Microservices Characteristics

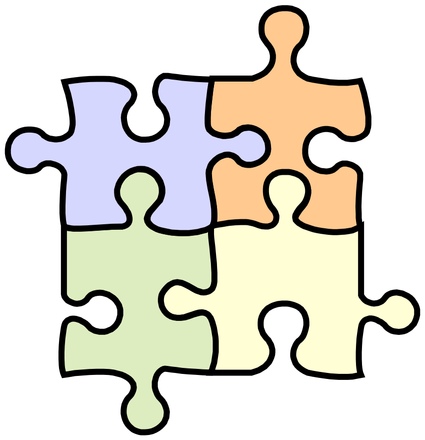
Characteristics of a Microservice Architecture

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

We talked about #2 and #5 already, let's discuss the rest of these characteristics

Componentization via Services

- "A component is a unit of software that is independently replaceable and upgradeable"
- We said before that "Services may be back by Components"
  - So what does *Componentization via Services* mean?
    - "Componentize" is a verb, so it is meant as a form of deomposition
    - It 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

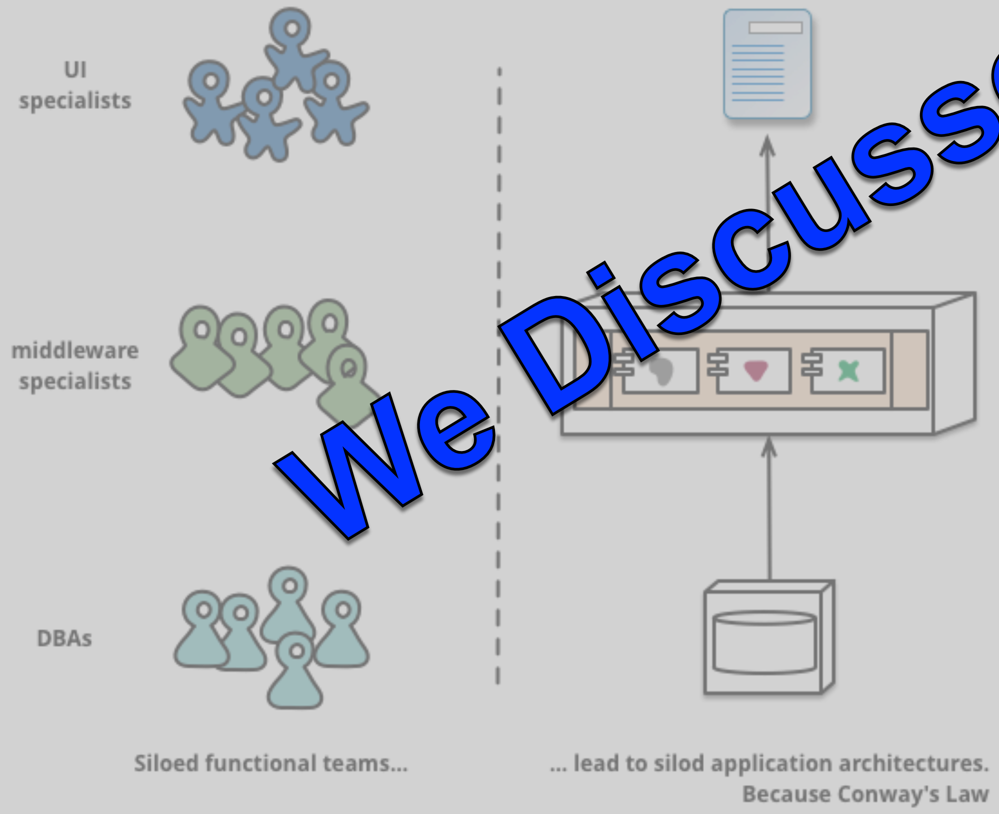
Conway's Law (1967)

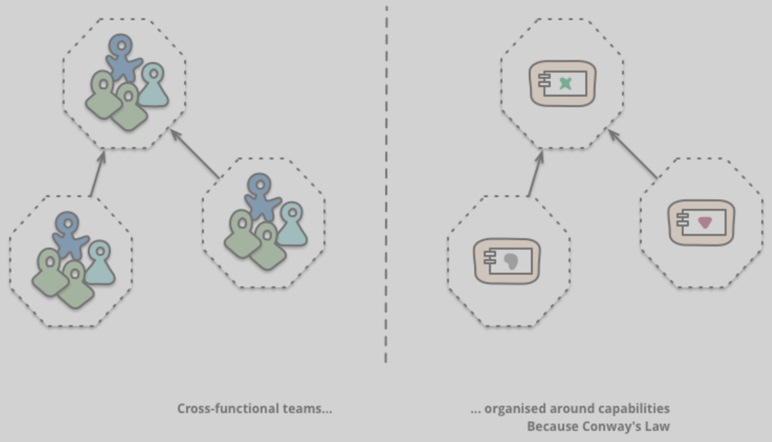
"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure"

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.)

We Discussed Already

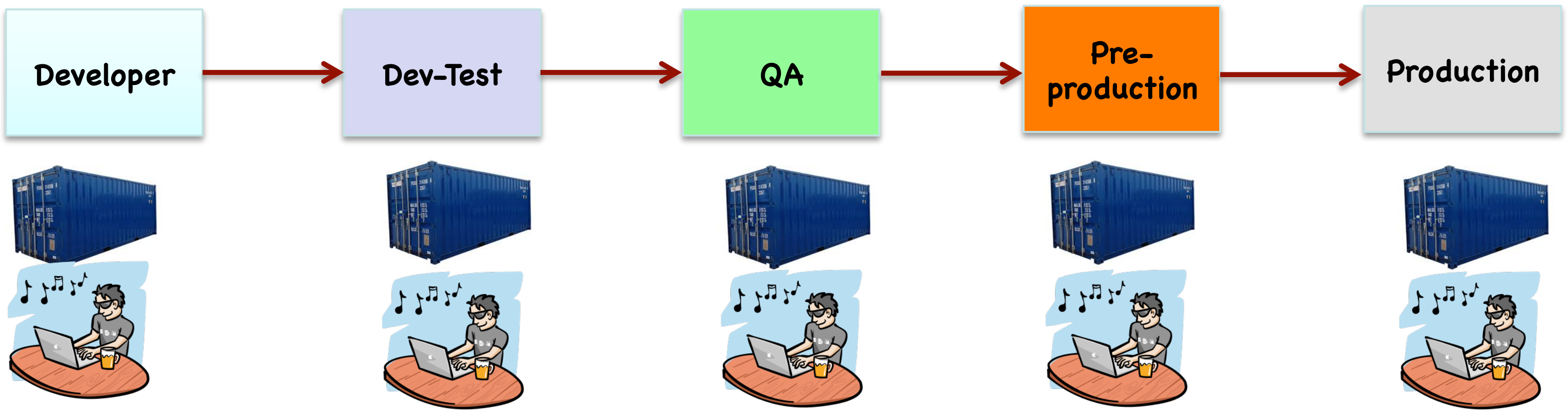

UI specialists
middleware specialists
DBAs
Siloed functional teams...
... lead to silod application architectures. Because Conway's Law


Cross-functional teams...
... organised around capabilities Because Conway's Law

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 <u>*Product*</u>

Smart Endpoints, Dumb Pipes

Microservice enthusiasts like to talk about something called 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.”

- See the “Pipe and Filter” architectural style
- We’ve discussed “where does the integration logic go”? Clearly the MS 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 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

We Discussed Already

Decentralized Data Management

We will revisit at the end

Infratructure Automation

Continuous Integration and Testing

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

Continuous Deployment

- The focus on reducing the 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

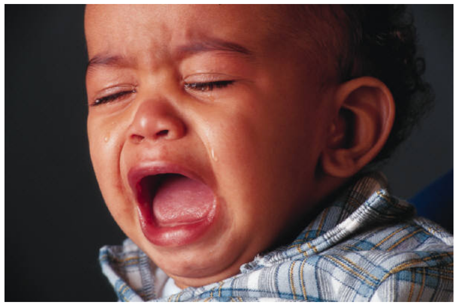
Design for Failure

Design for Failure is the concept of *Programming Safely* (remember SER316?) 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 mointoring 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?

There is no #10

Or really no #6

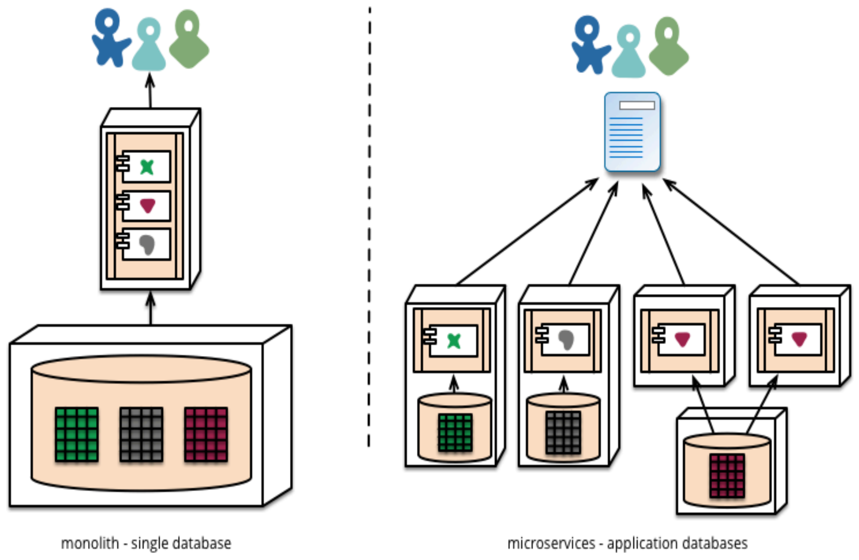
Decentralized Data Management

Why did we wait on this? Because it is another (tricky) part of the Splitting the Monolith problem.

- One tricky part was identifying software boundaries
- Another tricky part was getting away from n-tier deployed layers
- This tricky part is how do we separate data (or “world model state)

Behind every application is a big database that supports it

Microservices suggest you should partition your database to support each MS


monolith - single database
microservices - application databases

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!

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, 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)

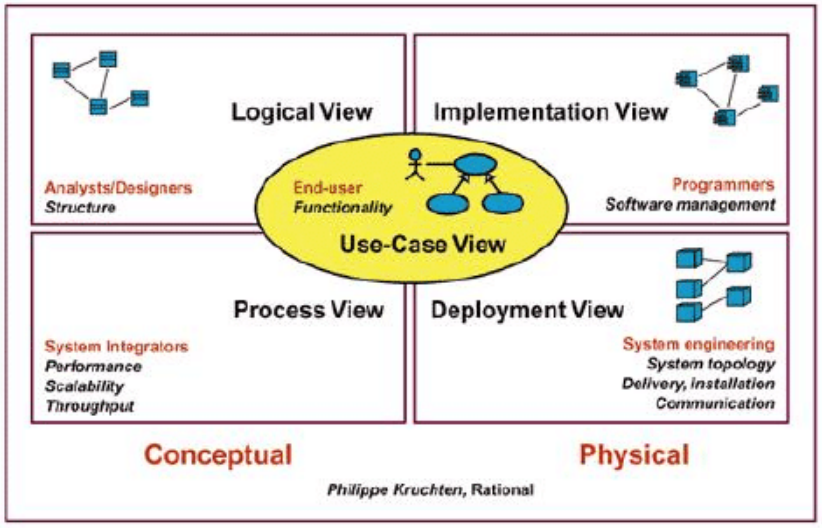
- One responsibility – *but how big is that responsibility*?
- Should "fit in one's head" – the cognitive question – but still vague
- The *2-pizza rule* (Amazon/Bezos) – a meeting of the team required to support the microservice should be fed by only two pizzas/

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

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.


Logical View
Analysts/Designers
Structure
Implementation View
Programmers
Software management
End-user
Functionality
Use-Case View
Process View
System Integrators
Performance
Scalability
Throughput
Deployment View
System engineering
System topology
Delivery, installation
Communication
Conceptual
Physical
Philippe Kruchten, Rational