

Microservices architecture

Antonio Brogi

Department of Computer Science
University of Pisa



Introduction

Introduction

How to decompose a system into components?

- Components can be developed in parallel by different teams
- Components can be reused, replaced, distributed across multiple computers
- Components must be easily replicated, run in parallel, and migrated to exploit cloud-based scalability, reliability, and elasticity

→ use stateless services that maintain persistent information in local db

Introduction

Software service

- Software component that can be accessed over the Internet
- Given an input, a service produces a corresponding output, without side effects
- Service is accessed through its published interface, all implementation details are hidden
- Services do not maintain any internal state
 - State information is either stored in a database or maintained by the service requestor
 - State information can be included in service request, updated state info in service result
 - Services can be dynamically reallocated from one virtual server to another → scalability

Introduction

Service-oriented architecture (SOA, late 1990s)

- Independent, stand-alone services, with public interfaces, implemented with different technologies

Web Services (early 2000s)

- XML, SOAP, WSDL ... plus dozens (!) of other standards
- Web services exchanging large and complex XML data → message management overhead

Modern service-oriented systems

- simpler, light weight service interaction protocols
- simpler interfaces, more efficient format for encoding message data
- lower overheads, faster execution

Introduction

Amazon's rethinking of what a service should be

- a service should be related to a **single business function**
- services should be completely **independent**, with their **own database**
- service should **manage their own user interface**
- it should be possible to replace/replicate a service without changing other services

→ microservices =

small-scale stateless services that have a single responsibility

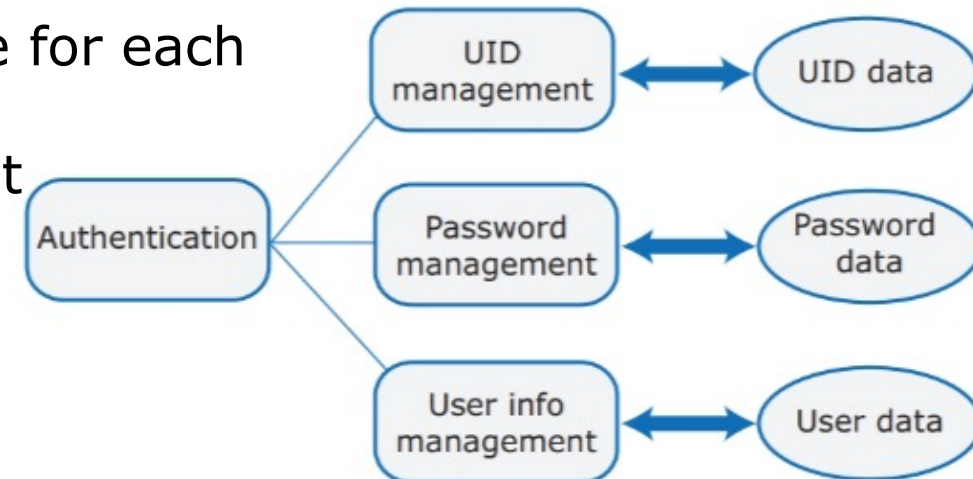
Example

System using an authentication module providing:

- user registration
- authentication using UID/password
- two-factor authentication
- user information management
- password reset

To identify the microservices that might be used for the authentication system

- break coarse-grain features into more detailed functions
- look at the data used and identify a microservice for each logical data item to be managed
- minimize amount of replicated data management



Introduction Microservices

Microservices

Microservices

- small-scale services that can be combined to create applications
- independent (service interface not affected by changes to other services)
- possible to modify and re-deploy service without changing/stopping other services

Characteristic	Explanation
Self-contained	Microservices do not have external dependencies. They manage their own data and implement their own user interface.
Lightweight	Microservices communicate using lightweight protocols, so that service communication overheads are low.
Implementation independent	Microservices may be implemented using different programming languages and may use different technologies (e.g., different types of database) in their implementation.
Independently deployable	Each microservice runs in its own process and is independently deployable, using automated systems.
Business-oriented	Microservices should implement business capabilities and needs, rather than simply provide a technical service.

Microservices

Coupling measures number of inter-component relationships

Low coupling → independent services, independent updates

Cohesion measures number of intra-component relationships

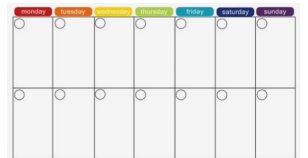
High cohesion → less inter-service communication overhead

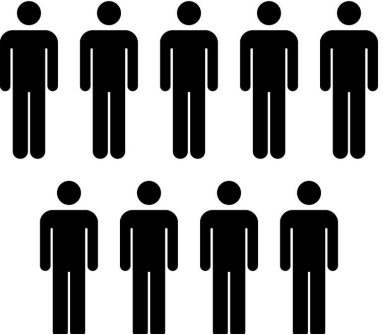
“**Single responsibility principle**”

- Each service should do one thing only and it should do it well
- Responsibility \neq single functional activity

How big should a microservice be? “Rule of twos”

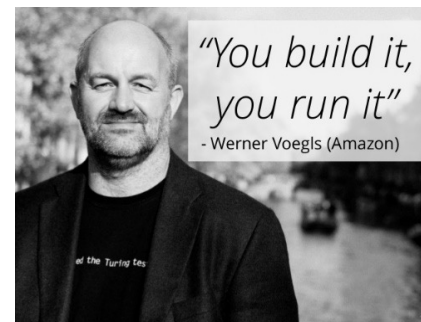
- Service can be developed, tested, and deployed by a team in two weeks
- Team can be fed with two large pizzas (8-10 people)





Why so many people for a microservice?

- implement service functionality
- develop code that makes service completely independent
 - processing incoming and outgoing messages
 - manage failures (service/interactions failures)
 - manage data consistency when data are used by other services
 - maintain service own interface
- test service and service interactions
- support service after deployment



Introduction
Microservices
Microservices architecture

Motivations



(1) Shorten lead time for new features and updates
→ accelerate rebuild and redeployment



(2) Scale, effectively

Each microservice can be deployed in a separate container
→ quick stop/restart without affecting other services
→ service replicas can be quickly deployed

Example

Photo-printing system for mobile devices

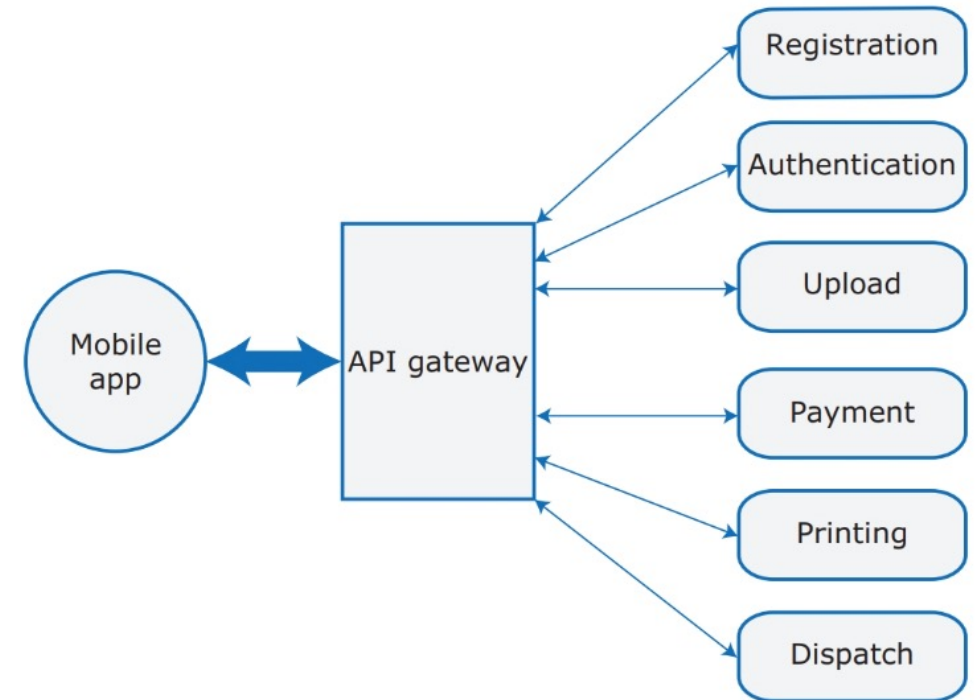
Imagine that you are developing a photo-printing service for mobile devices. Users can upload photos to your server from their phone or specify photos from their Instagram account that they would like to be printed. Prints can be made at different sizes and on different media.

Users can choose print size and print medium. For example, they may decide to print a picture onto a mug or a T-shirt. The prints or other media are prepared and then posted to their home. They pay for prints either using a payment service such as Android or Apple Pay or by registering a credit card with the printing service provider.

Separate services for each area of functionality

API gateway

- insulates user app from the system's microservices
- single point of contact
- translates app service requests into calls to microservices



Architectural design decisions

Development teams for each service are autonomous

How to decompose system into a set of microservices?

- Not too many (low cohesion -> communication overhead)
- Not too few (high coupling -> less independency for updates/deployment/..)

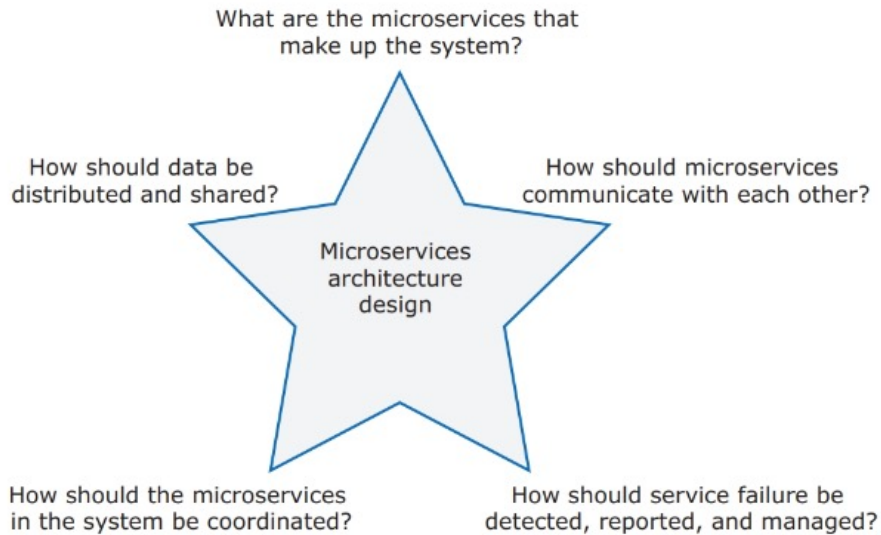
Difficult

Some tips

1. Balance fine-grain functionality and system performance
2. Follow the “common closure principle”
(elements likely to be changed at the same time should stay in same service)
3. Associate services with business capabilities
4. Services should have access only the data they need
(+ data propagation mechanisms)

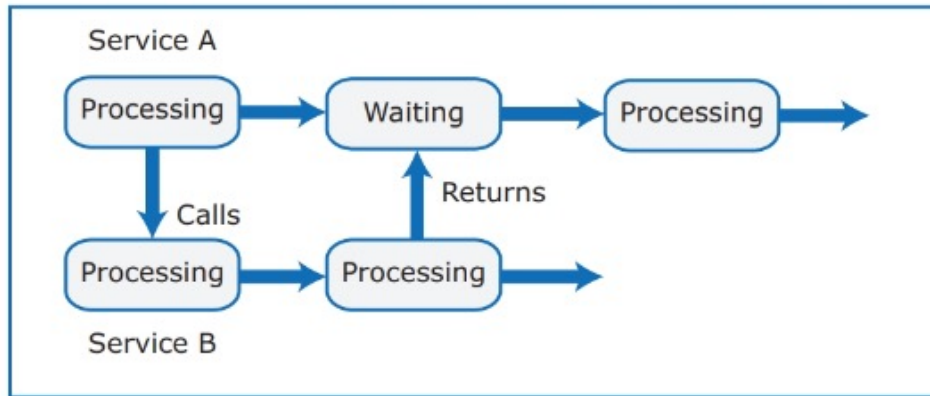
Starting points (tips)

- Start by looking at the data services have to manage
- Start with a monolith, and decompose it later

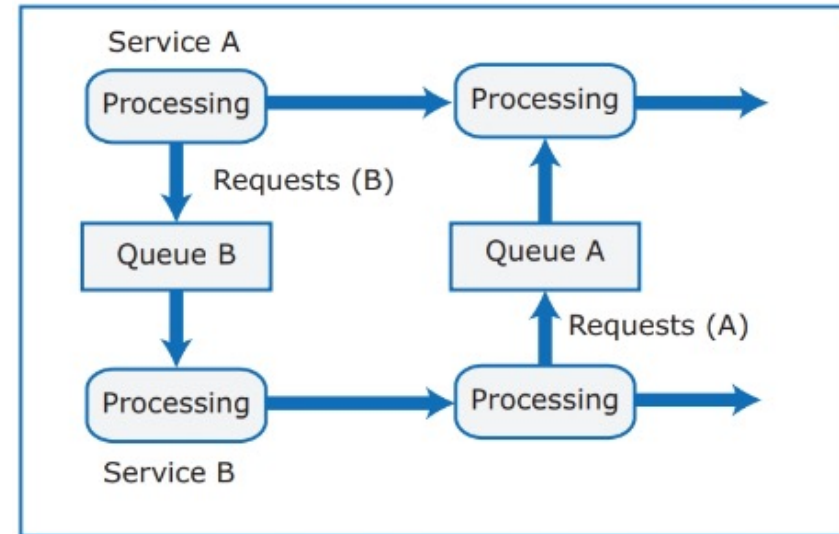


Service communications

Synchronous vs. asynchronous service interaction



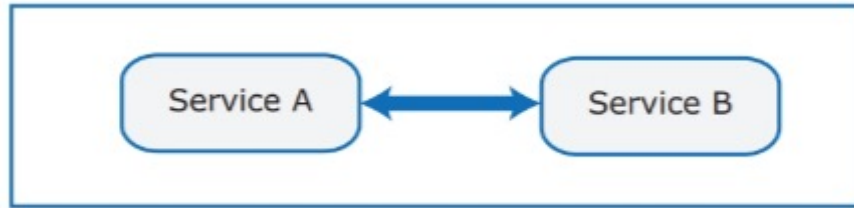
+ easier to write and understand



+ looser coupling, more efficient
- more difficult to write to write and understand

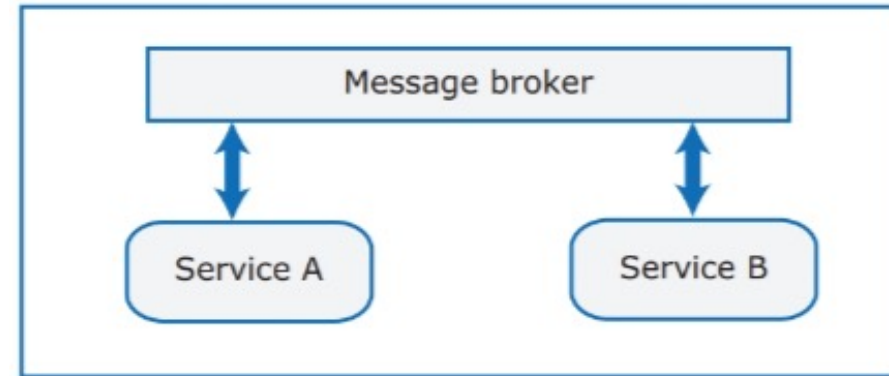
Service communications

Direct vs. indirect service communication



Messages directly sent to service address

- + Simpler
- + Faster
- Requester must know recipient's URI



Messages addressed to service name

Messages sent to message broker, which
finds address of requested service
can handle message translation

Example of message broker: RabbitMQ

- + Can support synchronous & asynchr. interactions
- + Easier to modify/replace services
- More complex
- Slower



Data distribution and sharing

Each microservice should manage its own data



To deal with data dependencies:

1. Data sharing should be as little as possible
2. Most sharing should be read-only, with few services responsible for data updates
3. Include a mechanism to keep db copies used by replicated services consistent

Data distribution and sharing

Shared database architectures employ ACID transactions to serialize updates and avoid inconsistency

In distributed systems we must trade-off data consistency and performance

→

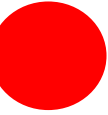
Microservices systems must be designed to tolerate some degree of data inconsistency

Two types of inconsistency have to be managed:

1. Dependent data inconsistency
Actions/failures of one service can cause data managed by another service to become inconsistent
2. Replica inconsistency
Several replicas of the same service may be executing concurrently
Each with its own db copy, each updates its own db copy
→ Need to make these dbs “eventually consistent”



CAP theorem

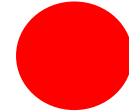


In presence of a network **P**artition, you cannot have both **A**vailability and **C**onsistency.

- *Consistency*: any read operation that begins after a write operation must return that value, or the result of a later write operation
- *Availability*: every request received from a non-failing node must result in a response
- *Network partition*: network can lose arbitrarily many messages sent from one group to another



CAP theorem



In presence of a network partition, you cannot have both Availability and Consistency.

- *Consistency*: any read operation that begins after a write operation must return that value, or the result of a later write operation
- *Availability*: every request received from a non-failing node must result in a response
- *Network partition*: network can lose arbitrarily many messages sent from one group to another

Proof - By contradiction, suppose that both Availability and Consistency hold.

Suppose that services $S1$ and $S2$ belong to two different network partitions, and that both contain value $v0$.

Consider the following sequences of events: $\alpha1 = "C \text{ sends } \textit{write}(v1)" \text{ to } S1; E1; "S1 \text{ responds to } C"$ where $E1$ is a (possibly) empty sequence of other events which contains neither other client requests to $S1$ or $S2$, nor messages from $S1$ received by $S2$, nor messages from $S2$ received by $S1$.

Consider the following sequences of events: $\alpha2 = "C \text{ sends } \textit{read}" \text{ to } S2; E2; "S2 \text{ responds to } C"$ where $E2$ is a sequence like $E1$.

Note that " $S1 \text{ responds to } C$ " and " $S2 \text{ responds to } C$ " belong to $\alpha1$ and $\alpha2$, respectively, because of the Availability hypothesis.

Now $\alpha1.\alpha2 = \alpha2$ for $S2$, hence $S2$ responds $v0$ to C .

Because of the Consistency hypothesis, $S2$ should instead respond $v1$ to C . Contradiction.

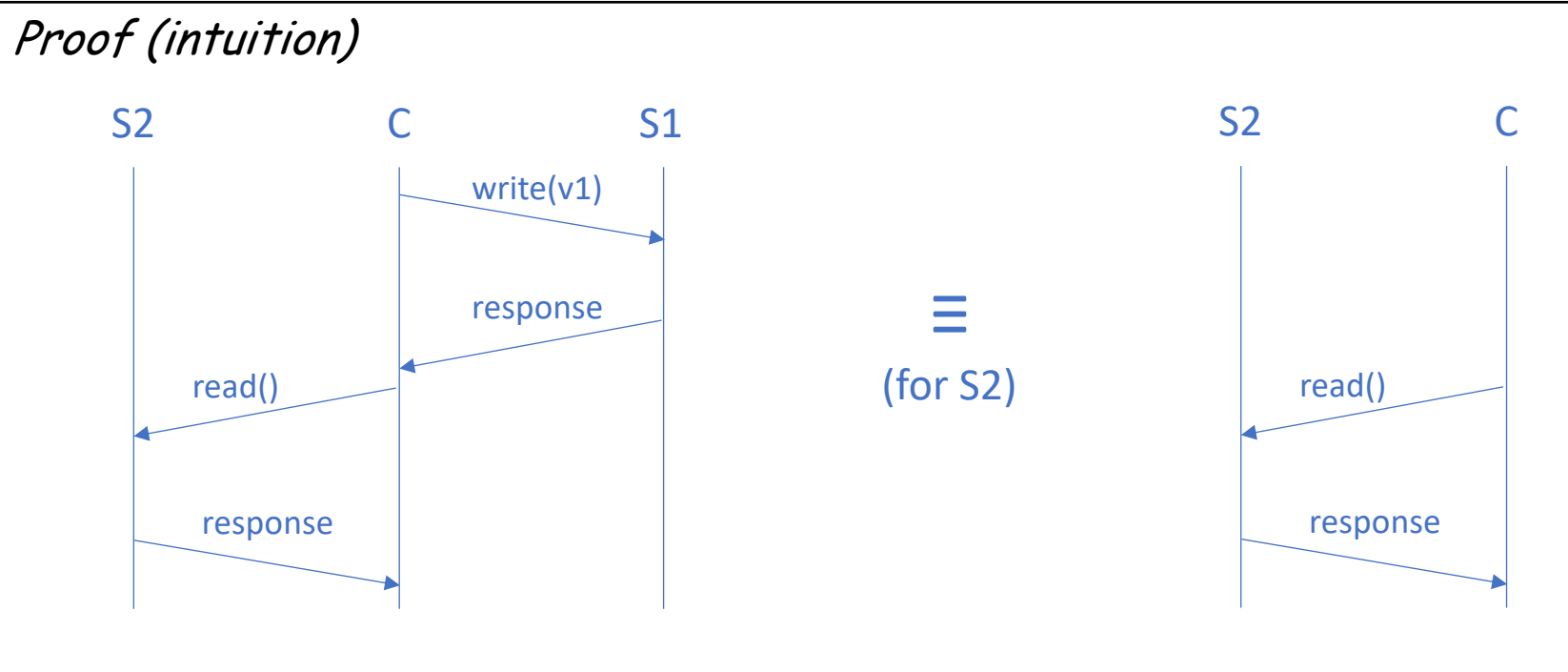


CAP theorem



In presence of a network partition, you cannot have both Availability and Consistency.

- *Consistency*: any read operation that begins after a write operation must return that value, or the result of a later write operation
- *Availability*: every request received from a non-failing node must result in a response
- *Network partition*: network can lose arbitrarily many messages sent from one group to another



The Saga pattern



Example



Implement each business transaction that spans multiple services as a *saga*

A saga is a sequence of local transactions

Each local transaction updates a database and triggers next local transaction(s) in the saga.

If a local transaction fails then the saga executes a series of compensating transactions

Two ways of coordinating sagas:

- *Choreography* - each local transaction publishes event that triggers next local transaction(s)
- *Orchestration* - an orchestrator tells participants which local transactions to execute

Compensating transactions:

- *Backward model* - undo changes made by previously executed local transactions
- *Forward model* - "retry later"

Netflix approach



To replicate data in n nodes:

- «write to the ones you can get to, then fix it up afterwards»
- use quorum: e.g. $(n/2 + 1)$ of the replicas must respond



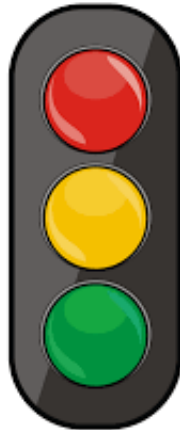
Service coordination

Orchestration



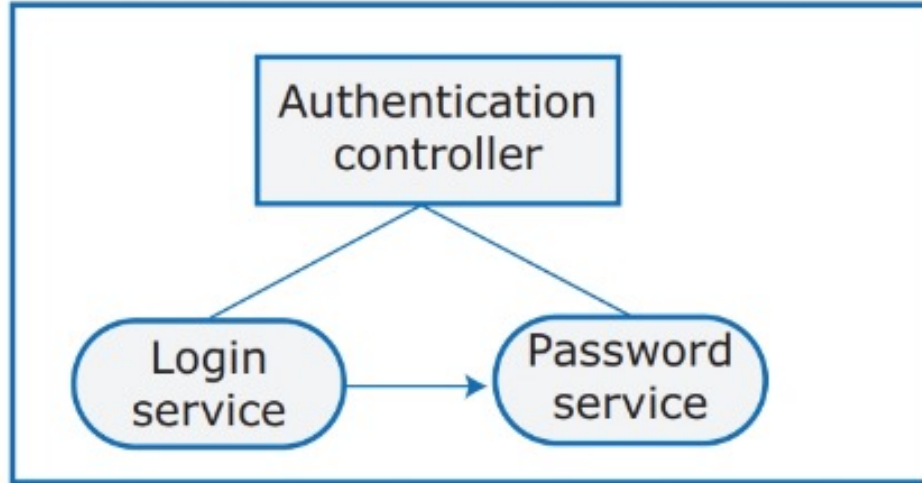
vs.

Choreography



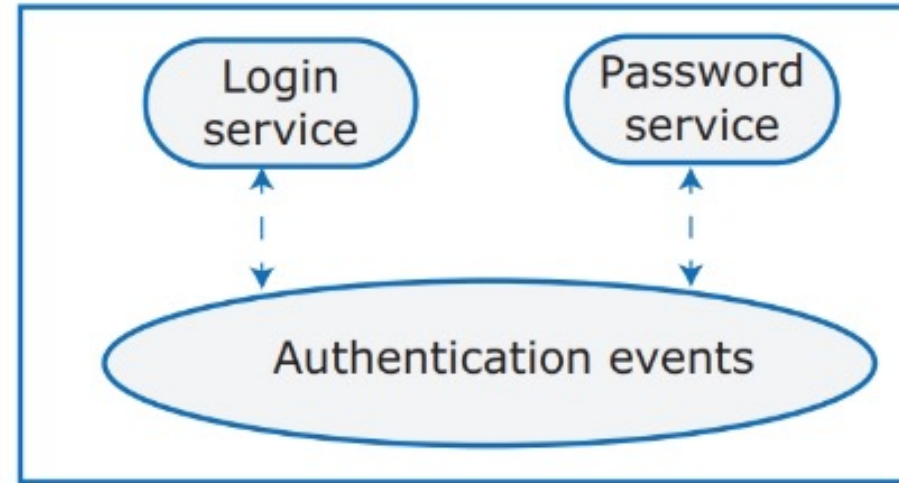
Service coordination

Service orchestration



Explicit controller orchestrating

Service choreography



Publish&subscribe mechanisms

- + no explicit controller
- harder debugging
- harder recovery

Tip: start with orchestration, switch to choreography only if product inflexible/hard to update

Failure management



Something will go wrong, unavoidably

Example

A service offers a functionality F by invoking two other services, each available 99% of time and independent one another.

F will probably be NOT available for around ____ each day

- ☐ 3 seconds
- ☐ 30 seconds
- ☐ 3 minutes
- ☒ 30 minutes

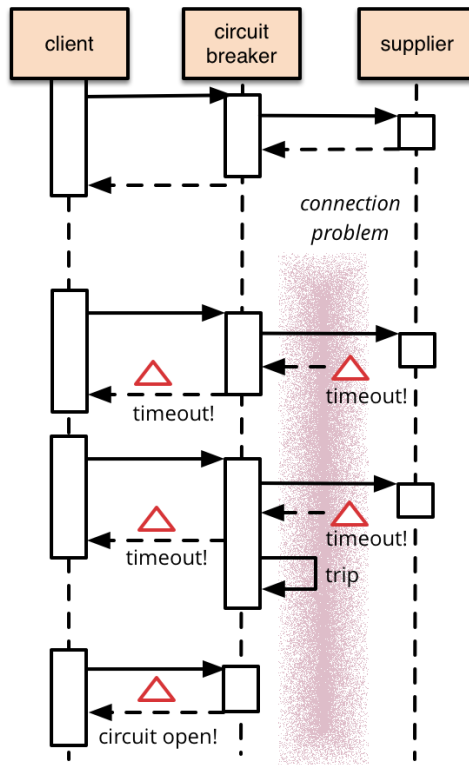
→ Services must be designed to cope with failures

Failure management

Failure type	Explanation
Internal service failure	These are conditions that are detected by the service and can be reported to the service requestor in an error message. An example of this type of failure is a service that takes a URL as an input and discovers that this is an invalid link.
External service failure	These failures have an external cause that affects the availability of a service. Failure may cause the service to become unresponsive and actions have to be taken to restart the service.
Service performance failure	The performance of the service degrades to an unacceptable level. This may be due to a heavy load or an internal problem with the service. External service monitoring can be used to detect performance failures and unresponsive services.

Failure management

Example: Circuit breaker



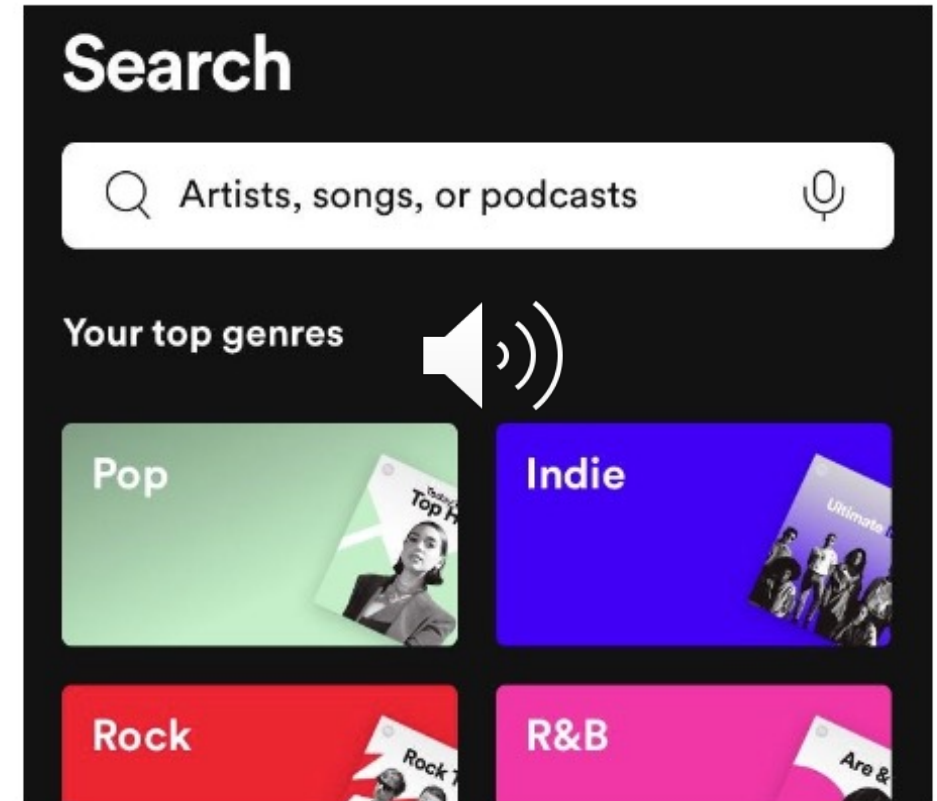
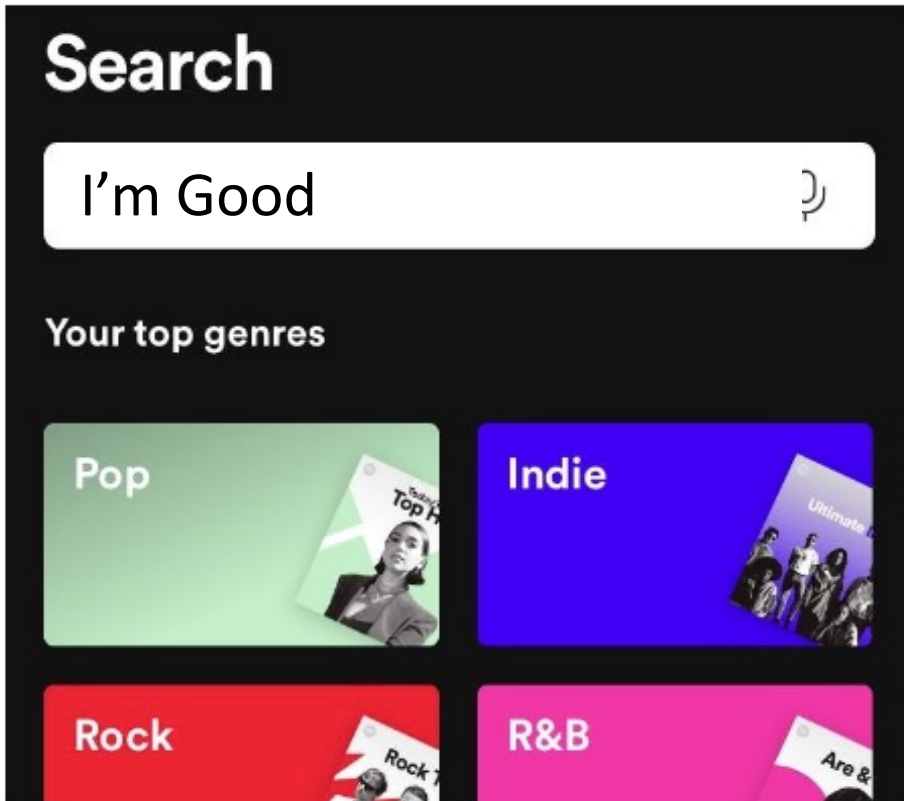
A service client invokes a remote service via a circuit breaker object that functions in a similar fashion to an electrical circuit breaker.

When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately.

After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, the timeout period begins again.

Failure management

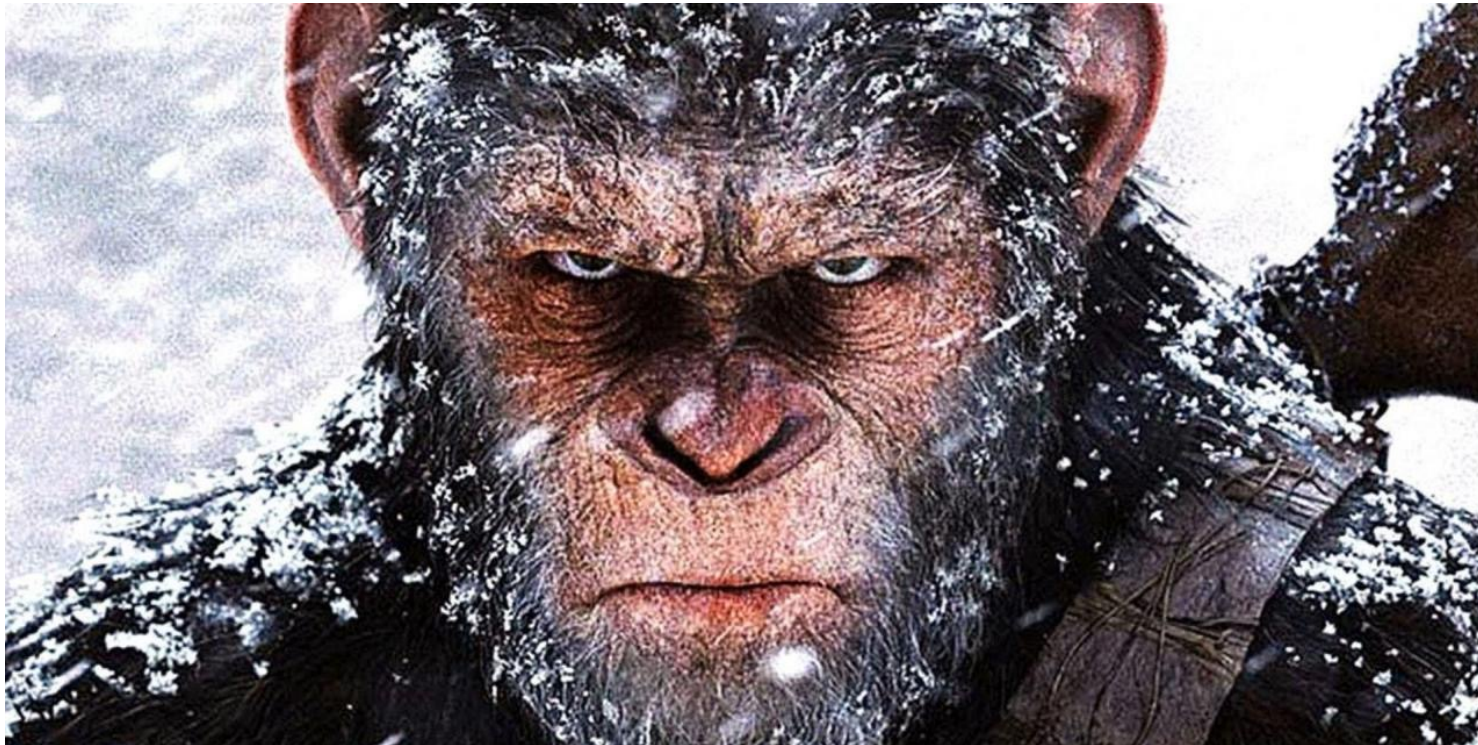
Application should tolerate failures



Failure management



Test (bravely)



Chaos Monkey randomly terminates VM instances and containers that run inside your **production** environment

A whiteboard with a silver frame and a white surface. The text is written in blue. At the bottom of the whiteboard, there are several markers (black, green, blue) and a black eraser.

Introduction
Microservices
Microservices architecture
RESTful services

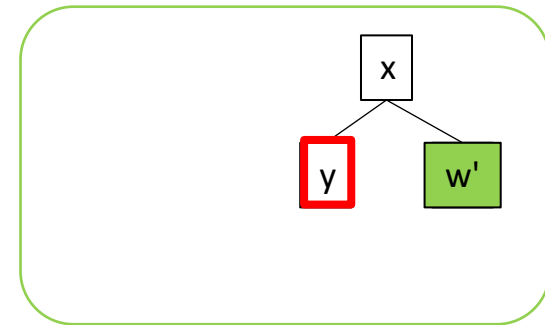
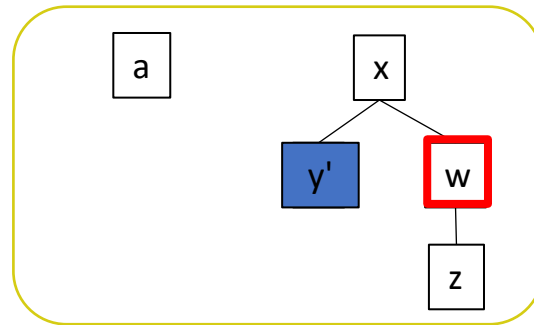
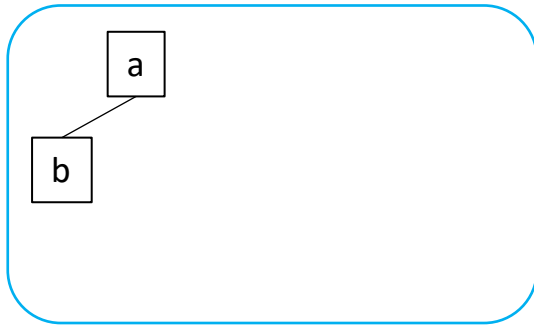
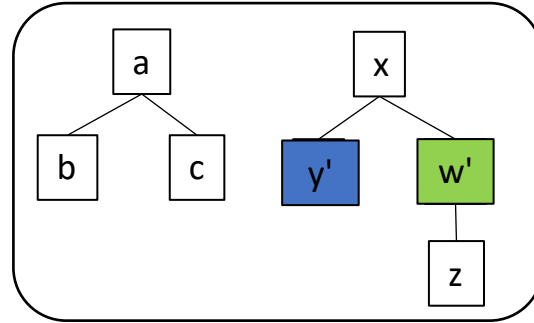
REpresentational State Transfer (REST) ●

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"



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

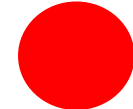
Customer wants to update his last food order





GET /customers/fred

barbera.com



```
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

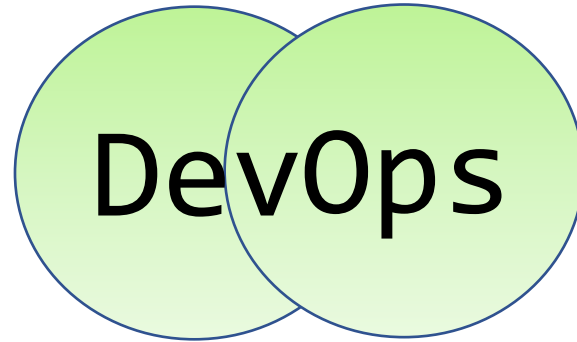
```
<order>
  <customer>http://barbera.com/customers/fred</customer>
  <item quantity="50">brontoburger</item>
</order>
```

200 OK

A whiteboard with a silver frame and a white surface. On the left side, there is a list of five topics written in blue text. At the bottom of the whiteboard, there is a black eraser and several markers (black, green, blue, and white) lying horizontally.

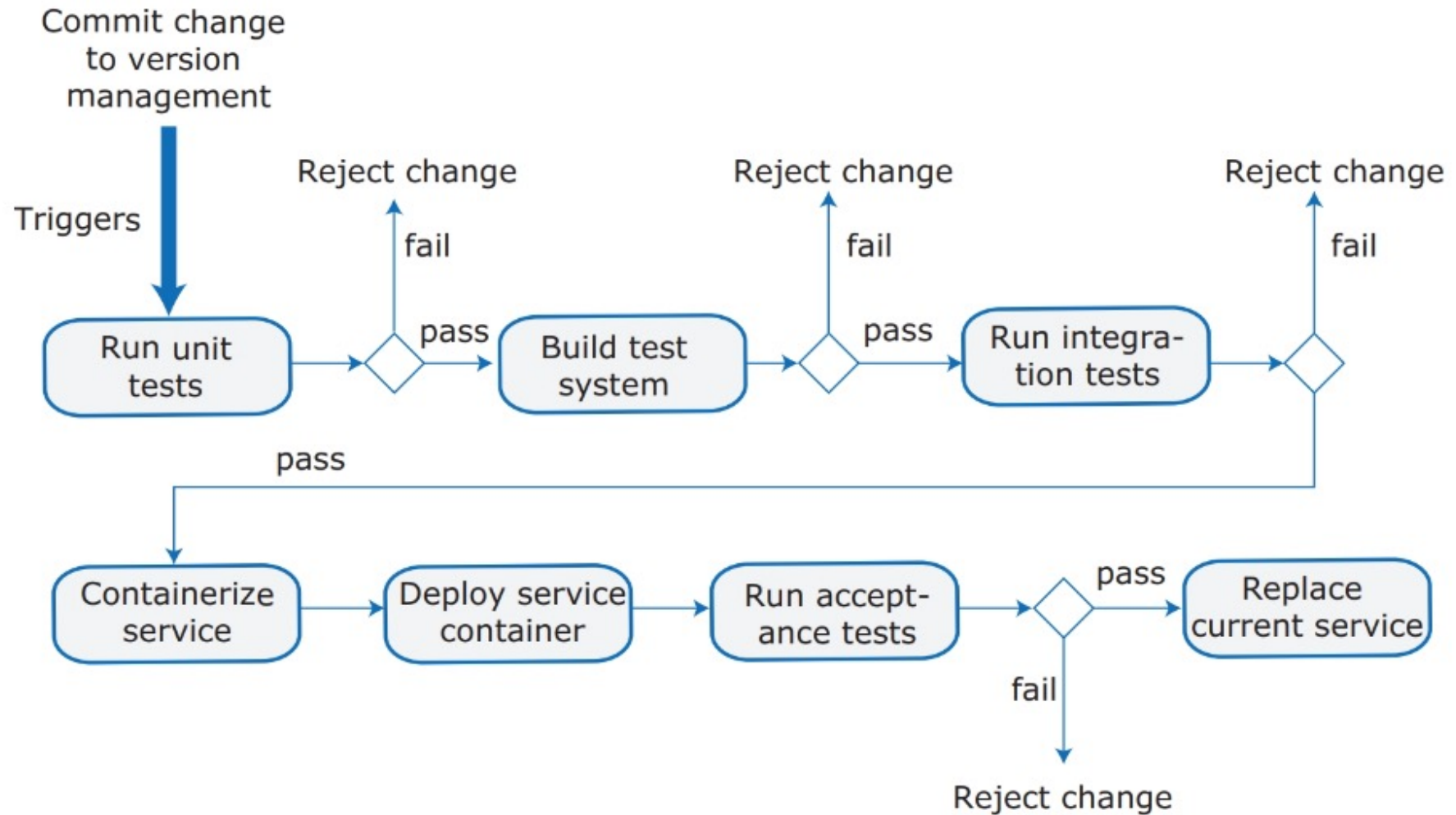
Introduction
Microservices
Microservices architecture
RESTful services
Service deployment

DevOps approach



Same team responsible for service development, deployment and managements

Continuous Deployment pipeline



Monitoring

Testing cannot prevent 100% of unanticipated problems



Need to monitor the deployed services



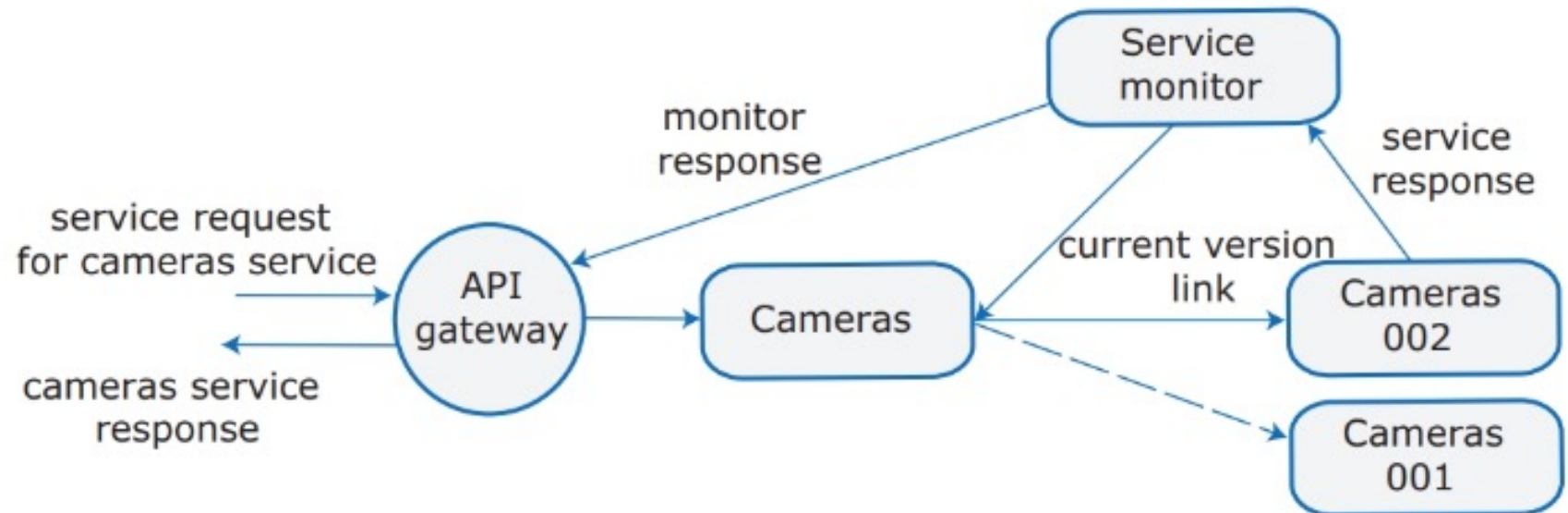
Monitoring

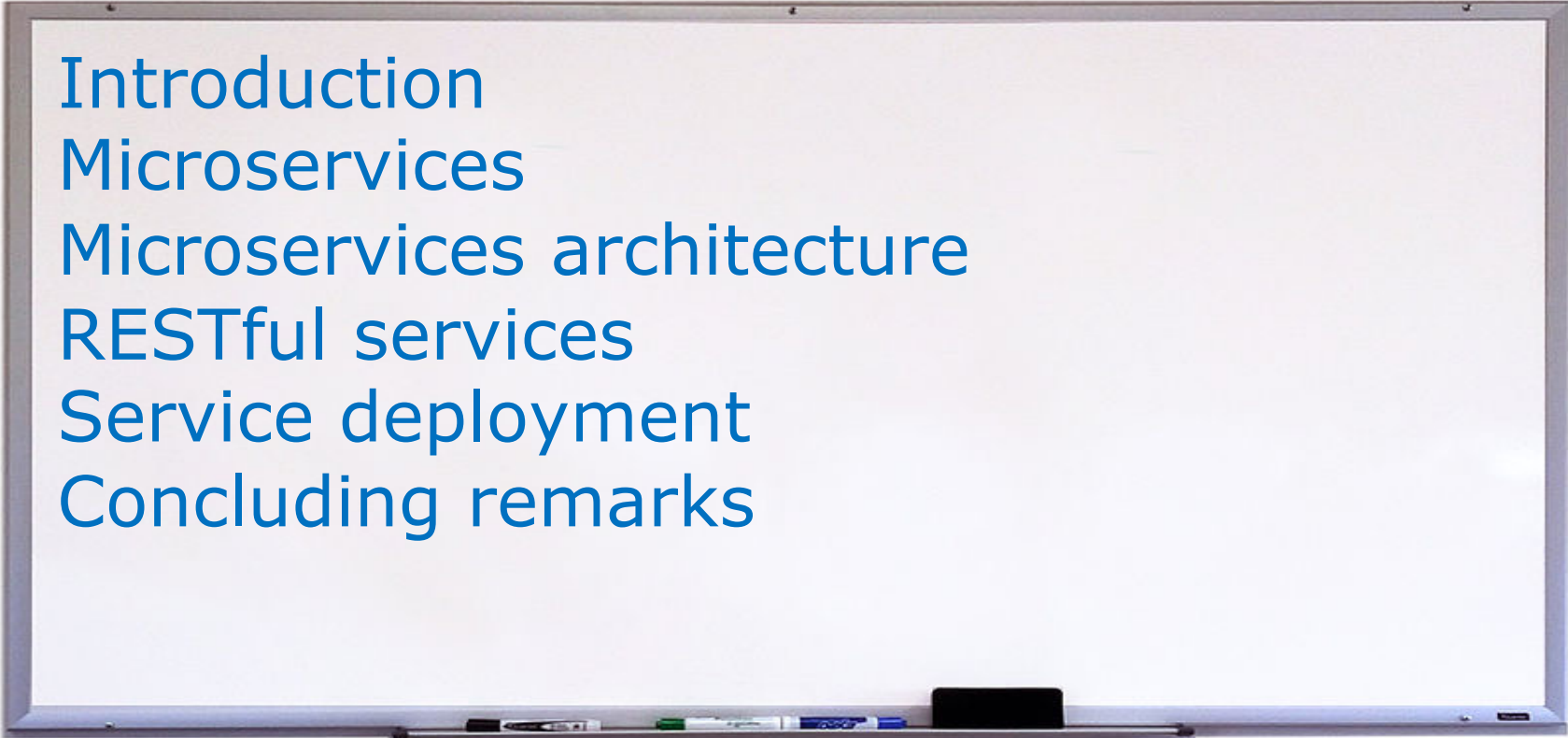
If a service fails, you can roll back to previous version

Example

When you introduce a new version of a service, you maintain the old version but change the “current version link” to point at the new service.

If monitor detects a problem with new version of “Cameras 002” service, it switches the “current version link” back to version 001 of the Cameras service.



A whiteboard with a silver frame and a white surface. On the left side, a list of six topics is written in blue text. At the bottom of the whiteboard, there is a black eraser and several markers in different colors (black, green, blue).

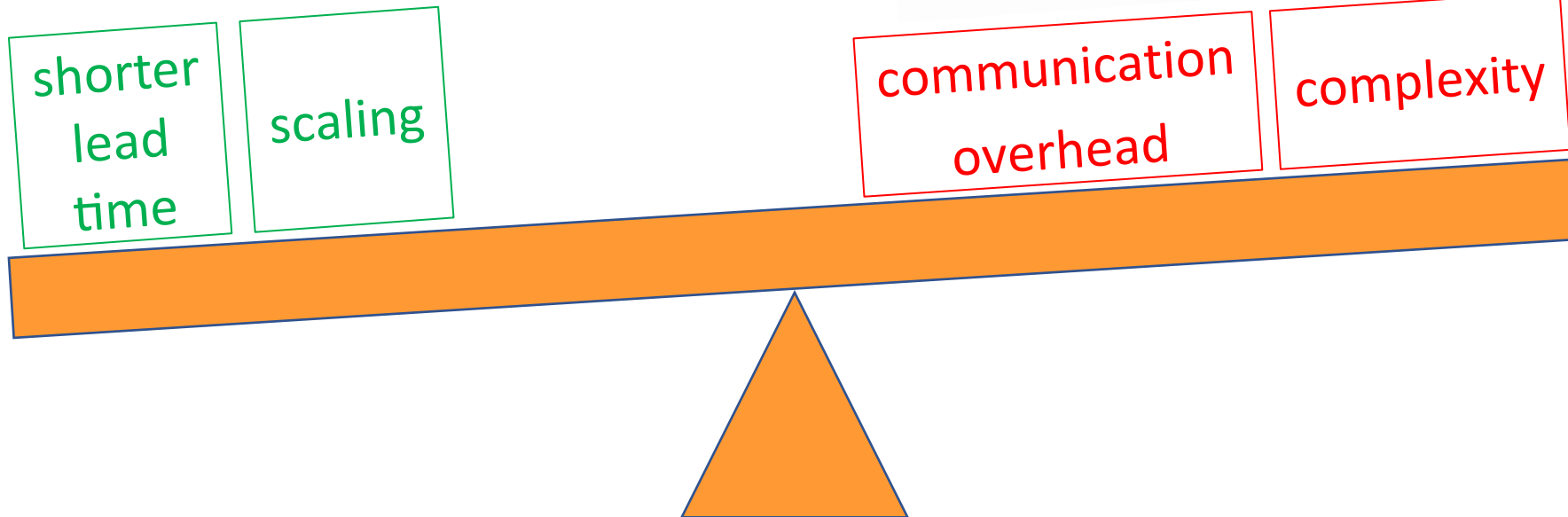
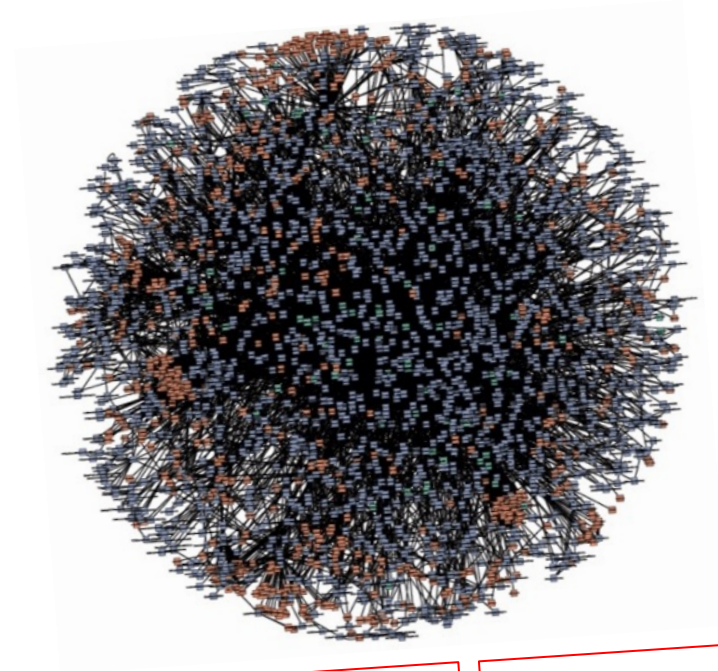
Introduction
Microservices
Microservices architecture
RESTful services
Service deployment
Concluding remarks

Concluding remarks



- Many pros of microservices, including
 - shorter lead time
 - effective scaling
- Cons
 - communication overhead
 - complexity
 - “wrong cuts”
 - “avoiding data duplication as much as possible while keeping microservices in isolation is one of the biggest challenges”
- “A poor team will always create a poor system”

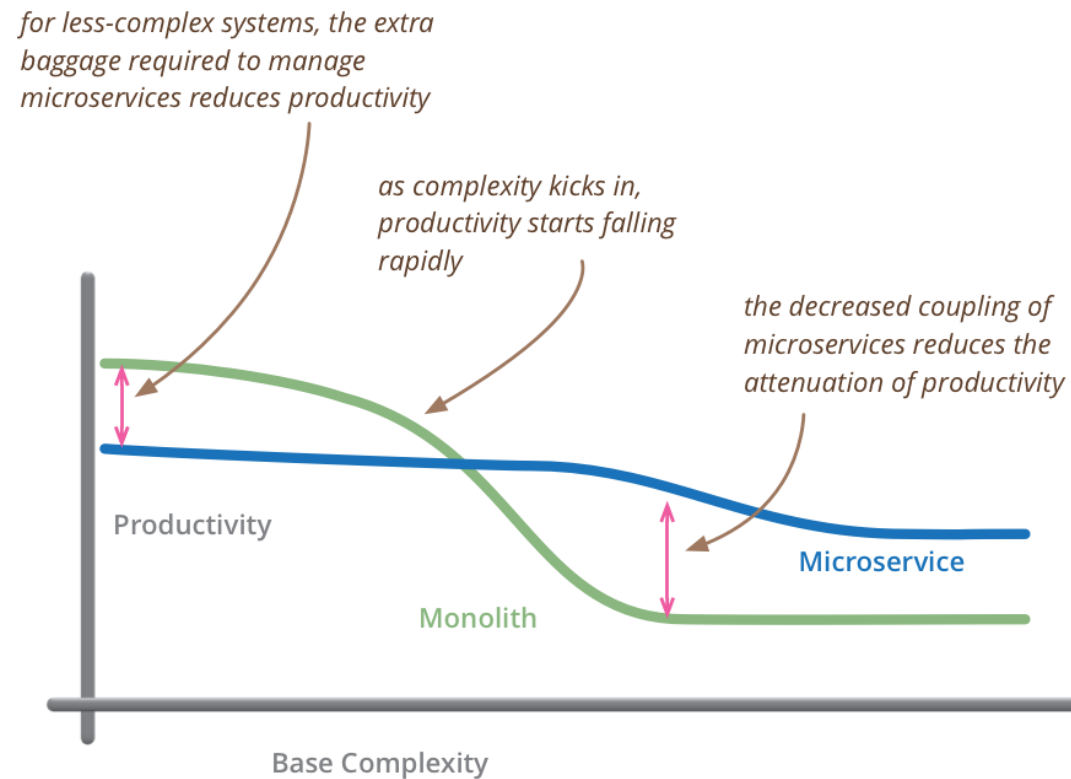
Concluding remarks



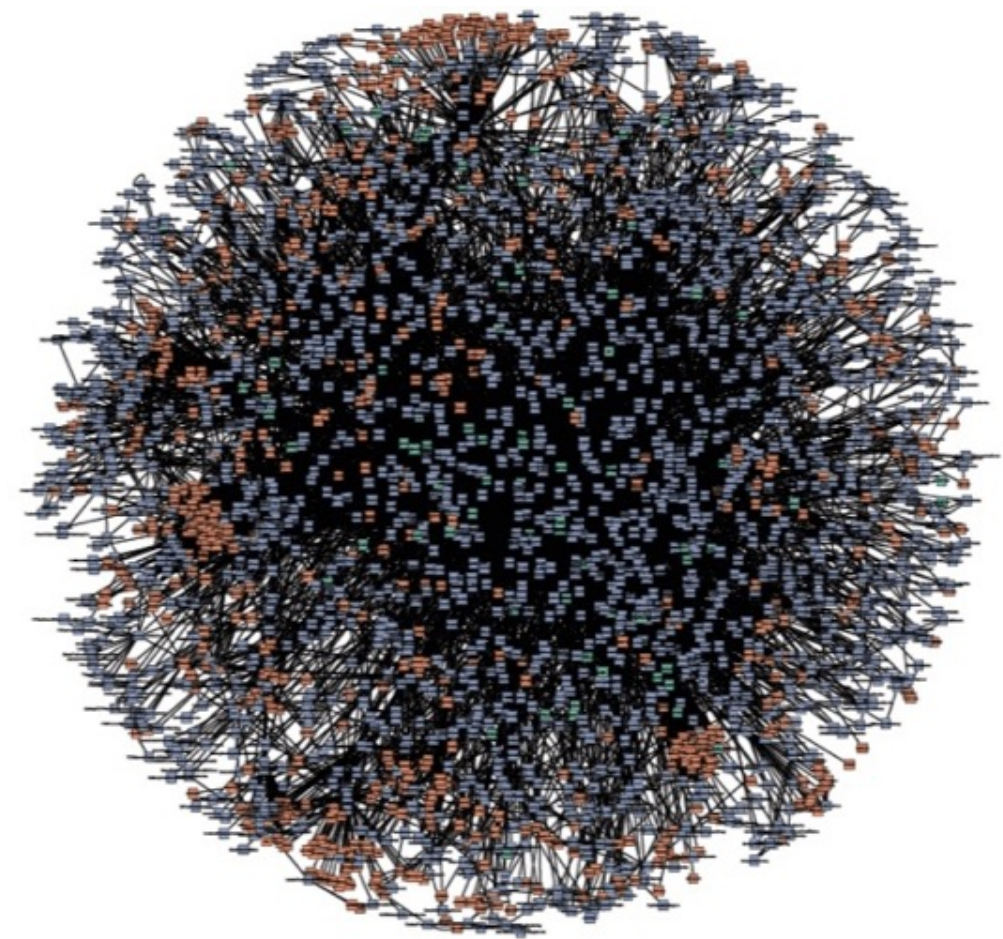
Concluding remarks

Don't even consider microservices unless you have a system that's too complex to manage as a monolith

[M. Fowler]



but remember the skill of the team will outweigh any monolith/microservice choice



amazon

Google

NETFLIX



Spotify



LinkedIn

ebay



GROUPON

Uber

...

e.g. (source www.businessofapps.com)

- 422 million people use Spotify once a month, 182 million are subscribers
- In 2022, Netflix had 222 million subscribers worldwide

Can I play with
microservices?



Reference



Chapter 6 – Microservices Architecture