A paper on ‘Hidden dividends of Microservices by Tom Killalea’

# Overview

The article “Hidden dividends of Microservices” written by Tom Killalea describes the less obvious dividends of microservices. The author argues that these dividends are significant, and the implementers must be aware of them to reap maximum benefit from the process of migrating from monolithic software architectures.

Do I agree? Yes. While this is a nuanced and highly subjective architecture issue - I find most arguments and the contexts in which they are made spot on.

# Definition

Microservices are in the way the opposite of monolithic architectures. Monolithic means large single complex systems.

When complex monolithic systems are broken into their essential services in such a way that the services are well defined and independent and only interact with each other in fixed distinct ways - they are called microservices. Their interactions can be accomplished by carefully designed API’s.

As per this article, the obvious benefits of microservices are:

1. Agility (Speed of development)
2. Resilience
3. Scalability
4. Developer productivity

However, the author points out that migrating process may be complicated and not for every company. So if chosen, the microservices should be implemented with some of the hidden dividends as a goal.

The next sections summarize my understanding of these hidden dividends, followed by an account of my personal experience (limited yet relevant as I took a fresh look at it after reading this article).

# The hidden dividends and my understanding from my experience

## Permissionless innovation:

Here, ***permission*** refers to the need to consult with teams, i.e., Between different teams that implement microservices which interact with each other. If API for interaction is agreed upon early - then there would be less need to take permission from other teams for implementing a service. This according to the author - enhances innovation because the implementers will have higher independence and will tend to experiment more.

## Enable failure:

Large complex systems are bound to fail. When failure happens in a large monolithic system, it is hard to trace and debug. In microservices, it will not get any easier!

The author argues that microservices may add complexity because debugging may need to be done across services and not just within a service. However, by acknowledging that failure rate may be higher and routine, the implementers can have useful discussions about issues related to the failure. This may, in turn, lead to preemptive measures to minimize effects of failures.

## Disruption of Trust:

In this ***“Trust”*** represents a problem for development. There is a limit to personal interaction, and beyond this limit (Dunbar’s number) trust based monitoring is impractical.

Microservices enable inter-service communication through APIs. Individuals can not carefully monitor the performance of these APIs - So microservices can achieve it by using SLAs (service level agreements) which govern the performance requirements.

The author quotes Conway’s rule that suggests that the communication structure of an organization is a mirror of the design structure of the system itself. This insight can be used to guide the implementers to scale well.

## You build it - You own it:

This refers to the ***“ownership”*** of a service by its implementing team. The team should be responsible for the entire lifecycle of the service - requirement gathering, design, development, and operation. The operation aspect brings in the customer into the loop. There will be a constant feedback loop between the client and the owning team which results in high-quality software. The quotes a ten-year-old conversation with Amazon CEO as we have the proof of such efforts in the constant improvements of the market leader in online retail today.

Also, in this time the adoption of such practices has become more common and lead to specific quality improvement methods such as

1. Continuous deployment
2. Virtualized or containerized capacity – Managing the ability of virtual machines and containers [2]
3. Automated elasticity
4. Self-healing techniques [3]

## Accelerate Deprecation:

Quickening the ***deprecation*** of slow or inefficient services (partly, or wholly with a new service) is easier with microservices architecture because it is easy to get a clear idea of the interaction between services. I believe, the author refers to the service that consumes the service being replaced as “consumer”.

The author also suggests how one service can be forced to deprecate fast by constraining teams so that resources are given to other services than the to-be-deprecated service. Then the onus to initiate the deprecation falls on the “consumer” service team.

## End centralized metadata:

Metadata is data about an implementation such as database schema, network, scaling and deployment policies [6]. Agents in the implementation process such as DB administrators, software engineers, and enterprise modelers may all need to take part in every review of a metadata change if metadata is centralized. With microservices implementation, care has to be taken to decentralize metadata as well. This will allow abstraction of data persistence mechanisms away from the APIs which is all the consumer is interested in. The dividend here will be the easy replacement of one mechanism with another.

## Concentrate the pain:

This is related to the compliance and governability aspect of microservices. With microservices it is possible to do “threat modeling” for the services - i.e., identify criticality of the data handled by each service and possibly rank them as least important to most important. Once such identification is made, it should be possible to ***“concentrate”*** the handling of such critical data into a minimal number of services so that the effort needed to make them robust will be concentrated on them and the other services can be implemented with more freedom and less caution.

## Test differently:

Software and system testing is not a solved problem. Compared to monoliths, microservices by their design offer a different perspective on testing. It is possible to think about testing much earlier in the design and development phase and also achieve greater coverage [7]. The author also offers an example of Yelp’s testing philosophy - Interface Testing and Functionality testing with a higher priority on interface testing first and then ensuring the functionality the clients see via interfaces.

Adoption of practices that go in hand with microservices implementation such as continuous or phased deployment, smoke tests (which I understand are preliminary tests of the most critical services) will ensure more robustness and decrease repair time when problems are found out during production. Finally, the author compiles a list of warning signs that can be helpful as a checklist for microservices.

## Personal experience

I worked for a company called Intergraph where I developed software for designing ships. In that company, we followed scrum process. We wrote APIs for which each member of our team is responsible for maintaining and fixing his or her code. The code is abstracted during the design phase and is 90% finalized before the implementation is stated. If there are any discoveries during the development phase, the design is modified.

ATPs (Automated Test Procedures) are written for each API which tests every line of the code we wrote in every possible way to make sure there are the least number of issues after it is delivered to the user. This way, the code is distributed and decentralized among the developers and was easy to maintain. The code is checked in and is made a part of the build process once in every four weeks. Other dependent APIs, ATPs, custom commands are run weekly and checked if there is any effect on the existing code. If there is a build failure, the owner of the API immediately fixes it. If the update doesn’t work, the changes were rolled back so that existing code is not affected. So, adopting microservice architecture principles in this company enabled continuous deployment and early testing. The code is easy to maintain. Any new developer could add or enhance any implementation with less effort.

## Uber Architecture - Another example [9]:

Uber started with a monolithic architecture offering its services for a single city. Later on, it began to expand to many countries. Their core domain models grew, and their components became tightly coupled. They had to add new features, fix bugs, deploy everything at once which made the developer activity increase rapidly.

So, they followed the successful companies’ architecture and adopted microservice architecture. Doing that, solved many of their problems. They still have to work on providing clear ownership and providing fault more tolerance.

# Conclusion

As evidenced by the concluding remarks and the warning signs mentioned in the article - microservices not done right is not microservices. It is rather difficult to migrate every system to microservices without understanding all the aspects of the migration and without aiming for all the non-obvious dividends mentioned in this article - it may not be worth the effort for all organizations. Also, I understand it is very subjective to each organization’s long-term goals. However, the giants in the industry such as Amazon, Netflix, and even Google are setting great examples and giving direction to others, and I believe microservices will be a way to go for most organizations of the future.

# References:

1. About the IETF

<https://www.ietf.org/about/>

1. What are containers and why do you need them?

<http://www.cio.com/article/2924995/enterprise-software/what-are-containers-and-why-do-you-need-them.html>

1. Technology Conversations

<https://technologyconversations.com/2016/01/26/self-healing-systems/>

1. be left holding the bag

<http://idioms.thefreedictionary.com/be+left+holding+the+bag>

1. Adopting Microservices at Netflix: Lessons for Architectural Design

<https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/>

1. 3 golden rules of microservices deployments

<http://devops.com/2015/05/07/3-golden-rules-microservices-deployments/>

1. What is test coverage in software testing?

<http://istqbexamcertification.com/what-is-test-coverage-in-software-testing-its-advantages-and-disadvantages/>

1. Smoke testing

<https://en.wikipedia.org/wiki/Smoke_testing_(software)>

1. SERVICE-ORIENTED ARCHITECTURE: SCALING THE UBER ENGINEERING CODEBASE AS WE GROW

<https://eng.uber.com/soa/>

1. The Hidden Dividends of Microservices

<http://queue.acm.org/detail.cfm?id=2956643>