Microservices the chain of calls and the radar of freedom

Davide Rossi

Webist 2018

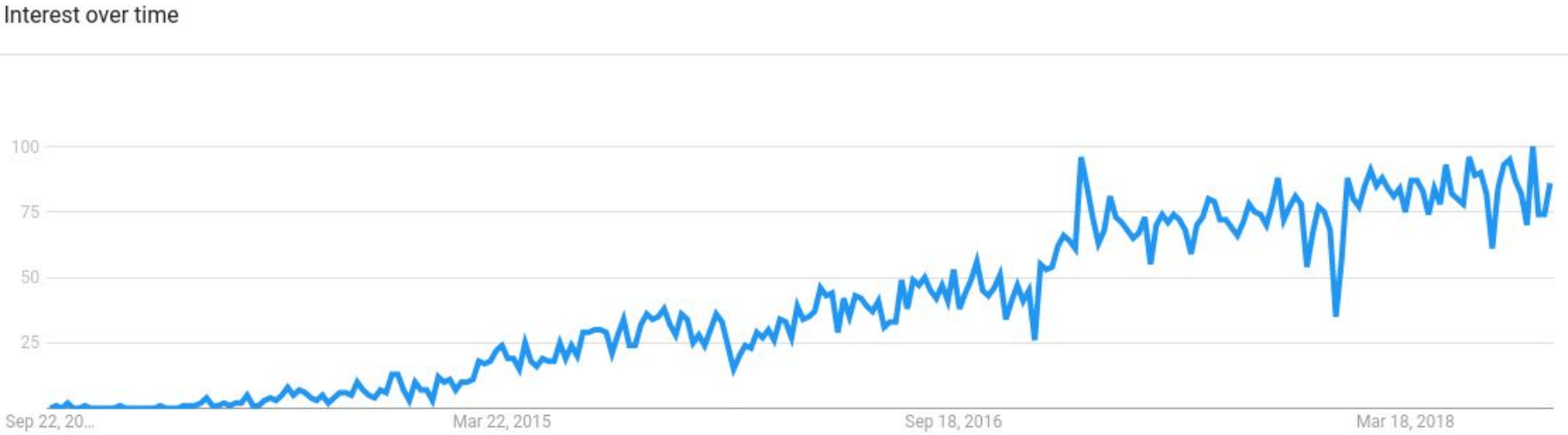
Foreword

This is a **position talk** that focuses **on the impact** of recent technical developments.

I am a research professional but this talk mostly focuses on technical aspects...

...or more precisely **on the perception** of these aspects more than on the technical details.

Microservices are hot



[Google trends]

The microservices dilemma

This is from a research professional point of view.

Which is the microservices research space?

What's difference with respect to other service-oriented architectures (e.g. E-SOA)? It is more about the product? About the process? About practical impact?

A lot of researchers squeezed their heads but apparently very little output has been produced.

Are microservices a topic in computer science research at all? Industry-academia dichotomy all over again?

Microservices success stories

From hero


Segment
ENGINEERING
Why Microservices Work For Us
Calvin French-Owen on December 15th 2015

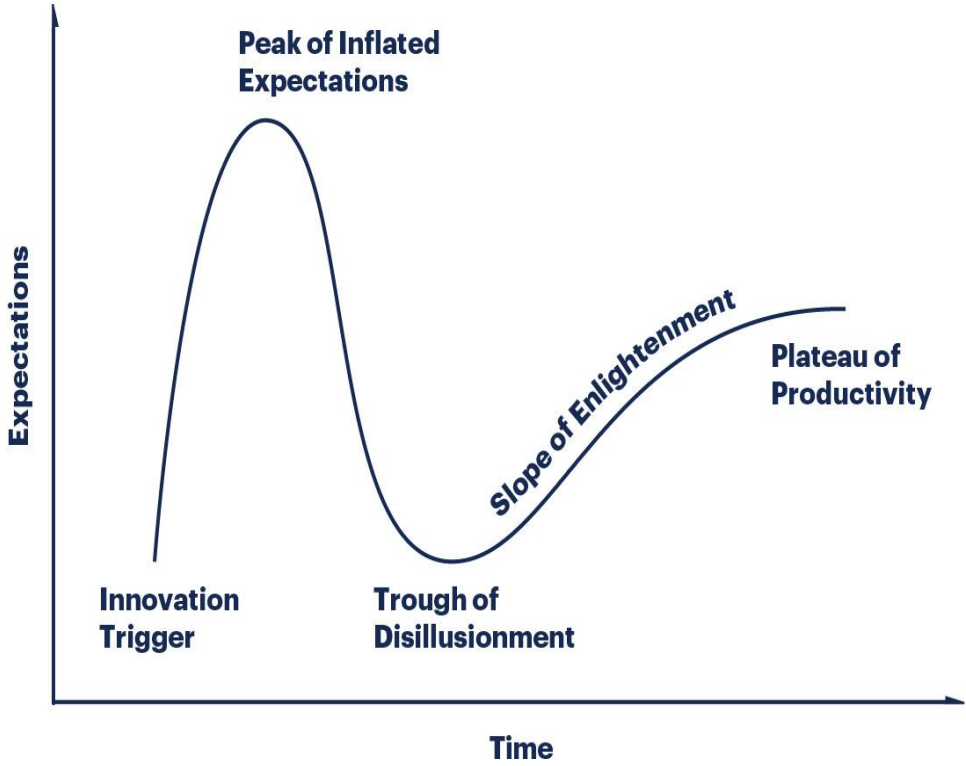
To zero


Segment
ENGINEERING
Why Microservices Work For Us
Calvin French-Owen on December 15th 2015

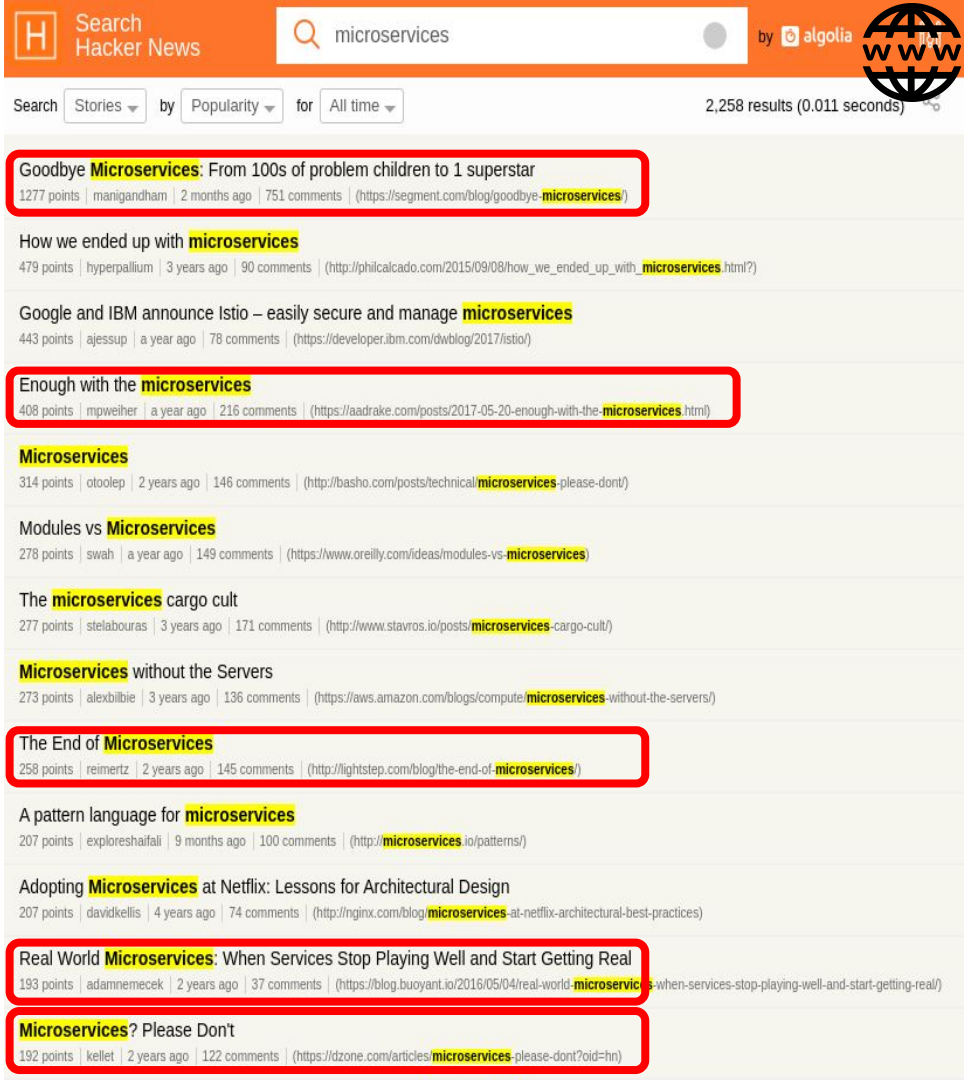

Segment
ENGINEERING
Goodbye Microservices: From 100s of problem children to 1 superstar
Alexandra Noonan on July 10th 2018

Living the hype cycle

Microservices have a place here. Somewhere.



Disillusionment?



What developers like about microservice?

A social media analysis

Freedom?

That's **design space freedom**.

Microservices are an **architectural style**.

An architectural style is about **imposing constraints** on the design on an architecture.

Constraints are enforced to improve software qualities: **non functional qualities** and internal qualities.

An architecture design space depends on these qualities.

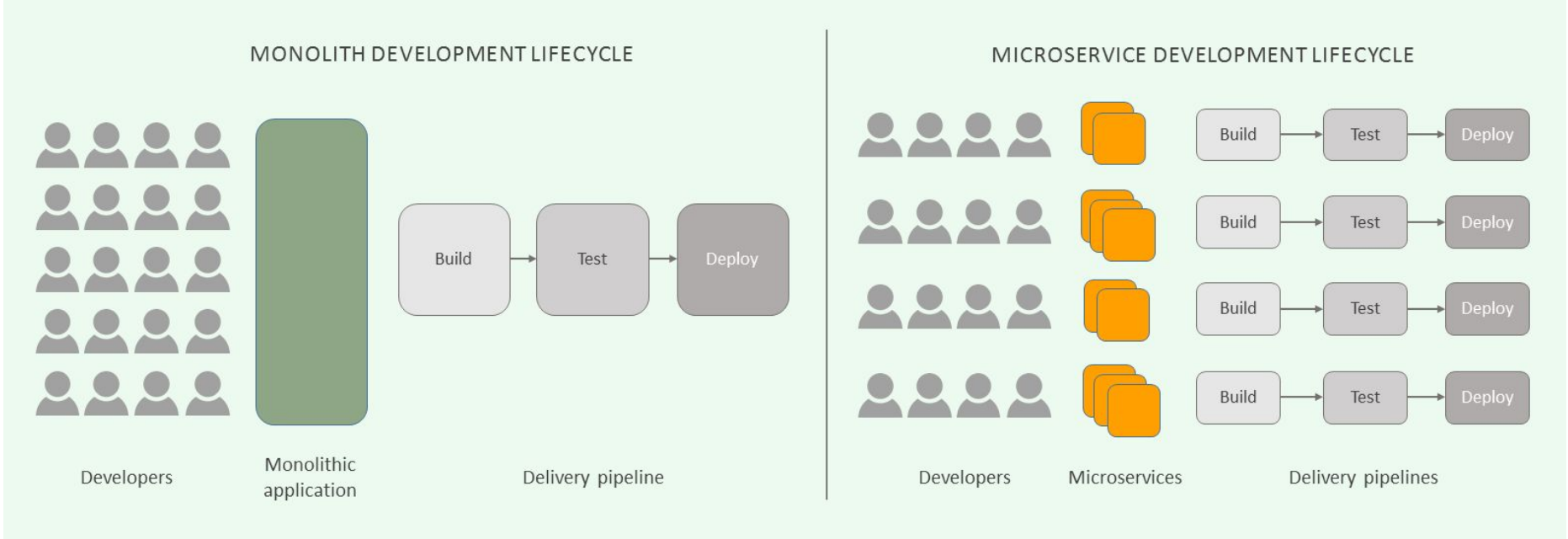
Microservices are different

In all declinations of a SOA services are loosely coupled, re-usable, autonomous, stateless, discoverable, composable, based on standards.

Microservices are designed to be **developed, deployed and scaled independently**.

Enablers: agile software development; virtualization/containers; evolution in data management.

Microservices pipelines



[Kristijan Arsov]

Microservices are distributed systems

Systems designed on top of a **microservices** architecture **are distributed systems**.

Even through the mist of containers and VMs.

We know something about distributed systems. This knowledge does not percolate so promptly towards people adopting microservices, though.

One of the things we know if that **distributed systems are hard**.

Microservices can be complex

The case for Netflix:

- Eureka
- Ribbon
- Hystrix
- Zuul
- Curator
- Astyanax
- Conductor
- Memcached
- …

Which in turn depend upon ZooKeeper, Servo, Cassandra, ...

Dimensions of freedom

Governance

Development

Language (polyglot programming)

Data management (polyglot persistence)

Platform/Infrastructure

Governance

Governance is about policies. Policies are pervasive and can touch almost every aspect of software development and operations.

Microservices bring the promise of **decentralized governance**: centralized governance is perceived as an overhead that should be avoided by supporting service-specific governance and intra-service contracts (which can be promoted by using patterns like tolerant reader and consumer-driven contracts).

Development

The microservices development pipeline translates to separate development projects and it is not unusual to have tenths if not hundreds of microservices in a single system. That calls for development methods with minimal overhead and agile programming is undeniably the best option.

So development freedom is about adopting different practices within an agile context. Most notably these practices could change between the projects of different microservices within the same application (this is really a sub-category of governance but since it receive significant interest from the microservices community it is presented separately).

Language (polyglot programming)

Polyglot programming has always been a strong selling point for microservices architectures.

Since each microservice is a separate product, it can be developed with the language perceived as the most fitting to solve the specific problems that microservice has to address. This could easily result in an application developed with an array of different languages.

Data management (polyglot persistence)

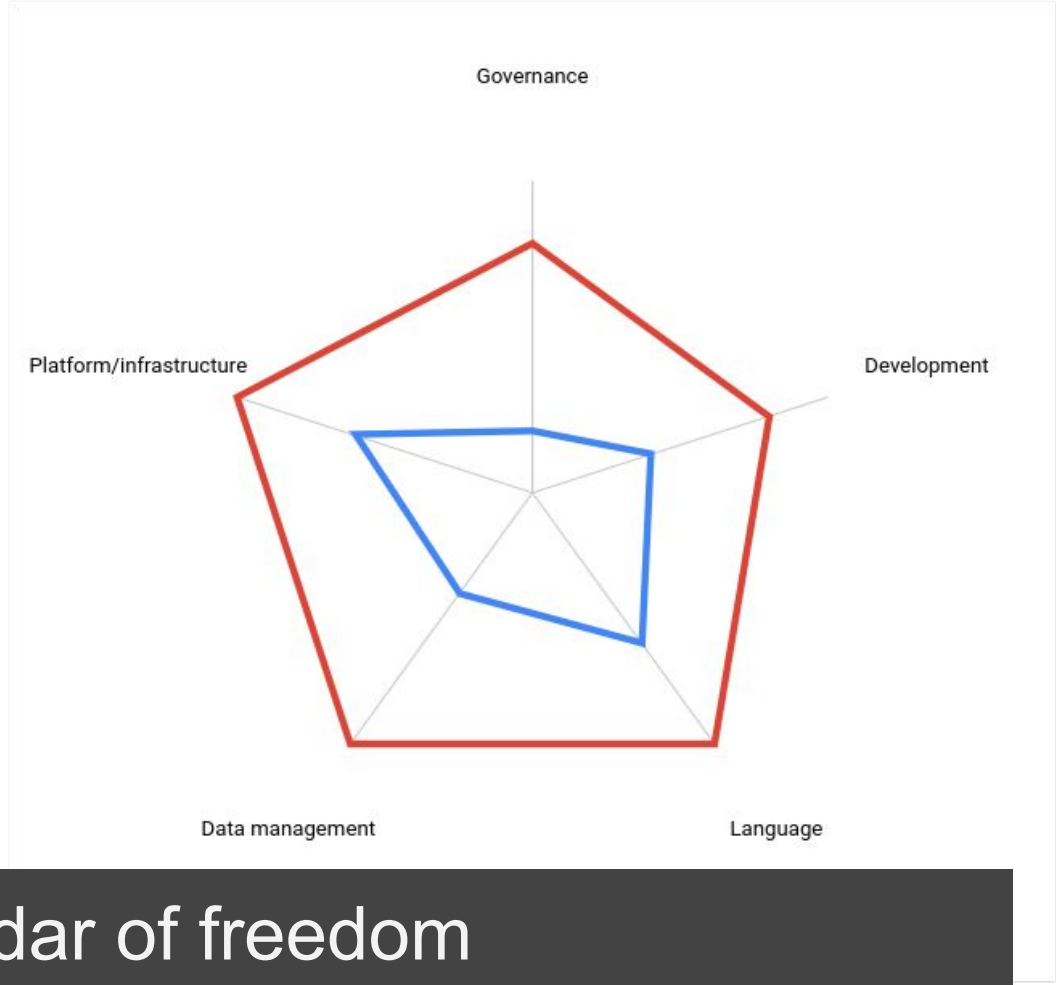
Just like polyglot programming, polyglot persistence too has always been linked with microservices.

A basic characteristics in SOA is that services should be autonomous and thus should take care of their own data. This is reflected in microservices at the conceptual level, where each microservice defines its own data conceptual model (typically inspired by the specific domain which the microservice is linked to, a practice also promoted by domain-driven design with bounded domains) but also at the implementation level, where it has the opportunity to select the most appropriate data storage solution.

Platform/Infrastructure

JEE and .NET provide well-defined ecosystems composed by libraries, frameworks and infrastructure services.

Microservices can choose *à la carte*: an array of options is available (which is also possible thanks to the wide diffusion of enterprise-grade open source software).



The radar of freedom

Perception vs reality

Governance: decentralized governance is indeed possible to a certain extent but several architectural choices have an immediate impact on policies that have to be enforced on all (or most of the) services in the system.

Development: not every software company has dozens of teams working on the same system.

Language: the need to share common libraries to assure a predictable behavior and the need to talk with specific infrastructure services can actually prevent polyglot programming from happening in practice.

Perception vs reality

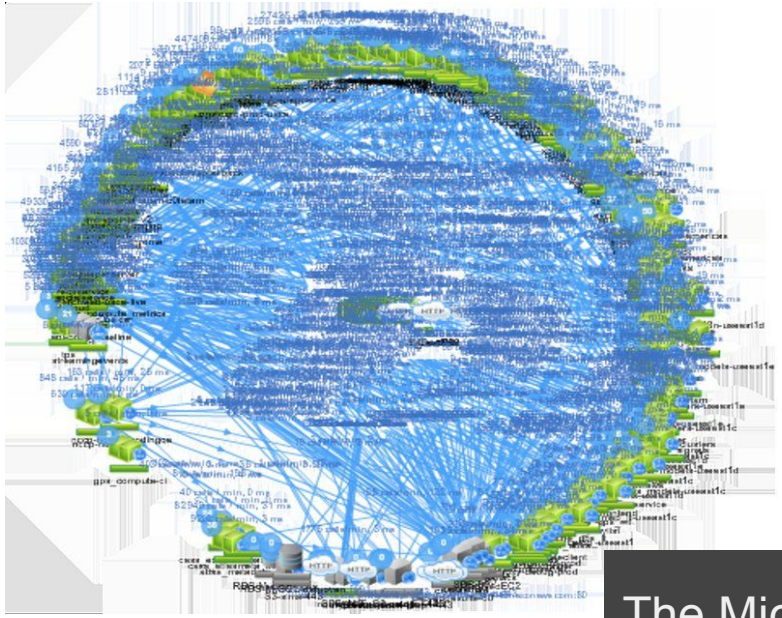
Data management: depending on the level of needed consistency strong constraints can be imposed on data management systems, the options could result to be much less than expected.

Platform/infrastructure: the decision to embrace a specific event-based framework rather than a specific message broker should not be made simply because of a preference in the programming model or languages supported but first and foremost because of the guarantees that this solution provides in terms of system qualities.

Let us see that in practice

1. Focus on a recurring problem.
2. Look at the possible solutions.
3. Analyze the solutions wrt quality dimensions.
4. Assess the impact on design space dimensions.

The chain of calls



The Microservices Death Star

The chain of calls

The chain of calls describes a **set of cascading logical dependencies** between microservices.

This does not necessarily translates into an actual sequence of direct invocations.

Option 1: actual invocations are performed.

Option 2: the interactions between dependent services are decoupled (usually by using asynchronous messaging supported by a message broker).

Quality dimensions

Consistency and **availability** are the most exemplary contrasting non-functional requirements that large, multi-user, distributed applications struggle with.

The **CAP theorem** (Gilbert and Lynch, 2002) states that in a partitionable system it is **not possible to achieve full consistency and maximum availability**.

Solutions to the chain of calls

- CC1. Perform direct invocation
  - CC1.1. Use a choreography-based approach
  - CC1.2. Use an orchestration-based approach
- CC2. Use messaging
  - CC2.1. Use a choreography-based or an orchestration-based approach
  - CC2.2. Use a DDD-inspired solution and actually avoid chaining microservices

About the domain-driven design-inspired solution

DDD: **command messages** are requests targeted to a domain, **domain events** signal relevant occurrence in a bounded domain (which usually correspond to a microservice).

Example: I want the product page to also show stock availability.
The most straightforward solution is to chain the Products microservice and the Inventory microservice (with or without an orchestrator, using sync or async messages).
DDD: when a the stock availability of a product changes Inventory raises a domain event. The microservice that composes the product page information listen to these events and updates its local copy of the availability.

Direct invocation: availability viewpoint

If the average chain size is N and the average availability of each service is A, the overall availability of the system cannot be more than N^A.

For example, if the average availability for the services is 99.999% (also known as five-nines, a measure usually perceived as very good for a real-world system) and the average chain length is 5, the resulting availability will be 99.995%. That means an increase in downtime from 5 minutes 15 seconds per year to 26 minutes 17 seconds per year.

Direct invocation: availability viewpoint

In IP-based networks **a crashed process is indistinguishable from a slow one**.

Usual approach: **timeouts**.

In presence of a chain of calls setting reasonable timeouts is difficult.

Current practice for microservices: **short timeouts**, **retries** (when in the presence of idempotent calls), **aggressive restart** of erratic/slow services (need to dialog with monitoring, message routing, and service hosting infrastructure services).

Duration of timeouts, number and frequency of retries, when a service has to be restarted are all heuristic-based decisions.

Direct invocation: availability viewpoint

Basic mitigations: **exponential backoff** and **back-pressure** for retries.

Circuit breaker (Nygard, 2018) is a pattern vastly employed to improve stability and resiliency in microservices architectures in the presence of direct service-to-service invocation.

Bulkhead (Nygard, 2018) is a pattern that suggests to partition service instances into different groups, based on consumer load and availability requirements in order to avoid the risk for a troubling connection to starve other concurrent workloads.

Direct invocation: availability viewpoint

It is not reasonable that all microservices independently implement these mitigations.

Option: libraries (e.g. Netflix's **Hystrix**, Twitter's **Finagle**).

Option: out-of-process proxy: **Sidecar** pattern (Burns and Oppenheimer, 2016).

A **service mesh** is based on this approach.

Use messaging: availability viewpoint

First of all we need a **messaging infrastructure**.

A more subtle aspect is related to **testing**: testing an event-driven system, while perfectly possible, is far from trivial: the test suite for a single service should also touch aspects related to the handling of events and test dummies, like mockups, **must include also non-directly coupled dependencies** like those generating or consuming events. While there are best practices to deal with these (and other) problem, testing these systems is difficult and requires specific discipline.

Availability viewpoint - summary

Use mitigation strategies for **direct invocations**
- **In-process** approach: libraries
  - **Language** is constrained
  - **Governance** is severely constrained
  - **Platform/Infrastructure** is constrained
- **Out-of-process** approach: sidecar/service mesh
  - **Platform/Infrastructure** is severely constrained
  - **Governance** is constrained

Use **messaging**
- **Platform/Infrastructure** is severely constrained
- **Data management** is severely constrained

Direct invocation: consistency viewpoint

In general we are dealing with a **distributed transaction**.

TPC is not an option, microservices-based solutions, for the most part, adopted ad hoc solutions, also known as **feral concurrency control** (Bailis at al., 2015).

The use of **long running compensating transactions** is starting to get traction (a.k.a. Distributed SAGA).

Long running compensating transactions

Choreography approach: the state of the transaction is a **distributed state**, in case of failures its consistency must be ensured (something that can be achieved using a robust distributed logging infrastructure). This also means that there are problems with **visibility** and **monitoring**.

Orchestration approach: the orchestrator, usually called the coordinator in this context, should **not be a single point of failure** and should be **highly available**.

Use messaging: consistency viewpoint

The data management needs of Web 2.0 companies shifted the focus from SQL and ACID to **NoSQL** and **BASE** (Pritchett, 2008).

With microservices it is usual to look for trade-offs in which a price is paid in terms of consistency in order to achieve better availability → the raise of **eventual consistency**.

From a chain of calls point of view this means that when services reads data they could be exposed to a **soft state** (i.e. the value that is read has not still be reconciled with the last updated value) whereas when services write data, **reconciliation** mechanism have to be adopted to guarantee eventual consistency.

Use messaging: consistency viewpoint

To guarantee eventual consistency with DDD-inspired approaches, **database update** and the generation of the **domain event** in the inventory microservice have to be **atomic**.

Easier solution: let the local database and the message queue participate in a multi-party atomic transaction (which is not a distributed one). This, however, **requires that the message broker supports atomic transactions (only a few do), and the same applies to the database (most NoSQL databases do not)**. **Other solutions** do exist but are complex, brittle, need message deduplication support from listeners and, of course, still need a transactional database (or a transactional message queue).

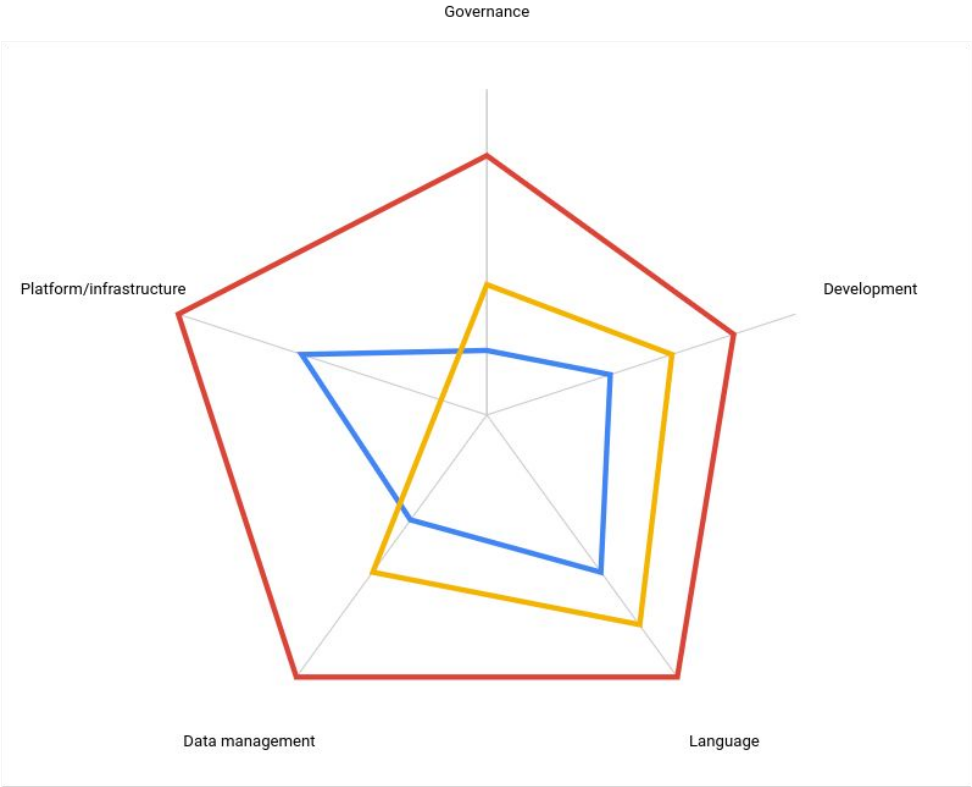
Consistency viewpoint - summary

Reach eventual consistency with **DDD-inspired solutions**

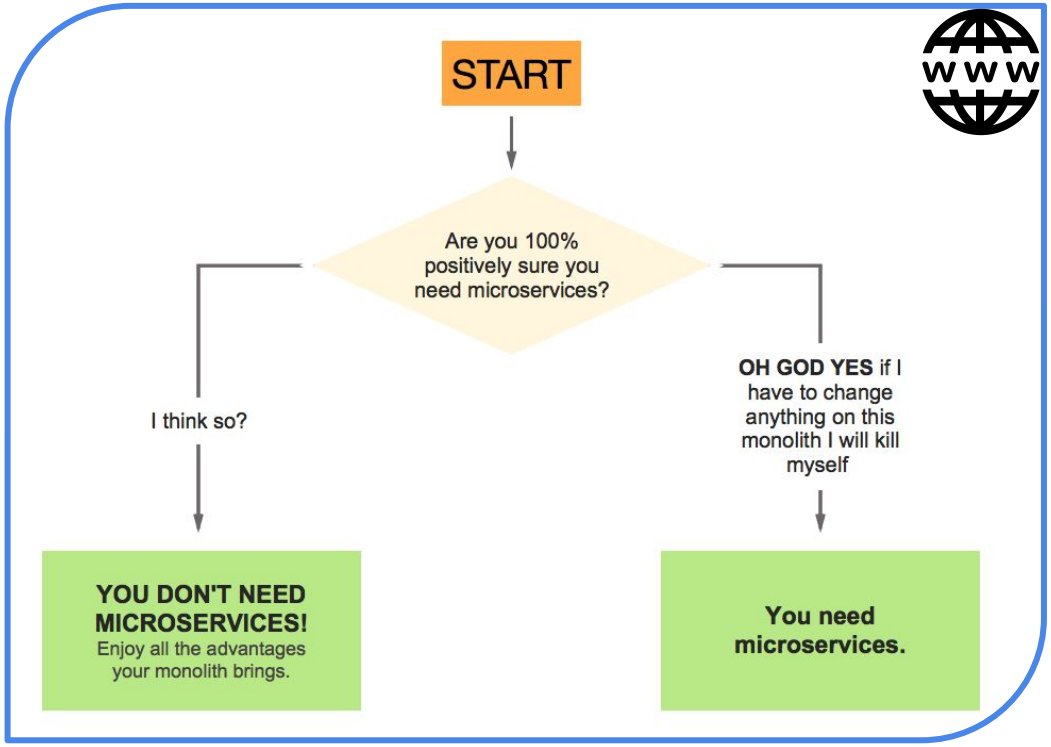
- **Platform/Infrastructure** is constrained
- **Data management** is severely constrained
- **Governance** is constrained

Reach quasi-atomicity with **compensating transactions**

- **Governance** is severely constrained
- **Platform/Infrastructure** is constrained



Which one then?



Microservices adoption flowchart

[Stavros Korokithaki]

Take away

The **main constraints** imposed to the architectural design space come from **required qualities**, not from an architectural style.

Microservices allow for mixing different kinds of solutions within the same system: the degree of freedom in **the design space can indeed be larger**.

But there is a **high price** in terms of **technical complexity**.

And: do not trust (IT) social media.