SICS Software-Intensive Cyber-Physical Systems (2020) 35:3–15
https://doi.org/10.1007/s00450-019-00407-8

SPECIAL ISSUE PAPER

Design principles, architectural smells and refactorings for microservices: a multivocal review

Davide Neri[1] · **Jacopo Soldani**[1] · **Olaf Zimmermann**[2] · **Antonio Brogi**[1]

Published online: 3 September 2019

Abstract
Potential benefits such as agile service delivery have led many companies to deliver their business capabilities through microservices. Bad smells are however always around the corner, as witnessed by the considerable body of literature discussing architectural smells that possibly violate the design principles of microservices. In this paper, we systematically review the white and grey literature on the topic, in order to identify the most recognised architectural smells for microservices and to discuss the architectural refactorings allowing to resolve them.

Keywords Microservices · SOA · Architectural principles · Architectural smells · Refactorings

1 Introduction

Microservices architectures, first discussed by Lewis and Fowler [30], bring various advantages such as ease of deployment, resilience, and scaling [34]. Many IT companies deliver their core business through microservice-based solutions nowadays, with Amazon, Facebook, Google, LinkedIn, Netflix and Spotify being prominent examples. To deliver on their promises, microservices must be designed in quality and style, which is unfortunately not always the case [47].

Microservice-based architectures can be seen as peculiar extensions of service-oriented architectures, characterized by an extended set of design principles [39,55]. These principles include shaping services around business concepts, decentralising all development aspects of microservice-based solutions (from governance to data management), adopting a culture of automation, ensuring the independent deployability and high observability of microservices, and isolating failures [34]. A key research question therefore is:

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

The currently available information on architectural smells indicating possible violations of the design principles of microservices is scattered over a considerable amount of literature. Unfortunately, this makes it difficult to consult the body of knowledge on the topic, both for researchers willing to investigate on microservices and for practitioners daily working with them.

Our objective here is to systematically analyse such literature, in order to identify the most recognised smells, as well as architectural refactorings for resolving the smells occurring in an application [54]. In particular, we focus on the design principles dealing with the dynamic aspects of the interactions between microservices at runtime, i.e., on the process viewpoint, as per the 4+1 viewpoint scheme [29]. More precisely, we consider the independent deployability of microservices, their horizontal scalability, isolation of failures and decentralisation.

As recommended by Garousi et al. [17], to capture both the state of the art and the state of practice in the field, we conducted a multivocal systematic review of the existing literature, including both white literature (i.e., peer-reviewed papers) and grey literature (i.e., blog posts, industrial whitepapers and books). We selected 41

✉ Davide Neri
davide.neri@di.unipi.it ; davide.neri@unipi.it

Jacopo Soldani
jacopo.soldani@unipi.it

Olaf Zimmermann
ozimmerm@hsr.ch

Antonio Brogi
antonio.brogi@unipi.it

[1] University of Pisa, Pisa, Italy

[2] University of Applied Sciences of Eastern Switzerland (HSR FHO), Rapperswil, Switzerland

studies, published since 2014 (when the microservice-based architectural style was first discussed [30]) until the end of January 2019. Then, following the guidelines for systematic reviews [17,40], we excerpted a taxonomy of design principles, architectural smells and corresponding refactorings. We then exploited this taxonomy to classify the selected studies, in order to distill the actual recognition of the identified smells and the usage of the corresponding refactorings.

In this paper, we illustrate the results of our study. More precisely, we first present the obtained taxonomy, including seven architectural smells and 16 refactorings, organised by design principles. We then discuss each smell, by illustrating why it can violate the design principle it is associated with, and by showing how to resolve it by means of an architectural refactoring.

We believe that the results presented in this study can provide benefits to both researchers and practitioners interested in microservices. A systematic presentation of the state of the art and practice on architectural smells and refactorings for microservices provides a body of knowledge to develop new theories and solutions, to analyse and experiment research implications, and to establish future research directions. At the same time, it can help practitioners to better understand the currently most recognised architectural smells for microservices, and to choose among the architectural refactorings allowing to resolve such smells. This can have a pragmatic value for practitioners, who can use our study as a starting point for microservices experimentation or as a guideline for day-by-day work with microservices.

The rest of the paper is organised as follows. Section 2 defines the research problem and illustrates the research methodology. Section 3 presents a taxonomy for design principles, architectural smells and refactorings, which is retaken in Sect. 4 to overview the current state of the art and practice on such smells and refactorings. Sections 5 and 6 discuss potential threats to the validity of our study and related work, respectively. Finally, Sect. 7 draws some concluding remarks.

2 Setting the stage

The objective of this survey is to identify architectural smells indicating possible violations of microservices principles, as well as the currently available solutions for refactoring microservice-based architectures in order to resolve these smells.

2.1 Scope of the survey

This survey focuses on the architectural principles of microservices that pertain to the process viewpoint, i.e., dealing with the dynamic aspects of microservices interacting at runtime [29]. We started from the principles proposed by Newman [34] and Lewis and Fowler [30], also considering the mapping to tenets proposed by Zimmermann [55]. From these works, we selected four principles:

1. The microservices forming an application should be *independently deployable*.
2. The microservices should be *horizontally scalable*.
3. *Failures* should be *isolated*.
4. *Decentralisation* should occur in all aspects of microservice-based applications, from data management to governance.

The above selection was based on three criteria:

a. *Roots* in highly significant design time and runtime quality attributes and style-defining elements,
b. *Consequences* of not adhering to a principle in terms of technical risk and re-engineering cost, and
c. *Generality*, i.e., if these four principles are met, others follow or can be achieved with similar means.

For instance, independent deployability is a defining tenet in most definitions of microservices and enables decentralized continuous delivery, thereby meeting criteria (a) and (c). Scalability is a quality attribute (a) and horizontal scalability is hard to retrofit (an aspect of (b)). Failure isolation meets criteria (a) and (b). Finally, decentralization is mentioned as crucial (and novel) in many introductions to microservices and enables independent, autonomous decision making, as required to achieve (a) and (c).

2.2 Search for studies

With the objective of capturing the state of the art and practice in the field, we searched for both white literature (i.e., peer-reviewed journal and conference articles) and grey literature (i.e., blog posts, industrial whitepapers and books), in line with what recommended by Garousi et al. [17].

The structuring of the search string was done by following the guidelines provided by Petersen et al. [40]. We indeed identified the search string guided by the PICO terms of our resarch problem, and the keywords were taken from each aspect of our research problem. Differently from Petersen et al. [40], we did not restrict our focus to specific research settings. By restricting ourselves to certain types of research settings, we could have obtained a biased or incomplete analysis, as some architectural smells or refactorings might have been over-/under-represented for a certain type of study.

As a result, our search string was formed by the following terms:

$$\texttt{microservice*} \wedge (\texttt{smell*} \vee \texttt{antipattern*} \vee \texttt{badpractice*} \vee \texttt{pitfall*} \vee \texttt{refactor*} \vee \texttt{reengineer*})$$

(where '*' matches lexically related terms). The search was restricted to studies published since the beginning of 2014 (when microservices were first proposed by Lewis and Fowler [30]) until the end of January 2019 (when the present study was initiated).

The search of white literature was carried out in the following indexing databases: ACM Digital Library, DBLP, EI Compendex, IEEE Xplore, INSPEC, ISI Web of Science, Science Direct, SpringerLink. Given the recency of the field and concerns with indexing, Google Scholar played a key role for the initial selection before the inclusion and exclusion stage. The search for industrial studies was instead carried out in renowned blogs in the software engineering community (such as DZone, InfoQ and TechBeacon), in the blog of ThoughtWorks, and in books published by practitioners.

2.3 Sample selection

The above described search criteria were matched by more than 150 studies, which we carefully screened to keep only those studies that were satisfying both the following inclusion criteria:

- A study is to be selected if it presents *at least one architectural smell* pertaining to one of the considered architectural principles of microservices (i.e., independent deployability, horizontal scalability, isolation of failure, or decentralisation).
- A study is to be selected if it presents *at least one refactoring* for resolving one of the architectural smells it discusses.

The inclusion criteria were defined with the ultimate goal of selecting only representative studies, discussing both the architectural smells (pertaining to the process viewpoint) and their corresponding refactorings.

As a result, 41 studies were selected to be analysed further. The list of references to the selected studies is in Table 1, which also classifies them by colour.

3 A taxonomy for design principles, architectural smells and refactorings

Figure 1 illustrates a taxonomy for the architectural smells pertaining to the considered design principles, and for the refactorings[1] allowing to resolve such smells. We obtained our taxonomy by following the guidelines for conducting systematic reviews in software engineering proposed by Petersen et al. [40]:

1. We established the *design principles*, by aligning them with those pertaining to the process viewpoint (as per [55]).
2. We identified the *architectural smells* by performing a first scan of the selected studies.
3. We excerpted the concrete *refactorings* directly from the selected studies after additional scans.

The identified design principles, architectural smells and refactorings were manually organised to obtain a taxonomy. The taxonomy underwent various iterations among the authors of this study, and it was submitted for validation to an external expert. This resulted in some corrections and amendments to the first version of the taxonomy, which resulted in the taxonomy displayed in Fig. 1.

4 Architectural smells and refactorings

Table 1 shows the classification of all selected studies based on the taxonomy introduced in Sect. 3. The table provides a first overview of the coverage of design principles, architectural smells and refactorings over the selected studies, despite (for reasons of readability and space) it only displays the classifications over the smells listed in the taxonomy.[2] Such coverage is also displayed in Fig. 2, from which we can observe that all architectural smells in the taxonomy are significantly recognised by the authors of the selected studies, hence making it worthy to discuss them in detail.

We hereafter illustrate how (according to the authors of the selected studies) each design principle can be affected by each corresponding architectural smell, as well as how each smell can be resolved by applying a corresponding refactoring. When multiple refactorings are applicable to resolve an architectural smell, to provide a first measurement of

[1] For the sake of clarity, in the taxonomy we follow the naming of integration patterns proposed by Hohpe and Woolf [22].

[2] The detailed classification, displaying each occurrence of each refactoring, is publicly available at https://github.com/di-unipi-socc/microservices-smells-and-refactorings.

Table 1 References to the selected studies, and their classification by *colour* (i.e., *w*hite or *g*rey literature) and according to the taxonomy in Fig. 1

	Colour	Independent deployab.	Horizontal scalability		Isolation of failures	Decentralisation		
		Multiple ser. in one cont.	*No API gateway*	*Endpoint-bas. ser. inter.*	*Wobbly ser. inter.*	*ESB misuse*	*Shared persistence*	*Single-layer teams*
[1]	*g*		✓					
[2]	*w*			✓	✓			
[3]	*w*	✓		✓	✓			✓
[4]	*w*	✓	✓	✓	✓			
[5]	*g*		✓	✓	✓		✓	
[7]	*g*		✓	✓	✓	✓		
[8]	*g*						✓	
[9]	*g*	✓	✓		✓		✓	
[10]	*w*	✓						✓
[11]	*g*				✓			
[13]	*w*		✓	✓	✓			
[14]	*w*		✓	✓	✓			
[15]	*w*	✓						
[16]	*w*						✓	
[18]	*g*							✓
[20]	*g*				✓			
[21]	*g*						✓	✓
[23]	*g*	✓		✓	✓	✓	✓	
[24]	*g*	✓	✓	✓	✓	✓	✓	
[25]	*w*	✓			✓			
[26]	*w*				✓		✓	✓
[27]	*w*				✓		✓	
[28]	*g*	✓	✓	✓	✓			
[30]	*g*			✓	✓	✓		✓
[32]	*g*	✓						
[31]	*g*			✓	✓			
[33]	*g*		✓	✓	✓		✓	✓
[34]	*g*	✓		✓	✓			
[35]	*g*	✓	✓	✓	✓			
[41]	*g*				✓		✓	
[42]	*g*		✓	✓			✓	
[43]	*g*		✓		✓		✓	
[44]	*g*				✓			
[45]	*g*			✓	✓		✓	

Table 1 continued

	Colour	Independent deployab.	Horizontal scalability		Isolation of failures	Decentralisation		
		Multiple ser. in one cont.	*No API gateway*	*Endpoint-bas. ser. inter.*	*Wobbly ser. inter.*	*ESB misuse*	*Shared persistence*	*Single-layer teams*
[46]	*w*	✓						
[47]	*w*	✓	✓		✓		✓	
[49]	*w*		✓	✓		✓	✓	
[50]	*w*						✓	✓
[51]	*w*	✓					✓	
[53]	*g*			✓	✓		✓	✓
[55]	*w*	✓				✓		

how much a refactoring is used to resolve it, we display the weight[3] of each refactoring by exploiting %-based pie charts.

4.1 Independent deployability

In microservice-based applications, each microservice should be operationally independent from the others, meaning that it should be possible to deploy and undeploy a microservice indepedently from the others [34]. This indeed impacts on the initial deployment of a microservice, which can get started without waiting for other microservices to be running, as well as on the possibility of adding/removing replicas of a microservice at runtime.

We discuss below the MULTIPLE SERVICES IN ONE CONTAINER smell, showing how it violates the above principle and how it can be resolved.

Multiple services in one container Containers (such as Docker containers) provide an ideal way to deploy microservices addressing the above requirement, if properly used. Each microservice can indeed be packaged in a container image, and different instances of a same microservice can be launched by spawning different containers from the corresponding image. With this view, the orchestration of the deployment and management of a microservice-based application can be performed by exploiting the currently available support for orchestrating Docker containers [23].

The above is the right way of using containers, at least according to the authors of 16 of the selected studies. They indeed highlight how placing multiple services in one container would constitute an architectural smell for the independent deployability of microservices. If two microservices would be packaged in the same Docker image, spawning a container from such image would result in launching both microservices. Similarly, stopping the container would result in stopping both microservices. In other words, by placing two microservices in the same container, these services would operationally depend one another, as it would not be possible to launch a new instance of one of such microservices, without also launching an instance of the other.

If the MULTIPLE SERVICES IN ONE CONTAINER smell occurs, the solution is to refactor the application in such a way that each microservice is packaged in a separate container image.

4.2 Horizontal scalability

The possibility of adding/removing replicas of a microservice is a direct consequence of the independent deployability

[3] We measure the weight of a refactoring as the percentage of its occurrences among all occurrences of all refactorings for the same smell. This is analogous to what done by Pahl et al. [37] to measure weights while classifying studies on cloud container technologies.

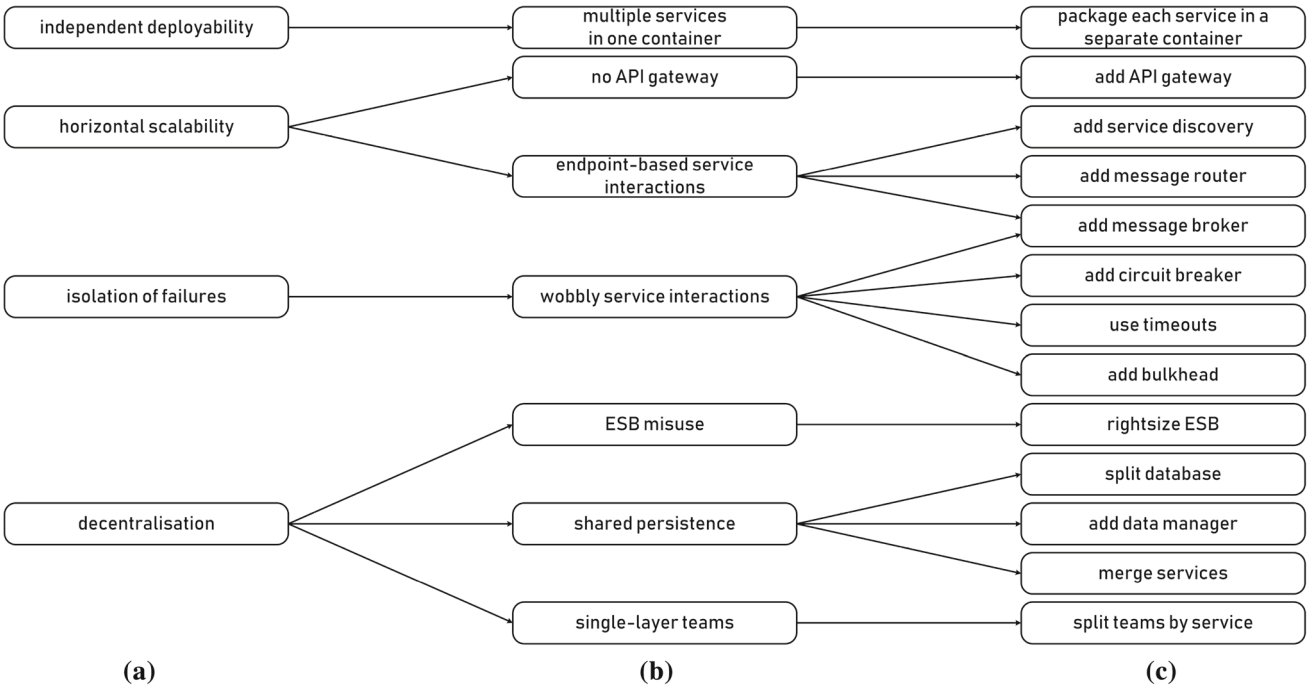


Fig. 1 A taxonomy for **a** the design principles pertaining to the process viewpoint, **b** the architectural smells possibly violating such principles, and **c** the refactorings resolving such smells

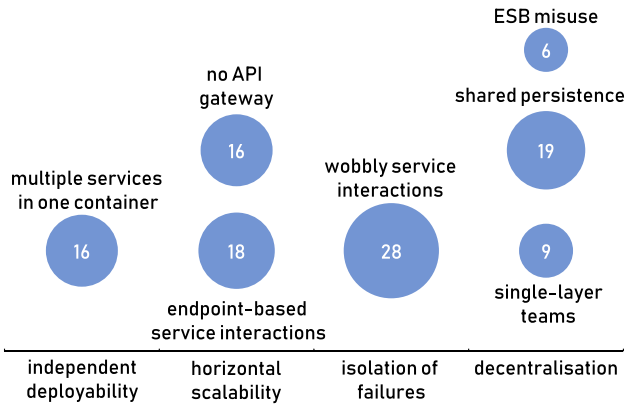


Fig. 2 Coverage of the architectural smells in the selected studies. The size of each bubble is directly proportional to the number of selected studies discussing the corresponding smell. This number is also reported within each bubble

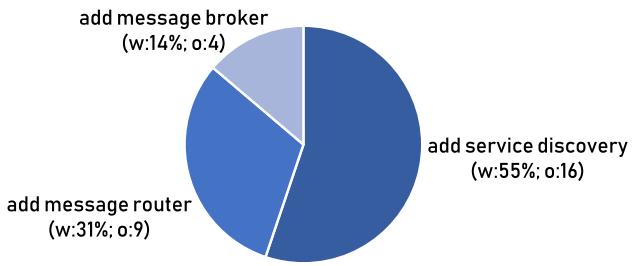


Fig. 3 Weights (w) and occurrences (o) of the refactorings for the ENDPOINT- BASED SERVICE INTERACTIONS smell

of microservices. To ensure its horizontal scalability, all the replicas of a microservice m should be reachable by the microservices invoking m [23].

In the selected studies, two architectural smells emerged as possibly violating the horizontal scalability of microservices, i.e., ENDPOINT- BASED SERVICE INTERACTIONS and NO API GATEWAY, which we discuss hereafter.

Endpoint-based service interactions This smell occurs in an application when one or more of its microservices invoke a specific instance of another microservice (e.g., because its location is hardcoded in the source code of the microservices invoking it, or because no load balancer is used). If this is the case, when scaling out the latter microservice by adding new replicas, these cannot be reached by the invokers, hence only resulting in a waste of resources.

From the selected studies, it became evident that the ENDPOINT- BASED SERVICE INTERACTIONS smell can be resolved by applying three different refactorings (Fig. 3). The most common solution is to introduce a service discovery mechanism. Such mechanism can be implemented as a service storing the actual locations of all instances of the microservices in an application [43]. Microservice instances send their locations to the service registry at startup, and they are unregistered at shutdown. When willing to interact with a microservice, a client can then query the service discovery to retrieve the location of one of its instances.

The other two possible solutions share the same goal, i.e., decoupling the interaction between two microservices by introducing an intermediate integration pattern. Nine of the selected studies indeed suggest to introduce a message router (e.g., a load balancer), so that the requests to a microservices are routed towards all its actual instances. Four of the selected studies instead suggest to exploit message brokers (e.g., message queues) to decouple the interactions between two or more microservices.

No API gateway When a microservice-based application lacks an API gateway, the clients of the application necessarily have to invoke its microservices directly. The result is a situation similar to that of the ENDPOINT- BASED SERVICE INTERACTIONS smell, with the invoker being a client of the application. The client indeed interacts only with the specific instances of the microservices it needs. If one of such microservices is scaled out and the client still keeps invoking the same instance of the microservice, then we have a waste of resources.

The authors of all the selected studies discussing the NO API GATEWAY smell agree that the solution to this smell is to add one API gateway to the application. The latter act as single entry points for all clients, and they handle requests either by routing them or by fanning them out to the instances of the microservices that must handle them [43].

It is worth noting that, even if the NO API GATEWAY smell results in a similar situation to that of the ENDPOINT-BASED SERVICE INTERACTIONS smell, the refactorings to resolve them are different. The reason for this resides in the main difference between the two architectural smells. The NO API GATEWAY smell occurs at the edge of the architecture of a microservice-based application, with the clients of the application directly invoking its microservices, while the ENDPOINT- BASED SERVICE INTERACTIONS smell occurs in between its microservices [33]. Given this, the introduction of an API gateway can be useful not only for facilitating the horizontal scalability of the microservices forming an application, but also for various other reasons. For instance, rather than implementing end-user authentication or throttling in each microservice, these can be implemented once for the whole application in the API gateway [1].

4.3 Isolation of failures

Microservices can fail for many reasons (e.g., network or hardware issues, application-level issues, bugs), hence becoming unavailable to serve other microservices. Additionally, communication fails from time to time in any kind of distributed system, and this is even more likely to occur in microservice-based systems, simply because of the amount of messages exchanged among microservices [25]. Microservice-based applications should hence be designed so that each microservice can tolerate the failure of any invocation to the microservices it depends on [30]. If this is ensured, then a microservice-based application results to be much more resilient than a monolithic application, simply because failures affects only few microservices in an application, instead of the whole monolith [34].

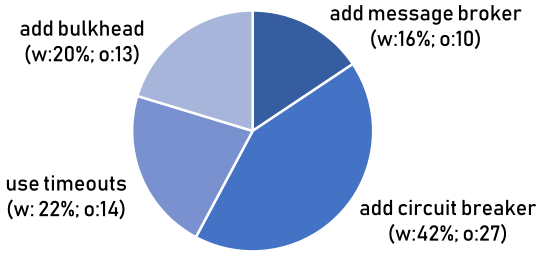


Fig. 4 Weights (w) and occurrences (o) of the refactorings for the WOBBLY SERVICE INTERACTIONS smell

The authors of the selected studies identify and discuss an architectural smell that can possibly violate the isolation of failures in microservice-based solutions. This is the WOBBLY SERVICE INTERACTIONS smell, which we discuss hereafter.

Wobbly service interactions The interaction of a microservice m_i with another microservice m_f is "wobbly" when a failure in m_f can result in triggering a failure also in m_i. This typically happens when m_i is directly consuming one or more functionalities offered by m_f, and m_i is not provided with any solution for handling the possibility of m_f to fail and be unresponsive. If this is the case, m_i will also fail in cascade, and (in a worst case scenario) the failure of m_i can result in triggering the failure of other microservices, which in turn trigger other cascading failures, and so on [25].

To avoid WOBBLY SERVICE INTERACTIONS (such as the one between m_i and m_f described above), the authors of the selected studies identify four possible solutions (Fig. 4). The most common solution is the usage of a circuit breaker to wrap the invocations from a microservice to another. In the normal "closed" state, the circuit breaker forwards the invocations to the wrapped microservice, and it monitors their execution to detect and count failing invocations. Once the frequency of failures reaches a certain (customisable) threshold, the circuit breaker trips and "opens" the circuit. All further calls to the wrapped microservice will "safely fail", as the circuit breaker will immediately return an error message to the calling microservices. The latter can then exploit the error messages returned by the circuit breaker to avoid failing themselves [30].

Following the same baseline idea of circuit breakers, ten of the selected studies propose to decouple the interaction between invoking and invoked microservices by exploiting a message broker (e.g., a message queue). The usage of a broker allows the invoker to send its requests to the broker, and allows the invoked microservice to process such requests when it is available. In this way, there is no direct interaction

between the two microservices, and the invoker does not fail when the invoked microservice fails (as the former continues to send messages to the broker). On the other hand, the usage of message brokers is more costly compared to circuit breakers. The reason is that message brokers require to intervene on the interaction protocol between two microservices, which should start putting and getting messages to/from the broker. Instead, with circuit breakers the interaction protocol between two microservices is unaltered, as a circuit breaker simply wraps the invocation of a microservice. This is the reason why message brokers are much less discussed than circuit breakers.

The most discussed alternative to circuit breakers are however timeouts, which are a simple yet effective mechanism allowing a microservice to stop waiting for an answer from another microservice, when the latter is unresponsive (e.g., since it failed or due to network issues). Well-placed timeouts provide fault isolation, as the fact that a microservice is unresponsive does not create any other issue in the microservices invoking it [34]. However, such a kind of solution might not likely to be applicable nowadays, as some of the APIs used to remotely invoke microservices have few or no explicit timeout settings [34]. Note that the timeout can be also set in the invoker (e.g., by setting the timeout on an HTTP request), hence it is not always requested to have a timeout setting on the invoked service.

Finally, another alternative is the usage of bulkheads, whose ultimate goal is to enforce the principle of damage containments (like bulkheads in ships, which prevent water to flow across sections). The idea is that, if cascading failures cannot be avoided, they should at least be limited by exploiting bulkheads. More precisely, the microservices forming an application should be logically and/or physically partitioned so as to ensure that the failure of a microservice can be propagated at most to the other microservices in the same partition, by preventing the rest of the system from being affected by such failure [35].

4.4 Decentralisation

Decentralisation should occur in all aspects of microservice-based applications [34]. This also means the business logic of an application should be fully decentralised and distributed among its microservices, each of which should own its own domain logic [55].

The authors of the selected studies indentify and discuss three architectural smells possibly violating the above principle, i.e., the ESB MISUSE, SHARED PERSISTENCE and SINGLE- LAYER TEAMS smells. We hereafter discuss them, by also illustrating the refactorings currently employed to resolve them.

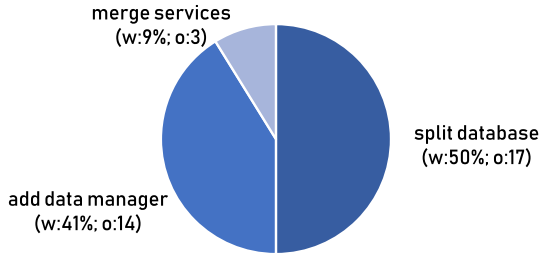


Fig. 5 Weights (w) and occurrences (o) of the refactorings for the SHARED PERSISTENCE smell

ESB misuse The misuse of Enterprise Service Buses (ESB) products is considered to be an architectural smell by the microservice community. When positioned as a single central hub (with the services as spokes), an ESB may become a bottleneck both architecturally and organizationally [39]. "Smart endpoints & dumb pipes" has been a recommended practice since the very beginnings of service-oriented architectures [55] that regrettably has not always been followed in all SOA implementations. Such ESB abuse may lead to undesired centralisation of business logic and dumb services [34]. The microservices community therefore (re-)emphasizes the decoupling of microservices and their cohesiveness [30].

Whenever a central ESB is used for connecting microservices in an application, the topology should be refactored to remove the dependency on a single middleware component instance. Multiple instances should instead be used, and they should implement queue-based asynchronous messaging. The latter only permits adding and removing messages, hence forming a "dumb pipe". The "smart" part should be left to the microservices, which implement the logic for deciding when/how to process the messages in the message broker [49]. Additional infrastructure logic, for instance traffic management capabilities, may be placed in side cars accompanying each service. This repositioning and rectification of ESB middleware improves the decoupling characteristics of the services architecture and reestablishes the original "smart endpoints & dumb pipes" recommendations from the first wave of service-orientation.

Shared persistence The SHARED PERSISTENCE smell occurs whenever two microservices access and manage the same database, possibly violating the decentralisation design principle [47].

The three currently available solutions for refactoring microservices and resolving the SHARED PERSISTENCE smell are shown in Fig. 5.

Although the ultimate goal of these three solutions is the same (i.e., having each database accessed by only one microservice), they are very diverse in spirit. They apply to different situations, highly depending on the microservices accessing the same database.

The most discussed solution is to actually split a database shared by multiple microservices, in such a way that each

microservice accesses and manages only the data it needs. This solution is the one requiring less intervention on the microservices, as they would continue to use the same protocol to interact with the databases. At the same time, splitting a database into a set of independent databases is not always possible or easy to achieve. Also, if some data is to be replicated among the databases obtained from the split, then mechanisms for (eventual) data consistency should be introduced after the refactoring [47]. Given the above, the split of database is recommended when the microservices accessing the same database implement separate business logics working on disjoint portions of such database [24].

The most discussed alternative is to introduce an additional microservice, acting as "data manager". The data manager becomes the only microservice interacting with and managing the database, and the microservices that were accessing the database now have to interact with the data manager to ask for accessing and updating the data. While this solution introduces some additional communication overhead, it is considered as always applicable, and the data manager can also be enriched with additional logic for processing the data it manages [24].

Finally, it is worth commenting on the refactoring discussed in three of the selected studies, i.e., merging the microservices accessing the same database. The idea is that, when multiple microservices access the same database, this may be a signal of the fact that the application has been split too much, by obtaining too fine-grained microservices processing the same data. If this is the case, then the possibility of merging such microservices is a concrete option to be evaluated [49].

Single-layer teams To maximize the autonomy that microservices make possible, the governance of microservices should be decentralised and delegated to the teams that own the microservices themselves. As pointed out by Zimmermann [55], even if this is not a technical concern, it is related to the process viewpoint due to its cross-cutting nature. The microservice community indeed strongly emphasizes the connection between architecture and organisation, especially concerning the integration of the microservices in an application [18,21,30].

The classical approach of splitting teams by technology layers (e.g., user interface teams, and middleware teams, and database teams) is hence considered an architectural smell, as any change to a microservice may result in a cross-team project having take time and budgetary approval [30]. This may be the case for the refactorings discussed so far.

The microservice approach to team splitting is orthogonal to the above, as each microservice should be assigned to a full-stack team whose members span across all technology layers. In this way, the interactions for updating a microservice (e.g., to apply one of the refactorings discussed in this section) are limited to the team managing such microservice, which can independently decide how to proceed and implement the updates [10].

In short, if the governance of a microservice-based is organised by SINGLE- LAYER TEAMS, this is an architectural smell. The solution is to split teams by microservice, rather than by technology layer [30].

5 Threats to validity

Following the taxonomy developed by Wohlin et al. [52], four potential theats may affect the validity of our study. These are the threats to *external* validity, the threats to *internal* and *construct* validity, and the threats to *conclusions* validity, which we discuss hereafter.

External validity As per Wohlin et al. [52], the external validity concerns the applicability of a set of results in a more general context. Since we selected the primary studies from a very large extent of online sources, the identified architectural smells and refactorings may only be partly applicable to the broad area of disciplines and practices on microservices, hence threatening external validity.

To reinforce the external validity of our findings, we organised two feedback sessions during our analysis of the existing literature. We analysed the discussion following-up from the feedback session, and we exploited this qualitative data to fine-tune both our research methods and the applicability of our findings. We also prepared a GitHub repository,[4] where we placed the artifacts produced during our analysis, so as to make it available to all who wish to deepen their understanding on the data we produced. We believe that this can help in making our results and observations more explicit and applicable in practice.

Additionally, one may argue that our selection criteria are too restrictive. The rationale behind such criteria is that we aim focusing only on representative studies, by requiring selected studies to discuss at least an architectural smell *and* a refactoring for resolving it. There is however a risk of having missed some relevant literature, as a study might not explicitly mention the architectural smells and refactorings in our taxonomy (Fig. 1). To mitigate this threat, we carefully checked both selection criteria against each candidate study, by verifying whether a study was discussing the problems characterised by an architectural smell, and whether it was discussing the architectural changes characterising a refactoring. Even if a study was not explicitly referring to a smell/refactoring, but it was reporting on the corresponding problems/changes, the study was included in the selected literature.

[4] http://github.com/di-unipi-socc/microservices-smells-and-refactorings.

Finally, there is a risk of having missed relevant grey literature, since industrial studies may exploit a different terminology than ours (e.g., a blog post discussing some architectural smells and refactorings may not employ the term "smell" or "refactor"). To mitigate this threat to validity, we included relevant synonyms in the search string, and we exploited the features offered by search engines, which naturally support including related terms in string-based searches.

Construct and internal validity The internal validity concerns the validity of the method employed to study and analyse data (e.g., the potential types of bias involved), while the construct validity concerns the generalisability of the constructs under study [52].

To mitigate the corresponding potential threats, the obtained taxonomy underwent various iterations among the authors of this study to avoid bias by triangulation, and it was submitted for validation to an external expert. The same process was applied to the classification of the selected studies, and to the results of the analysis.

Conclusions validity The conclusions validity concerns the degree to which the conclusions of a study are reasonably based on the available data [52].

In this perspective, and with the aim of performing a sound analysis of the data we retrieved, we exploited inter-rater reliability assessment to limit potential biases in our observations and interpretations. Additionally, the observations and conclusions discussed in this paper were independently drawn, and they were then double-checked against the selected studies and related studies in a joint discussion session.

6 Related work

There exist various studies on microservices, aimed at analysing and classifying the state of the art and practice on microservices. Pahl and Jamshidi [38] and Taibi et al. [51] present two first systematic mapping studies on microservices. Pahl and Jamshidi [38] discuss agreed and emerging concerns on microservices, position microservices with respect to current cloud and container technologies, and elicit potential research directions. Taibi et al. [51] instead report on architectural patterns common to microservice-based solutions, by discussing the advantages, disadvantages and lessons learned of each pattern. However, neither Pahl and Jamshidi [38] nor Taibi et al. [51] provide an overview both on the architectural smells applicable to microservices and on the refactorings for resolving such smells.

Two other examples are the industrial surveys by Di Francesco et al. [12] and by Ghofrani and Lübke [19], which both discuss the current state of practice on microservices in the IT industry. Both report on empirical studies conducted in the form of surveys for practictioners working everyday with microservices, to elicit the challenges and advantages on employing microservices. This differs from our study, as we aim at distilling the architectural smells that can affect the architecture of a microservice-based solution, as well as the refactorings allowing to resolve such smells.

Similar considerations apply to the systematic review by Soldani et al. [47], who provide an overview on the state of practice on microservices. Soldani et al. systematically analyse the grey literature on microservices, in order to identify the technical/operational advantages and disadvantages of the microservice-based architectural style. The objective of Soldani et al. hence differs from ours, as we aim at discussing concrete architectural smells and refactorings for the microservice-based architectural style.

In this perspective, the objective of the studies by Taibi and Lenarduzzi [49], by Bogner et al. [6], and by Carrasco et al. [10] is much closer to ours. Taibi and Lenarduzzi [49] report on a survey submitted to practictioners experienced with microservices. The survey allowed Taibi and Lenarduzzi to identify 11 microservice-specific architectural smells, each with a refactoring solution allowing to resolve it. Of such smells and refactorings, only four can be related to the design principles of microservices pertaining to the process viewpoint (see Table 1). By integrating the work by Taibi and Lenarduzzi with other carefully selected white/grey literature, we managed to extend the set of architectural smells and refactorings pertaining to the process viewpoint with three additional smells and ten additional refactorings.

Bogner et al. [6] present a systematic literature review identifying and documenting architectural smells in SOA-based architectural styles, including microservices. Altough the main focus of their review is on the broader SOA, several smells apply also to microservices. However, the review by Bogner et al. [6] differs from ours, as it focuses only on white literature, and since it does not discuss the architectural refactorings allowing to resolve the identified smells.

Carrasco et al. [10] systematically analyses the white and grey literature on architectural smells that can occur while migrating from monoliths to microservice-based solutions. They present nine common smells with their potential solutions, which all pertain to the actual development and operation of microservice-based applications (i.e., development and physical viewpoints). The study by Carrasco et al. [10] hence differs from ours, as we focus on the dynamic aspects of microservices that interact at runtime (i.e., process viewpoint).

Similar considerations apply to the study by Furda et al. [16], which focuses on multitenancy, statefulness, and

data consistency. Their objective is indeed supporting the migration of enterprise legacy source code to microservices. Finally, the Microservices API Pattern (MAP) language suggests design improvements in the form of an informal cheat sheet. The first MAP patterns have been published by Stocker et al. [48] and by Zimmermann et al. [56].

In summary, to the best of our knowledge, there is currently no study classifying the architectural smells possibly violating the design principles of microservices pertaining to the process viewpoint, together with the refactorings that permit resolving such smells. The latter is precisely the scope of our study, which we have presented in this paper.

7 Conclusions

We presented the results of a multivocal review focused on identifying architectural smells indicating possible violations of the independent deployability, horizontal scalability, fault isolation and decentralisation of microservices, as well as the refactorings allowing to resolve such smells. More precisely, we presented a taxonomy organising seven architectural smells and 16 refactorings, by associating each smell with the design principle(s) it violates, and each refactoring with the smell it resolves. We then provided an overview of the actual recognition of such smells and refactorings in the selected literature. We also discussed why each architectural smell violates the design principle it pertains to, and how each architectural refactoring allows resolving its corresponding smell.

We believe that our study can be of help to both researchers and practitioners interested in microservices. Together with the review by Carrasco et al. [10], our results can help them to understand the well-known architectural smells for microservices, and to choose among the refactorings allowing to resolve such smells. This can have a pragmatic value for practitioners, who can exploit the results of our study in their daily work with microservices. It can also help researchers to shape new solutions and to establish future research directions.

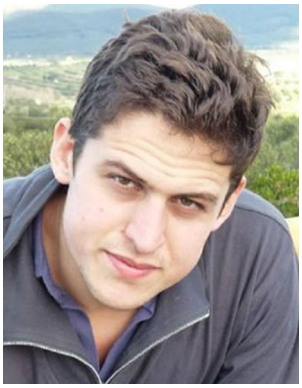
We plan to exploit our results to develop a design-time support for eliminating architectural smells from microservice-based applications. Our idea is to exploit existing languages for the specification of microservice-based applications (such as TOSCA [36], for instance). We then plan to develop a tool for processing the specification of a microservice-based application, to automatically detect the architectural smells occurring in such application, and to suggest the architectural refactorings resolving such smells.

Acknowledgements This work was partly funded by the POR-FSE project *AMaCA* (Regione Toscana), and by the project *DECLware* (PRA_2018_66, University of Pisa).

References

1. Alagarasan V (2015) Seven microservices anti-patterns. InfoQ. https://www.infoq.com/articles/seven-uservices-antipatterns. Accessed 5 June 2019
2. Alshuqayran N, Ali N, Evans R (2016) A systematic mapping study in microservice architecture. In: 2016 IEEE 9th international conference on service-oriented computing and applications (SOCA), pp 44–51. https://doi.org/10.1109/SOCA.2016.15
3. Balalaie A, Heydarnoori A, Jamshidi P (2016) Microservices architecture enables devops: migration to a cloud-native architecture. IEEE Softw 33(3):42–52. https://doi.org/10.1109/MS.2016.64
4. Balalaie A, Heydarnoori A, Jamshidi P, Tamburri DA, Lynn T (2018) Microservices migration patterns. Softw Pract Exp 48(11):2019–2042. https://doi.org/10.1002/spe.2608
5. Bhojwani R (2018) Design patterns for microservices. DZone. https://dzone.com/articles/design-patterns-for-microservices. Accessed 5 June 2019
6. Bogner J, Boceck T, Popp M, Tschechlov D, Wagner S, Zimmermann A (2019) Towards a collaborative repository for the documentation of service-based antipatterns and bad smells. In: 2019 IEEE international conference on software architecture workshops (ICSAW) (**in press**)
7. Bonér J (2016) Reactive microservice architecture: design principles for distributed systems. O'Reilly, Newton
8. Carneiro C, Schmelmer T (2016) Microservices from day one: build robust and scalable software from the start, 1st edn. Apress, Berkeley
9. Carnell J (2017) Spring microservices in action, 1st edn. Manning Publications Co., New York
10. Carrasco A, Bladel B, Demeyer S (2018) Migrating towards microservices: migration and architecture smells. In: Proceedings of the 2nd international workshop on refactoring, IWoR 2018. ACM, pp 1–6. https://doi.org/10.1145/3242163.3242164
11. Dall R (2016) Performance patterns in microservices-based integrations. DZone. https://dzone.com/articles/performance-patterns-in-microservices-based-integr-1. Accessed 5 June 2019
12. Di Francesco P, Lago P, Malavolta I (2018) Migrating towards microservice architectures: an industrial survey. In: 2018 IEEE international conference on software architecture (ICSA), pp 29–38. https://doi.org/10.1109/ICSA.2018.00012
13. Di Francesco P, Lago P, Malavolta I (2019) Architecting with microservices: a systematic mapping study. J Syst Softw 150:77–97. https://doi.org/10.1016/j.jss.2019.01.001
14. Di Francesco P, Malavolta I, Lago P (2017) Research on architecting microservices: trends, focus, and potential for industrial adoption. In: 2017 IEEE international conference on software architecture (ICSA), pp 21–30. https://doi.org/10.1109/ICSA.2017.24
15. Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, Safina L (2017) Microservices: yesterday, today, and tomorrow. Springer, Berlin, pp 195–216. https://doi.org/10.1007/978-3-319-67425-4_12
16. Furda A, Fidge C, Zimmermann O, Kelly W, Barros A (2018) Migrating enterprise legacy source code to microservices: on multitenancy, statefulness, and data consistency. IEEE Softw 35(3):63–72. https://doi.org/10.1109/MS.2017.440134612
17. Garousi V, Felderer M, Mäntylä MV (2016) The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In: Proceedings of the 20th international conference on evaluation and assessment

in software engineering (EASE'16). ACM, pp 26:1–26:6. https://doi.org/10.1145/2915970.2916008
18. Gehani N (2018) Want to develop great microservices? Reorganize your team. TechBeacon. https://techbeacon.com/app-dev-testing/want-develop-great-microservices-reorganize-your-team. Accessed 5 June 2019
19. Ghofrani J, Lübke D (2018) Challenges of microservices architecture: a survey on the state of the practice. In: Proceedings of the 10th workshop on services and their composition (ZEUS 2018). CEUR-WS.org, pp 1–8
20. Golden B (2017) 5 fundamentals to a successful microservice design. TechBeacon. https://techbeacon.com/app-dev-testing/5-fundamentals-successful-microservice-design. Accessed 5 June 2019
21. Golden B (2018) Creating a microservice: design first, code later. TechBeacon. https://techbeacon.com/app-dev-testing/creating-microservice-design-first-code-later. Accessed 5 June 2019
22. Hohpe G, Woolf B (2003) Enterprise integration patterns: designing, building, and deploying messaging solutions. Addison-Wesley, Longman, London
23. Indrasiri K (2016) Microservices in practice: from architecture to deployment. DZone. https://dzone.com/articles/microservices-in-practice-1. Accessed 5 June 2019
24. Indrasiri K, Siriwardena P (2018) Microservices for the enterprise: designing, developing, and deploying, 1st edn. Apress, Berkeley
25. Jamshidi P, Pahl C, Mendonca N, Lewis J, Tilkov S (2018) Microservices: the journey so far and challenges ahead. IEEE Softw 35(3):24–35. https://doi.org/10.1109/MS.2018.2141039
26. Kalske M, Mäkitalo N, Mikkonen T (2018) Challenges when moving from monolith to microservice architecture. In: Garrigós I, Wimmer M (eds) Current trends in web engineering. Springer, Berlin, pp 32–47
27. Knoche H, Hasselbring W (2018) Using microservices for legacy software modernization. IEEE Softw 35(3):44–49. https://doi.org/10.1109/MS.2018.2141035
28. Krause L (2015) Microservices: patterns and applications, 1st edn. Microservicesbook.io
29. Kruchten P (1995) The 4+1 view model of architecture. IEEE Softw 12(6):42–50. https://doi.org/10.1109/52.469759
30. Lewis J, Fowler M (2014) Microservices: a definition of this new architectural term. ThoughtWorks. https://www.martinfowler.com/articles/microservices.html. Accessed 5 June 2019
31. Long J (2015) The power, patterns, and pains of microservices. DZone. https://dzone.com/articles/the-power-patterns-and-pains-of-microservices. Accessed 5 June 2019
32. Meléndez C (2018) 7 container design patterns you need to know. TechBeacon. https://techbeacon.com/enterprise-it/7-container-design-patterns-you-need-know. Accessed 5 June 2019
33. Nadareishvili I, Mitra R, McLarty M, Amundsen M (2016) Microservice architecture: aligning principles, practices, and culture, 1st edn. O'Reilly, Newton
34. Newman S (2015) Building microservices, 1st edn. O'Reilly, Newton
35. Nygard M (2018) Release it!: Design and deploy production-ready software, 2nd edn. Pragmatic Bookshelf
36. OASIS: TOSCA Simple Profile in YAML (2014) http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.pdf. Accessed 5 June 2019
37. Pahl C, Brogi A, Soldani J, Jamshidi P (2017) Cloud container technologies: a state-of-the-art review. IEEE Trans Cloud Comput. https://doi.org/10.1109/TCC.2017.2702586
38. Pahl C, Jamshidi P (2016) Microservices: A systematic mapping study. In: Proceedings of the 6th international conference on cloud computing and services science, Volume 1 and 2 (CLOSER 2016). SCITEPRESS, pp 137–146. https://doi.org/10.5220/0005785501370146
39. Pautasso C, Zimmermann O, Amundsen M, Lewis J, Josuttis NM (2017) Microservices in practice, part 1: reality check and service design. IEEE Softw 34(1):91–98. https://doi.org/10.1109/MS.2017.24
40. Petersen K, Feldt R, Mujtaba S, Mattsson M (2008) Systematic mapping studies in software engineering. In: Proceedings of the 12th international conference on evaluation and assessment in software engineering (EASE'08). BCS Learning & Development Ltd, pp 68–77
41. Richards M (2016) Microservices antipatterns and pitfalls, 1st edn. O'Reilly Media, Inc., Newton
42. Richardson C (2014) Microservices: decomposing applications for deployability and scalability. InfoQ. https://www.infoq.com/articles/microservices-intro. Accessed 5 June 2019
43. Richardson C (2018) Microservices patterns, 1st edn. Manning Publications, New York
44. Ruecker B (2018) 3 common pitfalls of microservices integration and how to avoid them. InfoWorld. https://www.infoworld.com/article/3254777/3-common-pitfalls-of-microservices-integrationand-how-to-avoid-them.html. Accessed 5 June 2019
45. Saleh T (2016) Microservices antipatterns. InfoQ. https://www.infoq.com/presentations/cloud-anti-patterns. Accessed 5 June 2019
46. Savchenko D, Radchenko G, Taipale O (2015) Microservices validation: Mjolnirr platform case study. In: 2015 38th International convention on information and communication technology, electronics and microelectronics (MIPRO), pp 235–240. https://doi.org/10.1109/MIPRO.2015.7160271
47. Soldani J, Tamburri DA, Van Den Heuvel WJ (2018) The pains and gains of microservices: a systematic grey literature review. J Syst Softw 146:215–232. https://doi.org/10.1016/j.jss.2018.09.082
48. Stocker M, Zimmermann O, Lübke D, Zdun U, Pautasso C (2018) Interface quality patterns—communicating and improving the quality of microservices APIs. In: 23rd European conference on pattern languages of programs 2018
49. Taibi D, Lenarduzzi V (2018) On the definition of microservice bad smells. IEEE Softw 35(3):56–62. https://doi.org/10.1109/MS.2018.2141031
50. Taibi D, Lenarduzzi V, Pahl C (2017) Processes, motivations, and issues for migrating to microservices architectures: an empirical investigation. IEEE Cloud Comput 4(5):22–32. https://doi.org/10.1109/MCC.2017.4250931
51. Taibi D, Lenarduzzi V, Pahl C (2018) Architectural patterns for microservices: a systematic mapping study. In: Proceedings of the 8th international conference on cloud computing and services science—volume 1: CLOSER. SciTePress, pp 221–232. https://doi.org/10.5220/0006798302210232
52. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in software engineering: an introduction. Kluwer, Dordrecht
53. Wolff E (2016) Microservices: flexible software architecture, 1st edn. Addison-Wesley, Reading
54. Zimmermann O (2017) Architectural refactoring for the cloud: a decision-centric view on cloud migration. Computing 99(2):129–145. https://doi.org/10.1007/s00607-016-0520-y
55. Zimmermann O (2017) Microservices tenets. Comput Sci Res Dev 32(3–4):301–310. https://doi.org/10.1007/s00450-016-0337-0
56. Zimmermann O, Stocker M, Lübke D, Zdun U (2017) Interface representation patterns—crafting and consuming message-based remote APIs. In: 22nd European conference on pattern languages of programs (EuroPLoP 2017), pp 1–36. https://doi.org/10.1145/3147704.3147734



Davide Neri is a Ph.D. candidate at the University of Pisa (Italy), where he is member of the Service Oriented, Cloud and Fog Computing research group. He holds a MSc in Computer Science (2016, University of Pisa). His research interests include, but are not limited to, microservices and containerization. In particular, he is working on models for supporting the design of microservices and on tools for automatically deploying them by exploiting container-based virtualization.

Jacopo Soldani is a post-doc researcher at the University of Pisa (Italy). He holds a Ph.D. in Computer Science (2017, University of Pisa). His research interests include, but are not limited to, service-oriented and cloud computing, adaptation, coordination, and integration of software elements, and formal methods. He is member of the IFIP Working Group on Service-Oriented Systems (IFIP WG 2.14/6.12/8.10) and of the OASIS TOSCA technical committee, and he has also been involved in several research projects on service, cloud and fog computing both at local and EU level.

Olaf Zimmermann is a professor of software architecture and an institute partner at the University of Applied Sciences of Eastern Switzerland, Rapperswil who lectures and provides industry consulting and training. His research areas include service-oriented computing and architectural knowledge management. In previous roles, Zimmermann was a Senior Principal Scientist at ABB Corporate Research and a Research Staff Member and Executive IT Architect at IBM. He received his PhD in Computer Science in 2009 from Stuttgart University (Germany). As solution architect on professional services projects, Zimmermann has helped international clients in multiple industries build, integrate, and modernize enterprise applications and other information systems. The Open Group has awarded him a Distinguished IT Architect (Chief/Lead) Certification. He is a book author, an editor of the Insights column in IEEE Software, and the leader of the Microservices API Patterns (MAP) initiative.

Antonio Brogi is full professor at the Department of Computer Science, University of Pisa (Italy) since 2004. His research interests include service-oriented, cloud-based and fog computing, coordination and adaptation of software elements, and formal methods. He has published the results of his research in over 180 papers in international journals and conferences. He is member of the editorial board of the journals IEEE Transactions on Cloud Computing and Elservier Journal of Computer Languages.