Architectural smells and refactorings for microservices

Antonio Brogi

Department of Computer Science
University of Pisa

Microservices

RECALL

Motivations

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

(2) Scale, effectively

amazon Google NETFLIX Spotify twitter LinkedIn ebay zalando GROUPON Uber ...

Applications = sets of services

+ each running in its own container
+ communicating with lightweight mechanisms
+ built around business capabilities
+ decentralizing data management
+ independently deployable
+ horizontally scalable
+ fault resilient
+ DevOps culture & tools

Ok, got it

Now, does my app respect the "microservices principles"?

And if not, how can I refactor it?

Principles, smells and refactorings

How can architectural **smells** affecting design **principles** of microservices be detected, and then resolved via **refactoring**?

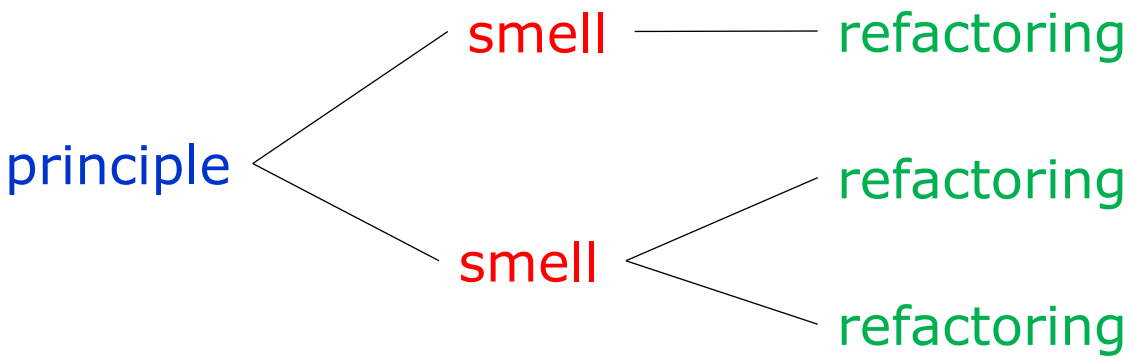
"An architectural smell is a commonly used architectural decision that negatively impacts system lifecycle qualities."

A. Garcia, D. Popescu, G. Edwards, M, Medvidovic. Toward a Catalogue of Architectural Bad Smells. QoSA 2009.

A multivocal review

Review of white and grey literature (41 references) aimed at identifying

- the most recognised ***architectural smells*** for microservices, and
- the architectural ***refactorings*** to resolve them



Design principles

Independent deployability

The microservices forming an application should be independently deployable

Horizontal scalability

The microservices forming an application should be horizontally scalable

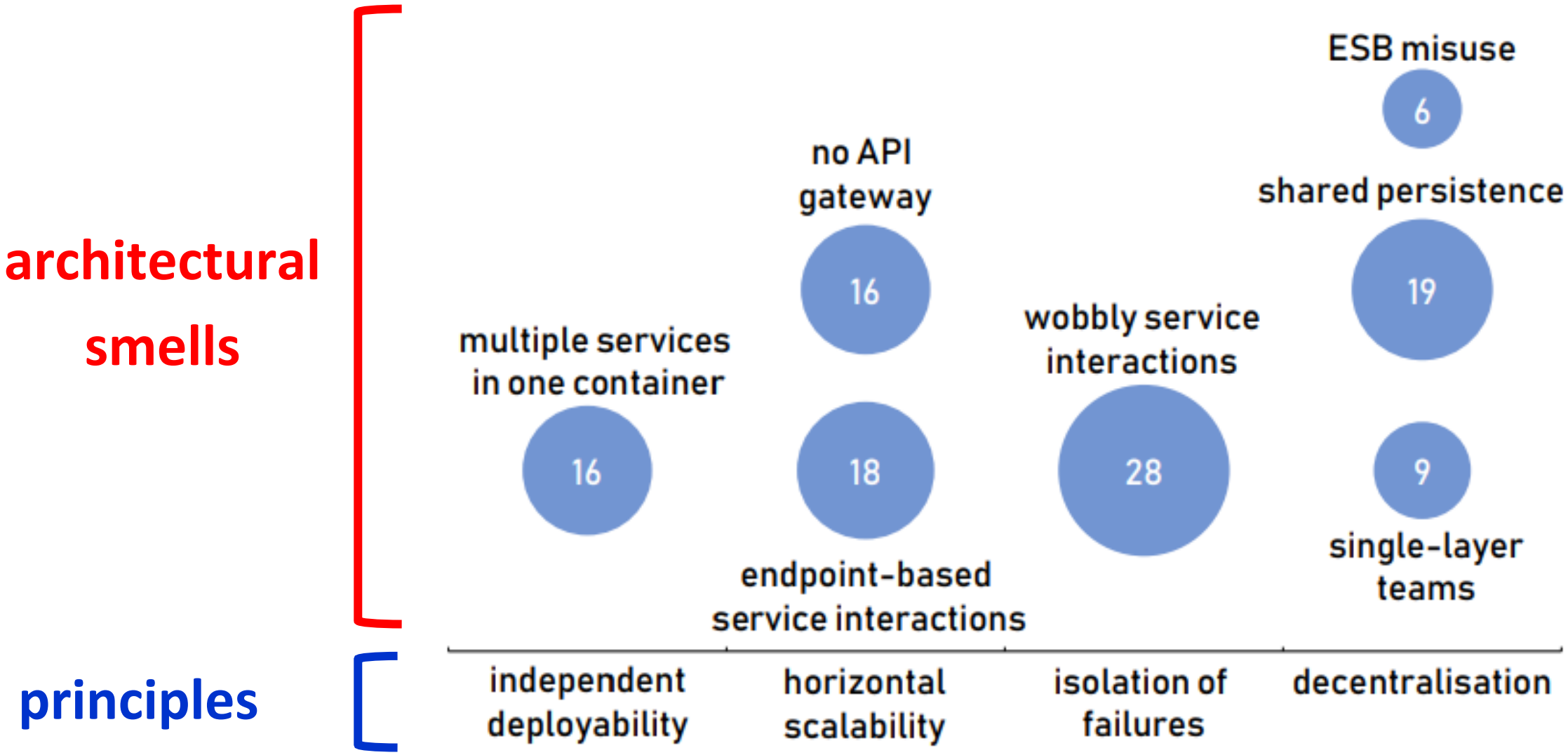
→possibility of adding/removing replicas of single microservices

Isolation of failures

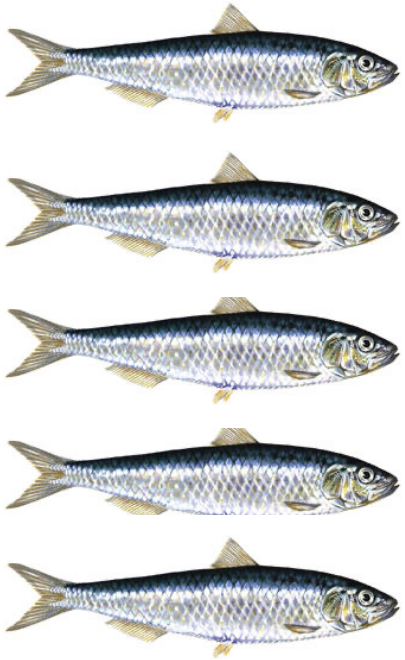
Failures should be isolated

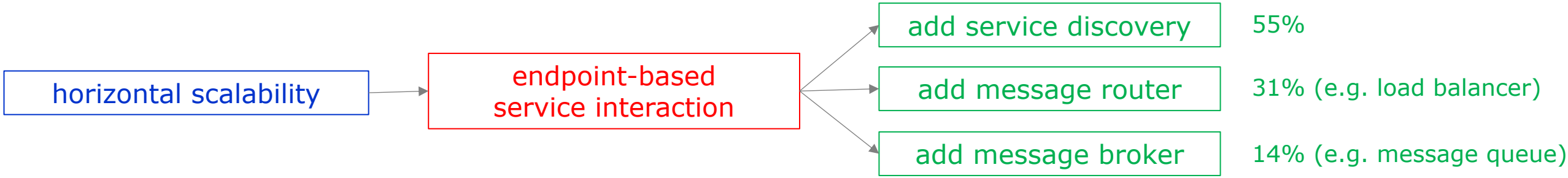
Decentralization

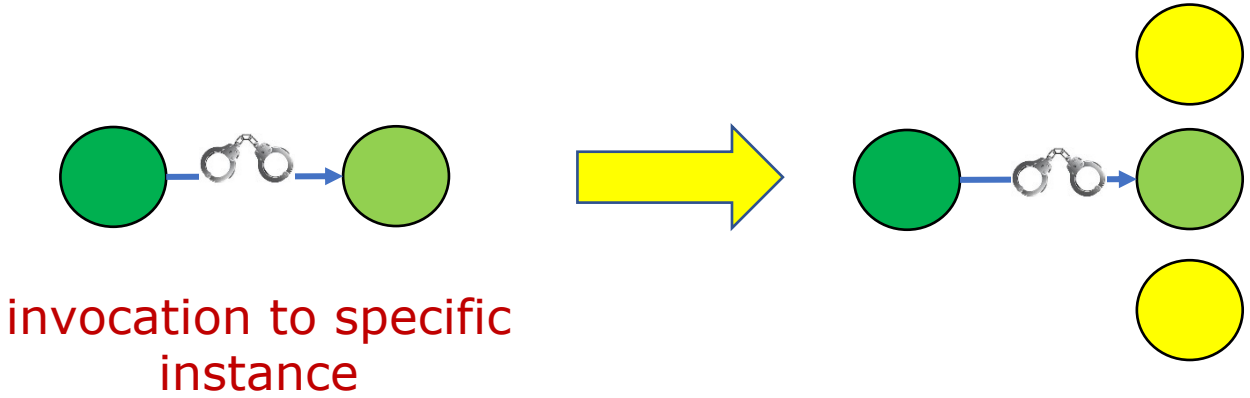
Decentralisation should occur in all aspects of microservice-based applications, from data management to governance


architectural smells
principles
multiple services in one container
16
no API gateway
16
endpoint-based service interactions
18
wobbly service interactions
28
ESB misuse
6
shared persistence
19
single-layer teams
9
independent deployability
horizontal scalability
isolation of failures
decentralisation

independent deployability → multiple services in one container → package each service in a separate container




horizontal scalability
endpoint-based service interaction
add service discovery
55%
add message router
31% (e.g. load balancer)
add message broker
14% (e.g. message queue)


invocation to specific instance

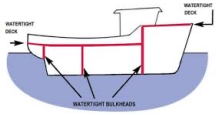
App clients invoke directly app services

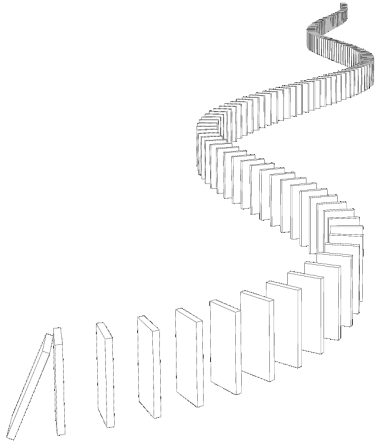
(similar to endpoint-based service interaction smell)

(Adding API gateway can be useful also for authentication, throttling, …)

isolation of failures → wobbly service interaction

- add message broker — 16%
- add circuit breaker — 42%
- use timeouts — 22%
- add bulkhead — 20%

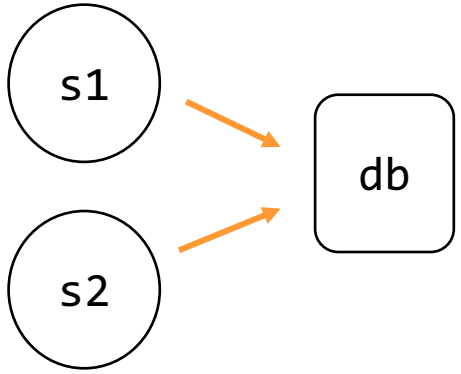




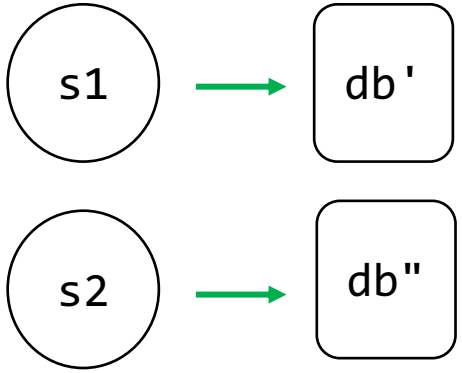
The interaction of m1 with m2 is *wobbly*
when a failure of m2 can trigger a failure of m1


decentralisation
shared persistence
split database
50%
add data manager
41%
merge services
9%

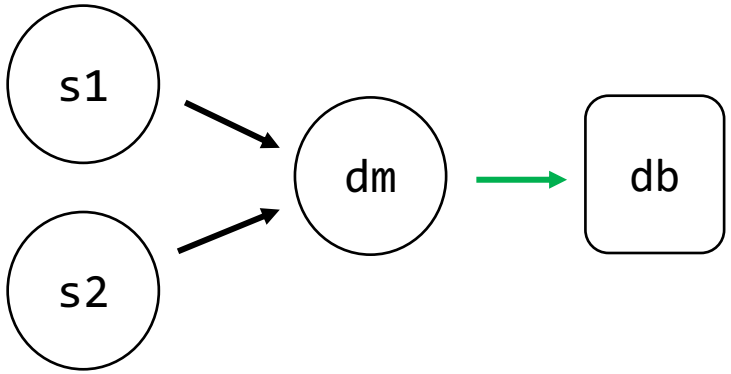

Multiple services access same DB



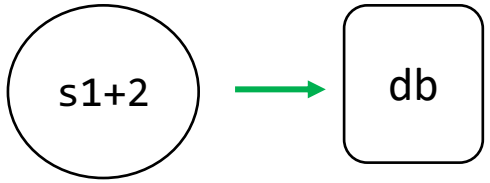
shared persistence



split database
- db splitted
- small changes to s1,s2
- not always possible/easy to implement
- eventual data consistency for replicated data

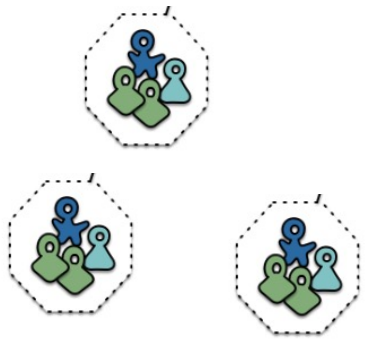


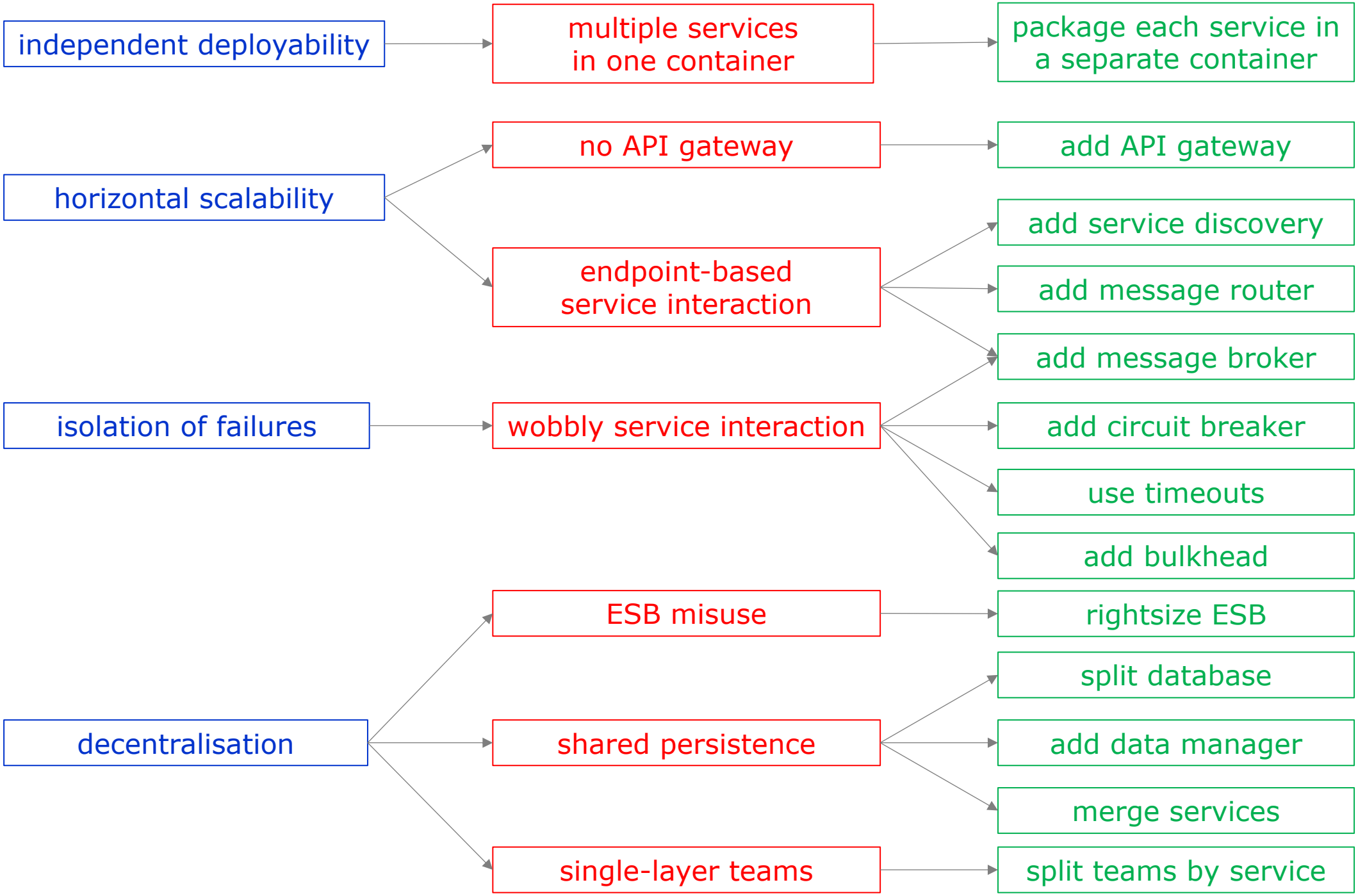
add data manger
- dm added
- very small changes to s1,s2
- added communication overhead



merge services
- s1 and s2 merged into single service
- not always easy/reasonable to implement

decentralisation → single-layer teams → split teams by service




independent deployability
multiple services in one container
package each service in a separate container
horizontal scalability
no API gateway
add API gateway
endpoint-based service interaction
add service discovery
add message router
add message broker
isolation of failures
wobbly service interaction
add circuit breaker
use timeouts
add bulkhead
decentralisation
ESB misuse
rightsize ESB
shared persistence
split database
add data manager
merge services
single-layer teams
split teams by service

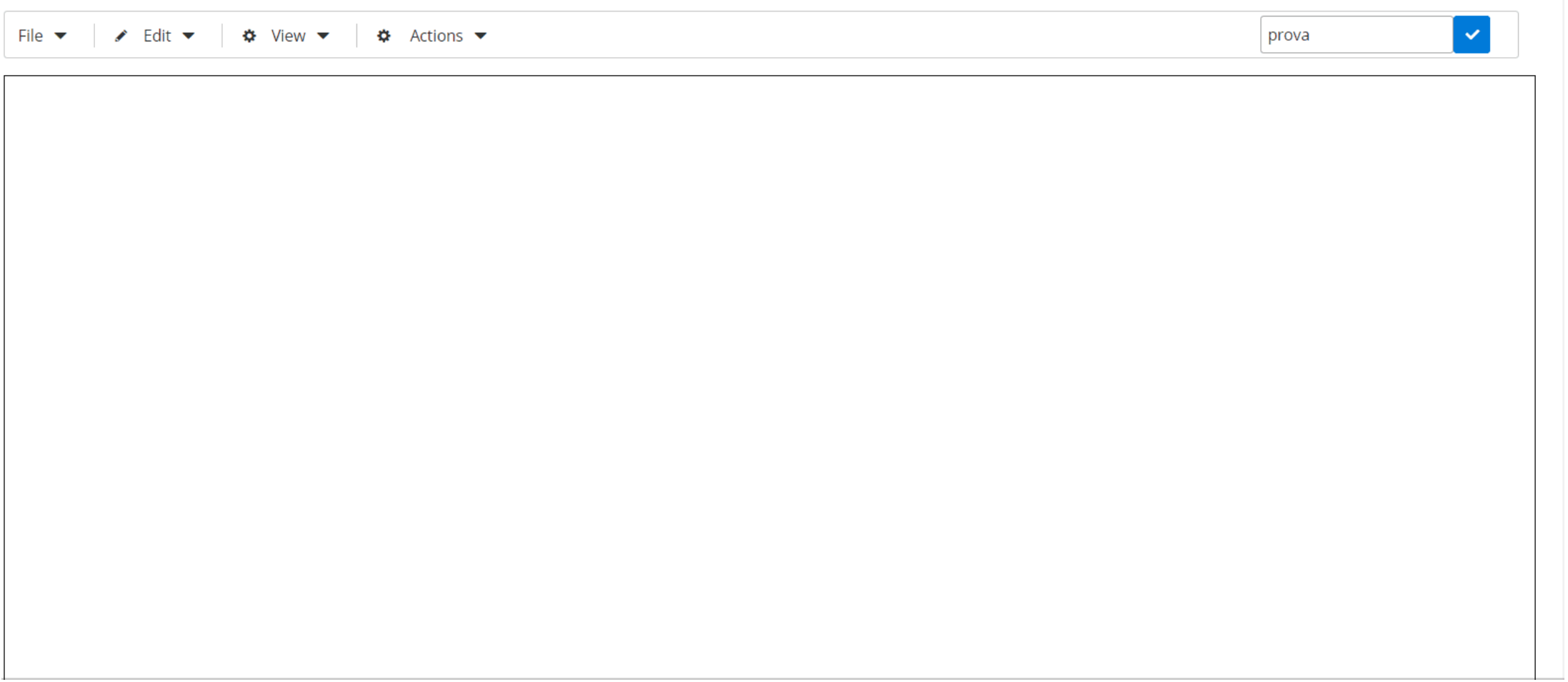
Principles, smells and refactorings

MicroFreshener

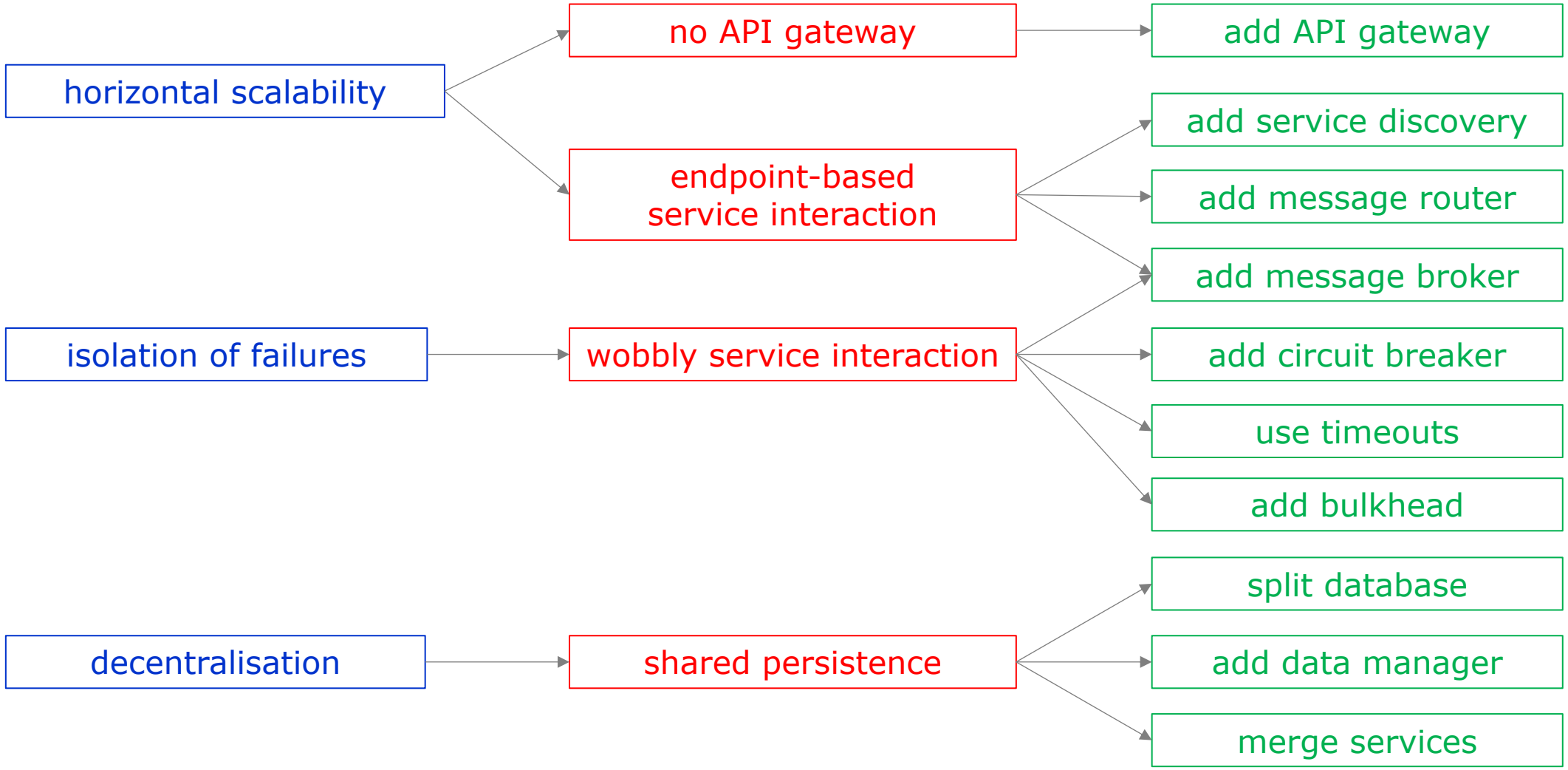
MicroFreshener

A tool for

- editing app specifications
- automatically identifying architectural smells
- applying architectural refactorings to resolve the identified smells

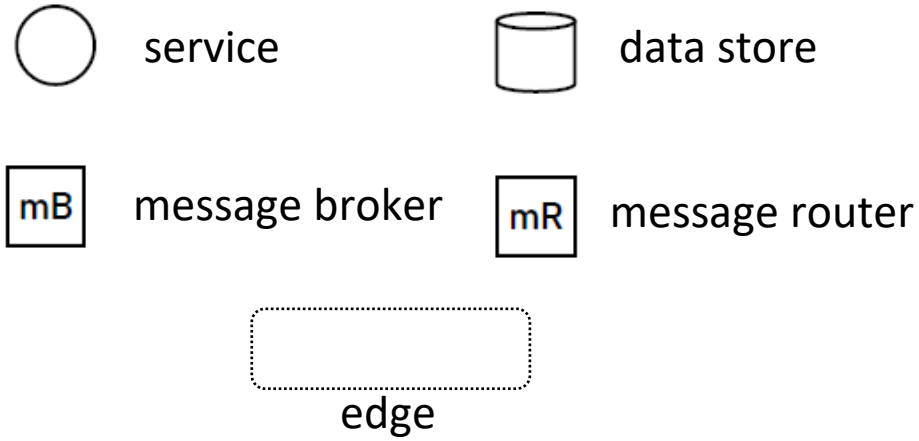


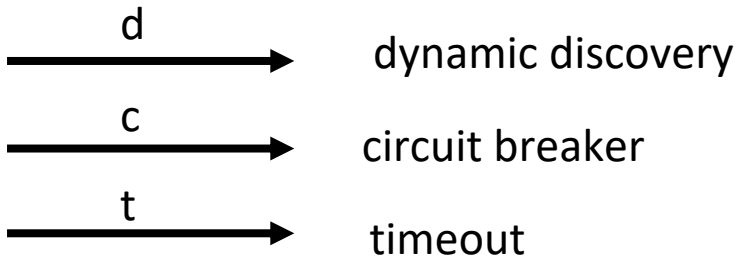
Excerpted taxononomy



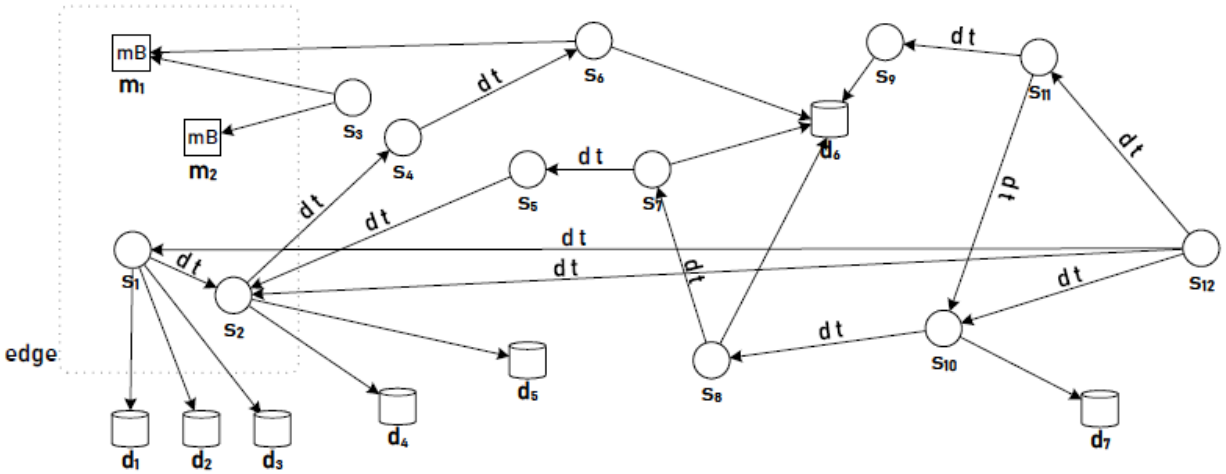
Modelling application architecture

Graphical representation

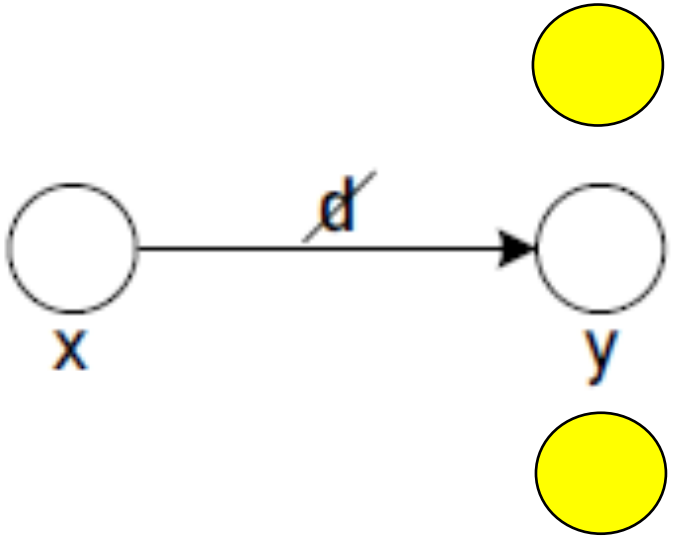


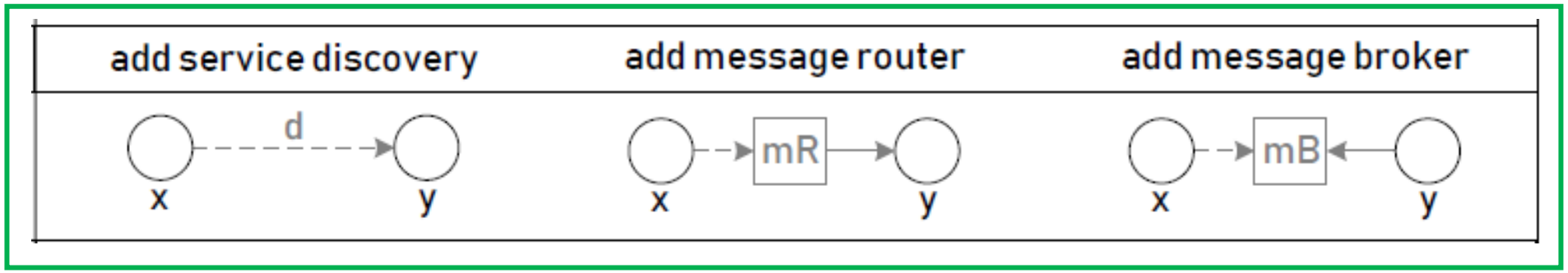


Example

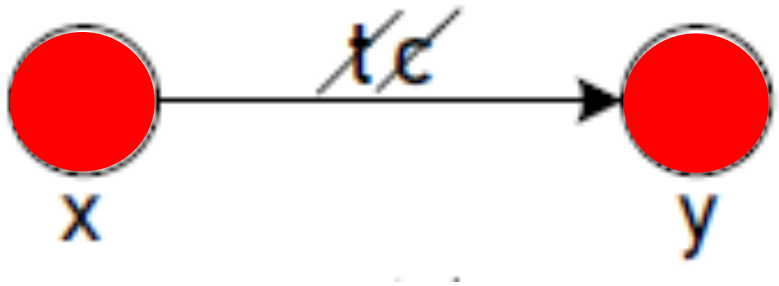


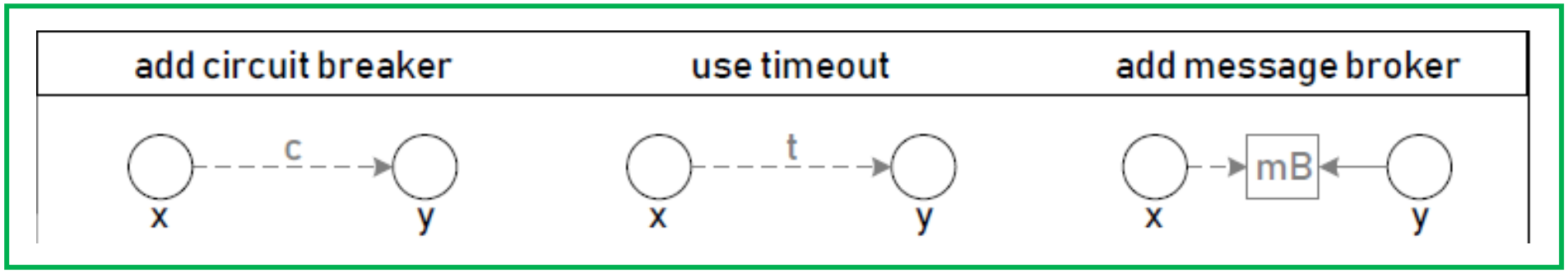
endpoint-based service interaction



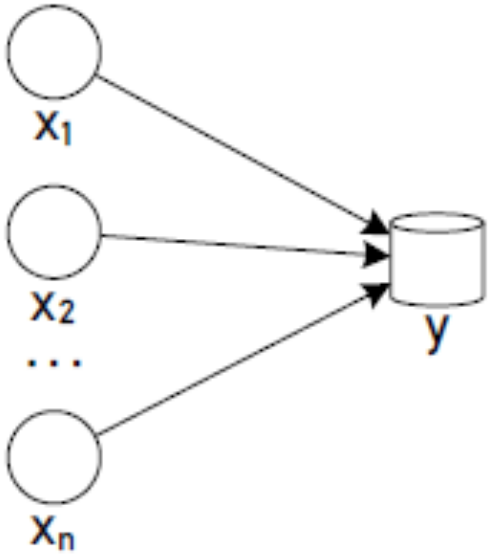


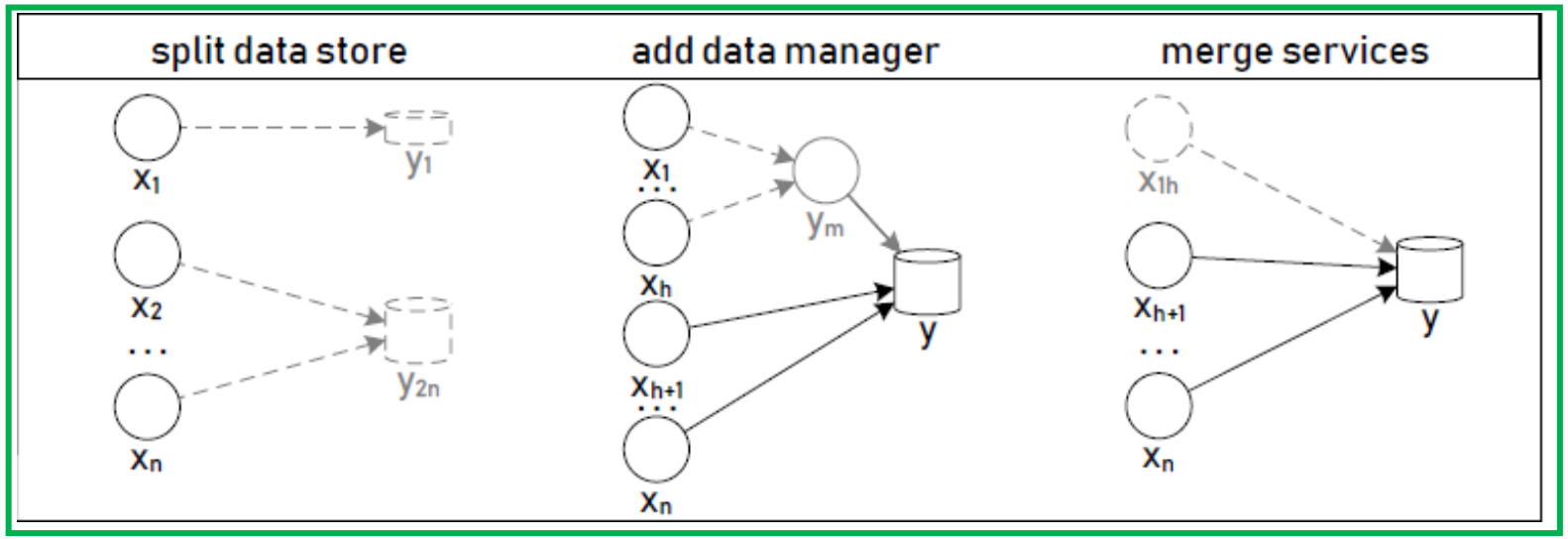
wobbly service interaction





shared persistence

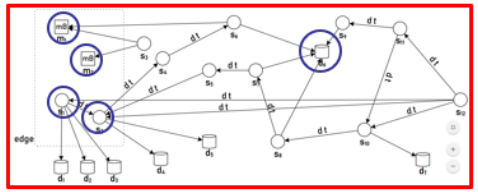




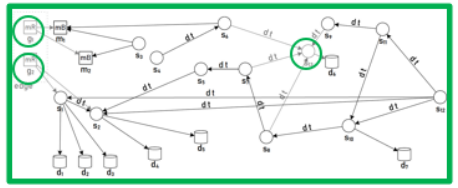
MicroFreshener

- (freely) usable to analyse & refactor microservice-based apps

- industrial case study



- 4 no API gateway smells
- 1 shared persistence smell



- 2 API gateways added
- 1 data manager added

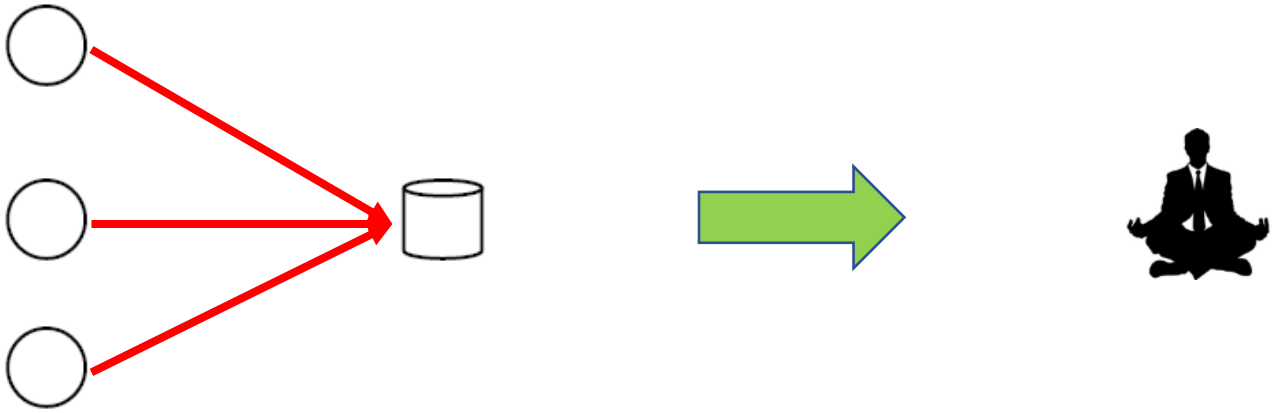
- controlled experiment
  - 100% vs. 49% smells identified
  - 83% vs. 1% participants resolved all smells

Principles, smells and refactorings

MicroFreshener

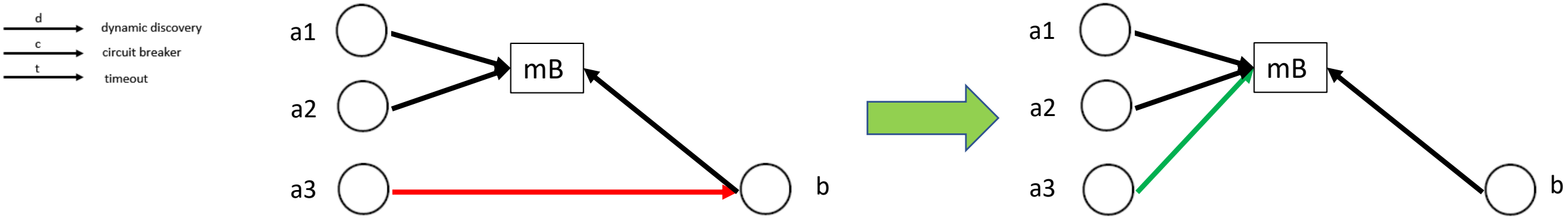
Remarks

«Let it be» refactoring supported



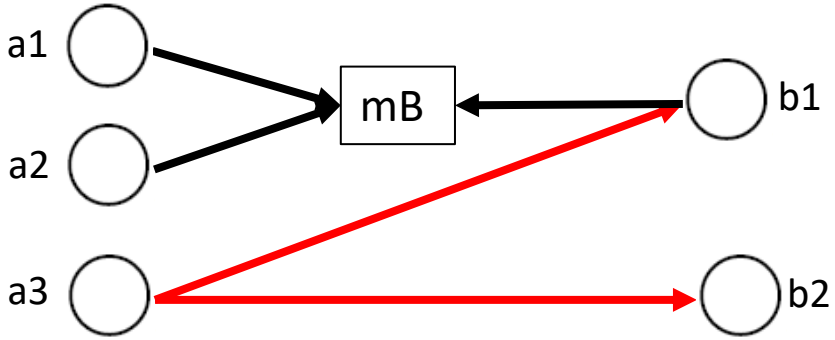
Architecture level ≠ Implementation level

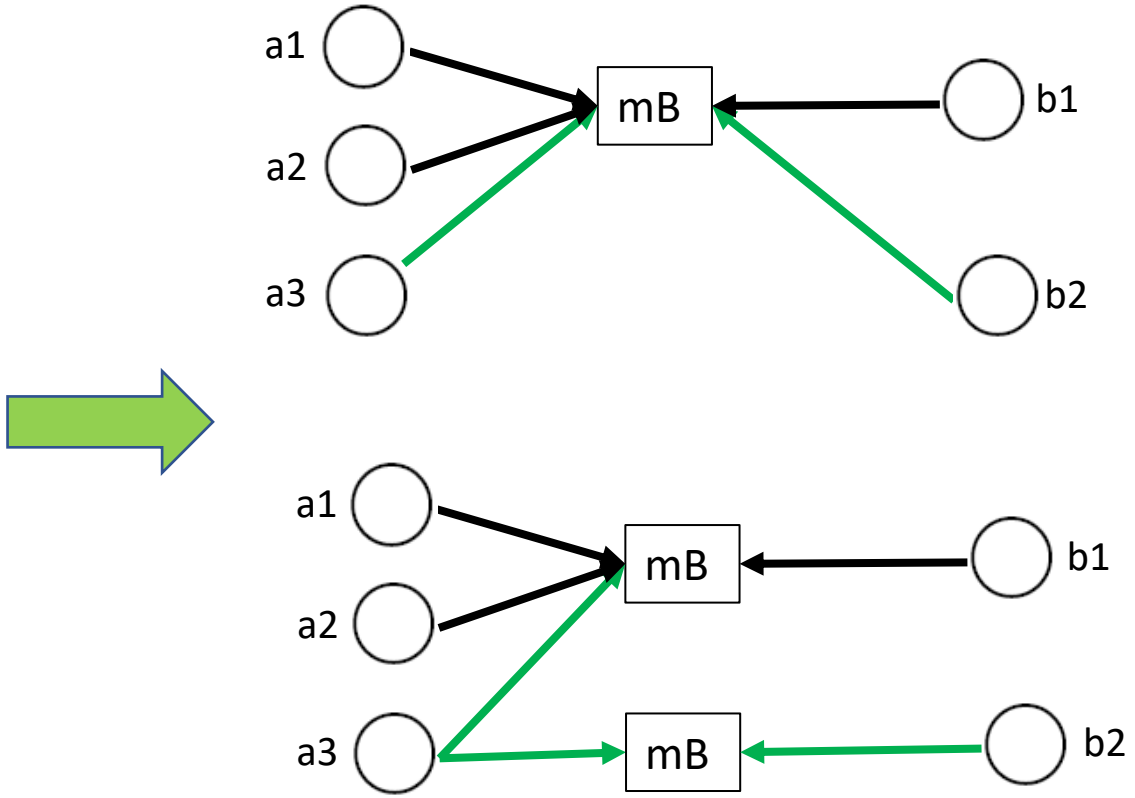
concrete implementation of refactoring left to application manager
(much like in design patterns)



modify a3

Q: what if I can apply different refactorings?
For instance, which of refactorings below is the «best» one?





A: It depends on the application

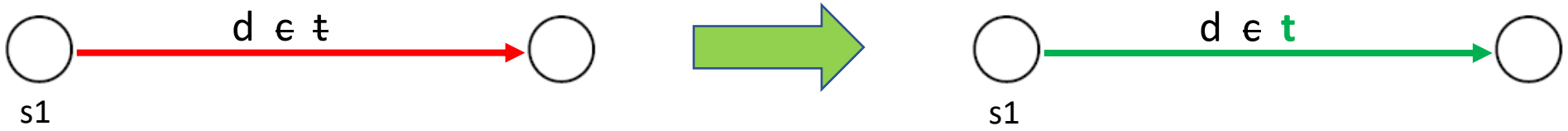
- Upper refactoring can be ok if the mB can (efficiently) support multiple «topics»
- Lower refactoring introduces a new mB

d → dynamic discovery

c → circuit breaker

t → timeout

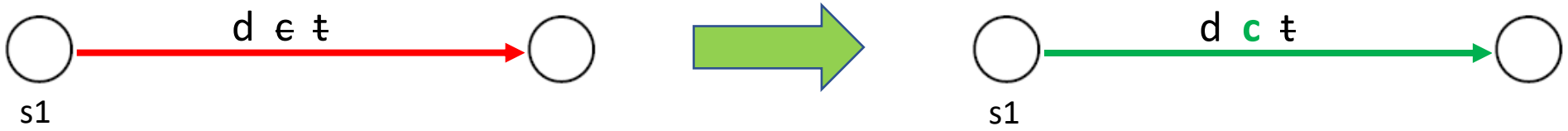
Example



add timeouts in s1

d → dynamic discovery
c → circuit breaker
t → timeout

Example (cont.)

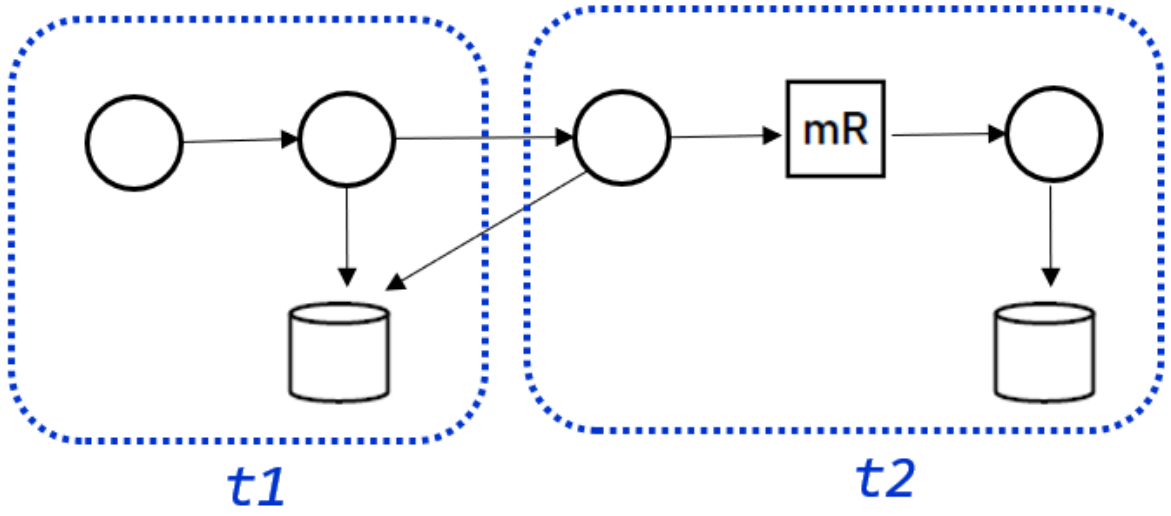


modify s1

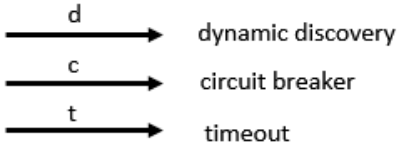
or

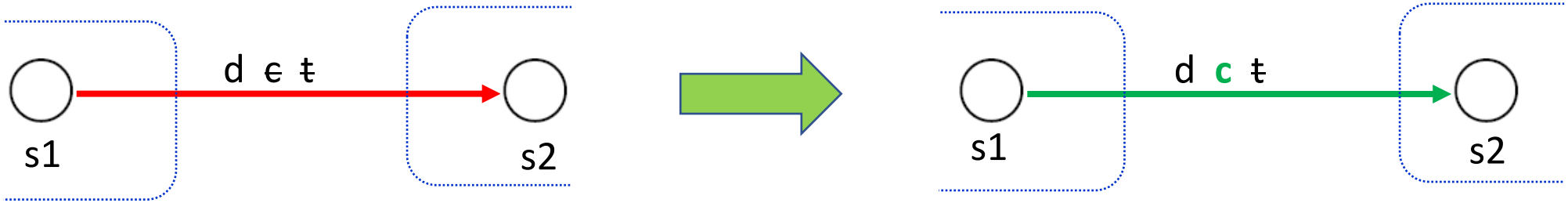
add new component

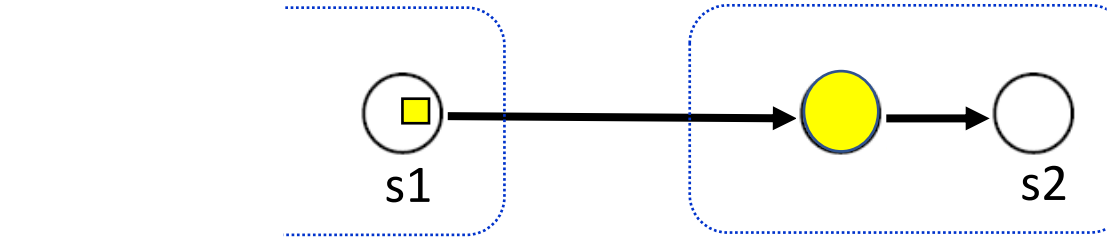
Scalability: team-based views



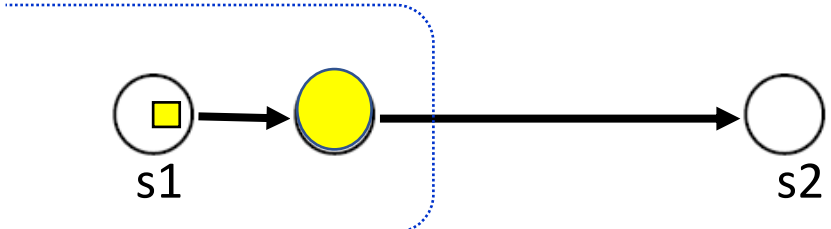
Example


d
dynamic discovery
c
circuit breaker
t
timeout

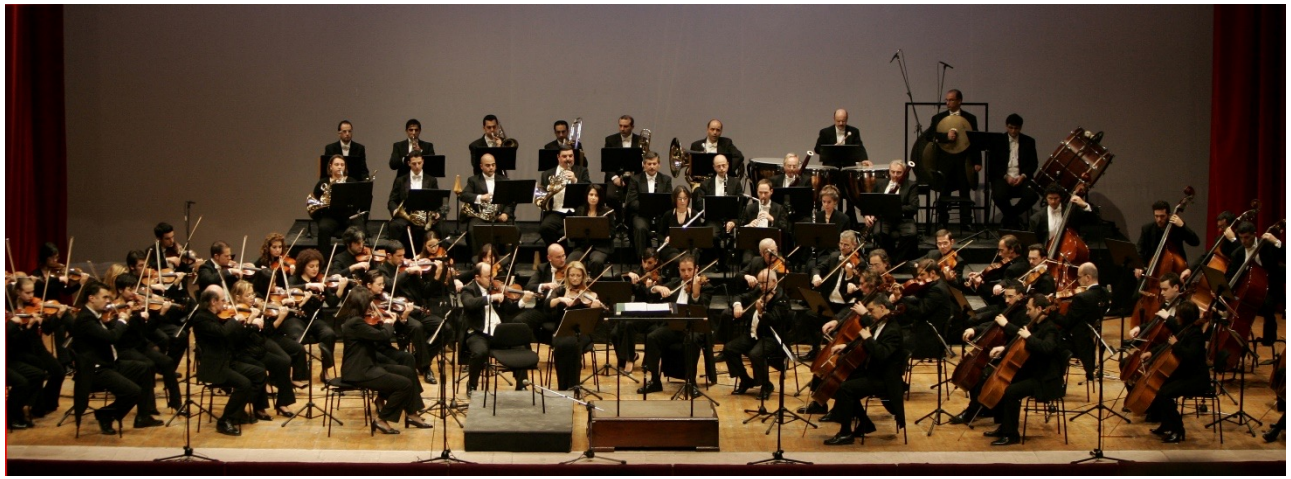

d c t
s1
s2
d c t
s1
s2

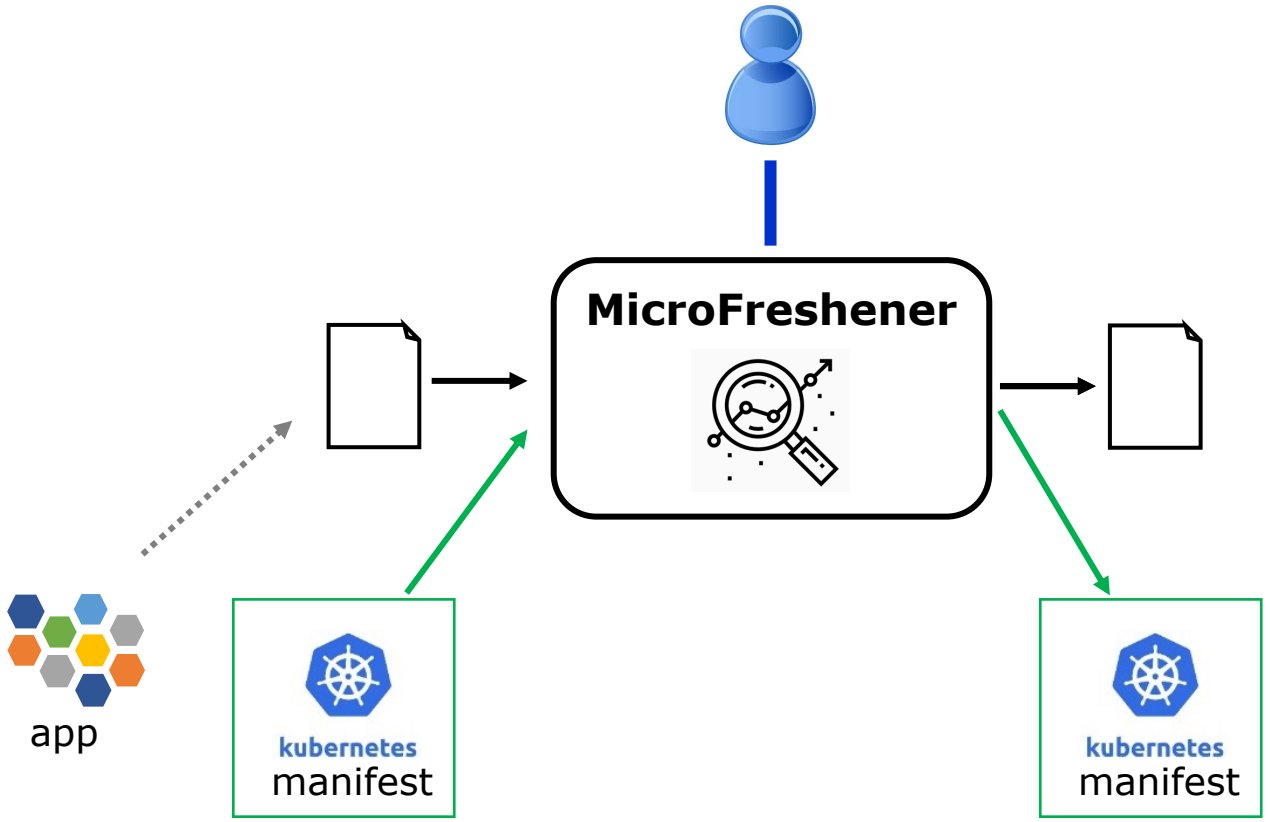

s1
s2

vs.


s1
s2

Container orchestration does change application behaviour





e.g. setting ingress node in Kubernetes adds API gateway to application

e.g. adding a sidecar container to introduce a circuit breaker

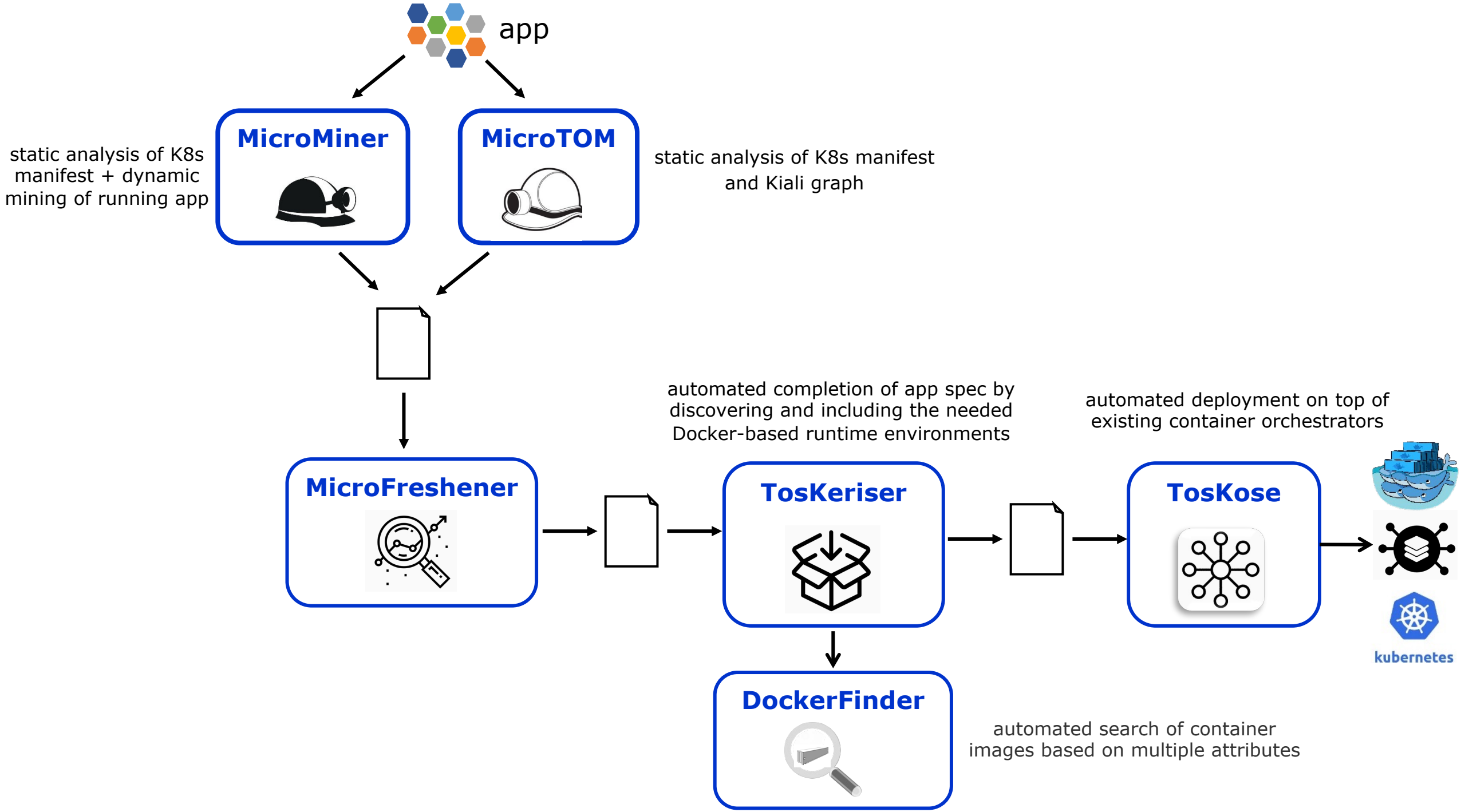
e.g. setting networking policies to introduce bulkheads

Principles, smells and refactorings

MicroFreshener

Remarks

A toolchain for microservices


app
MicroMiner
static analysis of K8s manifest + dynamic mining of running app
MicroTOM
static analysis of K8s manifest and Kiali graph
MicroFreshener
automated completion of app spec by discovering and including the needed Docker-based runtime environments
TosKeriser
automated deployment on top of existing container orchestrators
TosKose
kubernetes
DockerFinder
automated search of container images based on multiple attributes

Reference

D. Neri, J. Soldani, O. Zimmermann, A. Brogi. **Design principles, architectural smells and refactorings for microservices: A multivocal review**. SICS Software-Intensive Cyber-Physical Systems, 2020.

(Other references)

- *J. Soldani, G. Muntoni, D. Neri, A. Brogi.* ***The µTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures.*** *Software: Practice and Experience. 2021.*
- *A. Brogi, D. Neri, L. Rinaldi, J. Soldani.* ***Orchestrating incomplete TOSCA applications with Docker****. Science of Computer Programming. 2018.*
- *A. Brogi, D. Neri, J. Soldani.* ***A microservice-based architecture for (customisable) analyses of Docker images****. Software: Practice and Experience. 2018.*
- *M. Bogo, J. Soldani, D. Neri, A. Brogi.* ***Component-aware Orchestration of Cloud-based Enterprise Applications, from TOSCA to Docker and Kubernetes****. Software: Practice and Experience. 2020*

https://github.com/di-unipi-socc/