Do Microservices Prevent High Qualitative Code?

Marcus Hilbrich[1], Christine Jakobs[1], and Matthias Werner[1]

Technische Universitt Chemnitz, Chemnitz, Saxony, Germany
[marcus.hilbrich|christine.jakobs|matthias.werner]@informatik.tu-chemnitz.de

During the last years many industry areas started to use microservices. They show advantages like scalability, independent service development, and complexity management. Nevertheless, from a researcher point of view we observe also drawbacks in using microservices. The aim of this paper is to present some of them to initiate a discussion whether they are caused by misconception or by misuse of the concept.

We present three observations we want to emphasize and use the microservice definition from the paper of Lewis[1] et al. [6]. Therefore we see microservices as an architecture [... to describe a particular way of designing software applications as suites of independently deployable services] [6].

1 Reduced Code Quality

Lewis et al. see microservices as [... an approach to developing a single application as a suite of small services ...] [6]. From our observations this is often interpreted as small code size and leads to the following conclusions:

1. A service does not need maintenance because reimplementation appears to be easy. Thus, no documentation is needed.
2. The individual microservices decrease to poorly managed code fragments ("Grains of Sand" [9] antipattern).
3. Validation is not needed based on the clear (small) scale of a single service.
4. It is acceptable for a microservice to fail, the overall system will take care and cover the problem.

The first points may arise from an overall carelessness from the developers. Small and easy to re-implement code fragments are easy to oversee. Therefore, the developers tend to take less effort in preparing their code to validation and maintenance tasks. Overall, this may lead to bad code quality, as described by Pratt [8] as "Service Orphan", and by Alagarasan [1]. In personal discussions, we got even more feedback to this topic, but public statements are rare.

Especially the last point is also reflected in the literature through composition concepts for microservices. They include many providers of the same service to manage a high availability of the dependent service in case the preferred provider fails [2].

Microservice rely on the *design for failure* [6] principle to tolerate server failures. The service has to be able to cope with exceptions from missing responses. Therefore, instead of caring less the developers should put even more effort to safeguard their code for own failures and those of other services.

Our experience shows that many developers in all areas tend to rely on supporting concepts of the underlying software layers. This confidence reduces the overall system quality dramatically. Reduced code quality may impact security, availability and other non-functional properties, e.g. described in the "I Was Taught to Share" [9] antipattern. Therefore, the root cause is not misconception, and the solution might be to better emphasize the underlying concept, e.g.. One idea would be to start in an early stage of programming education.

[1]The paper was originally published in German but is available in an English version in the web.

2 Reduced Reuse of Code

Microservices are [... independently deployable by fully automated deployment machinery] [6]. This should result in forging the application by pinning together services of different providers. If one does not want to do this, the independence requirement leads to two problems:

1. The *not invented here syndrome* rises, or
2. code of (other teams) is reused by copy and paste.

On the one hand, the not invented here syndrome is a general problem and is complementary to the last mentioned problem. Either the developer do not even look for common solutions and follow code reuse strategies or they tend to copy other solutions without reflecting their assumptions.

In discussions with microservice developers from industry, software architects for microservices, consulting engineers, and academics we often discover this kind of problems. So far, we not have found reasonable academic research of identifying such antipatterns. At least we can identify that the problem is described for microservices, e.g. as "I Was Taught to Share" [9] antipattern, "Dare To Be Different" [8] antipattern, and is also mentioned in [5, 10].

A common approach in development teams is to build an own set of code for often needed concepts, sometimes even not on team level but by individual developers. Common solutions on individual level may lead to introducing bugs into their invented services. Bug fixes or updates introduce an additional effort since they have to be propagated into all projects the code is used in.

A code base managed on team level is sometimes done by introducing libraries. Without the microservice approach this is a good solution because changes, e.g. bug fixes are automatically propagated. Since the microservice concept relies on the *services as independent components* paradigm, using external libraries is an often discussed solution, e.g. in [1, 5, 9].

3 Missing Abstraction Mechanisms

Microservices are built on the assumption of independence. Horizontally services just define how its interfaces look like. In the vertical manner the service is independent from other services but relies on middle-ware and communication systems. Therefore, the microservice concept does not offer some well-known and established abstraction concepts, e.g. inheritance of services. This leads to two problems in relation to development complexity:

1. Re-implementation of concrete services
2. Cross-cutting concerns result in many abstract services (e.g. described by "Logging Can Wait" [8])

The first problem was seen as solved by the introduction of object orientation. Object orientation has been shown as an excellent method to present the structure of real world objects, and as a method to represent this structure as program structure. In microservices the basically same logic has to be implemented with a small add-on [3, 7].

Cross-cutting concerns like logging, authorization of users, and description of run-time behavior have to be implemented in separate services. In layered architectures those non-functional parts are covered by underlying software layers. The microservice architecture does not guarantee that e.g. security-related service are used, in contradiction to e.g. aspect oriented programming [4]. Especially security critical services are difficult to request from unknown service providers. In contrast to the business-logic services those cross-cutting services expand the number of maintained services in an unnecessary way.

The conceptually missing but commonly accepted abstraction mechanisms lead to a rising complexity in the development and evaluation. This is a current and ongoing challenges of microservice systems.

4 Conclusion and Open Points

We presented three observations related to microservices. The presented examples just illustrate some challenges which are common also in other programming paradigms but extend in the microservice concept. The code quality in general is not directly reduced by the microservice concept, but is getting less attention through it. Based on the independent development of services, code reuse is difficult. Not possible are some concepts of abstraction, that are e.g. used by object orientation. In conclusion we want to launch a discussion about possible improvements of the microservice concept.

References

[1] Vijay Alagarasan. Seven Microservices Anti-patterns, 2015.

[2] Rachid Hamadi and Boualem Benatallah. A Petri Net-based Model for Web Service Composition. In *Proceedings of the 14th Australasian Database Conference - Volume 17*, ADC '03, pages 191–200, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[3] Marcus Hilbrich and Markus Frank. Abstract Fog in the Bottle - Trends of Computing in History and Future. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 519–522, Aug 2018.

[4] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[5] Graham Lea. Why Dont Use Shared Libraries in Microservices is Bad Advice, 2016.

[6] James Lewis and Martin Fowler. Microservices: a Definition of this new Architectural Term, 2014.

[7] Reinhardt Lukas. Ein objektorientierter Ansatz zum Generieren von Microservices : Erstellung eines Prototyps. Bachelorthesis, Technische Universitt Chemnitz, Germany, 2018.

[8] Michael Pratt. Microservice Pitfalls & AntiPatterns, Part 1, 2016.

[9] Mark Richards. *Microservices Antipatterns and Pitfalls.* 2016.

[10] Nathan Vega. Answering Your Microservices Webinar Questions, 2015.