Micro Service Architecture

A software developer's view on the Cloud

Tobias Abarbanell
CTO, Frontiers Media

What are Micro services?

- Micro service architecture is an software architectural style with
  - Independently deployable services
  - Decentralized control of languages and data

- There are different views of how ”small” the ”Micro” is
  - typical examples are between 10 and hundreds of services in one environment.

Who is using Micro services?

- Netflix example
  - One of the best documented pioneers, writing regularly in their tech blog
- Uber
  - Recently announced they have now over 1’000 Micro services in production

But really most web-scale companies are using Micro services nowadays.

We are not Google – why would we bother?

Development without micro services is easier and faster

But less scalable if your system and team gets bigger.

Software development at scale

Challenges:

- Multiple teams in different locations
- Too complex for everybody to know everything

If you have one shared application code base

- Hard to change
  - too many people affected
  - Too much work to e.g. change the DB version, or the framework
- Hard to test
- Hard to deploy and scale in production

Software development at scale

Solution:

- Break down into multiple applications
- Applications talk to each other over standard interfaces (e.g. REST)
- Each team only needs to know their own application in detail
  - And the API contracts of the ones they use

So with 10 teams you would have 10 subsystems….

Conway’s Law

Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations

— *M. Conway*[1]

https://en.wikipedia.org/wiki/Conway%27s_law

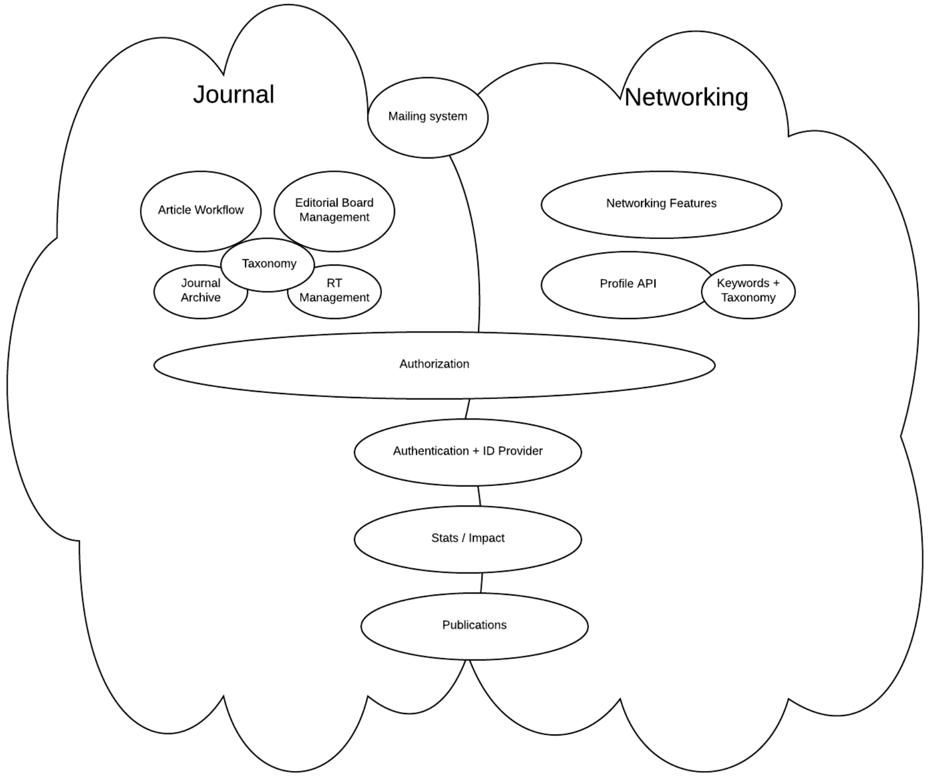
Examples

- If you want your software organization centrally directed, you will create a single application.
- If you want your teams to make independent decisions in their areas, you need to split the application
  - Or you will pay a big coordination overhead
  - We have been there and one of the results was ”code merge hell”

Our Journey at Frontiers

We started with one big team, and consequently one integrated application

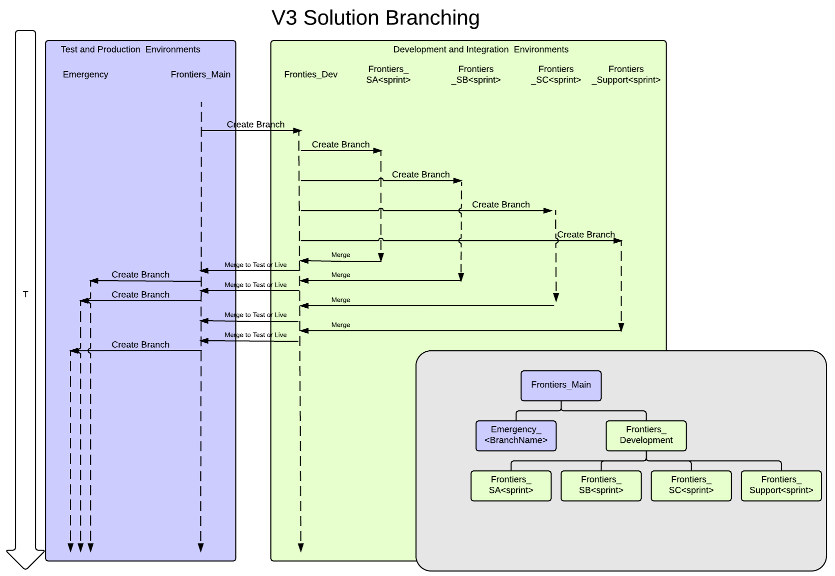
- According to Conway's law, this is what you get naturally
- And it worked for us at that time



Our Journey at Frontiers

Then we grew to teams in 3 countries and we knew we had to change

- Coordinating work on the same application in different time zones is a nightmare
- Branching and merging code for each team's sprint became a complex and time-consuming



Our Journey at Frontiers

More challenges….

- Deployments to live where tightly coupled, making test phases a long exercise
- With our goal of being agile (monthly releases) we spent more time in test than in development and still slipped to 6week cycles

The first idea

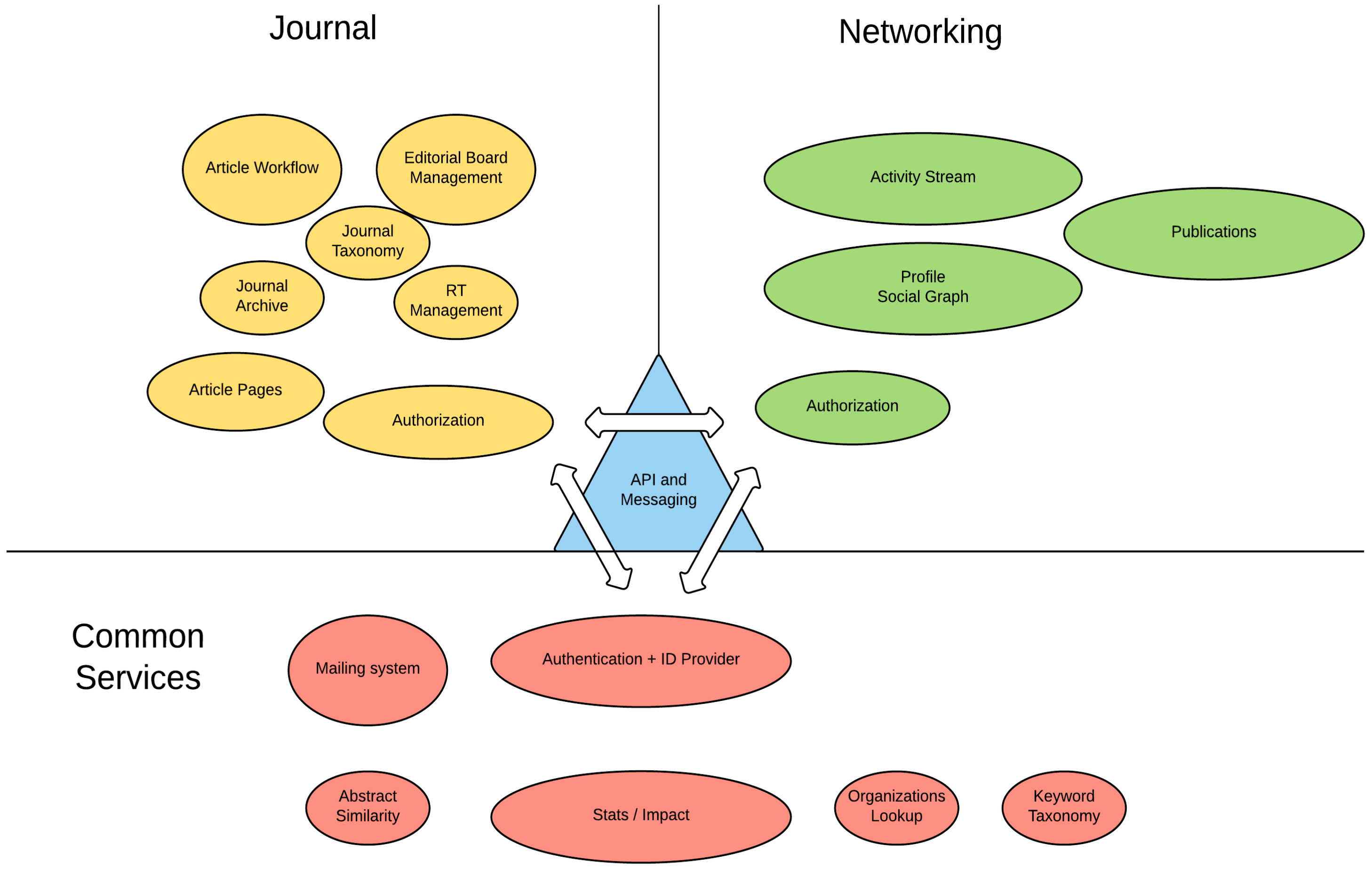
Three locations – split the system in three parts.

But we found that even teams in the same location have too much coordination effort

Then we moved to more subsystems, and it was easier to add new functionality in a new subsystem

- Our old monolithic system was getting smaller, but slowly.
- That is ok because we were always looking for the business value in each change,
  - the old system was over time keeping all the parts which were stable and needed no change at this time.

Overview


Journal
Networking
Article Workflow
Editorial Board Management
Journal Taxonomy
Journal Archive
RT Management
Article Pages
Authorization
Activity Stream
Publications
Profile Social Graph
Authorization
API and Messaging
Common Services
Mailing system
Authentication + ID Provider
Abstract Similarity
Stats / Impact
Organizations Lookup
Keyword Taxonomy

What we learned

- By our own experience
- But also by looking at others

Microservices

- Why? –
  - because a multi-team development on a monolithic system incurs too much coordination overhead between the teams.
  - Because we want product managers to deliver their products as independent as possible from each other
- How? –
  - Gradually separate monolith into multiple subsystems (currently 3-5 per team) which can be evolved independently.
  - Usually related Web pages and their backing API's are in separate subsystems.
  - Pages often consume multiple services.

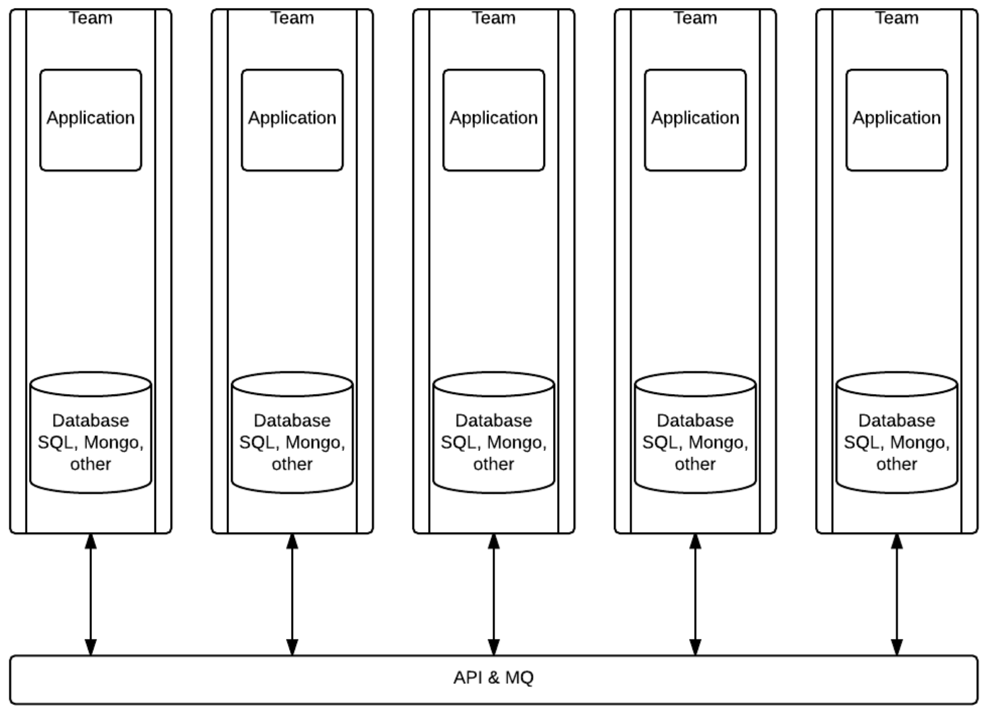
Micro service rules (1)

- Each service contains its own
  - presentation,
  - business logic,
  - data (persistence layer)
- Service communicate via standard interfaces
  - API (Push messages to an endpoint)
  - MQ (Pull message from a subscribed topic / queue)

Micro service rules (2)

- Backwards compatibility
  - Each ”breaking change” on the interfaces needs to be managed with the clients in mind:
    - Keep the old version running until all clients are upgraded
    - Think of your micro service clients as external customers even if they are in your company
  - This creates additional work, called sometimes the ”Micro service tax”
    - In effect you pay this tax on every development, for reaching scalability to bigger organizations

ARCHITECTURE - PURE MICROSERVICES


Team
Application
Database SQL, Mongo, other
Team
Application
Database SQL, Mongo, other
Team
Application
Database SQL, Mongo, other
Team
Application
Database SQL, Mongo, other
Team
Application
Database SQL, Mongo, other
API & MQ

Micro services and the Cloud

- Micro services could be used without cloud computing
- But only the agility of the cloud (change resources and pay as you go) matches the agility of the development (create new services almost weekly)

Micro services and the Cloud

- Micro services are perfect in the cloud if you can extract some generic services and “buy” or “rent” instead of “make”.
  - Examples: identity service,
  - API gateway,
  - mail service,
  - error logging service,
  - message queue

Micro services and the Cloud

- But there is also a great opportunity to use cloud services inside your micro services:
    - Cache and Persistence services like redis, mongodb, Azure SQL

Objective – increase release frequency

- High release frequency only possible with automation of
  - deployments (to test and live)
  - testing (functional, performance)
  - monitoring
  - Even automated rollback in case of problems
    - We are not there yet….

How do we manage releases?

We use a go-live calendar as planning tool:

- Release Dates and UAT phases are planned at more or less fixed intervals (about weekly)
- We forecast monthly which teams will have something ready for the begin of the UAT phase
- At the begin of the UAT we verify readiness
  - teams which are not ready get moved to a later UAT slot
- Then the group of teams goes through UAT days until signoff
- And goes live the day after signoff

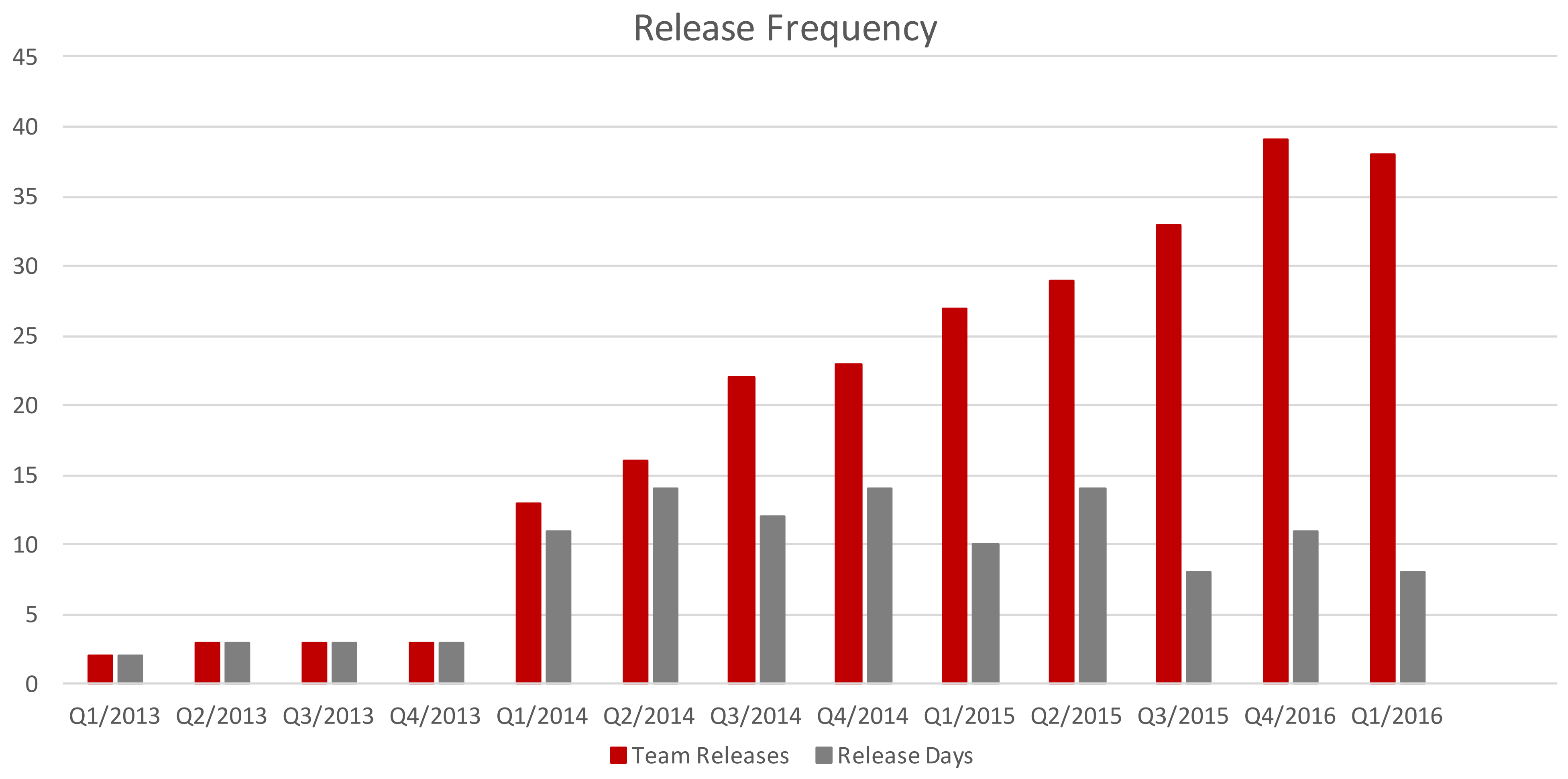
Release and Test Calendar

	Monday	Tuesday	Wednesday	Thursday	Friday
Week 1	UAT: team 1,3	UAT: team 1,3	UAT: team 1,3	signoff: team 1,3	Go-live: team 1,3
Week 2	UAT: team 2,4,5,6	UAT: team 2,4,5,6	UAT: team 2,4,5,6	UAT: team 2,4,5,6	UAT: team 2,4,5,6
Week 3	Signoff: team 2,4,5,6	Go-live: team 2,4,5,6	UAT: team 1,3,7,8	UAT: team 1,3,7,8	UAT: team 1,3,7,8
Week 4	UAT: team 1,3,7,8	UAT: team 1,3,7,8	signoff: team 1,3,7,8	Go-live: team 1,3,7,8	UAT: team 1,4
Week 5	UAT: team 1,4	Signoff: Team 1,4	Go-live: Team 1,4	<holiday>	…
…					

Release Trains?

- We think of this calendar as a train timetable:
    - if you miss the train, you take the next one. The other passengers in the train don't wait.
- Note that this is opposite from the SAFe concept of Release Trains
    - where teams join together on the train, for all the stations (sprints) in sequence
- This was too rigid for us. We needed more flexibility for our teams.

Increased release frequency



Summary

- Micro service Architecture is a good fit for development with multiple teams
  - You can scale release frequency with the bigger number of teams and services
  - This is only realistic with cloud servers and cloud services
- But it adds overhead which is costly if you do not need the scale benefits
- Monolith architecture still has a place
  - As a starting point for quick time to market
  - And if you first have to find the boundaries of your services
  - For applications which are expected to stay small

Thank you!

Tobias Abarbanell (tobias@abarbanell.de)

Blog: blog.abarbanell.de

Slides downloadable from blog.

That was too easy, right?

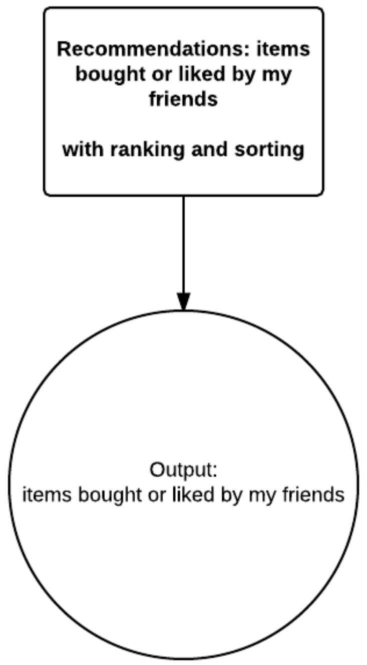
Lets look at some edge cases…

Challenges – integration tests

- Testing a single service is easy (because it is small) but testing a group of services together is difficult
  - Even more if the systems need to have consistent data for a test case.

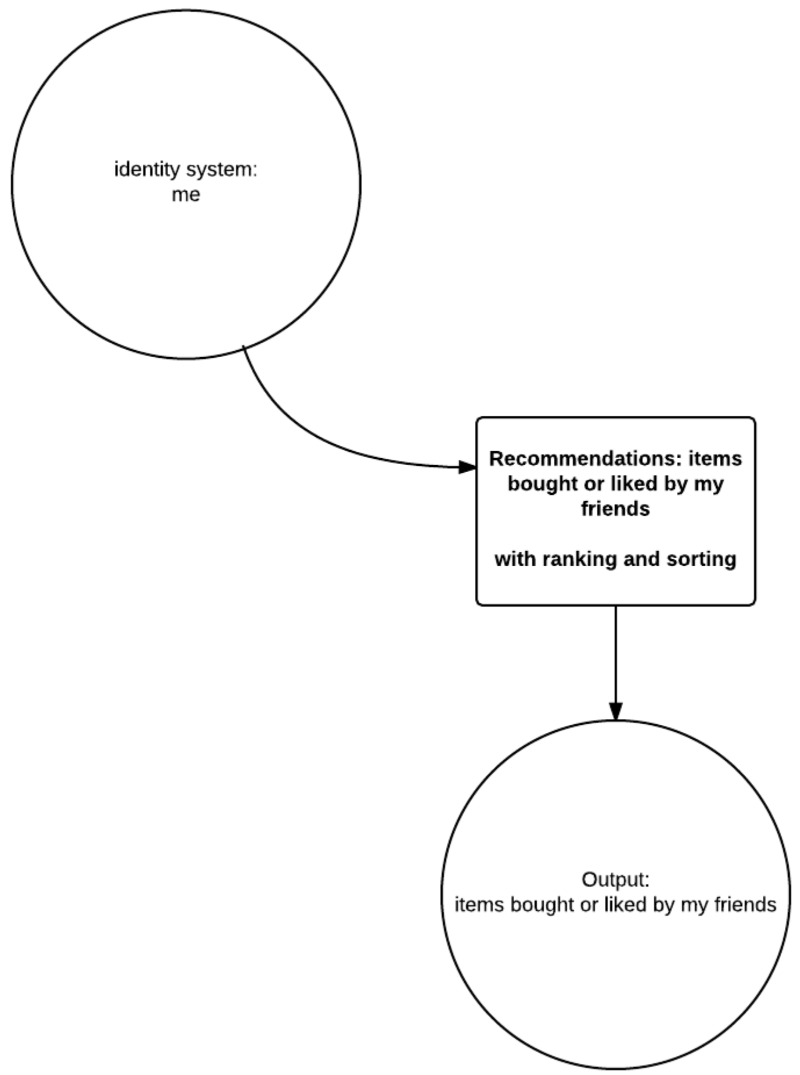
MICROSERVICES - TEST SCENARIO

Example of a connected test scenario
This requires data in several subsystems to "play" together


Recommendations: items bought or liked by my friends
with ranking and sorting
Output:
items bought or liked by my friends

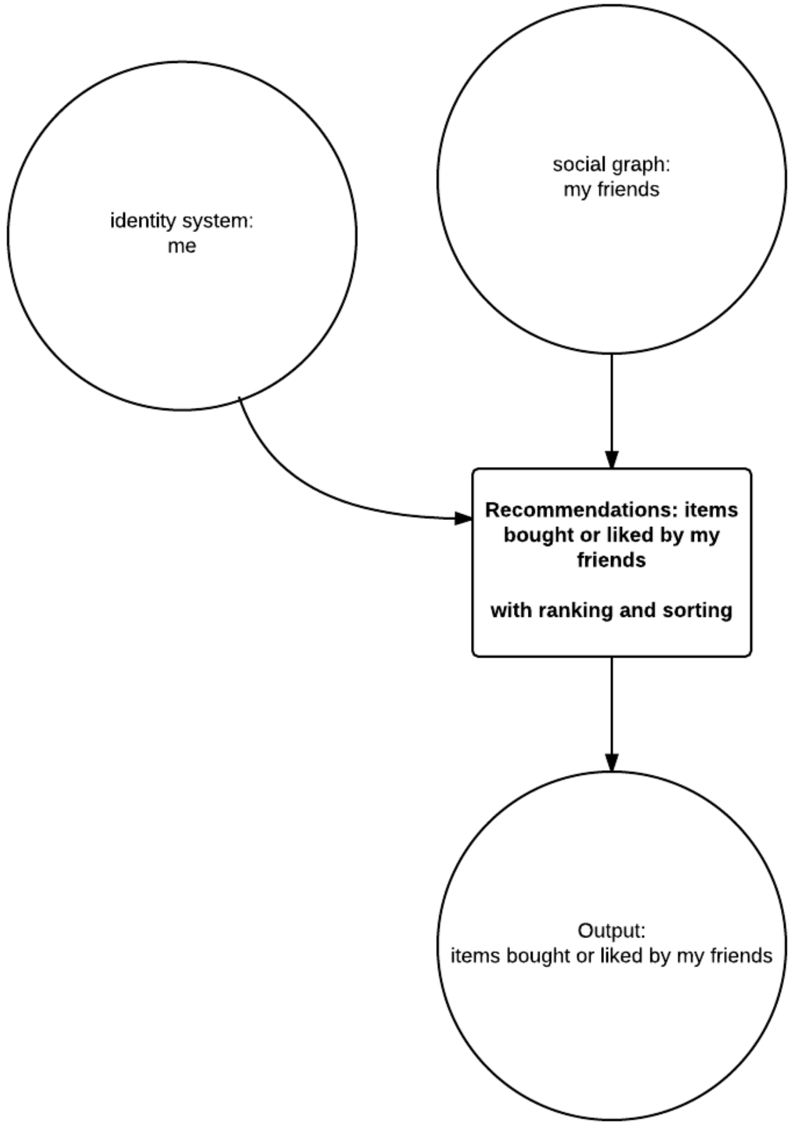
MICROSERVICES - TEST SCENARIO

Example of a connected test scenario
This requires data in several subsystems to "play" together


identity system:
me
Recommendations: items bought or liked by my friends
with ranking and sorting
Output:
items bought or liked by my friends

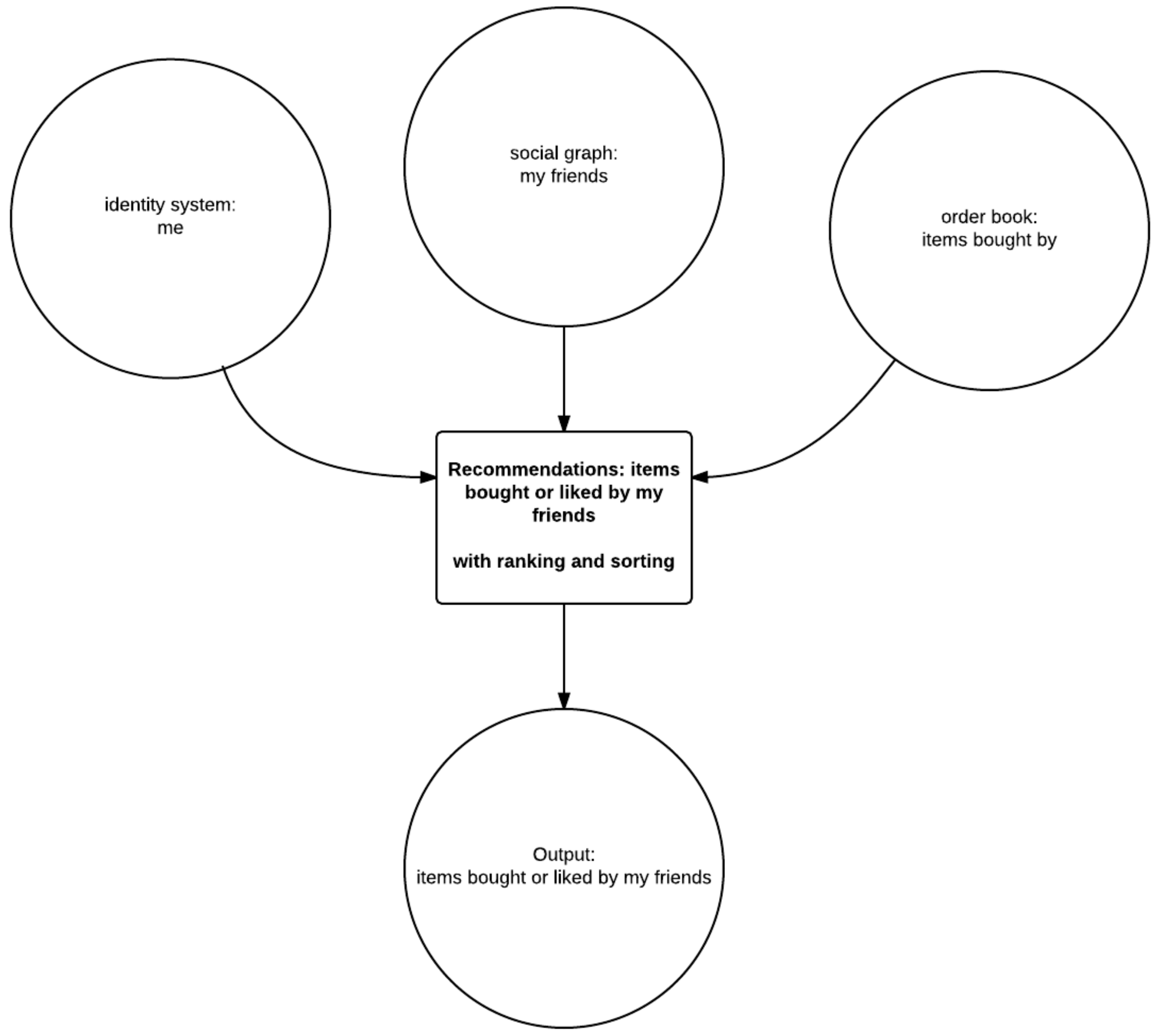
MICROSERVICES - TEST SCENARIO

Example of a connected test scenario
This requires data in several subsystems to "play" together

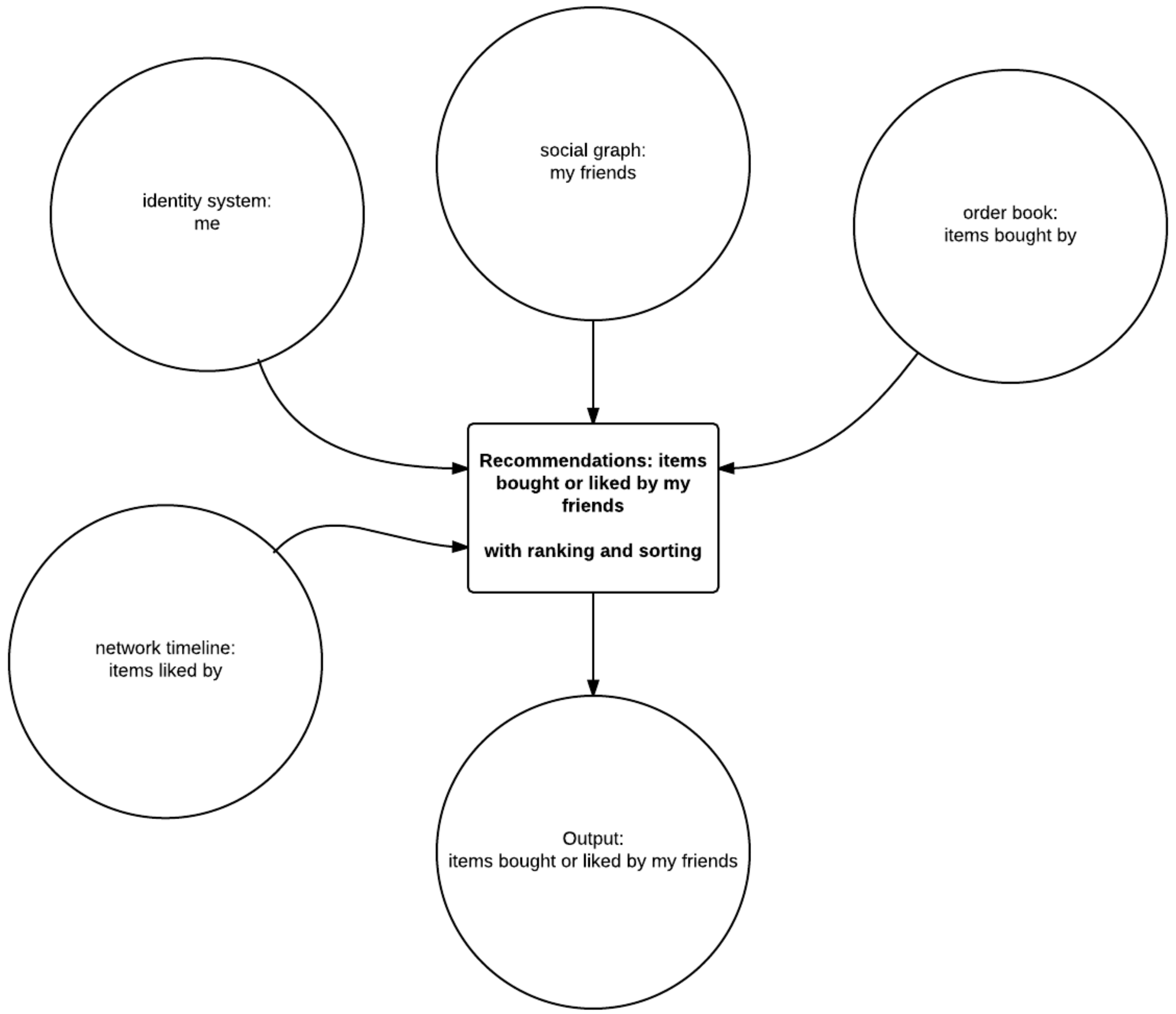

identity system:
me
social graph:
my friends
Recommendations: items bought or liked by my friends
with ranking and sorting
Output:
items bought or liked by my friends

MICROSERVICES - TEST SCENARIO

Example of a connected test scenario
This requires data in several subsystems to "play" together

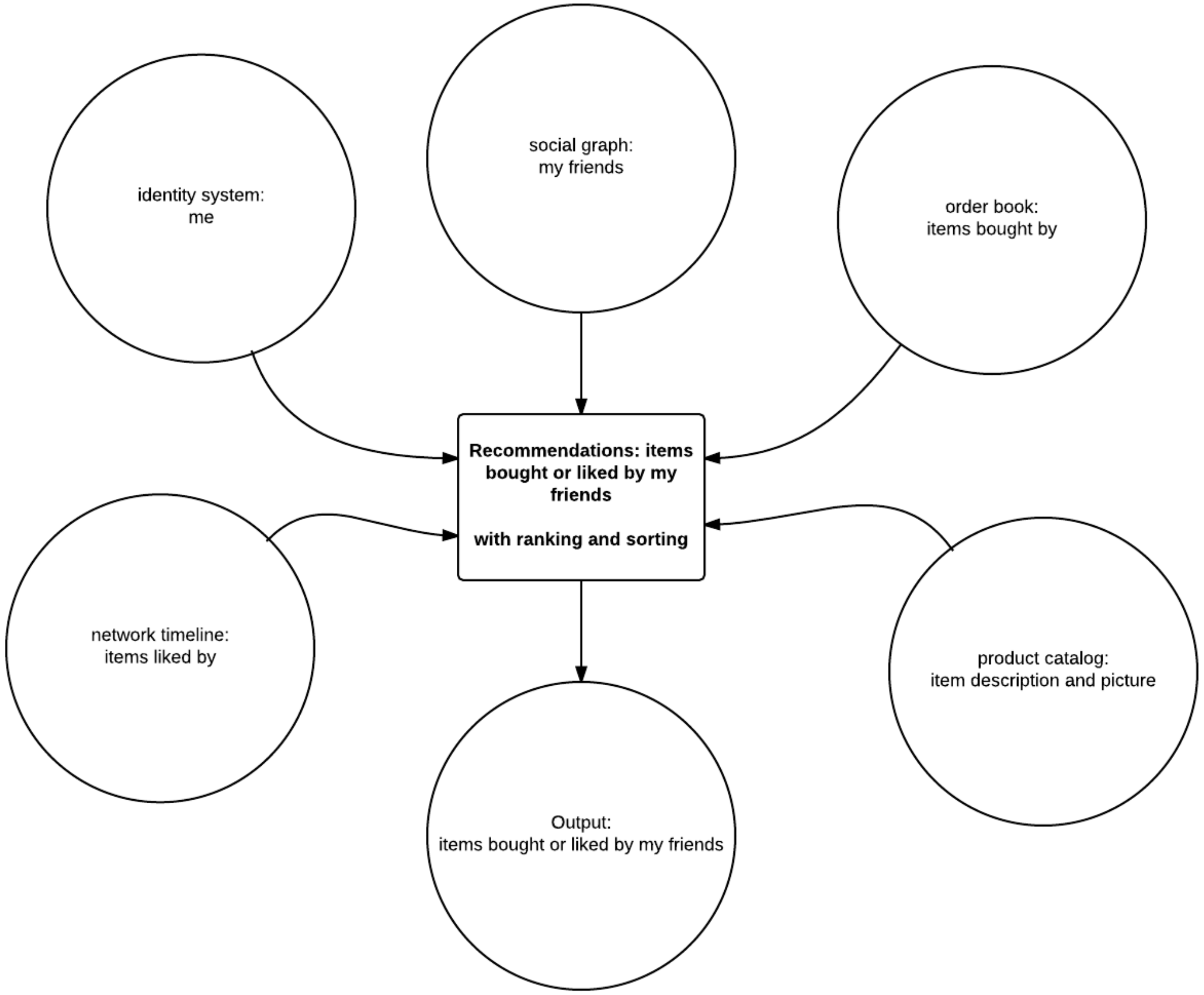


Example of a connected test scenario
This requires data in several subsystems to "play" together



MICROSERVICES - TEST SCENARIO

Example of a connected test scenario
This requires data in several subsystems to "play" together



Solutions for integration tests

We currently do not have a solution which we find recommendable

- Testing only contracts:
  - could miss the big picture of dependencies in the functionality,
  - Mock data for the tests is time consuming to construct
  - test look artificial and are difficult to evaluate
- testing on a copy of production data:
  - easy to understand,
  - but you may miss scenarios which do not exist in live,
  - and it requires a lot of resources (time, money, test system capacity)
  - Majority of our cloud servers are in integration environments for the teams

Challenges – chatty API’s

- There can be a lot of traffic if the API’s are not aligned with the needs of the clients
  - A simple CRUD interface is not good to keep a data warehouse updated
  - Data may need to be stored in multiple places to make the access more efficient
  - But this could introduce consistency problems

Challenges – code duplication

- All services need some basic features like
  - Authentication
  - Error logging
  - Application monitoring
- Solutions
  - Template projects
  - Code libraries (nuget, npm, etc.)

Thank you!

Tobias Abarbanell (tobias@abarbanell.de)

Blog: blog.abarbanell.de

Slides downloadable from blog.