Bachelor Thesis Project

Continuous architecture in a large distributed agile organization - a case study at Ericsson

Author: Magnus Standar
Supervisor: Johan Hagelbäck
Semester: VT 2017
Subject: Computer Science
Abstract

Agile practices have become norm, also in large scale organizations. Applying agile methods includes introducing continuous practices, including continuous architecture. For web scale applications microservices is a rising star. This thesis investigates if microservices could be an answer also for embedded systems to tackle the synchronizing problem of many parallel teams.

Keywords: Software architecture refactoring, embedded systems, continuous integration, large scale agile development
## Contents

1 Introduction
   1.1 Background
   1.2 Previous research
   1.3 Problem formulation
   1.4 Motivation
   1.5 Research Question
   1.6 Scope/Limitation
   1.7 Target group
   1.8 Outline

2 Method
   2.1 Method description, quantitative search:
   2.2 Method description, qualitative interviews
   2.3 Reliability and validity
   2.4 Ethical Considerations

3 Microservices

4 Architectural concerns for the equipment and resource management application in an Ericsson radio base station

5 Rationales for interviewed organization to go or not go service oriented

6 Analysis
   6.1 Literature: Microservices architecture related papers
   6.2 Literature: Container technology related papers
   6.3 Continuous refactoring

7 Discussion

8 Conclusion
   8.1 Future Research

9 References

10 Appendix
   10.1 Search result full text search "continuous architecture" OR "continuous architecting"
   10.2 Literature overview
   10.3 Interviews
1 Introduction

1.1 Background

1.1.1 Agile practices in large scale organizations

For many reasons large systems are decomposed into parts. Decomposition is to break a complex problem or system into parts that are easier to conceive, understand, program, and maintain, see reference [1]. How a system is partitioned is defined in the systems architecture. The systems architecture has the role of trading among the set of quality attributes the systems needs to fulfil, also known as the systems non-functional requirements. There are many aspects impacting what architecture fits a specific system. One of them is called Conway’s law, see reference [2], stating that a systems architecture/design will reflect the communication structure of the organization constructing it. Another is called BAPO, described by van der Linden et al. in their paper “Software Product Family Evaluation”, see reference [3]. This paper indicates there is a relation between the business need, the systems architecture, the process used in the organization developing the system, and last but not least the organizational structure.

Today agile practices have become norm, also in large scale development projects, see article “Scaling Agile Development” by Craig and Vodde, reference [4]. Agile practices embrace the concept of constant refactoring to keep the system as simple as possible given the latest set of requirement, described in the “Agile manifesto”, reference [5], which is further advocated by Berteig in his 4 key principles for refactoring, reference [6]. Consequently, with increasing number of teams working in parallel on the same components, it has become increasingly hard to perform required refactoring. In addition, short term feature delivery is often prioritized over long term development efficiency, which results in increased technical debt, which is further described in the papers by Martini et al., “Architecture Technical Debt: Understanding Causes and a Qualitative Model”, reference [7], and “Managing Architectural Technical Debt”, reference [8].

When multiple teams are working in parallel on a component it is critical to secure that one team does not destroy the work of other teams. A common tactic to address this risk is by applying continuous integration, securing the added functionality with automated test cases. Black box testing is seldom sufficient to cover all alternative flows; why black box tests are complemented with different levels of white box testing as described by Zalavadia on his web page “Basics of Software testing, Types of testing”, reference [9]. See Figure 1.1 for an overview of common test levels.
Introduction of continuous integration is good for many things, among others flow, described by Lacoste in the paper “Killing the Gatekeeper: Introducing a Continuous Integration System”, reference [11]. However, having white-box tests in the continuous integration loop cement component responsibilities and relations, making architectural refactoring increasingly hard.

This thesis is about architectural styles supporting continuous architecting in large scale agile organizations.

1.1.2 Application domain
The application domain is the equipment and resource management application in a radio base station. The application was initially designed for a single standard LTE base station developed by a pair of co-located teams. The teams were experienced, all had worked in the same domain before, but for other products. Supported number of configurations was few, focus was time to market. Availability requirements was relaxed. Upgrades were released twice a year. The two main driving architectural requirements was:
• It should be easy to support new variants of hardware.
• Upgrade should be automated.

Since then, way of working has changed. Agile methodologies have been applied, continuous integration and continuous deployment has been introduced. Software is released on a bi-weekly basis. The product supports multi-standard configurations; GSM, WCMA, and LTE running in parallel on shared hardware. Number of supported configurations counts in millions, and complexity has multiplied. Availability demands are more than 5 nines availability, software upgrades included. Teams have grown in numbers and are now spread over five sites, three countries and two time zones. With increased number of teams, level of experience differs within and between teams.

The applied architecture is a component based architecture heavily influenced by object oriented design practices. The application is deployed over a set of connected general purpose processors. The application is spread over a limited set of boards. High capacity communication is provided to interconnect the processors and the boards.

The product contains high speed data plane applications and a low speed Operations & Maintenance and control plane application. This thesis targets the Operations & Maintenance and control plane application.

1.1.3 Definitions (as used in this report)
Continuous architecture, reference [12]:
Before the era of agile practices, it was norm to specify a wanted architecture before the product was developed. The role of the architect was to create and maintain architectural description. Normally many views of the architecture were specified and maintained. The architecture was described as the blueprint for the product to be. This strategy often ended up with “big up-front design”.

Agile practitioners adhere to the idea that not everything is known upfront, and therefore postpone decision making to the last responsible moment. Decisions should be based on fact, not guesses. As a consequence, decision making is spread out over the complete development cycle.

The architect’s role has changed; instead of maintaining models over the product architecture, the prime responsibility is to support and take timely architecture related decision during the development of the product. Additionally, the focus is on the realization of the product architecture, not on the documentation of the architecture. That is, the architect shall secure that the systems architecture is fit for the current set of requirements. Not yesterday’s, nor tomorrow’s. Hence, architecting has become a continuous practice just as coding, testing and deploying.
Embedded real time application:

An embedded system is a system dedicated to performing a specific task. Embedded systems can vary in size; from small things such as smart watches to large industrial “things” such as a self-driving cars, reference [13]. In general, an embedded system has a static set of resources, often purpose made. An embedded system does normally not depend on an external operator to perform its task. Its primary interface to its surrounding is seldom a keyboard or a monitor, even though keyboards and monitors are commonly used during configuration of the embedded system.

Embedded systems often have real-time properties. A real-time property defines requirements on the system that the system must address within specified time constraints. A real-time property does not have to be fast or short, but failure to meeting the requirement cannot be corrected later. E.g. consider harvesting a cherry tree. One need to harvest the cherries before the birds have consumed them all. Once consumed, one cannot harvest any cherries and the mission to harvest cherries have failed.

1.2 Previous research

A search on IEEE Xplore for “continuous architecture” yields only two results, reference [14] and [15]. Both papers are interesting reads, but they do not bring guidance to quality attributes of an architecture supporting continuous architecture. Erder and Pureur’s paper “What's the Architect's Role in an Agile, Cloud-Centric World?”, reference [14], address the role of the architect. The architect’s focus should be on the realized architecture, securing timely decisions and maintain the architecture runway. Mou and Ratiu’s paper “Binding requirements and component architecture by using model-based test-driven development”, reference [15], is a paper argue for Model-Based Test-Driven Development. Actually continuous architecture is not even mentioned in the paper, only in the papers metadata on IEEE Xplore.

Rephrasing the search string to “continuous architecting” yields two additional hits, reference [16] and [17]. The paper of Bersani et al. “Continuous Architecting of Stream-Based Systems”, reference [16], is about big data streaming designs by OSTIA, a toolkit to assist designers and developers to facilitate static analysis of the architecture and provide automated constraint verification in order to identify design anti-patterns and provide structural refactoring. The paper of Martini and Bosch, “A Multiple Case Study of Continuous Architecting in Large Agile Companies: Current Gaps and the CAFFEIA Framework”, reference [17], look into the gaps in the activities for conducting agile architecting. The researchers have developed an organizational framework, CAFFEIA, including roles, teams and practices supporting agile architecting. In addition, Martini and Bosch reflects over the lack of success stories or research on large scale agile organizations. They reference a paper by Dingsøyra et al., “A decade of agile methodologies:
Towards explaining agile software development”, reference [18]. This paper concludes that there are several challenges that still need to be addressed.

The conclusion is that there is not much prior research in the area “Continuous architecture for large scale agile organizations”, and the little research there is focus on architecting rather than architecture.

1.3 Problem formulation
The goal of this thesis project is to investigate architectural styles supporting continuous practices for large scale agile organizations. Additionally, the architectural styles need to be possible to implement into existing products without seriously impacting parallel addition of features.

The architectural style shall support embedded systems, potentially constituting of sets of processors and boards.

In the absence of substantial research related to continuous architecture, what architecture styles are used by the agile organizations developing web-scale applications? And are the applied architectural styles applicable also in the embedded domain?

An established strategy when developing web-scale applications is to go service oriented to decouple parallel agile teams, see reference [19] and [20]. The latest trend is to make the services really small and decoupled. This architecture style is called microservices, see reference [21]. Many small services minimize friction between teams. A message routing infrastructure aids with decoupling. Services are deployed individually, enabling automated unsynchronized integration and deployment, a cornerstone for continuous practices, see Humble and Farley’s book “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation”, reference [22].

1.4 Motivation
Many embedded systems, still being evolved, were put on market before agile practices became main stream. The architecture for those systems are seldom optimized for, or adapted to, agile way of working. Finding strategies to migrating legacy architectures to an architecture better supporting agile way of working could increase productivity in order of magnitudes. The challenge is to do this migration without stopping feature growth during the migration.

Microservices architecture has become norm for cloud based applications from agile companies like Google, eBay, and Amazon, see reference [21] and [23]; why isn’t it an established architecture also in the embedded domain? Microservices architecture looks promising, see the paper of Betz and Wohlin about “Alignment of Business, Architecture, Process, and Organisation in a Software Development Context”, reference [24]; but does it fit also embedded distributed [real-time] applications?
1.5 Research Question

<table>
<thead>
<tr>
<th>RQ 1</th>
<th>Does a microservices based architecture better support continuous practices compared to the currently applied component based architecture in the studied domain (an embedded real time application)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ 2</td>
<td>What migration steps are needed for the studied domain to migrate to a microservices based architecture?</td>
</tr>
<tr>
<td>RQ 3</td>
<td>Are the findings from the studied domain generally applicable for embedded real time applications?</td>
</tr>
</tbody>
</table>

Table 1.1 Research questions

<table>
<thead>
<tr>
<th>RH 1</th>
<th>Microservices are expected to better support continuous practices since a service is focused on only one thing, and hence is less frequent affected by parallel changes and updates. Instead of frequent changes to components, new services are added to the system, either replacing or complementing old ones. However, infrastructure is expected to be needed to manage the complexity of handling relations and interactions between services. In addition, timing may be impacted.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RH 2</td>
<td>The application domain is well prepared since it already uses networked communication, however having independent upgrade domains put new requirements on the dependencies between components.</td>
</tr>
<tr>
<td>RH 3</td>
<td>Microservices are assumed to be a useful architectural pattern also outside native cloud deployments, however real-time applications with very short and strict real-time requirements may not be able to use microservices due to the added delay and variation introduced by networked communication.</td>
</tr>
</tbody>
</table>

Table 1.2 Research hypothesis
1.6 Scope/Limitation
The scope “Architectural styles supporting continuous architecture in a large scale agile architecture” is close to a limitless scope:

- There is not one style supporting all applications domains, each domain has its unique set of quality attributes that must be fulfilled.
- Even if the domain is set, there is a vast number of architectural styles supporting a domain. Going through them all is a task of a lifetime, and constantly new styles appear.
- All agile practices and organizations are not the same. Both BAPO, reference [3], and Conway’s law, reference [2], indicate organizations and processes impact the architecture.

To limit the scope, the studied domain is limited to embedded real-time applications; and the studied architectural styles are microservices based, chosen since it seems to be a common style by pioneering large scale agile organizations.

1.7 Target group
Despite the scoping of the study to embedded real-time applications, the findings from the study ought to be valid also for other domains where products have been designed before the paradigm of agile and continuous practices. Hence the target group for this thesis is anyone responsible for a software architecture with a long-lived legacy system still being developed.

1.8 Outline
The rest of the report is structured as follows:

- Chapter 2 describes the methods used.
- Chapter 3 gives an introduction to Microservices.
- Chapter 4 gives a bit more in-depth description of the constraints in the application domain of the equipment and resource management application in a radio base station.
- Chapter 5 contains the reasoning behind three other products positioning regarding microservices architecture.
- Chapter 6 contains the analysis based on literature and the interviewed organizations whether microservices is a valid architectural style for the equipment and resource management application of a radio base stations.
- Chapter 7 contains a discussion of the validity of the findings.
- Chapter 8 contains the conclusion.
- Chapter 9 contains references to used literature.
- Chapter 10 is the appendix with (filtered) transcriptions of the conducted interviews and short summaries of the included papers in the literature part.
2 Method

For this thesis, two research methods were used:

- For the literature study quantitative search was used, systematically searching research paper repositories for the current state of the art.
- The quantitative search was complemented with qualitative interviews researching applied industrial strategies and the validity of the architectural styles identified in the literature study.

2.1 Method description, quantitative search:

A systematic quantitative study is an evidence-based secondary study using a systematic, well-defined procedure. This type of study has the advantages of providing a comprehensive overview of the state of the art on the investigated research topic.

The study was conducted in three steps.

1. The first step was the planning step which yields the research questions to be answered, the search strings to be used, and the criteria for selecting primary studies.
2. The second step was the execution step, where the primary studies were identified, selected, and evaluated.
3. Finally, the analysis step aggregates the information extracted from the relevant primary studies considering the research questions.

Research Questions

Based on the research questions (RQs) outlined in Table 1.1, the search questions were compiled. First basic search of IEEE Xplore with search term: microservice AND embedded AND “continuous architecture” yields 25,521 hits, see Figure 2.1.
The filtering step indicates the search does not only return documents where all phrases are included. A strong indication of this is that search phrase “continuous architecture” “only” yields 4,971 hits, see Figure 2.2.

![IEEE Xplore Digital Library](image)

*Figure 2.2 Second search*

Elaborating on search strings indicates that the search not only return hits with the exact phrase “continuous architecture”, but any hit with the two words in the text. An evidence of this is that a search without quotation marks yields the same result, see Figure 2.3.

![IEEE Xplore Digital Library](image)

*Figure 2.3 Third search*

For an exact phrase search, the quotation marks must be straight and not curved. The search string "continuous architecture" yields two hits, reference [14] and [15], see Figure 2.4.
Search Strategy
To retrieve primary studies, the search process was executed in the IEEE Xplore database. The IEEE Xplore databases was selected due to (i) the good coverage of research paper in the electronic database, (ii) the regularity of updates, (iii) the availability of the full text of the studies, (iv) the assumed easiness of performing the search, (v) the accuracy of the returned results, and (vi) access rights to the databases.

The basic search for prior research that yielded four hits only searched Metadata. Repeating the search as an advance full text search using the search string ("continuous architecture") OR ("continuous architecting") yields 36 hits of which 33 were accessible, see chapter 10.1. Despite the additional number of papers, only two of the additional papers, Groher and Weinreich’s
both papers “Integrating Variability Management and Software Architecture”, reference [25], and “Supporting Variability Management in Architecture Design and Implementation”, reference [26], add relevant aspects into continuous architecture. These two papers present a tool, LISA, to support architecture variability management. But unfortunately not architecture evolution.

To get more relevant research data, the literature search needed to be extended. The experience from Amazon, Google, Ebay, Netflix etcetera, see reference [19], [20], and [21], indicates large successful organizations develop web scale applications using microservices. They do use agile methods as a strategy for increase speed in their development. Thus it is interesting to study what aspects of microservices architectures that provides improved support for continuously maintaining the architecture. It is further interesting to study if these aspects are applicable also for embedded systems. A search for microservice* OR micro-service* OR "micro service"* yields 129 hits (2017-03-24).

**Selection Criteria**

Selection criteria was used to evaluate retrieved primary study considering the defined research questions, see Table 2.1. The main goal was to include studies that would be potentially relevant to answer the research questions and to exclude the ones that would not contribute to answering them, see Figure 2.5.

<table>
<thead>
<tr>
<th>The following inclusion criteria was used:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IC1:</strong></td>
<td>The term microservice* or micro-service* or &quot;micro service&quot;* should be present in the paper Metadata.</td>
</tr>
<tr>
<td><strong>IC2:</strong></td>
<td>The term architect* should be present in title or abstract</td>
</tr>
<tr>
<td><strong>IC3:</strong></td>
<td>The introduction or conclusion in the paper should reference architecture properties</td>
</tr>
</tbody>
</table>

*Table 2.1 Inclusion criteria*
2.2 Method description, qualitative interviews

The interviews were conducted in multiple steps. For each interview:

- The interview was conducted with responsible architects for a product. The interview was recorded.
- The recorded interview was transcribed.
- The transcribed interview was translated and filtered.
- The respondents reviewed and confirmed both the transcription and the filtered translation.
- Key architectural properties for the application domain was identified.
- The properties were compared with the properties of a micro-service application.
- The validity of micro service architecture in the domain was validated with the responsible architects for the product (same architects as in the first interview).

In addition to the architecture related papers, some papers elaborated in container technologies as light weight alternative to full virtualization. Potentially interesting for embedded systems.
The respondents were deliberately selected to cover as large area of experience and expertise as possible. The rationales for the selection were these:

- All respondents architect applications with real-time properties.
- All respondents architect applications being developed using agile practices.
- All respondents’ architectures have been monolithic.
- One respondent architects a similar product, and with similar market positioning.
  - But with a smaller organization.
  - The architecture has not been migrated to a microservices architecture.
- One respondent architects a product within a similar organization, but with a different market segment.
  - The application has migrated to a microservices architecture.
- One respondent architects a product with a different organization, and within a different market segment.
  - The application has migrated to a microservices architecture.

2.3 Reliability and validity

The reliability of the research is expected to be high.

The literature part can be repeated by anybody with access to IEEE Xplore, a well renowned source of published research papers. A limitation to the validity of the literature study is the exclusion of other relevant sources such as ACM Digital Library.

The reliability of the qualitative interview part is harder to prove. In order to protect Ericsson intellectual properties, product specific details in the interviews cannot be published, and hence the published parts are filtered. It is further hard to repeat the interviews for an external researcher. The interviews were conducted within Ericsson by a known Ericsson employee, and hence no need to arrange with non-disclosure agreements since both parties are bound by same rules of conduct. There are no reasons to believe information was withheld during the interviews, since the interviews doesn’t contain any personal information risking position the responder in a troublesome situation. The risk lays in the filtering for publication; that relevant parts of the material are filtered out. However, the filtering was validated by responsible architects, both to secure important aspects for the application domain was still captured, as well as to avoid leakage of sensible intellectual properties.

A validity risk is that the studied domain is too narrow to guide a broader area of applications.
2.4 Ethical Considerations

There is no personal data in the collected dataset, still data is filtered for confidentiality reasons. Hence there is no ethical constraints in publishing the resulting findings.
3 Microservices

There is no precise definition of what a microservices architecture is. According to Wikipedia, reference [27], microservices are “an architectural style that structures an application as a collection of loosely coupled services”. Furthermore, “the benefit of decomposing an application into different smaller services is that it improves modularity and makes the application easier to understand, develop and test. It also parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently. It also allows the architecture of an individual service to emerge through continuous refactoring. The microservices architecture enables continuous delivery and deployment.” In the paper “Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption”, reference [28], Di Francesco et al. conclude most academic papers tend to lean towards the microservice definition provided by James Lewis and Martin Fowler, “an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies”, see reference [29].

Lewis and Fowlers definition was followed by the definition provided by Sam Newman in the book “Building Microservices”: “Microservices are small, autonomous services that work together.”, reference [30]. Both definition in reference [29] and [30] share most characteristics of what microservices are; and what characterize microservices and microservices based systems, see Table 3.1.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microservices are small.</td>
<td></td>
</tr>
<tr>
<td>Focused on doing one thing well.</td>
<td>Organized around business capabilities.</td>
</tr>
<tr>
<td>Run in its own process.</td>
<td>They potentially containing their own operating system instances and run on its own machines.</td>
</tr>
<tr>
<td>Independently deployable.</td>
<td>They can be independently deployed on their own machines.</td>
</tr>
</tbody>
</table>
| Microservices communicate using lightweight networked calls. | }
<table>
<thead>
<tr>
<th>Aspect</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microservices expose their services over application programming interface.</td>
<td>Service realization details are not exposed to service users.</td>
</tr>
<tr>
<td>Microservices can be realized with different technologies for each service.</td>
<td>Also known as polyglot.</td>
</tr>
<tr>
<td>Microservices can enhance resilience.</td>
<td>The system does not consist of a single monolithic application. Hence no “single point of failure”.</td>
</tr>
<tr>
<td>Microservices supports runtime scaling.</td>
<td>New instances of a loaded services can be spawned when needed.</td>
</tr>
<tr>
<td>Microservices support ease of deployment.</td>
<td>No need to coordinate deployment of multiple services.</td>
</tr>
<tr>
<td>Microservices scales with organization.</td>
<td>Due to the small size of the microservice, small teams can manage to implement a new service in days or weeks. Furthermore, due to the autonomy of microservices, many teams can work in parallel on different services without need for synchronization.</td>
</tr>
<tr>
<td>Microservices supports evolutionally architecture.</td>
<td>Additional microservices can be developed and deployed when needed.</td>
</tr>
<tr>
<td>Microservices need to be designed for failure.</td>
<td>Due to the distributed fashion of a microservice based system, access to other services may fail in any moment without prior notice.</td>
</tr>
<tr>
<td>Microservices need to consider additional time for distributed communication.</td>
<td>Since all interaction between services are networked, communication time is prolonged.</td>
</tr>
<tr>
<td>Microservices are distributed. Distribution adds complexity.</td>
<td>For low complex applications, the additional complexity of microservices may not be worth the price.</td>
</tr>
</tbody>
</table>
### Good practices

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apply an automated integration and delivery stream.</td>
<td></td>
</tr>
<tr>
<td>Put the intelligence in the microservices, avoid application logic in the communication infrastructure.</td>
<td>“Smart endpoints, dumb pipes”.</td>
</tr>
<tr>
<td>Decentralize data management.</td>
<td>Avoid unintentional relations across microservices introduced by centralized data.</td>
</tr>
<tr>
<td>Decentralize governance.</td>
<td>Centralized government tends to drive towards single standardized technology platforms.</td>
</tr>
<tr>
<td>Let the microservices follow the organization; consider Conway’s law, reference [2].</td>
<td></td>
</tr>
<tr>
<td>If refactoring legacy code into microservices, look for seams in the code.</td>
<td>See Michael Feathers’ book “Working Effectively with Legacy Code”, reference [32].</td>
</tr>
<tr>
<td>Make the services state-less.</td>
<td>Significantly enhance support for scalability since state data do not need to be synchronized across service instances.</td>
</tr>
</tbody>
</table>

Table 3.1 Microservice characteristics

Above indicates microservices are a means for improving efficiency in large scale agile organizations since it enables parallel development. It as well indicates microservices may enable continuous architecture.
4 Architectural concerns for the equipment and resource management application in an Ericsson radio base station

A mobile cellular network consists of many functions, see Figure 4.1. With the development of 3G, prior separate standardization organs came together and created the 3rd Generation Partnership Project (3GPP), reference [33], which since drives standardization forward. 3GPP is organized in three technical specification groups (TSG), where the Radio Base Station (RBS) specification is handled in the Radio Access Network group (RAN). The role of an RBS is to be a modem. Its task is to convert user data from e.g. internet to modulated Radio Frequency (RF) data to the mobile device (called downlink direction in the standardization), and vice versa in the direction from the mobile (called uplink direction in the standardization). A mobile device is called UE or User Equipment. Depending of generation, the RBS has different name; a 2G RBS is called BTS, Base Transceiver Station. A 3G RBS is called NodeB and a 4G RBS (Long Term Evolution or LTE) is called eNodeB (evolved NodeB).

A modern RBS can run both 2G, 3G, and 4G in parallel and hence realize the roles of both BTS, NodeB and eNodeB.

Figure 4.1 Overview of functions in mobile networks, source Spirent.com
An RBS today is normally built from two RBS specific units, a digital unit and a radio unit. In addition, there exist a wide range of supporting units. The digital unit is responsible for the base band processing function, which stack is illustrated in Figure 4.2. The radio unit contains digital to analog, and analog to digital conversion blocks, oscillator, mixer and power amplifier. The RF energy is fed to a built in or external antenna.

![Protocol stack in an LTE base station, source http://lteworld.org/lte-protocols-specifications](http://lteworld.org/lte-protocols-specifications)

**Radio Resource Control (RRC)**
The main services and functions of the RRC sublayer include:
- Broadcast of System Information related to the non-access stratum (NAS)
- Broadcast of System Information related to the access stratum (AS)
- Paging
- Establishment, maintenance and release of an RRC connection between the UE and E-UTRAN
- Security functions including key management
- Establishment, configuration, maintenance and release of point to point Radio Bearers
- Mobility functions
- Quality of Service management functions
- UE measurement reporting and control of the reporting
- NAS direct message transfer to/from NAS from/to UE

**Packet Data Convergence Protocol (PDCP)**
The main services and functions of the PDCP sublayer for the user plane include:
- Header compression and decompression: Robust Header Compression (ROHC) only
- Transfer of user data
- In-sequence delivery of upper layer Protocol Data Units (PDUs) at PDCP re-establishment procedure for RLC Acknowledge Mode (AM)
• Duplicate detection of lower layer Service Data Units (SDUs) at PDCP re-establishment procedure for RLC AM
• Retransmission of PDCP SDUs at handover for RLC AM
• Ciphering and deciphering
• Timer-based SDU discard in uplink

The main services and functions of the PDCP for the control plane include:
• Ciphering and Integrity Protection
• Transfer of control plane data

**Radio Link Control (RLC)**
The main services and functions of the RLC sublayer include:
• Transfer of upper layer PDUs
• Error Correction through automatic repeat request (ARQ) (only for AM data transfer)
• Concatenation, segmentation and reassembly of RLC SDUs (only for Unacknowledged Mode (UM) and AM data transfer)
• Re-segmentation of RLC data PDUs (only for AM data transfer)
• In sequence delivery of upper layer PDUs (only for UM and AM data transfer)
• Duplicate detection (only for UM and AM data transfer)
• Protocol error detection and recovery
• RLC SDU discard (only for UM and AM data transfer)
• RLC re-establishment

**Medium Access Control (MAC)**
The main services and functions of the MAC sublayer include:
• Mapping between logical channels and transport channels
• Multiplexing/demultiplexing of MAC SDUs belonging to one or different logical channels into/from transport blocks (TB) delivered to/from the physical layer on transport channels
• scheduling information reporting
• Error correction through Hybrid ARQ (HARQ)
• Priority handling between logical channels of one UE
• Priority handling between UEs by means of dynamic scheduling
• Transport format selection
• Padding

**L1 - Air Interface Physical Layer**
The LTE air interface physical layer offers data transport services to higher layers. The access to these services is through the use of a transport channel via the Medium Access Control (MAC) sub-layer. The physical layer is expected to perform the following functions in order to provide the data transport service:
• Error detection on the transport channel and indication to higher layers
- Forward Error Correction (FEC) encoding/decoding of the transport channel
- HARQ soft-combining
- Rate matching of the coded transport channel to physical channels
- Mapping of the coded transport channel onto physical channels
- Power weighting of physical channels
- Modulation and demodulation of physical channels
- Frequency and time synchronization
- Radio characteristics measurements and indication to higher layers
- Multiple Input Multiple Output (MIMO) antenna processing
- Transmit Diversity (TX diversity)
- Beamforming
- RF processing

The digital unit (DU) communicates with the radio unit (RU) over a standardized interface, Common Public Radio Interface (CPRI), [34], where the user data is transported in Antenna Carrier slots (AxC) in an In-Phase and Quadrature (IQ) modulate format, Figure 4.3, see reference [35] for a good overview of CPRI.

![Figure 4.3 Conceptual explanation of REC/RE functional split](image)

Mobile telecom networks are sometimes referred to as cellular networks, from the way they are built. The cellular concept is potentially subject for change for up-coming 5G standard, but for current standards it is used. In 3GPP a cell is defined as “Radio network object that can be uniquely identified by a User Equipment from a (cell) identification that is broadcasted over a geographical area from one UTRAN Access Point. A cell is either in frequency division duplex (FDD) or time division duplex (TDD) mode.” A term related to cell is sector. In 3GPP a sector is defined as: “A "sector" is a sub-area of a cell. All sectors within one cell are served by the same base station. A radio link within a sector can be identified by a single logical identification belonging to that sector”. A common way of building a network is to have a base station serving three cells, where the radio is located in one corner of the cell, as is the case for e.g. the green cells in Figure 4.4. Then the radio serves three sectors with one cell in each sector. One can also put the radio in the middle of the cell and create an omni-directional cell, as illustrated with the
light red cell in Figure 4.4. As stated in the definition of cell in 3GPP, the cell shall be identified by the UE. Therefore each cell has a unique cell identifier. This identifier is sent over a radio carrier. A radio can support multiple carriers in one sector, each providing a separate cell identifier, and hence multiple cells can cover the same geographical area, as illustrated in Figure 4.5. A radio unit normally support many carriers, which can serve one or many cells. The term used when multiple carriers support the same cell is carrier aggregation. The most common scenario is that two or four carriers serves one cell; the trend is increased carriers per cell to provide higher maximum bit rate. A digital unit normally supports many radios. How many depends on generation of the unit (basically how much calculation capacity the unit have) and the characteristics of the served cells, that is how wide bandwidth each carrier has. The radios can be connected directly to the digital unit, or connected in a cascade chain, or connected via a switch as illustrated in Figure 4.6 (the XMU is the switch). One base station consists of all these units, plus antenna units, antenna near units, power related unit, climate related unit plus some more.

Figure 4.4 Simplified cell view

Figure 4.5 Relation between Cell, Sector, and Carrier
The responsibility for the equipment management functionality in the base station is to configure and maintain all equipment in the base station operational. Everything related to equipment, sector and carrier handling is common for 2G, 3G, and 4G Radio Access Technology (RAT) standards, what differs is the cell handling. Therefore, the resource management responsibility is to provide an abstracted view of all equipment that is used to provide a carrier and provide logical carriers for the different RATs cells. Multiple RATs can share the same sector resources, but they use separate carriers.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Supported by Microservices</th>
<th>Addressed by the selected papers in the literature study</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>Not natively supported, solutions exist.</td>
<td>Trust: [36], [37], [38], [39]</td>
</tr>
<tr>
<td>A running application must not be possible to illegally manipulate.</td>
<td>Not natively supported, solutions exist</td>
<td>Not addressed in any paper.</td>
</tr>
<tr>
<td>Interaction within and between applications must be protected against bugging.</td>
<td>HTTPS frequently used, other encryption to be applied if using a message bus.</td>
<td>HTTPS or encryption: [40], [41], [42]</td>
</tr>
<tr>
<td>Cell availability shall be at least 99.999%.</td>
<td>Microservices architectures are designed to be robust to changes in available services. Redundancy</td>
<td>Robust: [43], [37], [44], [45], [42]</td>
</tr>
<tr>
<td>Aspect</td>
<td>Supported by Microservices</td>
<td>Addressed by the selected papers in the literature study</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The system shall support software upgrade without traffic disturbance.</td>
<td>Microservices architecture support dynamic adding and removing of services. Stateless servers ease upgrade.</td>
<td>Dynamic: [43], [36], [37], [46], [40], [47], [48], [49], [50]</td>
</tr>
<tr>
<td>The system must not waste system resources such as memory, CPU cycles, or energy</td>
<td>Microservices are by default virtualized. In principal this is not a requirement, but it eases independent deployment. Virtualization waste system resources. Container techniques can be applied to minimize resource leakage. Needs further studies how to handle bare metal applications.</td>
<td>System resource: [46]</td>
</tr>
<tr>
<td><strong>Robustness</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One faulty part must not compromise the complete system availability</td>
<td>Microservices are by design robust against faults. Circuit breakers can be built in to protect applications</td>
<td>Resilience [37], [51]</td>
</tr>
<tr>
<td>When a fault occurs, it shall be easy to identify and correct the fault</td>
<td>A logging service needs to be used</td>
<td>Logging: [52], [51], [48]</td>
</tr>
<tr>
<td>It shall be possible to replace a unit without shutting down the rest of the system</td>
<td>Microservices are by design robust towards services coming and going. If alternative units are available, a load balancer can</td>
<td>Replace: [53], [43], [46], [47], [51], [38], [48], [54]</td>
</tr>
<tr>
<td>Aspect</td>
<td>Supported by Microservices</td>
<td>Addressed by the selected papers in the literature study</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------</td>
</tr>
<tr>
<td>Scalability</td>
<td>move services to a working unit.</td>
<td></td>
</tr>
<tr>
<td>It shall be possible to take into use new hardware and new configurations without restarting the system</td>
<td>Microservices are by design robust towards services coming and going.</td>
<td>Replace: [53], [43], [46], [47], [51], [38], [48], [54]</td>
</tr>
<tr>
<td>It shall be possible to change interconnect routing in runtime</td>
<td>Moving services are part of a load balancer and natively supported.</td>
<td>Load balancing: [43], [36], [37], [46], [44], [51], [38], [39], [41]</td>
</tr>
<tr>
<td>The system shall support deployment scaling from a system on chip to a system constituting of hundreds of units</td>
<td>Individual microservices do not natively support heterogeneous execution environment. A microservice may be unique inside, but it expects homogeneous hosts, and different microservices may execute on separate types of execution environment.</td>
<td>Heterogeneous: [53], [43], [37], [51], [41], [55]</td>
</tr>
<tr>
<td>The system shall support partial deployment in cloud</td>
<td>Same issue as above, an individual microservice does not natively support heterogeneous execution environments.</td>
<td>Heterogeneous: [53], [43], [37], [51], [41], [55]</td>
</tr>
<tr>
<td>The system shall support integration with already deployed legacy hardware</td>
<td>As long as the hardware can support integration with the service discovery functions and communication</td>
<td>Legacy: [52], [36], [37], [40], [44], [38], [56], [49], [57], [55]</td>
</tr>
<tr>
<td>Aspect</td>
<td>Supported by Micro-services</td>
<td>Addressed by the selected papers in the literature study</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>Methods legacy hardware and services can be integrated with new hardware and services.</td>
<td>The architecture shall support multiple application system built from the same source system (also known as software product lines)</td>
<td>If using virtualization techniques this is not necessary. But if not, as long as the service can interact with service discovery and messaging functions, same source code can be used for different target builds.</td>
</tr>
<tr>
<td>Not addressed in any paper</td>
<td>The architecture shall support many teams across many sites working in parallel with no or minimal synchronization</td>
<td>If done right, microservices scale well with organization size, and cross team synchronization can be kept to a minimum.</td>
</tr>
<tr>
<td>Development team: [53], [52], [37], [46], [44], [45], [56], [49], [57], [50]</td>
<td>The architecture shall support teams with varying experience and domain knowledge</td>
<td>Microservices are a good fit for less experienced teams since they cover a smaller domain. However, boiler plate code is a good support for the added complexity of distribution. An automated delivery machine shall aid integration and deployment easy the technology domains the teams need to cover.</td>
</tr>
<tr>
<td>Domain knowledge: [51]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aspect</td>
<td>Supported by Microservices</td>
<td>Addressed by the selected papers in the literature study</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------</td>
<td>-------------------------------------------------------------</td>
</tr>
<tr>
<td>The architecture shall support test-driven development</td>
<td>A good fit, since microservices expose their services over well-defined APIs.</td>
<td>Test/Behavior-driven development: [37]</td>
</tr>
<tr>
<td>The architecture shall support fast feedback loops</td>
<td>The limited size of a microservice and its narrow scope makes it fast to integrate given a good continuous integration framework is supplied.</td>
<td>Feedback: [52], [39], [57], [50]</td>
</tr>
<tr>
<td>The architecture shall support continuous integration, delivery, and deployment</td>
<td>Microservices requires a CI/CD machinery.</td>
<td>Continuous integration: [53], [52], [46], [44], [51], [57], [50]</td>
</tr>
</tbody>
</table>

*Table 4.1: Architectural properties (a subset) for the equipment and resource management application*
5 Rationales for interviewed organization to go or not go service oriented

Of the three interviewed organizations, organization one has not gone service oriented, organization three has gone service-oriented and organization two has applied microservices “by the book”; see chapter 10 for transcriptions of the interviews. Each organization has a deliberate architecture which they all claim is very close to what they would have if they had the chance to restart from a blanc paper. A mapping of the characteristics of microservices to each of the organizations [expressed] needs is illustrated in Table 5.1.

One can notice a few things. Organization one and three claims autonomous delivery and deployment is not crucial, while for organization two it is. A notable difference is that organization two owns the deployment. Organization one and three does not own the deployments, their customers are individually responsible for the installations. Since organization two’s application is cloud based, deployment is fairly easy, organization one and three’s applications are installed in multiple instances which makes installations and upgrades a bit more cumbersome. Organization one’s application is distributed to more than a million installations, all at remote destinations costly to visit in case an upgrade fails. It is not clear if organization one and three down prioritize autonomous delivery and deployment due to lack of need, or because they consider it too costly to introduce.

Another noticeable difference between organization two on one hand and one and three on the other is scalability. Organization one and three are fairly large organizations, while organization two is a small one. The large organizations emphasize development efficiency over runtime efficiency. Not that runtime efficiency is not important, but organizational efficiency is a tougher puzzle to solve.

A third noticeable difference: organization one has selected not to go service oriented, while organization two and three has. If one looks on what parts of the system that has highest frequency of change, organization one is most impacted in driver and infrastructure layer, not in the business logic layer. The two organizations that have gone service oriented have most changes in the business domain layers.

A fourth noticeable difference, organization one and three are not overly concerned about the additional complexity introduced with networked applications. This is not because they neglect the aspect, but their applications were networked long before the term microservice was introduced, and hence is not an attribute they relate to microservices but to networked applications, which are a well-known domain for both organizations.
<table>
<thead>
<tr>
<th>Aspect</th>
<th>Organization one</th>
<th>Organization two</th>
<th>Organization three</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microservices are small.</td>
<td>Important</td>
<td>Important</td>
<td>Important</td>
</tr>
<tr>
<td>Focused on doing one thing well.</td>
<td>Important</td>
<td>Important</td>
<td>Important</td>
</tr>
<tr>
<td>Run in its own process.</td>
<td>Important</td>
<td>Important</td>
<td>Important</td>
</tr>
<tr>
<td>Independently deployable.</td>
<td>Important</td>
<td>Important</td>
<td>Important</td>
</tr>
<tr>
<td>Microservices communicate using lightweight networked calls.</td>
<td>Important</td>
<td>Important</td>
<td>Important</td>
</tr>
<tr>
<td>Microservices expose their services over application programming interface.</td>
<td>Important</td>
<td>Important</td>
<td>Important</td>
</tr>
<tr>
<td>Microservices can be realized with different technologies for each service.</td>
<td>Not important</td>
<td>Important</td>
<td>Not important</td>
</tr>
<tr>
<td>Microservices can enhance resilience.</td>
<td>Important</td>
<td>Important</td>
<td>Important</td>
</tr>
<tr>
<td>Microservices supports runtime scaling.</td>
<td>Not important</td>
<td>Important</td>
<td>Partly important</td>
</tr>
<tr>
<td>Microservices support ease of deployment.</td>
<td>Not important</td>
<td>Important</td>
<td>Not important</td>
</tr>
<tr>
<td>Microservices scales with organization.</td>
<td>Very important</td>
<td>Not important</td>
<td>Very important</td>
</tr>
<tr>
<td>Microservices supports evolutionally architecture.</td>
<td>Partly important</td>
<td>Very important</td>
<td>Important</td>
</tr>
<tr>
<td>Microservices need to be designed for failure.</td>
<td>Not an issue, existing property also</td>
<td>An issue</td>
<td>Not an issue, existing property also</td>
</tr>
<tr>
<td>Aspect</td>
<td>Organization one</td>
<td>Organization two</td>
<td>Organization three</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------------------</td>
<td>------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Microservice consequences</td>
<td>without microservices</td>
<td>An issue</td>
<td>Not an issue, existing property also without microservices</td>
</tr>
<tr>
<td>Microservices need to consider additional time for distributed communication.</td>
<td>Not an issue, existing property also without microservices</td>
<td>An issue</td>
<td>Not an issue, existing property also without microservices</td>
</tr>
<tr>
<td>Microservices are distributed. Distribution adds complexity.</td>
<td>Not an issue, existing property also without microservices</td>
<td>Not an issue, covered by selected 3PP framework</td>
<td>Not an issue, existing property also without microservices</td>
</tr>
<tr>
<td>Good practices</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apply an automated integration and delivery stream.</td>
<td>In place</td>
<td>In place</td>
<td>In place</td>
</tr>
<tr>
<td>Put the intelligence in the microservices, avoid application logic in the communication infrastructure.</td>
<td>In place</td>
<td>In place</td>
<td>In place</td>
</tr>
<tr>
<td>Decentralize data management.</td>
<td>Not in place</td>
<td>In place</td>
<td>Not in place</td>
</tr>
<tr>
<td>Decentralize governance.</td>
<td>Not in place</td>
<td>In place</td>
<td>Not in place</td>
</tr>
<tr>
<td>Let the microservices follow the organization, consider Conway’s law.</td>
<td>In place</td>
<td>In place</td>
<td>In place</td>
</tr>
<tr>
<td>Let the microservices follow natural product domains or “bounded contexts”, consider Domain-Driven Design.</td>
<td>In place</td>
<td>In place</td>
<td>In place</td>
</tr>
<tr>
<td>If refactoring legacy code into microservices, look for seams in the code.</td>
<td>Applied</td>
<td>Applied</td>
<td>Applied</td>
</tr>
<tr>
<td>Make the services stateless.</td>
<td>Not in place</td>
<td>In place</td>
<td>Not in place</td>
</tr>
</tbody>
</table>

*Table 5.1 Organisational view on microservices*
6 Analysis

Microservices support many of the architectural properties stated important for the equipment and resource management application in a radio base station. Specifically, the property of services evolving independently making it a good fit for supporting continuous architecting, see [30], [50], [51], [56], and [64]. The traditional objections raised for microservices, that it brings added complexity due to being a networked distributed architecture, is not a hinder since this is already the case in the current architecture.

In literature a cloud based infrastructure is considered norm when considering a microservices architecture. The reasons for that is that the community where microservices were born was the community of web based enterprise applications where cloud is norm. For that reasons a lot of supporting tools, like containers, has evolved making this step easier. If developing a microservices based architecture outside this natively supported deployment domain, it becomes trickier. In addition, most literature considers one or few instances of a system deployed in a potentially distributed cloud. And that the developing organization is the one responsible for deploying the application. This is not the case for radio base stations. For radio base stations it is the customer that owns the hardware and controls when software is deployed. Furthermore, each customer has thousand or tens of thousands of RBSs, each separately managed. Furthermore, Ericsson has hundreds of customers. Hence, it is not in the developer’s hand to decide when and what to deploy, and definitively not part of a constant delivery stream. With that said, Ericsson have Continuous Delivery and Deployment customers who every second week deploy latest stable version of the software on a subset of their live nodes.

The domain of a radio base station constitutes many different functions, some of them very real-time critical (order of nanoseconds), and some with more relaxed real time performance requirements. It is harder to guarantee very strict real-time performances in a networked application, why microservices may not be an option for the complete RBS application (e.g. areas that today run on bare metal to avoid the performance impact from an operating system). Other areas are better suited. The equipment and resource management application is such an area where real-time requirements are relaxed and where a microservices architecture may be a good fit. As indicated in the literature study, the most common way to perform “microservitization” is to start with an existing monolithic application and modularize it into microservices. A modularization journey has been ongoing for a while in the equipment and resource management application, but properties like independent deployability has not been addressed. So forth “only” independent continuous integration is in place. Another proposed property of microservices architecture is to make the services state less. The rational for that is to improve robustness, improve support for scalability, and independent deployability and
upgradeability. The current application has not suffered from any scalability issues. And since independent deployment is not (yet) supported, the drive for stateless components has not been prioritized. To improve robustness and support independent deployability, this needs to be addressed, which is a major refactoring take-on. Stateless servers are good for other reasons as well. For example, code gets better structured, making it easier to maintain; a proposed style in e.g. the DCI pattern, reference [58].

The general security requirements for the RBS is less supported in the microservices community. However, there are activities addressing it, like the one presented in journal article "Building Critical Applications Using Microservices", reference [38], and the conference paper “Security-as-a-Service for Microservices-Based Cloud Applications”, reference [40]. These works assume a cloud, or at least a virtualized execution environment. Perhaps it is a strategy to consider some kind of virtualization technique also for the equipment and resource management application. With virtualization comes also the benefit of supporting deployments in heterogeneous environments. Good also for testing, when tests then more easily can be executed in host environments. What then needs to be looked into is how to create an as resource efficient “cloud architecture” as possible, but that is a topic for another study.

Over all, a microservices based architecture seems to meet the architectural requirements for the equipment and resource management application well, but it needs to be developed stepwise. Potentially by breaking apart some large components into a microservices architecture internally in the component, and then take it from there.

6.1 Literature: Microservices architecture related papers

Not all papers define what a microservice or microservices architecture is, and when they do, the definitions differ slightly:

Journal article “Microservices in practice”, reference [53], reference Lewis and Fowlers definition “, reference [29] and then definition provided in the book “Microservice Architecture” by Amundsen et al., reference [59]. Amundsen in his turn reference Sam Newman’s definition “Microservices are small, autonomous services that work together”, and Adrian Cockcroft’s definition: “Service-oriented architecture composed of loosely coupled elements that have bounded contexts.”. In addition, they provide an own definition: “A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices.”

Journal article “Microservices”, reference [60], states “When you ask N people to define microservices or what the typical size of a microservice is, you’ll likely get N + M different definitions.” And then Yousif provides
following definition: “They’re programs with a single task (or unit of work) that also include all the connectivity to the outside world as well as the runtime requirements to run the task. (Note that the word “task” is generic and refers to the smallest function possible, but no smaller.)”

Conference paper “A Microservice Based Reference Architecture Model in the Context of Enterprise Architecture”, reference [49], defines microservices as: “A Microservice is an application on its own to perform the functions required. It evolves independently and can choose its own architecture, technology, platform, and can be managed, deployed and scaled independently with its own release lifecycle and development methodology.” And “A Microservice based architecture is defined as a "software architecture pattern" for development of distributed applications, where the application is comprised of a number of smaller "independent" components; these components are small application in themselves.”. The definition is based on Namiot and Sneps-Sneppe’s article “On Microservices Architecture” in International Journal of Open Information Technologies, reference [61], albeit this article rather describes than defines what a microservices architecture is.

Journal article “The Design and Architecture of Microservices” reference [42], does not provide any definition but reference “NIST Definition of Microservices, Application Containers and System Virtual Machines”, reference [62]. NIST defines microservices as: “A microservice is a basic element that results from the architectural decomposition of an application’s components into loosely coupled patterns consisting of self-contained services that communicate with each other using a standard communications protocol and a set of well-defined APIs, independent of any vendor, product or technology.”

Conference paper “Microservices and Their Design Trade-offs: A Self-Adaptive Roadmap”, reference [47], concludes “Despite the hype for microservitization, the state of the art still lacks consensus on the definition of microservices, their properties and their modelling techniques.” Based on informal sources they have tried to identify commonalities in different definitions and came up with this definition. Microservices are “autonomic, replaceable and deployable artefacts of microservitization that encapsulate fine-grained business functionalities presented to system users through standardized interfaces. The autonomy of these artefacts allows for governing them in a decentralized manner and tracing their changes.”. They base this definition on the Lewis’ definition, reference [29]; Sader’s definition, reference [63]; and Newman’s definition, reference [30].

6.1.1 Proposed target domains

Microservices has become norm for cloud based applications when scalability is crucial. It is best fit for complex problems since it comes with a cost. As Singleton state it in reference [44], “you should consider using a cloud-based
microservices architecture if you’re dealing with any of the following types of complexity:

- Large software systems with large numbers of developers or long and expensive test cycles
- A competitive environment that requires the rapid upgrading and release of online systems or business services
- Multiple software-based products or online services
- Migration from building and maintaining systems to buying more components that will be continuously upgraded by vendors
- Integration with systems on different platforms
- High volume of usage on cloud-based platforms
- Large flow of data, or rapidly changing data structures

There is no domain in above list, but there is a distinct phrasing in the recommendation, “consider cloud-based microservices architecture”. But what if your application is not cloud-based? I attended the ICSA 2017 conference, where microservices were the topic for some keynotes and papers. An observation was that all industrial presentations related to microservices discussed microservices outside the domain of cloud. Hence there seems to be a momentum of “microservitization” also outside the traditional cloud-based web-scale applications of Amazon, Google and the other giants. Stripping out the cloud related aspects in the list above makes it less deployment domain constrained:

- Large software systems with large numbers of developers or long and expensive test cycles
- A competitive environment that requires the rapid upgrading and release of online systems or business services
- Multiple software-based products or online services
- Migration from building and maintaining systems to buying more components that will be continuously upgraded by vendors
- Integration with systems on different platforms
- High volume of usage on cloud-based platforms
- Large flow of data, or rapidly changing data structures

It seems it is the size of the problem, the market momentum and the size of the developing organization that determines whether or not microservices are the answer or not. A limiting factor may the domains ability to support independent deployments. Could it be considered a microservices architecture also with “only” support for independent integration, omitting the independent deployment part, microservices are not excluded from any domain.

6.1.2 Claimed benefits

Journal article “Building Critical Applications Using Microservices”, reference [38], claims microservice architectures built on secure containers can
build critical applications with tools and services built for less critical software, which simplifies how to build critical applications.

Conference paper “Towards the Understanding and Evolution of Monolithic Applications as Microservices”, reference [56], lift forward that microservices architecture benefits maintainability due to separation of functionalities, because every microservice can evolve independently from the rest of the application.

Journal article “Microservices”, reference [60], lift forward that the key architectural advantage of modular architectures is that they tackle the complexity of monolithic architectures. In addition, there are other advantages, mainly as side benefits to breaking the code into smaller pieces:

- It’s easier to make changes, update, and test.
- There are fewer barriers to introducing new technology trends.
- They’re likely faster to start.
- It’s easier to mix and match modules with different profiles in terms of processor and memory needs, resulting in much better resource utilization.
- It’s easier to construct applications by bringing together modules with different functions.
- Conference paper “A Microservice Based Reference Architecture Model in the Context of Enterprise Architecture”, reference [49], lift forward the benefit “polyglot”: “A basic tenant of microservice based architecture is "choice of technology" when building microservices”, a great benefit for developers.


Journal article “Practical Use of Microservices in Moving Workloads to the Cloud”, reference [36], lift forward the benefit of being loosely coupled:

- In the location independence pattern, it doesn’t matter where the microservice exists; the other components that need to leverage the service can discover it within a directory and leverage it through the late binding process.
- In the communications independence pattern, all components can talk to each other, no matter how they communicate at the interface or protocol levels.
- The security independence pattern is based on the concept of mediating the difference between security models in and between components.
- In the instance independence patterns, the architecture should support component-to-component communications using both a synchronous and an asynchronous model, and not require that the
other component be in any particular state before receiving the request or message.

Another benefit is that you gain an understanding of the domain at a services level.

Journal article “The Economics of Microservices”, reference [44], is focused on the economics of microservices. Singleton states “Microservices are a solution - perhaps the only solution - to the problem of efficiently building and managing complex software systems”. He lists the following benefits one can gain when introducing microservices:

- For medium-sized systems, they can deliver cost reduction, quality improvement, agility, and decreased time to market.
- For large cloud systems, they fundamentally change the rules of the game. Microservices are the software equivalent of Lego bricks: they are proven to work, fit together nicely, and can be used to rapidly construct complex solutions.
- Microservices approaches fit particularly well with cloud computing, enabling the economic benefits of microservices to complement the economic benefits of cloud computing, such as cost and user experience optimization.
- The rapid release of microservices works well with cloud-based online systems that don’t require further distribution of software updates.
- If you have a large system that changes frequently and does need to scale, you benefit from a microservices architecture through several mechanisms:
  - It’s much easier to test and release the smaller components.
  - You get greater reliability through redundancy and scalability because of your ability to increase the instances of any service that’s a bottleneck.
  - You get greater quality through reuse of field-proven components packaged into microservices.
- A microservices-centric development team can test and release changes to smaller components more than once per day.
- Large software teams often have problems with merging code, in addition to testing. If many groups are merging, the process becomes difficult and unreliable. The microservices architecture solves this problem by skipping the merges. Each team can run an integration test on its code, and release it directly as a packaged service.
- Each microservice is assigned to a development team, which monitors it, fixes it, and releases improvements whenever they’re ready - whether once a month, once a week, once a day, or even more often.
- They maintain their APIs and feed their services into a continuous integration system to make sure that the whole system works correctly.
- This continuous process is more adaptable and easier to manage than the older Scrum-style agile development with its two-week cadence.
- You’ll benefit from a microservices architecture if you run continuous delivery of online services, if you do a lot of work merging code, or if you have long test cycles or high test expenses.
- Smaller microservices are easier to build, test, mix and match, configure, and deploy.
  - You can therefore have more frequent releases, smaller development teams, and faster onboarding of new developers and tech leads.
- Cloud vendors package their products into Web services. It’s easy to use these services if you already have a microservices architecture.
  - A microservices architecture will have a higher percentage of buy versus build. This can result in a very large reduction in build and maintenance costs, which may create a tradeoff in operational costs.
- Microservices architectures increase adaptability because you can swap out components and include components from different languages and platforms.
  - You can have different languages and platforms on different sides of an API call.
  - This is important if you do things like merger integration, where you’re combining unrelated software platforms.
  - It’s also important when you’re migrating between two platforms.
- Microservices are very useful when you’re supporting multiple products or developing new products and services. You can reuse services in more than one product. For example, most online service companies use many shared components for all of their products.
- When you divide your application into microservices, you gain more ways to increase its capacity.
  - You can add more instances of any microservice that comes under heavy load.
  - You can increase capacity more easily and cheaply than with a monolithic system, because you’re only adding the components that are heavily loaded, and not a whole complex app.
  - Your base load also declines, so you might get more advantages from moving to an on-demand cloud vendor.
In an extreme case, you might be able to take advantage of serverless architectures.

- “A microservices architecture imposes costs and complexity. However, it solves some expensive problems faced by the developers of complex systems.” “You should consider using a cloud-based microservices architecture if you’re dealing with any of the following types of complexity:
  - Large software systems with large numbers of developers or long and expensive test cycles
  - A competitive environment that requires the rapid upgrading and release of online systems or business services
  - Multiple software-based products or online services
  - Migration from building and maintaining systems to buying more components that will be continuously upgraded by vendors
  - Integration with systems on different platforms
  - High volume of usage on cloud-based platforms
  - Large flow of data, or rapidly changing data structures

He further states “I currently believe that if you have fewer than about 60 people working on your system, you don’t need a microservices architecture. Over this amount of product complexity, you’ll probably benefit from a microservices approach”.

Conference paper “Microservices Approach for the Internet of Things”, reference [51], lift forward the self-containment aspect of microservices. If adopting self-containment in the Internet of Things might create the following benefits:

- By having the back-end as part of the service, we can neglect dependencies for data storage. The data kept by a service should not directly be accessible from outside the service. This enforces the use of the service API and thus, decouples external data consumers from the internal representation of the data. Hence, this enables the independent evolution of services. In this case the internal data model can be freely changed, while maintaining interoperability.

- Having each service providing its own user interface would also enable independent evolution. This also omits the need for a centralized front-end that has to be aware of every possible device that might show up. Independent services might provide their e.g. HTML5 fragments, which can be put together into one dynamic panel.

- Providing the required libraries together with the service makes the deployment much easier, as the installation of dependencies is not required. Furthermore, if separated e.g. by containers, required libraries of different services do not interfere with each other.
• The limitation to have as less dependencies as possible leads to a better decoupling between services, an increase in autonomy and reduces the amount of required communication in the overall network. In contrast to that, limiting services to one concise business case leads to better independent evolvability, reduction in the services complexity, and a gain in freedom to compose services, but might raise the number of dependencies. So there is always a trade-off to consider.

In the Internet of Things, both the load balancer and the circuit breaker can be used either on its own or in combination. Both patterns have proven to be a good way to handle the fault of remote services. With regard to the constrained nature of many Internet of Things appliances these patterns have additional benefits:

• The circuit breaker prevents unnecessary messages sent to broken services.
  o This reduces the overall traffic in the constrained network and saves energy that was otherwise spent for retransmission.
• The load balancer can increase the lifetime of wireless sensors as the workload is shared among several devices, which enables them to stay longer in low power modes.
• As the circuit breaker can be used by every service (as needed) with no regard of the called service, this pattern is always possible even if the called service is provided by another vendor. Thus this is a good way to provide resilience to Internet of Things applications.

6.1.3 Highlighted risks

Conference paper “The Evolution of Distributed Systems Towards Microservices Architecture, reference [37], raises the following concerns related to microservices:

• Difficulty of coordination across development teams
• Avoiding Coupled Microservices
  o Addressed in the conference paper “Migrating to Cloud-native Architectures Using Microservices: An Experience Report”, reference [64], where the authors advice the use of consumer-driven contracts instead of service versioning to solve this matter.
• Increased network communication
• Need to consider adding network security
  o One methodology to provide safe access to host resources was provided in conference paper “Container and Microservice Driven Design for Cloud Infrastructure DevOps”, reference [46], where the propose use of special interfaces to access OS kernel functions from containers.
However, this is rather an aspect of stability rather than security, does not address the issue of malware services eavesdropping communication or maleficent service usage.

- The need to split out data repositories
- Cost of monitoring will increase
- Finding the right size and number of services

Journal article “The Economics of Microservices”, reference [44], raises the following concerns related to microservices:

- Microservices architectures are good for big systems but not for small systems. They require extra machinery to communicate between services, route to services, deploy services, and monitor services. If you have a small system that won’t change much and doesn’t need to scale, you can avoid this extra machinery and save time and money by building a more monolithic system
- A monolithic architecture works well for an installed software product that requires several months to be distributed, tested, and installed at customer sites. Bigger components (monolithic applications and macroservices) are easier to operate and have less code per feature because they have less inter-process communication.

Conference paper “Design and Implementation of a Decentralized Message Bus for Microservices”, reference [48], concludes microservices incur similar issues as any distributed system namely:

1) Operational complexity
2) Communication
3) Dependency between services
4) Availability and scalability

Conference paper “A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development”, reference [45], highlights risks related to system integration:

Unlike monolithic architecture based application (where systems integration generally happens during build time), systems or components integration for microservices architecture based application happens during runtime. This integration complexity shift from build to runtime phase increases the risk of failures in production environment. However, this risk can be reduced by having higher number of acceptance tests that exercise systems cohesiveness.

Conference paper “A Microservice Based Reference Architecture Model in the Context of Enterprise Architecture”, reference [49], lift forward the risk of “polyglot”: “A basic tenant of microservice based architecture is "choice of technology" when building microservices”, “but not from enterprise business perspective, as they are not tied into a specific stack or version of software. Considering the scenario where a few versions of 100 microservices are built each with its own unique set of toolset, tracking and managing license agreements of all these toolsets would be impossible at an enterprise level. A
"limited toolset" must be identified and updated as required and all microservices must be developed using the provided toolset at an enterprise level.”

Journal article “Microservices Architecture Enabled DevOps: Migration to a Cloud-Native Architecture”, reference [64], raises the following concerns related to microservices:

First, deployment in the development environment is difficult. Although the application code is now in isolated services, developers must also deploy the dependent services to run the isolated services on their machines. This problem occurred after we introduced dynamic service collaboration. To solve it, we chose Docker Compose and put a sample deployment description left in each service so that the dependent services can be easily deployed from our private Docker registry.

Second, service contracts are critical. Changing so many services that expose their contracts only to each other could be error-prone. Even a small change in the contracts can break part or even all of the system. One possible solution is service versioning, but it could make deploying each service even more complex. So, people usually don’t recommend service versioning for microservices. Thus, techniques such as the Tolerant Reader service design pattern are more advisable to avoid service versioning. Consumer-driven contracts could help greatly in this regard because the team responsible for a service can be confident that most of its customers are satisfied with the service.

Third, distributed-system development needs skilled developers. Microservices is a distributed architectural style. Furthermore, for such architectures to be fully functional, they need supporting services such as service discovery and a load balancer. During the early migration steps, we tended to spend much time describing these concepts and their corresponding tools and libraries to novice developers. Still, those developers often misused these things. So, to get the most out of microservices, teams need members who are familiar with these concepts and comfortable with this type of programming.

Fourth, creating service development templates is important. Polyglot persistence and the use of different programming languages are promises of microservices. Nevertheless, in practice, a radical interpretation of these promises could result in chaos in the system and even make it unmaintainable. As a solution, after architectural refactoring began, we started to create service development templates. We have different templates for creating microservices in Java using different data stores; these templates include a simple sample of a correct implementation. We’re also creating templates for Node.js. One simple rule is that a senior developer should first examine each new template to identify potential challenges.

Journal article, “Open Issues in Scheduling Microservices in the Cloud”, reference [41], notices the difference when it comes to heterogeneous
technologies within a microservice (polyglot) and the lacking research in heterogeneous support for the microservices execution environment. The paper concludes it needs more research in the areas:

- Configuration Selection and Management
- Application Topology Specification and Composition
- Performance Characterization and Isolation
- Microservice Monitoring
- Elastic Scheduling and Runtime Adaptation

For “Configuration Selection and Management” the challenge is dealing with heterogeneous configurations of microservices and cloud datacenter resources driven by heterogeneous performance requirements.

For “Application Topology Specification and Composition” the challenge is that different cloud technologies and provider specify their environment differently. Hence, an important research direction is to investigate an application-agnostic microservices composition framework.

For “Performance Characterization and Isolation” the challenge is that different deployments have different performance characteristics, see Figure 6.1. Furthermore, services have dynamic behavior. Hence container scheduling platforms must consider which microservices to combine to minimize workload interference and contention.

For “Microservice Monitoring” the challenge is that different resource types have different set of performance metrics attributes.

Due to all of the above uncertainties “Elastic Scheduling and Runtime Adaptation”, it is hard to accurately learn and fit statistical functions to the monitored distributions such as request arrival pattern, CPU usage patterns, memory usage patterns, I/O system behaviors, request processing time distributions, and network usage patterns.
6.1.4 Identified future research

Journal article, “Building Critical Applications Using Microservices”, reference [38], identify that microservices combined with secure containers can facilitate new ways of building critical applications. There are still several open research questions regarding whether and how it might support fail-operational applications.

Journal article, “Open Issues in Scheduling Microservices in the Cloud”, reference [41], raise these future research areas related to Scheduling and Resource Management:

**Configuration Selection and Management:**
We need new research that focuses on developing techniques for accurately modeling, representing, and querying configurations of microservices and datacenter resources. In addition, we need general-purpose decision-making techniques, driven by heterogeneous performance requirements, to automate the selection of microservice configurations and their mapping to heterogeneous datacenter resources.

**Application Topology Specification and Composition:**
An important research direction is to investigate an application-agnostic microservices composition framework, which will facilitate knowledge reuse and make it simpler for application engineers to interact with a complex computing platform.

**Performance Characterization and Isolation:**
Several new research topics are worthy of investigation:
- Performance isolation and characterization techniques when multiple microservices run in the same container or on the same physical host
- Live migration of containers to reduce interference and contention
- Tradeoffs between live migration and restarting

**Microservice Monitoring:**
Needs several new research topics, including development of holistic techniques for collecting and integrating monitoring data from all microservices and datacenter resources so administrators or a scheduler (a computer program) can track and understand the impact of runtime uncertainties (for example, failure, load-balancing efficiency, and overloading) on performance without understanding the whole platform’s complexity.

**Elastic Scheduling and Runtime Adaptation:**
Important new research is investigating predictive workload and performance models to forecast workload input and performance metrics across multiple, coexisting microservices deployed on cloud datacenter resources.

Conference paper “Microservices and Their Design Trade-offs: A Self-Adaptive Roadmap”, reference [47], identify the problem of finalizing the level of granularity of a microservice too early. “Splitting too soon can make things very difficult to reason about.” This problem is of significance both in brownfield and greenfield development since it affects the choice of concrete
realizations (and thereby microservice vendors when instantiating a system’s abstract architecture. One future work is to refine the understanding of the causal relationship between the granularity problem and its indicators. They also appreciate that integrating a self-adaptive, runtime solution into a running system raises practicality issues. Therefore, they envision potential in leveraging on symbiotic simulation approaches to implement the MAPE-K loop. Other areas that can be subject for future research is the lack of balance in the local/global non-functional requirement satisfaction trade-off.

6.2 Literature: Container technology related papers

This study is not about containers, but containers have grown popular in parallel with microservices which is indicated by the large number of cross references between microservices and containers in found papers. In principle every time microservices paper address deployment, containers are mentioned. Hence it is hard to ignore containers in a study related to microservices.

Compared to prior virtualization techniques, containers offers significantly higher performance and much faster deployment than traditional virtual machine based virtualization. With containers one do not need to pay the resource price of the hypervisor, nor the cost of a complete OS in each virtual machine. With containers all containers on one host share the same kernel. The security drawback of that is that if the kernel becomes compromised, it affects all containers running on that kernel, an area address in paper [65], “QoS Assurance with Light Virtualization - A Survey”.

Docker have almost become a synonym with containers. Paper [66], “Leveraging microservices architecture by using Docker technology”, address how to get the most out of Docker.

But containers existed before Docker, and other containers are still on the market. Paper [67], “Time Provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform”, compares a few container implementations from a performance perspective. Performance is not only about what implementation to use, but also how it is used. Paper [68], “Performance Evaluation of Microservices Architectures Using Containers”, compares two architectural styles how to use containers.

With microservices one need to address the problem of locating services. Paper [69], “Distributed Systems of Microservices Using Docker and Serfnode”, address this issue using container technology.

6.3 Continuous refactoring

As indicated in literature, and as well is the case for the equipment and resource management application in an Ericsson radio base station, a potential microservices based architecture is less frequent a green field architecture. It rather has evolved from a prior monolithic application. Evolving applications calls for refactoring.
Extreme Programming (XP), reference [70], is one of the methods predating agile development, but that now is considered an agile process among others. One of the practices XP brought to the Agile tool palette is refactoring. According to the XP rules one shall “refactor whenever and wherever possible” in order to keep the code as simple as possible to support the latest set of requirements (the KISS principle, see Wikipedia, reference [71]).

Chen and Babar’s paper “Towards an Evidence-Based Understanding of Emergence of Architecture through Continuous Refactoring in Agile Software Development”, reference [72], indicate a satisfactory architecture can emerge from continuous refactoring, albeit it is not given. 70% of the respondents in a survey reported they had observed a satisfactory architecture emerge from continuous refactoring, while 85% had observed failure to reach a satisfactory architecture. Table 6.1 lists identified success factors. Most of them are obvious, like it is an advantage with experienced and skilled teams with a positive mindset. But worth to mention is the need for proper architecture governance and the need for continuous integration.
<table>
<thead>
<tr>
<th>Factor</th>
<th>Success Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change</td>
<td>Medium to high rate of change</td>
</tr>
<tr>
<td>Size</td>
<td>Small</td>
</tr>
<tr>
<td>Type</td>
<td>Support small releases</td>
</tr>
<tr>
<td>Maturity of AK</td>
<td>Mature Architecture Knowledge (AK)</td>
</tr>
<tr>
<td>System Age</td>
<td>Green field</td>
</tr>
<tr>
<td>Type of Architecturally Significant Requirements (ASR)</td>
<td>No demanding ASR that cannot be satisfied by refactoring</td>
</tr>
<tr>
<td>Criticality</td>
<td>Low criticality</td>
</tr>
<tr>
<td>Experience</td>
<td>Experienced</td>
</tr>
<tr>
<td>Skill</td>
<td>Skilled</td>
</tr>
<tr>
<td>Personality and Mindset</td>
<td>Willing to make change, learn, have passion, with dedication to good design</td>
</tr>
<tr>
<td>Team Size</td>
<td>Small</td>
</tr>
<tr>
<td>Distribution</td>
<td>Collocated</td>
</tr>
<tr>
<td>Safe Net</td>
<td>Automated testing with good coverage</td>
</tr>
<tr>
<td>Continuous Integration</td>
<td>Continuous integration</td>
</tr>
<tr>
<td>Good Design Principles</td>
<td>Applying good design principles such as DRY, SOLID, KISS</td>
</tr>
<tr>
<td>Management</td>
<td>Management support and commitment</td>
</tr>
<tr>
<td>Culture</td>
<td>Good communication channels, encouraging for taking ownership and commitment, open, blame-free</td>
</tr>
<tr>
<td>Structure</td>
<td>Embraces the openness of Agile approaches</td>
</tr>
<tr>
<td>Governance</td>
<td>Proper architecture governance</td>
</tr>
<tr>
<td>Maturity</td>
<td>Certain Level of Maturity</td>
</tr>
</tbody>
</table>

*Table 6.1 Success factors continuous refactoring*
7 Discussion

The intended target for the research has been met. The research question has been answered. The expected answer has been confirmed both through literature studies, as well from the interviews. Next step is to prototype the results in small scale, before investing too much in a wide scale architectural refactoring activity.

Initial studies related to continuous architecture showed very poor hit rates in prior research. Furthermore, initial searches on microservices also showed very little penetration into the embedded domain. However, deepen the searches, networking with academia and other practitioners has uncovered there is a momentum regarding microservices also outside the cloud-native sphere. For e.g. Internet of Things applications, hybrid solutions are asked for. Some part of the system preferably resides in cloud, other parts of the system consist of embedded devices. Together with other activities, this creates a good nutrition for a flourishing microservices community outside the traditional cloud based.

Regarding continuous architecture, despite the term not being used in the microservices context, one of the aspects often lift forward with microservices is its light weight ability to add and remove new services. And to migrate service location, also in runtime. All this is attributes very valuable when to evolve an architecture. The loose coupling between services makes it a good architecture for continuous refactoring. The small size of microservices makes replacing a service a relatively simple activity.
8 Conclusion

Microservices seems to be a candidate also for embedded systems, albeit more challenging for real-time critical parts of embedded systems with very tight time constraints. Some of the aspects lift forward as benefits are relevant also for embedded systems such the better isolation between parts enabling more efficient development and integration. Unfortunately, literature and most discussions related to microservices take a black and white perspective on the problem, Microservices vs Monolithic application. Where the monolithic property is whether or not the application is “independently deployable by fully automated deployment machinery”, as Fowler state it, reference [29]. Not any of the other properties of microservices.

Embedded systems are typically hardware constrained and seldom execute in cloud. Most common execution environment is a microcontroller with potentially a light weight operation system. For small systems developed by small teams it does not add enough value to bring in isolation techniques like virtualization, or independent delivery support for the different parts of the system, which is confirmed by e.g. the journal article “The Economics of Microservices”, reference [44]. However most large embedded systems constitute parts or components which are developed and verified independently. However not necessarily released and deployed independently. Hence the answer to RQ 3 “Are the findings from the studied domain generally applicable for embedded real time applications?” is no. Microservices are not generally applicable for embedded applications. Design styles are, like creating the system using small parts which communicate (only) over well-defined light weight APIs. And it is beneficial for development speed to create a fully automated integration machinery. But taking the last step and introduce full independence between the parts if one does not have the problem of dynamic scaling or the luxury of being able to deploy independently, it is not worth the investment in the required added hardware resources, or the consequence of increased system complexity.

The studied domain is not an average embedded system; it shares many properties of enterprise applications. But an attribute it shares with most embedded applications is that it is shipped to customers as a monolith which is deployed by the customer as an atomic application. An important question is if it has to be this way? Or if it is sufficient to be able to update separate parts independently to qualify for being a microservices based architecture, even if that property is not utilized? From an architecture point of view, I claim it is the ability to update and deploy independent that is important. For a mission-critical application it is a short term utopia to believe that a customer is willing to be responsible for the integration, and that the integration first takes place in her live network without prior thorough integration tests. The question comes down to whom is at risk in case the integration of the new service fails, potentially risking the whole system. Consider a car. As a manufacturer you
would like all instances of the car to run the latest software set. But the manufacturer is typically not the one that owns the relation with the end user, that relation is between the car dealer and the car owner, which may be a different person than the car user. Consider a critical bug is identified in the cars breaking system, one that requires a new version of the breaking service. Can the car manufacturer upgrade this service over the air to the complete car fleet? Who is at risk? The car manufacturer? Absolutely. In case the upgrade fails the manufacturer may go out of business since the car is not considered trustworthy to drive and hence become unsellable. If the car is unsellable it as well put the car dealer at risk. But the person taking the real risk is the driver. She may sit on a curvy downhill road in the alps and suddenly the car loses its breaks. Perhaps an example taken to its extreme, but not an example not being discussed in industry related to how to get out new software to customer with acceptable risks. This risk is not only related to consequence of a failure in the intended software, but the systems vulnerability if someone deliberately injects a malicious software into the system. Which happens. Rumor tells that US cyber-attacks has caused frequent failures in the North Korean ballistic missile program, reference [73], which perhaps is good. But nevertheless, it is not a black and white question whether one want independent deployable services into live systems. However, all large scale organizations want efficient integration, and for that independence is crucial.

For the equipment and resource management application, microservices are a good fit. The answer to RQ 1 “Does a microservices based architecture better support continuous practices compared to the currently applied component based architecture in the studied domain (an embedded real time application)?” is yes. Current component based architecture has to o tight coupling between components to maximize development resource efficiency. Furthermore, the system resources are not that scarce, and the time constraints not that strict, that it disables microservices from being applicable. It is likely the ability to deploy independently does not stretch to customer installations in near time, but the ability makes a great difference for integration efficiency. Furthermore, building a system based on microservices are good for addressing system variability, reference [51], an important property of the studied domain.

To address RQ2, “What migration steps are needed for the studied domain to migrate to a microservices based architecture?”, the main challenge is to migrate the application during parallel development of new features. Breaking apart a large application to smaller services is not the worst thing, but migrate a very stateful application into stateless microservices is challenging. In addition, today’s architecture is well supporting Conway’s law, reference [2], with an organization that is rather layer oriented than domain oriented. As a result, components are rather defined to support organizational responsibilities rather than around bounded contexts in the problem domain. As a consequence, components gets too big, including many services in each component, and are used within more than one bounded context. In addition,
interfaces between components sometimes address more than one service. Interfaces has encapsulated the relation between components rather than between services. To address this, one need to identify the real bounded context. When found, reorganization would help securing the wanted architecture. To quote Evan Bottcher and the inverse Conway maneuver “Design the organization you want, the architecture will follow (kicking and screaming)”. But independent of support from a reorganization, one has to start breaking apart the interfaces into service APIs enabling later on breaking out service per service. A service discovery infrastructure is already in place; what needs to be added is strategy how to identify different instances of the same service when the services are context dependent such it matters which instance of a services that is addressed.

An unexpected finding during this study was containers. Not so much for the ability to deploy different services efficiently into live networks, but for the potential in dramatically shorten the time to instantiate a new service in the continuous integration machinery. A long term goal for the continuous integration machinery is that it shall not distract developers to trigger an integration. Current stretch goal is to support a continuous integration machinery that not prevents frequent deliveries to integration. That goal is considered met when a developer got a positive or negative confirmation from the machinery within 5 minutes after a commit. Too not be distracting, the same time limit is considered to be a few seconds. A time not possible to reach with current deployment strategy.

### 8.1 Future Research

In academia and industry in general, much focus is on virtualization. For microservices the important aspect is independence and that can be achieved also without separate machines, virtual or physical. Looking at the definition by Lewis and Fowler, reference [29], there is nothing that ties microservices only to the cloud or virtualized domain. More research is needed how to provide independence into the resource limit domain of embedded services where even container based virtualization may be to resource consuming.

Another area for continued research is hybrid cloud solutions, where part of the application is deployed in a cloud environment and other parts of the application is deployed on dedicated hardware.

A third area for continued research is related to trust. Current strategies are related to a centralized security manager. Microservices in general is about decentralizing, and preferably that would be the case also for trust. Is it for example possible to explore block-chain techniques to enable trust without a central authority?

A fourth area is related to performance. How to provided guaranteed runtime characteristics in a virtualized environment. And how to provide close to bare metal performance in a virtualized environment, e.g. nano second propagation delay for 25GBit vNICs.
9 References


10 Appendix

10.1 Search result full text search "continuous architecture" OR "continuous architecting"

10.2 Literature overview

This appendix contains a short description of each paper in the final set of papers. The full search result for microservice is found in the following excel document:

10.2.1 Microservices in Practice

The first selected microservice architecture related paper is a journal article “Microservices in Practice”, reference [53]. They notice that many characteristics assigned to microservices emerged long before the term microservice was established. Many of the practices as well, “The Unix philosophy”, reference [74], to mention one. They lift forward the aspect of keeping the middleware dumb, and put the smartness in the endpoints as a rational why microservices may succeed where SOA got stuck. In addition, they emphasize the importance of an automated delivery stream. They claim any domain could apply a microservices architecture, as long as the problem domain is complex enough justifying the added complexity and infrastructure of a microservices architecture. They lift forward the importance of Domain driven design and bounded contexts, see reference [31], and then primarily the strategic design aspects of DDD, and claim less importance for the tactical DDD patterns (Aggregate, Entity, …).

10.2.2 Architecture of an interoperable IoT platform based on microservices

Microservice architecture related paper two is a conference paper, “Architecture of an interoperable IoT platform based on microservices”, reference [43]. It describes a microservices based architecture for an Internet of Things platform aimed to foster an ecosystem for Internet of Things applications. One of the rationales for the choice of a microservices architecture is to manage the scalability and interoperability problem of a large
variation of end-devices, and rapid development of new devices. Communication is based on a REST API for Internet of Things applications, CoAP, see reference [75]. One concern related to microservices are trust, can one trust the other services in the application. This paper proposes a token based approach. Each applications provides credentials to obtain a token granting time limited access to specific resources in the system. The very core of an Internet of Thing system is the exchange of data. To provide interoperability among services, then apply semantic technology. What semantic technologies or rules to use is not defined in the paper.

10.2.3 DevOps
Microservice architecture related paper three is a journal article on DevOps, reference [52]. This article relation to microservices is the deployability attributes of microservices. In order to acceleration deployment, the paper claims microservices needs DevOps. Furthermore, it states microservices should be cloud-based, at least if the focus is on efficient service delivery.

10.2.4 Practical Use of Microservices in Moving Workloads to the Cloud
Microservice architecture related paper four is a journal article discussing the value of microservices when offloading [parts of] an application to cloud, “Practical Use of Microservices in Moving Workloads to the Cloud”, reference [36]. The article emphasis the value of deployment independence when moving parts of an application. This paper put less focus on REST, only that they interact using language-agnostic APIs. This article proposes container technologies to mitigate dependencies to underlying infrastructure services. It further addresses the trust, or security, issue with microservices. The proposed solution in this article is to leverage a federated security system to create trust between services. When it comes to instance independence, the article becomes a bit contradictory. It states that services (or components) shall support both synchronous and asynchronous communication, at the same time as they shall be state agnostic and state full. Majority of other paper advocate services shall be stateless and solely rely on asynchronous communication. In case a state needs to survive, the state shall be contained in the exchanged interaction such the service does not need to keep track of it.

10.2.5 The Evolution of Distributed Systems towards Microservices Architecture
Microservice architecture related paper five is a conference paper “The Evolution of Distributed Systems towards Microservices Architecture”, reference [37]. The paper describes the architecture journey from client-server based system via mobile agents and service oriented architecture to microservices based architecture. In their pre-work they analyzed the rationales
behind why organizations have applied a microservices architecture. What they found was this (in following order):

- Fast response
- Fast software delivery
- Functional separation
- Loose coupling
- Reusability
- Resilience
- Reduce resources cost

The paper frequently refers to Newman’s book for best practice, reference [30]. To attack the resource cost of virtualization they propose container techniques, such as Dockers, to improve performance and decrease some of the overhead introduced by virtual machines. To manage deployment aspects of containers, the propose use of some cluster management software. In addition, some service to handle service discovery is needed. Compared to Service Oriented Architecture where SOAP was frequently used, they propose a more light-weight communication framework, preferably REST based. They have identified a set of challenges one need to tackle when introducing a microservices based architecture:

- Difficulty of coordination across development teams
- Avoiding Coupled Microservices
  - Addressed in the conference paper “Migrating to Cloud-native Architectures Using Microservices: An Experience Report”, reference [64], where the authors advice the use of consumer-driven contracts instead of service versioning to solve this matter.
- Increased network communication
- Need to consider adding network security
  - One methodology to provide safe access to host resources was provided in conference paper “Container and Microservice Driven Design for Cloud Infrastructure DevOps”, reference [46], where the propose use of special interfaces to access OS kernel functions from containers. However, this is rather an aspect of stability rather than security, does not address the issue of malware services eavesdropping communication or maleficent service usage.
- The need to split out data repositories
- Cost of monitoring will increase
- Finding the right size and number of services
10.2.6 Security-as-a-Service for Microservices-Based Cloud Applications

Microservice architecture related paper six is a conference paper addressing “Security-as-a-Service for Microservices-Based Cloud Applications”, reference [40]. This paper focus on the security risk introduced by a microservices architecture. The assumptions in this paper is that all services belonging to the same application belong to the same trusted computing base (TCB), and hence by default trust each other. As a consequence, a single adversary application may compromise the security of the complete application. In this paper the author proposed a method to break apart the services from a common trusted computing base by introducing a software defined network (SDN) between the services. In the SDN infrastructure it is possible to monitor and control all communication between the services. Specific security services can be assigned this task, and by deploying them in separate “security” virtual machines they can be protected from the potential adversary services. The SDN network is not visible for the different application services, and hence they are not affected by introduced security filter (apart from reduces network performance). This approach builds on that one can trust
the integrity across separate virtual machines, that the hypervisor is not compromised.

10.2.7 The Economics of Microservices

Microservice architecture related paper seven is a journal article, “The Economics of Microservices”, reference [44]. As the title indicate the article address the economics of microservices architectures. The author admits microservices bring much value to complex systems, but at a cost. He quantifies that a system developed by less than 60 persons may not gain enough economic benefit from being made as a microservices architecture to motivate the added cost. He also discusses the size of microservices, and claim micro is not about lines of code, but about doing only one thing, but do the complete thing. This may end up in microservices of 10 lines of code or a million lines of code, what is necessary for making the service complete.

10.2.8 Microservices and Their Design Trade-offs: A Self-Adaptive Roadmap

Microservice architecture related paper eight is a conference paper, “Microservices and Their Design Trade-offs: A Self-Adaptive Roadmap”, reference [47]. They conclude there is still not a uniform definition of what a microservices architecture is, but given the commonly referenced sources of Newman, reference [30], Lewis et al., reference [29], but also a reference to Sader, reference [63], they came up with an own definition of microservices: ”Autonomic, replaceable and deployable artefacts of microservitization that encapsulate fine-grained business functionalities presented to system users through standardized interfaces. The autonomy of these artefacts allows for governing them in a decentralized manner and tracing their changes.”. They identify a lot of good value from microservices, but warn about finalizing the level of granularity of a microservice too early. “Splitting too soon can make things very difficult to reason about. It will likely happen that you (the software architect) will learn in the process.”

They promote taking an event approach to microservices using an event bus to propagate capture upstream events and make them available for downstream services. To define service boundaries, they refer to bounded context, reference [31].

Any system, also microservices based, need to conform to a set of non-functional requirements. In this paper they group the non-functional requirements into local and global. Local requirements are those defining the interior structure of the system not noticeable by the user of the system, while global non-functional requirements define user exposed system characteristics. When decided to go for a microservices architecture, non-trivial decision problems (DPs) that constitute addressing the size/number and the local/global non-functional requirements satisfaction needs to be addressed.
DP1: A solution which manages these trade-offs has to determine (given the current environment scenario and the stakeholders’ definition of “optimality” regarding the trade-offs of concern):

- When does decomposing a microservice into more fine-grained ones achieve the required optimality for both trade-offs?
- When does merging several fine-grained microservices into a coarse-grained one achieve the required optimality for both trade-offs?
- When should the current level of granularity be kept without further merging or decomposition?

DP2: The chosen architecture still has to guarantee the functional requirements of the system, regardless the level of granularity of the microservices in that architecture.

DP3: Much of the uncertainties that relate to the choice of the optimal architecture and the knowledge that relates to the expected behavior of the system cannot be fully captured at design time.

A reflection they made related to DP1 and DP3 is that conformance to non-functional requirements only can be guaranteed in runtime. As a consequence, they treat the microservices based system as a self-adaptive system where they deploy a MAPE-K, see reference [76] and [77], control loop for adapting the system to fulfill the non-functional requirements. To feed the control loop with goals, they define an abstract system with abstract services the control loop tries to satisfy using concrete services.

10.2.9 Microservices Approach for the Internet of Things

Microservice architecture related paper nine, “Microservices Approach for the Internet of Things”, reference [51], is a conference paper addressing microservices for Internet of Things applications. In the paper the authors give a brief overview on some new patterns and best practices that have emerged from the microservice approach or have made it possible. They covered the aspect of self-containment, dealing with service versions, monitoring and fault handling. The container technology was investigated as well as if orchestration or choreography should be used to put services together. As a result of this paper they notice that the architectural goals of both, microservices and the internet of things, are quite similar. The practice instead sometimes is different. The best practices and patterns that can be found in the microservices approach are partially already part of the SOA in the Internet of Things. Some, like to favor choreography, might already be known, but are in many cases not adopted in the internet of things, especially when using RPC or REST based protocols. The operating-system-level-virtualization is not yet adopted in the Internet of Things and might show a new possibility for the deployment and update of Internet of Things services and applications. When operating-system-level-virtualization would be used, the already existing patterns for the roll out of new versions can be used.
10.2.10 Building Critical Applications Using Microservices

Microservice architecture related paper ten is a journal article, “Building Critical Applications Using Microservices”, reference [38].

This article discuss how security can be addressed with the use of microservices. Traditionally critical applications have been built using a bottom-up approach. One begin with a dependable foundation—the correct CPU and system software—on top of which one build and run the critical applications. However, ensuring the correctness of the CPU and system software (OS, hypervisor, resource management system, and so on) is a grand challenge.

An alternative approach is to build critical applications on-top of untrusted foundation by using microservices. Microservices architecture are designed to be robust for other services availability. The issue is trust.

New CPU extensions, in particular Intel’s Software Guard Extensions (SGX; software.intel.com/en-us/sgx), allow applications to keep their states in encrypted memory (enclaves), thereby preventing even privileged software such as the OS and the hypervisor from accessing the data.

A major limitation of SGX’s current implementation is that it slows down by more than an order of magnitude if the working set of an enclave doesn’t t in the SGX extended page cache (EPC), which is currently only about 90 Mbytes. Hence, the state inside the enclave must be kept small—not only to minimize the trusted computing base’s size but also to ensure reasonable performance. To achieve this, they implemented and evaluated a microservices-based approach, see Figure 10.1 and Figure 10.2.

![Figure 10.1A microservice-based approach to keeping the state inside enclaves small. This not only minimizes the trusted computing base’s size but also ensures reasonable performance.](image-url)
Each secure container runs a single microservice instance inside an Intel Software Guard Extensions (SGX) enclave. This ensures confidentiality and integrity even against attacks from software with higher privilege such as the OS or hypervisor.

10.2.11 Design and Implementation of a Decentralized Message Bus for Microservices

Microservice architecture related paper eleven is a conference paper, “Design and Implementation of a Decentralized Message Bus for Microservices”, reference [48].

This paper concludes microservices incur similar issues as any distributed system namely:

1) Operational complexity
2) Communication
3) Dependency between services
4) Availability and scalability

Operational complexity: Microservices decompose an application into multiple small services. The more number of small services implies the more number of nodes in a system. To maintain the system with a high number of nodes, some tools, i.e., service discovery, configuration management and monitoring are necessary.
Communication: Services in microservices are separately deployed on different nodes. The communication between them is transformed from a local function call to a remote call. This would affect the system performance due to a high latency of network communication. Also, to provide an effective network communication, it requires some expertise. Thus, a lightweight communication infrastructure is required.

Dependency between services: A remote call between services needs communication. If using a basic connection between services, such as a static connection with a static IP, one would create dependencies between them. A communication format between services is also another cause of dependency. Dependencies make services hard to change and also decrease the flexibility of the system.

Availability and scalability: Many small services on several nodes in a microservice system would require some approach or tool that helps to increase availability and scalability. A tool like a load balance tool that is widely adopted to increase availability and scalability in a distributed system is needed.

This paper proposes a distributed message bus supporting service discovery, message passing, and load balancing, see Figure 10.3 and Figure 10.4.

![Figure 10.3 Logical view of the proposed message bus](image-url)
10.2.12 A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development


This paper addresses the issue of testing a system of autonomous, independently deliverable services. The paper promotes behavior-driven design, see reference [78]. Behavior-driven design is influenced by test-driven design. In behavior-driven design the test cases are black-box test preferably executed on a full system in a production like environment. The challenge with microservices developed in an agile way is to keep fully automated test suit, without spending too much effort maintaining it and to avoid duplicated test cases.

The proposed test architecture place behavior test cases in a separate repo from the services. The test repo contains a service folder with sub folders for each service, where all test specifications reside. In a separate folder, the test steps reside. Common test steps are broken out into a separate common library, see Figure 10.5.
The process of writing tests is the same as for test-driven development, one starts with a system that passes all tests, then one writes new test cases for the functionality to be added, such the tests fail. Then one implements the feature until tests once again passes, see Figure 10.6.

10.2.13 Towards the Understanding and Evolution of Monolithic Applications as Microservices

Microservice architecture related paper thirteen is a conference paper, “Towards the Understanding and Evolution of Monolithic Applications as Microservices”, reference [56].

It describes a method to factor apart a large monolithic application into a microservices based. The method is a three-phase process with a data-injection phase, a query phase, and a visualization phase.
In the data-injection phase one creates a model of the monolithic application. In the query phase the model is abstracted and refined, focusing on the objects, the data and the interaction in the system. The system is clustered into separate highly coupled parts with loose coupling in between.
The visualization phase visualizes proposed microservices, depending on selected clustering index, a courser or more fine grained solution is proposed.

Figure 10.9 Two different clusters of the same system differencing by used clustering factor

The implemented tool support Java beans applications.

10.2.14 Towards an Advanced Modeling System applying a Service-based Approach

Microservice architecture related paper fourteen is a conference paper, “Towards an Advanced Modeling System applying a Service-based Approach”, reference [39]. It describes an application built upon a mix of micro and macro-services. The services or architecture is not further described, and hence the paper does not add to the common knowledgebase around microservices architectures.

10.2.15 Episode 213 of Software Engineering Radio where Johannes Thönes talks with James Lewis about microservices

Microservice architecture related paper fifteen is a transcription of episode 213 of Software Engineering Radio where Johannes Thönes talks with James Lewis about microservices, reference [54]. The transcription has cut out parts, for the full episode, listen to the podcast, reference [79].

According to Lewis, a microservice is a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility. It is a single responsibility in the original sense that it’s got a single reason to change and/or a single reason to be replaced. But the other axis is a single responsibility in the sense that it does only one thing and one thing alone and can be easily understood.
Lewis states the majority of microservices architectures has been migrated into a microservices architecture from a prior monolithic application that has grown stale due to technical debt. The reason why microservices has grown popular now is because now tools exist to assist handling the complexity of many small things interacting over networked interfaces. In addition, cloud has established as an efficient platform to host and scale all those small services and infrastructure is given almost for free.

They discussed the pattern of "smart endpoints and dumb network". Lewis explains it emerged from ThoughtWorks observation that prior when introducing enterprise service busses, a lot of the application complexity went into the bus. The bus becomes the hiding place for the real spaghetti dependencies in the application and its rarely work well.

They as well discussed microservices relation to domain-driven design. Lewis states microservices have been influenced by many prior good practices, including domain-driven design. From domain driven design comes good practices like strategic design, bounded context, subdomains, how to separate out your domains, and how to partition a very big problem domain into smaller domains so that you can manage them.

Regarding the question "how big is micro", Lewis put focus on the doing one thing question. If a service does one thing and one thing only, one needs to focus less on size. It rarely goes beyond a few thousand lines of code. If you get to that point it is probably more important to think about how many of them you’re capable of supporting operationally than it is to think about how small they actually are because it’s better to have slightly bigger ones and fewer of them if you don’t have fully automated deployment into production.

10.2.16 Microservices

Microservice architecture related paper sixteen is a journal article, "Microservices" reference [60]. This article is the introducing article from the editor in chief column in an edition of IEEE Cloud Computing focusing on technologies supporting microservices. In the article Yousif states a definition of what microservices are: "They’re programs with a single task (or unit of work) that also include all the connectivity to the outside world as well as the runtime requirements to run the task. (Note that the word “task” is generic and refers to the smallest function possible, but no smaller.) Regardless, microservices inherit all the benefits of a modular architecture. Also increasing the developer community’s interest in microservices are containers and DevOps, which evolved around the same time that microservices did. Containers tailor themselves nicely to microservices because they can be deployed with much less overhead than virtual machines. DevOps represents an approach to developing, testing, and running code with tighter collaboration between developers, testers, and operators."

“Microservices (or modular architectures in general) are better suited for the many complex applications we’re building these days. This includes
enterprise applications (that is, confined within the enterprise) as well as Web-scale applications, where companies need to scale to reach consumers worldwide. Microservices, specifically, work well for new types of applications such as the Internet of Things, where single-function sensors and actuators are deployed in the field.”

10.2.17 A microservice based reference architecture model in the context of enterprise architecture

Microservice architecture related paper seventeen is a conference paper, “A microservice based reference architecture model in the context of enterprise architecture”, reference [49]. This paper concludes that microservice are high on Gartner's hype cycle for “application architecture”, see Figure 10.10. To assist developing cloud base enterprise microservice applications, this paper presents a microservice based reference architecture, see Figure 10.11.

The “microservice domain” is a business logical entity that contains a collection of related microservices. Its purpose is to aid with managing the complexity associated with governing the problems related to proliferation of microservices, such as multiple data models, platform and technology proliferation, cross-dependencies to other microservices, etc. The business functional domain would serve as the boundary of the business logical container.

The purpose of the “Enterprise API Registry” is to make the interfaces exposed by the microservice visible to consumers of the services both within and outside the enterprise. An “Enterprise API registry” is a shared component across the enterprise, whose location must be well known and accessible.

The “API Proxy” provides features such as providing protection to the underlying resources, providing a known location for remote resources, throttling, etc.

The “enterprise microservice repository” is a shared repository for storing information about microservices. It provides information such as microservice lifecycle status, versions, business and development ownership. Detailed information like its purpose, how it achieves the purpose, tools, technologies, architecture, the service it provides, any API's it consumes, data persisted and queried, any specific nonfunctional requirements. In the absence of well-defined repository standards, the enterprise must define its own standard specification artefacts for microservices.

The “enterprise service wrapper” component enables the access to legacy enterprise resources thru API technologies. It is an accessor component that will hook into the enterprise resources like the ESB and the legacy applications using Message Queuing, or SOAP or other technologies to expose services and make them available as APIs.

In any enterprise, operational intelligence is an important decision-making tool and any analytics tool is highly dependent on data. The fragmentation caused thru the deployment of microservices based architecture
means there are multiple disparate sources of information based on different implementations of microservices, each with a rather small subset of information. The “enterprise monitoring and tracking manager” requires that all emitters (microservices and other applications) publish standardized

Figure 10.10 Gartner Hype Cycle "Application Architecture"
10.2.18 Software Architects in Large-Scale Distributed Projects: An Ericsson Case Study

Microservice architecture related paper eighteen is a journal article, “Software Architects in Large-Scale Distributed Projects: An Ericsson Case Study”, reference [57]. It discusses the role of the architect in large distributed organizations and rationales why one product at Ericsson did not migrate to microservices. They define the architect roles using Martin Fowlers definition of architects, reference [80]

- **Architectus Reloadus** makes important decisions early on, ensuring a system’s conceptual integrity.
- **Architectus Oryzus** addresses problems in a project by closely collaborating with the developers.
For this product, the organization have identified a need for three levels of architects, see Figure 10.12.

![Diagram of software architects organization and workflow]

**Figure 10.12** The organization and workflow of the software architects. This centralized approach aims to ensure cross-team and cross-location alignment and to guard the complex system architectures.

Depending on the maturity of the team, they require different amount if guidance and got different amount of authority, see Figure 10.13.
Figure 10.13 The authority matrix. For a description of the maturity levels, see the main article. As a team matures, it should get more independent and require less support from the product-level architects.

Although the software architects’ hierarchy, roles, and responsibilities described aren’t unusual, they aren’t trendy, either. So, in this case study, why didn’t Ericsson fully embark upon trendier approaches such as microservices and agile ways of working? The answer is, because of legacy code and distributed development.

Although this case study implemented many agile practices such as continuous integration, continuous delivery, and DevOps, certain limitations prevented fully exploiting the new approaches. The product’s monolithic architecture made relying on coordination by mutual adjustment problematic.

However, this doesn’t automatically mean that Ericsson applied traditional coordination and control. The architects’ hierarchy wasn’t traditional in the sense that people higher up in the hierarchy could overrule the ones further down. Rather, it was a network of architects with different focus areas who were involved in decisions related to their competences. On the product level, the architects and other experienced developers had designated approval rights based on their competences. They governed this approval structure and process as they saw fit. This governance structure was similar to large distributed open source projects such as Eclipse and Android. That is, the open-source community has also concluded that large development
projects sometimes require a centralized approach to secure the software’s quality.

10.2.19 Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture

Microservice architecture related paper nineteen is a journal article, “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture”, reference [50]. This article describes the migration journey of Backtory when migrating to microservices.

Long gone is the time when Google searches on Microservices are “What is…”, now the searches are technology driven. The nature of microservices is a good fit for DevOps, and they have grown popular in parallel, see Figure 10.14

![Figure 10.14](image)

Figure 10.14 The increase in the use of the keywords “DevOps” and “microservices”, according to a Google Trends report.

Backtory is a mobile backend as a service application that prior to migration was deployed as a monolithic web service. The application was stepwise migrated to a microservices architecture when the need for a chat service that put new requirement on the application the old architecture could not meet. Their transformation journey is illustrated in Figure 10.15
Figure 10.15 Migrating Backory to microservices. Solid arrows indicate service calls; dashed arrows indicate library dependencies. (a) Backory’s architecture before the migration. (b) Transforming DeveloperData to a service. (c) Introducing the Configuration Server. (d) Introducing the Edge Server. (e) Introducing dynamic service collaboration. (f) Introducing ResourceManager. (g) Backstory’s target architecture after the migration.
To decouple microservices from each other they changed their integration machine such each service had its own flow enabling individual integration and deployment of services, see Figure 10.16.

![Figure 10.16](image-url)Moving from (a) a monolithic pipeline to (b) a microservices pipeline. The final delivery pipeline has independent delivery for each service, so each can be deployed independently.

To also gain deployment speed they at the same time changed into DevOps, a natural step to gain development speed, see Figure 10.17.

![Figure 10.17](image-url)DevOps team formation. (a) Traditional horizontal teams. (b) Vertical teams in DevOps. In DevOps, each team is responsible for a service and contains people with different skills, such as development and operations skills. The team members cooperate from the project’s start to create more value for the particular service’s end users.
Lessons learned when migrating to microservices:

First, deployment in the development environment is difficult. Although the application code is now in isolated services, developers must also deploy the dependent services to run the isolated services on their machines. This problem occurred after we introduced dynamic service collaboration. To solve it, we chose Docker Compose and put a sample deployment description left in each service so that the dependent services can be easily deployed from our private Docker registry.

Second, service contracts are critical. Changing so many services that expose their contracts only to each other could be error-prone. Even a small change in the contracts can break part or even all of the system. One possible solution is service versioning, but it could make deploying each service even more complex. So, people usually don’t recommend service versioning for microservices. Thus, techniques such as the Tolerant Reader service design pattern are more advisable to avoid service versioning. Consumer-driven contracts could help greatly in this regard because the team responsible for a service can be confident that most of its customers are satisfied with the service.

Third, distributed-system development needs skilled developers. Microservices is a distributed architectural style. Furthermore, for such architectures to be fully functional, they need supporting services such as service discovery and a load balancer. During the early migration steps, we tended to spend much time describing these concepts and their corresponding tools and libraries to novice developers. Still, those developers often misused these things. So, to get the most out of microservices, teams need members who are familiar with these concepts and comfortable with this type of programming.

Fourth, creating service development templates is important. Polyglot persistence and the use of different programming languages are promises of microservices. Nevertheless, in practice, a radical interpretation of these promises could result in chaos in the system and even make it unmaintainable. As a solution, after architectural refactoring began, we started to create service development templates. We have different templates for creating microservices in Java using different data stores; these templates include a simple sample of a correct implementation. We’re also creating templates for Node.js. One simple rule is that a senior developer should first examine each new template to identify potential challenges.

Finally, microservices architecture isn’t a silver bullet. It was beneficial for us because our system needed that flexibility and because we had Spring Cloud and Netflix OSS, which made migration and development much easier. However, as we mentioned before, adopting microservices will introduce complexities to the system that require much effort to resolve.
Open Issues in Scheduling Microservices in the Cloud

Microservice architecture related paper twenty is a journal article, “Open Issues in Scheduling Microservices in the Cloud”, reference [41]. This article notices the difference when it comes to heterogeneous technologies within a microservice (polyglot) and the lacking research in heterogeneous support for the microservices execution environment. The paper concludes it needs more research in the areas:

- Configuration Selection and Management
- Application Topology Specification and Composition
- Performance Characterization and Isolation
- Microservice Monitoring
- Elastic Scheduling and Runtime Adaptation

For “Configuration Selection and Management” the challenge is dealing with heterogeneous configurations of microservices and cloud datacenter resources driven by heterogeneous performance requirements.

For “Application Topology Specification and Composition” the challenge is that different cloud technologies and provider specify their environment differently. Hence, an important research direction is to investigate an application-agnostic microservices composition framework.

For “Performance Characterization and Isolation” the challenge is that different deployments have different performance characteristics, see Figure 10.18. Furthermore, services have dynamic behavior. Hence container scheduling platforms must consider which microservices to combine to minimize workload interference and contention.

![Figure 10.18](Comparison of cloud architectures: (a) hypervisor-based application deployment, (b) hypervisor-free containerized microservice, and (c) containerized microservice within a hypervisor-managed physical cloud hardware)

For “Microservice Monitoring” the challenge is that different resource types have different set of performance metrics attributes.

Due to all of the above uncertainties “Elastic Scheduling and Runtime Adaptation”, it is hard to accurately learn and fit statistical functions to the
monitored distributions such as request arrival pattern, CPU usage patterns, memory usage patterns, I/O system behaviors, request processing time distributions, and network usage patterns.

10.2.21 Migrating web applications to clouds with microservice architectures
Microservice architecture related paper twenty-one is a conference paper, “Migrating web applications to clouds with microservice architectures”, reference [81]. Focus in paper is the analysis of what cloud service model best meets the applications requirements. The paper does not go into detail how to transform a monolithic application into a microservices based application. Given the quality attributes of the resulting microservices, one can validate how well a specific cloud technology fulfills them and then base the selection of which cloud technology is the best fit for the application. The paper does not describe the evaluation process either.

10.2.22 Embedding security and privacy into the development and operation of cloud applications and services
Microservice architecture related paper twenty-two is a conference paper, “Embedding security and privacy into the development and operation of cloud applications and services”, reference [55]. This paper target security concerns that arises when migrating applications to cloud. The paper describes a proof-of-concept framework (ARCADIA) that utilize the isolation techniques virtualization offers.

An ARCADIA application is defined using a service graph which specified the applications set of microservices and their interaction. Each microservice must fulfill and provide a set of service Metadata. The ARCADIA framework is a layered architecture where the microservices are loaded and controlled by a Smart Controller, controlling the system based on the service graph and the services metadata.

The paper does not go into details regarding the process of deploying new services, or the trust issue which follows container techniques. Neither does it address techniques for securing communication if services shall be kept unknown to each other.

10.2.23 The Design and Architecture of Microservices
Microservice architecture related paper twenty-three is a journal article, “The Design and Architecture of Microservices”, reference [42]. The article addresses the relations between microservices and standards and focus on three areas; microservice deliveries using containers, data formats and APIs, and messaging standards. Albeit there is no enforced standard to use when go for a microservices architecture, microservices build on-top of prior work and standards. Container technologies has evolved in parallel with microservices,
and since the need for light weight virtualization increases by the number of microservices, containers have been the natural choice when deploying microservices in cloud.

When the article was written there were no published standard for containers, but a few standardization efforts where ongoing. For data formats JavaScript Object Notation (JSON) and RESTful APIs are popular. For both formats there exists standards. For JSON there exist Ecma International’s ECMA-4041 and IETF’s RFC 7159, and for REST one can build on-top RESTful API Markup Language (RAML, http://raml.org) and Swagger, which has evolved into the Open API Initiative (https://openapis.org). For Internet of Things applications new standards are evolving like the Sensor Network Object Notation (SNON, www.snon.org) which is a representation based on JSON that includes some predefined fields that are especially useful in dealing with sensor data. In addition, the Data Distribution Service (DDS, http://www.omg.org/spec/DDS) and DDS Data Local Reconstruction Layer (DDS-DLRL) specifications were developed by the Object Management Group.

Messaging is for microservices frequently carried by HTTP or HTTPS, since long standardized. TCP and UDP is of course fundamental, but for less capable hardware new standards are arising, e.g. Constrained Application Protocol (CoAP, IETF RFC 7252).

10.2.24QoS Assurance with Light Virtualization - A Survey

Container related paper one is a conference paper, “QoS Assurance with Light Virtualization - A Survey”, reference [65]. This paper notices the drawbacks with classical virtualization related to resource consumption and start-up time and look into container technologies. They notice that several container technologies offer light weight virtualization to a substantial lower resource cost that traditional full virtualization. However, container technology does not isolate the virtualized machines equally well since all containers on a host shares the same kernel.

This paper assesses a set of container orchestration tools in terms of Quality of Service (QoS) assurance capabilities such as High Availability (HA) management, service continuity, service discovery, and resource quota management. The assessed tools are: Mesos, Kubernetes, Docker Swarm, and Fleet.

While almost all of the tools support volume management to some extent to provide service continuity of the stateful services, Kubernetes and Aurora offer more flexibility. In terms of resource quota management, Kubernetes request and limit approach offers chance of better resource utilization compared to Aurora.
Marathon benefits from the quota management of Mesos. Fleet and Swarm are less powerful with this feature and accepts only some runtime constraints.

<table>
<thead>
<tr>
<th>Feature / Tool</th>
<th>Container Support</th>
<th>HA</th>
<th>Service Continuity</th>
<th>Chronic jobs</th>
<th>Quota Mgmt.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marathon</td>
<td>Mesos, Docker</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Mesos</td>
</tr>
<tr>
<td>Aurora</td>
<td>Mesos, Docker</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Kubernetes</td>
<td>Docker, Rocket</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Fleet</td>
<td>Docker, Rocket</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Swarm</td>
<td>Docker</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 10.2 Overall Comparison of container orchestration tools

In general, Aurora and Kubernetes have more to offer. Aurora is a multi-tenant framework that starts from scratch to download, deploy, configure an application and then protects its service availability and performance. It also accepts cron jobs. Kubernetes is a powerful container orchestration tool with fast service recovery, good availability and efficient resource management. Albeit being young, Kubernetes is evolving fast and is pretty mature. It is already ported to OpenStack, Mesos and many other Linux distributions. Supporting different container technologies is another advantage. Considering unified containers introduced by Mesos, Aurora which works with Mesos containers will be more flexible to accept different types of containers compared to Kubernetes.

10.2.25 Leveraging microservices architecture by using Docker technology

Container related paper two is a conference paper, “Leveraging microservices architecture by using Docker technology”, reference [66]. This paper starts with describing what microservices are:

- Small and focused
- Loosely coupled
- Language-neutral
- Bounded context

And then they lift forward a set of challenges that needs to be addressed before being able to benefit the value of microservices:

1) Failure isolation
2) Observability
3) Automation requirement
4) High independency
5) Testing
6) Scalability

The rest of the paper is focused on Docker, and how Docker support addressing above challenges:

- Accelerate automation
Docker containers are naturally very well fit for microservices architecture since each of them can be used as a deployment unit to granularly contain a service.

- Accelerate the independency
  - Each Docker container is an isolated box which can contain run time environment for a particular service.

- Accelerate portability
  - Docker puts application and all of its dependencies into a container which is portable among different platforms including Linux distributions and clouds.

- Accelerate resource utilization
  - Even though it’s not explicitly written down as a principle but being lightweight, portable are implication requirements towards being a microservices architecture friendly environment.
  - In Docker, each container is constituted by just the application and the dependencies which the app needs to be able to run, ideally neither more nor less. The container then runs as an isolated process on the host operating system, sharing the kernel with other containers.

- Secured
  - A lot of what Docker is offering are allowing developers to flexibly maximize the security of the code at different levels. When building the code, developers can freely use penetration test tools to stress test any part of build cycle.

10.2.26 Episode 217 of Software Engineering Radio where Charles Anderson talks with James Turnbull about Docker

Container related paper three is a transcription of episode217 of Software Engineering Radio where Charles Anderson talks with James Turnbull about Docker, reference [82]. The transcription has cut out parts, for the full episode, listen to the podcast, reference [83].

In this episode Turnbull, talks about Dockers, and concludes Lightweight Docker containers are rapidly becoming a tool for deploying microservices based architectures.

Two typical use cases for Dockers are:

- Continuous integration and continuous deployment
  - With Docker being so lightweight, developers can build stacks of Docker containers on their laptops that replicate some production environments. They can build and run their application. They can then move these containers around - they’re very portable.
o When running integration on a virtual machines (VMs), we have to spin up a new VM, install all the software, install your application source code, run the tests, and then probably tear it all down again because you may have destroyed the VM as part of the test process. Let’s say it would take 10 minutes to build those VMs. In the Docker world, you can build those VMs or the containers that replace them in a matter of seconds, which means if you’ve cut 10 minutes out of your build-test run, that’s an amazing cost saving.

• The other area where we’re seeing a lot of interest is what we call high capacity.
  o Traditional VMs have a hypervisor, which probably occupies about 10 to 15 percent of the capacity of a host. For many systems that 10 to 15 percent is quite an expensive 10 to 15 percent.

10.2.27 Distributed Systems of Microservices Using Docker and Serfnode

Container related paper four is a conference paper, “Distributed Systems of Microservices Using Docker and Serfnode”, reference [69]. This paper addresses the service discovery problem, “While containers can simplify the deployment and distribution of individual components, they do little to address the issue of communication between services over a complex networking layer”. It addresses the challenge of service discovery in microservice architectures by introducing Serfnode, a fully decentralized open source solution to the service discovery problem, based on the Serf project. Serfnode is a nonintrusive Docker image that composes one or more arbitrary Docker containers. The new images can be deployed into a cluster of Serfnodes, where it advertises itself and provides service discovery mechanisms, monitoring, and self-healing. The resulting cluster is a homogeneous and complete graph, with no master node.

The paper investigates what other tools address the area and compare them to Serfnode:

• Consul is a powerful open source solution for service discovery and distributed configuration management. It is among the best options for many deployments because it bundles service discovery with a strongly consistent key-value store, robust monitoring, and health checking. In some ways, Serfnode might be considered a lightweight alternative to Consul, both in features and complexity, that can be used for services running in Docker containers. The two solutions share the underlying Serf library for cluster management, but Serfnode focuses exclusively on service discovery which ultimately results in a more simplistic system.

• Synapse is another open source project for service discovery. Unlike Consul, Synapse was designed to support services running in Docker.
At the core of Synapse is an HAProxy instance that is used to route requests from a service consumer on the same host to a service provider running in the cluster. Updates to the HAProxy configuration are made by “watchers” - daemons that check for changes to the locations of services.

- CoreOS provides first class service discovery as a basis of their operating system through the etcd service. etcd is a distributed, consistent key value store for shared configuration and service discovery with a focus on ease of use and performance. While etcd is a viable, and often times the preferred solution for service discovery on CoreOS, when taken out of that ecosystem, it falls back to simply serving as another key/value store without tight integration into a broader service discovery solution such as the others listed here.

- The Docker platform has announced the coming of new offerings Docker Swarm and Docker Compose that, when combined together, may provide a solution to the service discovery problem. Docker Compose already provides a very limited service discovery mechanism, but it currently only works on a single host and it does not update as containers are stopped and restarted. It is unclear exactly what Swarm will enable as details of that feature have yet to be announced.

It is important to distinguish the service discovery problem from “orchestration” - the process of deploying containers on a cluster of machines. CoreOS, Mesos/Mesosphere, OpenShift, CloudFoundry, Kubernetes, Brooklyn/Clocker, Shipyard, and Crane are just a few of the many available orchestration solutions, and some of the larger projects like Kubernetes and Mesosphere include some service discovery features. A full survey and comparison is beyond the scope of this paper, but in general, orchestration solutions represent a fundamentally different approach to managing systems built with Docker. Adoption of these solutions is often a very heavy undertaking requiring highly specialized deployment environments.

10.2.28 Performance Evaluation of Microservices Architectures Using Containers

Container related paper five is a conference paper, “Performance Evaluation of Microservices Architectures Using Containers”, reference [68]. This paper look into performance tradeoffs between two variants of container technology; regular master-slave containers and nested-containers.
They performed a number of tests:

- CPU Performance Evaluation
- Comparing Overhead of Virtual Container Creation
- Overhead of Nested-Container Creation
- Network Performance - Local traffic in one Host
- Network Performance - Remote traffic across two hosts

Through the experiments of the paper, we evaluate the performance impact of choosing between the two models for implementing Related Processes Per Containers: in the first approach ("master-slave"), all child containers are peers of each other and a parent container serves to manage the slaves; the second approach ("nested-container"), involves the child being in its parent’s namespace.

Their results show that the nested-containers approach is a suitable model, thanks to improved resource sharing (same memory and disk), easily performing IPC and guaranteeing fate sharing among the containers in the same nested-container. The results show that nested-containers don’t have a significant impact on the performance of CPU, however, there are some trade-offs in terms of network performance compared to bare metal and regular containers. In any case, they add some of the simplicity that Virtual Machines offer in terms of infrastructure management flexibility and ease of workload deployment.

10.2.29 Time Provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform

Container related paper six is a conference paper, “Time Provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform”, reference [67]. The paper presents an evaluation of the provision time for OSv (Unikernel), Docker (Container), and KVM (Virtual Machine) on top of an OpenStack cloud platform.

The evaluation looked at these aspects:

- Instance and Operating System startup
- Image creation. Comprises the copy of the image from the repository and its preparation inside the compute node.
- Openstack Overhead. Considers the overhead generated by the platform with internal communication

<table>
<thead>
<tr>
<th>Virtualization</th>
<th>Format</th>
<th>Image Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>KVM</td>
<td>QCOW2</td>
<td>1.2G</td>
</tr>
<tr>
<td>Docker</td>
<td>TAR</td>
<td>342M</td>
</tr>
<tr>
<td>OSv</td>
<td>QCOW2</td>
<td>88M</td>
</tr>
</tbody>
</table>

Table 10.3 Images Properties

![Image of bar graph showing OS/Container Startup and Instance Startup times for 10, 20, and 30 instances]

Figure 10.20 Instance and Operating System/Container Startup for 10, 20 and 30 instances
10.2.3 Container and Microservice Driven Design for Cloud Infrastructure

DevOps

Container related paper seven is a conference paper, “Container and Microservice Driven Design for Cloud Infrastructure DevOps”, reference [46]. In the paper they explore the opportunities and challenges in using containers and applying the microservice design principles to operate and manage cloud infrastructure services. The goal of the work is to develop a more agile, reliable, and efficient DevOps approach for such infrastructure software.

Containerizing OpenStack is more than just running those management services in containers. For example, in the control plane the controller container should be informed where the database and the messaging services are, whenever these services are newly created, recovered from failure, or shutdown. Specifically, this means the services must have ways to register themselves, be discovered by other services, record their configuration/runtime state, and be generally orchestrated in their deployment and update processes. Therefore, we need an architecture that eases the operation of scaling-out, HA, and load balancing. Based on these requirements, the key components of the proposed architecture include:

- Service Proxy: all services are configured to be only accessible via a proxy, such that changes to a service instance do not affect others in the HA setup. Meanwhile, the proxy must have the up-to-date...
configurations of each service, e.g., IP address and port number. These configurations are retrieved from the configuration state manager.

- **Configuration State Manager (CSM):** manages the configurations of all services in the control plane. They implemented the CSM as a distributed key-value store based on etcd. To register the service in CSM, each container has its “sidekick” process. The sidekick process monitors the status of the service in the container periodically; depending on the status, the sidekick process will register or deregister the service in the CSM.

- **Service Orchestrator:** all services are managed by an orchestrator that decides how many instances of a service need to be run and where to run, based on certain policies (e.g., co-location or anti-colocation). The orchestrator not only launches containers along with their sidekicks on the same host, but also tracks the status of the container and takes action accordingly.

![Diagram](image_url)

*Figure 10.22A container-based microservice architecture of OpenStack. (a) OpenStack deployment, (b) Service registration and discovery in microservice architecture.*
Figure 10.23 Deployment comparison: containers vs. VMs. (a) Deploy one instance of each type on three hosts respectively; (b) Scale out instances on three hosts; (c) Host CPU utilization

10.2.31 Container management as emerging workload for operating systems

Container related paper eight is a conference paper, “Container management as emerging workload for operating systems”, reference [84]. This paper studied the scalability of container management operations for Docker, one of the most popular container management systems, from two aspects: core and container scalability, which indicate how much the number of processor cores and number of containers affect container management performance, respectively. They propose a hierarchical analysis approach to identify scalability bottlenecks where we analyze multiple layers of a software stack from the top to bottom layer. Their analysis reveals that core scalability has bottlenecks at a virtualization layer for storage and network devices, and that container scalability has bottlenecks at various components that inquire mount points. While those bottlenecks exist in a daemon process of Docker, the root causes are a couple of interfaces of the underlying kernel. This implies the operating system has room for improvement to more efficiently host emerging microservice applications. In the study the studied the scalability of two frequently used container management operations; building an image, and run a container.
Figure 10.24 Breakdown of processor cycles to build Nginx image when building eight images in single Docker daemon running on one and four processor cores with different types of storage drivers.

Figure 10.25 Breakdown of processor cycles to run Nginx container when running sixty-four containers in single Docker daemon on one and four processor cores with different types of storage drivers.
Running many containers does not scale, as shown in Figure 10.26.

**Figure 10.26** Breakdown of number of processor cycles to run BusyBox container when running 64 and 1,024 containers simultaneously in single Docker daemon on 1 processor core. The processor cycles are shown in seconds required to run one container.

**Figure 10.27** Top ten hot functions of packages github.com in Docker daemon with aufs running on one processor core when running multiple BusyBox containers concurrently. The numbers represent those of busy cycles required in each function to run one container.
Conclusion:

Container management is a new workload for OSs since it intensively stresses their administrative interface. The bottlenecks in container management with Docker they identified originate from the performance of administrative operations such as mounting and unmounting a filesystem, creating and removing a virtual block device, inquiring the mount point of a subsystem, and inquiring an attribute of a mount point. Operating systems have room for improvements to more efficiently host those emerging workloads.

10.3 Interviews

Following subchapters contains transcriptions of the performed interviews.

10.3.1 Interview organization one

Organization one implements a mobile network product similar to a base station. The product faces strong competition from many other vendors, why focus on cost is high. It contains a high speed data plane and a low speed control plane. Focus in this interview is the low speed control plain.

The development organization runs 10 cross-functional development teams spread over four sites, three countries and two time zones. They initiated a migration to agile way of working 2013, and consider themselves to have come far. Each team is product oriented, and has a fixed relation with a product owner.
The application consists of fairly few, quite large components. The application is distributed over a set of boards; a central control board, and a scalable set of signal processing boards. Most of the operation and maintenance application executes on the central control board. Local controllers execute on the signal processing boards. The control application primarily provides configuration and supervision support of the data plane. Normally an installed product is configured at installation and then executes over years without any further hands on. Occasionally software is upgraded, but configurations remains. The primary reason to upgrade the software is for adding support for new hardware variants with better cost or performance. New software is released twice a year, but customers do not normally upgrade until they really need to.

The organizations transformation to agile methods has dramatically changed the way they work with architecture. Before the agile transformation, a large group of system designers draw and maintained detailed system architecture models. Models that was not always aligned with the implemented architecture. Today only a handful system developers are left, and their prime responsibilities have transformed to look further into the future. Instead the cross functional teams are responsible to maintain the product architecture descriptions. As a consequence, only one architecture is maintained, the implemented. The agile transformation has resulted in improved product and architecture quality. In addition, flow efficiency has increased.

The organization uses continuous integration to secure quality and flow. There is currently no market drive for continuous deployment. The architecture is implicitly designed to enable full feature test using the products public interfaces, enabling architectural refactoring without impacting the test cases. It was a deliberate decision to avoid white-box testing in the CI loop to avoid lock-in effect on the architecture.

Success factors:
- Only one architecture to maintain, the implemented architecture
- Using only black-box feature testing enables continuous architecture refactoring

Challenges:
- Harder to build business case argumentation to gain funding to address technical debt

10.3.2 Interview organization two

Organization two implements a product utilizing the mobile network, rather than being part of the mobile network. The product is positioned in a fast growing market, “Internet of Things”, and offers a platform for “Internet of Thing” systems. There main objective is to be responsive to new demands, a new service shall be possible to develop and deploy in one day.
The development organization consists of three co-located agile teams. Before converting to agile practices, the organization was used to a waterfall based command and control process where developers were told what and how to implement. The organization existed before this application, but was assigned a new manager, a new architect and a new mission – this application.

First thing put in place was new ways of working, utilizing agile practices and empowered teams. This transformation was somewhat painful. Second priority was the architecture of the new application. The organization was uncertain where there were good seams in the domain and where natural boundaries would appear, and for that reason decided to start-off with a monolithic application to later on migrate it into a more fine-granular architecture.

When the uncertainties start to clear up, a stepwise restructuring started. The ultimate goal was a microservice based architecture where every service was possible to develop and deploy in isolation. Ambition for sizing a service was that it should be possible for two pair-programing developers to be able to rewrite the service from scratch in less than two days.

The system is event based, services are stateless and subscribes to events. When an event happens, the service that subscribed to that event get notified and execute its response to the event. The result of the execution is published as a new event, which is then consumed by another service. One service is not dependent on, or aware of, any other service, only the events it is dependent on and the infrastructure to subscribe for the events. Events are communicated using a REST API over HTTP. To keep the services stateless, the events carry the system state. To meet the demand on isolation, a cloud based infrastructure was selected supporting dynamic adding of services with possibilities to dynamically subscribe to events. In addition, the infrastructure supports scaling. If one service becomes a bottleneck, the platform can automatically spawn new instances of the service and load balance between them.

At the moment the former monolith has been refactored and the system consists of 1500 microservices, all focusing on doing one thing well. The services utilize different implementation techniques, what fits each service best, why the product manifest polyglot properties, one of the core potentials of a microservices architecture.

A drawback of many services are that it is hard to keep track of what services there are. To tackle this problem, they have created a service registry where services are categorized to ease navigation. They could invest in a tool to identify duplicates, but have not do so yet. They are more worried of development team refactoring existing services to tackle new problems, and as a consequence risk being multi-purpose oriented, than having duplicates. In addition, they like to have the freedom of performing A-B testing by having the same service implemented in two ways.
One identified advantage from the migration to microservices is the ease they can add new features to the system without affecting legacy. Since they “never” touch existing services (apart from error corrections), old features are not affected when new features are developed and deployed. In addition, they have no dependency between teams, why the development efficiency is high and administrative overhead is low.

10.3.3 Interview organization three

Organization three implements a product acting as a gateway between the mobile telecom network and the internet. The mobile network provides tunnels between the gateway and the mobile device. Each device can utilize multiple tunnels, but do not share tunnels with other devices. There are two types of tunnels, control and user plane tunnels. When the mobile roams, the tunnels need to be moved as well. The product consists of two parts. One part is responsible for session and mobility handling. It acts as the end point for the control tunnels. The other part address user packet processing and hence handles the user plane tunnels.

The organization has 40 cross-functional teams working with this product. The teams are spread over three sites, one in Sweden, one in India and one in China. To support the teams, the organization has 4 systems architects located in Sweden and 15 product architects co-located with the development teams, albeit not being part of the teams. The organization transitioned to agile way of working four to five years ago, and now considers themselves as agile. Already before the transition to agile way of working, they had some cross-functional teams, but it was when they got a new manager and reorganized they brought the different disciplines together across the whole organization. Initially there was a concern the teams should work across to divergent parts of the product resulting in lost domain knowledge. This has been addressed by letting teams belong to different areas of the product such they don’t feels like temporary visitors with the components but can act as guardians of the system. However, teams do not decide themselves what feature they are assigned to, product owners plan what team shall take on next feature given required competence and team availability.

The agile cross functional teams have the full responsibility for developing new features. There is a short preceding phase when features are pre-studied by system architects or system managers, but it is short and course grained. The real analysis, design, implementation, test, and documentation is done by the teams.

The team follows an internally developed process, inspired by public processes. The process depicts team to include the architects in early phases in the development of new features, to mitigate the risk of building technical debt and introduce illegal dependencies. To support keeping track of the dependencies in the product, the organization has developed a graphical
dependency tracking tool preventing new dependencies to be introduced in an uncontrolled fashion.

The organization’s current focus to improve development efficiency is to expand in continuous practices. They are now moving into continuous delivery with deliveries to customers every month.

The product has gradually been migrated towards a service based architecture. It can be discussed if it is a microservices architecture or not, it does not qualify all characteristics associated with microservices such as independent deployment of services. Nevertheless, it is service based. Before the migration to a service based architecture is was built around a monolith made of modules. There was little attention put on keeping the modules independent, which ultimately led to a large set of inter module dependencies which led to problems with code quality and development stagnation. When introducing support for LTE, the organization started splitting the monolith apart into separate processes. Each module should execute in its own process. This step made the inter module dependencies visible and easier to address. A framework was developed providing support for among other things process startup, inter process communication, service registration and look-up, event handling, and configuration. This framework could be considered as a microservices framework. It is designed following SOA-principles, where individual services have its own processes enabling separation between components. The frameworks support more flexibility than have been utilized. The reason why not all flexibility has been explored is due to conflicting product requirements. One important requirement is deterministic behavior, which is hard to achieve if one have not full control of what services there is and where they execute. To keep complexity down, the product uses predefined configuration of processes and services and try to avoid dynamic changes in the system. The organization is a bit skeptical towards some of the characteristics of “by the book” micro service architecture. How to guarantee predictability in case the system can dynamically evolve. And how to address the complexity related to scalability, redundancy, supervision, troubleshooting, security, and other dimensions that becomes more complex. And furthermore, it is important to keep track of CPU and memory utilization, overload protection and quality of service. Traditional enterprise applications do not address those questions; we need to communicate with our customers microservices are not a silver bullet answering everything. Furthermore, many of the advantages microservices lift forward are not limited to microservices, but are more general characteristics of a modular and distributed application.

One of the driving forces behind the migration to a service based architecture was to break apart the dependencies between components. This goal is reached, there is much fewer inter component dependencies now. Today the system consists of approximately 200 components. Size of a components is from only a few classes for a component up to some ten classes for larger
components. Each component has between 2 and 10 relations, all tracked and maintained. A service can be a single component or a group of components. A couple of teams share responsibility for a set of related components. The organization try to avoid single person or team responsibility for a component.

The product is layered, where some services are more infrastructure related and others are more related to the business domain. It is rare new features impact infrastructure related services which is a good indication that the architecture is fit for its purpose. In addition to the tool used during the migration phase, a second tool has been developed. A tool that models the complete set of services and components and the layering of the product. This tool helps in understanding the product and protect the product against layer violations.

The transition towards the service oriented architecture with a larger set of smaller independent components was conducted when we needed to scale development organization with an increased number of agile teams. The increased number of smaller components where a prerequisite to support more teams to work in parallel. The organization suffered when trying to scale the organization without adopting the architecture to the scaled number of teams. When splitting the former monolithic application into smaller chunks, the organization looked for seams in the application conducting domain analysis. The work did not follow any predefined process, but grouping together was made more ad hoc, from what fitted together. Some resulting components became too complex and has later need additional attention, and some components have been merged due to performance issues.

The transition to a service architecture was more driven from development organization efficiency than runtime scaling capabilities. The product did not suffer from performance issue before the transition, and the transition has not significantly improved performance.

The major advantage of the transition to a service based architecture is the improved separation between the components. In addition, the organization has been better in structuring new services. It has resulted in an improved architecture with better separation of concern. The drawback is that some parts become overcomplicated, with chains of services and many relations. The complete map can be a bit spaghetti-like. On the other hand, the product had the same level of spaghetti-relations also before, but then hidden within processes. With service oriented architecture these relations become visible and that simplifies the work of keeping APIs tidy. It furthermore prevents introduction of new “ugly” relations.

A strategy for the organization to keep the architecture fit, is to work with technical debt. The organization have collaborated with academia in research regarding technical debt, and the ability to communicate with management in monetary terms has resulted in support for maintaining a good architecture. Architects have a budget for maintaining the architecture. Approximately 50%
of the architects’ time is assigned for maintenance and the remaining time to support in development new features.

If the organization had to do the same journey ones more, it is likely they had ended up with a similar architecture.