices and Lean in 3 Diagrams

(1) The most common software architectural style is the layered “wedding cake” style



Components at each layer are “wired” to components at the next layer to provide services

- Creates reusability
- Each “tier” may be deployed independently
- There is a clear separation of concerns (SoC)

Conway’s Law (stay tuned) says we arrange teams and work accordingly

- Teams lose independence
- How does value flow?

Microservices and Lean in 3 Diagrams

(2) The complexity of the middle layers was factored into the *Enterprise Service Bus*

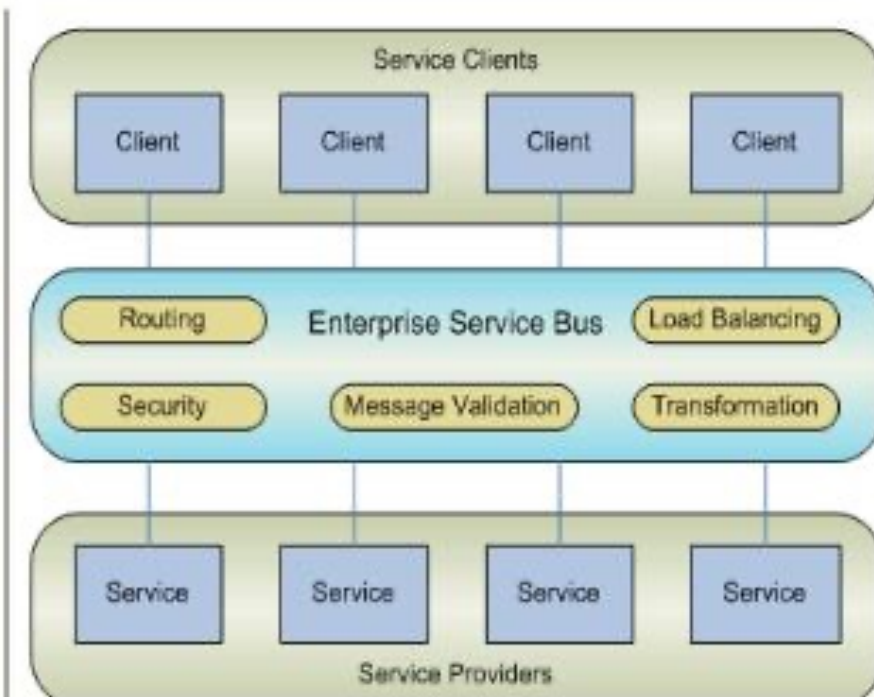


Image from SAP.com

ESB took responsibility for the complexity of integration

- Normalized messages
- Service orchestration
- Service integration adapters

Conway's Law (stay tuned) says we arrange teams and work accordingly

- Now you had the ESB team and the service teams and the app teams
- Both the service teams and the app teams waited on the ESB team (a queue formed!)

Microservices and Lean in 3 Diagrams

(3) What if instead we align our approach around the value to be delivered

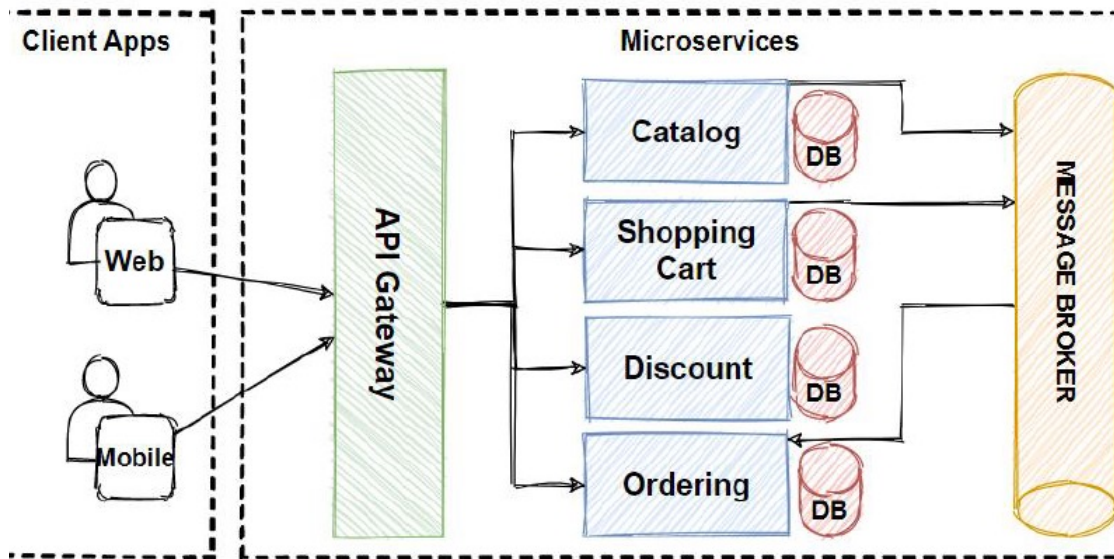


Image from [Mehmet Ozkaya Medium.com blog post](#)

There are significant challenges moving to a true Microservices architecture (and most aren't really doing it). But the enabling benefits are significant by applying our principles of *small* and *independent* so we can focus on flow

The layers in the middle are combined

- Blue boxes
- Note the mini-DBs

Conway's Law (stay tuned) says we arrange teams and work accordingly

- Teams are arranged around value-added services
- Value flows through *smallness* and *independence*

MICROSERVICES CHARACTERISTICS

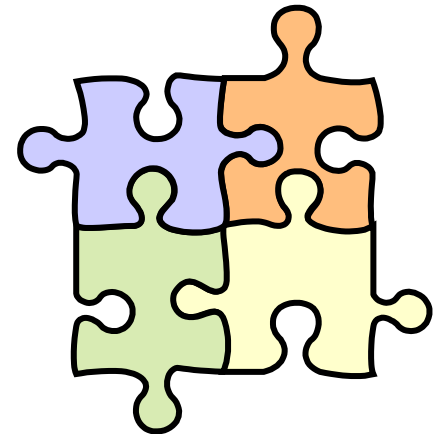
Characteristics of a Microservice Architecture

1. Componentization via Services
2. Organized around business capabilities
3. Products not Projects
4. Smart endpoints and dumb pipes
5. Decentralized Governance
6. There is no #6
7. Decentralized Data Management
8. Infrastructure Automation
9. Design for Failure
10. Evolutionary Design

We will drill-down on #2 and #5

Componentization via Services

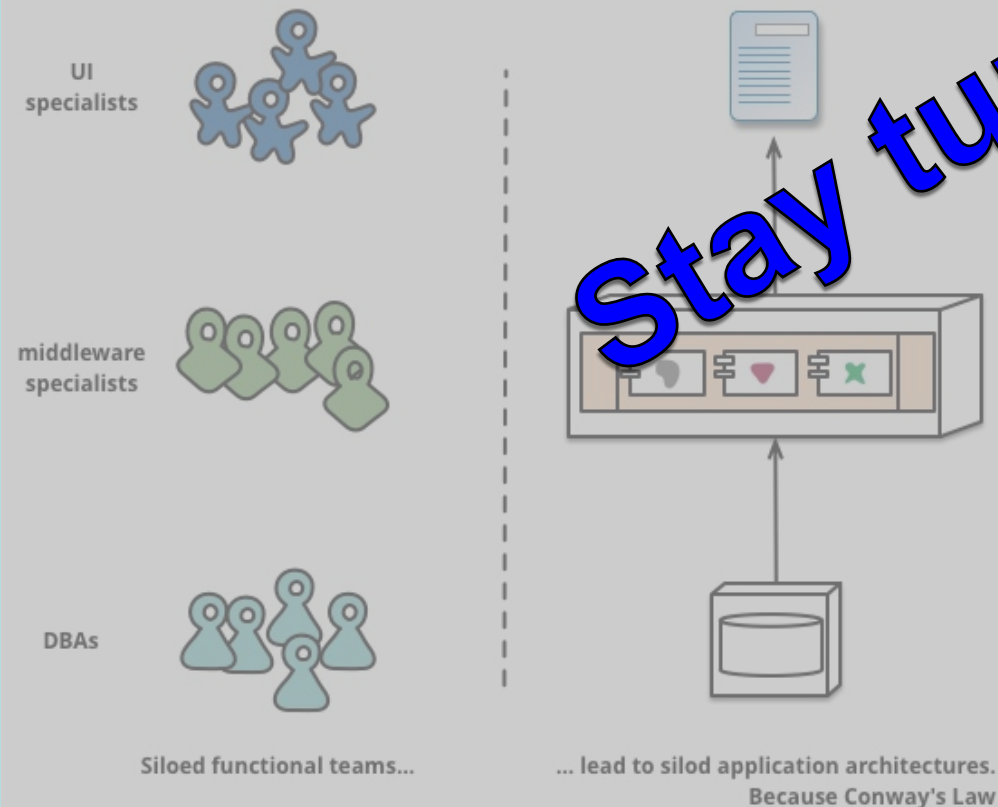
- “A component is a unit of software that is independently replaceable and upgradeable” -- *Lewis & Fowler*
- Services may be backed by Components
 - So what does *Componentization via Services* mean?
 - “Componentize” is a verb, so it is meant as a form of decomposition
 - In essence is it *modularization*
 - The important interpretation Lewis & Fowler make is that software components, running in the same process space, use in-memory calls and typed interfaces, *and these boundaries are often violated*
 - Implementing services (Service-ization?) forces the typed interfaces to become *service contracts*, and in-memory calls to be replaced by network calls.
 - Of course, network calls are more expensive so there is a risk here.



Organized around business capabilities

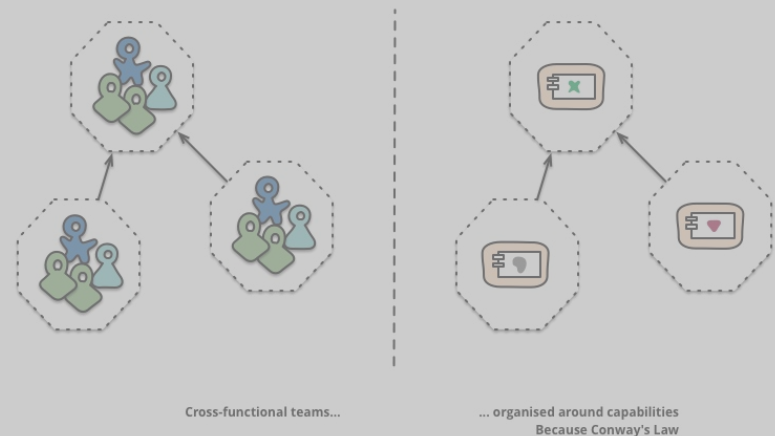
Conway's Law (1967)

"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure"



The idea is to use cross-functional teams.

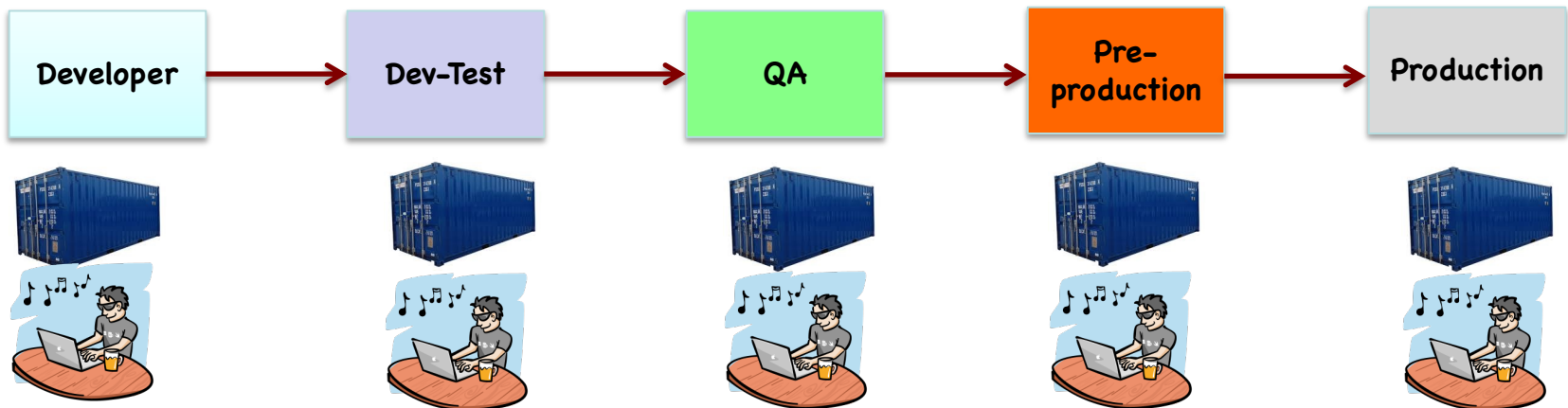
Instead of teams per tier, create a team with all of the specialities (UX, DB, PM, etc.)



Products not Projects

Teams follow Products through Production and beyond

- This is the rejection of the “throw it over the wall” assumption
- The traditional word “staging” for different environments is taken from the arts concept of arranging a stage for a performance
 - In our ops context, it means setting up the environment just so
 - The “performance” is the purpose of that environment
- But “staging” has become a synonym for “slowness”, as in a *stage-gate* process, where all workflows idle waiting to move on



- Now, the developers *follow their container through Production*
- That container is the Product

Smart Endpoints, Dumb Pipes

Microservice enthusiasts talk about the “Unix Philosophy”

- Simple dedicated functions as programs, which are assembled into larger functions through a *pipe*, a special form of composition.
- **Example:** `cut -f 3 -d, list.txt | awk '{print $1}' | sort | uniq`
 - “grab the 3rd field delimited by , print/sort it and remove duplicates”

More Railing Against the ESB:

“...we've seen many products and approaches that stress putting significant smarts into the communication mechanism itself...ESB products often include sophisticated facilities for message routing, choreography, transformation, and applying business rules ...Applications built from microservices aim to be as decoupled and as cohesive as possible - they own their own domain logic and act more as filters in the classical Unix sense - receiving a request, applying logic as appropriate and producing a response. These are choreographed using simple RESTish protocols rather than complex protocols such as WS-Choreography or BPEL or orchestration by a central tool.”

- “Where does the integration logic go”? Clearly the Microservice community believes it goes in smart endpoints with lightweight and simple communication protocols

Decentralized Governance (or, no ESB!)

Decentralized Governance is mainly about teams (like the previous principle #2)

1. Teams are cross-functional and own all aspects of a service
 - Design, Development, Maintenance, Test, Deployment, Support...
 - No more “throw it over the wall”!
2. Teams are fully empowered to decide the technology stack they want to use to implement the microservice

Aside: #1 is kinda new, but haven't we seen #2 before?

- Both Fowler and Freeman acknowledge we have seen technologies and approaches for interoperability before, and they hypothesize a number of factors and changes
 - In legacy days, the use of interoperable technology stacks was driven by legacy systems, or your need to have Java talk to .NET
 - Distributed computing vendors did not like decentralization since there was nothing to sell like an “ORB” or an “ESB” or a homogeneous stack

There is no #6

Decentralized Data Management

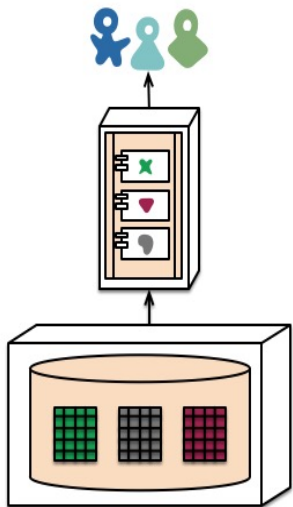
Dealing with Data (System of Record) is a very tricky part of developing a Microservices-oriented platform.

- One tricky part is identifying software boundaries
- Another tricky part is getting away from n-tier deployed layers
- But the trickiest part is how do we separate data (“world model state”)

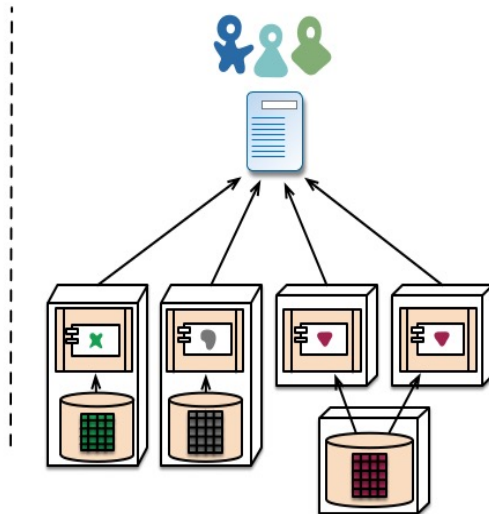
Behind every application is a big database that supports it
Microservices suggest you should partition your database to support each MS

MS Community Perspective

- Eventual consistency instead of transactions
- References in code, not foreign keys
- Use Domain-Driven Design
- Apply the *Polyglot Persistence* pattern
- Norman indicates there is no magic here, MS folks just seem to wave their hands and say YAGNI!



monolith - single database



microservices - application databases

Infrastructure Automation

Continuous Integration and Testing

- The best practices of automating your builds and deployments to staging environments, integrating multiple code streams as you go.

Continuous Deployment

- The focus on reducing cycle time from code development to release
 - Eases migration, earlier user feedback, faster ROI of development, incremental upgrades over large-scale migrations
 - The point: There is a lot of \$\$\$ at stake!
- To do Continuous* well, you need an Automated Infrastructure

Continuous* and Microservices

- Like containers, just because you do Continuous* doesn't mean you are, or have to do, Microservices – it helps with monoliths as well
- But then enable rapid evolution of a software system through small changes automatically and quickly rolled out, reducing/removing the need for elaborate staging (and stage-gate) processes



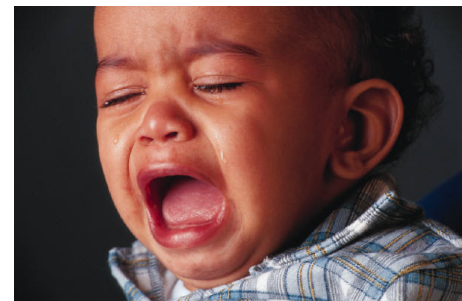
Design for Failure

Design for Failure is the concept of *Programming Safely* extended to a distributed system

- The network can fail, or is at least not as reliable as in-memory calls
- Emergent behaviors through Choreography includes bad behavior!

Lewis & Fowler are quick to point out that this is not a “good thing” for Microservices

- “Dan the Developer” has to write more code to detect and handle failure scenarios gracefully, and to avoid implicit state
 - That code can be more complex. For example, unlike an exception handler working on one call stack, now you have many call stacks
- “Oscar the Ops Person” needs powerful monitoring and logging tools providing a coherent image of what is going on in the service network, and tools to spin up new instances if there is a failure of some kind.
 - In other words, we still have the complexity of tracing user-facing information across multiple tiers



Evolutionary Design

Let's revisit 2 distinct camps in modern SE:

- Agile-oriented folks “embrace change”, “KISS”, “Refactor”, “use a central metaphor”, and believe in “emergent design” not BUFD
 - These folks think architecture folks are overly grandiose
- Architecture-centric folks believe it is imperative to have multiple architectural perspectives, a style, and well-defined interfaces
 - These folks think Agile ignores architecture (or just lie to themselves)

Evolutionary Design tries to have it both ways

- Keep services small so change is small & architectural drift is small
- Don't try too hard to evolve a service, it is so small it can be scrapped and replaced almost as easily as it is versioned
- *Service boundaries* should enforce *isolation*, meaning changes do not cross service boundaries (and things get hard if they do)
- As Norman points out, you absolutely have to get the service boundaries (or what Lewis&Fowler call *service contracts*) right early

IMHO there are some simplifying and self-serving assumptions made by the ED folks, like how do you magically evolve services independently?

Oh, and one more thing...

We wanted to revisit this notion of “size”

- or -

How “micro” is a microservice?

Norman: “Small enough and no smaller”

- As you get smaller, you maximize benefits & downsides of microservices
 - the benefits around autonomy, evolvability, flow, etc.
 - the downside is increased complexity of dealing with the moving parts
 - More complexity in the choreography
 - More network traffic and chattiness
 - Cognitive complexity of the big picture
 - Management complexity of the teams

Fowler (GOTO 2014 talk)

- One responsibility – *but how big is that responsibility?*
- Should “fit in one’s head” – the cognitive question – but still vague
- The *2-pizza rule* (Amazon/Bezos) – a meeting of the team required to support the microservice should be fed by only two pizzas/



So yeah, it is still pretty vague, and this is a risk as well

Organized Around Business Capabilities
Decentralized Governance

MICROSERVICES CHARACTERISTICS DRILL-DOWN

Characteristics of a Microservice Architecture

1. Componentization via Services
2. Organized around business capabilities
3. Products not Projects
4. Smart endpoints and dumb pipes
5. Decentralized Governance
6. There is no #6
7. Decentralized Data Management
8. Infrastructure Automation
9. Design for Failure
10. Evolutionary Design

#2 and #5 are the concepts most tied to Agile/Lean organizations

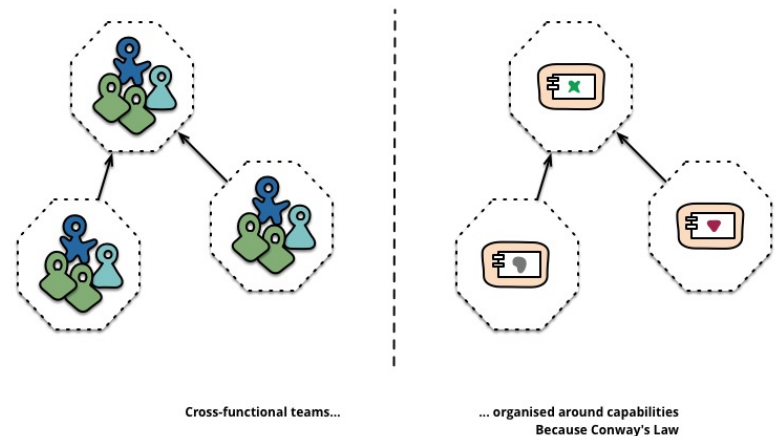
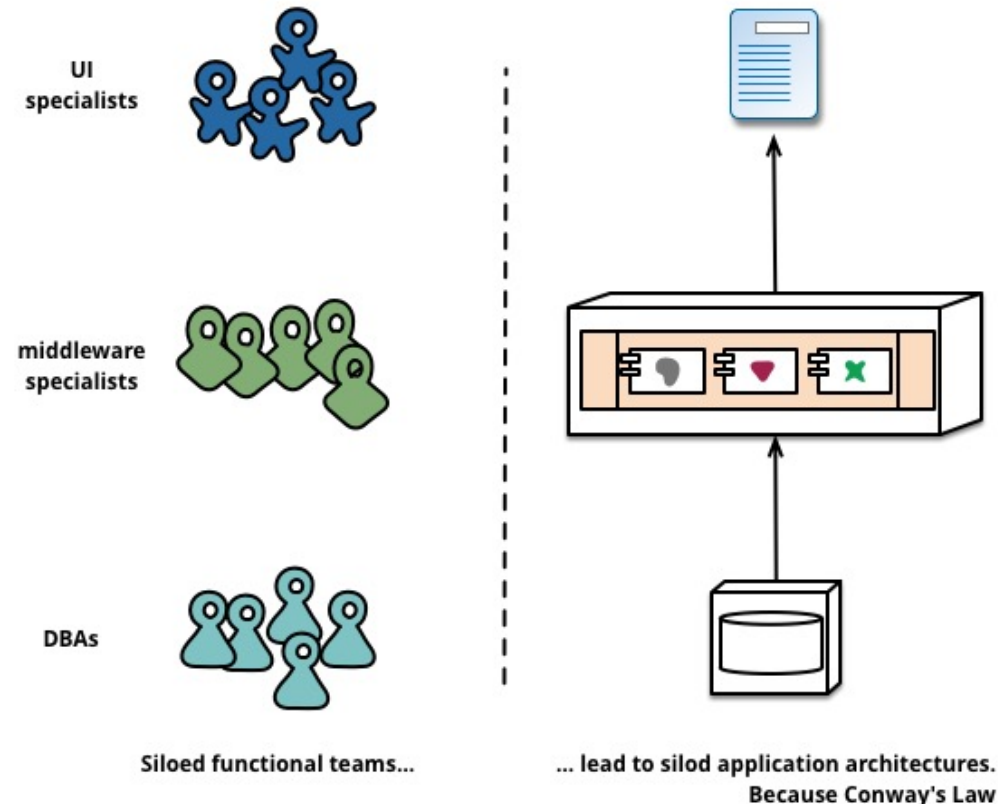
Organized around business capabilities

Conway's Law (1967)

"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure"

The idea is to use cross-functional teams.

- Instead of teams per tier, create a team with all of the specialities (UX, DB, PM, etc.)



Decentralized Governance (or, no ESB!)

Decentralized Governance is mainly about teams (like the previous principle #2)

1. Teams are cross-functional and own all aspects of a service
 - Design, Development, Maintenance, Test, Deployment, Support...
 - No more “throw it over the wall”!
2. Teams are fully empowered to decide the technology stack they want to use to implement the microservice

Aside: #1 is kinda new, but haven't we seen #2 before?

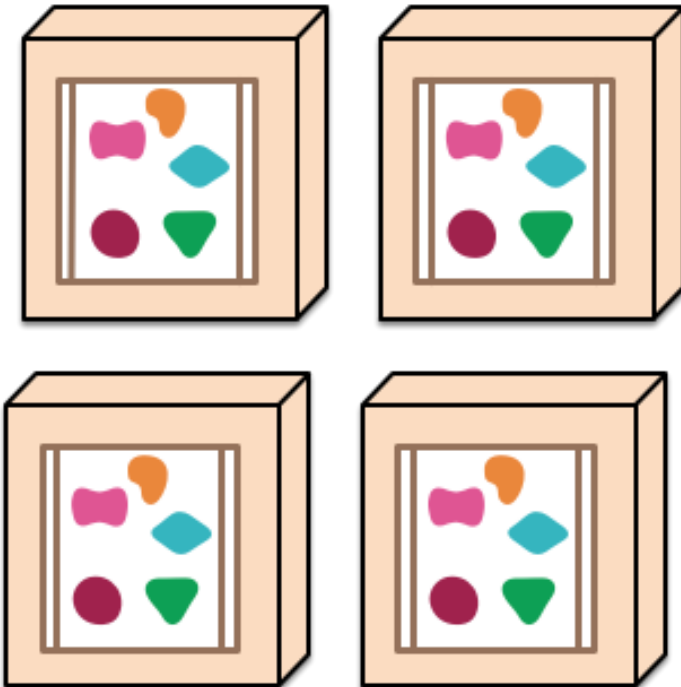
- Both Fowler and Norman acknowledge we have seen technologies and approaches for interoperability before, and they hypothesize a number of factors and changes
 - In legacy days, the use of interoperable technology stacks was driven by legacy systems, or your need to have Java talk to .NET
 - Distributed computing vendors did not like decentralization since there was nothing to sell like an “ORB” or an “ESB” or a homogeneous stack

Splitting the monolith

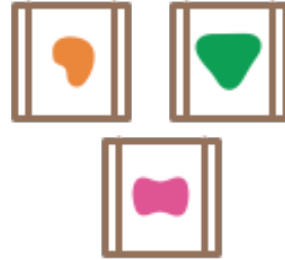
A monolithic application puts all its functionality into a single process...



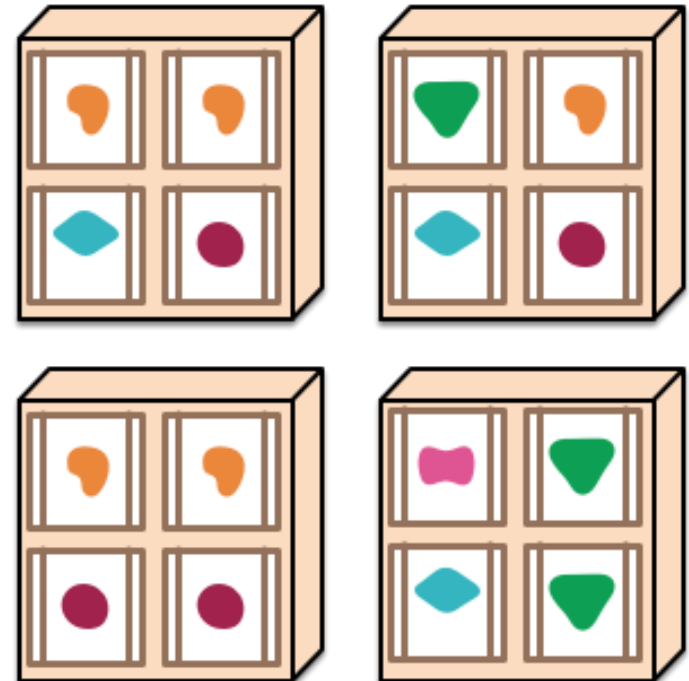
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



Pros and Cons of Microservices (Fowler)

Pros:

1. Strong Module Boundaries
 - Microservice enforce module structure
2. Independent Deployment
 - Simple services are easier to deploy
 - Less likely to cause system failures when they fail
3. Technology Diversity
 - Mix multiple languages, platforms, frameworks
4. Smaller is better
 - Just like work, we can manage small architecture change at a time

Cons:

1. Distribution
 - Harder to code
 - Remote calls introduce latency
 - Remote calls are always at risk of (network) failure
2. Eventual Consistency
 - Consistency of world model state is what we mean
 - Difficult here, so weaker consistency is proffered
3. Operational Complexity
 - There will be lots of services
 - These will be redeployed often (even continuously)

Summary: Dr. Gary's Thoughts

- Microservices are a trendy topic without a strong definition or agreement on what it exactly is
- Norman describes them as the nature evolution and convergence of a number of things –
 - Agility and the need for speed
 - Automation and better tools
 - Lessons good and bad from Distributed Objects and WS-* SOA
- The idea that teams follows products through to production is really a new and organization-altering concept

Microservices remind me of Krutchen's model

Microservices addresses an impedance mismatch between Logical View & Implementation/Deployment views. These are aligned so it reduces the cognitive burden of understanding an architecture, leading to fundamental evolution & process efficiency benefits.

