Aleksi Pekkala

Migrating a web application to serverless architecture

Master's Thesis in Information Technology

November 29, 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: 59+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 its relative immaturity. For example, the industry-standard NIST definitions of cloud computing (Mell and Grance 2011) have yet to catch up with the technology. Likewise the most recent ISO cloud computing vocabulary (ISO 2014) bears no mention of serverless computing. As a result boundaries between serverless and other areas of cloud computing areas are still somewhat blurred, and the terms seem to carry different meanings depending on the author and 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 leans heavily on the industry-headed CNCF Serverless Working Group's effort to formalize and standardize serverless computing (CNCF 2018), as well as Roberts's (2016) seminal introduction to the topic and 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 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 subscription-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 ARPANET chieft scientists, 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). 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 computational 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 underutilized 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 launch 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 launch 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)

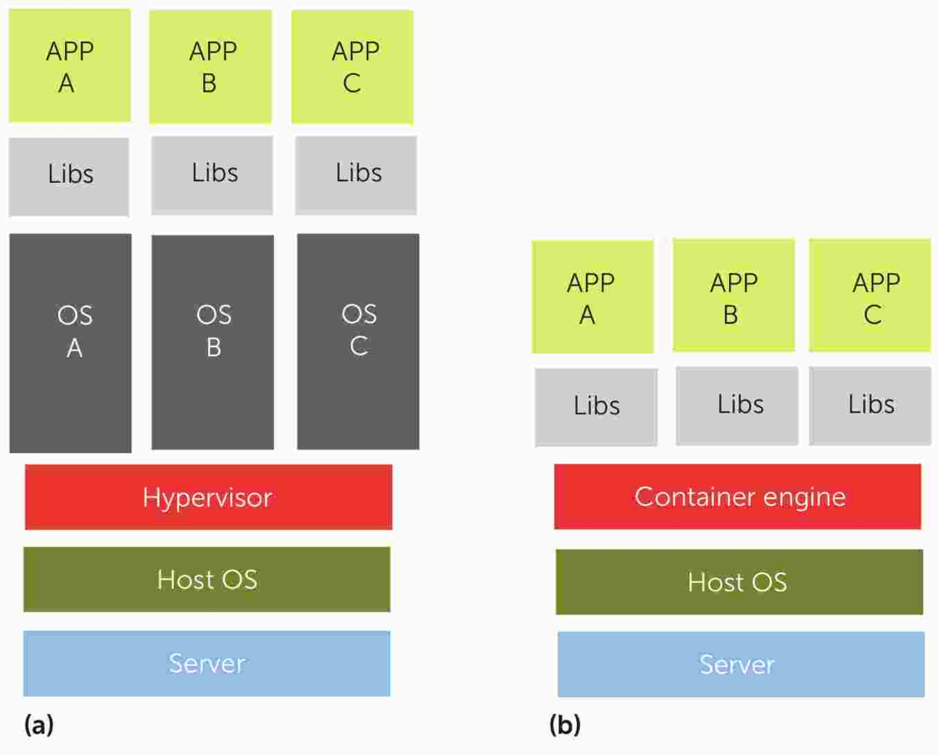

Figure 1. Comparison of a) virtual machine- and b) container-based deployments (Bernstein 2014)

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 bears no effect: the consumer ends up paying for the whole hour even if the application was actually performing computation for mere 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 in 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

Erwin van Eyk et al. (2017) define serverless computing as "a form of cloud computing that allows users to run event-driven and granularly billed applications, without having to address the operational logic". The definition breaks down into three key characteristics:

1. 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.
2. Granular billing: the user of a serverless model is charged only when the application is actually executing.
3. (Almost) no operational logic: operational logic, such as resource management and autoscaling, is delegated to the infrastructure, making those concerns of the infrastructure operator.

Fundamentally serverless computing is about building and running back-end code that does not require server management or long-lived 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 operational concerns are fully 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 and more high-level properties such as control, cost and flexibility. For the cloud customer this provides an abstraction where computation is disconnected from the infrastructure it runs on. (Roberts 2016; CNCF 2018)

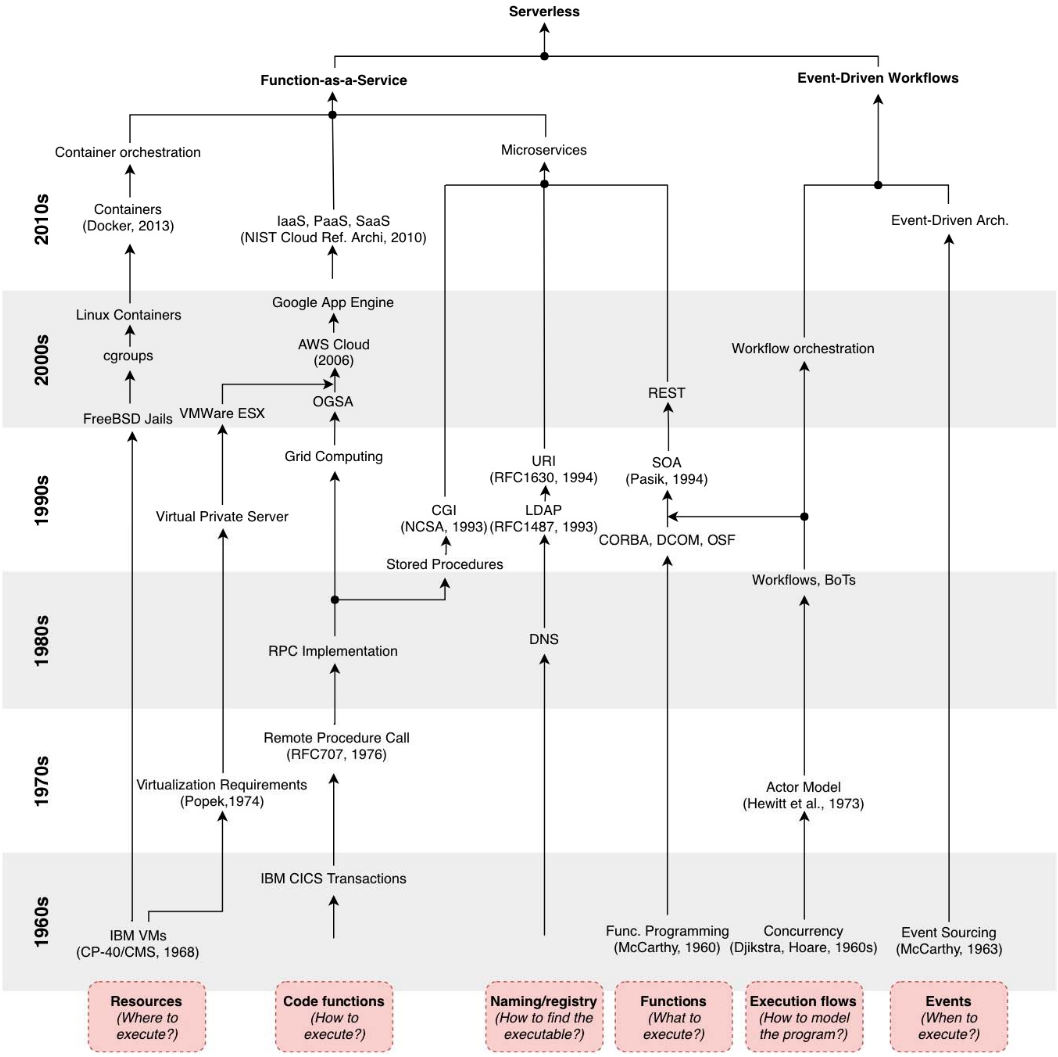
Serverless
Function-as-a-Service
Event-Driven Workflows
Container orchestration
Microservices
2010s
Containers (Docker, 2013)
IaaS, PaaS, SaaS (NIST Cloud Ref. Archi, 2010)
Event-Driven Arch.
2000s
Linux Containers
Google App Engine
AWS Cloud (2006)
cgroups
Workflow orchestration
OGSA
REST
FreeBSD Jails
VMWare ESX
1990s
Grid Computing
URI (RFC1630, 1994)
SOA (Pasik, 1994)
Virtual Private Server
CGI (NCSA, 1993)
LDAP (RFC1487, 1993)
CORBA, DCOM, OSF
Stored Procedures
Workflows, BoTs
1980s
RPC Implementation
DNS
1970s
Remote Procedure Call (RFC707, 1976)
Virtualization Requirements (Popek,1974)
Actor Model (Hewitt et al., 1973)
1960s
IBM CICS Transactions
IBM VMs (CP-40/CMS, 1968)
Func. Programming (McCarthy, 1960)
Concurrency (Djikstra, Hoare, 1960s)
Event Sourcing (McCarthy, 1963)
Resources (Where to execute?)
Code functions (How to execute?)
Naming/registry (How to find the executable?)
Functions (What to execute?)
Execution flows (How to model the program?)
Events (When to execute?)

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

Serverless platforms position themselves as the next step in the evolution of cloud computing architectures (Baldini, Castro, et al. 2017). E. van Eyk et al. (2018) trace the computing technologies that lead to the emergence of serverless computing in figure 2. First of all the rapid progress in systems infrastructure technologies, specifically virtualization and containerization as described in section 2.1, made serverless platforms technically feasible. Secondly, software architecture trends transitioning from "relatively large, monolithic applications, to smaller or more structured applications with smaller executions units" (Erwin van Eyk et al. 2017) paved the way for the serverless concept of functions as computation. E. van Eyk et al. (2018) see serverless computing continuing this trend of service specialization and abstraction, preceded by Service-Oriented Architecture and later by microservices. Finally the transition from synchronous systems to concurrent, event-driven distributed systems laid the groundwork for the serverless execution model: as per McGrath and Brenner (2017), serverless computing "is a partial realization of an event-driven ideal, in which applications are defined by actions and the events that trigger them".

2.2.1 Backend-as-a-Service and Function-as-a-Service

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 (Mell and Grance 2011) divides cloud offerings into three categories: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS), in increasing order of infrastructure abstraction (Mell and Grance 2011). As per Buyya et al. (2017), SaaS allows users to access complete applications hosted in the cloud, PaaS offers a framework for creation and development of more tailored cloud applications, and finally IaaS offers access to computing resources in form of leased VMs and storage space. 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 serverless characteristics 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-

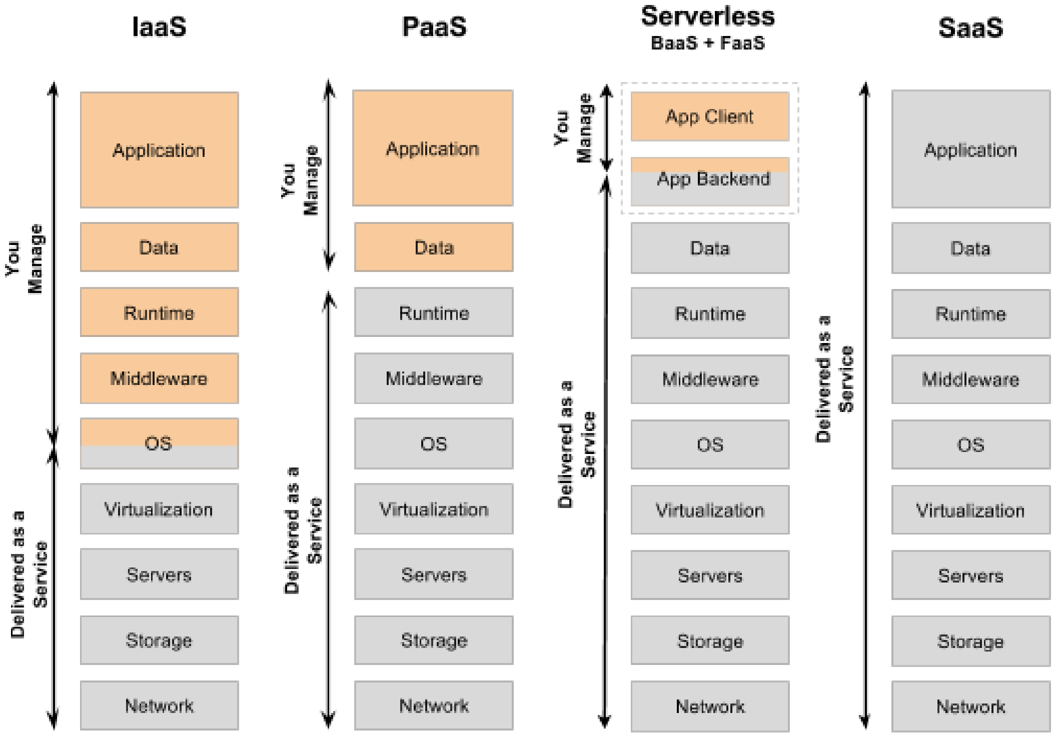

Figure 3. Degree of automation when using serverless (Wolf 2016)

demand resource allocation of FaaS. Put another way, "most PaaS applications are not geared 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

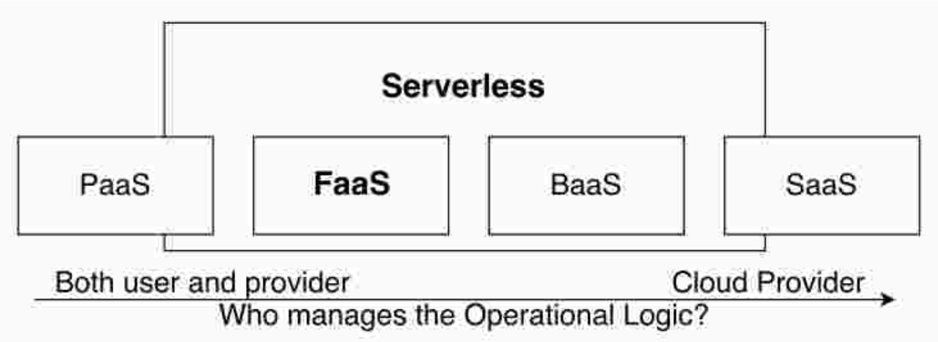

Figure 4. Serverless and FaaS vs. PaaS and SaaS (Erwin van Eyk et al. 2017)

better cost benefit for unpredictable or seasonal workloads with short per-request execution 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.

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 integrations. When there are multiple events to respond to simultaneously, more copies of

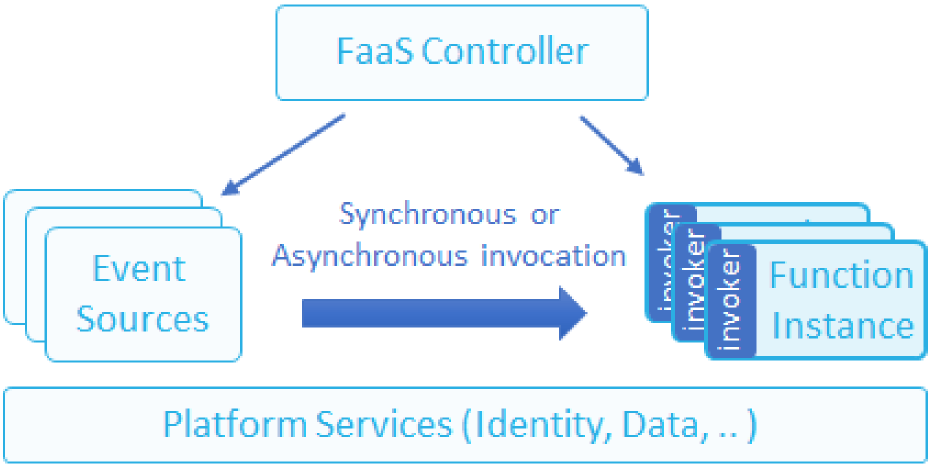
FaaS Controller
Event Sources
Synchronous or Asynchronous invocation
invoker
invoker
invoker
Function Instance
Platform Services (Identity, Data, ..)

Figure 5. Serverless processing model (CNCF 2018)

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 restrictions on what functions can do and how long they can operate. Statelessness here

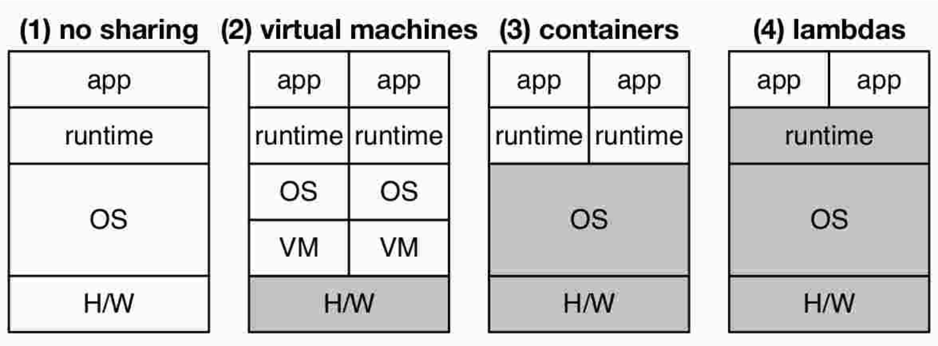

Figure 6. Evolution of sharing – gray layers are shared (Hendrickson et al. 2016)

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 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 are 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 schedulers among others. The details and metadata of the triggering event are passed to the function as input parameters, with exact implementation varying per event type and provider. In case of an 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 allow users to set environment variables which function instances can access through the context object – useful for handling configuration parameters and secret keys. As for output, functions can 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 provide auxiliary capabilities such as monitoring, versioning and logging. (CNCF 2018)

```python
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) suggest 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 devices 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 2017a).

The big four FaaS 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 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 pre-

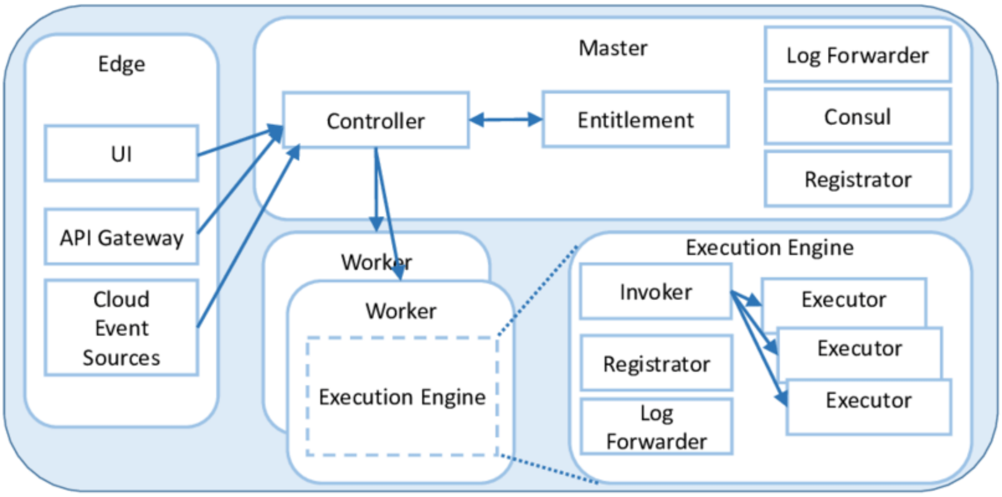

Figure 7. IBM OpenWhisk architecture (Baldini, Castro, et al. 2017)

dictably with the smallest 128MB functions occasionally reaching the performance of the largest 2048MB functions. The authors suggest 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.

A consequence of the high abstraction level of serverless computing is that the commercial FaaS platforms are essentially black boxes, with little guarantee about underlying resources. There are however efforts to gain insight into the platforms via reverse engineering. Wang et al. (2018) present 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”. One of the findings is that all service providers exhibit a variety of VMs as hosts, which may cause inconsistent function performance. The study also reveals differences on how serverless platforms allocate functions to host VMs. Both AWS Lambda and Azure Functions scale function instances on the same

VM, which results in resource contention as each function gets a smaller share of the network and I/O resources. Among the compared platforms, AWS Lambda achieved the best scalability and lowest start-up latency for new function instances.

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 with the benefit of a reduced 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 by the client, there's no protective server-side application in the middle which requires significant care in designing the client application.

The FaaS model has a number of advantages when it comes to security. First, FaaS applications are more resilient 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 Wagner and Sood (2016), "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 applications typically consist of many discrete functions chained together in a specific order. Application design might assume a function is only invoked under specific conditions and only by authorized invokers. A function might for example forego a sanity check on the assumption that 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.

The Open Web Application Security Project has also published a preliminary report reevaluating the top 10 web application security risks from a serverless standpoint (OWASP 2018). The report notes that the more standardized authentication & authorization models and fine-grained architecture inherent to serverless applications are an improvement over traditional applications security-wise. Individual functions are typically limited in scope and can thus be assigned a carefully crafted set of permissions, following the "least privilege" principle. On the other hand configuring access control for a large serverless application can be onerous and lead to backdoors in form of over-privileged functions. The report also deems serverless applications more susceptible to vulnerabilities in external components and 3rd party libraries due to each function bringing in its own set of dependencies. Similarly to Segal, Zin, and Shulman (2018), potential risks also include increased injection attack surface due to multitude of event sources and business logic & flow manipulation attacks. In summary, the authors conclude with the notion that “the risks were not eliminated, they just changed, for better and for worse”.

2.8 Economics of serverless

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 memory capacity. 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 e.g. 800,000 seconds on the 512MB function) of computation per month. Interestingly, as with most FaaS providers CPU allocation increases together with selected memory size, the smallest memory size might not always be the cheapest option: a higher memory size might lead to faster execution and thus offset the higher resource expenses.

Villamizar et al. (2017) 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 competitive 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.

Wagner and Sood (2016) describe how 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 serverless model can significantly reduce the cost of resiliency by offloading resource management to the provider. The authors conclude 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”. This was exemplified in the above case studies, where majority of cost savings arose from not having to pay for excess or idling resources.

One apparent flaw in FaaS pricing concerns network delays. A function that spends most of its execution time waiting for a network call is billed just the same as a function that spends an equivalent time doing actual processing. Fox et al. (2017) call into question the serverless promise of never paying for idle, noting that “serverless computing is a large step forward but we’re not there yet [...] as time spent waiting on network (function executions or otherwise) is wasted by both provider and customer”. The authors also observe that a part of a serverless provider’s income comes from offering auxiliary services such as traditional storage. Eivy (2017) similarly heeds caution with the potentially confusing FaaS pricing model of GB-seconds, reminding that on top of the per-hit fee and GB-seconds you end up paying for data transfer, S3 for storing static assets, API Gateway for routing and any other incidental services. It’s also notable that as FaaS GB-second pricing comes in rounded-up increments of 100ms, any optimization under 100ms is wasted in a financial sense. However, when comparing serverless to conventional cloud computing expenses, it’s worth bearing in mind the savings in operational overhead: “even though serverless might be 3x the cost of on-demand compute, it might save DevOps cost in setting up autoscale, managing security patches and debugging issues with load balancers at scale” (Eivy 2017). Finally, in a cloud developer survey by Leitner et al. (2018), majority of participants perceived the total costs of FaaS to be cheaper than alternative cloud platforms.

2.9 Drawbacks and limitations

Roberts (2016) observes two categories of drawbacks in serverless computing: trade-offs inherent to the serverless concept itself, and the ones tied to current implementations. Inherent trade-offs are something developers are going to have to adapt to, with no foreseeable solution in sight. Statelessness, for example, is one of the core properties of serverless: we cannot assume any function state will be available during later or parallel invocations of the same function. This property enables scalability, but at the same time poses a novel software engineering challenge as articulated by Roberts (2016): “where does your state go with FaaS if you can’t keep it in memory?” One might push state to an external database, in-memory cache or object storage, but all of these equate to extra dependencies and network latency. A common stateful pattern in web applications is to use cookie-based sessions for user authentication; in the serverless paradigm this would either call for an external state store or an alternative stateless authentication pattern (Hendrickson et al. 2016).

Another inherent trade-off relates to function composition, i.e. combining individual functions into full-fledged applications. Composing serverless functions is not like composing regular source code functions, in that all the difficulties of distributed computing – e.g. message loss, timeouts, consistency problems – apply and have to be dealt with. In complex cases this might result in more operational surface area for the same amount of logic when compared to a traditional web application (CNCF 2018). Baldini, Cheng, et al. (2017) explore the problem of serverless composition and identify a number of challenges. First of all when a function sequentially invokes and waits for the return of another function, the parent function must stay active during the child function’s execution. This results in the customer paying twice: once for the parent function and again for the invoked function. This phenomenon of *double billing* extends to any number of nested invocations and is thus highly undesirable. As well as billing, limits on execution duration constraint nested function composition. The authors describe another form of function composition where a function upon return fires a completion trigger that in turn asynchronously invokes another function, akin to continuation-passing style. This form avoids the problem of double billing, but in effect makes the resulting composition event-driven and thus not synchronously composable. One indicator of the complexity of composing serverless functions is that in a recent industry survey (Leitner et al. 2018) current FaaS applications were found to be small in size, generally consisting of 10 or fewer functions. The same study observes that adopting FaaS requires a mental model fundamentally different from traditional web-based applications, one that emphasizes “plugging together” self-contained microservices and external components. While novel, the serverless mental model was found to be easy to grasp. Finally, familiarity with concepts like functional programming and immutable infrastructures was considered helpful when starting with FaaS.

Vendor lock-in is another inherent serverless trade-off pointed out by multiple authors (e.g. Baldini, Castro, et al. 2017; CNCF 2018; Roberts 2016). While programming models among the major FaaS providers have evolved into fairly similar forms, FaaS applications tend to integrate tightly with various other platform services which means a lack of interoperability and difficulties in migration between cloud providers. Vendor lock-in is a general concern in cloud computing, but especially relevent here as serverless architectures incentivize tighter coupling between clients and cloud services (Adzic and Chatley 2017). Vendor control is another concern, as serverless computing intrinsically means passing control over to a third-party provider (Roberts 2016). This is partly addressed by FaaS platforms maturing and offering stronger Service Level Agreements: both AWS (2018) and Microsoft (2018) by now guarantee 99.95% availability.

Another category of serverless drawbacks are the ones related to current implementations. Unlike the inherent trade-offs described above, we can expect to see these problems solved or alleviated with time (Roberts 2016). The most apparent implementation drawbacks in FaaS are limits on function life-span and resource usage, as outlined in section 2.4. A function that exceeds either its duration or memory limit is simply terminated mid-execution, which means that larger tasks need to be divided and coordinated into multiple invocations. The lifespan limit is likewise problematic for Websockets and other protocols that rely on long-lived TCP connections, since FaaS platforms do not provide connection handling between invocations (Hendrickson et al. 2016).

Startup latency is one of the major performance concerns in current FaaS implementations (CNCF 2018). As per the on-demand structure, FaaS platforms tie up container resources upon function invocation and release them shortly after execution finishes. This leads to higher server utilization but incurs container initialization overhead. In case of frequent execution the overhead can be avoided as FaaS platforms reuse the function instance and host container from previous execution in a so called "warm start". A "cold start" in turn occurs when some time has elapsed since previous execution and the host container instance has been deprovisioned, in which case the platform has to launch a new container, set up the runtime environment and start a fresh function host process. Application traffic patterns and idle duration play a defining role in startup latency: a function invoked once per hour will probably see a cold start on each invocation, whereas a function processing 10 events per second can largely depend on warm starts. For background processing and other tasks where latency is not of great importance, cold starts are typically manageable. Latency-critical but infrequently executed functions might instead work around the problem with scheduled pings that prevent the instance from being deprovisioned and keep the function warm. (Roberts 2016)

Hendrickson et al. (2016) compare the warm and cold start behaviours in AWS Lambda, observing a 1ms latency in unpausing a container as opposed to hundreds of milliseconds of latency in restarting or fresh starting a container. Keeping containers in paused state until the next function invocation isn't feasible though due to high memory cost. Improving FaaS startup latency then becomes a problem of either reducing container restart overhead or reducing the memory overhead of paused containers. Lloyd et al. (2018) further subdivide function initialization into 4 possible states (in decreasing order of startup latency): *provider cold*, *VM cold*, *container cold* and *warm*. The first state occurs when a new function is invoked for the first time, requiring a new container image build. *VM cold* state requires starting a new VM instance and transferring the container image to the host. A *container cold* initialization involves spinning up a new container instance on an already running VM using the pre-built container image, and a *warm* run refers to reusing the same container instance as outlined above. Experimenting with AWS Lambda invocations interspersed with various idle periods, the authors observed that warm containers were retained for 10 minutes and VMs for 40 minutes. After 40 minutes of inactivity all original infrastructure was deprovisioned, leading to a 15x startup latency on the next invocation when compared to a warm start. Finally, the authors observed correlation between function memory size and cold start performance, with an approximately 4x performance boost when increasing memory size from 128MB to 1536MB.

Wang et al. (2018) provide empiric observations on startup latencies among various serverless platforms. Measuring the difference between invocation request time and execution start time using the NodeJS runtime, the authors discovered a median warm start latency of 25ms, 79ms and 320ms on AWS, Google and Azure, respectively. Median cold start latency on

AWS ranged from 265ms on a 128MB function to 250ms on a 1536MB function. Memory allocation had more impact on Google Functions with median cold start latency ranging from 493ms on a 128MB function to 110ms on a 2048MB function. Azure, with no memory size pre-allocation, revealed a considerably higher cold start latency at 3640ms. Runtime environment also had an observable effect, as Python 2.7 achieved median latencies of 167-171ms while Java functions took closer to a second. Apart from memory allocation and runtime environment, function size (consisting of source code, static assets and any third-party libraries) affects startup latency (Hendrickson et al. 2016). FaaS runtimes typically come preconfigured with certain common libraries and binaries, but any additional dependencies have to be bundled together with source code. On top of increasing download time from function repository to a fresh container, library code often has to be decompressed and compiled with further implications on startup latency. Hendrickson et al. (2016) propose adding package repository support to the FaaS platform itself. Oakes et al. (2017) in turn design a package caching layer on top of the open-source FaaS platform OpenLambda.

Erwin van Eyk et al. (2018) see tackling the novel performance-related challenges of FaaS crucial for more general adoption of the technology, particularly in the latency-critical use cases of web and IoT applications. The first challenge concerns the performance overhead incurred by splitting an application into fine-grained FaaS functions. Overhead in FaaS originates primarily from resource provisioning as described above, but request-level tasks like routing as well as function lifecycle management and scheduling also play a part. Performance isolation is another challenge noted by the authors: FaaS platforms typically deploy multiple functions on the same physical machine, which improves server utilization but has the drawback of reducing function performance due to resource contention. Function scheduling, i.e. deciding where an invoked function should be executed, is another complicated problem with multiple constraints: schedulers have to balance between available resources, operational cost, function performance, data locality and server utilization among other concerns. Finally, the authors note the lack of performance prediction and cost-performance analysis tools as well as a need for comprehensive and systematic platform benchmarks.

Leitner et al. (2018) surveyed cloud developers on FaaS challenges with interesting results:

the most prominent obstacles weren't performance-related, but rather pointed to a lack of tooling and difficulties in testing. Integration testing in particular remains a thorny subject, since serverless applications are by nature highly distributed and consist of multiple small points of integration. Reliance on external BaaS components also often necessitates writing stubs and mocks, which further complicates testing. On the other hand this is an area of rapid progress, with the advent of popular open-source frameworks as well as tools for local execution and debugging (Roberts 2016).

In general serverless is still an emerging computing model lacking in standardization, ecosystem maturity, stable documentation, samples and best practices (CNCF 2018). Current FaaS implementations in many ways fall short of the abstract notion of utility computing. Put another way, "a full-fledged general-purpose serverless computing model is still a vision that needs to be achieved" (Buyya et al. 2017). In addition to incurring a performance overhead, current FaaS platforms fail to completely abstract away all operational logic from the user, as users still have to allocate memory and set limits on execution duration and parallelism (Erwin van Eyk et al. 2017). Also despite improving utilization from previous cloud service models, FaaS platforms still operate in relatively coarse-grained increments: Eivy (2017) gives the pointed example that "the cost to use one bit for a nanosecond is no different than the cost to use 128MB for 100 milliseconds".

Future directions involve addressing these limitations, with a few interesting efforts already springing up: Boucher et al. (2018) for example propose a reimagining of the serverless model, eschewing the typical container-based infrastructure in favour of language-based isolation. The proposed model leverages language-based memory safety guarantees and system call blocking for isolation and resource limits, delivering invocation latencies measured in microseconds and a smaller memory footprint. The authors hypothesize that combining low network latencies available in modern data centers together with minuscule FaaS startup latency will enable "new classes and scales for cloud applications" as "fast building blocks can be used more widely". In fact one commercial FaaS platform, Cloudflare Workers, already offers a Javascript runtime which, instead of spawning a full NodeJS process per invocation, utilizes language-based isolation in shape of V8 isolates – the same technology used to sandbox Javascript running in browser tabs (Cloudflare 2018). Al-Ali et al. (2018) explore altogether different boundaries with ServerlessOS, an architecture where not functions, but user-supplied processes are fluidly scaled across a data center. Compared to the FaaS model of functions and events, a process-based abstraction “enables processing to not only be more general purpose, but also allows a process to break out of the limitations of a single server”. The authors also argue that the familiar process abstraction makes it easier to reploy existing code and migrate legacy applications on to a serverless platform.

3 Serverless design patterns

In this chapter we take a look at serverless design patterns. Design patterns describe commonly accepted, reusable solutions to recurring problems (Hohpe and Woolf 2004). A design pattern is not a one-size-fits-all solution directly translatable into software code, but rather a formalized best practice that presents a common problem in its context along a general arrangement of elements that solves it (Gamma et al. 1994). The patterns in this chapter are sourced from scientific literature on serverless computing as well as cloud provider documentation. Literature on object-oriented patterns (Gamma et al. 1994), SOA patterns (Rotem-Gal-Oz 2012) and enterprise integration patterns (Hohpe and Woolf 2004) was also reviewed for applicable practices.

Enterprise integration patterns, as serverless is all about integrations. 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. E.g. patterns like Idempotent Receiver, Dead-letter Channel as well as the 4 more general integration styles of File Transfer, Shared Database, RPC and Messaging. Many patterns implemented internally by FaaS platforms already!

SOA patterns: FaaS functions are self-contained nanoservices these might have some relevance. SOA patterns (Rotem-Gal-Oz 2012) include Saga, Decoupled Invocation and others. As with EIP, some patterns are already implemented by the FaaS platform.

FaaSification: Spillner (2017b) 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.

3.1 Orchestration patterns

The following patterns concern serverless function orchestration, i.e. managing control flow to compose functions together into more extensive sequences or workflows.

3.1.1 Routing Function

Problem: How to branch out execution flow based on request payload?

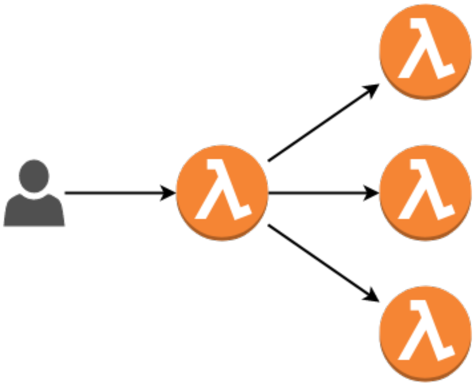

Figure 8. Routing Function

Solution: Use a central routing function to receive requests and invoke appropriate functions based on request payload.

This pattern involves instantiating a routing function that contains all the necessary information to route requests to other functions. All function invocations are directed to the routing function, which in turn invokes target functions according to request payload. The routing function finally passes target function return value over to the client.

It's notable that FaaS platforms commonly provide API gateways and other tools for routing, for example the Amazon API Gateway (AWS 2018). These tools are however mostly limited to path-based routing, whereas a routing function can be implemented to support more dynamic use cases. Also interestingly, according to an industry survey (Leitner et al. 2018), some practicioners opted for the Routing Function pattern over platform API gateway services as they found the latter cumbersome to manage. Sbarski and Kroonenburg (2017) similarly postulate that the pattern "can simplify the API Gateway implementation, because you may not want or need to create a RESTful URI for every type of request". One advantage of the pattern is that the routing function can be used to supplement request payload with additional context or metadata. A centralized routing function also means that all routing configuration is found in one place, and that public-facing API routes only need to be configured for one function, not all of them (Leitner et al. 2018).

The pattern's major disadvantage is double billing, as the routing function essentially has to block and wait until the target function finishes execution. Additionally, as routing is implemented in function code level, information about function control flow gets hidden in implementation rather than being accessible from configuration (Leitner et al. 2018).

The Routing Function pattern is related to the OOP Command pattern, which is used to decouple the caller of the operation from the entity that carries out the processing via an intermediary command object (Gamma et al. 1994). A related enterprise integration pattern is Content-Based Router, which "examines the message content and routes the message onto a different channel based on data contained in the message" (Hohpe and Woolf 2004). Hohpe and Woolf (2004) caution that the router should be made easy to maintain as it can become a point of frequent configuration.

3.1.2 Function Chain

Problem: Task exceeds maximum function execution duration, resulting in a timeout.

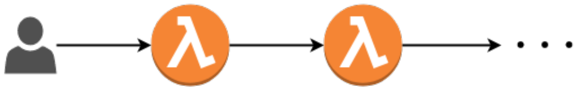

Figure 9. Function Chain

Solution: Split the task into separate functions that are chained together sequentially.

The Function Chain pattern comprises of an initial function and any number of subsequent functions. The initial function begins computation while keeping track of remaining execution time. In AWS Lambda for example the execution context contains information on how many milliseconds are left before termination (AWS 2018). Upon reaching its duration limit, the initial function invokes another function asynchronously, passing along as parameters any state necessary to continue task computation. Since the intermediary invocation is asynchronous (“fire-and-forget”), the initial function can terminate without affecting the next function in chain.

The Function Chain pattern is in effect a workaround over the duration limit that FaaS platforms place on function execution (Leitner et al. 2018). The pattern was reported to be used at least occasionally in an industry study by Leitner et al. (2018). Its disadvantages include strong coupling between chained functions, increase in the number of deployment units and the overhead of transferring intermediate execution state and parameters between each chained function. Leitner et al. (2018) also note that splitting some types of tasks into multiple functions can be difficult. Finally, as the pattern relies on asynchronous invocation, the last function in chain has to persist computation result into an external storage for the client to access it, which brings in further dependencies.

3.1.3 State Machine

Problem: How to coordinate complex, stateful procedures with branching steps?

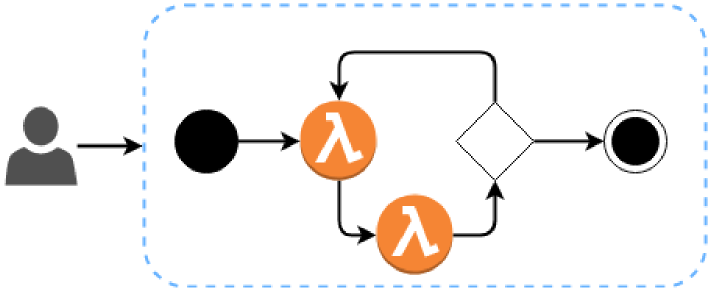

Figure 10. State Machine

Solution: Split a task into a number of discrete functions and coordinate its execution with an orchestration tool.

Hong et al. (2018) describe the State Machine pattern as “building a complex, stateful procedure by coordinating a collection of discrete Lambda functions using a tool such as AWS Step Functions”. These orchestration tools consist of a collection of workflow states and transitions between them, with each state having its associated function and event sources – essentially a serverless a state machine (CNCF 2018). Figure 10 could for example represent a workflow where the first function attempts a database insert, the second function checks whether the operation succeeded, and depending on the result either the operation is retried or execution is finished. The advantage of using provider tooling for workflow execution is that there's no need for external state storage as the orchestrator keeps track of workflow state. Downsides on the other hand include extra cost arising from orchestration tooling as well as the overhead of managing workflow descriptions.

López et al. (2018) provide a comparison of three major FaaS orchestration systems: AWS Step Functions, IBM Composer and Azure Durable Functions. The compared systems typically support function chaining, conditional branching, retries and parallel execution, with workflows defined either in a Domain-Specific Language or directly in code. One restriction in Amazon's orchestrator implementation is that a composition cannot be synchronously invoked and is thus not composable in itself: a state machine cannot contain another state machine. AWS Step Functions was also the least programmable among the compared systems, but on the other hand the most mature and performant. Finally, the authors observe that none of the provider-managed orchestration systems are prepared for parallel programming, with considerable overheads in concurrent invocations.

3.1.4 Thick Client

Problem: How to coordinate access to third party cloud services while avoiding extra costs?

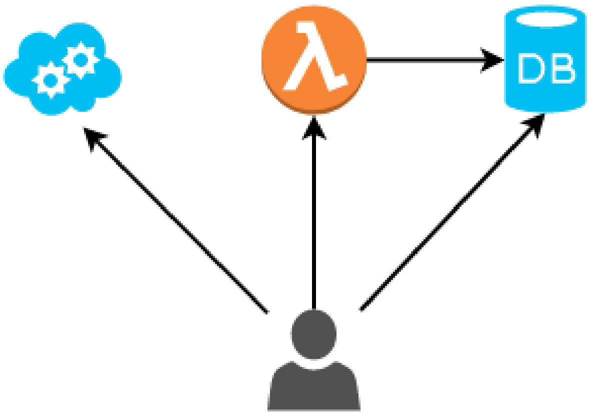

Figure 11. Thick Client

Solution: Create thicker, more powerful clients.

Serverless applications, as described in chapter 2, typically rely heavily on third party cloud services (BaaS) interspersed with custom logic in form of serverless functions. In a traditional layered web application architecture, as described by Roberts (2016), interaction with these external services would be handled by a server application that sits between the client and the service layers. Following this model, the client can be limited in functionality whereas the server application plays a larger role. Sbarski and Kroonenburg (2017) point out that the model of the backend as a gateway between client and services doesn't match well with the serverless paradigm. First of all, using FaaS as a middle layer in front of cloud resources directly translates into extra costs: on top of paying for the cloud service call, one has to pay for function invocation and execution for the duration of the network call as well as data transfer between the service and the FaaS provider. Secondly, a middle layer of FaaS results into extra network hops, thus increasing latency and reducing user experience. Sbarski and Kroonenburg (2017) thus advise against routing everything through a FaaS layer, and advocates building thick clients that communicate directly with cloud services and orchestrate workflows between them.

In addition to the cost benefit, a thicker client has the advantage of improved changeability and separation of concerns, as the single monolithic backend application is replaced by more isolated and self-contained components. Doing away with the central arbiter of a server application does come with its trade-offs, such as a need for distributed monitoring and further reliance on the security of the cloud services. Importantly not all functionality can or should be moved to the client: security, performance or consistency requirements among others can necessitate a server-side implementation. (Roberts 2016).

The Thick Client pattern depends on fine-grained, distributed, request-level authentication in lieu of a gatekeeper server application. This follows naturally from the way serverless functions operate: being stateless and continuously scaling up and down, maintaining a session between the backend and the cloud services is infeasible. Instead of automatically trusting all requests originating from the backend, each cloud service request has to be individually authorized. From a cloud service's point of view, requests originating from a serverless function or directly from the client are both equally untrusted. Hence in serverless architectures, skipping the backend layer is preferable whenever a direct connection between client and services is possible. The Valet Key pattern in section 3.4.2 describes one example of a request-level authentication mechanism. (Adzic and Chatley 2017)

3.2 Event patterns

Asynchronous messaging/event patterns.

3.2.1 Event processing

Trigger a function as a result of event occurrence.

3.2.2 Periodic invocation

Schedule function invocations using cron-like systems.

3.2.3 Fan-out events

Trigger multiple actions from a single event.

3.2.4 Fan-out/fan-in

Split event handling into parallel functions.

3.2.5 Pipes and filters

Handle event stream with a pipeline of small functions.

3.3 API/Integration patterns

Integrating with external systems.

3.3.1 API composition

Hide multiple API calls under a single function.

3.3.2 API aggregation

Hide a sequential multi-step API call under a single function.

3.3.3 API async

Turn a synchronized API into an async one.

3.3.4 Legacy API Proxy/Staged migration

Replace a legacy API with a new one step by step, a.k.a. Strangler.

3.3.5 Separate FaaS handler from core logic

Separate FaaS handler core logic in code level.

3.4 Data management/access patterns

Managing state and accessing external resources.

3.4.1 Externalized State

Store function state in external storage.

3.4.2 Valet Key

Sign tokens for clients to directly access resources.

3.4.3 Least privilege IAM role

Minimize attack surface by reducing function access roles to bare minimum.

3.5 Performance and scalability patterns

Address FaaS performance issues.

3.5.1 Function warming

Ping a function intermittently to avoid cold starts.

3.5.2 Oversized function

Choose maximum memory allocation to access faster CPU resources and improve cold start latency.

3.5.3 Singleton

Take advantage of function execution context to avoid reinitializing function dependencies.

3.6 Resiliency and availability patterns

Maximize serverless system resiliency.

3.6.1 Bulkhead

Isolate high-latency code into separate functions to avoid resource contention.

3.6.2 Flow control/throttling

Throttle invocations to avoid DDoSing yourself.

3.6.3 Circuit breaker

Keep track of component availability to avoid cascading failures.

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.

Al-Ali, Zaid, Sepideh Goodarzy, Ethan Hunter, Sangtae Ha, Richard Han, Eric Keller, and Eric Rozner. 2018. “Making Serverless Computing More Serverless”. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD),* 456–459. IEEE.

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. 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.

Cloudflare. 2018. “Cloudflare Workers”. Visited on November 21, 2018. https://www.cloudflare.com/products/cloudflare-workers/.

CNCF. 2018. *Serverless whitepaper.* Technical report. Cloud Native Computing Foundation. Visited on November 13, 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.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of reusable object-oriented software.*

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/.

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).* Boston, MA: USENIX Association. https://www.usenix.org/conference/hotcloud18/presentation/hong.

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.

ISO. 2014. *ISO/IEC 17788:2014 Information technology – Cloud computing – Overview and vocabulary.* Standard. International Organization for Standardization.

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.

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, 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.

López, Pedro García, Marc Sánchez Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto. 2018. “Comparison of Production Serverless Function Orchestration Systems”. *CoRR* abs/1807.11248.

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.

OWASP. 2018. *OWASP Top 10 (2017) Interpretation for Serverless.* Technical report. Open Web Application Security Project.

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.

Rotem-Gal-Oz, Arnon. 2012. *SOA patterns.* Manning.

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.

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

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

Spillner, Josef. 2017b. “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 Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, et al. 2017. “Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures”. *Service Oriented Computing and Applications* 11 (2): 233–247.

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.