On the Definition of Microservice Bad Smells

Article *in* IEEE Software · May 2018

DOI: 10.1109/MS.2018.2141031

2 authors:

Davide Taibi
Tampere University of Technology
80 PUBLICATIONS **426** CITATIONS

SEE PROFILE

Valentina Lenarduzzi
Tampere University of Technology
26 PUBLICATIONS **80** CITATIONS

SEE PROFILE

On the Definition of Microservice Bad Smells

Davide Taibi and Valentina Lenarduzzi, Tampere University of Technology

Abstract: Code smells and architectural smells (also called *bad smells*) are symptoms of poor design that can hinder code understandability and decrease maintainability. Several bad smells have been defined in the literature for both generic architectures and specific architectures. However, cloud-native applications based on microservices can be affected by other types of issues. In order to identify a set of microservice-specific bad smells, researchers collected evidence of bad practices by interviewing 72 developers with experience in developing systems based on microservices. Then, they classified the bad practices into a catalog of 11 microservice-specific bad smells frequently considered harmful by practitioners. The results can be used by practitioners and researchers as a guideline to avoid experiencing the same difficult situations in the systems they develop.

Keywords: microservice, antipattern, anti-pattern, code smell, architectural smell, bad smell, cloud computing, software development, software engineering

Microservices are currently enjoying increasing popularity and diffusion in industrial environments, being adopted by several big players such as Amazon, LinkedIn, Netflix, and SoundCloud. Microservices are relatively small and autonomous services that work together, are modeled around a business capability, and have a single and clearly defined purpose.[1,2] Microservices enable independent deployment, allowing small teams to work on separated and focused services by using the most suitable technologies for their job that can be deployed and scaled independently.[1,2] Microservices are a newly developed architectural style. Several patterns and platforms such as nginx (www.nginx.org) and Kubernetes (kubernetes.io) exist on the market. During the migration process, practitioners often face common problems, which are due mainly to their lack of knowledge regarding bad practices and patterns.[3,4]

In this article, we provide a catalog of bad smells that are specific to systems developed using a microservice architectural style, together with possible solutions to overcome these smells. To produce this catalog, we surveyed and interviewed 72 experienced developers over the course of two years, focusing on bad practices they found during the development of microservice-based systems and on how they overcame them. We identified a catalog of 11 microservice-specific bad smells by applying an open and selective coding[5] procedure to derive the smell catalog from the practitioners' answers.

The goal of this work is to help practitioners avoid these bad practices altogether or deal with them more efficiently when developing or migrating monoliths to microservice-based systems.

As with code and architectural smells, which are patterns commonly considered symptoms of bad design,[1,6] we define microservice-specific bad smells (called "microservice smells" hereafter) as indicators of situations—such as undesired patterns, antipatterns, or bad practices—that negatively affect software quality attributes such as understandability, testability, extensibility, reusability, and maintainability of the system under development.

Background

Several generic architectural-smell detection tools and practices have been defined in the past years.[7–9] Moreover, several microservice-specific architectural patterns have been defined.[10] However, to the best of our knowledge, no peer-reviewed work and, in particular, no empirical studies have proposed bad practices, antipatterns, or smells specifically concerning microservices.

However, some practitioners have started to discuss bad practices in microservices. In his ebook *Microservices AntiPatterns and Pitfalls*, Mark Richards introduced three main pitfalls: Timeout, I Was Taught to Share, and Static Contract Pitfall.[11] Moreover, in the past two years, practitioners have given technical talks about bad practices they experienced when building microservices. In Table 1, we summarize the main bad practices presented in these works. Unlike these works, we identified a set of microservice smells based on bad practices reported by 72 participants. Later, we map our set of microservice smells to the bad practices identified in Table 1.

Table 1. The main pitfalls proposed in non-peer-reviewed literature and practitioner talks.	
Bad practice	***Description***
Timeout[11] (also named Dogpiles[12])	The service consumer cannot connect to the microservice. Mark Richards recommends using a time-out value for service responsiveness or sharing the availability and the unavailability of each service through a message bus, so as to avoid useless calls and potential time-outs due to service unresponsiveness.[11]
I Was Taught to Share[11]	Sharing modules and custom libraries between microservices.
Static Contract Pitfall[11,12]	Microservice APIs that aren't versioned, possibly causing service consumers to connect to older versions of the services.
Mega-Service[13]	A service that is responsible for many functionalities and should be decomposed into separated microservices.
Shared Persistence[13,14]	Using shared data among services that access the same database.
Data Ownership[14]	Data should not be directly shared among different services. Microservices should own only the data they need and possibly share it via APIs.
Leak of Service Abstraction[13]	Designing service interfaces for generic purposes and not specifically for each service.
Hardcoded IPs and Ports[12]	Hard-coding the IP address and ports of communicating services, therefore making it harder to change the service location afterward.
Not Having an API Gateway[15]	Services directly exposed to the outside and connected to each other. Services should not be exposed through an API gateway layer and should not be connected directly, so as to simplify the connection and support monitoring, and authorization issues should be delegated to the API gateway. Moreover, changes to the API contract can be easily managed by the API gateway, which is responsible for serving the content to different consumers, providing only the data they need.
Lust[16]	Using the latest technologies.
Gluttony[16]	Using too many different communication protocols such as HTTP, protocol buffers, Thrift, etc.
Greed[16]	Services all belonging to the same team.
Sloth[16]	Creating a distributed monolith due to the lack of independence of microservices.
Wrath[16]	Blowing up when bad things happen.
Envy[16]	The shared-single-domain fallacy.
Pride[16]	Testing in the world of transience.

Setting the Stage

We conducted a survey among experienced developers, collecting bad practices in microservice architectures and how they overcame them. We collected information in interviews, both in a structured fashion, via a questionnaire with closed answers, and in a less structured way, by asking the interviewees open-ended questions to elicit additional relevant information (such as possible issues when migrating to microservices).

One of the most important goals of the questionnaire was to understand which bad practices have the greatest impact on system development and which solutions are being applied by practitioners to overcome them. Thus, we asked the interviewees to rank every bad practice on a scale from 0 to 10, where 0 meant "the bad practice is not harmful" and 10 meant "the bad practice is extremely harmful." Moreover, we clarified that only the ranking of the bad practices has real meaning.

For example, a value of 7 for the Hardcoded IPs bad practice and 5 for Shared Persistence shows that Hardcoded IPs is believed to be more harmful than Shared Persistence, but the individual values of 7 and 5 have no meaning in themselves. A harmful practice is a practice that has created some issue for the

practitioner, such as increasing maintenance effort, reducing code understandability, or increasing faultiness.

The interviews were based on a questionnaire organized into four sections, according to the information we aimed to collect:

- *Personal and company information*. The interviewee's role and company's application domain.
- *Experience in developing microservice-based systems*. The number of years of experience in developing microservices. This question was asked to ensure that data was collected only from experienced developers.
- *Microservice bad practices' harmfulness*. A list of the practices that created some issues during the development and maintenance of microservice-based applications, ranked according to their harmfulness on a 10-point Likert scale. Moreover, for each practice, we asked the practitioners to report what problems it generated and why they considered it harmful. For this answer, the interviewer did not provide any hints, letting the participants report the bad practices they had faced while developing or maintaining microservice-based systems. Moreover, in order to avoid influencing the interviewees, we asked them to list their own bad practices, without providing them with a list of pitfalls previously identified by practitioners.[4,12–15]
- *Bad-practice solutions*. For each bad practice identified, how the participants overcame it.

All interviews were conducted in person. We understand that an online questionnaire might have yielded a larger set of answers. However, we believe that face-to-face interviews are more reliable for collecting unstructured information, as they allow establishing a more effective communication channel with the interviewees and make it easier to interpret the answers to open-ended questions.

The interviewees were asked to provide individual answers, even if they worked in the same group. This allowed us to get a better understanding of different points of view, and not only of the company point of view. The interviews were designed to take 15 minutes per participant. However, the open discussion took longer than expected, resulting in an average of 21 minutes per participant.

We selected the participants from the attendees of practitioner events and conferences. That is, we interviewed 21 participants of the 2016 International Conference on Agile Software Development (XP 2016), seven participants of the 2017 Workshop on Microservices in Agile Software Development,[17] 13 participants of XP 2017, and 31 practitioners at several minor developers' events in Italy and Germany between January and July 2017.

During the interviews, we first introduced our goals to the participants. We then asked them if they had at least two years of experience in developing microservice-based systems, so as to save time and avoid bothering inexperienced practitioners.

The Survey Results

We conducted 72 interviews with participants belonging to 61 different organizations. No inexperienced participants, such as students, academics, or nonindustrial practitioners, were considered for the interviews. Of all the interviewees, 36% were software architects, 19% were project managers, 38% were experienced developers, and 7% were agile coaches. All the interviewees had at least five years of experience in software development. Of all the interviewees, 28.57% worked for banks, 28.57% worked for companies that produce and sell only their own software as a service (e.g., website builders, mobile app generators, and others), 23.81% worked in consultancy companies specializing in migration to microservices, 9.52% worked in the IT department of public administrations, and 9.52% worked in telecommunications companies. Seventeen percent had adopted microservices for more than five years, 60% had adopted them for three to four years, and the remaining 23% had adopted them for two to three years.

The practitioners reported a total of 265 different bad practices together with the solutions they had applied to overcome them. Each of us grouped similar practices (considering both the description and the justification of the harmfulness provided by the participants) by means of open and selective coding.[5] In cases where we interpreted the descriptions differently, we discussed incongruences so as to achieve agreement on similar issues. Each participant reported an average of 3.68 bad practices, which, after the selective-coding process, resulted in 11 microservice smells.

The list of the resulting smells, together with their descriptions and the possible solutions indicated by the practitioners, is reported in Table 2.

Table 2. Catalog of microservice smells.

Microservice smell	*Description (Desc.) / Detection (Det.)*	*Problem it may cause (P) / Adopted solutions (S)*
API Versioning	**Desc.:** APIs are not semantically versioned. **Det.:** A lack of semantically consistent versions of APIs (e.g., v1.1, 1.2, etc.) Also proposed as Static Contract Pitfall.[11,12]	**P:** In the case of new versions of non-semantically-versioned APIs, API consumers may face connection issues. For example, the returning data might be different or might need to be called differently. **S:** APIs need to be semantically versioned to allow services to know whether they are communicating with the right version of the service or whether they need to adapt their communication to a new contract.
Cyclic Dependency	**Desc.:** A cyclic chain of calls between microservices exists. **Det.:** The existence of cycles of calls between microservices; e.g., A calls B, B calls C, and C calls back A.	**P:** Microservices involved in a cyclic dependency can be hard to maintain or reuse in isolation. **S:** Refine the cycles according to their shape,[4] and apply the API Gateway pattern.[2]
ESB Usage	**Desc./Det.:** The microservices communicate via an enterprise service bus (ESB). An ESB is used for connecting microservices.	**P:** An ESB adds complexities for registering and deregistering services on it. **S:** Adopt a lightweight message bus instead of the ESB.
Hard-Coded Endpoints	**Desc./Det.:** Hardcoded IP addresses and ports of the services between connected microservices exist. Also proposed as Hardcoded IPs and Ports.[9]	**P:** Microservices connected with hardcoded endpoints lead to problems when their locations need to be changed. **S:** Adopt a service discovery approach.
Inappropriate Service Intimacy	**Desc.:** The microservice keeps on connecting to private data from other services instead of dealing with its own data. **Det.:** A request for private data of other microservices. A direct connection to other microservices' databases.	**P:** Connecting to private data of other microservices increases coupling between microservices. The problem could be related to a mistake made while modeling the data. **S:** Consider merging the microservices.
Microservice Greedy	**Desc.:** Teams tend to create new microservices for each feature, even when they are not needed. Common examples are microservices created to serve only one or two static HTML pages. **Det.:** Microservices with very limited functionalities (e.g., a microservice serving only one static HTML page).	**P:** This smell can generate an explosion of the number of microservices composing a system, resulting in a useless huge system that will easily become unmaintainable because of its size. **S:** Carefully consider whether the new microservice is needed.
Not Having an API Gateway	**Desc.:** Microservices communicate directly with each other. In the worst case, the service consumers also communicate directly with each microservice, increasing the complexity of the system and decreasing its ease of maintenance. **Det.:** Direct communication between microservices. Also proposed as Not Having an AP Gateway.[15]	**P:** Our interviewees reported being able to work with systems consisting of 50 interconnected microservices. However, if the number was higher, they started facing communication and maintenance issues. **S:** Apply the API Gateway pattern[2] to reduce the communication complexity between microservices.
Shared Libraries	**Desc./Det.:** Shared libraries between different microservices are used.	**P:** Microservices are tightly coupled together, leading to a loss of independence between them. Moreover, teams need to coordinate with each other when they need to modify the shared library.

		S: Two possible solutions are to 1. accept the redundancy to increase dependency among teams, or 2. extract the library to a new shared service that can be deployed and developed independently by the connected microservices.
Shared Persistency	**Desc./Det.:** Different microservices access the same relational database. In the worst case, different services access the same entities of the same relational database. Also proposed as Data Ownership.[14]	**P:** This smell highly couples the microservices connected to the same data, reducing team and service independence. **S:** Three possible solutions are to 1. use independent databases for each service, 2. use a shared database with a set of private tables for each service that can be accessed by only that service, or 3. use a private database schema for each service.
Too Many Standards	**Desc./Det.:** Different development languages, protocols, frameworks, etc. are used. Also proposed as the Lust and Gluttony bad practices.[16]	**P:** Although microservices allow the use of different technologies, adopting too many different technologies can be a problem in companies, especially in the event of developer turnover. **S:** Carefully consider the adoption of different standards for different microservices, without following the latest hype.
Wrong Cuts	**Desc.:** Microservices are split on the basis of technical layers (presentation, business, and data layers) instead of business capabilities.	**P:** The wrong separation of concerns and increased data-splitting complexity can occur. **S:** Perform a clear analysis of business processes and the need for resources.

Data Analysis and Interpretation

The answers were analyzed mainly using descriptive statistics. No noticeable differences emerged among different roles or domains. Three smells (Wrong Cuts, Hard-Coded Endpoints, and Shared Persistency) were reported and were considered very harmful or moderately harmful by more than 50% of the participants. Wrong Cuts turned out to be the most frequently mentioned smell and one of the two smells considered the most harmful. Based on the practitioners' answers, splitting a monolithic application is always a complex task, especially because developers are used to splitting applications into horizontal layers (database, business logic, etc.) and tend to adopt such an approach out of habit instead of considering splitting applications based on business processes.

Some smells were reported as symptoms of a lack of experience in using microservices on the part of the company or the developers. The practitioners reported facing most problems in the early stage of adopting microservices (from six months to one year). All the smells except for Not Having an API Gateway were perceived as harmful from the beginning of the adoption of microservices, while Not Having an API Gateway was usually perceived as a problem only when the number of microservices increased and communication between services became hard to manage. Cyclic Dependency, also considered a bad practice in different architectures,[18] was reported to be a very harmful practice, even though it was reported by only nine of the 72 practitioners (3.4%).

Most smells are easily removed by means of simple technical solutions; however, Wrong Cuts, Microservice Greedy, and Too Many Standards do not have a straightforward solution. In these cases, teams need to be trained and agree on the development strategies, such as when to create a new microservice or how to select the technology to be adopted in different services. Other smells, if experienced during the migration of a monolithic system, may be symptoms of an incomplete migration. For instance, the practitioners reported that using an enterprise service bus (ESB) for communication between microservices may be acceptable in early migration phases, but the ESB should be replaced by a lightweight message bus

as soon as possible.

Table 3 lists the microservices smells together with the number and percentage of the reported bad practices and the median of the reported perceived harmfulness.

Table 3. The microservice smells identified in the survey.

Microservice smell	Bad practices reported		Median perceived harmfulness (0–10)*
	No.	%	
Wrong Cuts	51	19.2	8
Hard-Coded Endpoints	38	14.3	8
Cyclic Dependency	9	3.4	7
Shared Persistency	41	15.5	6.5
API Versioning	19	7.2	6.5
ESB Usage	24	9.1	6
Not Having an API Gateway	17	6.4	5
Inappropriate Service Intimacy	15	5.7	5
Shared Libraries	31	11.7	4
Too Many Standards	7	2.6	4
Microservice Greedy	13	4.9	3

* Harmfulness was measured on a 10-point Likert scale, where 0 meant "the bad practice is not harmful" and 10 meant "the bad practice is extremely harmful."

The results of this work are subject to some threats to validity, due mainly to the selection of the survey participants and to the data interpretation phase. Different respondents might have provided a different set of answers. To mitigate this threat, we selected a relatively large set of participants working in different companies and different domains. During the survey, we did not propose a predefined set of bad practices to the participants; therefore, their answers are not biased by the results of previous work. However, as the surveys were carried out during public events, we are aware that some participants may have shared some opinions with others during breaks; therefore, some answers might have been partially influenced by previous discussions. Finally, the answers were aggregated independently by each of us by means of open and selective coding.[5] If this process had been carried out by different researchers, it might have led to a different set of smells.

In this article, we identified a set of 11 microservice smells based on 265 bad practices experienced by 72 practitioners while developing microservice-based systems. Out of the 16 bad practices described in practitioners' talks (see Table 1),[4,12–15] only six were confirmed by our interviewees.

The results show that splitting a monolith, including splitting the connected data and libraries, is the most critical issue, resulting in potential maintenance issues when the cuts are not done properly. Moreover, the conversion to a distributed system increases the system's complexity, especially when dealing with connected services that need to be highly decoupled from any point of view, including communication and architecture (the smells involved here are Hard-Coded Endpoints, Not Having an API Gateway, Inappropriate Service Intimacy, and Cyclic Dependency).

This work resulted in the following five lessons learned:

- *Lesson learned 1*. Besides traditional smells, microservice smells can also be problematic for the development and maintenance of microservice-based systems. Developers can already benefit from our catalog by learning how to avoid experiencing the related bad practices.
- *Lesson learned 2*. The role of the software architect is becoming important again. Architectural, system-level decisions must be made based upon deep knowledge of microservices.
- *Lesson learned 3*. Splitting a monolith into microservices is about identifying independent business processes that can be isolated from the monolith, and not only about extracting features in different web services.
- *Lesson learned 4*. The connections between microservices, including the connections to private data and shared libraries, must be carefully analyzed.
- *Lesson learned 5*. As a general rule, developers should be alerted if they need to have deep knowledge of the internal details of other services or if changes in a microservice require changes in another microservice.

The proposed catalog of smells can be used by practitioners as a guideline to avoid the same problems happening to them that were faced by our interviewees. Moreover, the catalog is also a starting point for additional research on microservices. It is important to note that, even though the identified smells reflect the opinion of 72 developers working in 61 different companies, the rating of the harmfulness of the reported smells is based only on the perception of the practitioners and needs to be empirically validated.

Indeed, a deeper investigation is needed to evaluate the harmfulness and the comprehensiveness of our catalog. This, together with more in-depth empirical studies (such as controlled experiments), will be part of our future work.

References

1. J. Lewis and M. Fowler, "Microservices," 25 Mar. 2014; www.martinfowler.com/articles/microservices.html.
2. S. Newman, *Building Microservices*, O'Reilly, 2015.
3. C. Richardson, *Microservices Patterns*, Manning Publications, 2017.
4. D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation," *IEEE Cloud Computing*, vol. 4, no. 5, 2017, pp. 22–32; doi:10.1109/MCC.2017.4250931.
5. A.L. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, SAGE Publications, 2008.
6. D. Taibi, A. Janes, and V. Lenarduzzi, "How Developers Perceive Smells in Source Code: A Replicated Study," *Information and Software Technology*, Dec. 2017, pp. 223–235, doi:10.1016/j.infsof.2017.08.008.
7. N. Moha et al., "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Trans. Software Eng.*, vol. 36, no. 1, 2010, pp. 20–36.
8. J. Garcia et al., "Identifying Architectural Bad Smells," *Proc. 13th European Conf. Software Maintenance and Reengineering*, 2009, pp. 255–258.
9. I. Macia et al., "Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity? An Exploratory Analysis of Evolving Systems," *Proc. 11th Ann. Int'l Conf. Aspect-Oriented Software Development* (AOSD 12), 2012, pp. 167–178.
10. D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study," to be published in *Proc. 8th Int'l Conf. Cloud Computing and Services Science* (CLOSER 18), 2018.
11. M. Richards, *Microservices AntiPatterns and Pitfalls*, O'Reilly, 2016.
12. T. Saleh, "Microservices Antipatterns," presentation at QCon London 2016, 2016; www.infoq.com/presentations/cloud-anti-patterns.

13. R. Shoup, “From the Monolith to Microservices: Lessons from Google and eBay,” presentation at Craft Conf. 2016, 2016; www.ustream.tv/recorded/61479577.
14. J. Bogard, “Avoiding Microservice Megadisasters,” presentation at 2017 NDC London Conf., 2017; www.youtube.com/watch?v=gfh-VCTwMw8.
15. V. Alagarasan, “Microservices Antipatterns,” presentation at API360 Microservices Summit, 2016; www.youtube.com/watch?v=uTGIrzzmcv8.
16. D. Bryant, “The Seven (More) Deadly Sins of Microservices,” presentation at O’Reilly OSCON 2016, 2016; www.youtube.com/watch?v=VG5ZOOb5T9o&list=PL055Epbe6d5ZfLyERAIRXVCATx9BUP3-Q&index=7.
17. D. Taibi et al., “Microservices in Agile Software Development: A Workshop-Based Study into Issues, Advantages, and Disadvantages,” *Proc. XP2017 Scientific Workshops* (XP 17), 2017, article 23.

F.A. Fontana et al., “Automatic Detection of Instability Architectural Smells,” *Proc. 2016 IEEE Int’l Conf. Software Maintenance and Evolution* (ICSME 16), 2016, pp 433–437