Aleksi Pekkala

Migrating a web application to serverless architecture

Master's Thesis in Information Technology

October 23, 2018

University of Jyväskylä

Department of Mathematical Information Technology

Author: Aleksi Pekkala

Contact information: alvianpe@student.jyu.fi

Supervisor: Oleksiy Khriyenko

Title: Migrating a web application to serverless architecture

Työn nimi: Web-sovelluksen siirtäminen serverless-arkkitehtuuriin

Project: Master's Thesis

Study line: Master's Thesis in Information Technology

Page count: 47+0

Abstract: This document is a sample gradu3 thesis document class document. It also functions as a user manual and supplies guidelines for structuring a thesis document.

The abstact is typically short and discusses the background, the aims, the research methods, the obtained results, the interpretation of the results and the conculsions of the thesis. It should be so short that it, the Finnish translation, and all other meta information fit on the same page.

The Finnish tiivistelmä of a thesis should usually say exactly the same things as the abstract.

Keywords: serverless, FaaS, architecture, cloud computing, web applications

Suomenkielinen tiivistelmä: Tämä kirjoitelma on esimerkki siitä, kuinka gradu3-tutkielmapohjaa käytetään. Se sisältää myös käyttöohjeet ja tutkielman rakennetta koskevia ohjeita.

Tutkielman tiivistelmä on tyypillisesti lyhyt esitys, jossa kerrotaan tutkielman taustoista, tavoitteesta, tutkimusmenetelmistä, saavutetuista tuloksista, tulosten tulkinnasta ja johtopäätöksistä. Tiivistelmän tulee olla niin lyhyt, että se, englanninkielinen abstrakti ja muut metatiedot mahtuvat kaikki samalle sivulle.

Sen tulee kertoa täsmälleen samat asiat kuin englannikielinen abstrakti.

Avainsanat: serverless, FaaS, arkkitehtuuri, pilvilaskenta, web-sovellukset

List of Figures

List of Listings

Contents

1 Introduction

Cloud computing has in the past decade emerged as a veritable backbone of modern economy, driving innovation both in industry and academia as well as enabling scalable global enterprise applications. Just as the adoption of cloud computing continues to increase, the technologies in which the paradigm is based on have continued to progress. Recently the development of novel virtualization techniques has lead to the introduction of *serverless computing*, an architectural pattern based on ephemeral cloud resources that scale up and down automatically and are billed for actual usage at a millisecond granularity. The main drivers behind serverless computing are both reduced operational costs through more efficient cloud resource utilization and improved developer productivity by shifting provisioning, load balancing and other infrastructure concerns to the platform. (Buyya et al. 2017)

As an appealing economic proposition, serverless computing has attracted significant interest in the industry. This is illustrated for example by its appearance in the 2017 Gartner Hype Technologies Report (Walker 2017). By now most of the prominent cloud service providers have introduced their own serverless platforms, promising capabilities that make writing scalable web services easier and cheaper (e.g. AWS 2018; Google 2018; IBM 2018; Microsoft 2018). A number of high-profile use cases have also been presented in the literature (CNCF 2018). Baldini, Castro, et al. (2017) however note a lack of corresponding degree of interest in academia despite a wide variety of technologically challenging and intellectually deep problems in the space.

One of the open problems identified in literature concerns the discovery of serverless design patterns: how do we compose the granular building blocks of serverless into larger systems? (Baldini, Castro, et al. 2017) Varghese and Buyya (2018) contend that one challenge hindering the widespread adoption of serverless will be the radical shift in the properties that a programmer will need to focus on, from latency, scalability and elasticity to those relating to the modularity of an application. Considering this and the paradigm's unique characteristics and limitations, it's unclear to what extent our current patterns apply and what kind of new patterns are best suited to optimize for the features of serverless computing. The object of this thesis is to fill the gap by re-evaluating existing design patterns in the serverless context

and proposing new ones through an exploratory migration process.

1.1 Research problem

The research problem addressed by this thesis distills down to the following 4 questions:

1. Why should a web application be migrated to serverless?
2. What kind of patterns are there for building serverless web application backends?
3. Do the existing patterns have gaps or missing parts, and if so, can we come up with improvements or alternative solutions?
4. How does migrating a web application to serverless affect its quality?

The first two questions are addressed in the theoretical part of the thesis. Question 1 concerns the motivation behind the thesis and introduces serverless migration as an important and relevant business problem. Question 2 is answered by surveying existing literature for serverless patterns as well as other, more general patterns thought suitable for the target class of applications.

The latter questions form the constructive part of the thesis. Question 3 concerns the application and evaluation of surveyed patterns. The surveyed design patterns are used to implement a subset of an existing conventional web application in the serverless architecture. In case the patterns prove unsuitable for any given problem, alternative solutions or extensions are proposed. The last question consists of comparing the migrated portions of the app to the original version and evaluating whether the posited benefits of serverless architecture are in fact realized.

1.2 Outline

The thesis is structured as follows: the second chapter serves as an introduction to the concept of serverless computing. The chapter describes the main benefits and drawbacks of the platform, as well as touching upon its internal mechanisms and briefly comparing the main service providers. Extra emphasis is placed on how the platform's limitations should be taken into account when designing web application backends.

The third chapter consists of a survey into existing serverless design patterns and recommendations. Applicability of other cloud computing, distributed computing and enterprise integration patterns is also evaluated.

The fourth chapter describes the process of migrating an existing web application to serverless architecture. The patterns discovered in the previous chapter are utilized to implemented various typical web application features on a serverless platform. In cases where existing patterns prove insufficient or unsuitable as per the target application's characteristics, modifications or new patterns are proposed.

The outcome of the migration process is evaluated in the fifth chapter. The potential benefits and drawbacks of the serverless platform outlined in chapter 2 are used to reflect on the final artifact. The chapter includes approximations on measurable attributes such as hosting costs and performance as well as discussion on the more subjective attributes like maintainability and testability. The overall ease of development – or developer experience – is also addressed since it is one of the commonly reported pain points of serverless computing (Erwin van Eyk et al. 2017).

The final chapter of the thesis aims to draw conclusions on the migration process and the resulting artifacts. The chapter contains a summary of the research outcomes and ends with recommendations for further research topics.

2 Serverless computing

This chapter serves as an introduction to serverless computing. Defining serverless computing succinctly can be difficult because of the relative immaturity of the field. The NIST definitions of cloud computing have yet to catch up with the technology (Mell and Grance 2011), and an effort to formalize and standardize serverless computing by the industry-headed Cloud Native Computing Foundation is still underway (CNCF 2018) TODO whitepaper now released. As a result the boundaries between serverless and other cloud computing terms are still somewhat blurred, and the terms seem to carry slightly different meanings depending on the author or context. To complicate matters further, serverless computing has come to appear in two different but overlapping forms. A multilayered approach is therefore in order.

We approach the formidable task of defining serverless by first taking a brief look at the history and motivations behind utility computing. After that we'll introduce the basic tenets of serverless computing, distinguish between its two main approaches and see how it positions itself relative to other cloud service models. This is followed by a more technical look at the most recent serverless model, as well as its major providers, use cases, security issues and economic implications. The chapter closes with notes on the drawbacks and limitations of serverless, particularly from the point of view of web application backends. This thesis' definition of serverless leans heavily on the CNCF Serverless Working Group's whitepaper (CNCF 2018), the seminal introduction to the topic by Roberts (2016) as well as a number of recent survey articles (e.g. Baldini, Castro, et al. 2017; Erwin van Eyk et al. 2017; Fox et al. 2017)

As a sidenote, although earliest uses of the term 'serverless' can be traced back to peer-to-peer and client-only solutions (Fox et al. 2017), we're dismissing these references since the name has evolved into a completely different meaning in the current cloud computing context. As per Roberts (2016), first usages of the term referring to elastic cloud computing seem to have appeared at around 2012.

2.1 Background

Utility computing refers to a business model where computing resources, such as computation and storage, are commoditized and delivered as metered services in a similarly to physical public utilities such as water, electricity and telephony. Utilities are readily available to consumers at any time whenever required and billed per actual usage. In computing, this has come to mean on-demand access to highly scalable subscribtion-based IT resources. The availability of computing as an utility enables organizations to avoid investing heavily on building and maintaining complex IT infrastructure. (Buyya et al. 2009)

This vision of utility computing can be traced all the way back to 1961, with the computing pioneer John McCarthy predicting that "computation may someday be organized as a public utility" (Foster et al. 2008). Likewise in 1969 Leonard Kleinrock, one of the chieft scientists in the ARPANET project, is quoted as saying, "as of now, computer networks are still in their infancy, but as they grow up and become sophisticated, we will probably see the spread of 'computer utilities' which, like present electric and telephone utilities, will service individual homes and offices across the country" (Kleinrock 2003). The creation of the Internet first facilitated weaving computer resources together into large-scale distributed systems. Onset by this discovery, multiple computing paradigms have been proposed and adopted over the years to take on the role of a ubiquitous computing utility, including cluster, grid, peer-to-peer (P2P) and services computing (Buyya et al. 2009). The latest paradigm, cloud computing, has in the past decade revolutionized the computer science horizon and got us closer to computing as an utility than ever (Buyya et al. 2017).

Sareen (2013) succinctly defines the cloud as "a pool of virtualized computer resources". Foster et al. (2008) present a more thorough definition of cloud computing as "a large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet". Cloud computing builds on the earlier paradigm of grid computing, and relies on grid computing as its backbone and infrastructure. Compared to infrastructure-based grid computing, cloud computing focuses on more abstract resources and services. Buyya et al. (2017) also note that cloud computing differs from grid computing in that it promises virtually unlimited compu-

tational resources on demand.

The first cloud providers were born out of huge corporations offering their surplus computing resources as a service in order to offset expenses and improve utilization rates. Having set up global infrastructure to handle peak demand, a large part of the resources were left under-utilized at times of average demand. The providers are able to offer these surplus resources at attractive prices due to the large scale of their operations, benefiting from economies of scale. To address consumers' concerns about outages and other risks, cloud providers guarantee a certain level of service delivery through Service Level Agreements (SLA) that are negotiated between providers and consumers. (Youseff, Butrico, and Silva 2008)

The key technology that enables cloud providers to transparently handle consumers' requests without impairing their own processing needs is *virtualization*. Virtualization is one of the main components behind cloud computing and one of the factors setting it apart from grid computing. Sareen (2013) defines virtualization as using computer resources to imitate other computer resources or whole computers. This enables the abstraction of the underlying physical resources as a set of multiple logical virtual machines (VM). Virtualization has three characteristics that make it ideal for cloud computing: 1) *partitioning* supports running many applications and operating systems in a single physical system; 2) *isolation* ensures boundaries between the host physical system and virtual containers; 3) *encapsulation* enables packaging virtual machines as complete entities to prevent applications from interfering with each other.

Virtual machines manage to provide strong security guarantees by isolation, i.e., by allocating each VM its own set of resources with minimal sharing between the host system. Minimal sharing however translates into high memory and storage requirements as each virtual machine requires a full OS image in addition to the actual application files. A virtual machine also has to go through the standard OS boot process on startup, resulting in launching times measured in minutes. Rapid innovation in the cloud market and virtualization technologies has recently lead to an alternative, more lightweight *container*-based solution. Container applications share a kernel with the host, resulting in significantly smaller deployments and fast launching times ranging from less than a second to a few seconds. Due to resource sharing a single host is capable of hosting hundreds of containers simultaneously.

Differences in resource sharing between VM- and container-based deployment is illustrated in figure 1. As a downside containers lack the VM's strong isolation guarantee and the ability to run a different OS per deployment. On the other hand, containers provide isolation via namespaces, so processes inside containers are still isolated from each other as well as the host. *Containerization* has emerged as a common practice of packaging applications and related dependencies into standardized container images to ease development efficiency and interoperability. (Pahl 2015)

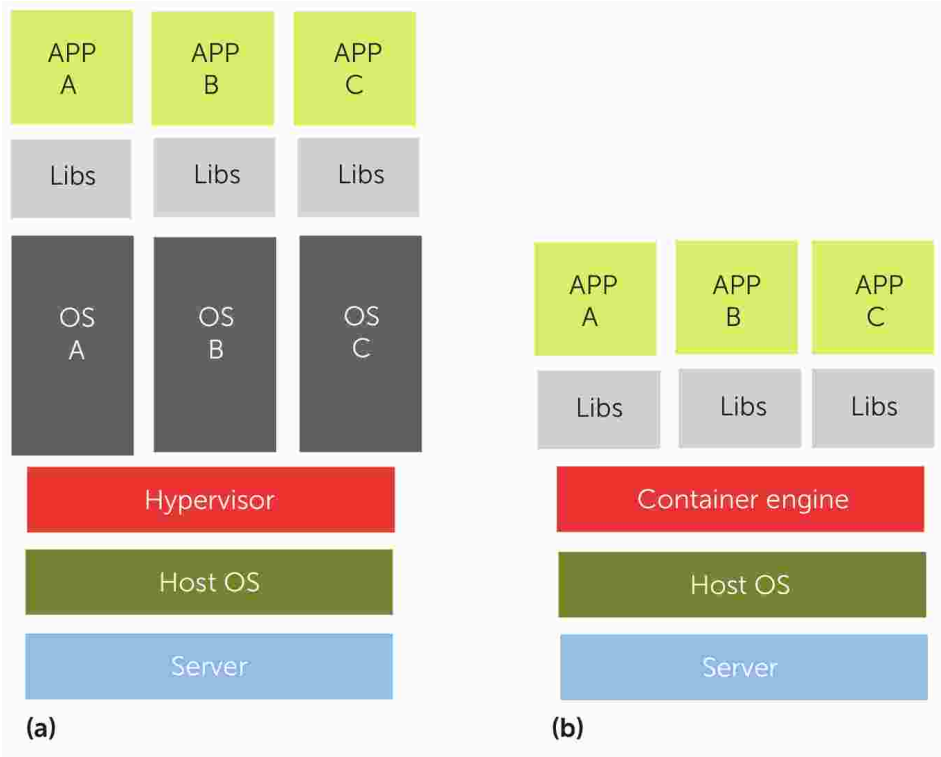

Figure 1. Comparison of a) virtual machine- and b) container-based deployments (Bernstein 2014)

Cloud computing is by now a well-established paradigm that enables organizations to flexibly deploy a wide variety of software systems over a pool of externally managed computing resources. Both major IT companies and startups see migrating on-premise legacy systems to the cloud as an opportunistic business strategy for gaining competetive advantage. Cost savings, scalability, reliability and efficient utilization of resources as well as flexibility are identified as key drivers for migrating applications to the cloud (Jamshidi, Ahmad, and Pahl 2013). However, although the state-of-the-art in cloud computing has advanced significantly

over the past decade, several challenges remain.

One of the open issues in cloud computing concerns pricing models. In the current cloud service models pricing typically follows the "per instance per hour" model; that is, the consumer is charged for the duration that an application is hosted on a VM or a container (Varghese and Buyya 2018). The flaw in this model is that idle time is not taken into account. Whether the application was used or not doesn't have an effect: the consumer ends up paying for the whole hour even if the application was actually performing computation for just a couple of seconds. This makes sense from the provider's point of view, since for the duration billed, the instance is provisioned and dedicated solely to hosting the consumer's application. However, paying for idle time is of course undesirable for the consumer, and the problem is made worse in case of applications with fluctuating and unpredictable workloads.

Continuously hosting non-executing applications is problematic on the provider side as well as it leads to under-utilization. Just as consumers end up paying for essentially nothing, providers end up provisioning and tying up resources to do essentially nothing. Fundamentally the problem of under-utilization boils down to elasticity and resource management. The current cloud computing models are incapable of automatically scaling up and down to meet current demand while at the same time maintaining their stringent Quality-of-Service (QoS) expectations (Buyya et al. 2017). Lacking automatic scaling mechanisms, cloud consumers are left to make capacity decisions on their own accord, and as Roberts (2016) notes, consumers typically err on the side of caution and over-provision. This inin turn leads to inefficiencies and under-utilization as described above.

The problem of low utilization rates in data centers is particularly relevant in the current energy-constrained environment. ICT in general consumes close to 10% of all electricity world-wide, with the CO_2 impact comparable to air travel (Buyya et al. 2017). It's estimated that in 2010 data centers accounted for 1-2% of global energy usage, with data center carbon emissions growing faster than the annual global footprint as well as the footprint of other ICT subcategories. While data centers are improving in energy efficiency, so is the demand for computing services with both the magnitude of data produced and complexity of software increasing. Operational factors such as excessive redundancy also affect data center energy efficiency heavily. A survey of Google data centers – considered to represent the higher end

of utilization – revealed utilization of 60% or less 95% of the time and 30% or less half of the time. Another analysis found that data centers spend on average only 6% to 12% of the electricity powering servers that do computation, with the rest used to keep servers idling for redundancy. (Horner and Azevedo 2016)

Cloud computing, having "revolutionized the computer science horizon and enabled the emergence of computing as the fifth utility" (Buyya et al. 2017), will face considerable new requirements in the coming decade. It's predicted that by 2020 over 20 billion sensor-rich devices like phones and wearables will be connected to the Internet generating trillions of gigabytes of data. Varghese and Buyya (2018) argue that increasing volumes of data pose significant networking and computing challenges that cannot be met by existing cloud infrastructure, and that adding more centralized cloud data centers will not be enough to address the problem. The authors instead call for new computing models beyond conventional cloud computing, one of which is serverless computing.

2.2 Defining serverless

TODO: link to previous section, present serverless as in many ways the next reductive step in IaaS to PaaS abstraction. Explain figure 2 (E. van Eyk et al. 2018).

Fundamentally serverless computing is about building and running back-end code that does not require server management or server applications. The term itself can seem disingenuous, since serverless computing obviously still involves servers. The name – coined by industry – instead carries the meaning that resources used by the application are managed by the cloud service provider. As tasks such as provisioning, maintenance and capacity planning are outsourced to the serverless platform, developers are left to focus on application logic. For the cloud customer this provides an abstraction where computation is disconnected from the infrastructure it runs on. (Roberts 2016; CNCF 2018)

Erwin van Eyk et al. (2017) further define serverless computing by three key characteristics:

1. Granular billing: the user of a serverless model is charged only when the application is actually executing

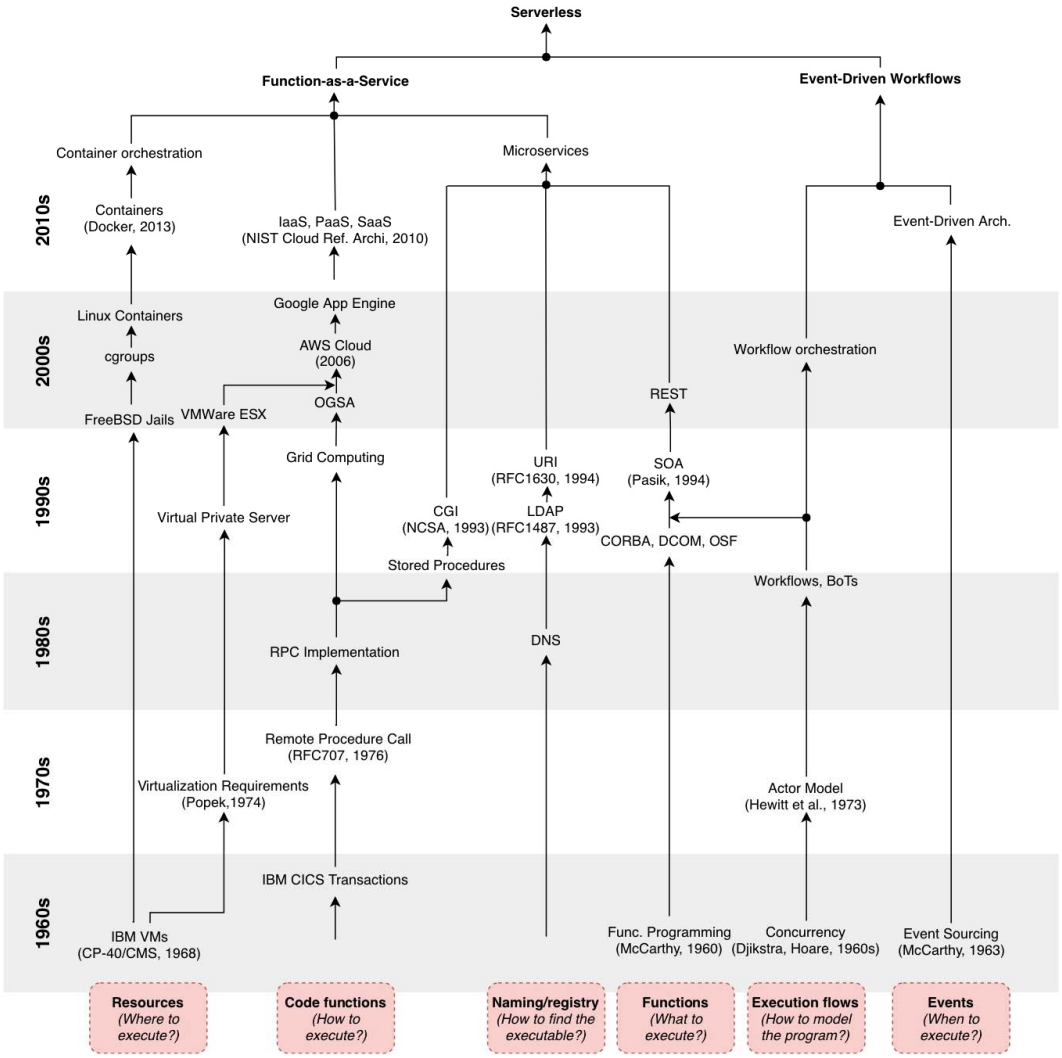

Figure 2. A history of computer science concepts leading to serverless computing (E. van Eyk et al. 2018)

2. (Almost) no operational logic: operational logic, such as resource management and autoscaling, is delegated to the infrastructure, making those concerns of the infrastructure operator
3. Event-driven: interactions with serverless applications are designed to be short-lived, allowing the infrastructure to deploy serverless applications to respond to events, so only when needed

TODO: link to SOA and microservices: “We see the emerging model based on running individual cloud functions as a consequence of the slow but sustained evolution of computing.

For many decades we have witnessed a transition from relatively large, monolithic applications, to smaller or more structured applications with smaller execution units (e.g, workflows with many small tasks). This transition is captured qualitatively by various software architectures, for example, the Service-Oriented Architecture, and quantitatively by various workload-characterization and modeling studies, for example of scientific workloads running on supercomputers and grids between 1990 and 2010." (Erwin van Eyk et al. 2017) Also Hammond and Rymer (2016) with "Microservices lead to a serverless" and Villamizar et al. (2016) on SOA.

TODO: link to the event-driven paradigm: "Serverless computing is a partial realization of an event-driven ideal, in which applications are defined by actions and the events that trigger them. This language is reminiscent of active database systems, and the event-driven literature has theorized for some time about general computing systems in which actions are processed reactively to event streams. Serverless function platforms fully embrace these ideas, defining actions through simple function abstractions and building out event processing logic across their clouds." (Schmidt, Anicic, and Stühmer 2008)

Serverless computing has in effect come to encompass two distinct cloud computing models: Backend-as-a-Service (BaaS) as well as Function-as-a-Service (FaaS). The two serverless models, while different in operation as explained below, are grouped under the same serverless umbrella since they deliver the same main benefits: zero server maintenance overhead and elimination of idle costs. (CNCF 2018)

Backend-as-a-Service refers to an architecture where an application's server-side logic is replaced with external, fully managed cloud services that carry out various tasks like authentication or database access (Buyya et al. 2017). The model is typically utilized in the mobile space to avoid having to manually set up and maintain server resources for the more narrow back-end requirements of a mobile application. In the mobile context this form of serverless computing is also referred to as Mobile-Backend-as-a-Service or MBaaS (Sareen 2013). An application's core business logic is implemented client-side and integrated tightly with third party remote application services. Since these API-based BaaS services are managed transparently by the cloud service provider, the model appears to the developer as serverless.

Function-as-a-Service is defined in a nutshell as “a style of cloud computing where you write code and define the events that should cause the code to execute and leave it to the cloud to take care of the rest” (Gannon, Barga, and Sundaresan 2017). In the FaaS architecture an application’s business logic is still located server-side. The crucial difference is that instead of self-managed server resources, developers upload small units of code to a FaaS platform that executes the code in short-lived, stateless compute containers in response to events (Roberts 2016). The model appears serverless in the sense that the developer has no control over the resources on which the back-end code runs. Albuquerque Jr et al. (2017) note that the BaaS model of locating business logic on the client side carries with it some complications, namely difficulties in updating and deploying new features as well as reverse engineering risks. FaaS circumvents these problems by retaining business logic server-side.

Out of the two serverless models FaaS is a more recent development: the first commercial FaaS platform, AWS Lambda, was introduced in November 2014 (AWS 2018). FaaS is also the model with significant differences to traditional web application architecture (Roberts 2016). These differences and their implications are further illustrated in section 2.4. As the more novel architecture, FaaS is especially relevant to the research questions in hand and is thus paid more attention to in the remainder of this thesis.

Another perspective on the difference between the two serverless models is to view BaaS as a more tailored, vendor-specific approach to FaaS (Erwin van Eyk et al. 2017). Whereas BaaS-type services function as built-in components for many common use cases such as user management and data storage, a FaaS platform allows developers to implement more customized functionality. BaaS plays an important role in serverless architectures as it will often be the supporting infrastructure (e.g. in form of data storage) to the stateless FaaS functions (CNCF 2018). Conversely, in case of otherwise BaaS-based applications there’s likely still a need for custom server-side functionality; FaaS functions may be a good solution for this (Roberts 2016). Serverless applications can utilize both models simultaneously, with BaaS platforms generating events that trigger FaaS functions, and FaaS functions acting as a ’glue component’ between various third party BaaS components. Roberts (2016) also notes convergence in the space, giving the example of the user managemement provider Auth0 starting initially with a BaaS-style offering but later entering the FaaS space with a ’Auth0

Webtask' service.

It's worth noting that not all authors follow this taxonomy of FaaS and BaaS as the two subcategories of a more abstract serverless model. Baldini, Castro, et al. (2017) explicitly raise the question on whether serverless is limited to FaaS or broader in scope, identifying the boundaries of serverless as an open question. Some sources (Hendrickson et al. 2016; McGrath and Brenner 2017; Varghese and Buyya 2018, among others) seem to strictly equate serverless with FaaS, using the terms synonymously. Considering however that the term 'serverless' predates the first FaaS platforms by a couple of years (Roberts 2016), it seems sensible to at least make a distinction between serverless and FaaS. In this thesis we'll stick to the CNCF (2018) definition as outlined above.

2.3 Comparison to other cloud computing models

Another approach to defining serverless is to compare it with other cloud service models. The commonly used NIST definition divides cloud offerings into three categories: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS), in order of increasing degree of abstraction of cloud infrastructure (Mell and Grance 2011). On this spectrum serverless computing positions itself in the space between PaaS and SaaS, as illustrated in figure 3 (Baldini, Castro, et al. 2017). Figure 4 illustrates how the two serverless models relate, with the cloud provider taking over a larger share of operational logic in BaaS. Erwin van Eyk et al. (2017) note that there's some overlap and give examples of non-serverless products in both the PaaS and SaaS worlds that nonetheless exhibit the main characteristics of serverless defined in section 2.2.

Since the gap between PaaS and FaaS can be quite subtle it warrants further consideration. Indeed some sources (e.g. Adzic and Chatley 2017) refer to FaaS as a new generation of PaaS offerings. Both models provide a high-level and elastic computing platform on which to implement custom business logic. There are however a number of substantial differences between the two models, which ultimately boil down to PaaS being an instance-based model with multiple server processes running on always-on server instances, as opposed to the on-demand resource allocation of FaaS. Put another way, "most PaaS applications are not geared

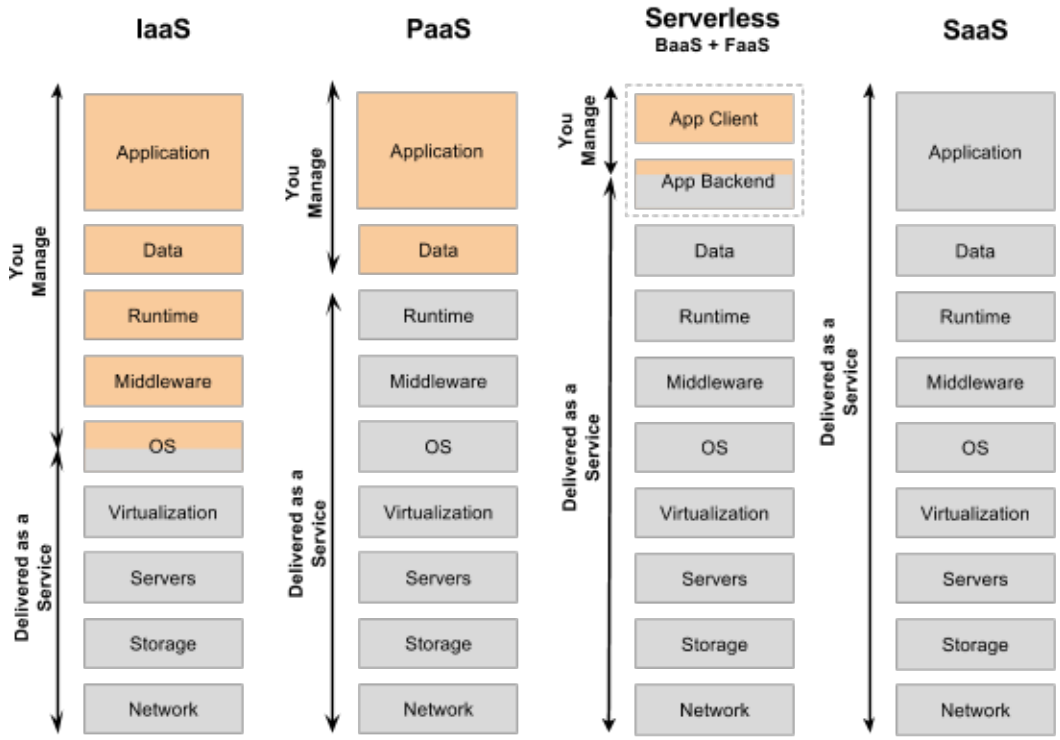

Figure 3. Degree of automation when using serverless (Wolf 2016)

towards bringing entire applications up and down for every request, whereas FaaS platforms do exactly this" (Roberts 2016).

Albuquerque Jr et al. (2017) derive a number of specific differences between PaaS and FaaS in their comparative analysis. First of all the units of deployment vary: PaaS applications are deployed as services, compared to the more granular function-based deployment of FaaS. Second, PaaS instances are always running whereas serverless workloads are executed on-demand. Third, PaaS platforms, although supporting auto-scaling to some extent, require the developer to explicitly manage the scaling workflow and number of minimum instances. FaaS on the other hand scales transparently and on-demand without any need for resource pre-allocation. Perhaps the most important distinction lies in billing: PaaS is billed by instantiated resources whether they're used or not, whereas FaaS is billed per-event only for the execution duration. The analysis concludes that PaaS is well suited for predictable or constant workloads with long or variable per-request execution times; FaaS in turn provides better cost benefit for unpredictable or seasonal workloads with short per-request execution

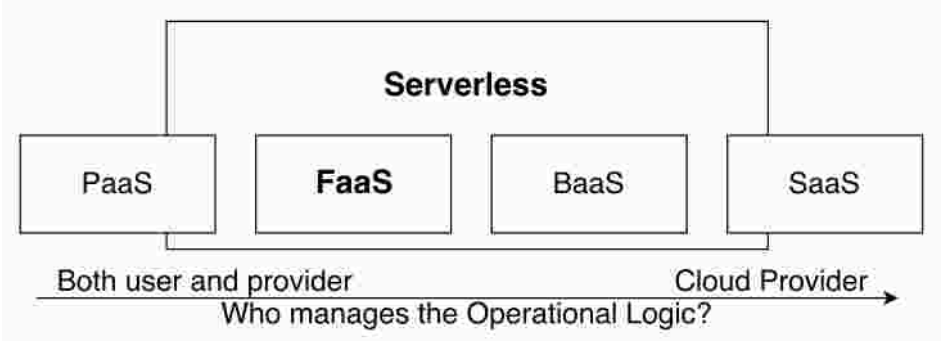

Figure 4. Serverless and FaaS vs. PaaS and SaaS (Erwin van Eyk et al. 2017)

times. It's also to be noted that PaaS doesn't suffer from limits on execution duration and many other restrictions of FaaS as described in section 2.9.

Another recent cloud-native technology is Container-as-a-Service (CaaS)... (Roberts 2016; Buyya et al. 2017)

2.4 Serverless processing model

The CNCF (2018) whitepaper divides a generalized serverless solution into four constituents, as illustrated in figure 5:

- Event sources - trigger or stream events into one or more function instances
- Function instances - a single function/microservice, that can be scaled with demand
- FaaS Controller- deploy, control and monitor function instances and their sources
- Platform services - general cluster or cloud services (BaaS) used by the FaaS solution

Interrelation of the various parts is further demonstrated with an example of a typical serverless development workflow. First, the developer selects a runtime environment (e.g. Python 3.6), writes a piece of code and uploads it on a FaaS platform where the code is published as a serverless function. The developer then maps one or more event sources to trigger the function, with event sources ranging from HTTP calls to database changes and messaging services. Now when any of the specified events occurs, the FaaS ontroller spins up a container, loads up the function along with its dependencies and executes the code. The function code typically contains API calls to external BaaS resources to handle data storage and other

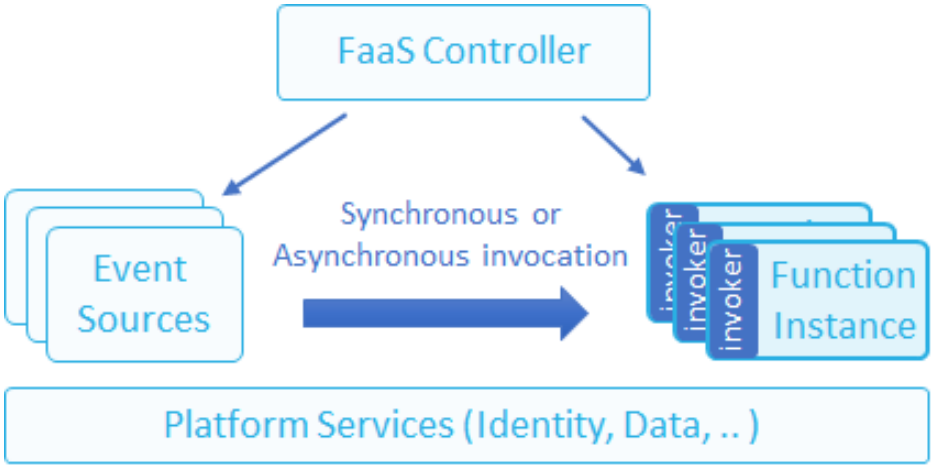

Figure 5. Serverless processing model (CNCF 2018)

integrations. When there are multiple events to respond to simultaneously, more copies of the same function are run in parallel. Serverless functions thus scale precisely with the size of the workload, down to the individual request. After execution the container is torn down. Later the developer is billed according to the measured execution time, typically in 100 millisecond increments. (AWS 2018)

At the heart of serverless architecture is the concept of a *function* (also *lambda function* or *cloud function*). A function represents a piece of business logic executed in response to specified events. Functions are the fundamental building block from which to compose serverless applications. A function is defined as a small, stateless, short-lived, on-demand service with a single functional responsibility (Erwin van Eyk et al. 2017). As discussed in section 2.1, the technology underlying cloud computing has evolved from individual servers to virtual machines and containers. Hendrickson et al. (2016) see the serverless function model as the logical conclusion of this evolution towards more sharing between applications (figure 6).

Being stateless and short-lived, serverless functions have fundamentally limited expressiveness compared to a conventional PaaS-hosted application. This is a direct result of being built to maximise scalability. A FaaS platform will need to execute the arbitrary code in a function in response to any number of events, without explicitly specifying resources required for the operation (Buyya et al. 2017). To make this possible, FaaS platforms pose

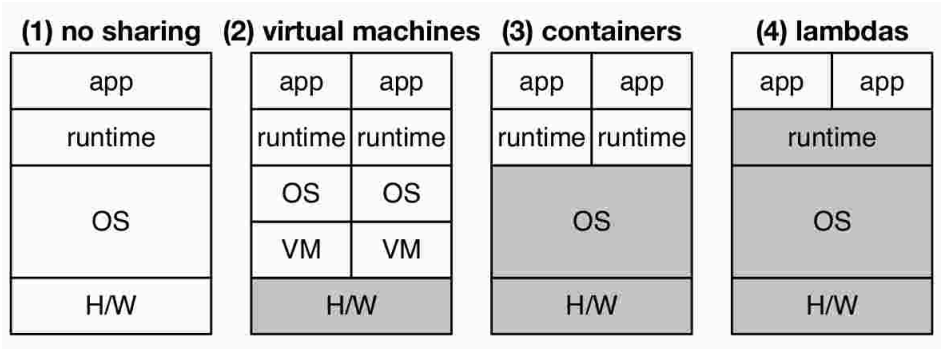

Figure 6. Evolution of sharing – gray layers are shared (Hendrickson et al. 2016)

restrictions on what functions can do and how long they can operate. Statelessness here means that a function loses all local state after termination; none of the local state created during invocation will necessarily be available during subsequent invocations of the same function. This is where BaaS services come in, with external stateful services such as key-value stores, databases and file or blob storages providing a persistence layer. In addition to statelessness, FaaS platforms limit a funtion's execution duration and resource usage: AWS Lambda for example has a maximum execution duration of 15 minutes and a maximum memory allocation of 3008 MB (AWS 2018).

FaaS event sources can be divided into two categories, synchronous and asynchronous. The first category follows a typical request-response flow: a client issues a request and blocks while waiting for a response. Synchronous event sources include HTTP and RPC calls which can be used to implement a REST API, a command line client or any other service requiring immediate feedback. Asynchronous event sources on the other hand result in non-blocking function execution, and they're typically used to implement background workers, scheduled event handlers and queue workers. Asynchronous event sources include message queues, publish-subscribe systems, database or file storage change feeds and CRON jobs among others. The details and metadata of the triggering event are passed to the function as an input parameter, with exact implementation varying per event type and provider. In case of a HTTP call, for example, the event object might include the request path, headers, body and query parameters. A function instance is also commonly supplied a context object, which in turn contains runtime information and other general properties that span multiple function invocations. Function name, version, memory limit and remaining execution time are examples

of typical context variables. FaaS platforms also usually allow users to set environment variables which function instances can access through the context object – useful for handling security keys and tokens. As for output, functions can either directly return a value (in case of synchronous invocation) or either trigger the next execution phase in a workflow or simply log the result (in case of asynchronous invocation). An example function handler is presented in listing 1. In addition to publishing and executing serverless functions, FaaS platforms also typically provide auxiliary capabilities such as monitoring, statistics, versioning and logging. (CNCF 2018)

```
def main(event, context):
    return {"payload": "Hello, " + event.name}
```

Listing 1. Example FaaS handler

As mentioned in section 2.2, serverless is *almost* but not completely devoid of operational management. In case of serverless functions, this qualification means that parameters such as memory reservation size, maximum parallelism and execution time are still left for the user to configure. Whereas the latter parameters are mainly used as safeguards to control costs, memory reservation size has important implications regarding execution efficiency (Lloyd et al. 2018). There are however tools available to determine the optimal memory reservation size per given workload. Also some platforms automatically reserve the required amount of memory without pre-allocation (Microsoft 2018).

Even with the restrictions on a serverless function's capabilities, implementing a FaaS platform is a difficult problem. From the customer's point of view the platform has to be as fast as possible in both spin-up and execution time, as well as scale indefinitely and transparently. The provider on the other hand seeks maximum resource utilization at minimal costs while avoiding violating the consumer's QoS expectations. Given that these goals are in conflict with each other, the task of resource allocation and scheduling bears crucial importance (HoseinyFarahabady et al. 2017). A FaaS platform must also safely and efficiently isolate functions from each other, and make low-latency decisions at the load balancer-level while

considering session, code, and data locality (Hendrickson et al. 2016).

2.5 Use cases

Serverless computing has been utilized to support a wide range of applications. Baldini, Castro, et al. (2017) note that from a cost perspective, the model is particularly fitting for bursty, CPU-intensive and granular workloads, as well as applications with sudden surges of popularity such as ticket sales. Serverless is less suitable for I/O-bound applications where a large period of time is spent waiting for user input or networking, since the paid-for compute resources go unused. In the industry, serverless is gaining traction primarily in three areas: Internet-of-Things (IoT) applications with sporadic processing needs, web applications with light-weight backend tasks, and as glue code between other cloud computing services (Spillner, Mateos, and Monge 2018).

A number of real-world and experimental use cases exists in literature. Adzic and Chatley (2017) present two industrial case studies implementing mind-mapping and social networking web applications in serverless architectures, resulting in decreased hosting costs. McGrath et al. (2016) describe a serverless media management system that easily and performantly solves a large-scale image resizing task. Fouladi et al. (2017) present a serverless video-processing framework. Yan et al. (2016) and Lehvä, Mäkitalo, and Mikkonen (2018) both implement serverless chatbots, reaching gains in cost and management efficiency. Ast and Gaedke (2017) describe an approach to building truly self-contained serverless web components.

In the domain of high-performance and scientific computing, Jonas et al. (2017) argue that "a serverless execution model with stateless functions can enable radically-simpler, fundamentally elastic, and more user-friendly distributed data processing systems". Malawski et al. (2017) experiment with running scientific workflows on a FaaS platform and find the approach easy to use and highly promising, noting however that not all workloads are suitable due to execution time limits. Spillner, Mateos, and Monge (2018) similarly find that "in many domains of scientific and high-performance computing, solutions can be engineered based on simple functions which are executed on commercially offered or self-hosted FaaS

platforms". Ishakian, Muthusamy, and Slominski (2017) evaluate the suitability of a serverless computing environment for the inferencing of large neural network models. Petrenko et al. (2017) present a NASA data exploration tool running on a FaaS platform.

The novel paradigms of edge and fog computing are identified as particularly strong drivers for serverless computing (Fox et al. 2017). These models seek to include the edge of the network in the cloud computing ecosystem to bring processing closer to the data source and thus reduce latencies between users and servers (Buyya et al. 2017). The need for more localized data processing stems from the growth of mobile and IoT devices as well as the demand for more data-intensive tasks such as mobile video streaming. Bringing computation to the edge of the network addresses this inreasing demand by avoiding the bottlenecks of centralized servers and latencies introduced by sending and retrieving heavy payloads from and to the cloud (Baresi, Filgueira Mendonça, and Garriga 2017). Nastic et al. (2017) explain how the increasing growth of IoT devices has lead to "an abundance of geographically dispersed computing infrastructure and edge resources that remain largely underused for data analytics applications" and how "at the same time, the value of data becomes effectively lost at the edge by remaining inaccessible to the more powerful data analytics in the cloud due to networking costs, latency issues, and limited interoperability between edge devices".

Despite the potential efficiencies gained, hosting and scaling applications at the edge of the network remains problematic with edge/fog computing environments suffering from high complexity, labor-intensive lifecycle management and ultimately high cost (Glikson, Nastic, and Dustdar 2017). Simply adopting the conventional cloud technologies of virtual machines and containers at the edge is not possible since the underlying resource pool at the edge is by nature highly distributed, heterogeneous and resource-constrained (Baresi, Filgueira Mendonça, and Garriga 2017). Serverless computing, with its inherent scalability and abstraction of infrastructure, is recognized by multiple authors as a promising approach to address these issues. Nastic et al. (2017) present a high-level architecture for a serverless edge data analytics platform. Baresi, Filgueira Mendonça, and Garriga (2017) propose a serverless edge architecture and use it to implement a low-latency high-throughput mobile augmented reality application. Glikson, Nastic, and Dustdar (2017) likewise propose a novel approach that extends the serverless platform to the edge of the network, enabling IoT and Edge de-

vices to be seamlessly integrated as application execution infrastructure. In addition, Erwin van Eyk et al. (2017) lay out a vision of a vendor-agnostic FaaS layer that would allow an application to be deployed in hybrid clouds, with some functions deployed in an on-premise cluster, some in the public cloud and some running in the sensors at the edge of the cloud.

2.6 Service providers

Lynn et al. (2017) provide an overview and multi-level feature analysis of the various enterprise serverless computing platforms. The authors identified seven different commercial platforms: AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, IBM Bluemix OpenWhisk, Iron.io Ironworker, Auth0 Webtask, and Galactic Fog Gestal Laser. All the platforms provide roughly the same basic functionality, with differences in the available integrations, event sources and resource limits. The most commonly supported runtime languages are Javascript followed by Python, with secondary support for Java, C#, Go, Ruby, Swift and others. The serverless platforms of the big cloud service providers, Amazon, Google, Microsoft and IBM, benefit from tight integration with their respective cloud ecosystems. The study finds that AWS Lambda, the oldest commercial serverless platform, has emerged as a *de facto* base platform for research on enterprise serverless cloud computing. AWS Lambda has also the most cited high profile use cases ranging from video transcoding at Netflix to data analysis at Major League Baseball Advanced Media. Google Cloud Functions remains in beta stage at the time of writing, and has limited functionality but is expected to grow in future versions (Google 2018). The architecture of OpenWhisk is shown in figure 7 as an example of a real-world FaaS platform. Besides the commercial offerings, a number of self-hosted open-source FaaS platforms have emerged: the CNCF (2018) whitepaper mentions fission.io, Fn Project, kubeless, microcule, Nuclio, OpenFaaS and riff among others. The core of the commercial IBM OpenWhisk is also available as an Apache open-source project (IBM 2018). In addition, research-oriented FaaS platforms have been presented in literature, including OpenLambda (Hendrickson et al. 2016) and Snafu (Spillner 2017b).

The big four serverless platforms are compared in a recent benchmark by Malawski et al. (2018). Each platform requires the user to configure a function's memory size allocation – apart from Azure Functions which allocate memory automatically. Available memory sizes

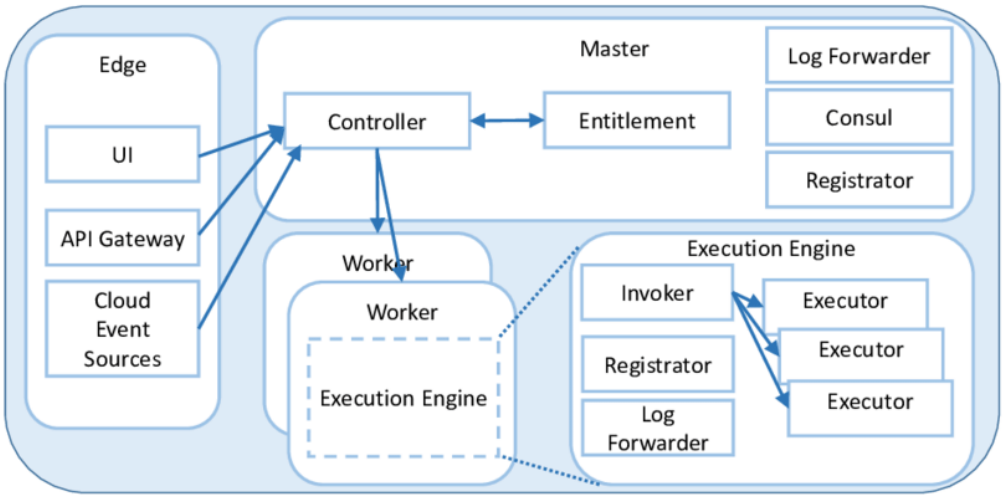

Figure 7. IBM OpenWhisk architecture (Baldini, Castro, et al. 2017)

range from 128 to 2048MB, with the per-invocation cost increasing in proportion to memory size. Measuring the execution time of CPU-intensive workloads with varying function sizes, the authors observe interesting differences in resource allocation between the different providers. AWS Lambda performs fairly consistently with CPU allocation increasing together with memory size as per the documentation. Google Cloud Functions instead behave less predictably with the smallest 128MB functions occasionally reaching the performance of the largest 2048MB functions. The authors suggest that this results from an optimization in container reuse, since reusing already spawned faster instances is cheaper than spinning up new smaller instances. Azure Functions show on average slower execution times, which the authors attribute to the underlying Windows OS and virtualization layer. On both Azure Functions and IBM Bluemix performance does not depend on function size.

Wang et al. (2018) “conduct the largest measurement study to date, launching more than 50,000 function instances across these three services, in order to characterize their architectures, performance, and resource management efficiency”.

2.7 Security

Similarly to PaaS, serverless architecture addresses most of the OS-level security concerns by pushing infrastructure management to the provider. Instead of users maintaining their own servers, security-related tasks like vulnerability patching, firewall configuration and intrusion detection are centralized, which has the benefit of reducing the attack surface. On the provider side the key issue becomes guaranteeing isolation between functions, as arbitrary code from many users is running on the same shared resources (McGrath and Brenner 2017). Since strong isolation has the downside of longer container startup times, the problem becomes finding an ideal trade-off between security and performance. (Erwin van Eyk et al. 2017)

In case of the BaaS model, the main security implication is greater dependency to third party services (Segal, Zin, and Shulman 2018). Each BaaS component represents a potential point of compromise, so it becomes important to secure communications, validate inputs and outputs and minimize and anonymize the data sent to the service. Roberts (2016) also notes that since BaaS components are used directly from the clients there's no protective server-side application in the middle, which requires significant care in designing the client application.

The FaaS model has a couple of advantages concerning security. First, FaaS applications are more resistant towards Denial of Service (DoS) attacks due to the platform's near limitless scalability – although such an attack can still inflate the monthly bill and inflict unwanted costs. Second, compromised servers are less of an issue in FaaS since functions run in short-lived containers that are repeatedly destroyed and reset. Overall, as put by *wagner16resilient*, "there is a much smaller attack surface when executing on a platform that does not allow you to open ports, run multiple applications, and that is not online all of the time". On the other hand application-level vulnerabilities remain as much of an issue in FaaS as in conventional cloud platforms. The architecture has no inherent protection against SQL injection or XSS and CSRF attacks, so existing mitigation techniques are still necessary. Vulnerabilities in application dependencies are another potential threat, since open-source libraries often make up the majority of the code in actual deployed functions. Also, the ease and low cost of deploying a high number of functions, while good for productivity, requires new

approaches to security monitoring. With each function expanding the application's attack surface it's important to keep track of ownership and allocate a function only the minimum privileges needed to perform the intended logic. Managing secure configuration per each funtion can become cumbersome with fine-grained applications consisting of dozens or hundreds of functions. (Podjarny 2017)

A study by the security company PureSec lists a number of prominent security risks specific to serverless architectures (Segal, Zin, and Shulman 2018). One potential risk concerns event data injection, i.e. functions inadvertently executing malicious input injected among the event payload. Since serverless functions accept a rich set of event sources and payloads in various message formats, there are many opportunities for this kind of injection. Another risk listed in the study is execution flow manipulation. Serverless architectures are particularly vulnerable to flow manipulation as serverless applications typically consist of many discrete functions chained together in a specific order. Application design might assume that a function is only invoked under specific conditions and only by authorized invokers. For example a function might forego a sanity check on the assumption that such a check has already been passed in some previous step. By manipulating execution order an attacker might be able to sidestep access control and gain unwanted entry to some resource. Overall the study stresses that since serverless is a new architecture its security implications are not yet well understood. Likewise security tooling and practices still lack in maturity.

2.8 Economics of serverless

Eivy (2017) and Villamizar et al. (2016) both focus on the economic aspects of serverless. Adzic and Chatley (2017) explain how novel design patterns are used to significantly optimize costs – just running traditional web apps inside Lambda containers doesn't necessarily equate to savings. Adzic and Chatley (2017) also report savings between 66 and 95% in two case studies, and present a handly table comparing hosting prices for intermittent service tasks. Spillner (2017a) exploits the control plane of AWS Lambda to implement services practically for free. Leitner, Cito, and Stöckli (2016) present an approach to model deployment costs of AWS Lambda applications in real-time. Kuhlenkamp and Klems (2017) present another cost-tracing system that enables per-request cost-tracing for

cloud-based software services, noting that cost testing should not only rely on isolated tests of single services but consider comprehensive end-to-end cost traces. Albuquerque Jr et al. (2017) have a detailed price comparison running the same app in FaaS and PaaS. Wagner and Sood (2016) argue that "managed code execution services such as AWS Lambda and GCP's Google Cloud Functions can significantly reduce the cost of operating a resilient system even in comparison to spot and preemptible VMs". The benchmark by Malawski et al. (2018) show that small functions run on faster resources on Google's platform.

The basic serverless pricing models follow a pay-per-use paradigm. As reported by Lane (2013) in a survey on the BaaS space, the most common pricing models offered by BaaS providers are billing on either the number of API calls or the amount of cloud storage consumed. The popularity of these pricing models reflects on the other hand the central role of API resources in BaaS as well as the fact that storage forms the biggest cost for BaaS providers. Beyond API call and storage pricing there are also numerous other pricing models to account for the multitude of BaaS types. Among the surveyed BaaS providers some charge per active user or consumed bandwidth, whereas others charge for extra features like analytics and tech support.

Pricing among FaaS providers is more homogeneous. FaaS providers typically charge users by the combination of number of invocations and their execution duration. Execution duration is counted in 100ms increments and rounded upwards, with the 100ms unit price depending on the selected function size. Each parallel function execution is billed separately. For example at the time of writing in AWS Lambda the price per invocation is $0.0000002 and computation is priced at $0.00001667 per GB-second (AWS 2018). The unit of GB-second refers to 1 second on execution time with 1GB of memory provisioned. Given this price per GB-second, the price for 100ms of execution ranges from $0.000000208 for 128MB functions to $0.000004897 for 3008MB functions. At this price point, running a 300ms execution on a 128MB function 10 million times would add up to about $8.25. The other major providers operate roughly at the same price point (Microsoft 2018; IBM 2018; Google 2018). Most providers also offer a free tier of a certain amount of free computation each month. The AWS Lambda free tier for example includes 1 million invocations and 400,000 GB-seconds (which adds up to 800,000 seconds on the 512MB function, for example) of

computation per month.

Villamizar et al. (2016) present an experiment comparing the cost of developing and deploying the same web application using three different architecture and deployment models: monolithic architecture, microservices operated by the cloud customer, and microservices operated by the cloud provider i.e. FaaS. The results come out in favor of FaaS, with more than a 50% cost reduction compared to self-operated microservices and up to a 77% reduction in operation costs compared to the monolithic implementation. The authors note however that for applications with small numbers of users, the monolithic approach can be a more practical and faster way to start since the adoption of more granular architectures demands new guidelines and practices both in development work and in an organizational level. Looking only at infrastructure costs, FaaS emerges as the most competetive approach.

To demonstrate how FaaS pricing works out in the customer's advantage in the case of intermittent computation, Adzic and Chatley (2017) compare the cost of running a 200ms service task every 5 minutes on various hosting platforms. Running a 512MB VM with an additional fail-over costs $0.0059 per hour, whereas a similarly sized Lambda function executing the described service task costs $0.000020016 for one hour – a cost reduction of more than 99.8%. The authors also present two real-world cases of FaaS migration. The first case, a mind-mapping web application, was migrated from PaaS to FaaS and resulted in hosting cost savings of about 66%. In the second case a social networking company migrated parts of their backend services from self-operated VMs to FaaS, and estimated a 95% reduction in operational costs.

TODO other real-world cases

A large part of the expenses incurred in developing today's computer systems derive from the need for *resiliency*. Resiliency means the ability to withstand a major disruption caused by unknown events. A resilient system is expected to be up and functioning at all times, while simultaneously providing good performance and certain security guarantees. Meeting these requirements forces organizations to over-provision and isolate their cloud resources which leads to increased costs. The FaaS model can significantly reduce the cost of resiliency by offloading resource management to the provider. This was exemplified in the above uses

cases, where majority of the cost savings arose from not having to pay for excess or idling resources. (Wagner and Sood 2016)

a big part of provider income comes from auxiliary services (Fox et al. 2017)

however eivy not quite there yet as observed by Fox et al. (2017), since network waiting time is wasted

2.9 Drawbacks and limitations

What to take into consideration when migrating to serverless?

Lloyd et al. (2018) analyze serverless performance and elasticity, identifying the cold start phenomenon. Differences in runtimes/languages, and larger library dependencies lead to slower starts. Oakes et al. (2017) address the problem by caching package dependencies on platform-level. Boucher et al. (2018) identify latency as an important problem ("a microservice is only as fast as the slowest service it relies on") and introduce restructuring of FaaS architecture centered around low-latency.

Baldini, Cheng, et al. (2017) identify three competing constraints in serverless function composition: functions should be considered as black boxes; function composition should obey a substitution principle with respect to synchronous invocation; and invocations should not be double-billed.

Roberts (2016), Adzic and Chatley (2017) and Baldini, Castro, et al. (2017) each list a number of limitations, including lack of strong SLA, vendor lock-in, short life-span, immature local development tools, statelessness and many others.

Kuhlenkamp and Klems (2017) discover two serverless cost tradeoffs: the retry cost effect and the cost ripple effect.

Malawski et al. (2018) talk about interoperability challenges running a heterogeneous benchmark, as well as discussion on RAM allocation.

CNCF (2018) on technical immaturity, lack of interoperability, standardization, tools, docu-

mentation, best practices.

Eivy (2017) notes that function size choosing is in fact paying for allocation, not the "pay what you use"-promise of serverless. Microsoft (2018) has automatic memory provisioning.

The need for circuit breakers (risk of DDoSing yourself) when interacting with non-serverless components like a database. Mention novel cloud-native database services like Google's Cloud Spanner and AWS Aurora. Figure out a source for this – Hohpe and Woolf (2004) might have a relevant pattern.

Address maintainability: debugging serverless functions, following the flow of control can be tough.

Composing serverless functions is not like composing regular functions. All the difficulties of distributed computing – message loss, timeouts and others – apply and have to be handled. Possible solutions include retry policies, dead-letter queues and idempotent functions.

A full-fledged general-purpose serverless computing model is still a vision that needs to be achieved. (Buyya et al. 2017)

Unlike the abstract concept of cloud functions, FaaS cannot completely abstract away all operation logic from the user. The FaaS user can still change parameters and configurations, such as the suggested memory size or number of CPUs of the underlying function host, which influence the operation of the deployed cloufunction. (Erwin van Eyk et al. 2017)

AWS whitepaper (AWS 2017) lists best practices.

Erwin van Eyk et al. (2018) identify 6 performance-related challenges for the serverless domain and plot a roadmap for alleviating these challenges.

Leitner et al. (2018) find that building serverless applications requires a "different mental model that emphasizes plugging together small microservices".

3 Serverless design patterns

Survey of serverless design patterns. Baldini, Castro, et al. (2017) put the question as follows:

> Will there be patterns for building serverless solutions? How do we combine low granularity basic building blocks of serverless into bigger solutions? How are we going to decompose apps into functions so that they optimize resource usage? For example how do we identify CPU-bound parts of applications built to run in serverless services? Can we use well-defined patterns for composing functions and external APIs? What should be done on the server vs. client (e.g., are thicker clients more appropriate here)? Are there lessons learned that can be applied from OOP design patterns, Enterprise Integration Patterns, etc.?

3.1 Serverless patterns

A limited number of purely serverless design patters. Sbarski and Kroonenburg (2017) introduce five patterns – Command, Messaging Priority queue, Fan-out and Pipes and filters – but they seem mostly to be reinterpretations of classic Enterprise Integration Patterns.

Common low-level design patterns such as function chaining and fanout in Github.

McGrath et al. (2016) demonstrate a fan-out pattern, easily and performantly solving a large-scale image resizing task.

API Gateway and messaging patterns as described in platforms' own documentation. (AWS 2018)

Hong et al. (2018) introduce a "taxonomy of serverless design patterns that we realize using Lambda and primitives provided by AWS. We categorize serverless design patterns into six groups: 1) periodic invocation, 2) event-driven, 3) data transformation, 4) data streaming, 5) state machine, and 6) bundled pattern."

Leitner et al. (2018) find that "Current FaaS applications are commonly small, and often

consist of 10 functions or less. Developers use various application patterns and workarounds to deal with the inherent limitations of current FaaS platforms". Authors further point out 5 patterns that are mainly workarounds over limitations of serverless platforms.

Adzic and Chatley (2017) suggest 3 methods for optimizing resource usage: use distributed authorization, let clients orchestrate workflows and allow clients to directly connect to AWS resources. The authors also discuss how paying only for actual utilization has two additional benefits of 1) removing incentives for bundling and 2) removing barriers to versioning: examples of serverless economics affecting architecture. Presenting these (or applications of these) as more formal patterns could be of some value.

Roberts (2016) asks how big can FaaS functions get before they get unwieldy? Assuming we can atomically deploy a group of FaaS functions what are good ways of creating such groupings - do they map closely to how we'd currently clump logic into microservices or does the difference in architecture push us in a different direction? Extending this further what are good ways of creating hybrid architectures between FaaS and traditional 'always on' persistent server components?

Extending this further what are good ways of creating hybrid architect

Ast and Gaedke (2017) describe self-contained web components with serverless backends.

3.2 Enterprise Integration Patterns

Hohpe and Woolf (2004) present a number of asynchronous messaging architectures in the seminal book on EIP. While predating the whole serverless phenomenon the patterns are still relevant. Hohpe even demonstrated implementing one of his patterns on top of Google's serverless platform in a blog post.

3.3 FaaSification

Spillner (2017c) describes an automated approach to transform monolithic Python code into modular FaaS units by partially automated decomposition. Doesn't really seem suitable for

the web application migration process covered in this thesis but worth mentioning.

4 Migration process

Implement a subset of the target app in a serverless style, utilizing the surveyed patterns and keeping log of the tricky parts. In case the patterns prove unsuitable for the given problem, try to come up with an alternative solution.

Describe the web application to migrate.

Decide on the parts to migrate. Should demonstrate the features and limitations of serverless as outlined above. Possible features include

- A simple REST API endpoint to showcase API Gateway and synchronous invocation. Shouldn't require any big changes to application code.
- Interaction between multiple services to demonstrate distributed transactions.
- A scheduled (cron) event.
- Interacting with an external SaaS service like Twilio, Auth0. Demonstrate event-driven invocation.
- others?

5 Evaluation

Evaluation the outcome of migration process. Estimate the effects on performance and hosting costs. Weigh in on maintainability, testability, developer experience etc.

6 Conclusion

What can we conclude about the research questions? Mention limitations and further research directions.

Bibliography

Adzic, Gojko, and Robert Chatley. 2017. “Serverless computing: economic and architectural impact”. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering,* 884–889. ACM.

Albuquerque Jr, Lucas F, Felipe Silva Ferraz, Rodrigo FAP Oliveira, and Sergio ML Galdino. 2017. “Function-as-a-Service X Platform-as-a-Service: Towards a Comparative Study on FaaS and PaaS”. *ICSEA 2017:* 217.

Ast, Markus, and Martin Gaedke. 2017. “Self-contained Web Components Through Serverless Computing”. In *Proceedings of the 2Nd International Workshop on Serverless Computing,* 28–33. WoSC ’17. Las Vegas, Nevada: ACM. ISBN: 978-1-4503-5434-9. doi:10.1145/3154847.3154849. http://doi.acm.org/10.1145/3154847.3154849.

AWS. 2017. *Serverless Architectures with AWS Lambda: Overview and Best Practices.* Technical report. Visited on February 26, 2018. https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf.

———. 2018. “AWS Lambda”. Visited on February 1, 2018. https://aws.amazon.com/lambda/.

Baldini, Ioana, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen J. Fink, Vatche Ishakian, Nick Mitchell, et al. 2017. “Serverless Computing: Current Trends and Open Problems”. *CoRR* abs/1706.03178. arXiv: 1706.03178. http://arxiv.org/abs/1706.03178.

Baldini, Ioana, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. “The Serverless Trilemma: Function Composition for Serverless Computing”. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software,* 89–103. Onward! 2017. Vancouver, BC, Canada: ACM. ISBN: 978-1-4503-

5530-8. doi:10.1145/3133850.3133855. http://doi.acm.org/10.1145/3133850.3133855.

Baresi, Luciano, Danilo Filgueira Mendonça, and Martin Garriga. 2017. "Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture". In *Service-Oriented and Cloud Computing,* 196–210. Cham: Springer International Publishing. ISBN: 978-3-319-67262-5.

Bernstein, D. 2014. "Containers and Cloud: From LXC to Docker to Kubernetes". *IEEE Cloud Computing* 1, number 3 (): 81–84. ISSN: 2325-6095. doi:10.1109/MCC.2014.51.

Boucher, Sol, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2018. "Putting the "Micro" Back in Microservice". In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18),* 645–650. USENIX Association.

Buyya, Rajkumar, Satish Narayana Srirama, Giuliano Casale, Rodrigo N. Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, et al. 2017. "A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade". *CoRR* abs/1711.09123. arXiv: 1711.09123. http://arxiv.org/abs/1711.09123.

Buyya, Rajkumar, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. 2009. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility". *Future Generation Computer Systems* 25 (6): 599–616. ISSN: 0167-739X. doi:https://doi.org/10.1016/j.future.2008.12.001. http://www.sciencedirect.com/science/article/pii/S0167739X08001957.

CNCF. 2018. *Serverless whitepaper.* Technical report. Cloud Native Computing Foundation. Visited on February 7, 2018. https://github.com/cncf/wg-serverless.

Eivy, Adam. 2017. "Be Wary of the Economics of" Serverless" Cloud Computing". *IEEE Cloud Computing* 4 (2): 6–12.

Eyk, E. van, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup. 2018. "Serverless is More: From PaaS to Present Cloud Computing". *IEEE Internet Computing* 22, number 5 (): 8–17. ISSN: 1089-7801. doi:10.1109/MIC.2018.053681358.

Eyk, Erwin van, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. 2018. “A SPEC RG Cloud Group’s Vision on the Performance Challenges of FaaS Cloud Architectures”. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering,* 21–24. ICPE ’18. Berlin, Germany: ACM. ISBN: 978-1-4503-5629-9. doi:10.1145/3185768.3186308. http://doi.acm.org/10.1145/3185768.3186308.

Eyk, Erwin van, Alexandru Iosup, Simon Seif, and Markus Thömmes. 2017. “The SPEC cloud group’s research vision on FaaS and serverless architectures”. In *Proceedings of the 2nd International Workshop on Serverless Computing,* 1–4. ACM.

Foster, I., Y. Zhao, I. Raicu, and S. Lu. 2008. “Cloud Computing and Grid Computing 360-Degree Compared”. In *2008 Grid Computing Environments Workshop,* 1–10. doi:10.1109/GCE.2008.4738445.

Fouladi, Sadjad, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. “Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads.” In *NSDI,* 363–376.

Fox, Geoffrey C., Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2017. “Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research”. *CoRR* abs/1708.08028. arXiv: 1708.08028. http://arxiv.org/abs/1708.08028.

Gannon, D., R. Barga, and N. Sundaresan. 2017. “Cloud-Native Applications”. *IEEE Cloud Computing* 4, number 5 (): 16–21. doi:10.1109/MCC.2017.4250939.

Glikson, Alex, Stefan Nastic, and Schahram Dustdar. 2017. “Deviceless Edge Computing: Extending Serverless Computing to the Edge of the Network”. In *Proceedings of the 10th ACM International Systems and Storage Conference,* 28:1–28:1. SYSTOR ’17. Haifa, Israel: ACM. ISBN: 978-1-4503-5035-8. doi:10.1145/3078468.3078497. http://doi.acm.org/10.1145/3078468.3078497.

Google. 2018. “Google Cloud Functions”. Visited on February 7, 2018. https://cloud.google.com/functions/.

Hammond, Jeffrey S., and John R. Rymer. 2016. *How To Capture The Benefi ts Of Microservice Design: Three Phases Of Adoption Begin With Today's Programming Models And Platforms.* Technical report. Forrester Research.

Hendrickson, Scott, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. "Serverless Computation with openLambda". In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing,* 33–39. HotCloud'16. Denver, CO: USENIX Association. http://dl.acm.org/citation.cfm?id=3027041.3027047.

Hohpe, Gregor, and Bobby Woolf. 2004. *Enterprise integration patterns: Designing, building, and deploying messaging solutions.* Addison-Wesley Professional.

Hong, Sanghyun, Abhinav Srivastava, William Shambrook, and Tudor Dumitras. 2018. "Go serverless: securing cloud via serverless design patterns". In *10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18).* USENIX Association.

Horner, Nathaniel, and Inês Azevedo. 2016. "Power usage effectiveness in data centers: overloaded and underachieving". *The Electricity Journal* 29 (4): 61–69. ISSN: 1040-6190. doi:https://doi.org/10.1016/j.tej.2016.04.011. http://www.sciencedirect.com/science/article/pii/S1040619016300446.

HoseinyFarahabady, MohammadReza, Young Choon Lee, Albert Y. Zomaya, and Zahir Tari. 2017. "A QoS-Aware Resource Allocation Controller for Function as a Service (FaaS) Platform". In *Service-Oriented Computing,* 241–255. Cham: Springer International Publishing. ISBN: 978-3-319-69035-3.

IBM. 2018. "IBM Cloud Functions". Visited on February 7, 2018. https://www.ibm.com/cloud/functions.

Ishakian, Vatche, Vinod Muthusamy, and Aleksander Slominski. 2017. "Serving deep learning models in a serverless platform". *CoRR* abs/1710.08460. arXiv: 1710.08460. http://arxiv.org/abs/1710.08460.

Jamshidi, P., A. Ahmad, and C. Pahl. 2013. “Cloud Migration Research: A Systematic Review”. *IEEE Transactions on Cloud Computing* 1, number 2 (): 142–157. ISSN: 2168-7161. doi:10.1109/TCC.2013.10.

Jonas, Eric, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. “Occupy the Cloud: Distributed Computing for the 99%”. *CoRR* abs/1702.04024. arXiv: 1702.04024. http://arxiv.org/abs/1702.04024.

Kleinrock, Leonard. 2003. “An Internet vision: the invisible global infrastructure”. *Ad Hoc Networks* 1 (1): 3–11. ISSN: 1570-8705. doi:https://doi.org/10.1016/S1570-8705(03)00012-X. http://www.sciencedirect.com/science/article/pii/S157087050300012X.

Kuhlenkamp, Jörn, and Markus Klems. 2017. “Costradamus: A Cost-Tracing System for Cloud-Based Software Services”. In *Service-Oriented Computing,* 657–672. Cham: Springer International Publishing. ISBN: 978-3-319-69035-3.

Lane, Kin. 2013. *Overview of the backend as a service (BaaS) space.* Technical report.

Lehvä, Jyri, Niko Mäkitalo, and Tommi Mikkonen. 2018. “Case Study: Building a Serverless Messenger Chatbot”. In *Current Trends in Web Engineering,* 75–86. Cham: Springer International Publishing. ISBN: 978-3-319-74433-9.

Leitner, P., J. Cito, and E. Stöckli. 2016. “Modelling and Managing Deployment Costs of Microservice-Based Cloud Applications”. In *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC),* 165–174.

Leitner, Philipp, Erik Wittern, Josef Spillner, and Waldemar Hummer. 2018. “A mixed-method empirical study of Function-as-a-Service software development in industrial practice”. *PeerJ Preprints* 6 (): e27005v1. ISSN: 2167-9843. doi:10.7287/peerj.preprints.27005v1. https://doi.org/10.7287/peerj.preprints.27005v1.

Lloyd, Wes, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. “Serverless Computing: An Investigation of Factors Influencing Microservice Performance”. *The IEEE International Conference on Cloud Engineering (IC2E).* Forthcoming.

Lynn, Theo, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha. 2017. "A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms". In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom),* 162–169. IEEE.

Malawski, Maciej, Kamil Figiela, Adam Gajek, and Adam Zima. 2018. "Benchmarking Heterogeneous Cloud Functions". In *Euro-Par 2017: Parallel Processing Workshops,* 415–426. Cham: Springer International Publishing. ISBN: 978-3-319-75178-8.

Malawski, Maciej, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. 2017. "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions". *Future Generation Computer Systems.* ISSN: 0167-739X. doi:https://doi.org/10.1016/j.future.2017.10.029. http://www.sciencedirect.com/science/article/pii/S0167739X1730047X.

McGrath, G., and P. R. Brenner. 2017. "Serverless Computing: Design, Implementation, and Performance". In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW),* 405–410. doi:10.1109/ICDCSW.2017.36.

McGrath, G., J. Short, S. Ennis, B. Judson, and P. Brenner. 2016. "Cloud Event Programming Paradigms: Applications and Analysis". In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD),* 400–406. doi:10.1109/CLOUD.2016.0060.

Mell, Peter, Tim Grance, et al. 2011. "The NIST definition of cloud computing".

Microsoft. 2018. "Microsoft Azure Functions". Visited on February 7, 2018. https://azure.microsoft.com/en-us/services/functions/.

Nastic, S., T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. 2017. "A Serverless Real-Time Data Analytics Platform for Edge Computing". *IEEE Internet Computing* 21 (4): 64–71. ISSN: 1089-7801. doi:10.1109/MIC.2017.2911430.

Oakes, E., L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2017. “Pipsqueak: Lean Lambdas with Large Libraries”. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW),* 395–400. doi:10.1109/ICDCSW.2017.32.

Pahl, C. 2015. “Containerization and the PaaS Cloud”. *IEEE Cloud Computing* 2, number 3 (): 24–31. ISSN: 2325-6095. doi:10.1109/MCC.2015.51.

Petrenko, Maksym, Mahabal Hegde, Christine Smit, Hailiang Zhang, Paul Pilone, Andrey A Zasorin, and Long Pham. 2017. “Giovanni in the Cloud: Earth Science Data Exploration in Amazon Web Services”. American Geophysical Union (AGU) Fall Meeting.

Podjarny, Guy. 2017. “Serverless Security implications—from infra to OWASP”. Visited on February 28, 2018. https://snyk.io/blog/serverless-security-implications-from-infra-to-owasp/.

Roberts, Mike. 2016. “Serverless Architectures”. Visited on February 1, 2018. https://martinfowler.com/articles/serverless.html.

Sareen, Pankaj. 2013. “Cloud Computing: Types, Architecture, Applications, Concerns, Virtualization and Role of IT Governance in Cloud”. *International Journal of Advanced Research in Computer Science and Software Engineering* 3 (3).

Sbarski, Peter, and S Kroonenburg. 2017. *Serverless Architectures on AWS: With examples using AWS Lambda.* Manning Publications, Shelter Island.

Schmidt, Kay-Uwe, Darko Anicic, and Roland Stühmer. 2008. “Event-driven reactivity: A survey and requirements analysis”. In *3rd International Workshop on Semantic Business Process Management,* 72–86.

Segal, Ory, Shaked Zin, and Avi Shulman. 2018. *The Ten Most Critical Security Risks in Serverless Architectures.* Technical report.

Spillner, Josef. 2017a. “Exploiting the Cloud Control Plane for Fun and Profit”. *CoRR* abs/1701.05945. arXiv: 1701.05945. http://arxiv.org/abs/1701.05945.

Spillner, Josef. 2017b. “Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation”. *CoRR* abs/1703.07562. arXiv: 1703.07562. http://arxiv.org/abs/1703.07562.

———. 2017c. “Transformation of Python Applications into Function-as-a-Service Deployments”. *CoRR* abs/1705.08169. arXiv: 1705.08169. http://arxiv.org/abs/1705.08169.

Spillner, Josef, Cristian Mateos, and David A. Monge. 2018. “FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC”. In *High Performance Computing,* 154–168. Cham: Springer International Publishing. ISBN: 978-3-319-73353-1.

Varghese, Blesson, and Rajkumar Buyya. 2018. “Next generation cloud computing: New trends and research directions”. *Future Generation Computer Systems* 79:849–861. ISSN: 0167-739X. doi:https://doi.org/10.1016/j.future.2017.09.020. http://www.sciencedirect.com/science/article/pii/S0167739X17302224.

Villamizar, Mario, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, et al. 2016. “Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures”. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on,* 179–182. IEEE.

Wagner, B., and A. Sood. 2016. “Economics of Resilient Cloud Services”. In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C),* 368–374. doi:10.1109/QRS-C.2016.56.

Walker, Mike J. 2017. “Hype Cycle for Emerging Technologies, 2017”. Visited on February 7, 2018. https://www.gartner.com/smarterwithgartner/top-trends-in-the-gartner-hype-cycle-for-emerging-technologies-2017/.

Wang, Liang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. “Peeking Behind the Curtains of Serverless Platforms”. In *2018 USENIX Annual Technical Conference (USENIX ATC 18),* 133–146. Boston, MA: USENIX Association. ISBN: 978-1-931971-44-7. https://www.usenix.org/conference/atc18/presentation/wang-liang.

Wolf, Oliver. 2016. “Serverless Architecture in short”. Visited on February 16, 2018. https://specify.io/concepts/serverless-baas-faas.

Yan, Mengting, Paul Castro, Perry Cheng, and Vatche Ishakian. 2016. “Building a Chatbot with Serverless Computing”. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs,* 5:1–5:4. MOTA ’16. Trento, Italy: ACM. ISBN: 978-1-4503-4669-6. doi:10.1145/3007203.3007217. http://doi.acm.org/10.1145/3007203.3007217.

Youseff, L., M. Butrico, and D. Da Silva. 2008. “Toward a Unified Ontology of Cloud Computing”. In *2008 Grid Computing Environments Workshop,* 1–10. doi:10.1109/GCE.2008.4738443.