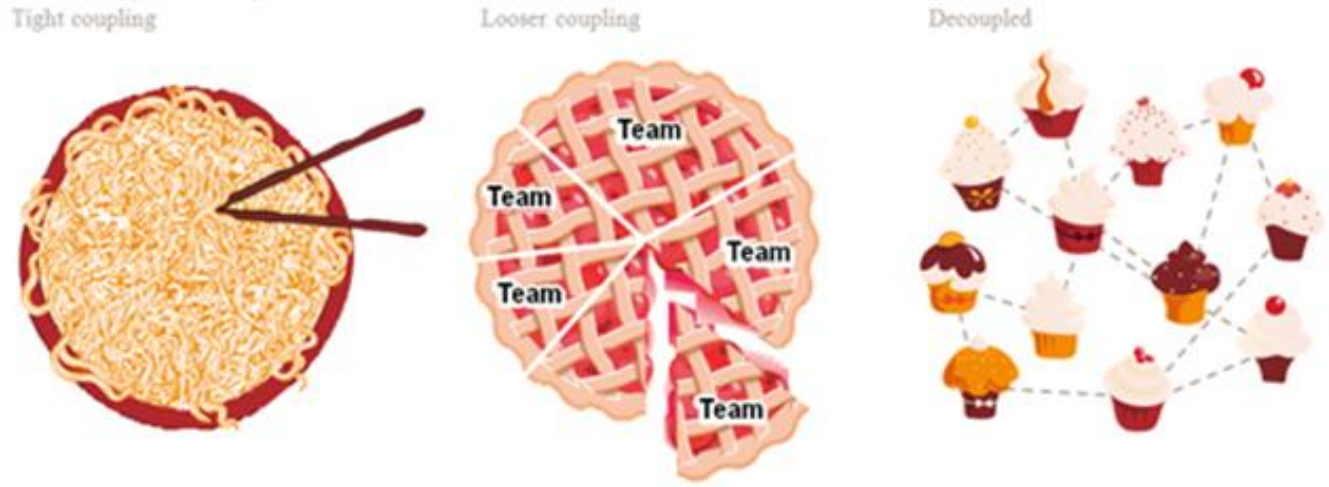
Microservices for a Non-Technical Audience

Pamela Toman

January 2017

Microservices are decoupled units that communicate through network APIs

- Microservices apply two core ideas:
  - "**Decoupling**": no shared knowledge between units
  - "**Networked communication**": all inter-unit communication happens over web APIs
- Microservice architectures are suites of independently deployable services
- People often call the alternative "a monolith"



(source)

Spoiler:

Small teams won't benefit from microservices. Large teams might.

- Microservices are costly
- The benefits emerge when:
  (a) the team is large, and
  (b) everyone agrees on well-defined separable units of work

- Recommendations (spoiler):
  - Follow best-practices (moderated by need for speed & maintainability);
    wait until the monolithic code base naturally suggests the need for microservices
  - A humorous rule-of-thumb:
    "Divide the number of full-time backend engineers by 5
    to get the ideal number of microservices"

Decoupling is always beneficial

- "Coupling" is the amount of dependency between two entities

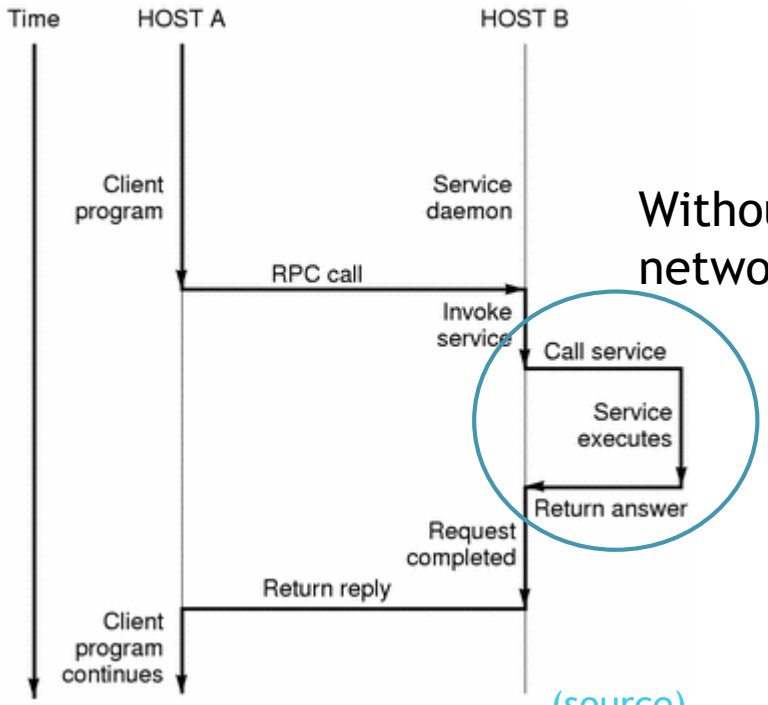
(source)

(source)

- Strongly coupled systems are expensive and hard to maintain
- To "decouple" components, we redesign to stop making assumptions about how other entities work internally
- *CS principles at play*:
  separation of concerns, modular design, abstraction, encapsulation

Forcing all communication to use a network has ambiguous benefits

RPC = “Remote Procedure Call” (networked communication)



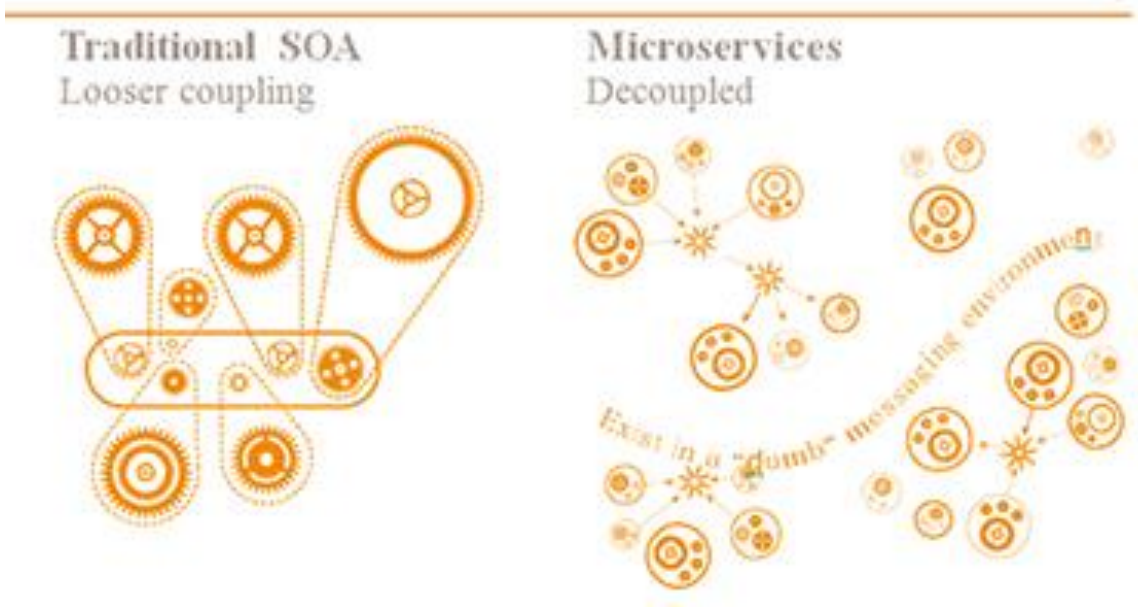
Without network

(source)

- In a monolith, a series of *procedure calls* (functions) perform small amounts of work and return the result
  - Calls are processed in real-time
  - Code can run without a network
  - All code uses the same language; the code regularly & automatically self-checks function signatures/data types
  - Full history (“what called what?”) is available locally during debugging
- Microservices encapsulate all this code, define a text-based API, and require querying-folks to wrap their requests for the network & send it

Visualizing microservices

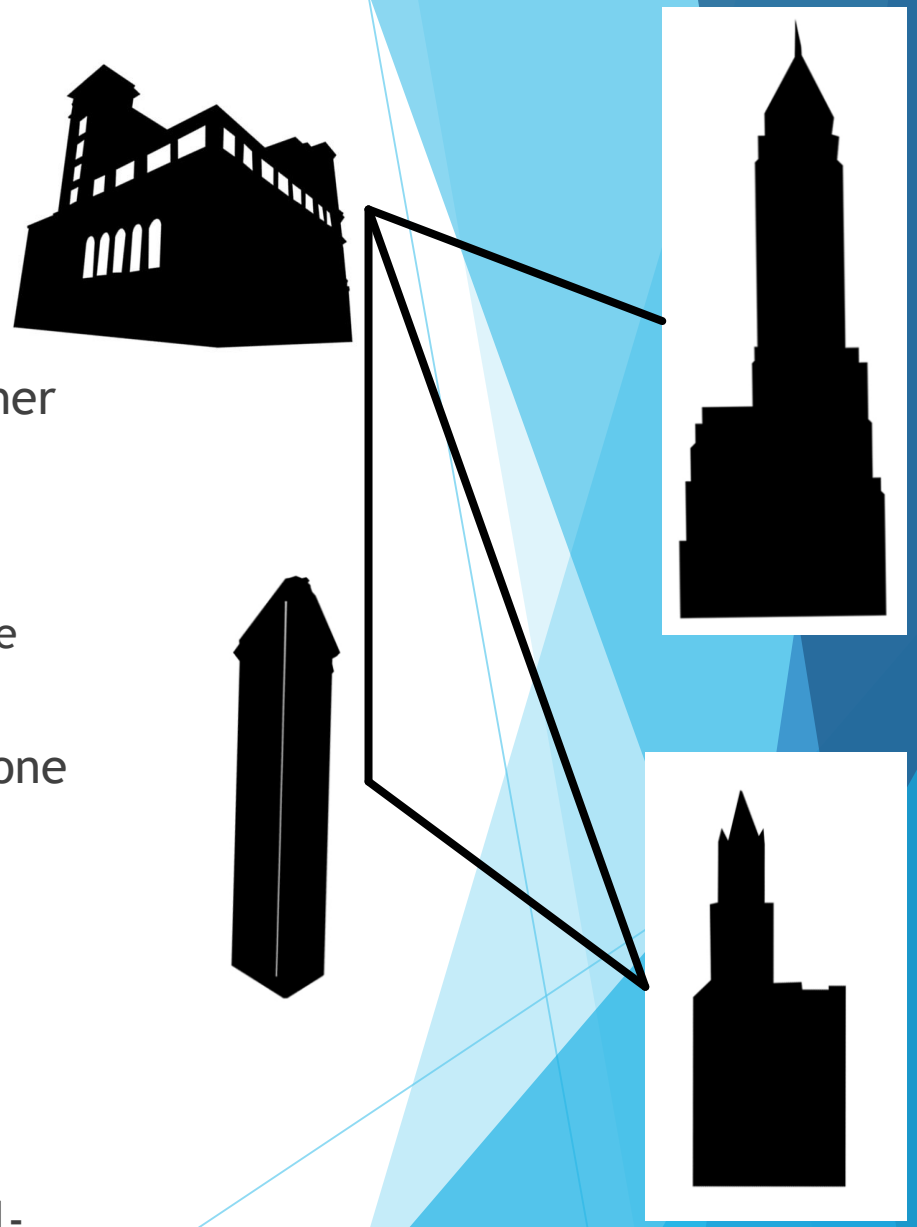
- Traditional architectures have loosely coupled parts,
  which communicate with each other at exposed touchpoints
- Microservices have multiple instantiations of very small units,
  which communicate with any other units they want in a very naïve way



(source)

A loose coupling analogy: phone numbers

- A traditional architecture is like a group of people sitting together
  - People have roles
  - But sometimes people take on another role
  - And sometimes people rely on someone's internal, incomplete state (say, notes) -- and everyone knows this is not ideal
- A microservices architecture has separate buildings, each with one external phone number
  - Each building has a single purpose
  - Each phone call exchanges only needed information
  - A call triggers a flurry of work
  - Once the answer is available, the building calls back with it
- Microservices succeed when the interplay of work units are well-defined – and in those cases, extending for higher demand is easy



(source)

Technical and organizational trade-offs

Microservices increase the friction for cheating on best practices

- Microservice structure encourages following these universal best practices:
  - Loose coupling / proper partitioning (resiliency to failure)
  - No leaking of implementation details
  - Careful validation of all inputs
  - Simple, narrow, flat APIs between components
  - Independence of parts through API versioning

- It hurts a lot more when microservices don't follow these practices

- Following these practices always comes at an initial cost (-3x?) and offers future savings (+5x?) in time and money; the project manager and architect need to balance the trade-off

(source)

Microservices introduce unique **technical** benefits & downsides

(source)

Benefits

- Teams *fully* own their products (deployment, scaling, performance monitoring, error handling, database, migrating to new libraries or languages, …)
- Teams *must* code to the possibility that dependencies are unavailable
- Project can scale better on the same number of machines

Downsides

- Teams write more code (e.g., tests for interface, backup plans for unavailable services)
- It is harder to debug problems that span microservices
- Unless the services are well-designed, the code will be slow
- Refactoring is more painful
- Deployment & monitoring workload skyrockets (~10x ?)

Ambiguous: Each microservice can be written in a different language.


Honest Status Page
@honest_update
Follow
We replaced our monolith with micro services so that every outage could be more like a murder mystery.
RETWEETS
1,987
LIKES
1,435
1:10 AM - 8 Oct 2015

(source)

Microservices introduce unique **organizational** benefits and downsides

Benefits

- Teams *fully* own their products (deployment, scaling, performance monitoring, error handling, database, migrating to new libraries or languages, …)
- Encourages the architecture to be carefully considered, clarified and made visible; links get explicitly discussed in meetings; no man's land of responsibility disappears
- Especially helpful if the team is distributed – communicating through APIs may be more effective than calls/Slack/in person visits

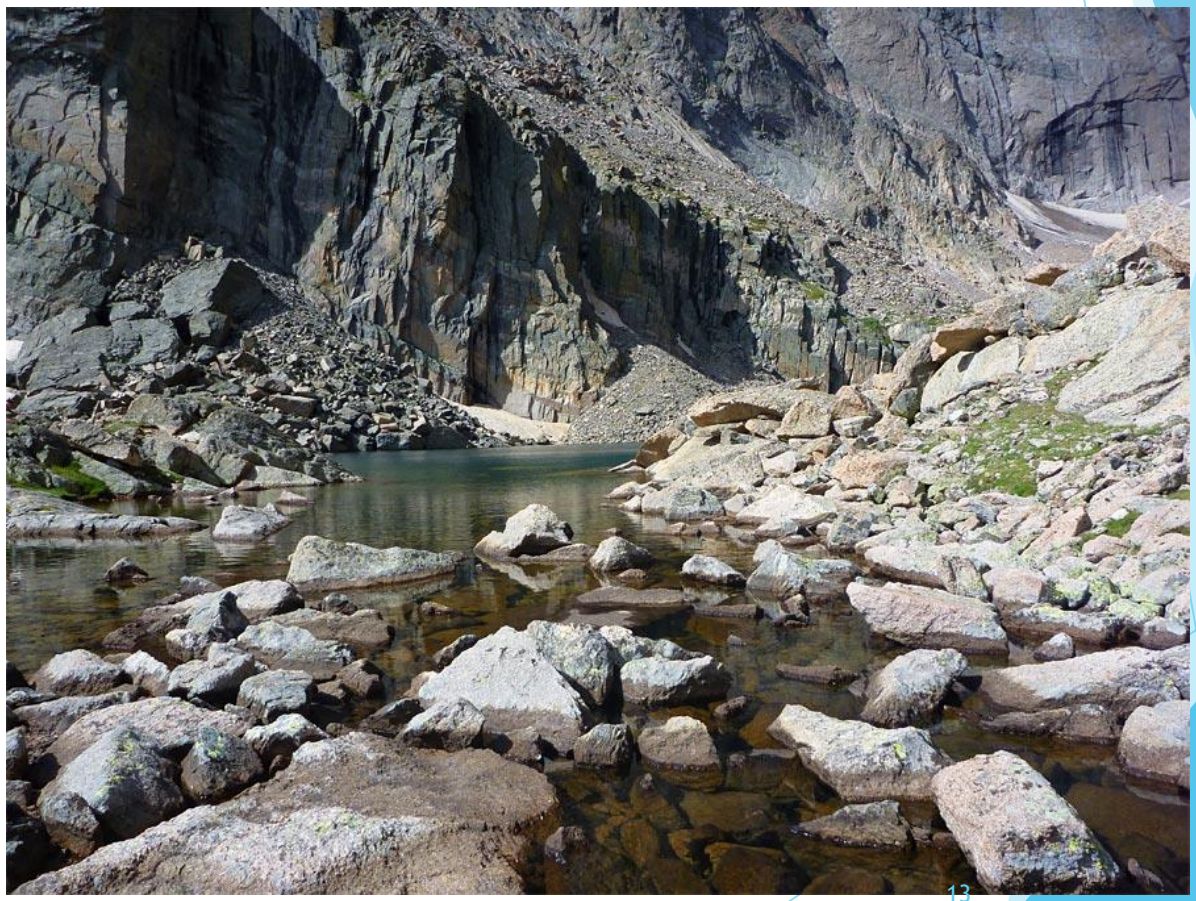
Downsides

- Microservice architectures have higher fixed costs
- Each team size must be large enough to not allow a single point of failure

(source)

Ideal microservices fit the organization & needs

- Microservices are a way to **clearly distribute ownership & autonomy**
- Teams should be around 4-6 people
- Microservices should be standalone pieces that emerge from a working system

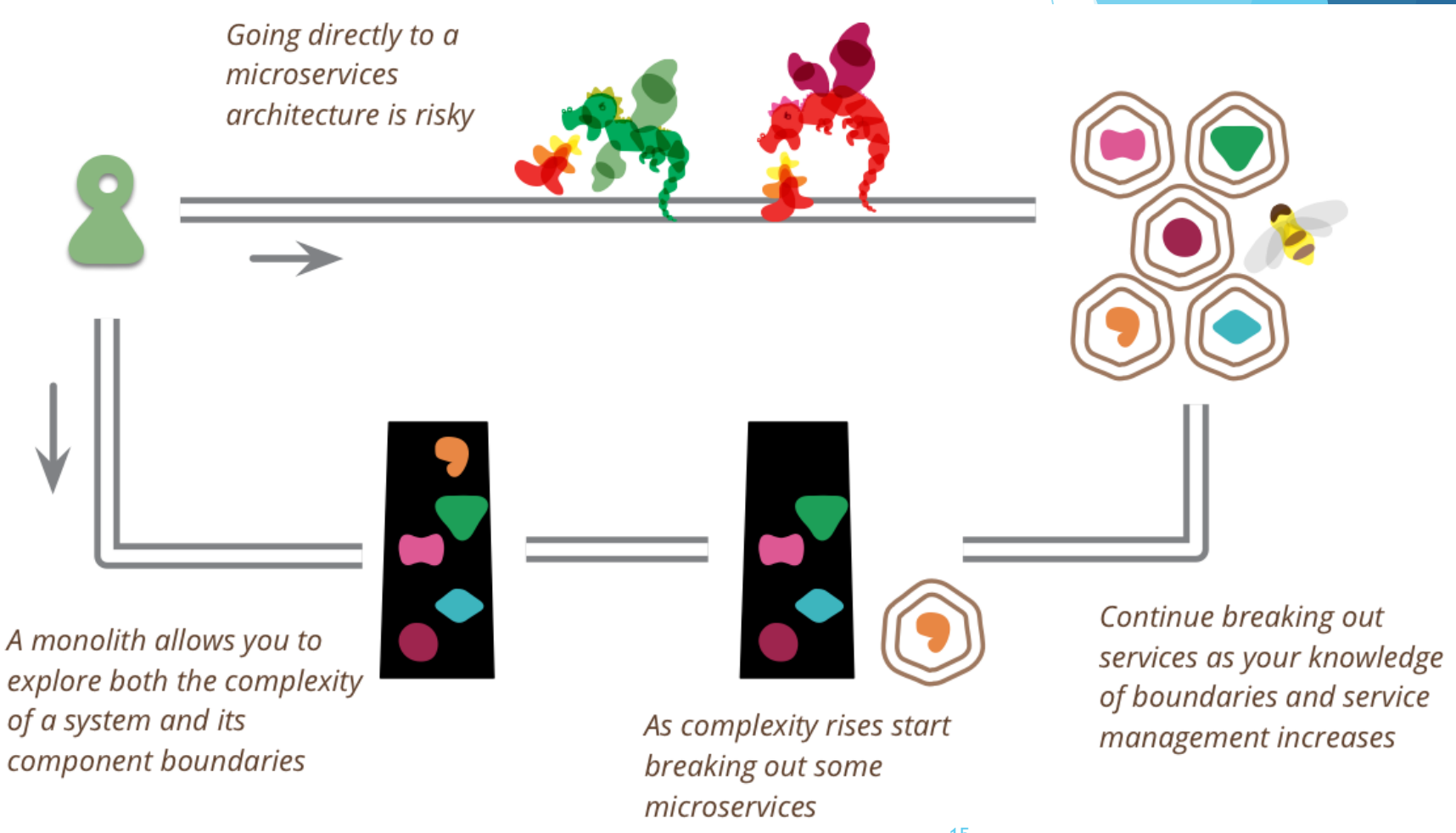


(source)

Knowing when to pull the trigger

Where do microservices come from?

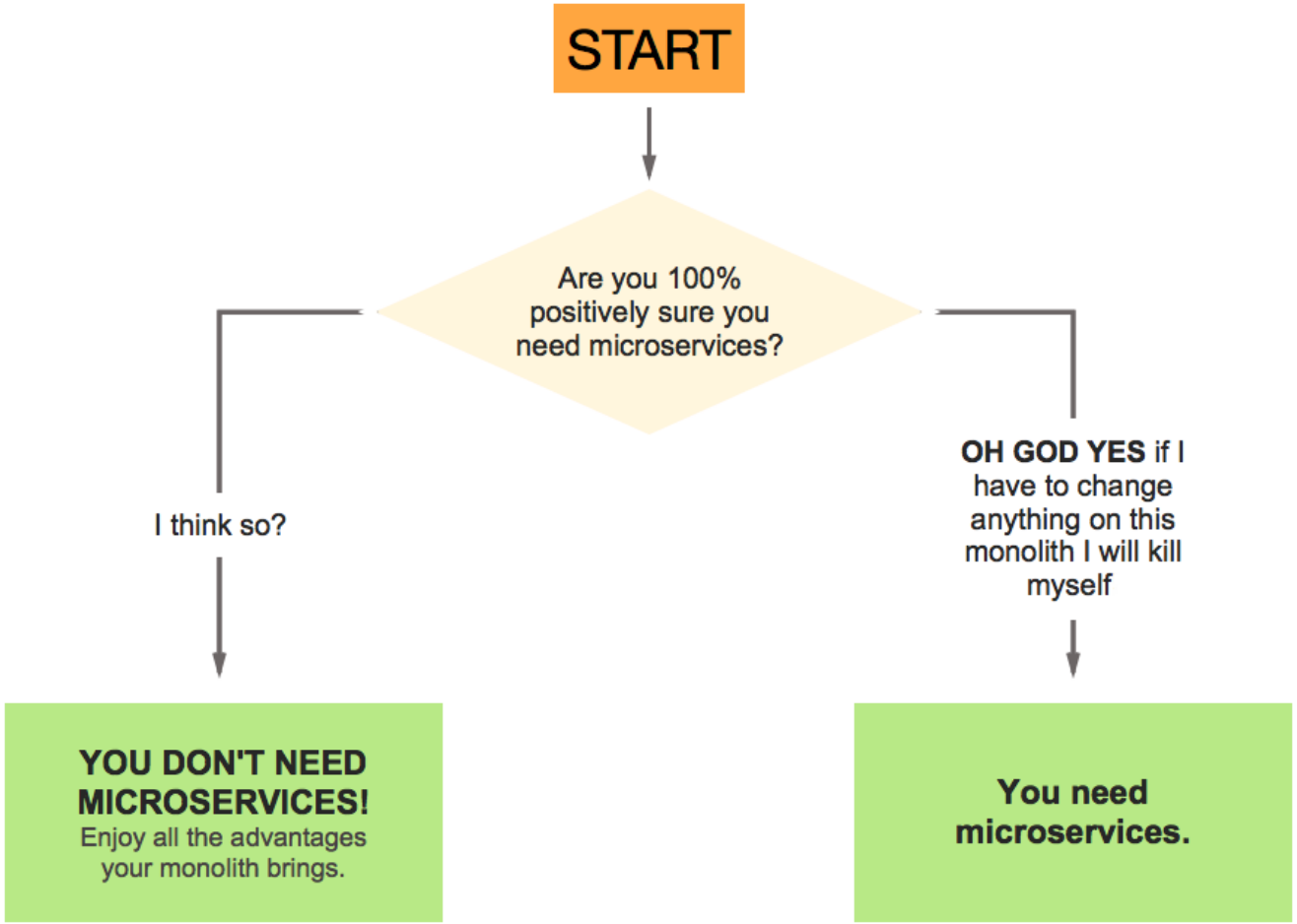
- Anecdotally….
- Almost all successful microservice stories started with a “too big” monolith
  Netflix, SoundCloud, Twitter, …
- Starting from scratch with microservices leads to trouble
  a graveyard of failed startups…



(source)

Who needs microservices?

(from an October 2015 blog)



(source)

Final review:

Small teams won't benefit from microservices. Large teams might.

- Microservices apply two core ideas:
  - "**Decoupling**": no shared knowledge between units (always good!)
  - "**Networked communication**": all inter-unit communication happens over web APIs (ambiguous)
- Microservices facilitate unit-level complete ownership & autonomy for larger teams
- Microservices are costly for small teams and small products
- Recommendations:
  - Follow best-practices (moderated by need for speed & maintainability); wait until the monolithic code base naturally suggests the need for microservices
  - A humorous rule-of-thumb:
    "Divide the number of full-time backend engineers by 5 to get the ideal number of microservices"