Aleksi Pekkala

Migrating a web application to serverless architecture

Master's Thesis in Information Technology

May 7, 2019

University of Jyväskylä

Faculty of 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: 100+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

Figure 1. Comparison of a) virtual machine- and b) container-based deployments (Bernstein 2014) 7
Figure 2. A history of computer science concepts leading to serverless computing (E. van Eyk et al. 2018) 12
Figure 3. Degree of automation when using serverless (Wolf 2016) 16
Figure 4. Serverless and FaaS vs. PaaS and SaaS (Erwin van Eyk et al. 2017) 17
Figure 5. Serverless processing model (CNCF 2018) 18
Figure 6. Evolution of sharing – gray layers are shared (Hendrickson et al. 2016) 19
Figure 7. IBM OpenWhisk architecture (Baldini, Castro, et al. 2017) 24
Figure 8. Pattern language 37
Figure 9. Routing Function 38
Figure 10. Function Chain 39
Figure 11. Fan-out/Fan-in 41
Figure 12. Externalized State 42
Figure 13. State Machine 43
Figure 14. Thick Client 44
Figure 15. Event Processor 46
Figure 16. Periodic Invoker 48
Figure 17. Polling Event Processor 49
Figure 18. Event Broadcast 50
Figure 19. Aggregator 51
Figure 20. Proxy 53
Figure 21. Strangler 54
Figure 22. Valet Key 55
Figure 23. Function Warmer 57
Figure 24. Singleton 59
Figure 25. Bulkhead 60
Figure 26. Throttler 62
Figure 27. Circuit Breaker 64
Figure 28. Image Manager components 68
Figure 29. Image Manager upload sequence 69
Figure 30. Serverless Image Manager components 73
Figure 31. Serverless Image Manager upload sequence (steps 2.1-2.3 run in parallel) 74
Figure 32. Async Response 75
Figure 33. Serverful Image Manager load test results 82
Figure 34. Serverless Image Manager load test results 83

List of Tables

Table 1. Eight issues to be addressed in setting up an environment for cloud users. (Jonas et al. 2019) 9

List of Listings

Listing 2.1. Example FaaS handler in Python 20
Listing 5.1. Image labeler function handler 80

Contents

1 INTRODUCTION 1
1.1 Research problem 2
1.2 Outline 2

2 SERVERLESS COMPUTING 4
2.1 Background 5
2.2 Defining serverless 10
2.3 Backend-as-a-Service and Function-as-a-Service 13
2.4 Comparison to other cloud computing models 15
2.5 FaaS processing model 18
2.6 Use cases 21
2.7 Service providers 23
2.8 Security 25
2.9 Economics of serverless 27
2.10 Drawbacks and limitations 30

3 SERVERLESS DESIGN PATTERNS 37
3.1 Composition patterns 38
3.1.1 Routing Function 38
3.1.2 Function Chain 39
3.1.3 Fan-out/Fan-in 40
3.1.4 Externalized State 42
3.1.5 State Machine 42
3.1.6 Thick Client 44
3.2 Event patterns 45
3.2.1 Event Processor 46
3.2.2 Periodic Invoker 48
3.2.3 Polling Event Processor 48
3.2.4 Event Broadcast 50
3.3 Integration patterns 51
3.3.1 Aggregator 51
3.3.2 Proxy 53
3.3.3 Strangler 54
3.3.4 Valet Key 55
3.4 Availability patterns 56
3.4.1 Function Warmer 57
3.4.2 Singleton 59
3.4.3 Bulkhead 60
3.4.4 Throttler 62
3.4.5 Circuit Breaker 64

4 MIGRATION PROCESS 67
4.1 Image Manager 67

4.2 Serverless Image Manager 71
4.2.1 Pattern selection 72
4.3 New patterns 74
4.3.1 Async Response 75
4.3.2 Task Controller 76
4.3.3 Local Threading 77
4.3.4 Prefetcher 78
4.3.5 Throttled Recursion 79

5 EVALUATION 80
5.1 Developer perspective 80
5.2 Performance perspective 82
5.3 Economic perspective 82

6 CONCLUSION 84

BIBLIOGRAPHY 85

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*, a novel form of cloud computing based on ephemeral 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 2018a; Google 2018; IBM 2018; Microsoft 2018b). A number of high-profile use cases have been presented in the literature (CNCF 2018), and some researchers have gone as far as to predict that "serverless computing will become the default computing paradigm of the Cloud Era, largely replacing serverful computing and thereby bringing closure to the Client-Server Era" (Jonas et al. 2019). 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 applications?
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 a 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 applications.

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, Grance, et al. 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 compu-

tational resources on demand.

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

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

Virtual machines manage to provide strong security guarantees by isolation, i.e., by allocating each VM its own set of resources with minimal sharing between the host system. Minimal sharing however translates into high memory and storage requirements as each virtual machine requires a full OS image in addition to the actual application files. A virtual machine also has to go through the standard OS boot process on startup, resulting in 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 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)

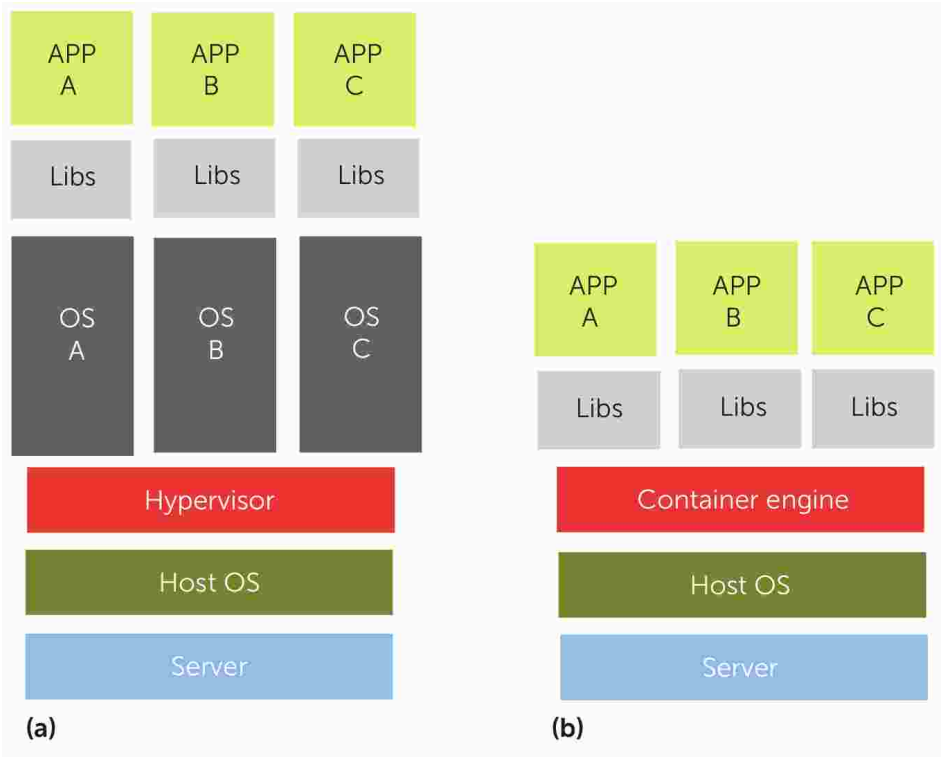


Figure 1: Comparison of a) virtual machine- and b) container-based deployments (Bernstein 2014)

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 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 here 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 actual computation took 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)

Another cloud computing shortfall concerns operational overhead. In an influential paper on the prospects of cloud computing, Armbrust et al. (2009) foresaw simplified operations as one of the model's potential advantages, hypothesizing reduced operation costs and seamless elasticity. However in a recent follow-up paper Jonas et al. (2019) observe a failure in realizing this advantage, with cloud users continuing to "bear a burden of complex operations" (the other observed shortfall concerns utilization rates as described above). Leading to this outcome was the marketplace's eventual embrace of low-level cloud resources such as virtual machines in favour of cloud-native ones like Google's PaaS offering, which in turn resulted from the early cloud adopters practical need of porting on-premise applications to a familiar computing environment. In consequence, "cloud computing relieved users of physical infrastructure management but left them with a proliferation of virtual resources to manage". To illustrate this problem the authors list a number of operational tasks required to spin up an elastic cloud environment in table 1. In case of a simple web service, the development work required to accomplish these tasks can be manyfold compared to the actual application logic.

1. Redundancy for availability, so that a single machine failure doesn't take down the service.
2. Geographic distribution of redundant copies to preserve the service in case of disaster.
3. Load balancing and request routing to efficiently utilize resources.
4. Autoscaling in response to changes in load to scale up or down the system.
5. Monitoring to make sure the service is still running well.
6. Logging to record messages needed for debugging or performance tuning.
7. System upgrades, including security patching.
8. Migration to new instances as they become available.

Table 1: Eight issues to be addressed in setting up an environment for cloud users. (Jonas et al. 2019)

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.

In a partially overlapping definition, Jonas et al. (2019) describe serverless computing by specifying three crucial distinctions between it and conventional serverful cloud computing:

1. *Decoupled computation and storage*: the storage and computation scale separately and are provisioned and priced independently. In general, the storage is provided by a separate cloud service and the computation is stateless.
2. *Executing code without managing resource allocation*: instead of requesting resources, the user provides a piece of code and the cloud automatically provisions resources to

execute that code.

3. *Paying in proportion to resources used instead of for resources allocated*: billing is by some dimension associated with the execution, such as execution time, rather than by a dimension of the base cloud platform, such as size and number of VMs allocated.

Fundamentally serverless computing is about building and running back-end code that does not require server management or long-lived server applications. Sbarski and Kroonenburg (2017) summarize "the ultimate goal behind serverless" as "moving away from servers and infrastructure concerns, as well as allowing the developer to primarily focus on code". Jonas et al. (2019) in turn draw parallels between higher-level programming languages and serverless computing: just like a high-level programming language frees developers from manually selecting registers and loading values in and out of them, the serverless paradigm frees developers from manually reserving and managing computing resources in the cloud. The term *serverless* itself can seem disingenuous, since the model evidently still involves servers. The industry-coined name instead carries the meaning that operational concerns are fully managed by the cloud service provider. As tasks such as provisioning, maintenance and capacity planning (listed in table 1) 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 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 services. E. van Eyk et al. (2018) see serverless computing continuing this trend of service specialization and abstraction, preceded by service-oriented architecture (SOA) and later by microservices. Finally the transition from synchronous systems to concurrent, event-driven distributed sys-

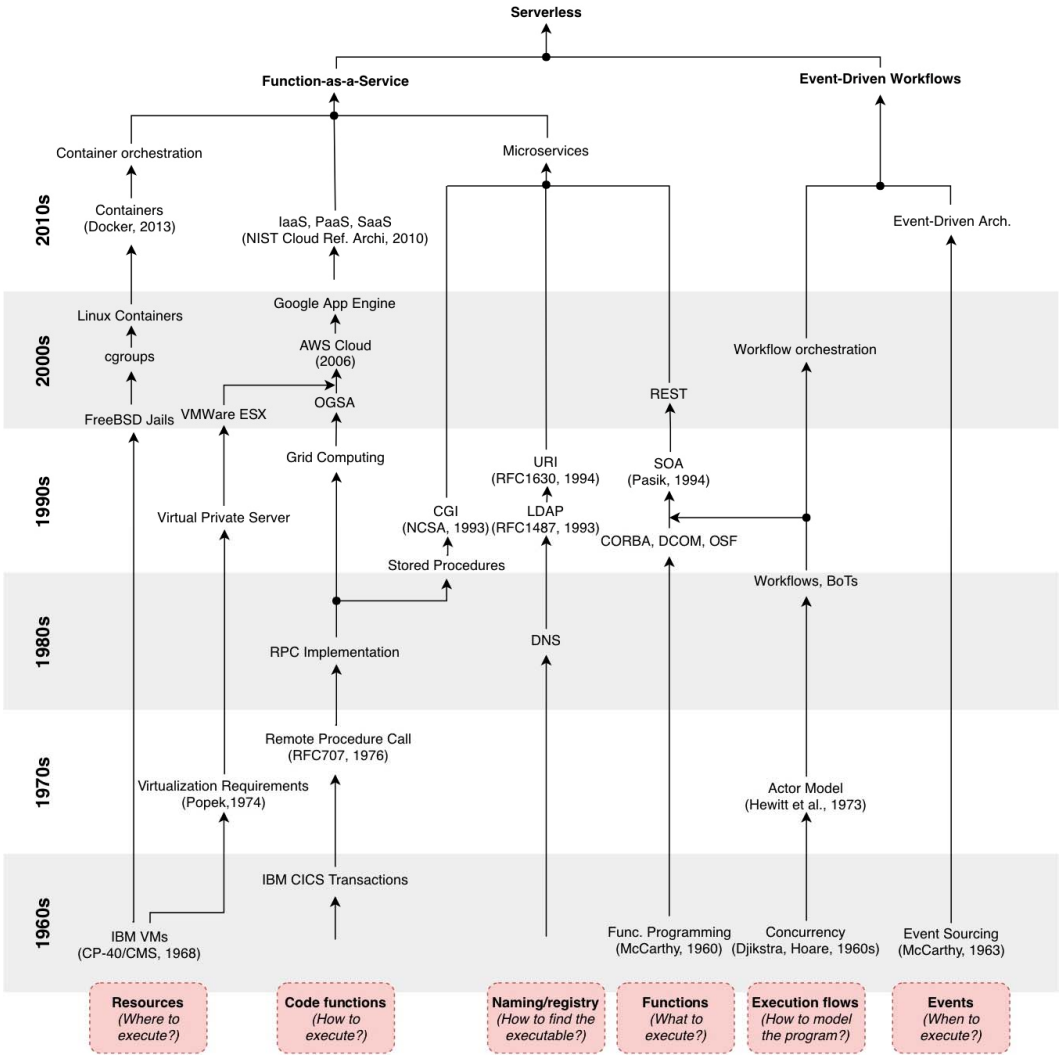


Figure 2: A history of computer science concepts leading to serverless computing (E. van Eyk et al. 2018)

tems 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".

Sbarski and Kroonenburg (2017) similarly view serverless architecture, along with microservices, as "spiritual descendants of service-oriented architecture". SOA is an architectural style where systems are composed out of many independent and loosely coupled services that communicate via message passing. Serverless architecture retains the SOA principles of service reusability, autonomy and composability while "attempting to address the complex-

ity of old-fashioned service-oriented architectures" – a reference to specifications like SOAP, WSDL and WS-I that SOA is often associated with although being nominally technology-independent. One area where serverless architecture diverges from SOA is service size: in SOA context, fine service granularity is considered problematic due to the management and performance overhead incurred. Rotem-Gal-Oz (2012) distills the problem into the Nanoservice antipattern: "a service whose overhead (communications, maintenance and so on) outweighs its utility". Serverless platforms on the other hand aim to reduce this overhead and thus tip the scale towards smaller services. Adzic and Chatley (2017) make a similar observation on how the novel technical qualities of serverless platforms drive architectural decisions: "without strong economic and operational incentives for bundling, serverless platforms open up an opportunity for application developers to create smaller, better isolated modules, that can more easily be maintained and replaced".

2.3 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 2018a). 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 2.5. 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 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, which is where FaaS functions step in (Roberts 2016). Serverless applications can utilize both models simultaneously, with BaaS platforms generating events that trigger FaaS functions, and FaaS functions acting as 'glue components' 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 the '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.

2.4 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, Grance, et al. 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, Grance, et al. 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 substantial differences between the two models, which boil down to PaaS being an instance-based model with multiple server processes running on always-on server instances, as opposed to the on-demand resource allocation of FaaS. Put another way, "most PaaS applications are not geared towards bringing

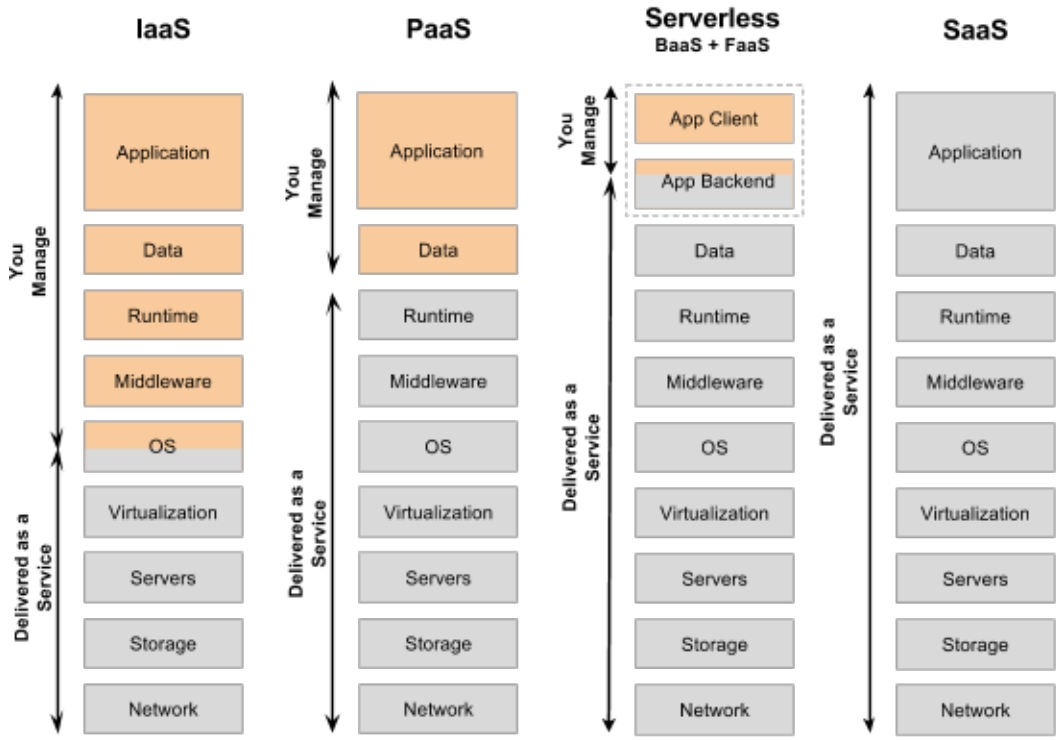


Figure 3: Degree of automation when using serverless (Wolf 2016)

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

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

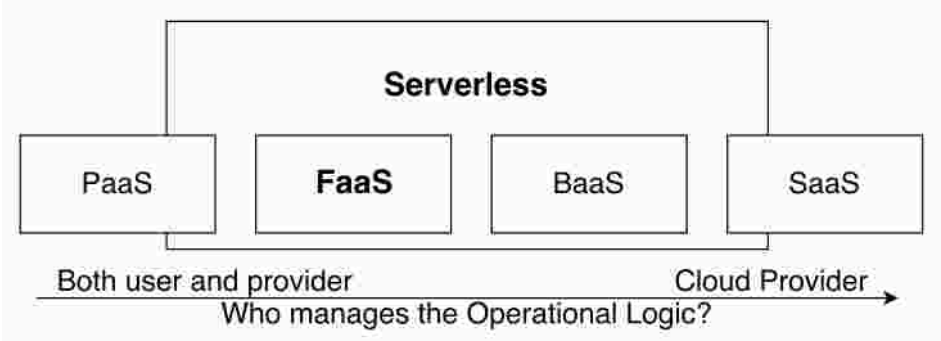


Figure 4: Serverless and FaaS vs. PaaS and SaaS (Erwin van Eyk et al. 2017)

times. It's also to be noted that PaaS doesn't suffer from limits on execution duration and some of the other limitations of FaaS described in section 2.10.

Another recent cloud computing technology gaining rapid adoption is *container orchestration*, also referred to as Containers-as-a-Service or CaaS (CNCF 2018). Using a container orchestration tool like Docker Swarm, Mesos or Kubernetes, the developer sets up a cluster of infrastructure resources which can then be used as a deployment target for containerized applications, with additional facilities for scaling and monitoring. The model enables maximum control over what's being deployed and on which resources, as well as enabling portability between different cloud vendors and on-premise infrastructure. Of course, greater control of underlying resources comes with the downside of operational responsibility. As to how container orchestration relates to serverless, Jonas et al. (2019) sums up the former as "a technology that simplifies management of serverful computing" whereas the latter "introduces a paradigm shift that allows fully offloading operational responsibilities to the provider". In a similar vein Roberts (2016) contends that what's true with PaaS still holds with CaaS: tools like Kubernetes lack the automatically managed, transparent, and fine grained resource provisioning and allocation of FaaS. The author however observes convergence in this space, and indeed a number of serverless platforms have been implemented on top of container orchestration platforms.

2.5 FaaS processing model

The CNCF (2018) whitepaper divides a generalized FaaS platform into four constituents 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.

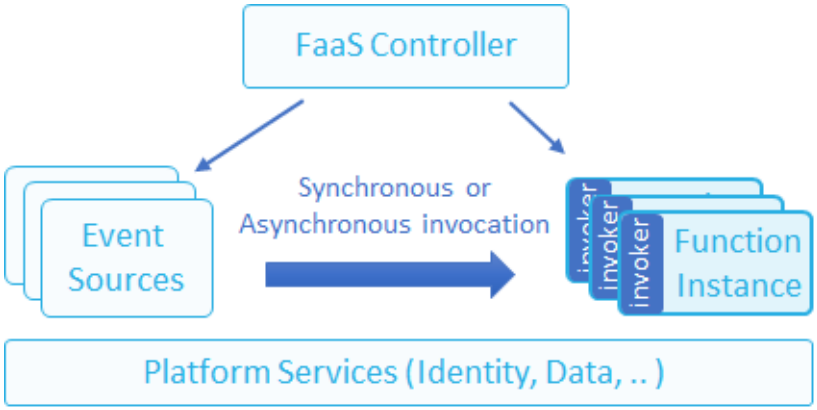


Figure 5: Serverless processing model (CNCF 2018)

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 controller 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 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 2018a)

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).

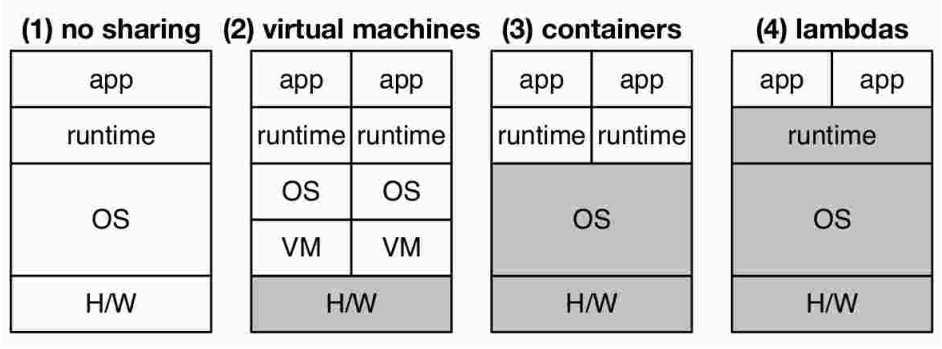


Figure 6: Evolution of sharing – gray layers are shared (Hendrickson et al. 2016)

Being stateless and short-lived, serverless functions have fundamentally limited expressiveness compared to a conventional server application. This is a direct result of being built to maximise scalability. A FaaS platform will need to execute the arbitrary function code 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 means that a function loses all local state after termination: none of the local state created during invocation will necessarily be available during subsequent or parallel invocations of the same function. This is where BaaS services come in, with external stateful services such as key-value stores, databases and file 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 2018a).

FaaS event sources can be divided into two categories of 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 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 includes request path, headers, body and query parameters. A function instance is also 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 support user-defined 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 2.1. In addition to publishing and executing serverless functions, FaaS platforms provide auxiliary capabilities such as monitoring, versioning and logging. (CNCF 2018)

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

Listing 2.1: Example FaaS handler in Python

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 (Wes 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 2018b).

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).

2.6 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. Finally, an AWS whitepaper on serverless economics includes industry use-cases ranging from financial institutions Fannie Mae and FINRA to Autodesk and Thomson Reuters (AWS 2017).

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.

2.7 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. AWS Lambda was the first platform to roll out support for custom runtimes in late 2018, which enables writing serverless functions with virtually any language (AWS 2018a). The serverless platforms of the big cloud service providers, Amazon, Google, Mi-

crosoft 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 2017).

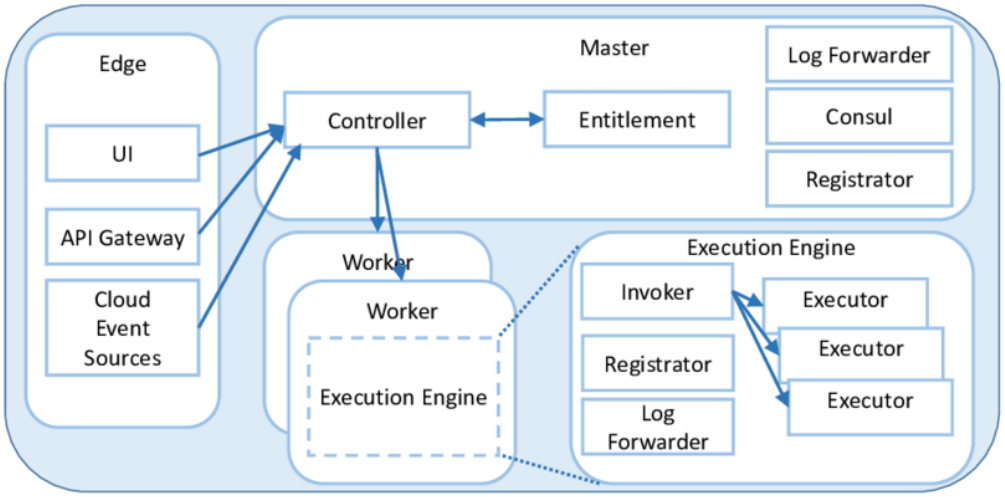


Figure 7: IBM OpenWhisk architecture (Baldini, Castro, et al. 2017)

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 predictably 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.

2.8 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 re-evaluating 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 third-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".

2.9 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 2018a). 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 2018b; 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 favour of FaaS, with more than a 50% cost reduction compared to self-operated microservices and up to a 77% re-

duction 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 overprovision 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.

2.10 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 several 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 difficulty 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). One solution to tackle the vendor lock-in problem is to utilize a serverless framework. Kritikos and Skrzypek (2018) review a number of frameworks that either "abstract away from serverless

platform specificities" or "enable the production of a mini serverless platform on top of existing clouds" and thus aim for provider-independency. 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 (2018a) and Microsoft (2018b) 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 2.5. 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. Wes 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. In another study, Jackson and Clynch (2018)

discover significant differences on performance between the different language runtimes on AWS Lambda and Azure Functions. The top performers in terms of "optimum performance and cost-management" were found to be Python on AWS Lambda and C# .NET on Azure Functions.

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 challenges crucial for more general adoption of FaaS, 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".

(Hellerstein et al. 2018) present a pointed critique of serverless computing, concluding that current first-generation serverless architectures fall short of the vision of utility computing. "One step forward, two steps back" in terms of cloud innovation, serverless computing fails to enable developers to seamlessly harness the practically unlimited storage and processing power of the cloud. First of all the authors observe that FaaS functions, running on isolated VMs separate from data, are an architectural anti-pattern: FaaS "ships data to code" instead of "shipping code to data", a bad design decision in terms of latency and bandwidth. Second, FaaS functions are limited in terms of distributed computing since they offer no network addressability: a function cannot directly communicate with another function instance, which rules out any design based on concurrent message-passing and distributed state. The approach FaaS takes is to rely on external shared state storage for exchanging data between functions, which means that all communication passes through cloud storage. The authors note that "communicating via cloud storage is not a reasonable replacement for directly-addressed networking" since it is "at least one order of magnitude too slow." Finally the

authors see FaaS discouraging innovation in both hardware and open source, as serverless platforms run on fairly uniform virtual machines and lock users into proprietary services. Having said that, the authors concede that some constraints inherent to FaaS can in fact benefit cloud innovation. For example the lack of guarantee over sequential execution or physical hardware locality across functions can lead to more general-purpose program design. The critique finishes with a set of challenges to be addressed by next-generation serverless platforms: data and code colocation, heterogeneous hardware support, long-running addressable software agents, new asynchronous and granular programming language metaphors and improvements in service-level guarantees and security.

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 (AWS 2018b; Microsoft 2018a). Literature on object-oriented patterns (OOP) (Gamma et al. 1994), SOA patterns (Rotem-Gal-Oz 2012), cloud design patterns (Microsoft 2018a) as well as enterprise integration patterns (EIP) (Hohpe and Woolf 2004) was also reviewed for applicable practices.

The patterns are grouped into four categories for better readability. How the patterns fit together is sketched out in figure 8. These interrelations form a *pattern language*, i.e. a structural organization of pattern relations (Rotem-Gal-Oz 2012).

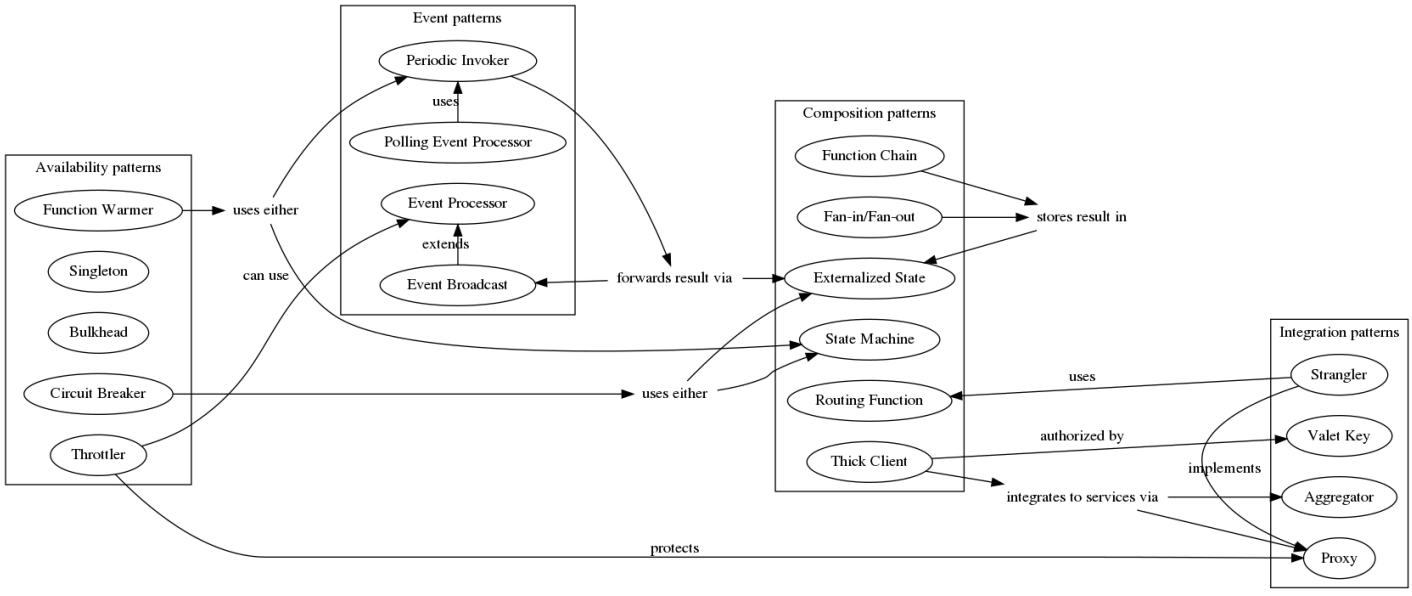


Figure 8: Pattern language

3.1 Composition patterns

How to compose and orchestrate serverless functions together into more expansive sequences or workflows?

3.1.1 Routing Function

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

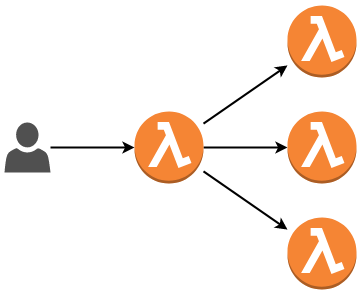


Figure 9: Routing Function

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 2018a). These tools however are mostly limited to path-based routing, whereas a routing function can be implemented to support more dynamic use cases. Also notably, 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). From a client's point of view, the Routing Function has the benefit of abstracting backend services so that calls can be rerouted to different services without changing client implementation; this can be put to use for example in A/B testing by partially rolling out new updates to selected clients (Microsoft 2018a).

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 at function code level, information about function control flow gets hidden in implementation rather than being accessible from configuration (Leitner et al. 2018). Also, like any centralized service, the Routing Function can potentially introduce a single point of failure or a performance bottleneck (Microsoft 2018a).

The Routing Function resembles the OOP Command pattern which is used to decouple caller of the operation from the entity that carries out the processing via an intermediary command object (Gamma et al. 1994). A related EIP pattern is the 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). Also pertinent to the serverless Routing Function, Hohpe and Woolf (2004) caution that the Content-Based Router should be made easy to maintain as it can become a point of frequent configuration. Finally, Microsoft's cloud design patterns includes the Gateway Routing pattern that's similarly employed to "route requests to multiple services using a single endpoint" (Microsoft 2018a).

3.1.2 Function Chain

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

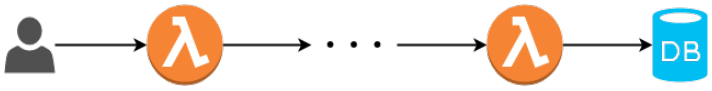


Figure 10: Function Chain

Solution: Split the task into separate function invocations that are chained together sequen-

tially.

The Function Chain comprises of an initial function invocation and any number of subsequent invocations. The initial function begins computation while keeping track of remaining execution time. For example in AWS Lambda the execution context contains information on how many milliseconds are left before termination (AWS 2018a). 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 Fan-out/Fan-in

Problem: Resource limits on a single function lead to reduced throughput.

Solution: Split task into multiple parallel invocations.

As discussed above, serverless functions are limited both in execution duration as well as CPU and memory capacity. The Function Chain pattern (section 3.1.2) works around the former limitation but is still constrained by a single function's computing resources, which can result in prohibitively slow throughput for compute-intensive tasks. The Fan-out/Fan-in pattern is an alternative approach that takes advantage of serverless platforms' inherent parallelism. The pattern consists of a master function that splits the task into segments and then asynchronously invokes a worker function for each segment. Having finished processing,

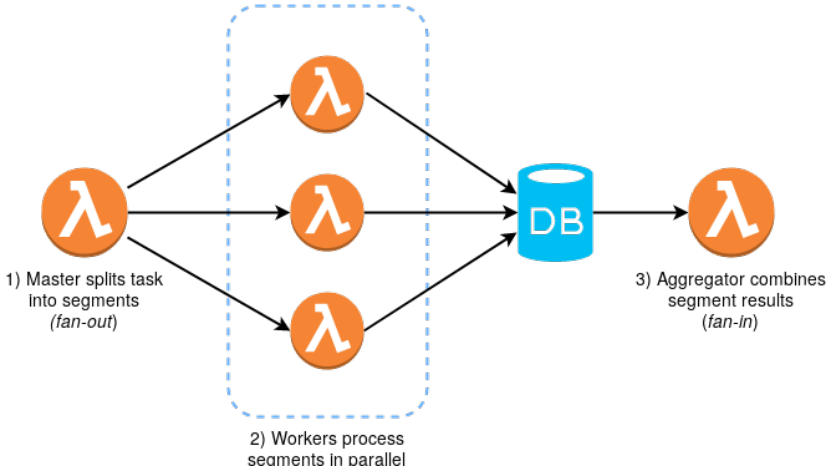


Figure 11: Fan-out/Fan-in

each worker function stores its result on a persistence layer, and finally an aggregator function combines the worker results into a single output value – although the aggregation step can be omitted in cases where intermediary results suffice. As each worker function invocation runs in parallel with its own set of resources, the pattern leads to faster completion of the overall task. (Zambrano 2018)

The Fan-out/Fan-in pattern lends itself well to tasks that are easily divisible into independent parts: the efficiency gained depends on the granularity of each subdivision. Conversely, an apparent limitation to the pattern is that not all tasks can be easily distributed into separate worker functions. McGrath et al. (2016) utilize the pattern in "easily and performantly solving a large-scale image resizing task". The authors point out how the pattern reduces development and infrastructure costs compared to a traditional multi-threaded application which "typically demands the implementation of a queueing mechanism or some form of worker pool". Lavoie, Garant, and Petrillo (2019) similarly study "the efficiency of a serverless architecture for running highly parallelizable tasks" in comparison to a conventional MapReduce solution running on Apache Spark, concluding that "the serverless technique achieves comparable performance in terms of compute time and cost".

Hohpe and Woolf (2004) present a similar approach to messaging with the EIP pattern of Composed Message Processor, which "splits the message up, routes the sub-messages to the appropriate destinations and re-aggregates the responses back into a single message."

3.1.4 Externalized State

Problem: How to share state between sequential or parallel serverless function instances?

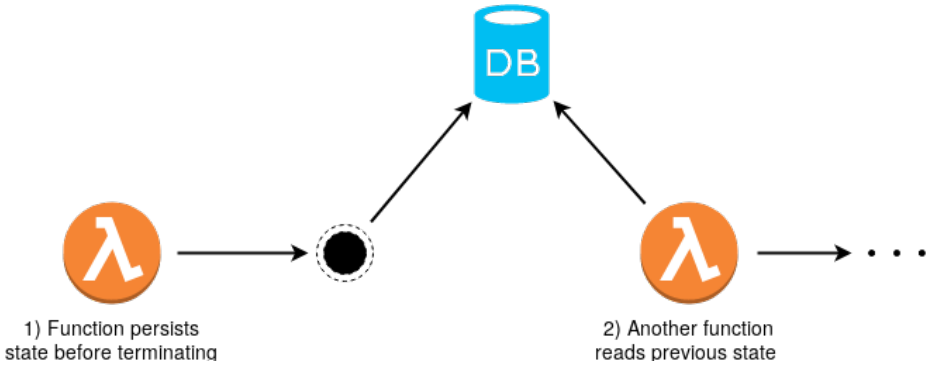


Figure 12: Externalized State

Solution: Store function state in external storage.

Serverless functions are, as discussed, stateless by design. Function instances are spawned and terminated ephemerally in a way that an instance has no access to any preceding or parallel instance state. Not all serverless use cases are purely stateless however, so being able to store and share state between function instances comes up as a common requirement. This is evidenced by a survey on serverless adoption in which two thirds of respondents reported at least sometimes applying the Externalized State pattern, making it by far the most common among the surveyed patterns (Leitner et al. 2018).

The Externalized State pattern is a fundamental pattern that consists of storing a function's internal state in external storage such as a database or a key-value store. The pattern is used to reliably persist state between sequential function invocations, and on the other hand to share state between parallel invocations. Imposing state on a stateless paradigm doesn't come free though, as relying on external storage induces latency and extra programming effort as well as the operational overhead of managing a storage component. (Leitner et al. 2018)

3.1.5 State Machine

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

Solution: Split a task into a number of discrete functions and coordinate their execution with

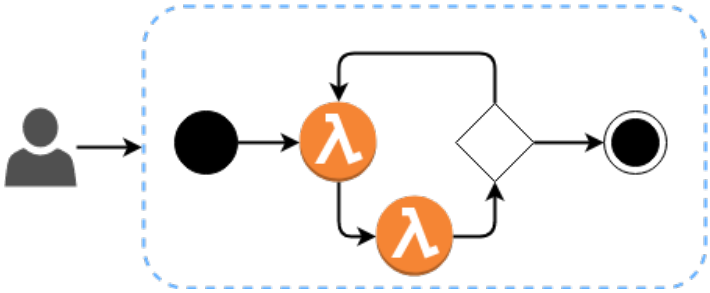


Figure 13: State Machine

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 13 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 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) compare 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 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 invocation.

A SOA pattern analogous to the State Machine is the Orchestrator, in which "an external workflow engine activates a sequence (simple or compound) of services to provide a complete business service". The Orchestrator aims to keep business processes agile and adaptable by externalizing them from service implementations: instead of hard-coding service interactions they are defined, edited and executed within a workflow engine. Used properly, the Orchestrator can add a lot of flexibility to the system. Difficulty however lies in implementing services as composable and reusable workflow steps while still keeping them useful as autonomous services. (Rotem-Gal-Oz 2012).

3.1.6 Thick Client

Problem: Routing client-service requests through an intermediary server layer causes extra costs and latency.

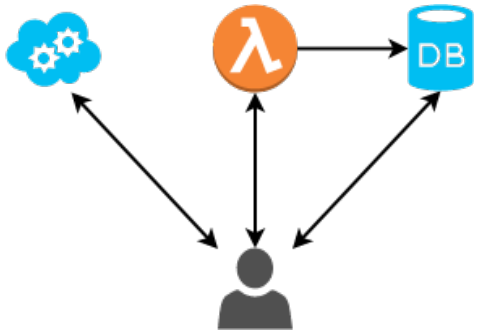


Figure 14: Thick Client

Solution: Create thicker, more powerful clients that directly access services and orchestrate workflows.

Serverless applications, as described in chapter 2, typically rely heavily on third-party cloud services (BaaS) interspersed with custom logic in form of FaaS functions. In a traditional three-tier web application architecture interaction with these external services would be handled by a server application that sits between client and service layers (Roberts 2016). 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 gatekeeper between client and services is in conflict 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 in extra network hops which increases latency and reduces user experience. The authors thus advise against routing everything through a FaaS layer, and advocate building thick clients that communicate directly with cloud services and orchestrate workflows between them.

In addition to improving cost and network efficiency, the Thick 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, including a need for distributed monitoring and further reliance on the security of third-party 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 3.3.4 describes one example of a request-level authentication mechanism. (Adzic and Chatley 2017)

3.2 Event patterns

How to handle asynchronous workflows triggered by external events?

3.2.1 Event Processor

Problem: How to execute a task on-demand upon event occurrence?

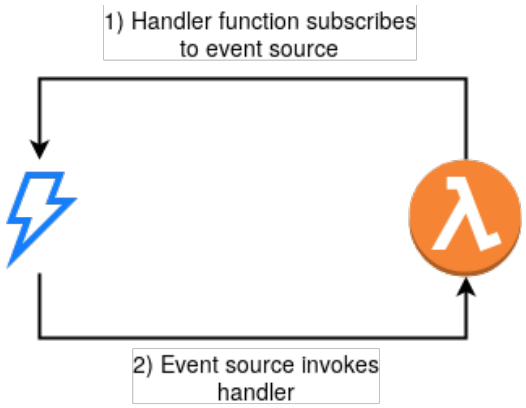


Figure 15: Event Processor

Solution: Subscribe a serverless function to a cloud event such as file upload or database change.

The Event Processor pattern consists of subscribing a serverless function to a cloud event source so that when the event occurs, the subscribed function gets invoked with access to the event context (Hong et al. 2018). Serverless platforms typically offer a number of integration points to events that originate from platform services. An AWS Lambda function, for example, can be triggered by file uploads, database change events, message queue and notification services, IoT events and others (AWS 2018a).

Baldini, Castro, et al. (2017) mention thumbnail generation triggered by image upload as an exemplary use case of serverless event processing: a bursty, compute-intensive task triggered on-demand by a cloud event. A traditional approach would be to implement a poller system that regularly checks for new images and generates thumbnails as images are detected. Such a system would require constant operation, and depending on polling interval the design leads to either extra network traffic or potentially long delay between event occurrence and processing. The design is especially wasteful in cases where new images come in infrequently. The Event Processor pattern, in turn, can bring considerable cost benefit in case of infrequent or irregular workflows as computation is only ran when necessary (Hong

et al. 2018). Another advantage is scalability, as functions are automatically invoked as per the number of events: a large number of events occurring at once leads to a similarly large number of functions executing in parallel (Hong et al. 2018).

The Event Processor has two counterparts among SOA patterns. In terms of scalability, a serverless Event Processor essentially implements the Service Instance pattern which involves "deploying multiple instances of service business logic" to address increased service load. Aptly, the Service Instance pattern is "best suited for stateless service implementations". Another related SOA pattern is the Inversion of Communications in which services eschew point-to-point communication in favour of event-driven architecture to reduce coupling between event sources and consumers. The pattern's downsides include the added complexity of designing a system as events and the difficulty of debugging complex event chains. (Rotem-Gal-Oz 2012)

The Event Processor can also be seen as a serverless form of the Event-Driven Consumer EIP pattern: a message consumer that sits dormant with no active threads until invoked by the messaging system. In essence, the pattern bridges the gap between external events and application-specific callbacks. A notable feature of an Event-Driven Consumer is that it automatically consumes messages as soon as they become available, which in effect means that the consumer has no control on its consumption rate: see the Polling Event Processor in section 3.2.3 for an alternative solution. (Hohpe and Woolf 2004)

Another point to keep in mind when implementing the Event Processor pattern is that some cloud event sources operate in at-least-once message delivery semantics: due to the highly distributed and eventually consistent nature of cloud platforms, events are guaranteed to be triggered at least once, not exactly once (AWS 2018a). This means that the triggered serverless function should in effect act idempotently, i.e. multiple executions with the same context should result in identical side effects. Hohpe and Woolf (2004) introduce a similar concept with the Idempotent Receiver pattern, a listener that can "safely receive the same message multiple times". The authors introduce two primary means for achieving idempotency: either explicit deduplication at the receiving end, or defining message semantics to support idempotency. The first approach calls for keeping track of the messages received thus far and ignoring any duplicates among incoming messages, leaving us with the problem

of where and for how long to store the message history. The alternative approach is to design messages themselves in a way that "resending the message does not impact the system": for example *set account balance to $10* instead of *increase account balance by $1*.

3.2.2 Periodic Invoker

Problem: How to execute a task periodically in predefined intervals?

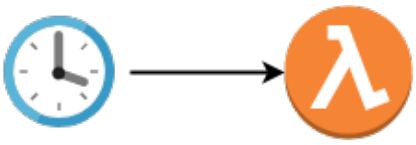


Figure 16: Periodic Invoker

Solution: Subscribe a serverless function to a scheduler.

The Periodic Invoker represents an arrangement where a serverless function is invoked periodically by a scheduler, akin to a cron task in Unix-based systems. First, the scheduler invokes the subscribed function according to its configuration. Second, the function carries out its task. Finally, after execution the function can report execution result out to a notification channel, store it in database or shut down if we're not interested in the outcome. The pattern is both conceptually simple and easy to implement, as all the major serverless providers offer integration to a cloud-based scheduler such as AWS CloudWatch Events (AWS 2018a). Potential use cases include periodical backups, compliance checks, service health checks, database cleanup tasks and other background jobs that are not latency-critical. (Hong et al. 2018)

3.2.3 Polling Event Processor

Problem: How to react to state change in an external service that doesn't publish events?

Solution: Use the Periodic Invoker to actively poll for state changes and trigger events accordingly.

The Event Processor pattern (section 3.2.1) is used to perform a task in reaction to some

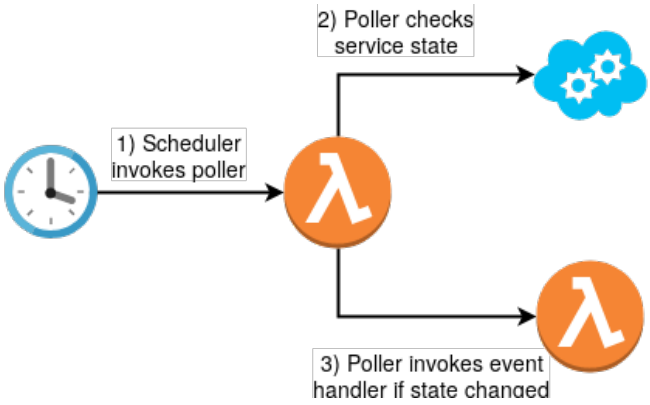


Figure 17: Polling Event Processor

state change in another system. The pattern depends on the external system to actively invoke the subscribed function when said state change occurs. Not all systems however are capable of performing such callbacks on state changes, which renders the pattern unusable in some cases. The Polling Event Processor works around this limitation by combining the Event Processor (section 3.2.1) and Periodic Invoker (section 3.2.2) patterns to essentially implement an event-driven integration point in front of a service where no such event source originally exists. The pattern consists of a Periodic Invoker that repeatedly checks the state of another service and performs a task when found state matches some condition. The task performed can be either implemented in the polling function itself or separated to another function that the poller invokes.

The Polling Event Processor is equivalent to the EIP pattern of Polling Consumer, where a receiver synchronously polls for a message, processes it and then polls for another. As well as offering eventful integration to non-eventful services, the pattern has the advantage of controlling its consumption rate. Whereas the Event Processor executes tasks as soon as events occur, the Polling Event Processor explicitly polls for new events when it is ready for them. The polling interval can also be configured to implement batching. As a downside, a sparse polling interval leads to increased latency between event occurrence and task execution. On the other hand a more fine-grained polling interval results in wasted resources when there are no events to consume. In short, "polling enables the client to control the rate of consumption, but wastes resources when there's nothing to consume." (Hohpe and Woolf 2004)

3.2.4 Event Broadcast

Problem: How to invoke multiple parallel functions as a result of a single event occurrence?

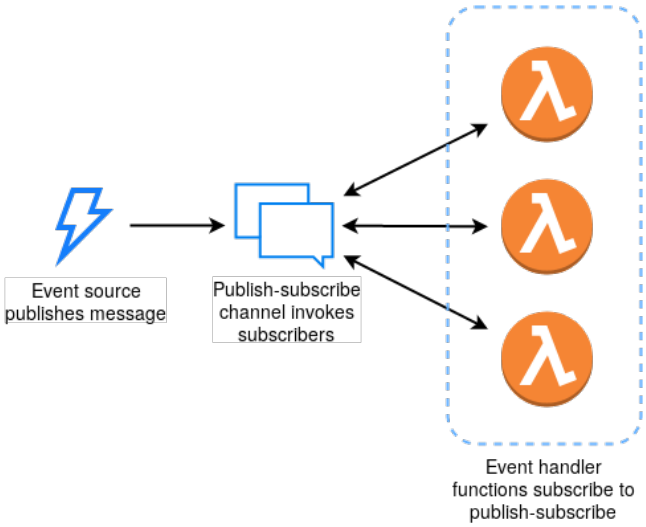


Figure 18: Event Broadcast

Solution: Subscribe multiple functions to a notification service, publish notification on event occurrence.

The Event Processor pattern (section 3.2.1) is applicable within cases of 1-to-1 relationship between events and tasks, as exemplified above with image upload triggering thumbnail generation. However in other cases a single event can result in multiple independent tasks. For example the image upload event could as well trigger a database update and a notification email, adding up to three self-contained and parallel tasks. Most event sources only support invoking one function per event which leaves us with a couple of options. First, we could set up a new function that subscribes to the image upload event and in turn asynchronously invokes any number of processor functions, as a sort of a parallel Routing Function . This approach comes with the operational overhead of needing to set up and maintain an additional function per each event broadcast. An alternative solution is to utilize a publish-subscribe channel such as the AWS Simple Notification Service (AWS 2018a). The key feature of a publish-subscribe channel is that any number of listeners can subscribe to a single channel, which can be used to overcome the 1-to-1 relationship between event sources and functions. Now instead of subscribing a function directly to an event source, functions subscribe to a

message channel that the event source publishes a message to upon event occurrence. In addition to achieving parallel fan-out to multiple functions, the pattern has the added benefit of loosed coupling between event sources and handler functions. (Sbarski and Kroonenburg 2017)

The Event Broadcast's object-oriented counterpart is the Observer: "define a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically" (Gamma et al. 1994). Just like above, the pattern decouples observers from the subject; that is, the object that we're interested in publishes its state regardless of the number of interested observers. The EIP pattern of Publish-Subscribe Channel expands the same decoupling to messaging: one input channel is split into multiple output channels, with each subscriber having its own channel and thus receiving its own copy of the original message (Hohpe and Woolf 2004).

3.3 Integration patterns

How to integrate to external – including legacy – systems?

3.3.1 Aggregator

Problem: An operation consists of multiple API requests, resulting in extra network hops between client and service.

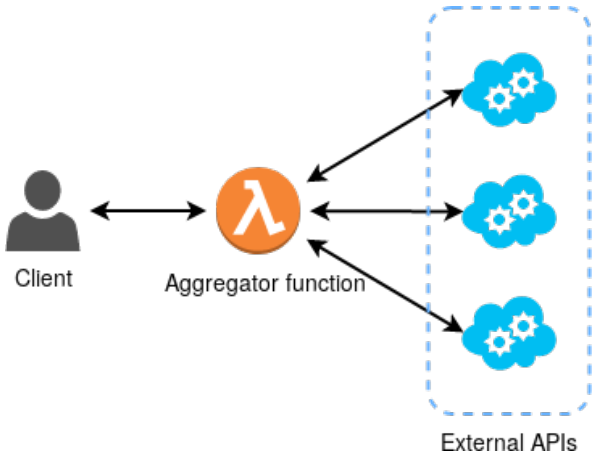


Figure 19: Aggregator

Solution: Aggregate multiple API requests under a single serverless function.

Service clients often need to deal with operations that involve performing several API calls, either in parallel or sequentially, and then filtering or combining the results. The operation might utilize multiple different services or just different endpoints of a single service. Baldini, Castro, et al. (2017) use the example of combining geolocation, weather and language translation APIs to render a localized weather forecast. Another example concerns a sequential multi-step API call of first fetching an API key, then resource location, and finally performing the actual operation. Composing operations out of multiple cross-service calls is a natural outcome of service oriented architectures, but incurs the penalty of extra resource usage and network latency in clients. The problem is further magnified in microservice and serverless architectures due to the fine service granularity. (Microsoft 2018a)

The Aggregator pattern consists of wrapping the required API calls into a single serverless function which is then exposed as a singular endpoint to clients. The Aggregator calls each target API and combines the results so that the client is left with a single network call, reducing the risk of network failure. Client resource usage is also reduced since any filtering or aggregation logic is offloaded to the Aggregator. Also ideally the Aggregator function is located near backend services to minimize network latency, and individual API responses are cached whenever possible. (Baldini, Castro, et al. 2017)

The Aggregator is largely equivalent to the Gateway Aggregation cloud design pattern (Microsoft 2018a). Baldini, Castro, et al. (2017) in turn split the pattern into API composition and API aggregation, for combined and sequential request flows respectively. It's worth noting that the Aggregator doesn't address the problems of network failure and incomplete requests, as the aggregating function might still encounter failed requests from downstream services. The pattern rather outsources the risk from service consumer to a backend service, working thus opposite to the Thick Client pattern's consumer-driven service orchestration (section 3.1.6). To ensure reliable operation when one of the API requests fails the Aggregator might internally implement the Compensating Transactions cloud design pattern, i.e. pairing each request with a compensating action that is performed in case of failure (Microsoft 2018a). The SOA patterns of Transactional Service and the more heavyweight Saga could also be used to enforce transactional guarantees inside the Aggregator (Rotem-Gal-Oz

2012).

3.3.2 Proxy

Problem: How to make a legacy service easier to consume for modern clients?

Figure 20: Proxy

Solution: Implement a serverless function as a proxy layer that translates requests between clients and the legacy service.

Applications often need to integrate to a legacy system for some resource or functionality. This requirement might present itself when an outdated but crucial system is in the process of being migrated, or cannot be migrated at all due to reasons of complexity or cost. Legacy systems might suffer from quality issues and use older protocols or data formats, which makes interoperation with modern clients problematic. A client would have to implement support for legacy technologies and semantics, which might adversely affect its own design goals. (Microsoft 2018a)

The serverless Proxy pattern essentially "makes legacy services easier to consume for modern clients that may not support older protocols and data formats" (Sbarski and Kroonenburg 2017). The pattern consists of a serverless function that acts as a proxy in front of the legacy service, handling any necessary protocol or data format translation and sanity checks. Conversely for client applications, the Proxy offers a clean and modern API for easier consumption. Sbarski and Kroonenburg (2017) use the example of offering a JSON API in front of a SOAP service. The pattern is also referred to as the Anti-Corruption Layer, alluding to how it works to contain a system's quality issues: "this layer translates communications between the two systems, allowing one system to remain unchanged while the other can avoid compromising its design and technological approach" (Microsoft 2018a).

3.3.3 Strangler

Problem: How to migrate an existing service to serverless architecture in a controlled fashion?

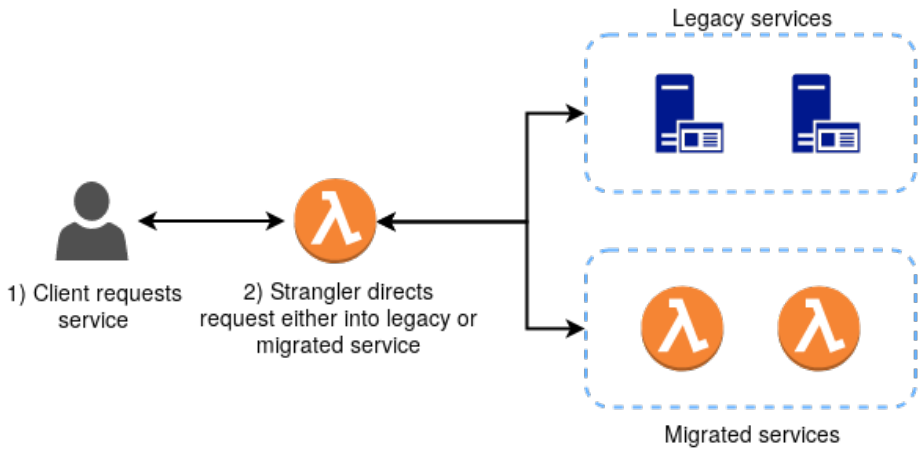


Figure 21: Strangler

Solution: Create a façade in front of the legacy API and incrementally replace individual routes with serverless functions.

Migrating an extensive application to serverless in one go could be a lengthy endeavour and lead to service downtime. Instead, it's often safer to perform a gradual migration where parts of an API are replaced one by one with the old system still running in the background and serving the yet to be migrated features. The problem with running two versions of the same API, however, is that clients need to update their routing every time a single feature is migrated. (Microsoft 2018a)

The Strangler solves the problem of gradual migration by first wrapping the whole legacy API behind a simple façade that initially just proxies requests to the legacy API as before. Then, as individual features are migrated to serverless, the façade's internal routing is updated to point to the serverless function instead of the legacy API. Thus "existing features can be migrated to the new system gradually, and consumers can continue using the same interface, unaware that any migration has taken place" (Microsoft 2018a). Eventually when all features have completed migration, the old system can be phased out. Zambrano (2018) proposes implementing the façade with an API gateway that matches and proxies all routes,

but the Routing Function pattern (3.2.4) is equally applicable here. The author also points out how the Strangler makes it easy to roll back a new implementation in case of any problems, and thus helps to reduce the risk in migration.

3.3.4 Valet Key

Problem: How to authorize resource access without routing all traffic through a gatekeeper server process?

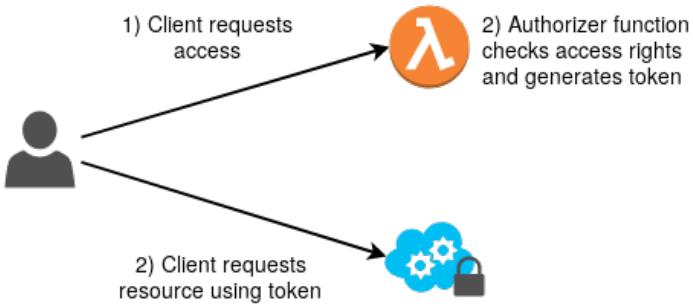


Figure 22: Valet Key

Solution: Let the client request an access token from an authorizer function, use the token to directly access a specific resource.

As put forth in section 3.1.6, serverless function instances do not form long-lived sessions with backend services which means that each service request must be individually authorized. With this in mind, routing client-service requests through a serverless function brings us no apparent security advantage, as both the client and the serverless function are equally untrusted from a service's point of view; on the contrary, having an extra server layer in the middle would only introduce additional latency and cost in data transfer (Adzic and Chatley 2017). The problem then becomes one of authorizing client-service communication without storing service credentials in the client and thus losing control of service access, and on the other hand without routing each request through the backend and thus in effect paying twice for data transfer.

One authorization pattern that fits the above requirements is the Valet Key. In this pattern the client, when looking to access a resource, first requests access from a special authorizer

serverless function. The authorizer function checks the client's access rights and then signs and returns an access token that is both short-lived and tightly restricted to this specific resource and operation. Now for any subsequent calls until token expiration, the client can call the resource directly by using the token as authentication. This removes the need for an intermediate server layer and thus reduces the number of network round-trips and frees up resources. At the same time the pattern avoids leaking credentials outside the authorizer function since the token's cryptographic signature is enough for the resource to validate request authenticity. (Microsoft 2018a)

The Valet Key relies heavily on cloud services' fine-grained authorization models, as the access token needs to be tightly restricted to a specific set of access permissions; for example read access to a single file in file storage or write access to a single key in a key/value store. Specifying the allowed resources and operations accurately is critical since granting excessive permissions could result in loss of control. Also, extra care should be taken to validate and sanitize all client-uploaded data before use since a client might have either inadvertently or maliciously uploaded invalid content. (Microsoft 2018a)

Adzic and Chatley (2017) raise a point about how the Valet Key model of direct client-resource communication can enable significant cost optimization in serverless architectures. For example in case of sending a file to a storage service like AWS S3, having a serverless function in the middle would require a high memory reservation to account for large files as well as paying for function execution time throughout file transfer. As the storage service itself only charges for data transfer, cutting out the middle man and sending files directly from the client reduces costs significantly. The authors emphasize that as FaaS costs "increase in proportion to maximum reserved memory and processing time [...] any service that does not charge for processing time is a good candidate for such cost shifting".

3.4 Availability patterns

How to guarantee availability in serverless systems and deal with the platform's performance constraints?

3.4.1 Function Warmer

Problem: The cold start phenomenon leads to high latency in infrequently invoked functions.

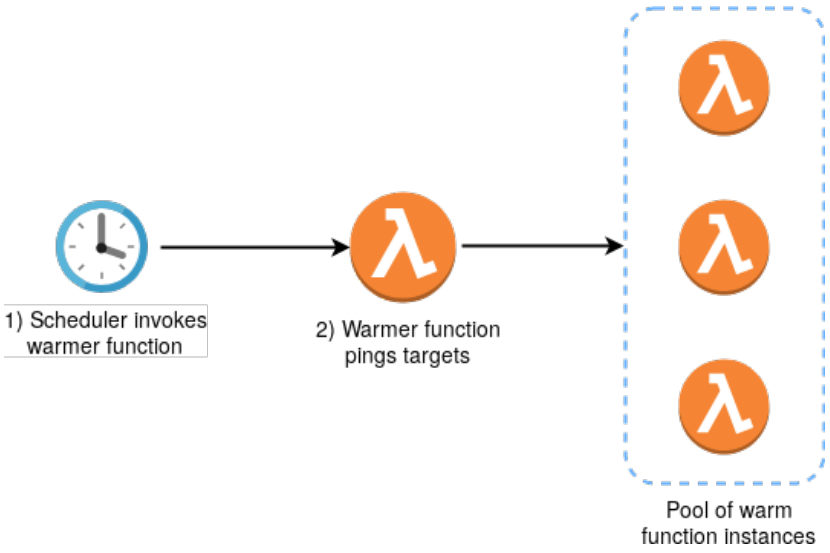


Figure 23: Function Warmer

Solution: Ping a function periodically to retain infrastructure and keep the function "warm".

Section 2.10 introduced cold starts as a major FaaS pain point. To reiterate, a cold start refers to an infrequently invoked function's start-up latency which results from its infrastructure allocation being deprovisioned after a period of inactivity. Wes Lloyd et al. (2018) for example observed a 15x increase in startup delay for cold starts as opposed to warm starts where existing infrastructure gets reused. While this problem is tied to limitations in current FaaS implementations and can be expected to be mitigated by advances in container technology, cold starts can hamper the adoption of FaaS technology in latency-critical applications.

The Function Warmer tackles the cold start problem by "pinging" (invoking) a function periodically with an artificial payload to prevent the FaaS platform from deprovisioning its infrastructure (Leitner et al. 2018). In detail, the pattern is implemented with a scheduled function (using the Periodic Invoker 3.2.2 or the State Machine 3.1.5) that invokes the target function in predefined intervals. Additionally, the target function's handler is instrumented with logic to handle warming invocations: the handler identifies the artificial payload and

replies accordingly without running the whole function. If the aim is to keep several concurrent function instances warm, the Function Warmer should invoke the same function multiple times with delayed executions. Now with the function warmed up and its infrastructure retained, any subsequent invocations can be served with minimal latency.

Leitner et al. (2018) note two drawbacks inherent to the pattern: the extra code needed in functions to manage warming invocations, and the invocation cost induced by pinging. The first point is largely a matter of tooling, and indeed many of the FaaS frameworks reviewed by Kritikos and Skrzypek (2018) are already equipped to handle pinging logic. As for the latter point, the cost depends on how long the platform retains idling containers for. Investigating infrastructure retention in AWS Lambda, Wes Lloyd et al. (2018) find that host containers are reused when invoked again within 5 minutes; a pinging request in 5 minute intervals then comes down to 8640 invocations per month which is well beyond the free tier offered by most platforms. Bardsley, Ryan, and Howard (2018) likewise note that "this approach will not significantly raise the cost of the deployment since calls to ping the relevant Lambdas would only have to be made infrequently". It's also notable that even with the extra pinging costs a FaaS solution can be more economical than a VM-based one: W. Lloyd et al. (2018), while leveraging the Function Warmer to retain a 100 concurrent function instances, observe a 400% improvement over cold function performance and a 17.6x reduction in hosting costs compared to VM instances.

A more fundamental issue with the Function Warmer pattern is its necessity in the first place, as ideally the problem of start-up latency would be done away with by the platform. Leitner et al. (2018) note that patterns such as the Function Warmer "can be seen as developers struggling with the inherent limitations of FaaS and working around them". Wes Lloyd et al. (2018) also point fingers to the platform side, urging cloud providers to "consider opportunistically retaining infrastructure for longer periods when there is idle capacity to help offset the performance costs of container initialization". The pattern represents, however, a practical solution to a real limitation that all FaaS platforms suffer from at least for the time being. Bardsley, Ryan, and Howard (2018) similarly conclude that "whilst this approach somewhat defeats the purpose of a system which should dynamically scale in response to demand it is nevertheless a viable strategy in mitigating the effect of cold Lambdas on over-

all latency".

3.4.2 Singleton

Problem: External dependencies like database connections are re-initialized upon each function invocation, leading to degraded performance.

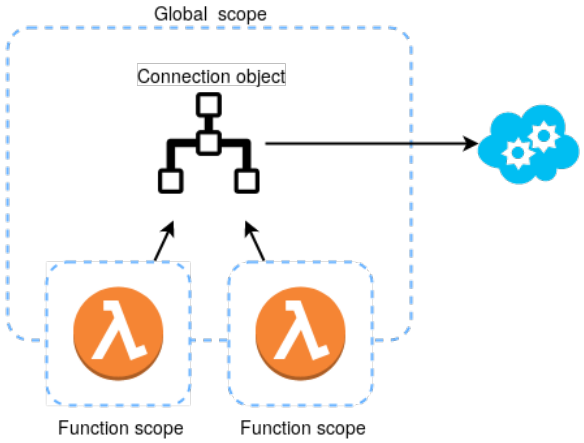


Figure 24: Singleton

Solution: Leverage global scope within your serverless function code to take advantage of container reuse.

Web applications often perform some sort of initialization as a part of their start-up process. Typical examples include connecting to a database and reading configuration or other static objects into runtime memory. In a conventional web application the cost of this initialization phase is negligible as it happens only once in the beginning of a long-running server process. Serverless function instances on the other hand are continuously torn down and recreated, so any delay in initialization has a significant impact on overall performance.

The way containers are reused by FaaS platforms enables us to mitigate the problem. As established earlier in section 2.10, after a function instance terminates the platform retains its container for a while in case of another invocation. The next function instance, when reusing this "warm" container, has access to the runtime variables already initialized during the previous execution. In detail, a reused container retains all of the global or static variables defined outside the function handler (an example of which is presented in listing 2.1).

The serverless Singleton pattern consists of reusing these pre-established variables and thus saving us the cost of re-initialization. For example, instead of reconnecting to a database in the beginning of each invocation, a single database connection can be shared by a number of subsequent instances. To implement the Singleton involves declaring such connections and other reusable objects in the global scope and having logic in place to check if a connection already exists before creating a new one. Like its object-oriented namesake (Gamma et al. 1994), the pattern ensures a single global instance per given resource as well as its "lazy initialization" i.e. creation on first use. (AWS 2018b)

3.4.3 Bulkhead

Problem: Multiple workflows in a single function leads to a high-latency execution path degrading the others' performance.

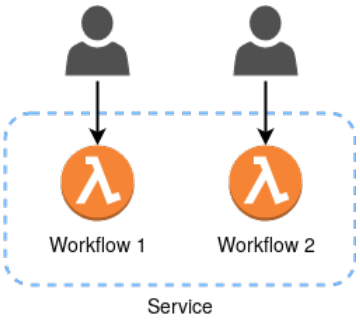


Figure 25: Bulkhead

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

A *bulkhead* refers to a sectioned partition in a ship's hull designed to prevent the whole ship from filling with water and sinking in case of an accident. Similarly in cloud computing context the Bulkhead pattern refers to isolating "elements of an application into pools so that if one fails, the others will continue to function" (Microsoft 2018a). The idea is to prevent a situation where a single service, when facing resource exhaustion due to a load peak or some sort of an application error, brings down all of its consumers with it causing a cascading failure and large-scale service interruption. The way this is achieved is by partitioning resources into self-contained blocks divided along lines of expected load and availability requirements: a separate service instance for each consumer, for example, or conversely a

separate set of consumer resources (such as threads) for each consumed service (Nygard 2007). Now even with one service unavailable the system as a whole can still deliver some level of functionality.

What are the serverless implications of this inter-service resource contention problem? It seems that FaaS platforms already ensure resource isolation as each function instance runs in its own resource-constrained container. Declaring the problem solved might be premature, however, namely for two reasons: first of all as shown by Wang et al. (2018), function instances, albeit isolated on a container-level, do suffer from resource contention on a VM level – a byproduct of some platforms' scheduling strategy of scaling a single function in the same VM. Secondly, a single function can contain multiple execution paths with varying latencies, which can lead to intra-service resource contention. Bardsley, Ryan, and Howard (2018) for example demonstrate customer data function with endpoints for both refreshing a token and simply requesting it: a case where the "asymmetrical nature of the performance profile between the refresh and request calls, with the refresh operation suffering significantly higher latency than the request call, could cause refresh calls to unnecessarily divert requests to cold Lambdas". Put another way, resource contention can occur inside a single FaaS function when an infrequently invoked high-latency workload degrades the performance of a more frequently invoked low-latency one.

While the VM-level resource contention is only addressable by platform providers, the function-level one is mitigated by a finer function granularity. A serverless Bulkhead pattern then comes to mean splitting high-latency workflows from a single function into their own functions. In the customer data example above, for example, this required "separating the request and refresh functionality into separate Lambdas to prevent high latency in one part of the system adversely affecting another" (Bardsley, Ryan, and Howard 2018). As per Adzic and Chatley (2017), the serverless paradigm already does away with economic incentives for bundling applications together in larger deployment units; likewise AWS best practices guide towards "smaller functions that perform scoped activities" (AWS 2018b). Maintaining a larger number of functions does still incur an operations overhead, however, and partitioning intra-function workflows might not be trivial.

3.4.4 Throttler

Problem: A rapidly scaling FaaS function overwhelms a non-scaling downstream service, resulting in function timeout.

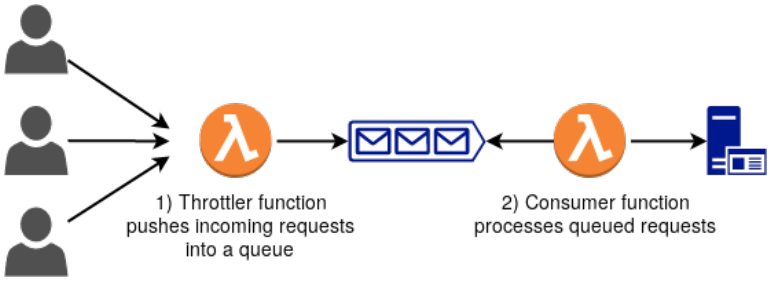


Figure 26: Throttler

Solution: Add a message queue in front of downstream service to throttle bursts of concurrent requests originating from FaaS.

Serverless platforms scale out horizontally in response to increased demand – up to a 1000 concurrent function instances in case of AWS Lambda (AWS 2018a). This configuration-free elasticity is a major FaaS selling point and plays well together with similarly autoscaling BaaS services. Scalability can turn into a pitfall, however, when interacting with conventional non-scaling resources, since a rapidly scaling FaaS function has the potential to overwhelm components that lack similar scaling properties or are otherwise not designed to accept high-volume traffic (CNCF 2018). This can have the effect of rendering the whole system unavailable by what is essentially a self-inflicted Denial of Service-attack. How would for example an on-premise database with strict connection limits fare with said 1000 concurrent instances? Avoiding non-scaling components in serverless architectures isn't feasible either as few systems have the luxury of not depending on any pre-existing components. Migrating these components into the cloud or deploying enough on-premise hardware to meet peak demand can also be prohibitively costly and organizationally difficult.

Addressing the problem then calls for some mechanism to protect downstream services from intermittent load spikes caused by FaaS scaling. The simplest solution is to apply a limit on the number of concurrent function instances – a configuration option most FaaS platforms provide. AWS documentation for example states that "concurrency controls are some-

times necessary to protect specific workloads against service failure as they may not scale as rapidly as Lambda", particularly in case of "sensitive backend or integrated systems that may have scaling limitations" and "database connection pool restrictions such as a relational database, which may impose concurrent limits" (AWS 2018b). While managing to ease the pressure on downstream services, this solution causes non-responsivity in consuming applications and arguably fails to take full advantage of the FaaS platform.

The Throttler represents a more sophisticated approach, based on buffering the bursts of concurrent requests originating from FaaS and resolving them asynchronously in a more controlled pace. A serverless adaptation of the Queue-Based Load Leveling cloud design pattern (Microsoft 2018a), the Throttler adds a message queue between the function and the downstream service so that upon a new request, instead of immediately invoking the service and waiting for a reply, the function pushes the request into the queue and responds to the consumer with an acknowledgement of receipt. It's then the job of another function to consume the queue and resolve service requests. By adjusting queue consumption rate, request spikes can be resolved in a smoother thus avoiding service resource exhaustion and function timeouts. A persistent queue also provides a further level of robustness as requests are not lost even in case of service outage.

Baldini, Castro, et al. (2017) utilize the Throttler to "control the flow of data between two services" in a hypothetical issue tracking system, further improving on the queue consumer's network efficiency by implementing batching. Rotem-Gal-Oz (2012) in turn presents an equivalent SOA pattern, the Decoupled Invocation: "acknowledge receipt at the service edge, put the request on a reliable queue, and then load-balance and prioritize the handler components that read from the queue". The pattern helps with the coupling and performance bottleneck problems of request-reply communication but comes with the downside of latency. Also since message queues are a one-way communication mechanism, it may be necessary to utilize a pattern like the Event Processor (section 3.2.1) for the consumer to access the actual service response. Finally, the Throttler can be seen analogous to the Service Activator EIP pattern which consists of adding a messaging layer (the queue and its consumer function in our case) in front of a synchronous service to turn it into an asynchronous one (Hohpe and Woolf 2004).

3.4.5 Circuit Breaker

Problem: A non-responsive third-party service causes accumulating FaaS charges as a function needlessly performs and waits for requests that time out.

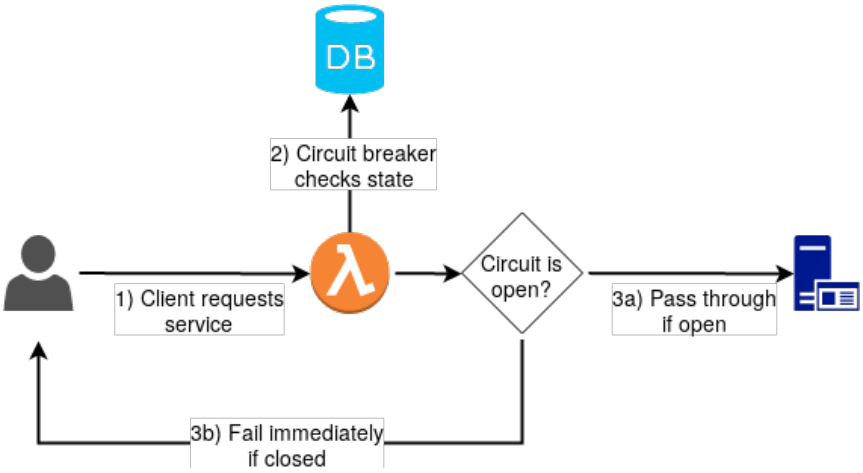


Figure 27: Circuit Breaker

Solution: Restrict service access in times of degraded operation to reduce service load and prevent performing operations that are likely to fail.

Transient network errors and temporary service unavailability are common and unavoidable occurrences in distributed systems. A simple request retry mechanism is usually enough for the system to recover from such temporary interruptions, but unexpected events – human error, hardware failure etc. – can occasionally lead to much longer downtime (Microsoft 2018a). In these cases request retry becomes less valid of a strategy, leading instead to wasted consumer resources and latency in error handling. Furthermore, a large number of consumers bombarding an unresponsive service with repeated requests can end up exhausting service resources and thus inadvertently prevent recovery. These points are particularly relevant for serverless consumers where the pay-as-you-go pricing model means that waiting for timeouts directly translates into extra costs. A serverless consumer's scaling properties also make it more prone to service exhaustion: as a consumer takes longer to execute while waiting for timeout, the platform ends up spawning more concurrent consumer instances which in turn tie up more service resources in a spiraling effect. As observed by Bardsley, Ryan, and Howard (2018), "it is in situations like this that retry is not beneficial and may well have

harmful effects if it ends up spinning up many cold Lambdas".

Instead of reexecution we're then looking to prevent calling an unresponsive service in the first place. The Circuit Breaker pattern, as popularized by Nygard (2007), does just that "by wrapping dangerous operations with a component that can circumvent calls when the system is not healthy". Akin to an electrical circuit, the pattern keeps track of requests passing through it and in case an error threshold is reached it momentarily blocks all requests. In closer detail, the Circuit Breaker operates either in *closed*, *open* or *half-open* mode. In times of normal operation the circuit is closed, i.e. requests get proxied to the service as usual. When the service becomes unresponsive and the number of error requests exceeds a threshold the circuit breaker trips and opens the circuit, after which service requests fail immediately without attempts to actually perform the operation. After a while when the service has had a change to recover, the circuit goes into half-open mode, passing through the next few requests. If these requests fail, the circuit trips open and again waits for a while before the next try; if they succeed, the circuit is closed and regains normal operation. Via this mechanism the Circuit Breaker benefits both the consumer and the service, as the consumer avoids waiting on timeouts and the service avoids being swamped by requests in times of degraded operation.

As a stateful pattern the Circuit Breaker needs to keep track of circuit mode, number of errors and elapsed timeout period. A serverless implementation can utilize either the Externalized State (section 3.1.4) or State Machine (section 3.1.5) pattern for managing this information. Additionally, the pattern can be implemented either alongside an existing consumer or as its own function between a consumer and a service similarly to the Proxy pattern (section 3.3.2). As to further implementation details, Nygard (2007) notes it is important to choose the right error criteria for tripping the circuit and that "changes in a circuit breaker's state should always be logged, and the current state should be exposed for querying and monitoring". Instead of returning a plain error message the open circuit can also implement a fallback strategy of returning some cached value or directing the request to another service. An open circuit could even record requests and replay them when the service regains operation (Microsoft 2018a).

The Circuit Breaker is similar to the SOA pattern of Service Watchdog (Rotem-Gal-Oz 2012)

in the sense that both implement self-healing by means of restricted service access. What distincts the two is who's responsible: the Service Watchdog depends on an integrated component inside the service to monitor its state whereas the Circuit Breaker only acts external to the service, on the basis of failed requests. This distinction makes the Circuit Breaker easier to deploy against black-box components.

4 Migration process

This chapter describes the process of migrating a web application to serverless architecture. The goal of the process is to explore the catalogued patterns' feasibility by applying them on common problems in the domain of web application development. As well as exploring the patterns we're seeing how the distinct serverless features drive application design and trying to gain deeper understanding of the advantages and shortcomings of the paradigm. The chapter begins with the description of the migrated application along with its functional and non-functional requirements. We then identify the ways in which the current implementation fails to meet these requirements and thus set a target for the serverless implementation. Lastly a new serverless design is proposed using the pattern catalogue of chapter 3, and in cases where the patterns prove insufficient or unsuitable to the problem in hand, modifications or new patterns are proposed.

4.1 Image Manager

The migrated application, Image Manager, is a tool for managing image assets. Image Manager is adapted from a real-world application, although modified in places for the sake of illustration. Similarly to a SaaS offering such as Cloudinary, the application takes user-uploaded images, performs various forms of processing and then hosts and serves the processed images to be consumed by other applications. In case of Image Manager the processing needs are threefold: rendering a thumbnail, rendering a low quality image placeholder (LQIP), and automatic label detection. Thus in short Image Manager can be split into three basic functional requirements: image upload, image processing and image hosting.

The pre-migration serverful Image Manager consists of a single server application that connects to a number of BaaS-type cloud services. These components are depicted in figure 28. The server application publishes an HTTP API endpoint for image uploads which is consumed by a browser client. In place of access control this public-facing API uses a CAPTCHA: before image upload the client requests a challenge from Google reCAPTCHA API, solves it and sends the obtained token along with the image upload request. The server

application then also connects to reCAPTCHA API to verify token validity before proceeding with the upload request. A CAPTCHA is used instead of full-blown authentication to allow for anonymous users while still providing some degree of protection against bots and other illicit usage. After CAPTCHA verification the application proceeds with image processing. The thumbnail and LQIP rendering tasks are performed locally whereas labeling is handled by a network call to an external image analysis service, Google Cloud Vision API. The three processing tasks are independent and performed concurrently. Finally both the original and processed images are uploaded to Google Cloud Storage where they can be fetched via publicly accessible URLs. Figure 29 illustrates the image upload sequence in detail, showing how the components fit together.

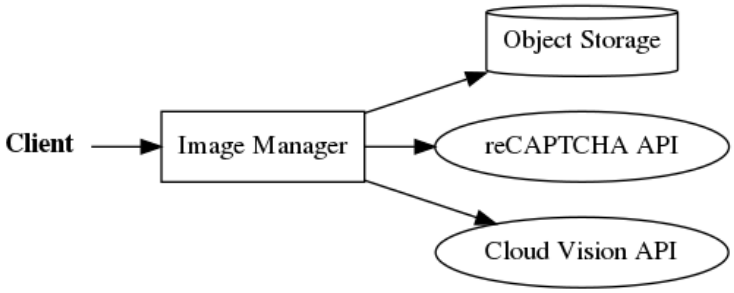


Figure 28: Image Manager components

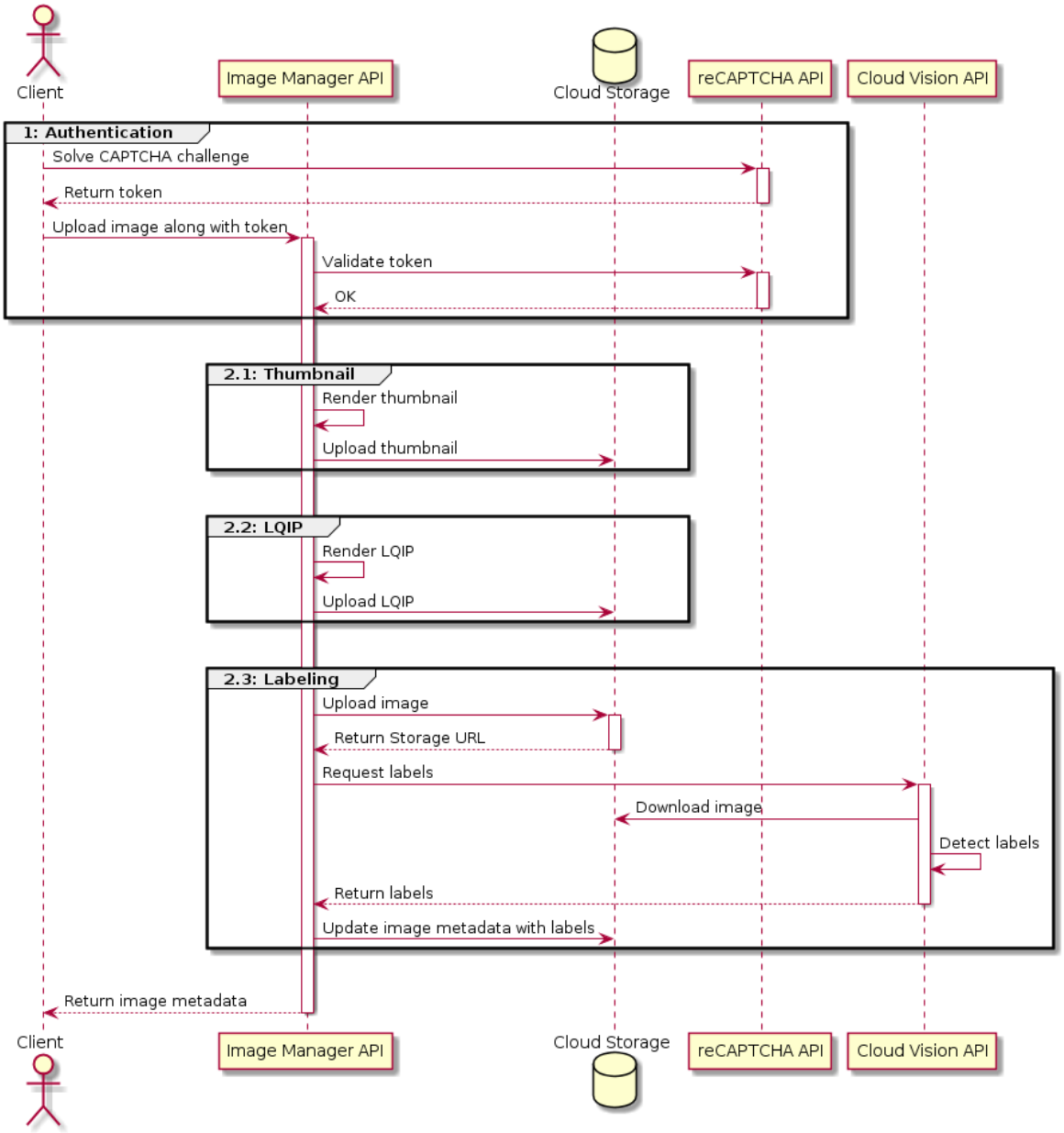


Figure 29: Image Manager upload sequence

Overall the image upload task is both CPU-intensive due to rendering and I/O-heavy due to cloud service requests and also since processing results are temporarily written on disk before cloud storage upload. As for technical details, Image Manager is written in TypeScript, transpiled into JavaScript and running on NodeJS v10. The server application is containerized into a Docker image and deployed on a single VM on Google Cloud Platform's us-east1 region. The VM exposes an IP address through which the application is accessed from public

internet.

Image Manager can even in its pre-migration state be considered “cloud-ready” in the sense that it was originally designed and developed to run specifically in a cloud environment (Pozdniakova and Mazeika 2017). This is reflected in the usage of container technology and the reliance on cloud platform services in favour of self-managed code. The degree of cloud-readiness should be kept in mind when considering the migration process as the practices and observations might not apply when starting off with a more conventional on-premise application architecture.

The motivation to migrate Image Manager to serverless architecture stems from a number of shortcomings in the current implementation, specifically relating to the non-functional requirements of availability, scalability, cost-efficiency and isolation. First, the obvious drawback of the server application’s single-VM deployment is poor availability as there is no failover instance to take over in case of VM failure. Likewise the application’s capacity to serve traffic is limited by a single VM’s computing resources as there is no scaling mechanism in play. Achieving this double goal of availability and scalability, i.e. ensuring a correct number of VMs to meet current demand at all times would require a considerable amount of infrastructure configuration involving load balancing, clusterization and scaling policies (Jonas et al. 2019). This inelasticity also results in cost-inefficiency as the VM instance is constantly running and accumulating charges whether or not there’s any traffic. Lack of isolation is also a major concern, primarily since all application logic including image processing tasks are bundled together into a single monolithic application. This causes resource contention, as for example high CPU usage in one of the rendering tasks can divert resources from the API and result in connection timeouts. This combined with processing tasks’ highly asymmetrical performance profiles also further complicates scaling as we can only scale the whole application, not just the bottlenecks. Lack of isolation also presents itself in how all traffic is routed through the server application, which in case of image uploads means an extra network trip before reaching Cloud Storage. Finally, the server application’s monolithic design has negative maintainability implications since modifications cannot be developed or deployed independently.

4.2 Serverless Image Manager

Rewriting an application in serverless architecture is clearly not a trivial task nor does it have a single correct solution. Comparable to building a system out of microservices or plotting class hierarchy in object-oriented software design, the same outcome can be achieved with a variety of different but equally valid compositions of FaaS and BaaS components. As a baseline the serverless Image Manager should fulfill the same functional requirements as its predecessor. Building on that the migration should improve the application's quality attributes, particularly concerning the deficiencies listed above.

Looking at Image Manager's components in figure 28, it's notable that the system's non-functional deficiencies stem from the server application and not from the integrated cloud services. The services are fully provider-managed, scale to demand and follow a pay-per-use pricing model: they don't constitute an operational overhead nor limit the application's elasticity scaling- or pricing-wise. A serverless Image Manager can therefore retain these integrations while reimplementing the server application in FaaS. While the services themselves remain the same, what changes is the way they're interfaced with since a FaaS consumer can necessitate different communication patterns than a conventional one: publish-subscribe instead of request-response, for example. As for the server application, it has two main responsibilities: first, it acts as a glue component that binds together BaaS components. Second, it provides the kind of custom server-side functionality that we cannot or choose not to offload to external services, namely thumbnail and LQIP rendering. Seeing how these responsibilities match identically with the role of FaaS in serverless systems as discussed in section 2.3, we can expect FaaS to provide a fitting serverless alternative and migration target for the server application. The specific FaaS platform used here is Cloud Functions (Google 2018) due to all the integrated services and the previous VM deployment residing on Google Cloud as well.

The simplest FaaS implementation of Image Manager involves wrapping the whole server application into a single function that is then invoked synchronously via HTTP. This arrangement is from the client's point of view identical to the container deployment since the function's exposed HTTP trigger acts just like the server application's HTTP API. It also already manages to shift a majority of operational concerns on to the cloud provider. How-

ever in many cases this approach is limited by the platform's restrictions on function size and computing resources. A single monolithic function also does little to improve isolation and maintainability. If allowed by platform limits the approach can nonetheless be a good starting point for migration: akin to the Strangler pattern (3.3.3) first migrate the application into a single function and then incrementally split it off into smaller units.

4.2.1 Pattern selection

Taking full advantage of the FaaS platform's capabilities requires a more thorough redesign than simply packaging an application as-is into a single function. The design process used here is based on the pattern catalogue of chapter 3. First each pattern is evaluated against the migrated application's functional and non-functional requirements. The patterns that most closely work towards meeting these requirements are then selected, and the selected patterns are finally composed together to form the proposed serverless design. Applying the process to Image Manager resulted in the following set of patterns: Event Processor, Fan-out/Fan-in, Thick Client, Valet Key and Bulkhead.

First of all the requirement for cost-efficiency leads towards the Event Processor pattern (3.2.1). The image processing tasks should be event-driven: executed on-demand whenever there are images to process and not consume any resources or amass charges otherwise. Technically this is implemented by splitting image processing into a new function that subscribes to Cloud Storage upload events so that each image upload triggers a function invocation. The pattern also improves scalability since a large number of uploads results in a similarly large number of function invocations.

The requirement for scalability leads towards the Fan-out/Fan-in pattern (3.1.3). In our case the pattern consists of splitting the three image processing tasks into their own functions: a simple decision to make with the tasks already being essentially independent. As each upload now triggers not one but three parallel functions, CPU-intensive rendering tasks can scale independently from the less intensive labeling task. We can also expect a performance benefit as the overall task is completed faster.

The Thick Client pattern (3.1.6) is selected with cost-efficiency in mind. Since the Event

Processor breaks coupling between the API and image processing, it is now possible to omit the API layer and have the client orchestrate uploads instead. Retaining the API would mean paying for function execution for the duration of image upload; by uploading images from the client directly to Cloud Storage this expense as well as an extra network trip can be avoided.

The Valet Key pattern (3.3.4) is chosen to authorize requests between the client and Cloud Storage and thus facilitate the client-driven workflow. This involves extending the CAPTCHA validation behaviour so that a temporary access token to Cloud Storage is returned in case of a valid CAPTCHA. Applying the Bulkhead pattern (3.4.3), the Valet Key implementation is split off into a separate authorizer function to ensure its availability when the rest of the system is under high load. The Function Warmer pattern (3.4.1) could optionally be used to further improve the authorizer's availability by keeping a warm function instance ready to serve requests at all times. In case of Image Manager there are no strict response time requirements so this pattern is omitted.

The design process results in the proposed design of the serverless Image Manager. This outcome is presented in figure 30 where rectangular blocks with a λ prefix signify a FaaS function. In the serverless Image Manager the server application is replaced by four different functions: one for each image processing task plus an authorizer function. Out of these functions only the authorizer exposes an HTTP API whereas the others are event-driven. The new image upload sequence, as illustrated in figure 31, is also more event-driven as opposed to being orchestrated by the server application.

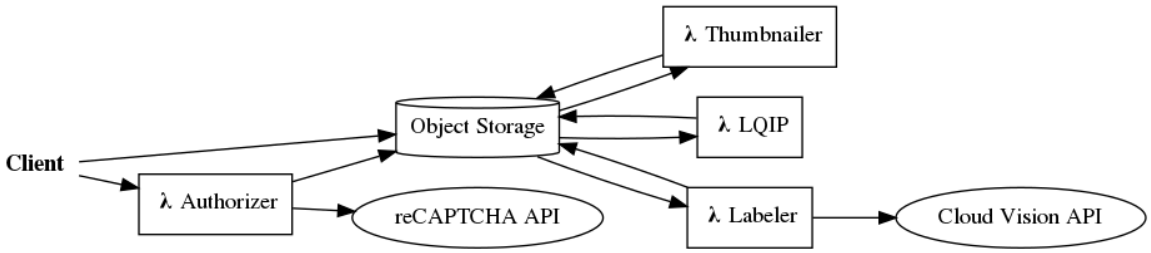


Figure 30: Serverless Image Manager components

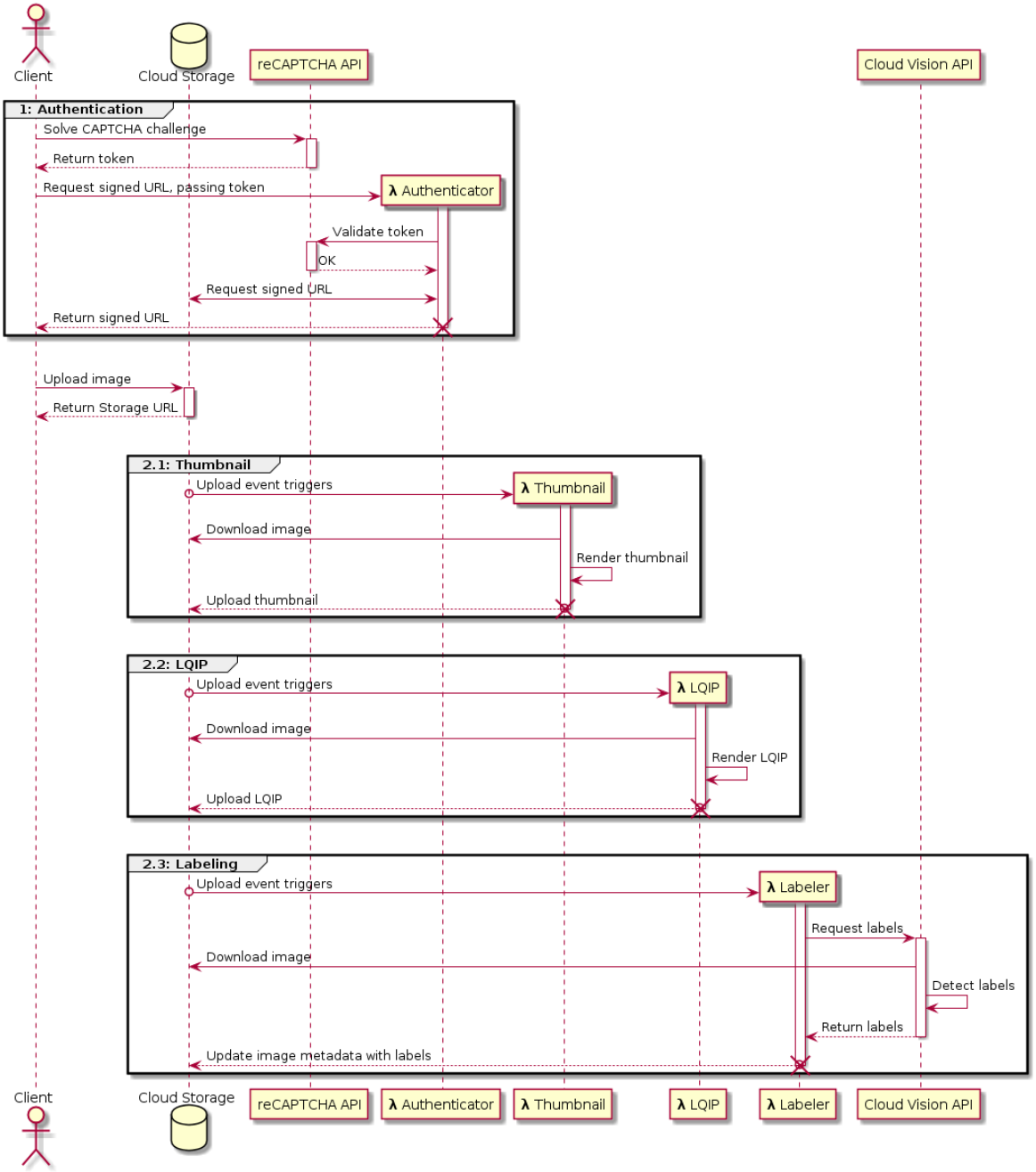


Figure 31: Serverless Image Manager upload sequence (steps 2.1-2.3 run in parallel)

4.3 New patterns

This section lists five new patterns extracted from migration process. The patterns present solutions to problems in Image Manager's serverless design e.g. in places where the serverless implementation performs worse than the original one or fails to provide the same behaviour.

In addition we're introducing patterns that could be used to extend the serverless design to further improve its quality or add functionality beyond the original requirements.

4.3.1 Async Response

Problem: Client doesn't get any feedback from the asynchronous tasks it triggers.

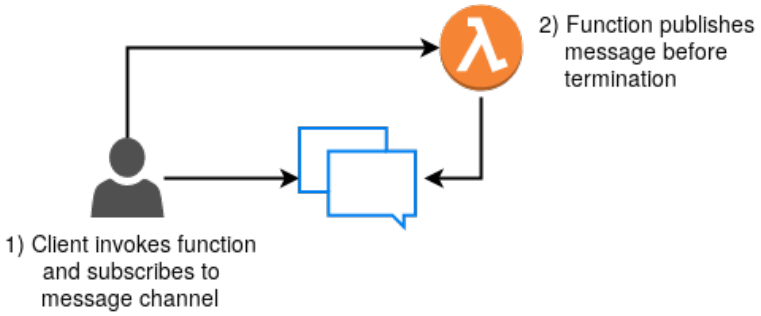


Figure 32: Async Response

Solution: Use a pub/sub channel to notify the client before function termination.

In the pre-migration Image Manager all processing happens in the span of the single upload request. The request blocks and the client won't receive a response before all processing and storage uploads are finished. On the other hand the serverless Image Manager client only receives an acknowledgment of receipt of the original image since image processing is continued asynchronously after the initial Cloud Storage upload. This means the serverless client has no way of notifying the user of a finished processing task. Solving this problem requires a way for the image processing functions to notify the client after they finished with execution.

In more general terms the problem is one of an asynchronously invoked function instance re-establishing communication with the original task initiator, i.e. turning fire-and-forget into something more resembling of request-and-response semantics. As additional complexity, the initiator might not be the actual function invoker but further down the call stack, as in the case of Image Manager where client first activates Cloud Storage which then invokes processing functions.

The Async Response pattern tackles the problem using publish-subscribe messaging. The

client, after initiating the task, subscribes to a message channel and waits for notification of the task completion. Conversely the last function responsible for task execution publishes a message to the same channel before terminating. After receiving acknowledgment of task completion the client can dismantle the message channel and proceed to update the the user interface, for example. Instead of a single completion message we could also send status messages during execution for the client to keep track of task progress.

The Async Response can be implemented with any technology that enables publish-subscribe messaging between the client and function: Firebase Realtime Database on Google Cloud or API Gateway Websockets on AWS, for example. One implementation challenge concerns identifying the completion message: how should the message be sent so that it ends up at the right client? One approach is to wrap task initialization in another function that first creates a message channel with an unique identifier and then passes the identifier to both the client and the processing function. This however incurs the overhead of having to pass the channel identifier along each processing step. Another approach is to compute the channel identifier from task payload: in case of Image Manager the client could for example listen to messages on a channel named after the uploaded file's name.

4.3.2 Task Controller

Problem: Client has no way of controlling or cancelling an asynchronous task after triggering it.

Solution: Make each function instance open a messaging channel to the client in the beginning of its execution in order to listen to client commands.

Possible future use cases for Image Manager would be to track processing task progress and cancel processing tasks midway. We could for example want to terminate a long-running and expensive batch job if its made redundant before completion, or tweak task parameters after initiating it.

The Task Controller provides a general way for clients to issue commands to function instances mid-execution. The pattern extends the Async Response pattern by turning one-way function-to-client messaging into a two-way channel: instead of just publishing messages at

the end of their lifespan, functions also start listening for messages right after initialization.

4.3.3 Local Threading

Problem: Scaling an I/O bound operation out to parallel function instances is inefficient since the instances compete of the same I/O resources.

Solution: Use local OS threads inside a single function instance to efficiently scale out operations like network requests.

Serverless Image Manager's labeling function is heavily I/O bound since the majority of its execution duration is spent waiting for network requests: first for the labeling request to Cloud Vision API and then for the metadata update request to Cloud Storage (see the sequence diagram in figure 31). Scaling this function is therefore inefficient since a larger amount of reserved computing resources doesn't make network requests finish any faster. In addition as shown in section 2.7, on some FaaS platforms parallel function instances could end up being allocated on the same physical machine where they share and contest for the same network resources. In fact scaling out instances just to wait for network requests can lead to runaway costs since a waiting function is billed just the same as a processing one; this was discussed in section 2.9 on the economics of serverless.

The Local Threading pattern circumvents the problem by performing I/O-bound operations concurrently inside a single function instance instead of allocating a new instance for each operation. For example in Image Manager's labeling task, Local Threading can be applied to batch image upload events and then send multiple Cloud Vision API requests concurrently inside a single function instance. The pattern takes advantage of the fact that serverless functions, while limited in computing resources and lifespan, are still essentially full-fledged container instances with access to OS threads and processes. For example an AWS Lambda instance can use up to 1024 threads (AWS 2018a).

The potential cost savings depend largely on the nature of the I/O-bound operation, number of concurrent operations and the way the FaaS platform handles I/O. While an interesting area for future work, optimizing and benchmarking this is outside the scope of the thesis.

4.3.4 Prefetcher

Problem: Each event handler triggered in response to a single event starts execution by fetching the identical event metadata, resulting in redundant network traffic.

Solution: Trigger a single event handling function that fetches event metadata once and then triggers the other event handlers, passing metadata as function payload.

Image Manager's two rendering functions follow largely the same behaviour: triggered by an image upload event emitted from Cloud Storage, they receive the image file URL as input, make a network request to Cloud Storage to fetch the file, render a thumbnail or an LQIP respectively, and finally upload the result to Cloud Storage (see the sequence diagram in figure 31). The first step is the same for both functions: the Cloud Storage upload event payload doesn't include the actual file but just the URL, so all processing functions have to start by downloading the file. In Image Manager's case this is not especially problematic since the FaaS platform and storage service share an internal network. In other cases fetching additional event information could however incur considerable latency or cost overhead in form of network and service charges.

The Prefetcher is an optimization pattern for avoiding expensive duplicate event metadata requests. It's implementation involves adding a new function between the event and its handler functions. Upon event occurrence, the prefetcher function first fetches the metadata and then invokes the actual event handlers, passing the fetched metadata as invocation payload. As a result the event handlers now don't need to separately fetch the extended event metadata. The Event Broadcast pattern (3.2.4) can be utilized to avoid coupling the original event handlers with the prefetcher function.

As before with the Local Threading pattern, Prefetcher's efficiency gains are highly dependent on the nature of the metadata request, the number of event handlers as well as on how the FaaS platform handles networking. The pattern is also limited by the maximum function payload size which for AWS Lambda is 6MB for synchronous and 256KB for asynchronous invocations (AWS 2018a).

4.3.5 Throttled Recursion

Problem: A spike of recursive function invocations can exceed the platform's maximum concurrency limit.

Solution: Pass recursive invocations through a message queue in order to throttle their execution.

Another potential future use case for Image Manager is to transform extremely large images or even video material recursively. Using a divide-and-conquer algorithm, a function can split its payload into two or more subtasks and invoke itself recursively until the subtasks become simple enough to solve. The problem with recursion in FaaS however is that the number of concurrent instances can quickly grow out of hand and exceed the platform limit, placing a constraint on recursion depth. We might additionally want to control the rate of recursive branching in order not to overwhelm any potential external services used in subtask solving. A throttled rate of execution can also be desirable to serve as a safety mechanism against infinite loops.

The Throttled Recursion pattern consists of a supplementing the recursive function with a message queue through which each subtask invocation is passed. Instead of directly invoking itself, the recursive function sends its subtasks into the message queue. On the other hand the message queue also invokes the recursive function with queued messages. The pattern is similar to the Throttler pattern (3.4.4) with the exception that here the single function acts both as a producer and as a consumer on the same queue. By adjusting the queue's consumption rate we can control the recursive execution speed. Also now the spikes are not limited anymore by FaaS concurrency limits but instead by maximum queued messages count which is typically far greater.

5 Evaluation

This chapter evaluates the outcome of the migration process.

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

5.1 Developer perspective

development ease, operations, maintainability

From a development point of view the process of rewriting server application code into FaaS function handlers was found relatively simple. The same runtime environment (NodeJS v10) was used in both versions of the application and the deployment artifacts for both were built from the same codebase. The server application's code also benefited from being modularized so that the server-specific request handling implementation was separated from image processing logic, for example. These factors together enabled a high degree of code reuse when implementing FaaS functions. The source code of the image labeling function is presented in listing 5.1 to exemplify how a comparatively small amount of boilerplate code is required to instantiate a new function: in this example the labeling logic is imported from a *label* module which is used in both the server application and the serverless one.

```
import { Storage } from '@google-cloud/storage'
import { getBucketFileLabels } from '../label'

const gcs = new Storage()

export async function labeler(data) {
  const labels = await getBucketFileLabels(data)

  return gcs
    .bucket(data.bucket)
    .file(data.name)
    .setMetadata({ metadata: { Labels: labels } })
```

```
}
```

Listing 5.1: Image labeler function handler

qualitative vs quantitative

How did the migration help in future extendability and maintenance of Image Manager? - development perspective - less bundling/coupling - operational perspective - ease of deployment - monitoring - financial perspective - performance perspective - cold start averted by 1) optionally using function warmer, 2) only having one synchronous bottleneck, otherwise event-driven - others?

Old implementation's shortcomings of availability, scalability, cost-efficiency and isolation – we're these addressed?

Maintainability. The modular architecture of micro-services allows reducing the complexity of mono-lithic systems. Breaking a system into independent and self-deployable services enables developer teams to make changes and test their service independent of other developers, which simplifies distributed development. Moreover, the small size of each microservice contributes to increasing code understandability, and thus to improving the maintainability of the code. Scalability. Scaling microservices is easier than scaling monoliths. Scaling monolithic systems requires huge investment in terms of hardware and often finetuning of the code. If there is a bottleneck in some component, a more powerful piece of hardware can be used, or multiple instances of the same monolithic application can be executed across several services and managed by a load balancer.In contrast, microservices are not automatically scalable, even if they are commonly deployed in elastic and stateless architectures. However, in a microservicesbased system, each microservice can be deployed on a different server, with different levels of performance, and can be written in the most appropriate development language. If there is a bottleneck in one microservice in such a case, the specific microservice can be containerized and executed across multiple hosts in parallel, without the need to deploy the whole system to a more powerful machine.

5.2 Performance perspective

5.3 Economic perspective

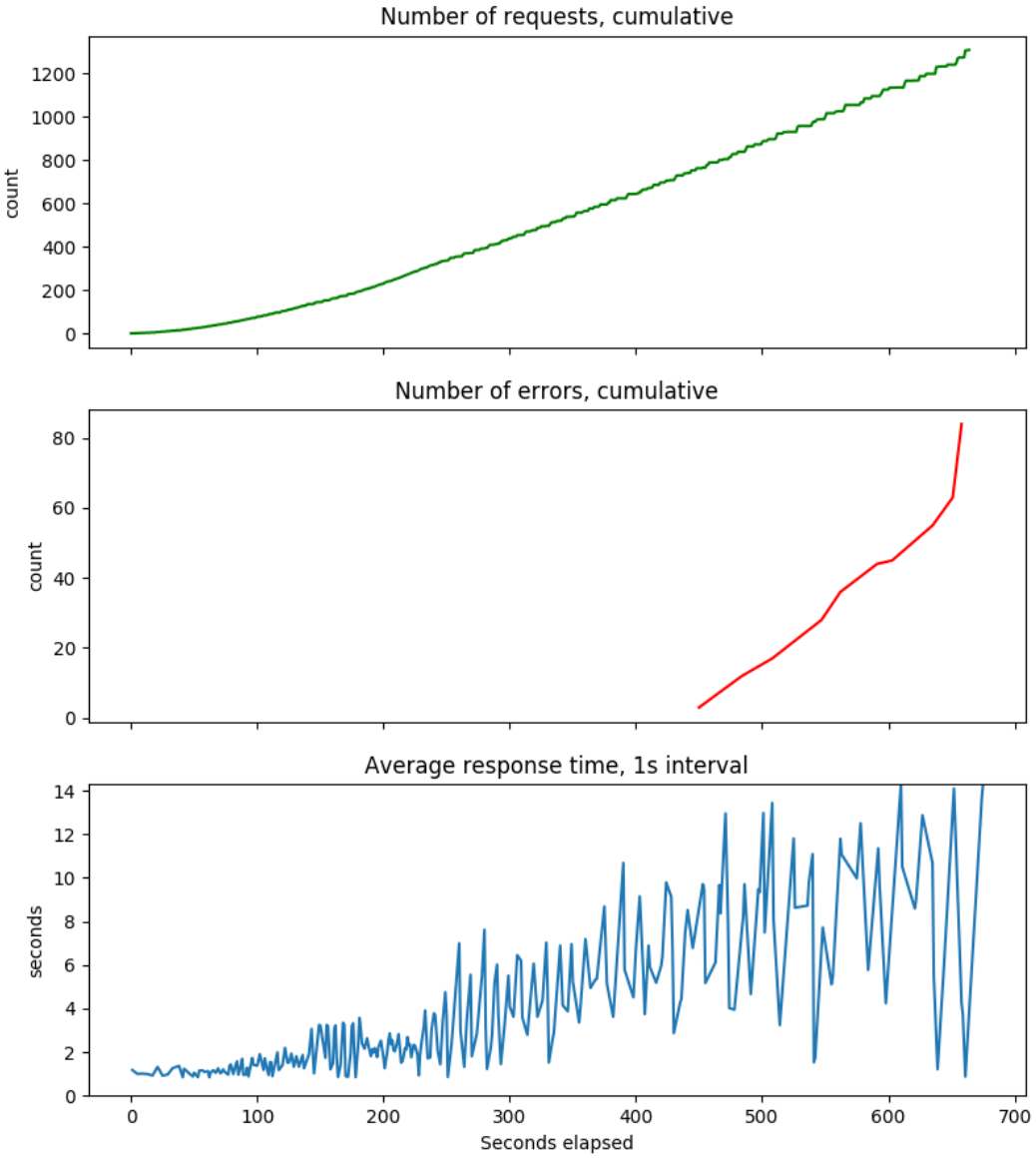
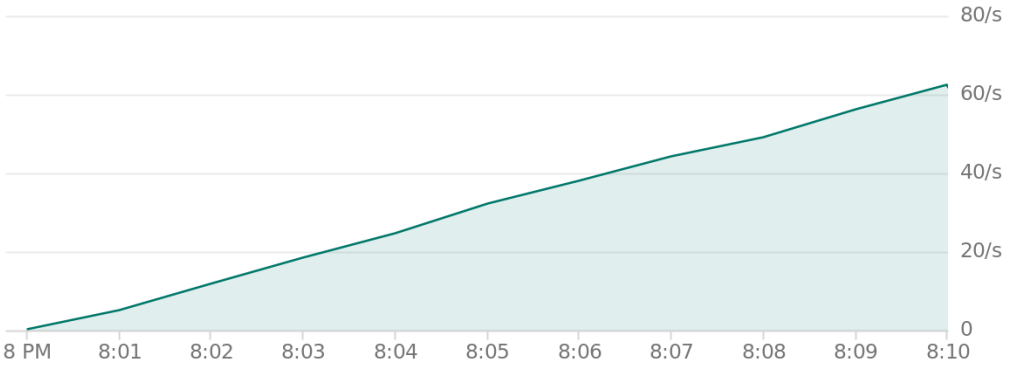
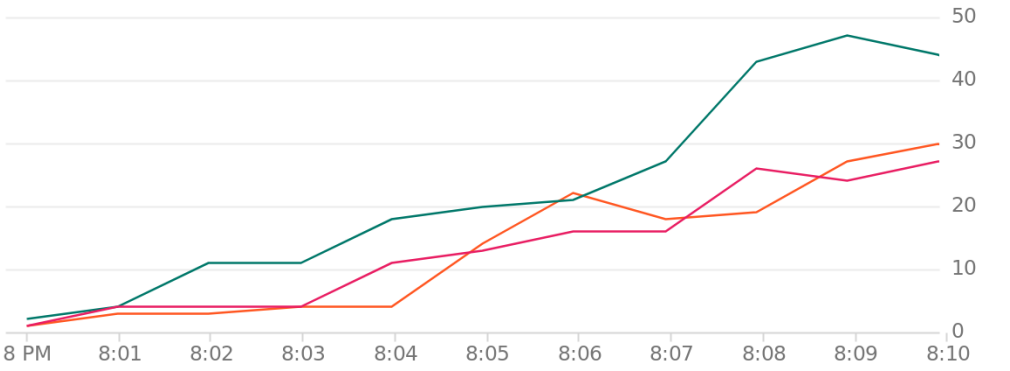


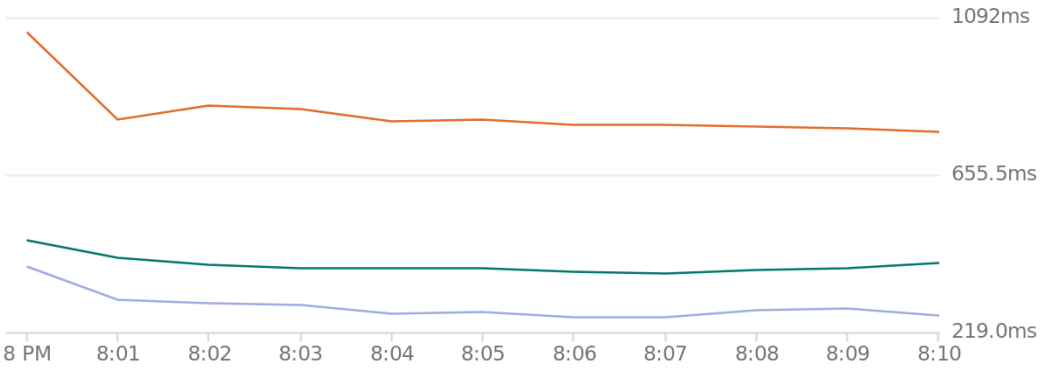
Figure 33: Serverful Image Manager load test results



(a) Invocations per second, sum of all functions



(b) Active instances per function



(c) Mean execution times per function

Figure 34: Serverless Image Manager load test results

6 Conclusion

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

Threats to validity: -

Related work: - Taibi's anti-patterns

Future work: - evaluating and benchmarking new patterns, especially Local Threading and Fetcher

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.

Armbrust, Michael, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, et al. 2009. *Above the Clouds: A Berkeley View of Cloud Computing*. Technical report UCB/EECS-2009-28. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

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

AWS. 2017. *Optimizing Enterprise Economics with Serverless Architectures.* Technical report. Visited on January 16, 2019. https://d1.awsstatic.com/whitepapers/optimizing-enterprise-economics-serverless-architectures.pdf.

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

AWS. 2018b. *Serverless Application Lens: AWS Well-Architected Framework.* Technical report. Visited on January 16, 2019. https://d1.awsstatic.com/whitepapers/architecture/AWS-Serverless-Applications-Lens.pdf.

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. New York, NY, USA: ACM. doi:10.1145/3133850.3133855. http://doi.acm.org/10.1145/3133850.3133855.

Bardsley, D., L. Ryan, and J. Howard. 2018. "Serverless Performance and Optimization Strategies". In *2018 IEEE International Conference on Smart Cloud (SmartCloud),* 19–26. doi:10.1109/SmartCloud.2018.00012.

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/.

Hellerstein, Joseph M, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. "Serverless Computing: One Step Forward, Two Steps Back". *arXiv preprint arXiv:1812.03651.* https://arxiv.org/abs/1812.03651.

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.

Jackson, D., and G. Clynch. 2018. "An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions". In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion),* 154–160. doi:10.1109/UCC-Companion.2018.00050.

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, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, et al. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* Technical report UCB/EECS-2019-3. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.

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.

Kritikos, K., and P. Skrzypek. 2018. "A Review of Serverless Frameworks". In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion),* 161–168. doi:10.1109/UCC-Companion.2018.00051.

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

Lavoie, Samuel, Anthony Garant, and Fabio Petrillo. 2019. "Serverless architecture efficiency: an exploratory study". *arXiv preprint arXiv:1901.03984.*

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, W., M. Vu, B. Zhang, O. David, and G. Leavesley. 2018. “Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads”. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion),* 195–200. doi:10.1109/UCC-Companion.2018.00056.

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. 2018a. “Cloud Design Patterns”. Visited on January 16, 2019. https://docs.microsoft.com/en-us/azure/architecture/patterns/.

———. 2018b. “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.

Nygard, Michael T. 2007. *Release it!: Design and Deploy Production-Ready Software.* Pragmatic Bookshelf.

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/.

Pozdniakova, Olesia, and Dalius Mazeika. 2017. “Systematic Literature Review of the Cloud-ready Software Architecture”. *Baltic Journal of Modern Computing* 5 (1): 124.

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. 2017. “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, 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.

Zambrano, Brian. 2018. *Serverless Design Patterns and Best Practices: Build, secure, and deploy enterprise ready serverless applications with AWS to improve developer productivity.* Packt Publishing.