Peer-reviewed version available at *Appl. Sci.* **2018**, *8*, 1368; doi:10.3390/app8081368

Review

A Brief History of Cloud Application Architectures

From Deployment Monoliths via Microservices to Serverless Architectures and Possible Roads Ahead - A Review from the Frontline (invited paper)

Nane Kratzke [1]

[1] Lübeck University of Applied Sciences, 23562 Lübeck, Germany
* Correspondence: nane.kratzke@fh-luebeck.de

Academic Editor: name
Version July 16, 2018 submitted to Preprints

Abstract: This paper presents a review of cloud application architectures and its evolution. It reports observations being made during the course of a research project that tackled the problem to transfer cloud applications between different cloud infrastructures. As a side effect we learned a lot about commonalities and differences from plenty of different cloud applications which might be of value for cloud software engineers and architects. Throughout the course of the research project we analyzed industrial cloud standards, performed systematic mapping studies of cloud-native application related research papers, performed action research activities in cloud engineering projects, modeled a cloud application reference model, and performed software and domain specific language engineering activities. Two major (and sometimes overlooked) trends can be identified. First, cloud computing and its related application architecture evolution can be seen as a steady process to optimize resource utilization in cloud computing. Second, this resource utilization improvements resulted over time in an architectural evolution how cloud applications are being build and deployed. A shift from monolithic servce-oriented architectures (SOA), via independently deployable microservices towards so called serverless architectures is observable. Especially serverless architectures are more decentralized and distributed, and make more intentional use of independently provided services. In other words, a decentralizing trend in cloud application architectures is observable that emphasizes decentralized architectures known from former peer-to-peer based approaches. That is astonishing because with the rise of cloud computing (and its centralized service provisioning concept) the research interest in peer-to-peer based approaches (and its decentralizing philosophy) decreased. But this seems to change. Cloud computing could head into future of more decentralized and more meshed services.

Keywords: cloud computing; service-oriented architecture; SOA; cloud-native; serverless; microservice; container; unikernel; distributed cloud; P2P; service-to-service; service-mesh

---

1. Introduction

Even very small companies can generate enormous economical growth and business value by providing cloud-based services or applications: Instagram, Uber, WhatsApp, NetFlix, Twitter - and much astonishing small companies (if we relate the modest headcount of these companies in their founding days to their noteworthy economical impact) whose services are frequently used. However, even a fast growing start-up business model should have its long-term consequences and dependencies in mind. A lot of these companies rely on public cloud infrastructures – currently often provided by Amazon Web Services (AWS). But will AWS be still the leading and dominating cloud service provider in 20 years? The IT history is full of examples that companies fail: Atari, Hitachi, America Online, Compaq, Palm. Even Microsoft – still a prospering company – is no longer *the* dominating software company it was used to be in the 1990's, and 2000's. Microsoft is even a good example for a company, that has evolved and transformed into a cloud service provider. Maybe because cloud providers becoming more and more critical for national economies. Cloud providers run a large amount

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

of mission critical business software for companies that no longer operate their own data-centers. And it is very often economical reasonable if workloads have a high peak-to-average ratio [1]. So, cloud providers might become (or even are) a to-big-to-fail company category that seems to become equally important for national economies like banks, financial institutions, electricity suppliers, public transport systems. Although essential for national economies, these financial, energy, or transport providers provide just replaceable goods or services – commodities. But the cloud computing domain is still different here. Although cloud services could be standardized commodities, they are mostly not. Once a cloud hosted application or service is deployed to a specific cloud infrastructure, it is often inherently bound to that infrastructure due to non-obvious technological bindings. A transfer to another cloud infrastructure is very often a time consuming and expensive one-time exercise. A good real-world example here is Instagram. After being bought by Facebook, it took over a year for the Instagram engineering team to find and establish a solution for the transfer of all its services from AWS to Facebook datacenters. Although no downtimes were planned noteworthy outages have been observed during that period.

The NIST definition of cloud computing defines three basic and well accepted service categories [2]: Infrastructure as a Service (iaaS), Platform as a Service (PaaS), and Software as a Sevice (SaaS). IaaS provides maximum flexibility for arbitrary consumer created software but hides almost no operation complexity of the application (just of the infrastructure). SaaS on the opposite hides operation complexity almost completely but is to limited for a lot of use cases involving consumer created software. PaaS is somehow a compromise enabling the operation of consumer created software with a convenient operation complexity but at the cost to follow resource efficient application architectures and to accept to some degree lock-in situations resulting from the platform.

Throughout the course of a project called CloudTRANSIT we searched intensively for solutions to overcome this "cloud lock-in" – to make cloud computing a true commodity. We developed and evaluated a cloud application transferability concept that has prototype status but already works for approximately 70% of the current cloud market, and that can be extended for the rest of the market share [3]. But what is more essential: We learned some core insights from our action research with practitioners.

1. Practitioners prefer to transfer platforms (and not applications).
2. Practitioners want to have the choice between platforms.
3. Practitioners prefer declarative and cybernetic (auto-adjusting) instead of workflow-based (imperative) deployment and orchestration approaches.
4. Practitioners are forced to make efficient use of cloud resources because more and more systems are migrated to cloud infrastructures causing steadily increasing bills.
5. And practitioners rate pragmatism of solutions much higher than full feature coverage of cloud platforms and infrastructures.

All these points influence ulteriorly how practitioners nowadays construct cloud application architectures that are intentionally designed for the cloud. This paper investigates the observable evolution of cloud application architectures over the last decade.

2. Methodology and Outline of this Paper

Figure 1 presents the research methodology for this paper. The reminder of this paper follows basically this structure. **Section 3** presents an overview of the research project CloudTRANSIT that build the foundation of our cloud application architecture problem awareness. The project CloudTRANSIT tackled intentionally the cloud lock-in problem of cloud-native applications and analyzed how cloud-applications can be transfered between different cloud infrastructures at runtime without downtime. From several researcher as well as reviewer feedbacks, we get to know that the insights we learned about cloud architectures merely as a side-effect might be of general interest for the cloud computing research and engineering community.

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

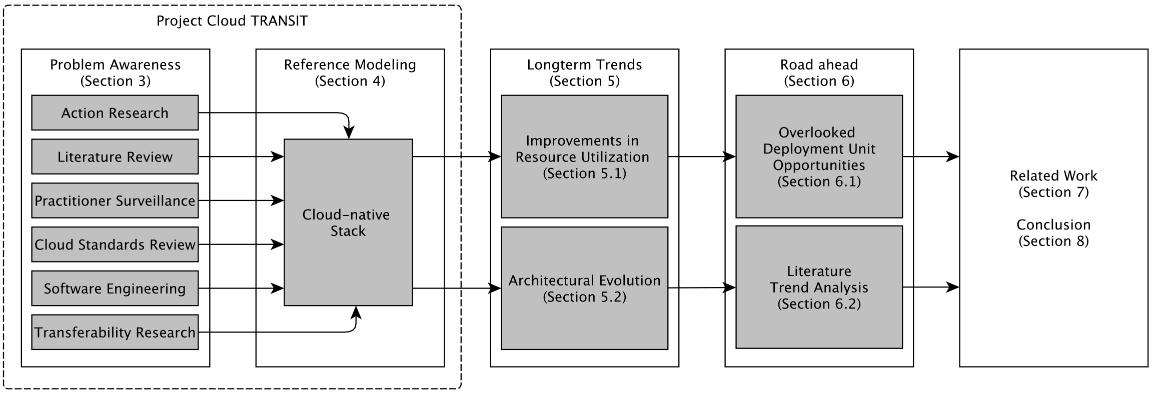

Figure 1. Research methodology

One thing we learned was the fact, that cloud-native applications – although they are all different – follow some common architectural patterns that we could exploit for transferability. **Section** 4 presents a reference model that structures such observable commonalities of cloud application architectures. Based on that insight, the obvious question arises what longterm trends exist that influence current shapes of cloud application architectures? **Section** 5 will investigate such observable long-term trends. In particular we will investigate the resource utilization evolution in **Section** 5.1 and the architectural evolution in **Section** 5.2. This ends to some degree the observable status quo. But the question is, whether these longterm trends will go on in the future and can they be used for forecasts? Although forecasts are tricky in general and our research has not invented a crystal ball, **Section** 6 will take a look on the road ahead mainly by extrapolating these identified trends. Some aspects can be derived from the observed long-term-trends regarding optimization of resource efficiency in **Section** 6.1 and architectural changes by a Scopus based literature trend analysis in **Section** 6.2. Obviously this paper is not the only one reflecting and analyzing cloud application architecture approaches and the reader should take related work in **Section** 7 into account as well. Finally we look at our brief history of cloud architectures and long-term trends. Assuming that these long-term trends will go on in the future for a while, we draw some conclusions on the road ahead in **Section** 8.

3. Problem Awareness (from the research project Cloud TRANSIT)

Our problem awareness result mainly from the conducted research project CloudTRANSIT. This project dealt with the question how to **transfer cloud applications and services at runtime** without downtime across cloud infrastructures from different public and private cloud service providers to tackle the existing and growing problem of vendor lock-in in cloud computing. Throughout the course of the project more than 20 research papers have been published. But the intent of this paper is not to summarize these papers. The interested reader is referred to the corresponding technical report [3] that provides an integrated view of these outcomes.

This paper strives to make a step back and review the observed state-of-the-art how cloud-based systems are being build today and how they might be build tomorrow. But obviously, it is of interest for the reader to get an impression how the foundation for these insights have been derived by understanding the mentioned research project.

The project analyzed commonalities of existing public and private cloud infrastructures via a review of industrial cloud standards and of cloud applications via a systematic mapping study of cloud-native application related research [4]. This was accompanied by action research projects with practitioners. Latest evolutions of cloud standards and cloud engineering trends (like containerization) were used to derive a reference model that guided the development of a pragmatic cloud-transferability solution. We evaluated this reference model using a concrete project from our action research

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 July 2018 doi:10.20944/preprints201807.0276.v1

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

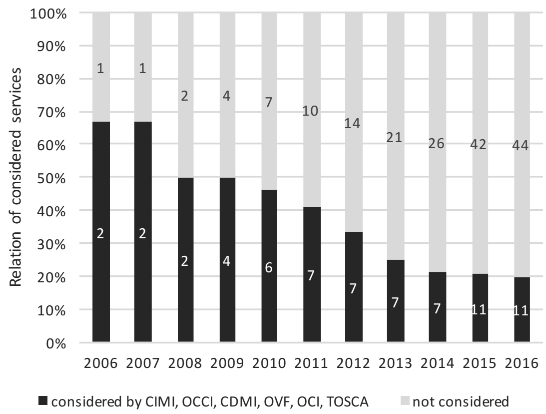

Figure 2. Decrease of standard coverage over years (by example of AWS)

activities [5]. This solution intentionally separated the **infrastructure-agnostic operation** of elastic container platforms (like Swarm, Kubernetes, Mesos/Marathon, etc.) via a **multi-cloud-scaler** and the **platform-agnostic** definition of cloud-native applications and services via an **unified cloud application modeling language**. Both components are independent but complementary and provide a solution to operate elastic (container) platforms in an infrastructure-agnostic, secure, transferable, and elastic way. This multi-cloud-scaler is described in [6,7]. Additionally we had to find a solution to describe cloud applications in an unified format. This format can be transformed into platform specific definition formats like Swarm compose, Kubernetes manifest files, and more. This unified cloud application modeling language UCAML is explained in [8,9]. Both approaches mutually influenced each other and therefore have been evaluated in parallel by deploying and transferring several cloud reference applications [10] at runtime [7,9]. This solution supports the public cloud infrastructures of AWS, Google Compute Engine (GCE), and Azure and open source infrastructure OpenStack. This alone covers approximately 70% of the current cloud market. Because the solution can be extended with cloud infastructure drivers also the rest of the market share can be supported by additional drivers of moderate complexity.

But what is more essential: We learned some core insights about cloud application architectures in general by asking the question how this kind of applications can be transferred without touching their application architectures. Let us investigate this in the following Section 4.

4. Reference modeling – how cloud applications look like

Almost all cloud system engineers focus a common problem. The core components of their distributed and cloud-based systems like virtualized server instances and basic networking and storage can be deployed using commodity services. However, further services – that are needed to integrate these virtualized resources in an elastic, scalable, and pragmatic manner – are often not considered in standards. Services like load balancing, auto scaling or message queuing systems are needed to design an elastic and scalable cloud-native system on almost every cloud service infrastructure. Some standards like AMQP [11] for messaging (dating back almost to the pre-cloud era) exist. But especially these integrating and "glueing" service types – that are needed for almost every cloud application on a higher cloud maturity level (see Table 1) – are often not provided in a standardized manner by cloud providers [12]. It seems that all public cloud service providers try to stimulate cloud customers to use their non-commodity convenience service "interpretations" in order to bind them to their infrastructures and higher-level service portfolios.

What is more, according to an analysis we performed in 2016 [13], the percentage of these commodity service categories that are considered in standards like CIMI [14], OCCI [15,16], CDMI [17], OVF [18], OCI [19], TOSCA [20] is even decreasing over the years. That has mainly to do with

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 July 2018 doi:10.20944/preprints201807.0276.v1

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

Table 1. Cloud Application Maturity Model, adapted from *OPEN DATA CENTER ALLIANCE [22]*

Level	Maturity	Criteria
3	Cloud native	- Transferable across infrastructure providers at runtime and without interruption of service. - Automatically scale out/in based on stimuli.
2	Cloud resilient	- State is isolated in a minimum of services. - Unaffected by dependent service failures. - Infrastructure agnostic.
1	Cloud friendly	- Composed of loosely coupled services. - Services are discoverable by name. - Components are designed to cloud patterns. - Compute and storage are separated.
0	Cloud ready	- Operated on virtualized infrastructure. - Instantiateable from image or script.

the fact that new cloud service categories are released faster than existing service categories can be standardized by standardization authorities. Figure 2 shows this effect by example of AWS over the years. That is how mainly vendor lock-in emerges in cloud computing. For a more detailed discussion the reader is referred to [5,13,21].

Therefore, all reviewed cloud standards focus a very small but basic subset of popular cloud services: compute nodes (virtual machines), storage (file, block, object), and (virtual private) networking. Standardized deployment approaches like TOSCA are defined mainly against this commodity infrastructure level of abstraction. These kind of services are often subsumed as IaaS and build the foundation of cloud services and therefore cloud-native applications. All other service categories might foster vendor lock-in situations. This all might sound disillusioning. But in consequence, a lot of cloud engineering teams follow the basic idea that a cloud-native application stack should be only using a very small subset of well standardized IaaS services as founding building blocks. Because existing cloud standards cover only specific cloud service categories (mainly the IaaS level) and do not show an integrated point of view a more integrated reference model that take best-practices of practitioners into account would be helpful.

Very often cloud computing is investigated from a service model point of view (IaaS, PaaS, SaaS), a deployment point of view (private, public, hybrid, community cloud) [2]. Or one can look from an actor point of view (provider, consumer, auditor, broker, carrier) or a functional point of view (service deployment, service orchestration, service management, security, privacy) as it is done by [23]. Points of view are particular useful to split problems into concise parts. However, the above mentioned view points might be common in cloud computing and useful from a service provider point of view but not from cloud-native application engineering point of view. From an engineering point of view it seems more useful to have views on technology levels involved and applied in cloud-native application engineering. This is often done by practitioner models. However, these practitioner models have been only documented in some blog posts[1] and do not expand into any academic papers as far as the author is aware.

Taking the insights from our systematic mapping study [24] and our review of cloud standards [5] we compiled a reference model of cloud-native applications. This layered reference model is shown and explained in Figure 3. The basic idea of this reference model is to use only a small subset of well

[1] Jason Lavigne, "*Don't let a PaaS you by - What is a PaaS and why Microsoft is excited about it*", see http://bit.ly/2nWFmDS (last access 13th Feb. 2018)

Johann den Haan, "*Categorizing and Comparing the Cloud Landscape*", see http://bit.ly/2BY7Sh2 (last access 13th Feb. 2018)

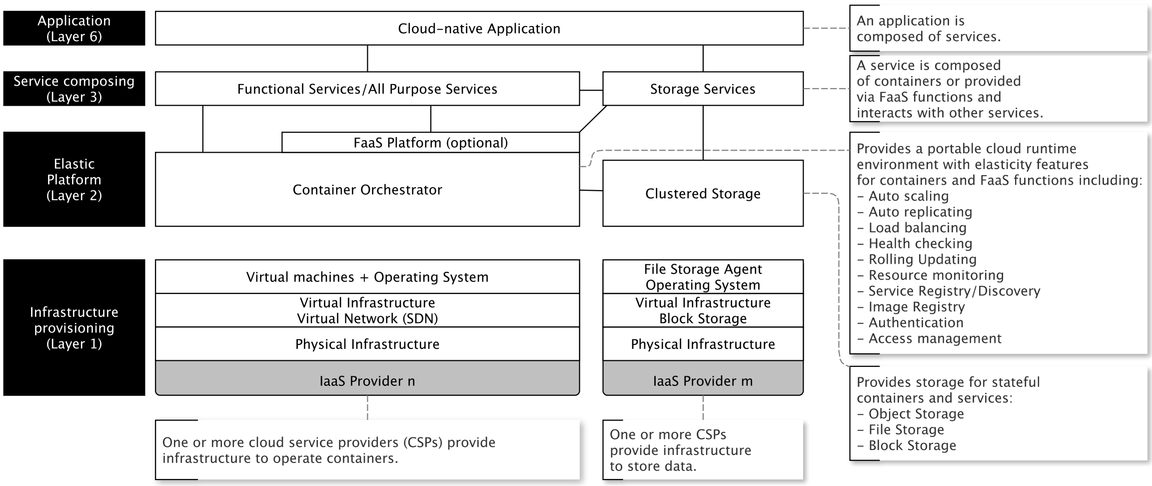

Figure 3. Cloud-native stack observable in a lot of cloud-native applications

standardized IaaS services as founding building blocks (Layer 1). Four basic view points form the overall shape of this model.

1. **Infrastructure provisioning:** This is a view point being familiar for engineers working on the infrastructure level. This is how IaaS can be understood. IaaS deals with deployment of isolated compute nodes for a cloud consumer. It is up to the cloud consumer what it is done with these isolated nodes (even if there are provisioned hundreds of them).
2. **Clustered elastic platforms:** This is a view point being familiar for engineers who are dealing with horizontal scalability across nodes. Clusters are a concept to handle many Layer 1 nodes as one logical compute node (a cluster). Such kind of technologies are often the technological backbone for portable cloud runtime environments because they are hiding complexity (of hundreds or thousands of single nodes) in an appropriate way. Additionally, this layer realizes the foundation to define services and applications without reference to particular cloud services, cloud platforms or cloud infrastructures. Thus, it provides a foundation to avoid vendor lock-in.
3. **Service composing:** This is a view point familiar for application engineers dealing with Web services in service-oriented architectures (SOA). These (micro)-services are operated on a Layer 2 cloud runtime platform (like Kubernetes, Mesos, Swarm, Nomad, and so on). Thus, the complex orchestration and scaling of these services is abstracted and delegated to a cluster (cloud runtime environment) on Layer 2.
4. **Application:** This is a view point being familiar for end-users of cloud services (or cloud-native applications). These cloud services are composed of smaller cloud Layer 3 services being operated on clusters formed of single compute and storage nodes.

For more details we refer to [3,5]. However, the remainder of this paper is aligned to this model.

5. Observable Longterm-Trends in Cloud Systems Engineering

Cloud computing emerged some 10 years ago. In the first adoption phase existing IT systems were simply transferred to cloud environments without changing the original design and architecture of these applications. Tiered applications were simply migrated from dedicated hardware to virtualized hardware in the cloud. Cloud system engineers implemented noteworthy improvements in cloud platforms (PaaS) and infrastructures (IaaS) over the years and established several engineering trends currently observable. But often these engineering trends listed in Table 2 seem somehow isolated. We want to review these trends from two different perspectives.

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

Table 2. Some observable software engineering trends coming along with CNAs

Trend	Rationale
Microservices	Microservices can be seen as a "pragmatic" interpretation of SOA. In addition to SOA microservice architectures intentionally focus and compose small and independently replaceable horizontally scalable services that are "doing one thing well". [25–29]
DevOps	DevOps is a practice that emphasizes the collaboration of software developers and IT operators. It aims to build, test, and release software more rapidly, frequently, and more reliably using automated processes for software delivery [30,31]. DevOps foster the need for independent replaceable and standardized deployment units and therefore pushes microservice architectures and container technologies.
Cloud Modeling Languages	Softwareization of infrastructure and network enables to automate the process of software delivery and infrastructure changes more rapidly. Applications and services and their elasticity behavior that shall be deployed to such infrastructures or platforms can be expressed by cloud modeling languages. There is a good survey on this kind of new "programming languages" [32].
Standardized Deployment Units	Deployment units wrap a piece of software in a complete file system that contains everything needed to run: code, runtime, system tools, system libraries. This guarantees that the software will always run the same, regardless of its environment. This is often done using container technologies (OCI standard []) Unikernels would work as well but are not yet in widespread use. A deployment unit should be designed and interconnected according to a **collection of cloud-focused patterns** like the *twelve-factor app* collection [33], the *circuit breaker* pattern [34] or *cloud computing patterns* [35,36].
Elastic Platforms	Elastic platforms like Kubernetes [37], Mesos [38], or Swarm can be seen as a unifying middleware of elastic infrastructures. Elastic platforms extend resource sharing and increase the utilization of underlying compute, network and storage resources for custom but standardized deployment units.
Serverless	The term serverless is used for an architectural style that is used for cloud application architectures that deeply depend on external third-party-services (Backend-as-a-Service, BaaS) and integrating them via small event-based triggered functions (Function-as-a, FaaS). FaaS extend resource sharing of elastic platforms by simply by applying time-sharing concepts [39–41].
State Isolation	Stateless components are easier to scale up/down horizontally than stateful components. Of course, stateful components can not be avoided, but stateful components should be reduced to a minimum and realized by intentional horizontal scalable storage systems (often eventual consistent NoSQL databases) [35].
Versioned REST APIs	REST-based APIs provide scalable and pragmatic communication, means relying mainly on already existing internet infrastructure and well defined and widespread standards [42].
Loose coupling	Service composition is done by events or by data [42]. Event coupling relies on messaging solutions (e.g. AMQP standard). Data coupling relies often on scalable but (mostly) eventual consistent storage solutions (which are often subsumed as NoSQL databases) [35].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 July 2018 doi:10.20944/preprints201807.0276.v1

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

- In Section 5.1 we will investigate cloud application architectures from a resource utilization point of view over time.
- And in Section 5.2 we will investigate cloud application architectures more from an architecture evolutionary point of view.

In both cases we will see, that the wish to make more efficient use of cloud resources had impacts on architectures and vice versa.

5.1. A review of the resource utilization evolution and its impact on cloud technology architectures

Cloud infrastructures (IaaS) and platforms (PaaS) are build to be elastic. Elasticity is understood as the degree to which a system adapts to workload changes by provisioning and de-provisioning resources automatically. Without this, cloud computing is very often not reasonable from an economic point of view [1]. Over time, system engineers learned to understand this elasticity options of modern cloud environments better. Eventually, systems were designed for such elastic cloud infrastructures, which increased the utilization rates of underlying computing infrastructures via new deployment and design approaches like containers, microservices or serverless architectures. This design intention is often expressed using the term "cloud-native".

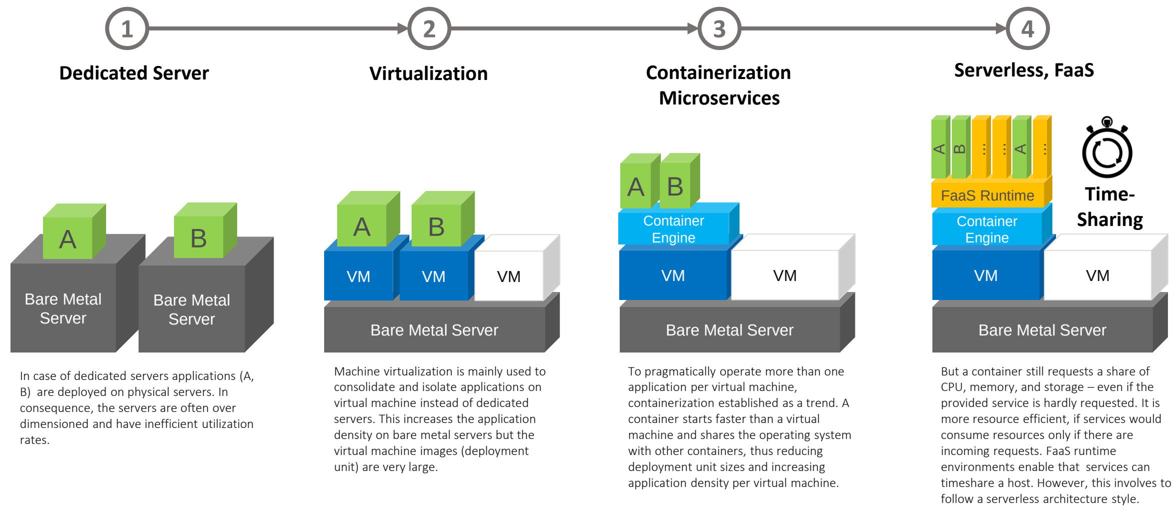

Figure 4. The cloud architectural evolution from a resource utilization point of view

Figure 4 shows an observable trend over the last decade. Machine virtualization was introduced to consolidate plenty of bare metal machines in order to make a more efficient utilization of physical resources. This machine virtualization forms the technological backbone of IaaS cloud computing. Virtual machines might be more lightweight than bare metal servers but they are still heavy, especially regarding their image sizes. Containers improved a standardized way of deployment but also increased the utilization of virtual machines, mainly because containers are more fine grained. Nevertheless, although containers can be scaled easily they are still always-on components. And "recently", Function-as-a-Service (FaaS) approaches emerged and applied time sharing of containers on underlying container platforms. Using FaaS only units are executed that have requests to be processed. Using this time-shared execution of containers on the same hardware. FaaS enables even a scale-to-zero capability. This improved resource efficiency can be even measured monetarily [43]. So, over time the technology stack to manage resources in the cloud got more complex and harder to understand but followed one trend – to run more workload on the same amount of physical machines.

5.1.1. Service-oriented Deployment Monoliths

An interesting paper the reader should dive into is [44]. Service-Oriented Computing (SOC) is a paradigm for distributed computing and e-business processing and has been introduced to manage the complexity of distributed systems and to integrate different software applications. A service offers functionalities to other services mainly via message passing. Services decouple their interfaces from their implementation. Workflow languages are used to orchestrate more complex actions of services (e.g. WS-BPEL). Corresponding architectures for such kind of applications are called consequently Service-Oriented Architectures (SOA). A lot of business applications have been developed over the last decades following this architectural paradigm. And due to its underlying service concepts these applications can be deployed into cloud environments without much problems. So, they are *cloud ready/friendly* according to Table 1. But the main problem for cloud system engineers emerges from the problem that – although these kind of applications are composed of distributed services – their deployment is not! These kind of distributed applications are conceptually monolithic applications from a deployment point of view. Dragoni et al. define such monolithic software as:

> *"A monolithic software application is a software application composed of modules that are not independent from the application to which they belong. Since the modules of a monolith depend on said shared resources, they are not independently executable. This makes monoliths difficult to naturally distribute without the use of specific frameworks or ad hoc solutions [...]. In the context of cloud-based distributed systems, this represents a significant limitation, in particular because previous solutions leave synchronization responsibilities to the developer [44]".*

In other words, the complete distributed application must be deployed all at once in case of updates or new service releases. This even leads to situations where complete applications are simply packaged as one large virtual machine image. That fits perfectly to situations shown in Figure 4(1 + 2). But depending on the application size, this normally involves noteworthy downtimes of the application for end users and limits the capability to scale the application in case of increasing or decreasing workloads. While this might be acceptable for some services (e.g. some billing batch processes running somewhere in the night), it might be problematic for other kind of services. What if messaging services (e.g. WhatsApp), large scale social networks (e.g. Facebook), credit card instant payment services (e.g. Visa), traffic-considering navigational services (e.g. Google Maps), or ridesharing services (e.g. Uber) would go down for some hours just because of a new service release or a scaling operation?

It is obvious that especially cloud-native applications come along with such 24x7 requirements and the need to deploy, update, or scale single components independently from each other at runtime without any downtime. Therefore, SOA evolved into a so called microservice architectural style. One might mention that microservices are mainly a more pragmatic version of SOA. But what is more essential, microservices are intentionally designed to be independently deployable, updateable, and horizontally scalable. This has some architectural implications that will be investigated in Section 5.2.1. But deployment units should be standardized and self-contained as well in this setting. We will have a look on that in the following Section 5.1.2.

5.1.2. Standardized and Self-contained Deployment Units

While deployment monoliths are mainly using IaaS resources in form of virtual machines that are deployed and updated in a less frequent manner, microservice architectures split up the monolith into independently deployable units that are deployed and terminated much more frequently. What is more, this deployment is done in a horizontal scalable way that is very often triggered by request stimuli. If there are a lot of requests hitting a service, more service instances are launched to distribute the requests across more instances. If the requests are decreasing, service instances are shut down to free resources (and save money). So, inherent elasticity capabilities of microservice architectures are much more in the focus compared with classical deployment monoliths and SOA approaches. One of

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

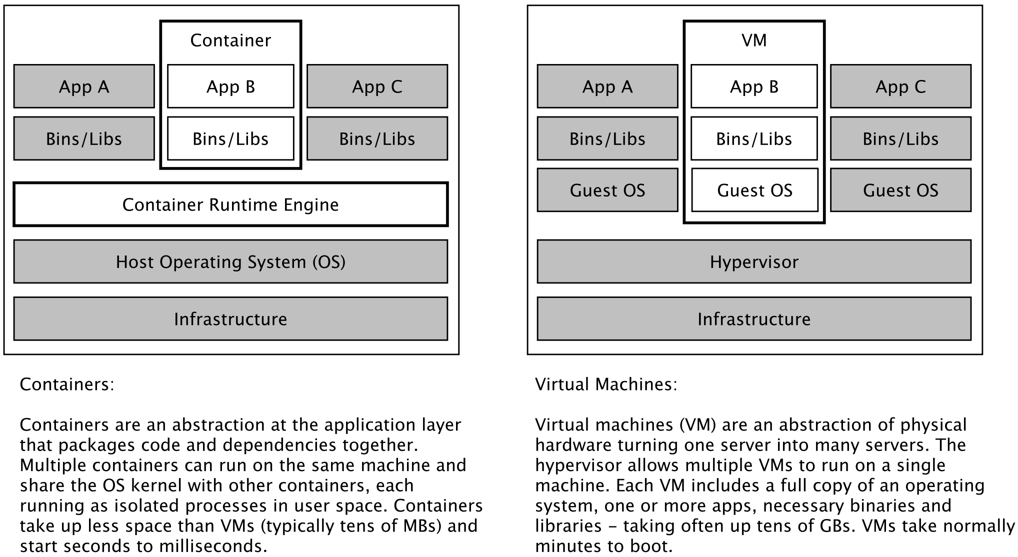

Figure 5. Comparing containers and virtual machines (adapted from Docker website)

the key success factors that microservice architectures gained so much attraction over the last years might be the fact, that the deployment of service instances could be standardized as self-contained deployment units – so called containers [45]. Containers make use of operating system virtualization instead of machine virtualization (see Figure 5) and are therefore much more lightweight. Containers enable to make scaling much more pragmatic, and faster and because containers are less resource consuming compared with virtual machines, the instance density on underlying IaaS hardware could be improved.

But even in microservice architectures the service concept is an always-on concept. So, at least one service instance (container) must be active and running for each microservice[2] at all times. Thus, even container technologies do not overcome the need for always-on components. And always-on components are one of the most expensive and therefore avoidable cloud workloads according to Weinmann [1]. Thus the question arises, whether it is possible to execute service instances only in the case of actual requests? And the answer leads to Function-as-a-Service concepts and corresponding platforms that will be discussed in Section 5.1.3.

5.1.3. Function-as-a-Service

Microservice architectures propose a solution to efficiently scale computing resources that are hardly realizable with monolithic architectures [44]. The allocated infrastructure can be better tailored to the microservices' needs due to the independent scaling of each one of them via standardized deployment units addressed in Section 5.1.2. But microservice architectures face additional efforts like to deploy each single microservice, and to scale and operate them in cloud infrastructures. To address these concerns container orchestrating platforms like Kubernetes [37], or Mesos/Marathon [46] emerged. But this shifts mainly the problem to the operation of these platforms and these platforms are still always-on components. Thus, so called Serverless architectures and Function-as-a-Service platforms have emerged in the cloud service ecosystem. The AWS lambda service might be the most prominent one but there exist more like Google Cloud Functions, Azure Functions, OpenWhisk, Spring

[2] And microservice architectures make use of plenty of such small services. To have a lot of small services is the dominant design philosophy of the microservice architectural approach.

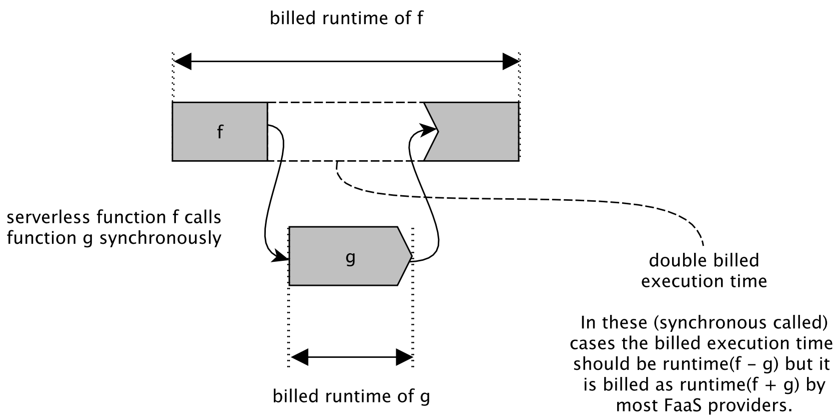

Figure 6. The double spending problem resulting from the Serverless trilemma [41]

Cloud Functions to name just a few. But all (commercial platforms) follow the same principle to provide very small and fine grained services (just exposing one stateless function) that are billed on a runtime-consuming model (millisecond dimension). The problem with the term Serverless is that it occurs in two different notions.

1. *"Serverless was first used to describe applications that significantly or fully incorporate third-party, cloud-hosted applications and services, to manage server-side logic and state. These are typically "rich client" applications—think single-page web apps, or mobile apps—that use the vast ecosystem of cloud-accessible databases, authentication services, and so on. These types of services can described as "Backend as a Service (BaaS) [39]".*
2. *"Serverless can also mean applications where server-side logic is still written by the application developer, but, unlike traditional architectures, it's run in stateless compute containers that are event-triggered, ephemeral (may only last for one invocation), and fully managed by a third party. One way to think of this is "Functions as a Service" or "FaaS". AWS Lambda is one of the most popular implementations of a Functions-as-a-Service platform at present, but there are many others, too [39]".*

In this Section the term Serverless computing is used in the notion of FaaS and we will mainly investigate the impact on resource utilization. The upcoming Section 5.2.2 will investigate Serverless more in architectural terms. FaaS was specifically designed for event-driven applications that require to carry out lightweight processing in response to an event [47]. FaaS is more fine grained than microservices and facilitates the creation of functions. Therefore, these fine-grained functions are sometimes called *nanoservices*. These functions can be easily deployed and automatically scaled, and provide the potential to reduce infrastructure and operation costs. Other like the deployment unit approaches of Section 5.1.2 – that are still always-on software components – functions are only processed if there are active requests. Thus, FaaS can be much more cost efficient than just containerized deployment approaches. According to a cost comparison of monolithic, microservice and FaaS architectures case study by Villamizar et al. cost reductions up to 75% are possible [43]. On the other hand, there are still open problems like the Serverless trilemma identified by Baldini et. al.. The Serverless trilemma *"captures the inherent tension between economics, performance, and synchronous composition"* [41] of serverless functions. One evident problem stressed by Baldini et al. is the "double spending problem" shown in Figure 6. This problem occurs when a serverless function f is calling another serverless function g synchronously. In this case, the consumer is billed for the execution of f and g - although only g is consuming resources because f is waiting on the result of g. To avoid this double spending problem a lot of serverless applications delegate the composition of fine grained serverless functions into higher order functionality to client applications and edge devices outside

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 July 2018 doi:10.20944/preprints201807.0276.v1

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

the scope of FaaS platforms. This leads to new – more distributed and decentralized – forms of cloud-native architectures that will be discussed in Section 5.2.2.

5.2. A review of the architectural evolution

The reader has seen in Section 5.1 that Cloud-native applications strived for a better resource utilization mainly by applying more fine-grained deployment units in shape of lightweight containers (instead of virtual machines) or in shape of functions in case of FaaS approaches. And these improvements of resource utilization rates had impact on how architectures of cloud applications evolved. Two major architectural trends of Cloud application architectures could be observed in the last decade. We will investigate Microservice architectures in Section 5.2.1 and Serverless architectures in Section 5.2.2.

5.2.1. Microservice architectures

Microservices form *"an approach to software and systems architecture that builds on the well-established concept of modularization but emphasizes technical boundaries. Each module — each microservice — is implemented and operated as a small yet independent system, offering access to its internal logic and data through a well-defined network interface. This increases software agility because each micro service becomes an independent unit of development, deployment, operations, versioning, and scaling [29]"*. According to [28,29] often mentioned benefits of microservice architectures are faster delivery, improved scalability and greater autonomy. Different services in a microservice architecture can be scaled independently from each other according to their specific requirements and actual request stimuli. What is more, each service can be developed and operated by different teams. So microservices do not only have an technological but also an organizational impact. These teams can make localized decisions per service regarding programming languages, libraries, frameworks, and more. So, best-of-breed breaches are possible within each area of responsibility on the one hand – on the other hand this might increase obviously the technological heterogenity across the complete system and corresponding longterm effects regarding maintainability of such systems might be not even observed so far [4].

Alongside microservice architectures several other accompanying trends could be observed. We already investigated containerization as such a trend in Section 5.1.2. **First generation microservices** formed of individual services that were packed using container technologies (see Figure 7). These services were then deployed and managed at runtime using container orchestration tools, like Mesos. Each service was responsible for keeping track of other services, and invoking them by specific communication protocols. Failure-handling was implemented directly in the services' source code. With an increase of services per application, the reliable and fault-tolerant location and invocation of appropriate service instances became a problem itself. If new services were implemented using different programming languages, but that made reusing existing discovery and failure-handling code became increasingly difficult. So, freedom of choice and "polyglott programming" is an often mentioned benefit of microservices but obviously has its drawbacks that needs to be managed.

Therefore, **second generation microservice architectures** (see Figure 7) made use of discovery services and reusable fault-tolerant communication libraries. Common discovery services (like Consul) were used to register provided functionalities. During service invocation, all protocol-specific and failure-handling features were delegated to an appropriate communication library, such as Finagle. This simplified service implementation and reuse of boilerplate communication code across services.

The **third generation** (see Figure 7) introduced service proxies as transparent service intermediates with the intent to improve software reusability. So called sidecars encapsulate reusable service discovery and communication features as a self-contained services that can be accessed via existing fault-tolerant communication libraries provided by almost every programming language nowadays. Because of its network intermediary conception, sidecars are more than suited for monitoring the behavior of all service interactions in a microservice application. This is exactly the idea behind service mesh technologies such as Linkerd. These tools extend the notion of self-contained sidecars

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

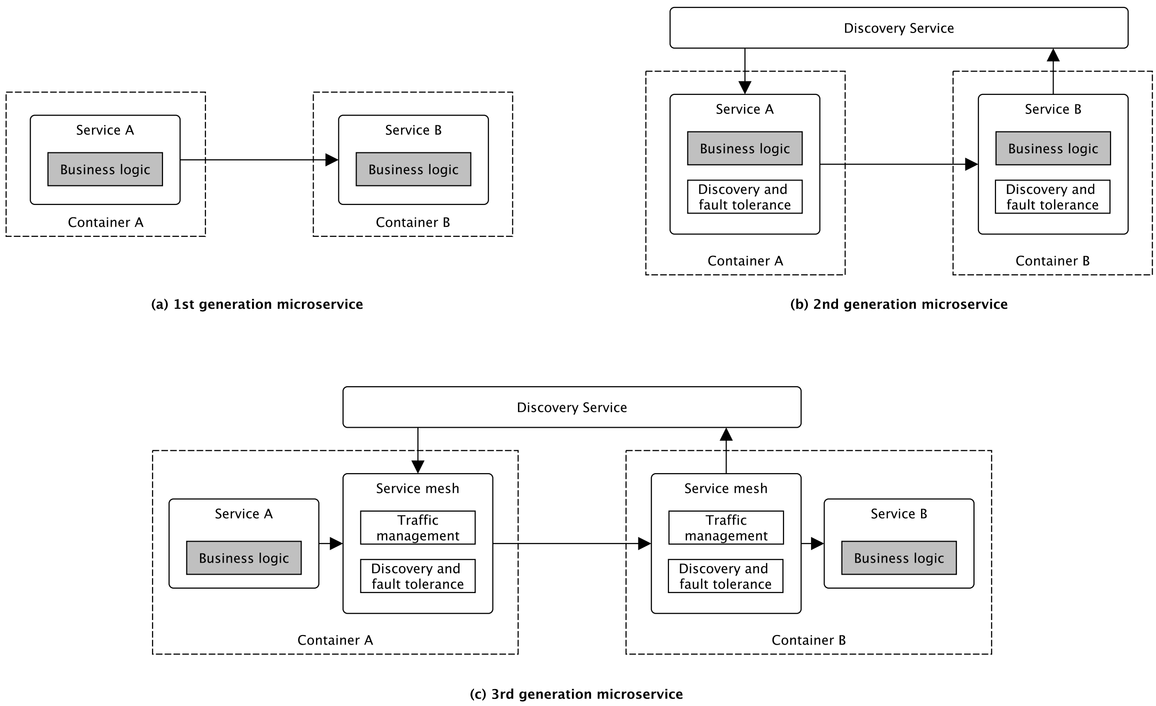

Figure 7. Microservice architecture evolution - adapted from [29]

to provide a more integrated service communication solution. Using service meshs operators have much more fine-grained control over the service-to-service communication including service discovery, load balancing, fault tolerance, message routing, and even security. So, beside the pure architectural point of view, the following tools, frameworks, services, and platforms (see Table 3) form our current understanding of the term *microservice*:

- Service discovery technologies let services communicate with each other without explicitly referring to their network locations.
- Container orchestration technologies automate container allocation and management tasks and abstracting away the underlying physical or virtual infrastructure from service developers. That is the reason we see this technology as an essential part of any cloud-native application stack (see Figure 3).
- Monitoring technologies that are often based on time-series databases to enable runtime monitoring and analysis of the behavior of microservice resources at different levels of detail.
- Latency and fault-tolerant communication libraries let services communicate more efficiently and reliably in permanently changing system configurations with plenty of service instances permanently joining and leaving the system according to changing request stimuli.
- Continuous-delivery technologies integrate solutions often into third party services that automate many of the DevOps practices typically used in a web-scale microservice production environment [30].
- Service proxy technologies encapsulate mainly communication-related features such as service discovery and fault-tolerant communication and exposes them over HTTP.
- Finally, latests service mesh technologies build on sidecar technologies to provide a fully integrated service-to-service communication monitoring and management environment.

Table 3 shows that a complex tool-chain evolved to handle the continuous operation of microservice-based cloud applications.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 July 2018 doi:10.20944/preprints201807.0276.v1

Peer-reviewed version available at *Appl. Sci.* **2018**, *8*, 1368; doi:10.3390/app8081368

Table 3. Some observable microservice engineering ecosystem components (adapted from [29])

Ecosystem component	Example tools, frameworks, services and platforms *(last access 11/07/2018)*
Service discovery	Zookeeper (https://zookeeper.apache.org), Eureka (https://github.com/Netflix/eureka), Consul (https://www.consul.io), etcd (https://github.com/coreos/etcd, Synapse (https://github.com/airbnb/synapse)
Container orchestration	Kubernetes (https://kubernetes.io, [37]), Mesos (http://mesos.apache.org, [46], Swarm (https://docs.docker.com/engine/swarm), Nomad (https://www.nomadproject.io)
Monitoring	Graphite (https://graphiteapp.org), InfluxDB (https://github.com/influxdata/influxdb), Sensu (https://sensuapp.org), cAdvisor (https://github.com/google/cadvisor), Prometheus (https://prometheus.io), Elastic Stack (https://elastic.co/elk-stack)
Fault tolerant communication	Finagle (https://twitter.github.io/finagle), Hystrix (https://github.com/Netflix/Hystrix), Proxygen (https://github.com/facebook/proxygen), Resilience4j (https://github.com/resilience4j)
Continuous delivery services	Ansible (https://ansible.com), Circle CI (https://circleci.com/), Codeship (https://codeship.com/), Drone (https://drone.io), Spinnaker (https://spinnaker.io), Travis CI (https://travis-ci.org/)
Service proxy	Prana (https://github.com/Netflix/Prana), Envoy (https://www.envoyproxy.io)
Service meshs	Linkerd (https://linkerd.io), Istio (https://istio.io)

5.2.2. Serverless Architectures

Serverless computing is a cloud computing execution model in which the the allocation of machine resources is dynamically managed and intentionally out of control of the service customer. The ability to scale to zero instances is one of the key differentiators of serverless platforms compared with container focused PaaS or virtual machine focused IaaS services. This enables to avoid billed always-on components and therefore excludes the most expensive cloud usage pattern according to [1]. That might be one reason why the term "serverless" is getting more and more common since 2014 [29]. But what is "serverless" exactly? Obviously, servers must still exist somewhere.

So called serverless architectures replace server administration and operation mainly by using Function-as-a-Service (FaaS) concepts [39] and integrating 3rd party backend services. Figure 4 showed the evolution of how resource utilization has been optimized over the last 10 years ending in the latest trend to make use of FaaS platforms. FaaS platforms apply time-sharing principles and increase the utilization factor of computing infrastructures, and thus avoid expensive always-on components. As already mentioned at least one study showed, that due to this time-sharing, serverless architectures can reduce costs by 70% [43]. The core capability of a serverless platform is that of an event processing system (see Figure 8). According to [41] serverless platforms take an event (sent over HTTP or received form a further event source in the cloud), determine which functions are registered to process the event, find an existing instance of the function (or create a new one), send the event to the function instance, wait for a response, gather execution logs, make the response available to the user, and stop the function when it is no longer needed. Beside API composition and aggregation to reduce API calls [41], especially event-based applications are very much suited for this approach [48].

Serverless platform provision models can be grouped into the following categories:

- **Public (commercial) serverless services** of public cloud service providers offer compute runtimes, also known as function as a service (FaaS) platforms. Some well known type representatives include AWS Lambda, Google Cloud Functions, or Microsoft Azure Functions. All of the mentioned commercial serverless computing models are prone to create vendor lock-in (to some degree).
- **Open (source) serverless platforms** like Apache's OpenWhisk or OpenLambda might be an alternative with the downside that these platforms need infrastructure to be executed on.

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

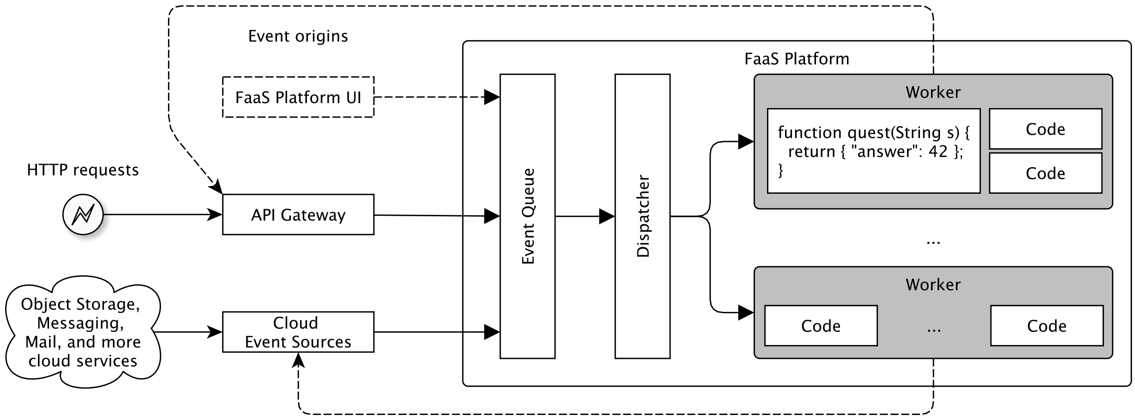

Figure 8. Blueprint of a serverless platform architecture (adapted from [41])

- **Provider agnostic serverless frameworks** provide a provider and platform agnostic way to define and deploy serverless code on various serverless platforms or commercial serverless services. This is an option to avoid (or reduce) vendor lock-in without the necessity to operate an own infrastructure.

So, on the one hand, serverless computing provides some inherent benefits like resource and cost efficiency, operation simplicity, and a possible increase of development speed and improved time-to-market [39]. But serverless computing comes also along with some noteworthy drawbacks, like runtime constraints, state constraints and still unsatisfactorily solved function composition problems like the double spending problem (see Figure 6). What is more, resulting serverless architectures have security implications. They increase attack surfaces and shift parts of the application logic (service composing) to the client-side (which is not under complete control of the service provider). Furthermore, FaaS increases vendor lock-in problems, client complexity, as well as integration and testing complexity. Table 4 summarizes some of the most mentioned benefits but also drawbacks of FaaS from practitioner reportings [39].

Furthermore, Figure 9 shows that serverless architectures (and microservice architectures as well) require a cloud application architecture redesign, compared to classical e-commerce applications. Much more than microservice architectures, serverless architectures integrate 3rd party backend services like authentication or database services intentionally. To reduce own development efforts, only very service specific, security relevant, or computing intensive functionality is provided via functions on FaaS platforms. In fact all functionality that would haven been provided classically on a central application server is now provided as a lot of isolated micro- or even *nanoservices*. The integration of all these isolated services as meaningful end user functionality is delegated to end devices (very often in the shape of native mobile applications or progressive web applications). In summary, we can see the following observable engineering decisions in serverless architectures:

- Former cross-sectional but service-internal (or via a microservice provided) logic like authentication or storage is sourced to external 3rd party services.
- Even nano- and microservice composition is shifted to end user clients or edge devices. That means, even service orchestration is not done anymore by the service provider itself but by the service consumer via provided applications. This has two interesting effects: (1) Resources needed for service orchestration are now provided by the service consumer. (2) Because the service composition is done outside the scope of the FaaS platform, still unsolved FaaS function composition problems (like the double spending problem) are avoided.
- Such client or edge devices are interfacing 3rd party services directly.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 July 2018 doi:10.20944/preprints201807.0276.v1

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

Table 4. Serverless architecture benefits and drawbacks (mainly compiled from [39])

Benefits	Drawbacks
RESOURCE EFFIENCY (service side)	
- auto-scaling based on event stimulus - reduced operational costs - scale to zero capability (no always-on)	- maximum function runtime is limited - startup latencies of functions must be considered - function runtime variations - functions can not preserve a state across function calls - external state (cache, key/value stores, etc.) can compensate this but is a magnitude slower - double spending problems (FaaS functions call other FaaS functions)
OPERATION (service side)	
- simplified deployment - simplified operation (see auto-scaling)	- increased attack surfaces - each endpoint introduces possible vulnerabilities - missing protective barrier of a monolithic server application - parts of the application logic are shifted to the client-side (that is not under control of the service provider) - increased vendor lock-in (currently no FaaS standards for API gateways and FaaS runtime environments)
DEVELOPMENT SPEED (service side)	
- development speed - simplified unit testing of stateless FaaS functions - better time to market	- increased client complexity - application logic is shifted to the client-side - code replication on client side across client platforms - control of application workflow on client side to avoid double-sending problems of FaaS computing - increased integration testing complexity - missing integration test tool-suites

- Endpoints of very service specific functionality is provided via API gateways. So, HTTP- and REST-based/REST-like communication protocols are generally preferred.
- Only very domain or service specific functions are provided on FaaS platforms. Mainly when this functionality is security relevant and should be executed in a controlled runtime environment by the service provider, or the functionality is too processing or data-intensive to be executed on consumer clients or edge devices, or the functionality is so domain-, problem-, or service-specific that simply no external 3rd party service exists.

Finally, the reader might observe the trend in serverless architectures that this kind of architecture is more decentralized and distributed, makes more intentional use of independently provided services, and is therefore much more intangible (more cloudy) compared with microservice architectures.

6. The road ahead

So far, we have identified and investigated two major trends. First, cloud computing and its related application architecture evolution can be seen as a steady process to optimize resource utilization in cloud computing. This was visualized in Figure 4 and discussed in Section 5.1. Second, in Section 5.2 it was emphasized that this resource utilization improvements resulted over time in an architectural evolution how cloud applications are being build and deployed. We observed a shift from monolithic SOA, via independently deployable microservices towards so called serverless architectures that are more decentralized and distributed, and make more intentional use of independently provided services.

The question is, whether and how are these trends continuing? To forecast the future is difficult, but having current trends and the assumption that these trends will go on to some degree makes it a bit easier. This is done in Section 6.1 for the optimization of resource utilization trend, and Section 6.2

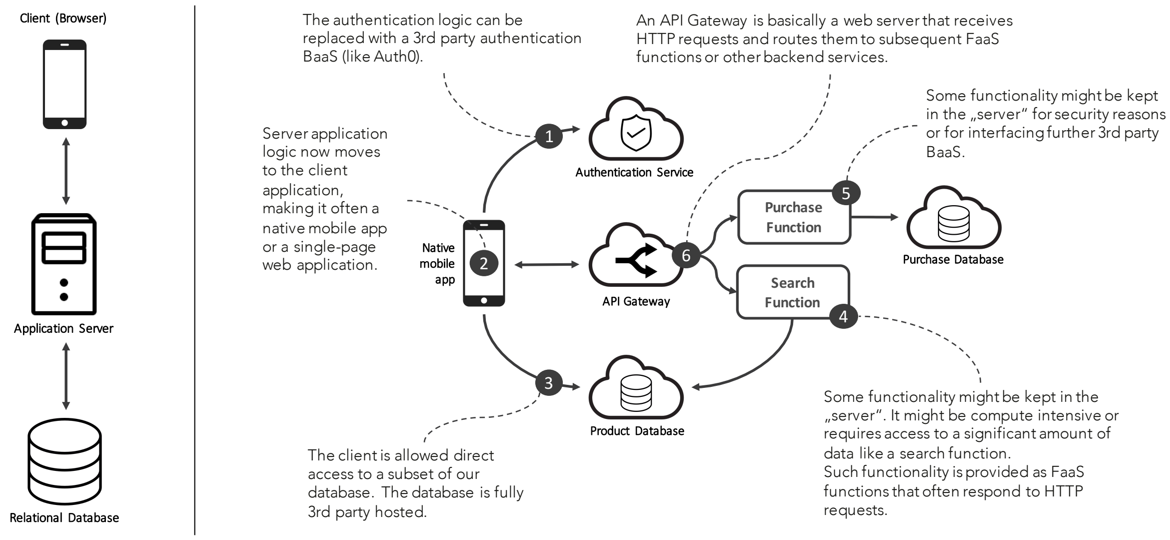

Figure 9. Serverless architectures result in a different and less centralized composition of application components and backend services compared with classical tiered application architectures.

will take a look how cloud application architectures may evolve in the future simply by extrapolating the existing SOA-microservice-serverless path.

6.1. Unikernels - the overlooked deployment unit?

The resource utilization optimization trend has been massively influenced by operating system virtualization based container technologies. However, containers are not about virtualization from a cloud application deployment point of view. They are about a standardized and self-contained way to define deployment units. But are containers the only solution and the most resource efficient solution already existing? The answer is no, and roads ahead might follow directions with the same intent to define standardized and self-contained deployment units but with a better resource utilization.

One option would be unikernels. A unikernel is a specialized, single address space machine image constructed via library operating systems. The first such systems were Exokernel (MIT Parallel and Distributed Operating Systems group) and Nemesis (University of Cambridge, University of Glasgow, Swedish Institute of Computer Science and Citrix Systems) in the late 1990s. The basic idea is, that a developer selects a minimal set of libraries which correspond to the OS constructs required for their application to run. These libraries are then compiled with the application and configuration code to build sealed, fixed-purpose images (unikernels) which run directly on a hypervisor or hardware without an OS. So, unikernels are self-contained deployment units like containers we investigated in Section 5.1.2 with the advantage to avoid a container overhead, a container runtime engine, and a host operating system (see Figure 5). So, interesting aspects to investigate on the road ahead would be:

- Because unikernels make operating systems and container runtime engines obsolete this could further increase resource utilization rates.
- FaaS platforms workers are normally container based. However unikernels are a deployment option as well. An interesting research and engineering direction would be, how to combine unikernels with FaaS platforms to apply the same time-sharing principles?

However, although there is research following the longterm trend to improve resource utilization [49,50], most cloud computing related unikernel research [51–54] mainly investigates unikernels as a security option to reduce attack surfaces (which are increased by serverless and microservice architectures as we have seen in Section 5.2). But the resource optimization effect of unikernels might be still not aware to cloud engineers. Other than container technology, unikernel technology is not hyped.

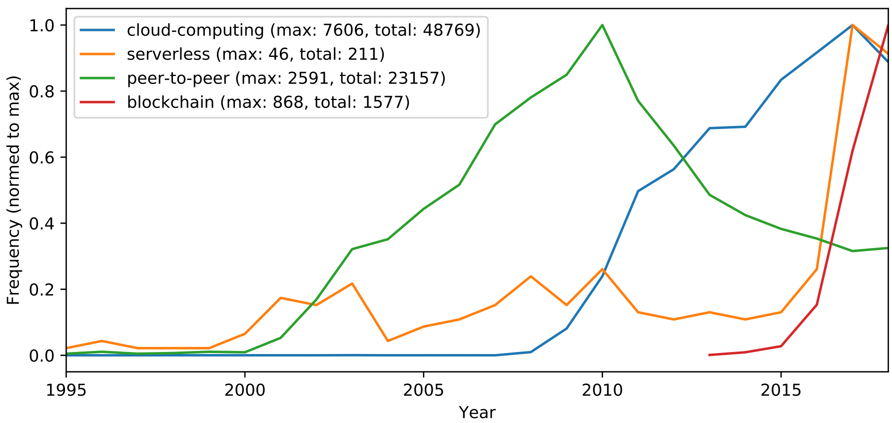

Figure 10. Trends of papers dealing with the terms cloud-computing, serverless, P2P, and blockchain (as latest P2P based trend). Retrieved from Scopus (limited to computer science), 2018 extrapolated.

6.2. Overcoming conceptual centralized approaches

This Section investigates some longterm trends in cloud and service computing research. This is done by support of a quantitative trend analysis. Scopus has been used to count the number of published papers dealing with some relevant terms over the years. This search has been limited to the computer science domain. The terms that have been searched in titles, abstracts, or keywords were:

- *Cloud computing* - to collect the amount of cloud computing related research in general.
- *SOA* - to collect the service computing related research which is still a major influencing concept in cloud computing.
- *Microservices* - to collect microservice related research (which is more modern and pragmatic interpretation of SOA and very popular in cloud computing).
- *Serverless* - to collect serverless architecture related research (which is the latest observable architecture trend in cloud computing).
- *Peer-to-peer* - to collect P2P related research (because recently more decentralizing concepts are entering cloud computing).
- *Blockchain* - to collect blockchain related research (which is the latest observable P2P related research trend/hype).

The presented architectural evolution can be seen as the perpetual fight of centralism and decentralism. Centralized architectures are known since decades. These kind of architectures make system engineering easier. Centralized architectures simply have less problems with data synchronization and data redundancy. They are easier to handle from a conceptual point of view. The client-server architecture is still one of the most basic but dominant centralized architectural style.

However, at various point in times centralized approaches are challenged by more decentralized approaches. Take the mainframe versus personal computer as one example dating back to the 1980's. Figure 10 shows the amount of papers per year for research that is dealing with cloud computing in general, and relates it with serverless architectures, P2P based related research (including blockchains as latest major P2P trend). We see a rise of interest in research about peer-to-peer (that means decentralized) approaches starting in 2000 that reached its peak in 2010. What is interesting, peer-to-peer based research decreased with the starting increase of cloud computing related research in 2008. So, cloud computing (mainly a concept to provide services in a conceptually centralized manner) decreased the interest in peer-to-peer related research. P2P computing is a distributed application architecture that partitions tasks or workloads between peers. Peers are equally privileged and equipotent participants in the application. Peers make a portion of their resources, such as processing

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

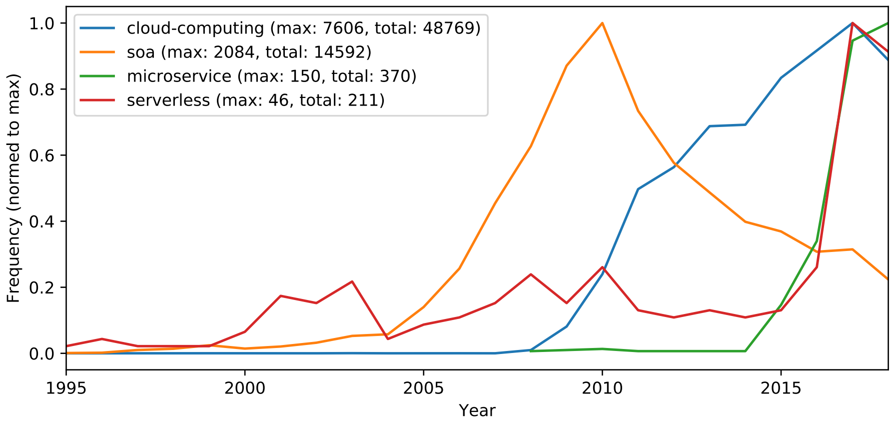

Figure 11. Trends of papers dealing with cloud-computing, SOA, microservices and serverless. Retrieved from Scopus (limited to computer science), 2018 extrapolated.

power, disk storage or network bandwidth, directly available to other network participants, without the need for central coordination by servers or stable hosts. So, peers are both suppliers and consumers of resources, in contrast to the cloud computing consumer-service model.

One astonishing curve in Figure 10 is the research interest in serverless solutions. Although on a substantial lower absolute level, a constant research interest in serverless solutions can be observed since 1995. To have "serverless" solutions seems to be a long standing dream in computer science. The reader should be aware that the notion of serverless changed over time. Serverless has been used until 2000 very often in file storage research contexts. With the rise of P2P based solutions it has been mainly used alongside P2P based approaches. And since 2015 it has been gained a lot of momentum alongside cloud-native application architectures (see Figure 11). So nowadays, it is mainly used in the notion described in Sections 5.1.3 and 5.2.2.

Figure 11 shows some further interesting correlation. With the rise of cloud computing in 2008 there is a steady decline in SOA related research. So, to deploy monolithic SOA applications in the cloud was not seen useful from the very beginning of cloud computing. However, it took almost five years in research that further and more cloud suited application architectures (microservice and serverless architectures) have been investigated.

If we look at the Figures 10 and 11 we see a decline of classical architecture approaches like SOA and an rising interest in new architecture styles like microservice and serverless architectures. It was already mentioned that especially serverless architectures come along with some decentralizing philosophy that is observable in P2P based research as well. The author does not think, that cloud application architectures will strive for the same level of decentralizing and distribution like peer-to-peer based approaches. But a more distributed service-to-service trend is clearly observable in cloud application architecture research [55]. So, the cloud computing trend started a decline in SOA (see Figure 11) and P2P (see Figure 10). But if we compare SOA and P2P (including blockcain related research), we see an increasing interest in decentralized solutions again (see Figure 12).

If we are taking all this together to forecast the road ahead, we could assume that service computing will be dominated by new architecture styles like microservices and serverless architectures. And SOA seems to die. But we see a resurgence of interest in decentralized approaches known from P2P related research. Therefore, the author assumes that especially serverless architectures will more and more evolve into cloud application architectures that follow distributed service-to-service principles (much more in the notion of peer-to-peer).

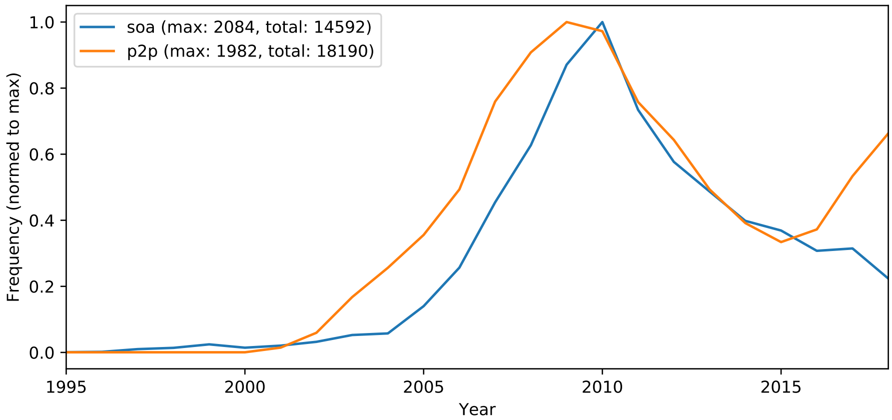

Figure 12. Trends of papers dealing with SOA, and P2P (including blockchain). Retrieved from Scopus (limited to computer science), 2018 extrapolated.

7. Related work

As far as the author knows, there is no survey that focused intentionally observable trends in cloud applications architectures over the last decade from a "big picture" architectural evolution point of view. This paper grouped that evolution mainly into the following point of views.

- Resource utilization optimization approaches like **containerization** and **FaaS** approaches have been investigated in Section 5.1.
- The architectural evolution of cloud applications that is dominated by **microservices** and evolving into **serverless architectures**. Both architectural styles have been investigated in Section 5.2.

For all of these four specific aspects (containerization, FaaS, microservices, serverless architectures) there exist surveys that should be considered by the reader. The studies and surveys [45,56–58] deal mainly with containerization and its accompanying resource efficiency. Although FaaS is quite young and could be only little reflected in research so far, there exist first survey papers [41,59,60] dealing with FaaS approaches deriving some open research questions regarding tool support, patterns for serverless solutions, enterprise suitability and whether serverless architectures will extend beyond traditional cloud platforms and architectures.

Service computing is quite established and there are several surveys on SOA related aspects [61–65]. However, more recent studies focus mainly microservices. [27,29,44] focus especially the architectural point of view and the relationship between SOA and microservices. All these papers are great to understand the current microservice "hype" better. It is highly recommended to study these papers. However, these papers are somehow bound to microservices and do not take the "big picture" of general cloud application architecture evolution into account. [29] provides a great overview on microservices and even serverless architectures, but serverless architectures are subsumed as a part of microservices to some degree. The author is not quite sure whether serverless architectures do not introduce fundamental new aspects into cloud application architectures that evolve from the "scale-to-zero" capability on the one hand and the unsolved function composition aspects (like the double spending problem) on the other hand. Resulting serverless architectures push former conceptually centralized service composing logic to end user and edge devices out of direct control of the service provider.

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

8. Conclusion

Two major trends in cloud application architecture have been identified and investigated. First, cloud computing and its related application architecture evolution can be seen as a steady process to optimize resource utilization in cloud computing. Unikernels – a technology from late 1990's – might be one option for future improvements. Like containers they are self-contained but avoid a container overhead, a container runtime engine, and even a host operating system. But astonishing little research is conducted in that field. Second, each resource utilization improvement resulted in an architectural evolution how cloud applications are being build and deployed. We observed a shift from monolithic SOA (machine virtualization), via independently deployable microservices (container) towards so called serverless architectures (FaaS function). Especially serverless architectures are more decentralized and distributed, and make more intentional use of independently provided services. What is more, service orchestration logic is shifted to end devices outside the direct scope of the service provisioning system.

So, service computing will be dominated by new architecture styles like microservice and serverless architectures. What is more, a resurgence of interest in decentralized approaches known from P2P related research is observable. That is astonishing because with the rise of cloud computing (and its centralized service provisioning concept) the research interest in peer-to-peer based approaches (and its decentralization philosophy) decreased. But this seems to change and might be an indicator where cloud computing could be heading in the future. Baldini et al. [41] asked the interesting question, whether serverless extend beyond traditional cloud platforms. If we are looking at the trends investigated in Section 6.2 this seems likely. Modern cloud applications might loose clear boundaries and could evolve into something that could be named *service-meshes*. Such service-meshes would be composed of small and fine-grained services provided by different and independent providers. And the service composition and orchestration might be done by mobile and edge devices not explicitly belonging to the service provisioning system anymore. This path might have already started with FaaS and serverless architectures. This all sounds astonishing familiar. In the 1960s the Internet was designed to be – decentralized and distributed.

Funding: This research was funded by German Federal Ministry of Education and Research under grant number 13FH021PX4 (Project CloudTRANSIT).

Acknowledgments: I would like to thank **Peter-Christian Quint**, (Lübeck University of Applied Sciences, Germany), **Dirk Reimers** (buchhalter.pro GmbH, Lübeck, Germany), **Derek Palme** (fat IT solutions GmbH, Kiel, Germany), **Thomas Finnern** (wilhelm.Tel GmbH, Stadtwerke Norderstedt, Germany), **René Peinl** (Hof University of Applied Sciences, Germany), **Bob Duncan** (University of Aberdeen, UK), **Magnus Westerlund** (Arcada University of Applied Sciences, Helsinki, Finland), and **Josef Adersberger** (QAWare GmbH, Munich, Germany) for their direct or indirect contributions to our research. Without their hard work, their inspiring ideas, their practitioner awareness, or their outside-the-box-thinking this contribution would not have been possible.

Conflicts of Interest: The author declares no conflict of interest. The founding sponsor had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

Peer-reviewed version available at Appl. Sci. 2018, 8, 1368; doi:10.3390/app8081368

AMQP	Advanced Message Queing Protocol
API	Application Programming Interface
GCE	Google Compute Engine
CDMI	Cloud Data Management Interface
CIMI	Cloud Infrastructure Management Interface
CNA	Cloud-native Application
DLT	Distributed Ledger Technology (aka blockchain)
IaaS	Infrastructure as a Service
FaaS	Function as a Service
HTTP	Hypertext Transfer Protocol
OCI	Open Container Initiative
OCCI	Open Cloud Computing Interface
OVF	Open Virtualization Format
OS	Operating System
P2P	Peer-to-Peer
PaaS	Platform as a Service
REST	Representational State Transfer
SaaS	Software as a Service
SOA	Service-Oriented Architecture
SOC	Service-Oriented Computing
TOSCA	Topology and Orchestration Specification for Cloud Applications
UCAML	Unified Cloud Application Modeling Language
VM	Virtual Machine
WS-BPEL	Web Service - Business Process Execution Language

References

1. Weinmann, J. Mathematical Proof if the Inevitability of Cloud Computing, 2011. last access 10/7/2018.
2. Mell, P.M.; Grance, T. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011.
3. Kratzke, N.; Quint, P.C. Preliminary Technical Report of Project CloudTRANSIT - Transfer Cloud-native Applications at Runtime. Technical report, Lübeck University of Applied Sciences, 2018. Preliminary technical report.
4. Kratzke, N.; Quint, P.C. Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study. *Journal of Systems and Software* **2017**, *126*, 1–16. doi:10.1016/j.jss.2017.01.001.
5. Kratzke, N.; Peinl, R. ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects. 2016 IEEE 20th Int. Enterprise Distributed Object Computing Workshop (EDOCW), 2016, pp. 1–10. doi:10.1109/EDOCW.2016.7584353.
6. Kratzke, N. Smuggling Multi-Cloud Support into Cloud-native Applications using Elastic Container Platforms. Proceedings of the 7th Int. Conf. on Cloud Computing and Services Science (CLOSER 2017), 2017, pp. 29–42.
7. Kratzke, N. About the Complexity to Transfer Cloud Applications at Runtime and how Container Platforms can Contribute? In *Cloud Computing and Services Science (revised selected papers)*; Helfert, M.; Ferguson, D.; Munoz, V.M.; Cardoso, J., Eds.; Communications in Computer and Information Science (CCIS), Springer, 2018. to be published.
8. Quint, P.C.; Kratzke, N. Towards a Description of Elastic Cloud-native Applications for Transferable Multi-Cloud-Deployments. Proceedings of the 1st Int. Forum on Microservices (Microservices 2017, Odense, Denmark), 2017. Book of extended abstracts.
9. Quint, P.C.; Kratzke, N. Towards a Lightweight Multi-Cloud DSL for Elastic and Transferable Cloud-native Applications. Proceedings of the 8th Int. Conf. on Cloud Computing and Services Science (CLOSER 2018, Madeira, Portugal), 2018.

10. Aderaldo, C.M.; Mendonça, N.C.; Pahl, C.; Jamshidi, P. Benchmark Requirements for Microservices Architecture Research. Proc. of the 1st Int. Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering; IEEE Press: Piscataway, NJ, USA, 2017; ECASE '17, pp. 8–13. doi:10.1109/ECASE.2017..4.
11. OASIS. Advanced Message Queueing Protocol (AQMP), Version 1.0, 2011.
12. Kratzke, N. Lightweight Virtualization Cluster - Howto overcome Cloud Vendor Lock-in. *Journal of Computer and Communication (JCC)* **2014**, *2*. doi:10.4236/jcc.2014.212001.
13. Kratzke, N.; Quint, P.C.; Palme, D.; Reimers, D. Project Cloud TRANSIT - Or to Simplify Cloud-native Application Provisioning for SMEs by Integrating Already Available Container Technologies. In *European Project Space on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges*; Kantere, V.; Koch, B., Eds.; SCITEPRESS, 2016.
14. Hogan, M.; Fang, L.; Sokol, A.; Tong, J. Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol, Version 2.0.0c, 2015.
15. Nyren, R.; Edmonds, A.; Papaspyrou, A.; Metsch, T. Open Cloud Computing Interface (OCCI) - Core, Version 1.1, 2011.
16. Metsch, T.; Edmonds, A. Open Cloud Computing Interface (OCCI) - Infrastructure, Version 1.1, 2011.
17. SNIA. Cloud Data Management Interface (CDMI), Version 1.1, 2015.
18. System Virtualization, Partitioning, and Clustering Working Group. Open Virtualization Format Specification, Version 2.1.0, 2015.
19. OCI. Open Container Initiative, 2015. last access 2016-02-04.
20. OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA), Version 1.0, 2013.
21. Opara-Martins, J.; Sahandi, R.; Tian, F. Critical review of vendor lock-in and its impact on adoption of cloud computing. Int. Conf. on Information Society (i-Society 2014), 2014, pp. 92–97. doi:10.1109/i-Society.2014.7009018.
22. Ashtikar, S.; Barker, C.; Clem, B.; Fichadia, P.; Krupin, V.; Louie, K.; Malhotra, G.; Nielsen, D.; Simpson, N.; Spence, C. OPEN DATA CENTER ALLIANCE Best Practices: Architecting Cloud-Aware Applications Rev. 1.0. Technical report, 2014.
23. Bohn, R.B.; Messina, J.; Liu, F.; Tong, J.; Mao, J. NIST Cloud Computing Reference Architecture. World Congr. on Services (SERVICES 2011); IEEE Computer Society: Washington, DC, USA, 2011; pp. 594–596. doi:10.1109/SERVICES.2011.105.
24. Quint, P.C.; Kratzke, N. Overcome Vendor Lock-In by Integrating Already Available Container Technologies - Towards Transferability in Cloud Computing for SMEs. Proceedings of CLOUD COMPUTING 2016 (7th. International Conference on Cloud Computing, GRIDS and Virtualization), 2016.
25. Newman, S. *Building Microservices*; O'Reilly Media, Incorporated, 2015.
26. Namiot, D.; Sneps-Sneppe, M. On micro-services architecture. *Int. Journal of Open Information Technologies* **2014**, *2*.
27. Cerny, T.; Donahoo, M.J.; Pechanec, J. Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems. Proceedings of the International Conference on Research in Adaptive and Convergent Systems - RACS '17, 2017. doi:10.1145/3129676.3129682.
28. Taibi, D.; Lenarduzzi, V.; Pahl, C. Architectural Patterns for Microservices: a Systematic Mapping Study. 8th International Conference on Cloud Computing and Services Science (CLOSER'18), 2018, number March.
29. Jamshidi, P.; Pahl, C.; Mendonça, N.C.; Lewis, J.; Stefan Tilkov, T. Microservices The Journey So Far and Challenges Ahead.
30. Balalaie, A.; Heydarnoori, A.; Jamshidi, P. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software* **2016**, [1606.04036]. doi:10.1109/MS.2016.64.
31. Jabbari, R.; bin Ali, N.; Petersen, K.; Tanveer, B. What is DevOps? A Systematic Mapping Study on Definitions and Practices. 2016. doi:10.1145/2962695.2962707.
32. Bergmayr, A.; Breitenbücher, U.; Ferry, N.; Rossini, A.; Solberg, A.; Wimmer, M.; Kappel, G.; Leymann, F. A Systematic Review of Cloud Modeling Languages. *ACM Computing Surveys* **2018**, *51*, 39. doi:10.1145/3150227.
33. Adam Wiggins. The Twelve-Factor App, 2014. last access 2016-02-14.

34. Martin Fowler. Circuit Breaker, 2014. last access 2016-05-27.
35. Fehling, C.; Leymann, F.; Retter, R.; Schupeck, W.; Arbitter, P. *Cloud Computing Patterns*; Springer, 2014.
36. Erl, T.; Cope, R.; Naserpour, A. *Cloud Computing Design Patterns*; Springer, 2015.
37. Verma, A.; Pedrosa, L.; Korupolu, M.; Oppenheimer, D.; Tune, E.; Wilkes, J. Large-scale cluster management at Google with Borg. *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15* **2015**, pp. 1–17. doi:10.1145/2741948.2741964.
38. Hindman, B.; Konwinski, A.; Zaharia, M.; Ghodsi, A.; Joseph, A.D.; Katz, R.; Shenker, S.; Stoica, I. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation; USENIX Association: Berkeley, CA, USA, 2011; NSDI'11, pp. 295–308.
39. Mike Roberts. Serverless Architectures, 2016.
40. Baldini, I.; Cheng, P.; Fink, S.J.; Mitchell, N.; Muthusamy, V.; Rabbah, R.; Suter, P.; Tardieu, O. The serverless trilemma: function composition for serverless computing. Proc. of the 2017 ACM SIGPLAN Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017, 2017, [1611.02756]. doi:10.1145/3133850.3133855.
41. Baldini, I.; Castro, P.; Chang, K.; Cheng, P.; Fink, S.; Ishakian, V.; Mitchell, N.; Muthusamy, V.; Rabbah, R.; Slominski, A.; Suter, P., Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*; Springer Singapore: Singapore, 2017; pp. 1–20. doi:10.1007/978-981-10-5026-8_1.
42. Martin Fowler. Microservices - A Definition of this new Architectural Term, 2014. last access 2016-05-27.
43. Villamizar, M.; Garcés, O.; Ochoa, L.; Castro, H.; Salamanca, L.; Verano, M.; Casallas, R.; Gil, S.; Valencia, C.; Zambrano, A.; Lang, M. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications* **2017**. doi:10.1007/s11761-017-0208-y.
44. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L., Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*; Mazzara, M.; Meyer, B., Eds.; Springer International Publishing: Cham, 2017; pp. 195–216. doi:10.1007/978-3-319-67425-4_12.
45. Pahl, C.; Brogi, A.; Soldani, J.; Jamshidi, P. Cloud Container Technologies: a State-of-the-Art Review. *IEEE Transactions on Cloud Computing* **2017**, pp. 1–1. doi:10.1109/TCC.2017.2702586.
46. Hindman, B.; Konwinski, A.; Zaharia, M.; Ghodsi, A.; Joseph, A.D.; Katz, R.; Shenker, S.; Stoica, I. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation; USENIX Association: Berkeley, CA, USA, 2011; NSDI'11, pp. 295–308.
47. Pérez, A.; Moltó, G.; Caballer, M.; Calatrava, A. Serverless computing for container-based architectures. *Future Generation Computer Systems* **2018**, *83*, 50 – 59. doi:https://doi.org/10.1016/j.future.2018.01.022.
48. Baldini, I.; Castro, P.; Cheng, P.; Fink, S.; Ishakian, V.; Mitchell, N.; Muthusamy, V.; Rabbah, R.; Suter, P. Cloud-native, event-based programming for mobile applications. Proc. of the Int. Conf. on Mobile Software Engineering and Systems. ACM, 2016, pp. 287–288.
49. Cozzolino, V.; Ding, A.Y.; Ott, J. FADES: fine-grained edge offloading with unikernels. Proc. of the Workshop on Hot Topics in Container Networking and Networked Systems. ACM, 2017, pp. 36–41.
50. Koller, R.; Williams, D. Will Serverless End the Dominance of Linux in the Cloud? Proceedings of the 16th Workshop on Hot Topics in Operating Systems. ACM, 2017, pp. 169–173.
51. Bratterud, A.; Happe, A.; Duncan, R.A.K. Enhancing cloud security and privacy: the Unikernel solution. 8th Int. Conf. on Cloud Computing, GRIDs, and Virtualization, 2017.
52. Happe, A.; Duncan, B.; Bratterud, A. Unikernels for cloud architectures: how single responsibility can reduce complexity, thus improving enterprise cloud security. *Submitt. to Complexis* **2017**, *2016*, 1–8.
53. Duncan, B.; Happe, A.; Bratterud, A. Cloud Cyber Security: Finding an Effective Approach with Unikernels. *SECURITY IN COMPUTING AND COMMUNICATIONS* **2017**, p. 31.
54. Compastié, M.; Badonnel, R.; Festor, O.; He, R.; Lahlou, M.K. Unikernel-based Approach for Software-Defined Security in Cloud Infrastructures. NOMS 2018-IEEE/IFIP Network Operations and Management Symposium, 2018.
55. Westerlund, M.; Kratzke, N. Towards Distributed Clouds - A review about the evolution of centralized cloud computing, distributed ledger technologies, and a foresight on unifying opportunities and security

implications. Proc. of the 16th Int. Conf. on High Performance Computing and Simulation (HPCS 2018), 2018.

56. Kaur, T.; Chana, I. Energy Efficiency Techniques in Cloud Computing: A Survey and Taxonomy. *ACM Comput. Surv.* **2015**, *48*, 22:1–22:46. doi:10.1145/2742488.
57. Tosatto, A.; Ruiu, P.; Attanasio, A. Container-Based Orchestration in Cloud: State of the Art and Challenges. 2015 Ninth Int. Conf. on Complex, Intelligent, and Software Intensive Systems, 2015, pp. 70–75. doi:10.1109/CISIS.2015.35.
58. Peinl, R.; Holzschuher, F.; Pfitzer, F. Docker Cluster Management for the Cloud - Survey Results and Own Solution.
59. Spillner, J. Practical Tooling for Serverless Computing. Proc. of the10th Int. Conf. on Utility and Cloud Computing; ACM: New York, NY, USA, 2017; UCC '17, pp. 185–186. doi:10.1145/3147213.3149452.
60. Lynn, T.; Rosati, P.; Lejeune, A.; Emeakaroha, V. A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms. 2017 IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom), 2017, pp. 162–169. doi:10.1109/CloudCom.2017.15.
61. Huhns, M.N.; Singh, M.P. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing* **2005**, *9*, 75–81.
62. Dustdar, S.; Schreiner, W. A survey on web services composition. *Int. Journal of Web and Grid Services* **2005**, *1*, 1–30.
63. Papazoglou, M.P.; Traverso, P.; Dustdar, S.; Leymann, F. Service-Oriented Computing: State of the Art and Research Challenges. *Computer* **2007**, *40*, 38–45. doi:10.1109/MC.2007.400.
64. Papazoglou, M.P.; van den Heuvel, W.J. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal* **2007**, *16*, 389–415. doi:10.1007/s00778-007-0044-3.
65. Razavian, M.; Lago, P. A Survey of SOA Migration in Industry. ICSOC, 2011.