Microservices

Beyond the Hype

Eric Kepes

What is a Microservice?

* Good question…

My Definition

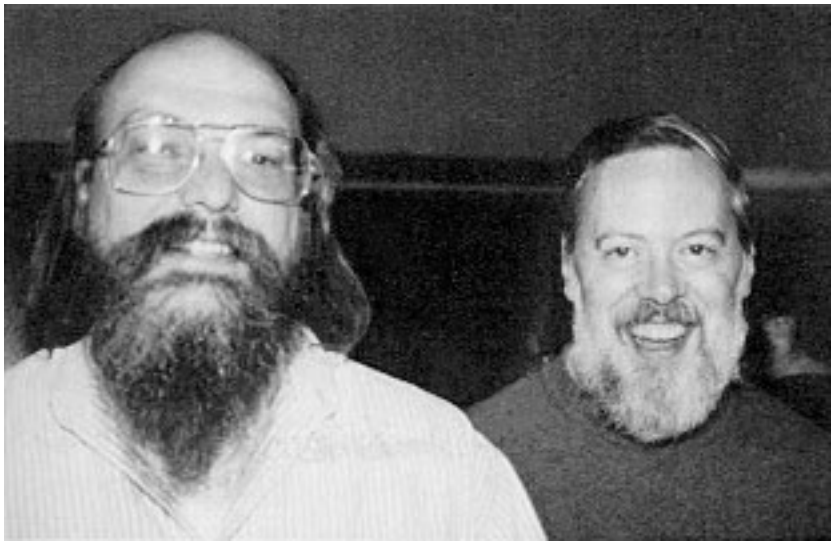
A system broken into smaller parts, which

* encapsulate a well-defined business context
* are deployed independently
* communicate loosely over known channels using well-defined interface specifications

How did we get here?

History

UNIX Philosophy



Ken Thompson and Dennis Ritchie

https://en.wikipedia.org/wiki/Unix_philosophy

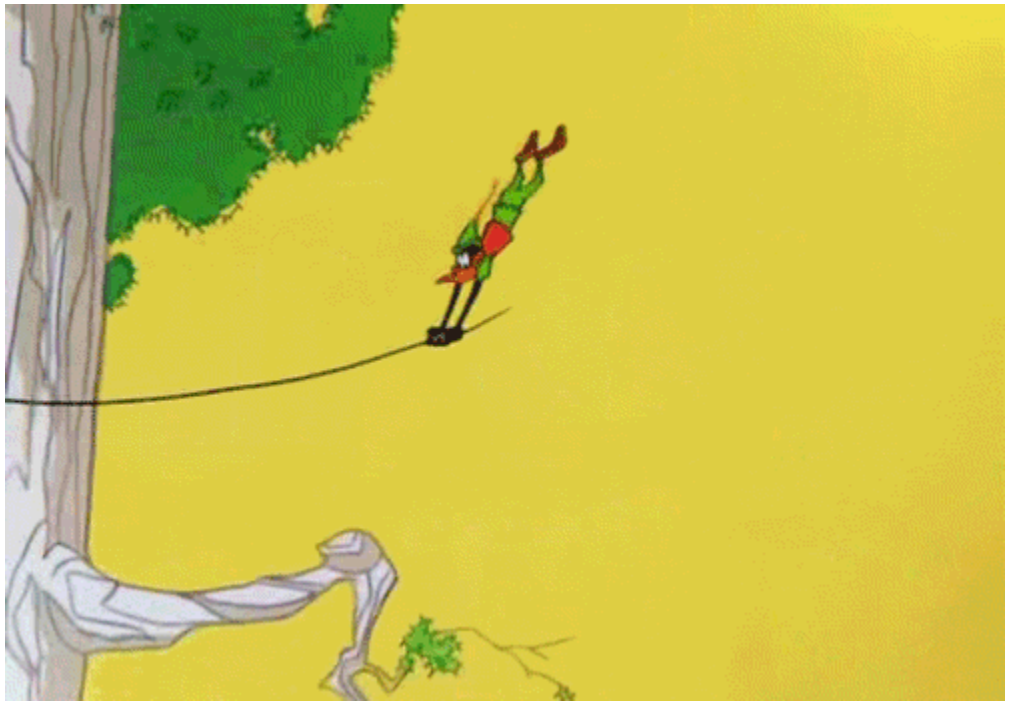
Remote Procedure Calls, Take 1

* CORBA
* COM/DCOM/COM+

Remote Procedure Calls, Take 2

* SOAP
* XML-RPC
* etc.

* “Web Services”



Remote Procedure Calls, Take 3

WCF

Service Oriented Architectures

* Enterprise Service Buses (ESB) and the like
* Various technologies, depending upon your tech stack…

CQRS

Along with Event Sourcing, REST, etc.

Mono-Rails

* Twitter, for example

The Hype Cycle

Key Concepts

* Containment
* Trading Business Complexity for Infrastructure Complexity
* Automated Testing
* Real, Actual DevOps
* “Cloud”

Containment

* One Business Concern
  * Product Catalog
  * Order Entry
  * Payment Processing
  * Order Workflow
  * Shipping
  * …

Domain Drive Design (DDD)

"Bounded Contexts" are how you determine the concerns for each service

Data Storage

* Separate Data Store per Service

Reduced Business Complexity

* Encapsulate Business Logic
* Loose Coupling
* Easier to Test

Aside – Can't I do this without Microservices?

* Absolutely!
* It just requires strong discipline and static analysis tools…

Increased Infrastructure Complexity

* More Infrastructure
  * HTTP/REST
  * Queues
  * Message Brokers
  * etc.

A Word of Caution

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable"

- Leslie Lamport

Fallacies of Distributed Systems

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

http://www.rgoarchitects.com/Files/fallacies.pdf

Keeping Things Running

* Centralized Logging
* Proactive Monitoring

The “More Everything” Plan

* More Hardware
* More Latency
* More Storage

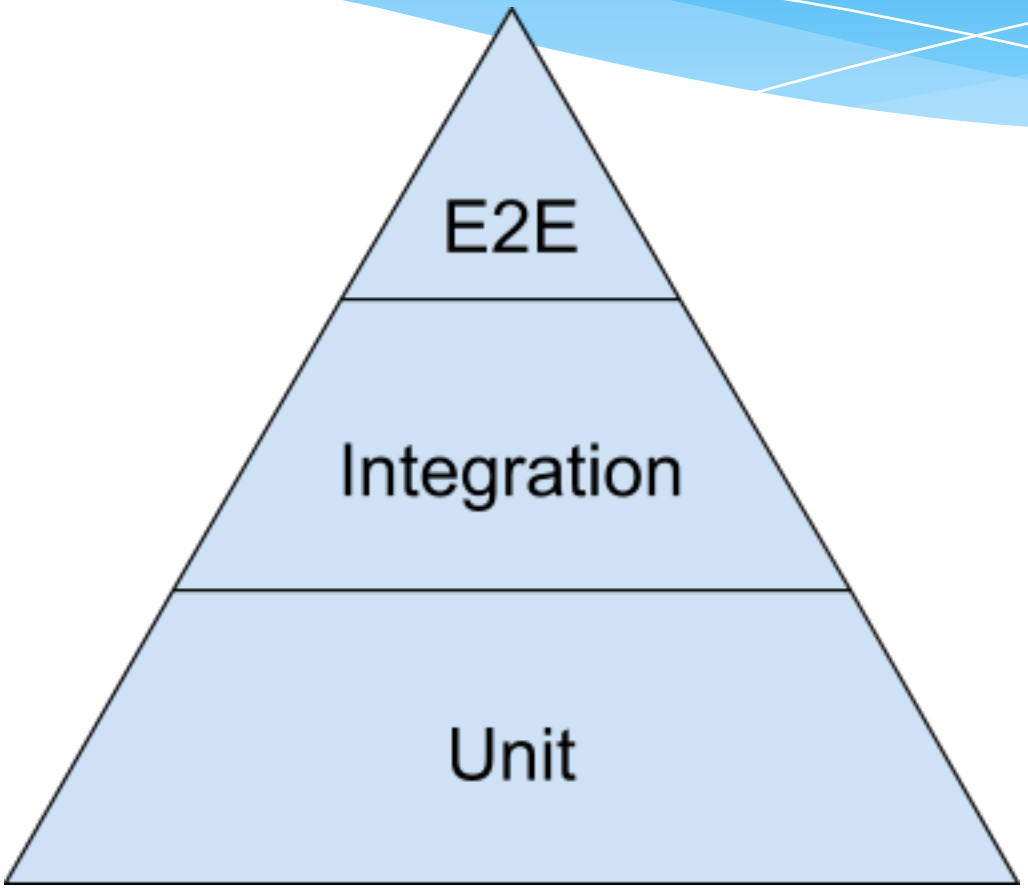
Waste?

* No – that's not what you are optimizing for…
* This is the tradeoff you choose to make

Automated Testing

* Cover the known paths
* Free up to explore the unknown
* Minimize End-to-End testing

Testing Pyramid



Real DevOps

* All groups must participate as partners
  * Developers
  * Data People
  * Testers
  * SysAdmins
  * etc.

Deployments

* Automated
* Able to deploy only one service
* Robust
  * Think cattle, not pets

Support

* Nobody goes into the machines
* Support via Logs and Monitoring

Let it Burn

* Build for fast recovery
* Handle exceptions when you reasonably can
* Allow to crash and recover when you can't

To the Cloud!

Cloud and Microservices

are like

Peanut Butter and Chocolate

Why Would You?

Build Fast

* Once over initial hurdles, things can move fast

Reliability

* Being able to reason about the pieces leads to more robust software (aka less bugs)

Maintainability

* Smaller parts are easier to maintain
* Worst Case – blow a part away and replace with a new service

Scalability?

* Can break parts out to different machines
* Latency could cause problems

Cloud Deployment

* Can make use of PaaS/Containers

Experimentation

* Easy to add a service
* Low cost if it doesn’t work out

Why Wouldn’t You?

Not a Lot of Business Complexity

* Straightforward business logic

Don’t Understand the Domain

* You still don’t understand all of the nuances
* Can’t figure out where to split

“Monolith First”

* Martin Fowler: http://martinfowler.com/bliki/MonolithFirst.html

Not Long-Lived

* Solves a short term problem
* Cheaper to build and rebuild
* IT Systems

Can’t “Do DevOps”

* Wrong culture
* No Management Support

~~Lack of~~ Talent

* Cuts both ways:
  * In Microservices:

  Team members need to understand distributed systems
  * In “Monolith”:

  Team members need discipline to keep things apart

Installing On-Premises

* Because:
  * Business Model
  * “That’s the way we’ve always done it”
  * Customers Demand It

Quick to Market

* Might be able to build up an MVP faster
* BUT:
  * Might move faster once you need to pivot if you have Microservices

It’s a Continuum…

Techniques

Build Loosely Coupled

* Well-decoupled Packages
* Separate Data Stores per concern
  * Could just be schemas
  * As long as you don't allow cross-schema queries
* Hide behind well-defined Interface Contracts

Event-Driven Architectures

* Everything that happens is an Event
* Immutable
* Append-Only

Message Bus

* RabbitMQ
* ZeroMQ
* Kafka
* etc.

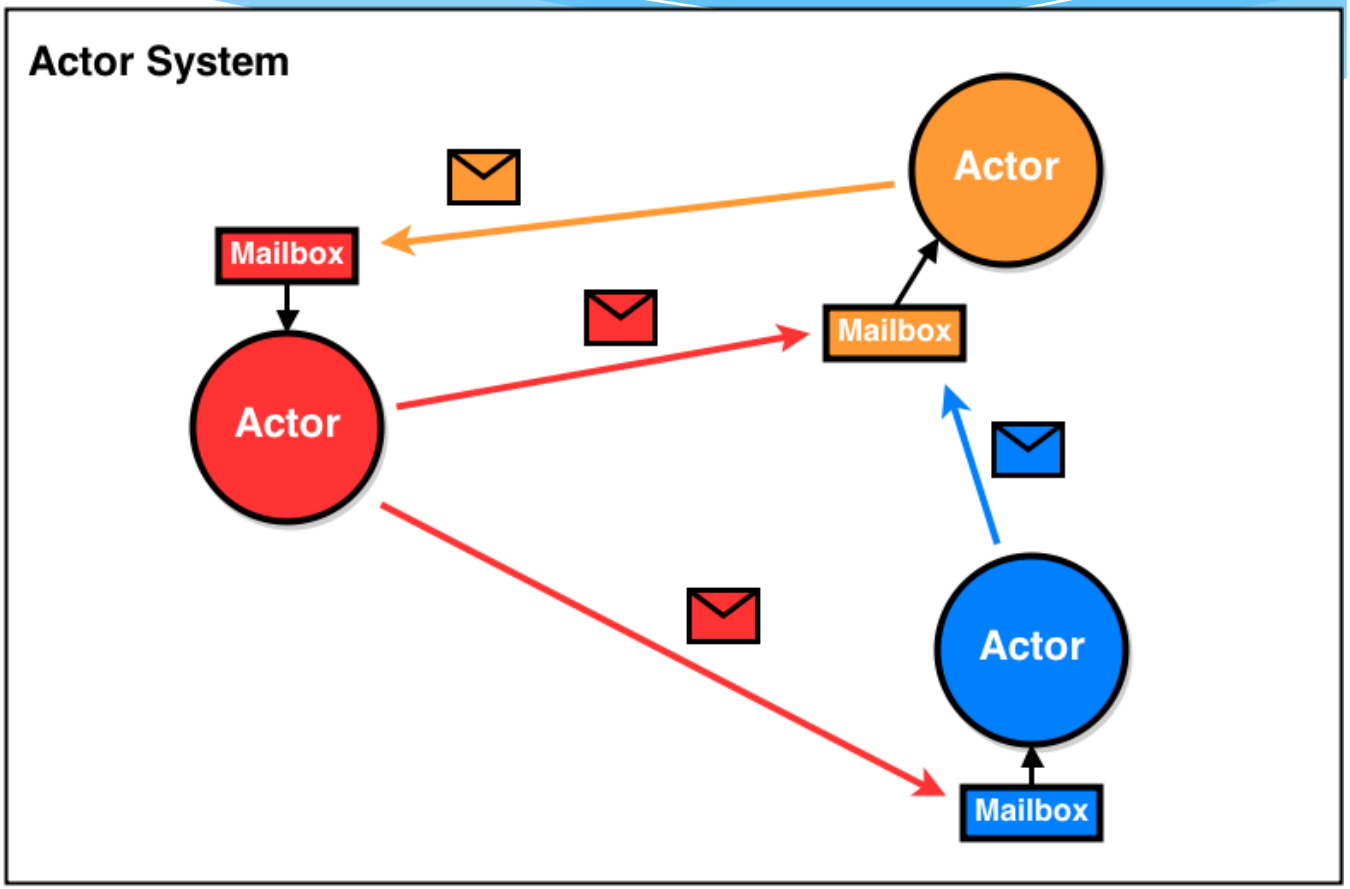
DO NOT USE AN ACTUAL ESB!

Build Lightweight

* In Ruby – Sinatra
* In Python – Flask
* In C# – Nancy
* In Java – Good luck!

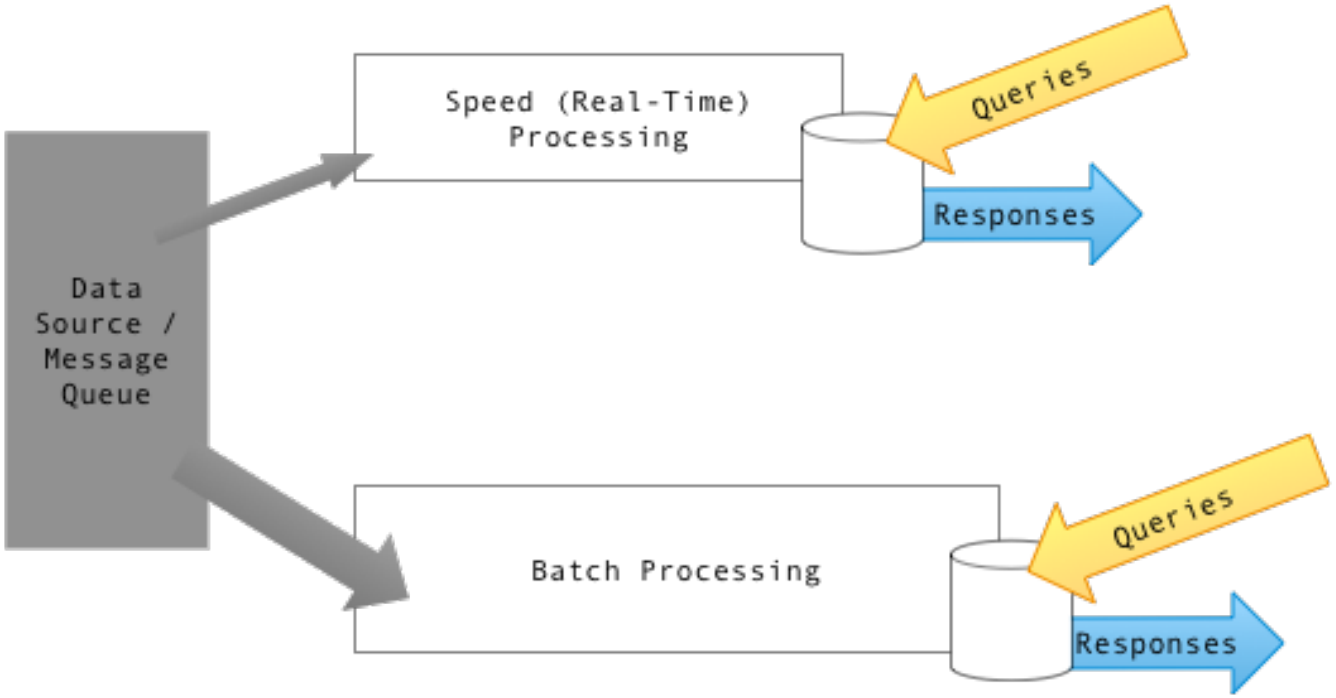
Actor Model

* Erlang
* Akka
* Orleans



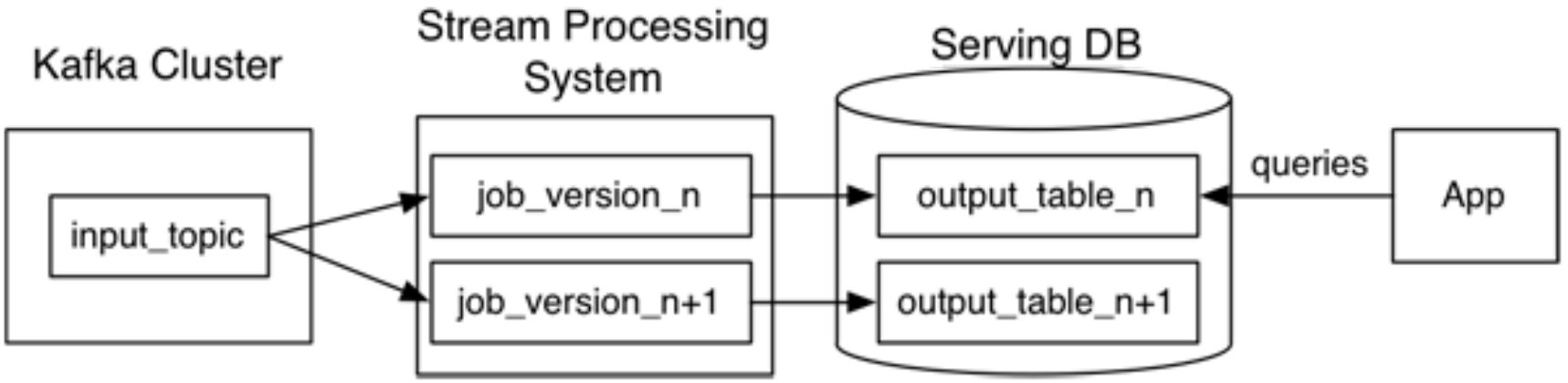
Source: http://blog.scottlogic.com/2014/08/15/using-akka-and-scala-to-render-a-mandelbrot-set.html

Lambda Architecture



Source: https://commons.wikimedia.org/wiki/File:Diagram_of_Lambda_Architecture_(generic).png

Kappa Architecture



Source: http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html

Break into Services

* The 13th Step…

Deployment

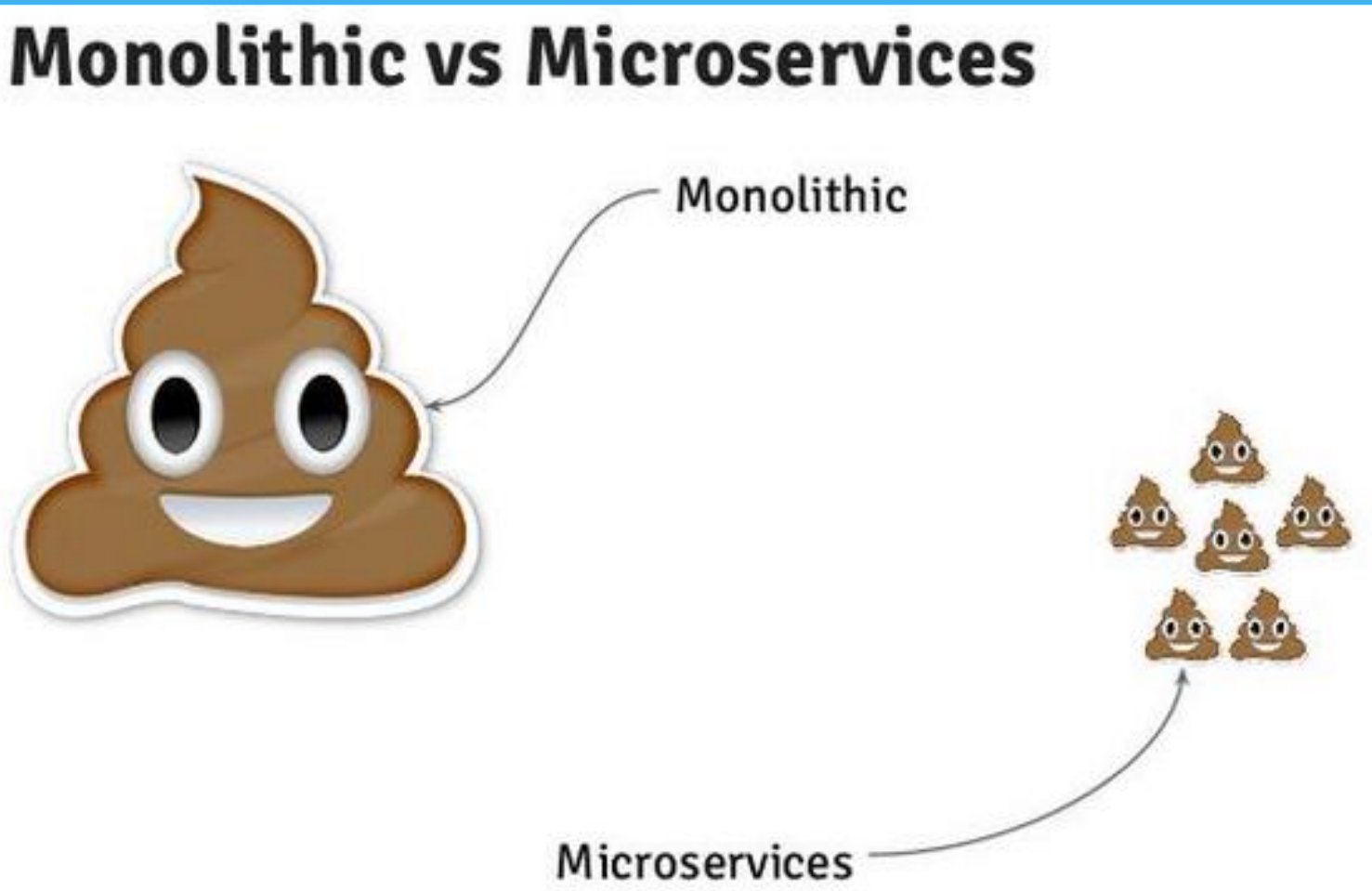
Every Service gets it's own box*

* *Where box means Server, VM, Container…*

Remember…

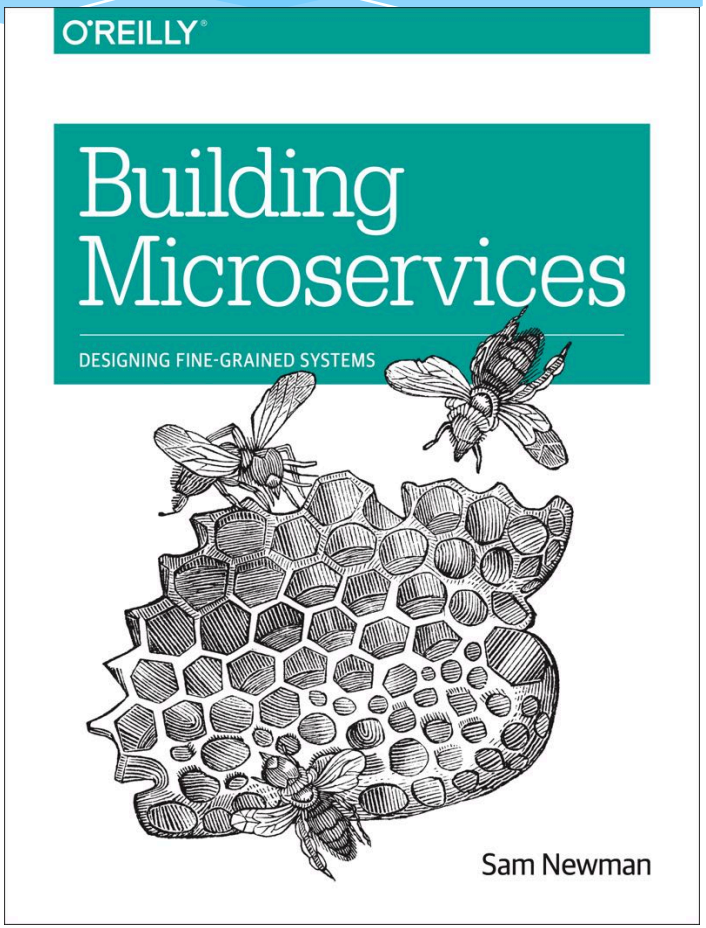
* Every service must be able to be deployed independently

Summary



Sam Newman

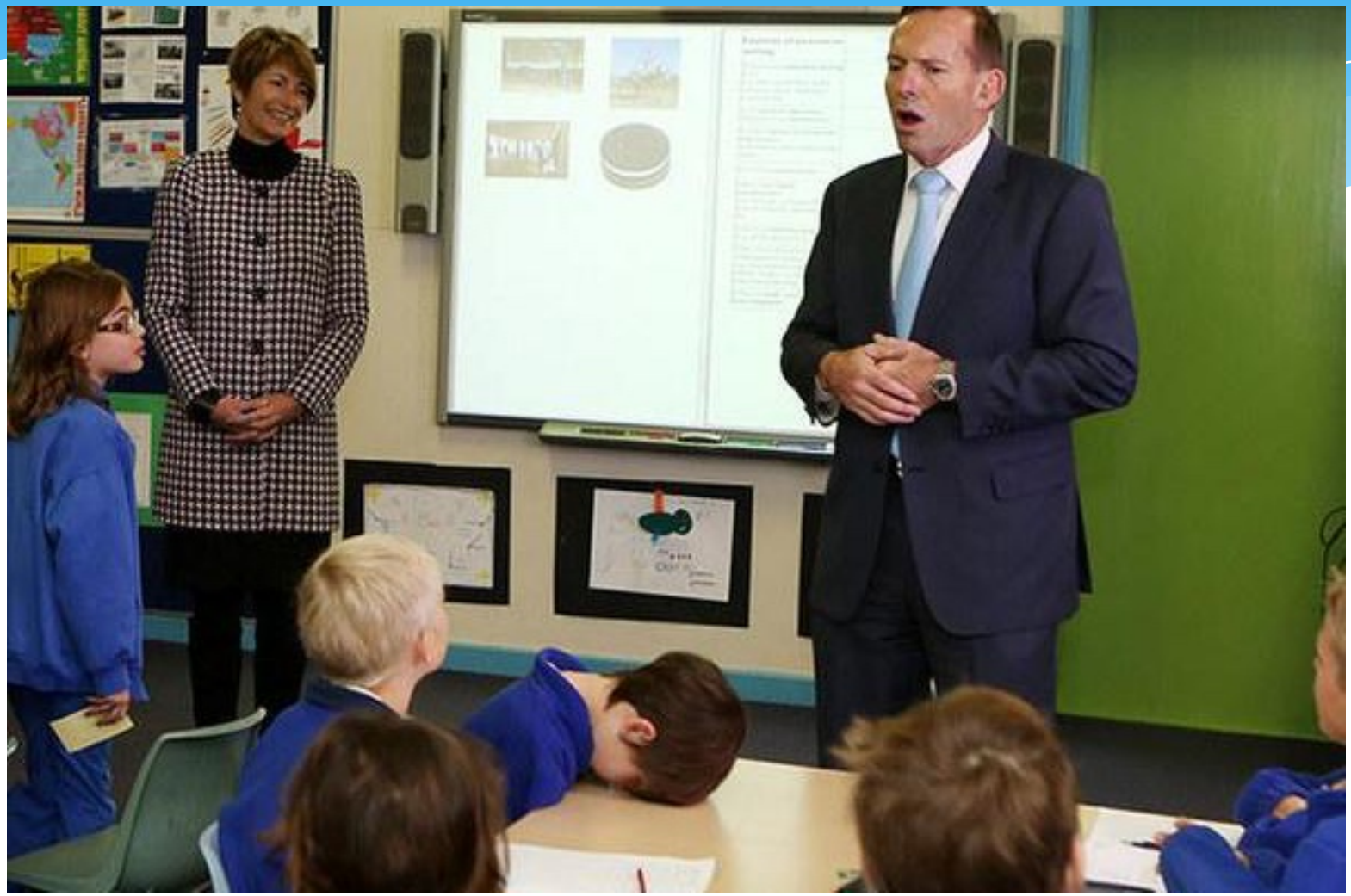
* Video: https://www.youtube.com/watch?v=OTSlg7_y3bA
* Book: http://samnewman.io/books/building_microservices/



12 Factor App

http://12factor.net/

Questions?



Thank You!

Eric Kepes

@ekepes
http://erickepes.com
eric@kepes.net