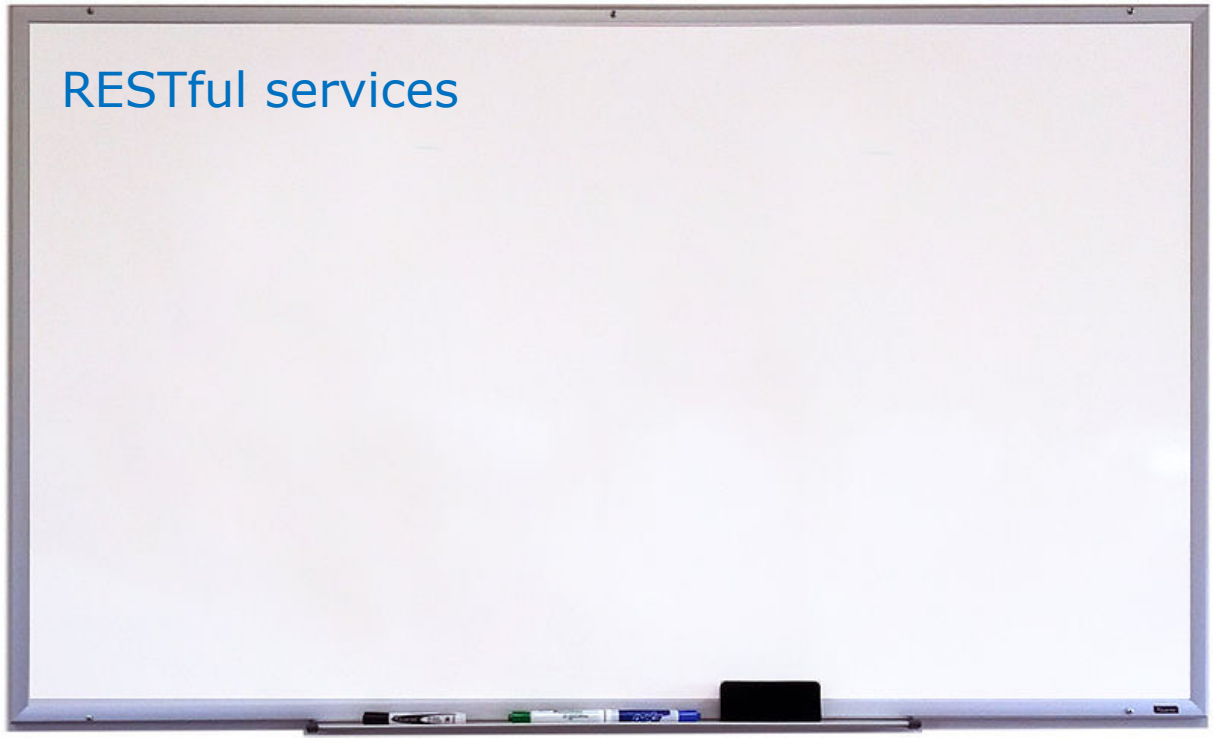
Software services

Antonio Brogi

Department of Computer Science
University of Pisa


RESTful services

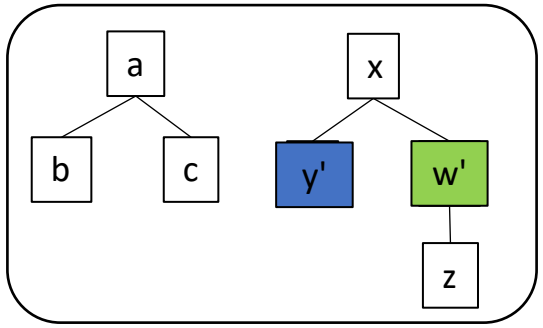
REpresentational State Transfer (REST)

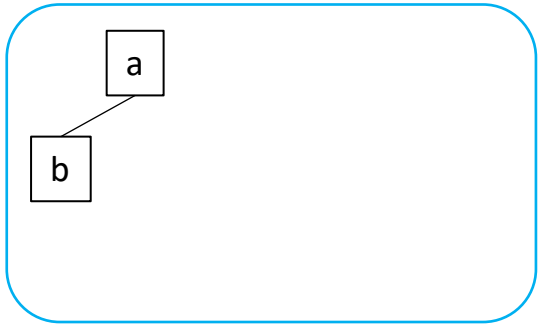
Originally introduced as an *architectural style*, developed as an abstract model of the Web architecture to guide the redesign and definition of HTTP and URIs

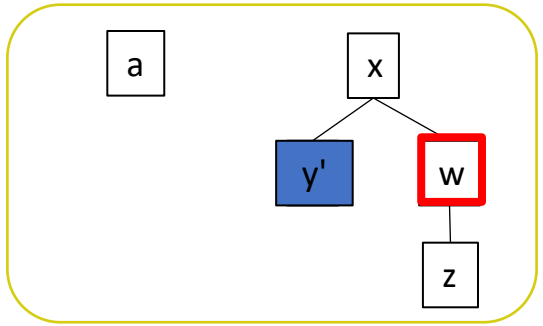
"each action resulting in a transition to the next state of the application by transferring a representation of that state to the user"

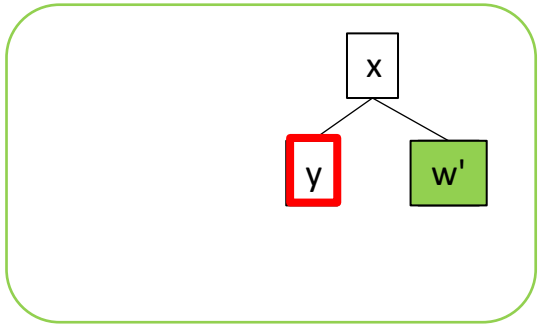
R.T. Fielding. Architectural styles and the design of network-based software architectures. PhD Thesis, University of California, Irvine. 2000.

State transfer









REST principles

1. Resource identification through URIs
   - Service exposes set of resources identified by URIs

2. Uniform interface
   - Clients invoke HTTP methods to create/read/update/delete resources:
     - POST and PUT to create and update state of resource
     - DELETE to delete a resource
     - GET to retrieve current state of a resource

3. Self-descriptive messages
   - Requests contain enough context information to process message
   - Resources decoupled from their representation so that content can be accessed in a variety of formats (e.g., HTML, XML, JSON, plain text, PDF, JPEG, etc.)

4. Stateful interactions through hyperlinks
   - Every interaction with a resource is stateless
   - Server contains no client state, any session state is held on the client
   - Stateful interactions rely on the concept of explicit state transfer

Example

Customer wants to update his last food order

barbera.com

GET /customers/fred

```
200 OK
<customer>
   <name>Fred Flinstone</name>
   <address> 45 Cave Stone Road, Bedrock</address>
   <orders>http://barbera.com/customers/fred/orders</orders>
</customer>
```

GET /customers/fred/orders

```
200 OK
<orders>
   <customer>http://barbera.com/customers/fred</customer>
   <order id="1">
      <orderURL>http://barbera.com/orders/1122</orderURL>
      <status>open</status>
   </order>
   ...
</orders>
```

GET /orders/1122

```
200 OK
<order>
   <customer>http://barbera.com/customers/fred</customer>
   <item quantity="1">brontoburger</item>
</order>
```

PUT /orders/1122

```
<order>
   <customer>http://barbera.com/customers/fred</customer>
   <item quantity="50">brontoburger</item>
</order>
```

```
200 OK
```

Example

Using a simple Doodle service
to organize next Friday night

```
POST /polls
<title>Friday night</title>
<options>
   <option>bowling</option>
   <option>pool</option>
   <option>poker</option>
</options>
...
```

→

←

```
201 Created
Content-Location: /polls/112233
```

```
GET /polls/112233
```

←

→

```
200 OK
<poll>
<title>Friday night</title>
...
<votes href="/vote">
</poll>
```

```
POST /polls/112233/vote
<name>Barnie>/name>
<choice>pool</choice>
```

←

→

```
GET /polls/112233
```

→

←

```
200 OK
<title>Friday night</title>
...
<votes>
   <vote id="1">
      <name>Barnie</name>
      <choice>pool>/choice>
   </vote>
</votes>
```

```
201 Created
Content-Location: /polls/112233/vote/1
```

```
DELETE /polls/112233
```

→

←

```
200 OK
```

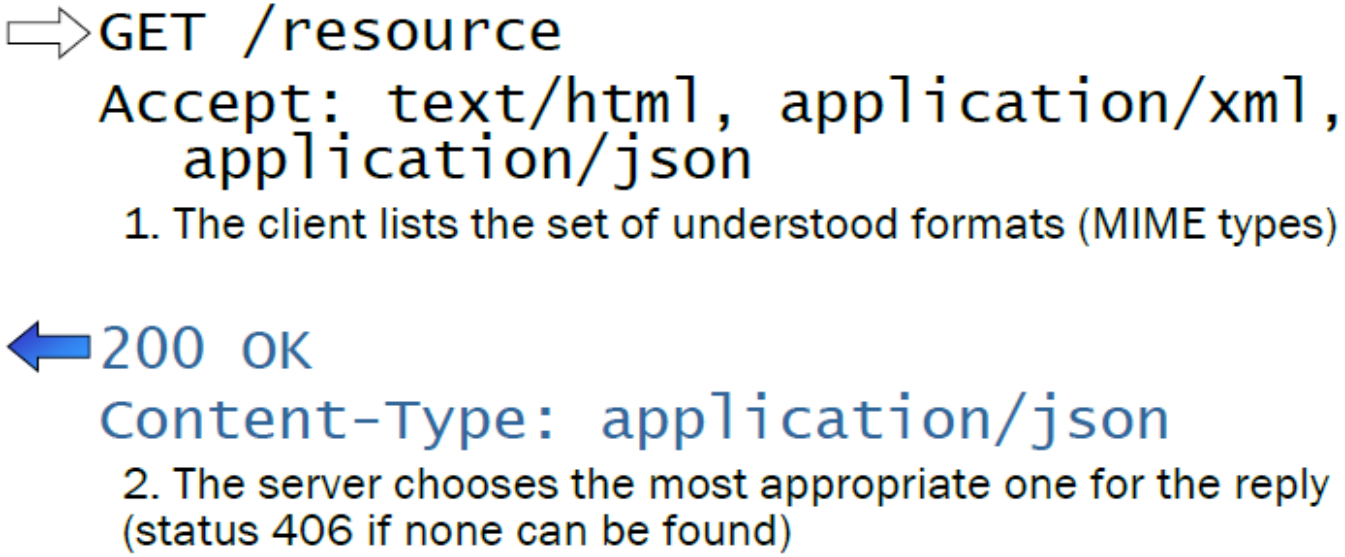
```
GET /polls/112233
```

←

→

```
404 Not Found
```

Content negotiation

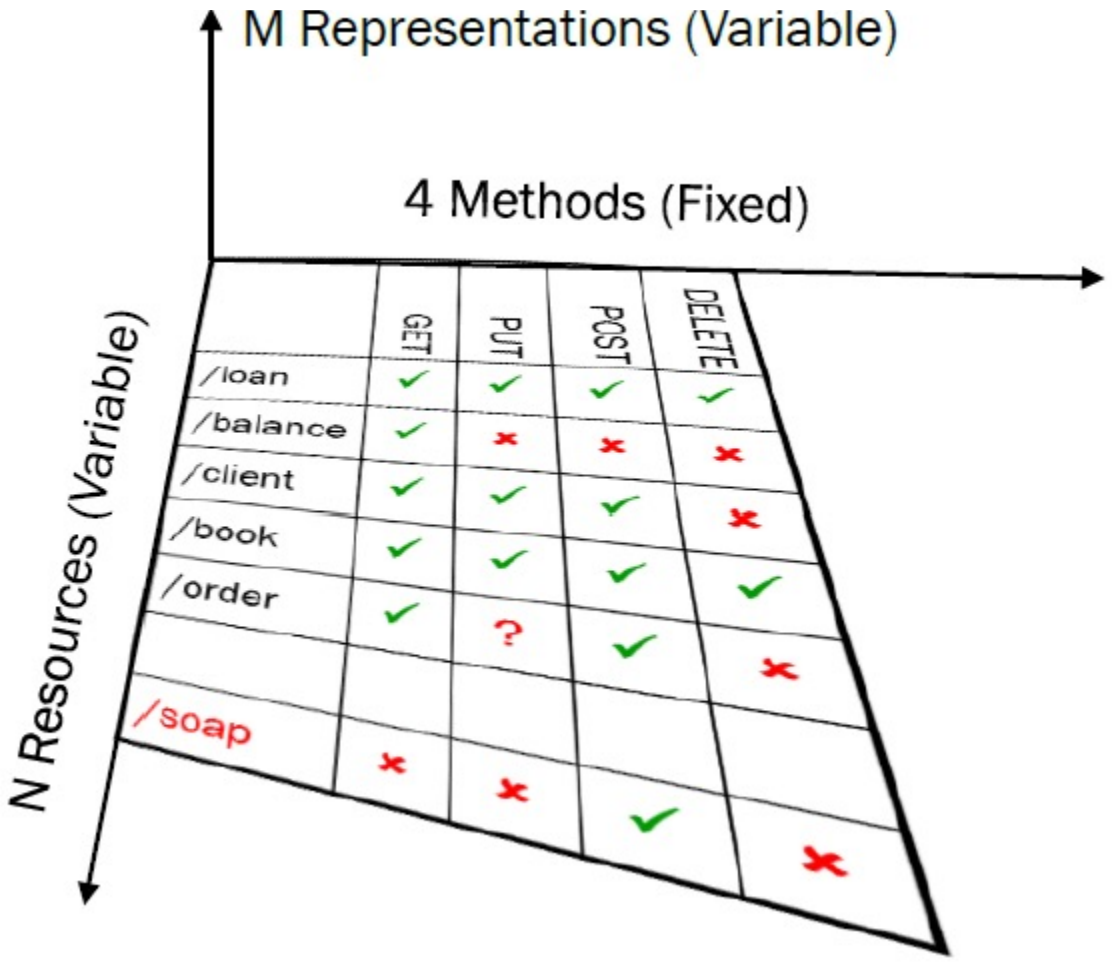


Design methodology

1. Identify resources to be exposed as services
2. Model relationships between resources with hyperlinks
3. Define «nice» URIs to address resources
4. Understand what it means to do GET/POST/PUT/DELETE for each resource (and whether to allow it or not)
5. Design and document resource representations
6. Implement and deploy on Web service
7. Test with web browser

C. Pautasso. REST tutorial. 2010.

Design space



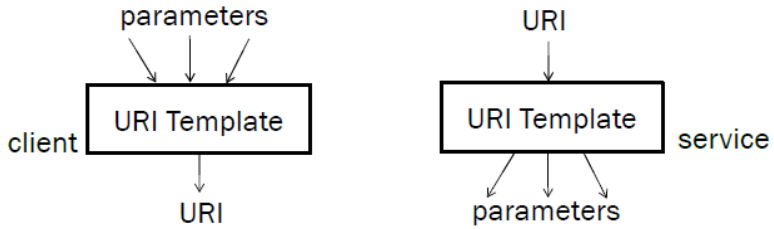
URI design guidelines

- Prefer nouns to verbs

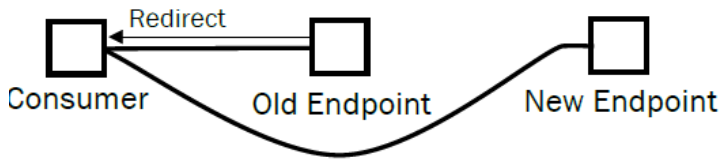
  GET /book?isbn=24&action=delete

  DELETE /book/24

- Keep URIs short
- Use URI templates to construct and parse parametric URIs



- Do not change URIs, use redirection if needed



REST pros and cons

Simplicity

- low learning curve
  - REST leverages well-known standards (HTTP, XML, URI)
  - needed infrastructure is already there
- minimal tooling
  - deploying a service similar to building dynamic Web site
  - developers can begin testing service from ordinary Web browsers

Efficiency

- lightweight protocol
- lightweight message formats

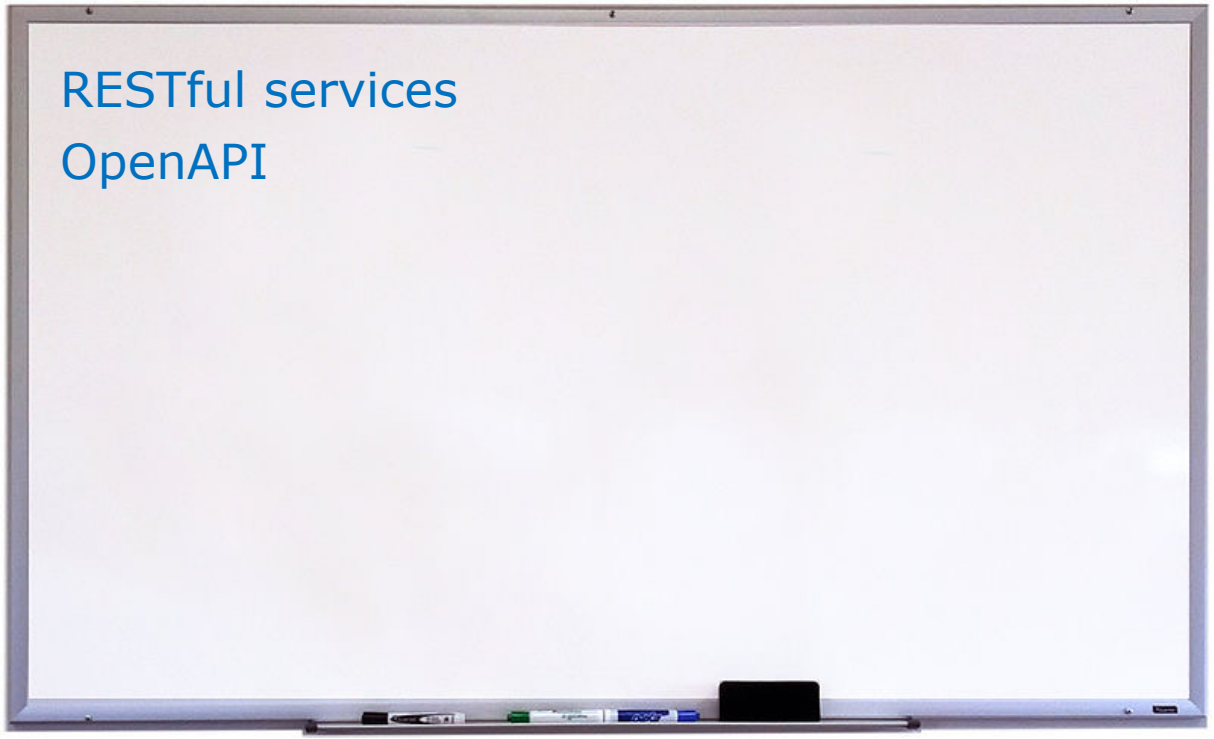
Scalability

- stateless RESTful services can serve a huge number of clients

Clients over-/under-fetch data

Limit on number of client requests

Inconsistent naming conventions


RESTful services
OpenAPI

Amazon API mandate (Bezos' 2002 memo)

1. All teams will henceforth expose their data and functionality through service interfaces.
2. Teams must communicate with each other through these interfaces.
3. There will be no other form of interprocess communication allowed.
4. It doesn't matter what technology they use.
5. All service interfaces, without exception, must be designed from the ground up to be externalizable.
6. Anyone who doesn't do this will be fired.

OpenAPI

OpenAPI Initiative (Linux Foundation Collaborative Project) aims at creating a standardized, vendor neutral description format of REST APIs

- f.k.a. Swagger
- simple (JSON-based) description language to specify HTTP API endpoints, how they are used, and the structure of data that comes in and out

OpenAPI members

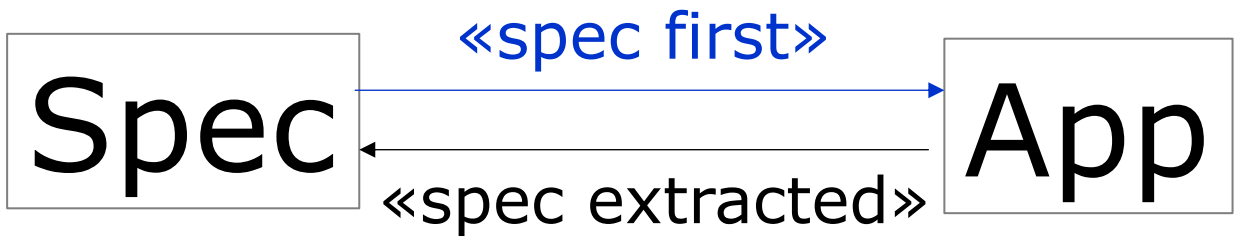
OpenAPI

/* Simple example: One endpoint /api/users_id supporting GET to retrieve list of user Ids */

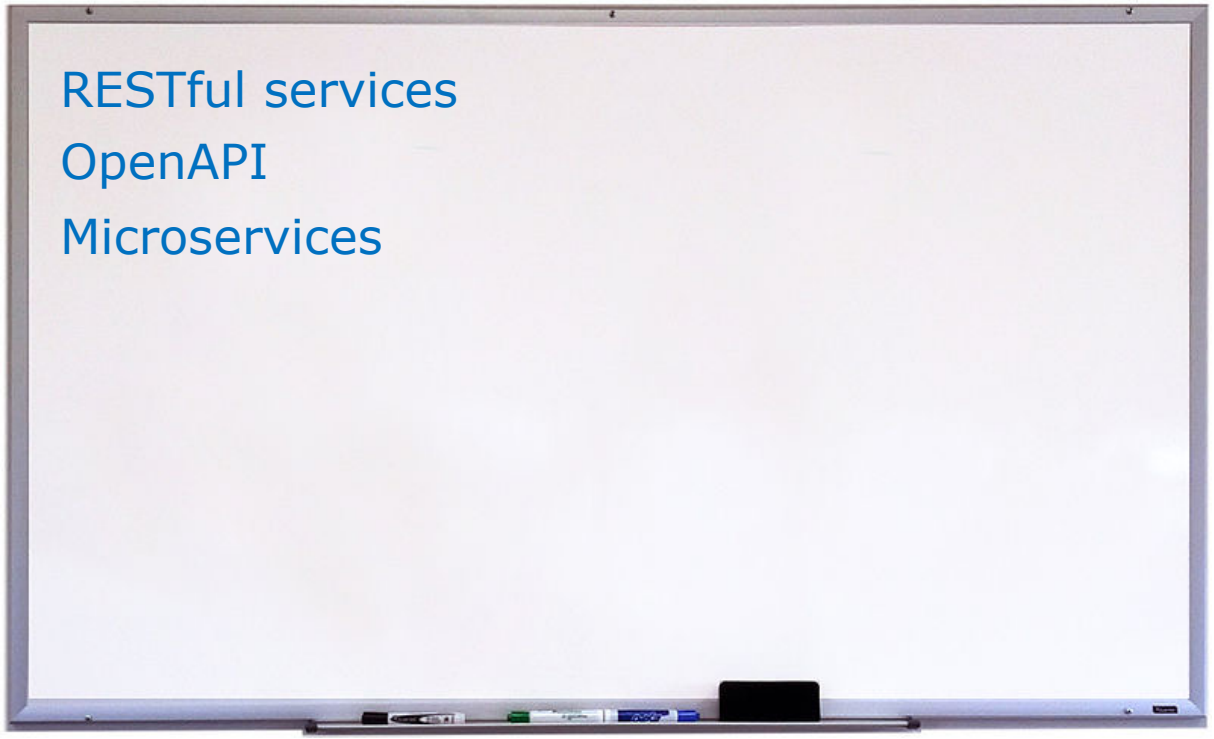
```
swagger: "2.0"
info:
  title: BipBip Data Service
  description: returns info about BipBip data
  license:
    name: APLv2
    url: https://www.apache.org/licenses/LICENSE-2.0.html
  version: 0.1.0
  basePath: /api
  paths:
    /user_ids:
      get:
        operationId: getUserIds
        description: Returns a list of ids
        produces:
        - application/json
        responses:
          '200':
            description: List of Ids
            schema:
            type: array
            items:
              type: integer
```

OpenAPI

E.g., Connexion framework for Flask automagically handles HTTP requests based on OpenAPI Specification of your API



➔lab


RESTful services
OpenAPI
Microservices

Motivations

(1) Shorten lead time for new features and updates

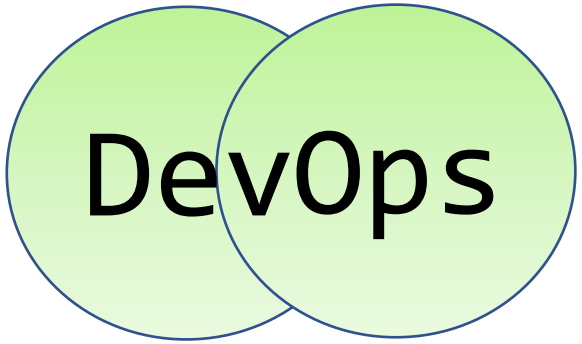
→ accelerate rebuild and redeployment

(2) Scale, effectively

Essence of microservices

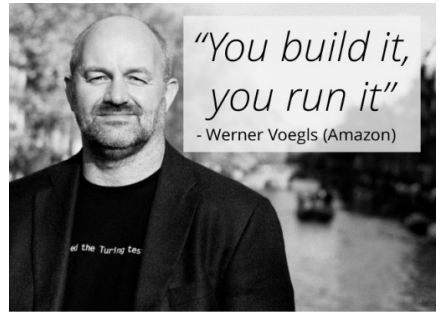
Develop applications as sets of services:

- each running in its own ~~process~~ container
- communicating with lightweight mechanisms
- built around business capabilities
- decentralizing data management
- independently deployable
- horizontally scalable
- fault resilient



Culture:

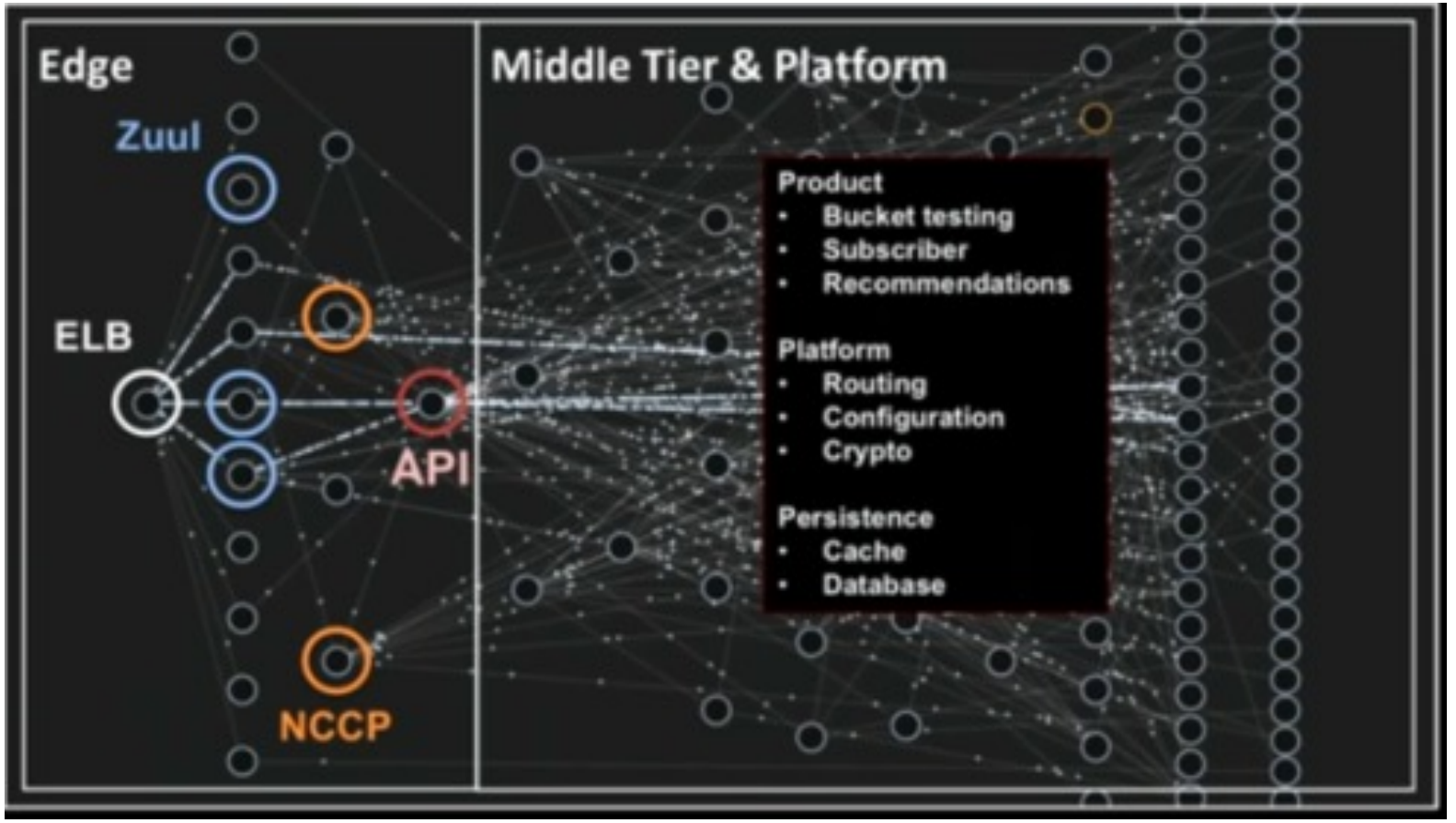
Same team responsible for service development, deployment and management



Tools:

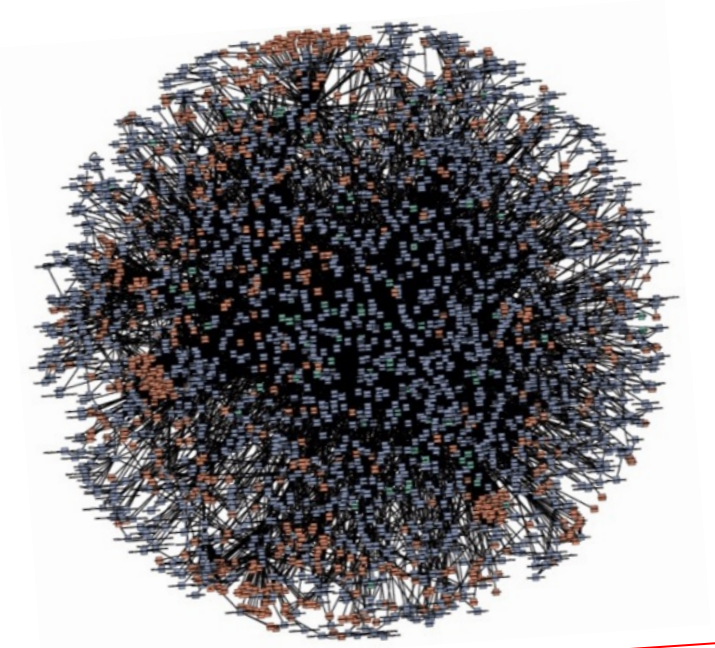
VCS (Version Control Systems)
CI/CD (Continuous Integration/Continuos Deployment)
IaC (Infrastructure As Code)

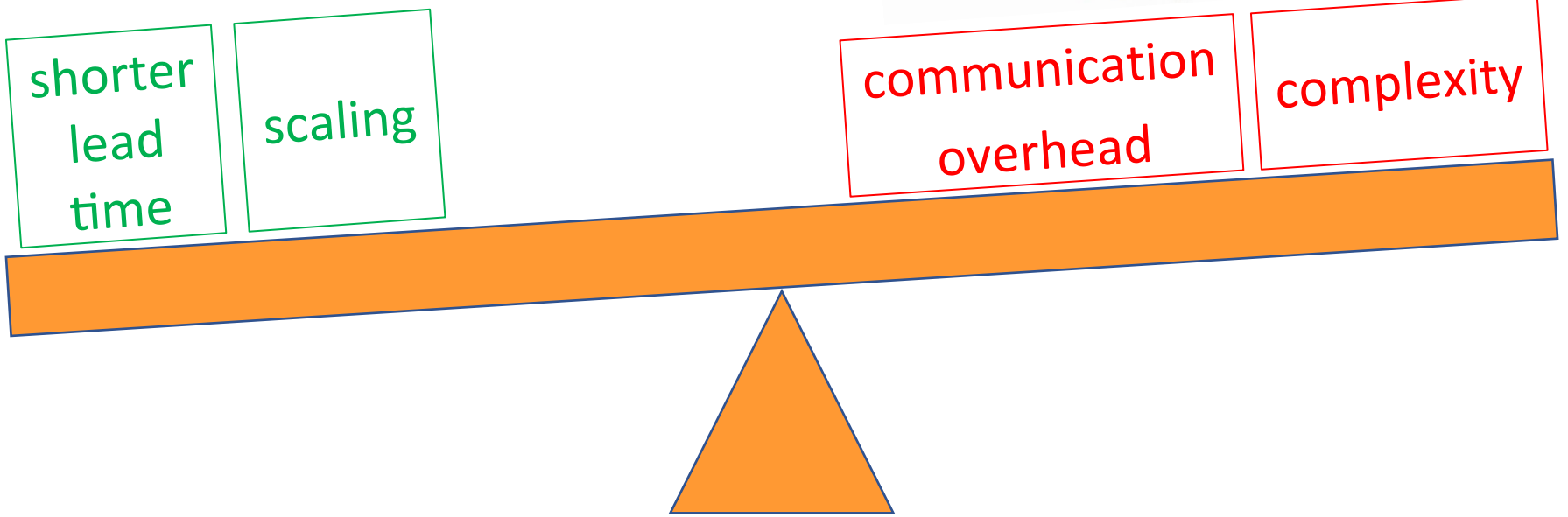
Netflix

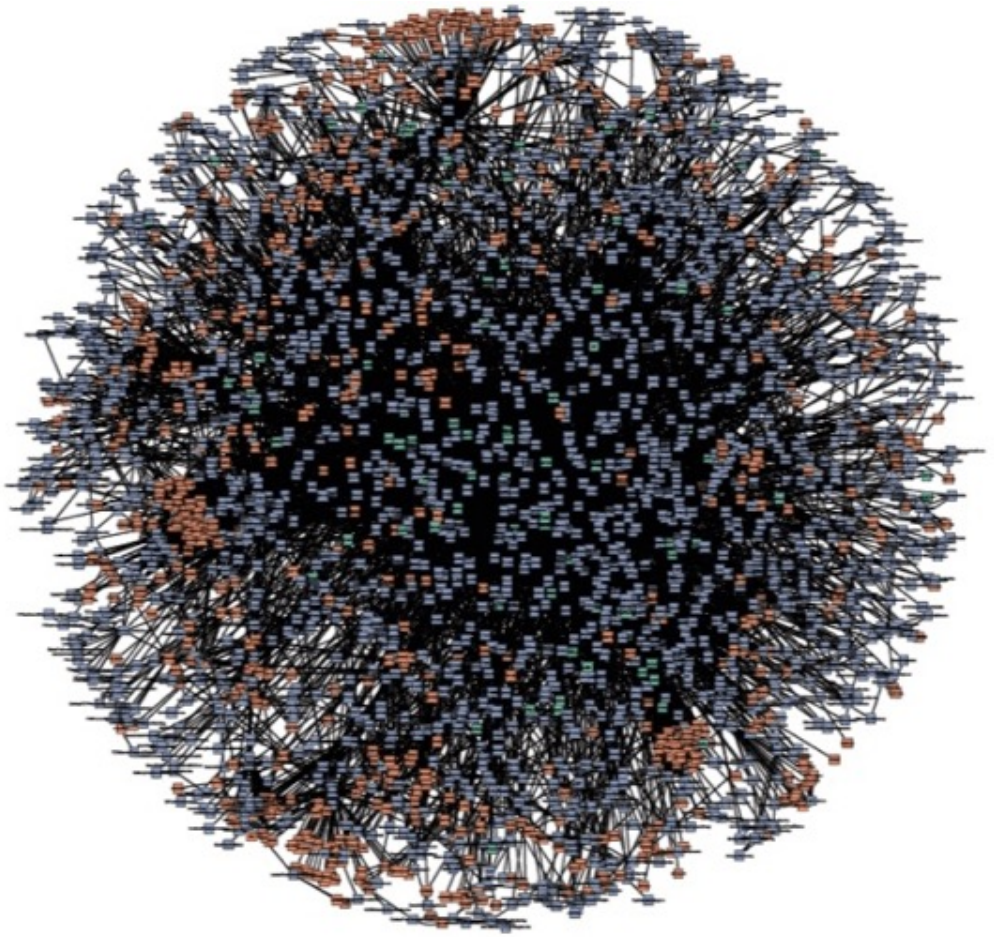


- **ELB**: AWS Elastic Load Balancing
- **Zuul**: proxy layer, performs dynamic routing
- **NCCP**: legacy tier, supporting earlier devices
- Netflix's **API** gateway, calling all other services

Concluding remarks







amazon Google NETFLIX Spotify twitter Linked in ebay zalando GROUPON Uber ...

e.g. (December 2023)

- Spotify: 602 million monthly active users
- Netflix: 260 million paid subscribers

Can I play with microservices?