LUT University
School of Engineering Science
Master’s Programme in Software Engineering and Digital Transformation

Master’s Thesis

Pauli Huhtiniemi

EXPERIENCES OF USING SERVERLESS MICROSERVICES ON VIDEO PUBLISHING PLATFORM

Examiners:	Professor Jari Porras
	D.Sc. (Tech.) Ari Happonen
Supervisor:	D.Sc. (Tech.) Ari Happonen

Abstract

LUT University
School of Engineering Science
Master’s Programme in Software Engineering and Digital Transformation

Pauli Huhtiniemi

Experiences of using serverless microservices on video publishing platform

Master’s Thesis

2019

50 pages, 15 figures

Examiners: Professor Jari Porras
D.Sc. (Tech.) Ari Happonen

Keywords: serverless, microservices, cloud computing

This thesis presents concepts of microservices and serverless computing. Subsequently, a video on demand service is used as a case study and various examples of serverless microservices are presented and experiences of building them are discussed. First goal is to study the literature and then discuss real life experiences of building serverless microservices and see if it frees time from infrastructure management to building business logic while saving money on hosting costs. Second goal is to gather information on suitable use cases for serverless approach as of today. Microservices are small, autonomous and independently deployable services that work together using message-based communication in a highly automated infrastructure. Serverless computing refers to concept of developing applications built with functions and backend as a service components whose usage is invoiced based on actual execution times. These applications are deployed to infrastructure fully managed by a cloud vendor and thus there’s no need to manage servers by a cloud user. While considerable amount of effort is still needed for defining cloud resources with infrastructure as code process, serverless approach was found to be beneficial as it allowed teams to focus on business logic instead of challenges like capacity planning, auto-scaling and high-availability. Cost savings were discovered, but mostly because of more efficient development time was possible. Serverless is a good option for services where really low latency isn’t required and things like high-throughput and time-to-market are important.

Tiivistelmä

LUT-yliopisto
School of Engineering Science
Master's Programme in Software Engineering and Digital Transformation

Pauli Huhtiniemi

Kokemuksia serverless-mikropalveluiden käyttämisestä videojulkaisujärjestelmässä

Diplomityö

2019

50 sivua, 15 kuvaa

Työn tarkastajat: Professori Jari Porras
Tutkijatohtori Ari Happonen

Hakusanat: serverless, mikropalvelut, pilvipalvelut
Keywords: serverless, microservices, cloud computing

Tässä diplomityössä käsitellään mikropalveluiden ja serverless-arkkitehtuurin käsitteitä ja hyödynnetään videopalvelua käytännön esimerkkinä. Työssä esitellään esimerkkejä ja kokemuksia serverless-mikropalveluiden kehittämisestä. Tavoitteena on tutkia kirjallisuutta ja verrata sitä kokemuksiin serverless-mikropalveluiden kehittämisestä, jotta nähdään vapautuuko aikaa infrastruktuurin hallinnoinnista liiketoimintalogiikan kehittämiseen halvempien ylläpitokustannusten lisäksi. Toisena tavoitteena on selvittää serverless-arkkitehtuuriin soveltuvia käyttökohteita nykytiedon valossa. Mikropalvelut ovat pieniä, autonomisia ja itsenäisesti julkaistavia palveluita, jotka kommunikoivat keskenään automatisoidussa infrastruktuurissa. Serverless tarkoittaa palveluita, jotka koostuvat käytön mukaan laskutettavista funktioista ja backend as a service -palveluista. Palvelut julkaistaan palveluntarjoajan infrastruktuurissa, jossa asiakkaan ei tarvitse hallinnoida palvelimia. Vaikka edelleen tarvitaan työtä pilvipalveluiden resurssien ja infrastruktuurin konfiguroinnissa, serverless on hyödyllinen, sillä se mahdollistaa keskittymisen liiketoimintalogiikkaan kapasiteetin tarpeen ennakoimisen, automaattisen skaalautuvuuden ja korkean saatavuuden sijaan. Työssä havaittiin kustannussäästöjä, mutta lähinnä tehokkaamman ajankäytön vuoksi. Serverless on hyvä vaihtoehto palveluille, joissa huomattavan alhainen vasteaika ei ole vaatimus ja korkea läpisyöttö sekä nopea markkinoille pääsy ovat tärkeitä.

Acknowledgements

I’d like to thank my employer Wunder Finland Oy for making it possible to take some time for my studies. Thank you Ari Happonen for your valuable support and guidance when writing this thesis. I’d also like to thank Pollard et al. (2016) for providing template for writing thesis in Markdown as open source, making writing of this thesis a much more pleasant experience.

Last and most importantly, thank you Emma – for everything.

May 2019 in Helsinki, Finland

Pauli

Table of Contents

Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
BaaS	Backend as a Service
CaaS	Containers as a Service
CD	Continuous Delivery
CDN	Content Delivery Network
CI	Continuous Integration
DDD	Domain Driven Design
EDA	Event Driven architecture
ENI	Elastic Network Interface
ERP	Enterprise Resource Planning
ESB	Enterprise Service Bus
FaaS	Function as a Service
I/O	Input/output
IaC	Infrastructure as Code
JSON	JavaScript Object Notation
JWT	JSON Web Token
MVP	Minimum Viable Product
PaaS	Platform as a Service
REST	REpresentational State Transfer
RPS	Requests per second
SOA	Service-oriented architecture
SOAP	Simple Object Access Protocol
SOC	Service-oriented communications
SSO	Single sign-on
SVOD	Subscription Video On Demand
TCP	Transmission Control Protocol
UI	User Interface
VM	Virtual Machine
VOD	Video On Demand
VPC	Virtual Private Cloud
XML	Extensible Markup Language

1. Introduction

The trend of batch-size reduction can be seen in various aspects in software industry. Things need to be smaller and faster. Answers for these needs have been the attempts to reduce the size and scope of the problems, smaller deployable units in form of microservices, smaller development cycles in form of agile methologies and faster deployments with continuous delivery, to mention but a few (Nadareishvili et al. 2016, pp. 65–66).

There has been also progress towards cloud native applications, that are applications specifically built to be run in elastic cloud environments and enable building loosely coupled systems typically made with microservices and containers, and that are resilient, observable and manageable. With high level of automation and continuous delivery workflows, making changes is less risky and can be done more frequently (CNCF 2019; Toffetti et al. 2017; Patrizio 2018). During this progress towards cloud native applications, evolution of shrinking business logic from monoliths to microservices all the way to bare functions on FaaS (Function as a Service) platforms has been ongoing.

When AWS (Amazon Web Services) launched their FaaS service called Lambda in 2014, it quickly drew a lot of attention to concept of *serverless computing*. Developers would be able to focus on business logic and let cloud provider fully handle server provisioning and other infrastructure management tasks, and in addition save money on hosting costs as well (AWS 2014; Jonas et al. 2019). While PaaS (Platform as a Service) services like Heroku and BaaS (Backend as a Service) services like S3 object storage already abstracted server management away from the cloud user, it was the Lambda functions, that would autoscale and that were to be billed based on execution time in a fine-grained increments, that popularized the serverless paradigm (Jonas et al. 2019). Serverless can be said to be one of the three main cloud native development and deployment models, as seen in figure 1.

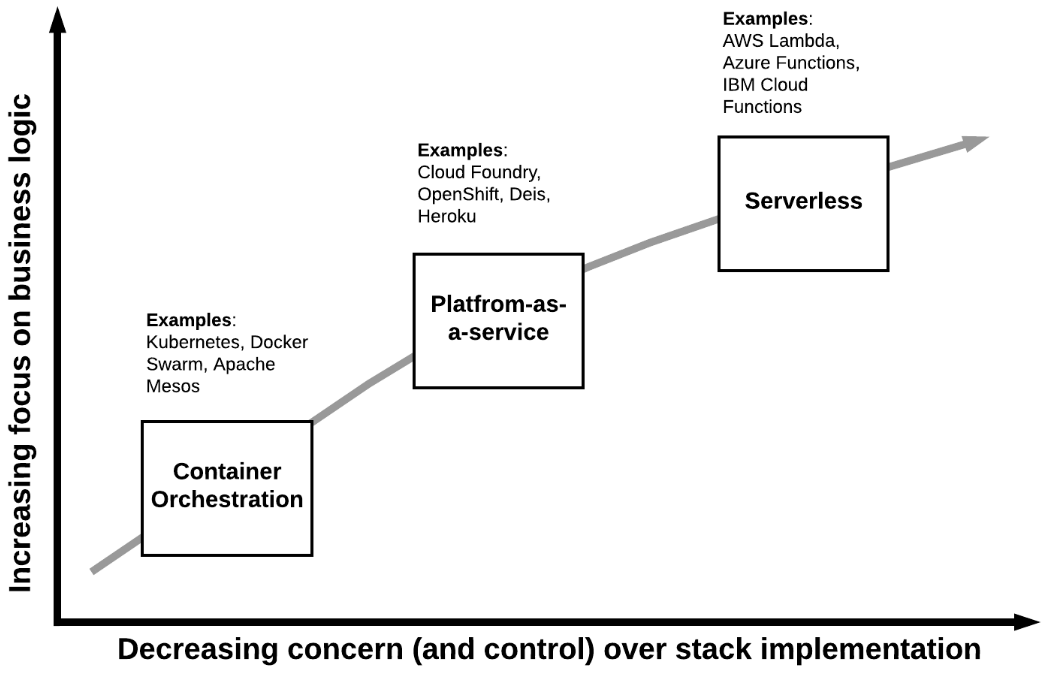

Examples: AWS Lambda, Azure Functions, IBM Cloud Functions
Examples: Cloud Foundry, OpenShift, Deis, Heroku
Serverless
Platfrom-as-a-service
Examples: Kubernetes, Docker Swarm, Apache Mesos
Container Orchestration
Increasing focus on business logic
Decreasing concern (and control) over stack implementation

Figure 1: Cloud native development and deployment models, adapted from CNCF (2018).

In figure 1, three main cloud native development and deployment models with examples of such services are presented. These are container orchestration, or CaaS (Containers as a Service), PaaS and finally serverless. With CaaS, cloud user has maximum control over infrastructure which in turn means that effort is needed to manage this responsibility (CNCF 2018). PaaS ease operational work by adding abstraction layer on top of cloud services while some architectural flexibility is lost (Adzic & Chatley 2017; CNCF 2018). Serverless approach has the lowest requirement for infrastructure management and it's also the most comprehensive option in regards of automatic scaling, load balancing and the most granular billing model (CNCF 2018).

During my career, I've witnessed in many projects how challenging infrastructure management and things like scaling it for high performance needs might be. I have also worked with monolithic applications and experienced first hand the complexity on multiple developers working on a same large codebase in parallel – coordination of even smallest changes would require a lot of time that could have been used building functionality directly beneficial to business. This is why I've been really interested in the idea of building smaller services and the promise of serverless approach that would allow developers to focus on business logic instead of maintaining servers.

1.1 Goals and delimitations

In this thesis, ideas of microservices and serverless computing are examined. A Finnish VOD (Video On Demand) service is used as a case-study and its transition from monolithic applications towards microservices and later usage of serverless computing within AWS cloud services are examined - from both technical and operational point of view. This thesis aims to study the literature and then compare real life experiences of building serverless microservices and see if it already lives up to its promise of freeing time from infrastructure management to building actual business logic while saving money on hosting costs. The main goal of this thesis is to examine the experiences of using microservices and serverless in a video on demand service. Second goal is to gather information on what are the suitable use cases for serverless approach as of today. Definition of microservices and their main characteristics are presented. Term serverless is also defined and current status of serverless computing with benefits and limitations are examined.

1.2 Structure of the thesis

In the following two chapters theoretical background for this thesis is presented. First, definition of microservices is examined in chapter 2 and its impacts on software development and operations are discussed. Second, definition of term ‘serverless’ is presented in chapter 3 and its current benefits and limitations are examined. In chapter 4, a Finnish video on demand service is used as a case-study, various serverless implementations are presented and experiences of microservices and serverless approach are discussed. Finally, in chapter 5 are conclusions.

2. Microservices

The term "microservice" is said to originate from 2011 when group of software architects discussed about common architectural style they had practiced lately. Dragoni et al. (2017), citing Lewis & Fowler (2014) and Wang & Tonse (2013), mentions that same kind of principles were used in Netflix under the term of "fine grained SOA" (Service-oriented architecture). Others might have practiced these principles simply under label of SOA (Lewis & Fowler 2014). Nadareishvili et al. (2016) writes that these principal themes were that microservices are meant for big systems, they are goal-oriented rather than focused on solution and they are replaceable.

Newman (2015, pp. 2–3) defines microservices as "small, autonomous services that work together". Nadareishvili et al. (2016) also includes architectural element, ending up with the following definition:

> "A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices."
>
> – Nadareishvili et al. (2016)

Yermakov (2018a; 2018b) argues that concept of microservices should not be considered as self-sufficient architecture but an implementation approach for distributed systems that are designed based on wider concepts such as SOA, Event Driven architecture (EDA) and Domain Driven Design (DDD). Cockcroft (2017) talks about things that made microservices possible. Greatly faster networks have enabled low latency messaging and thus it's now possible to send orders of magnitude more messages in the same time than before and this has been one of the main developments that have made it possible to move from monoliths to microservices that require a lot of messaging. Another important enabler has been the rise of NoSQL databases as they are scalable and cost effective.

2.1 Microservices and SOA

Lewis & Fowler (2014) argue that SOA has always been a bit vague concept and can mean many different things. One of the biggest differences between SOA and microservices is that in SOA implementations it's common to have a lot of logic in the communication mechanism itself, one example being Enterprise Service Bus (ESB). Newman (2015, pp. 8–9) is along the same lines, claiming that there has been a lack of consensus on how to do SOA well and microservice approach has emerged from real-world learnings on how to do SOA right. Thus, one could think microservices as one way to implement SOA in the same sense that Scrum is a specific way to approach agile software development. Also Dragoni et al. (2017) defines microservices as evolution of SOA and argue that microservices are the second iteration of the idea of SOA and service-oriented communications (SOC). One important difference between SOA and microservices is that SOA uses centralised governance where as with microservices governance is decentralized (Xiao et al. 2016).

2.2 Monoliths

Before explaining the main characteristics of microservices, it helps to define the opposite of microservices that is a monolithic application, defined by Dragoni et al. (2017) as "a software application composed of modules that are not independent from the application to which they belong". Using languages like Java and Python it's possible to split server-side applications into components but they will rely on shared resources like memory, files and database. Because these components are part of the same single logical executable, there's a technology lock-in as same programming language and framework need to be used in all of them (Dragoni et al. 2017). This also means that it's challenging to try new languages, databases or frameworks (Newman 2015, pp. 4–5). Communication between components happen via function calls or method invocations (Lewis & Fowler 2014).

A single monolithic application may offer tens or hundreds of different services via various interfaces like HTML pages, Web services or REST APIs (Villamizar et al. 2016). As new features are added, codebases of monolithic applications tend to grow big and difficult to maintain (Newman 2015, pp. 2–3). With large codebase it also becomes more difficult to keep good modular structure and thus make changes that only affect a single component (Lewis & Fowler 2014). Large codebases also make deployments more challenging because the whole monolith needs to be deployed even for the smallest changes. In practice this tends to lead more infrequent deployments (Newman 2015, p. 6; Lewis & Fowler 2014).

As Lewis & Fowler (2014) explain, monolithic application can be scaled horizontally by having multiple instances behind a load balancer. Scaling a monolith means scaling the whole application, not just the component needing more resources, as shown in figure 2.

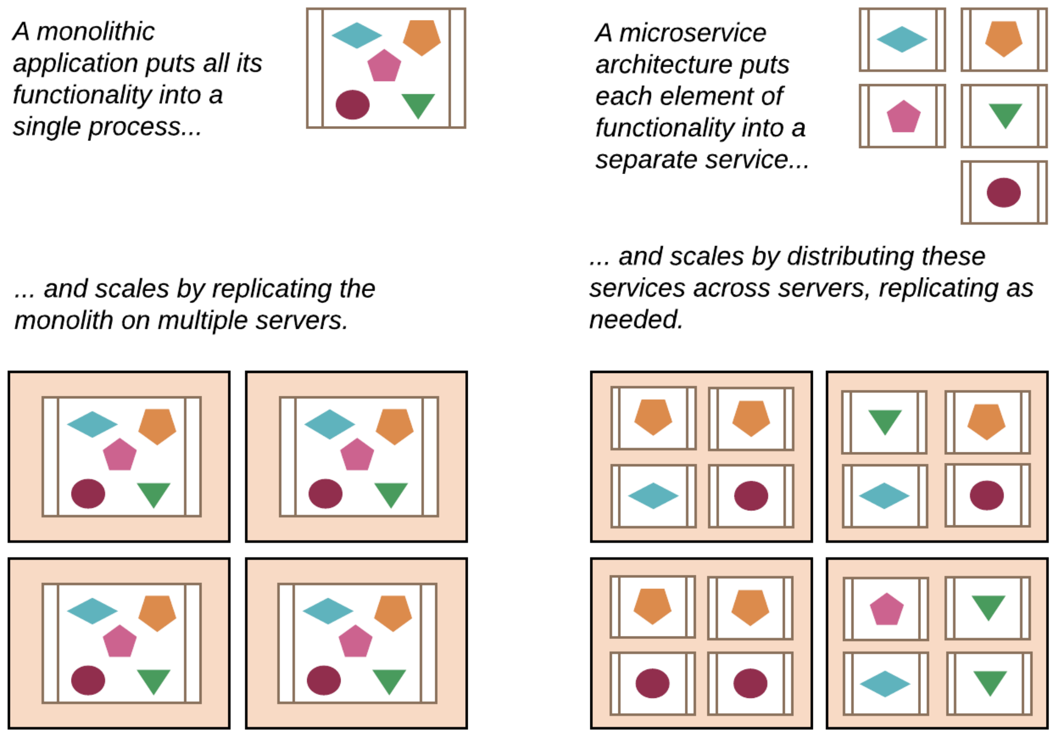

A monolithic application puts all its functionality into a single process...
A microservice architecture puts each element of functionality into a separate service...
... and scales by replicating the monolith on multiple servers.
... and scales by distributing these services across servers, replicating as needed.

Figure 2: Scaling monoliths vs scaling microservices, adapted from Lewis & Fowler (2014).

As seen in figure 2, monolithic application has all its functionality in one process and it scales by replicating the whole monolith on multiple servers. Microservices instead put each element of functionality into a separate service and scales by distributing services across servers (Lewis & Fowler 2014). The way how microservices allow more flexibile scaling and thus might generate direct cost savings, is important reason for choosing microservice model (Newman 2015, pp. 5–6). With monoliths, implementing auto-scaling is more difficult due to how long it takes to start new server instances – this makes it difficult to auto-scale monoliths for spikey traffic as spike might be already over when new instance has booted (Roberts 2016). If one component breaks it will most likely affect the whole monolithic application, taking the whole application down in the worst case (Newman 2015).

2.3 Characteristics of microservices

Based on various sources (Newman 2015; Lewis & Fowler 2014; Nadareishvili et al. 2016; Dragoni et al. 2017), the characteristics presented in figure 3 can be said to be common for microservices.

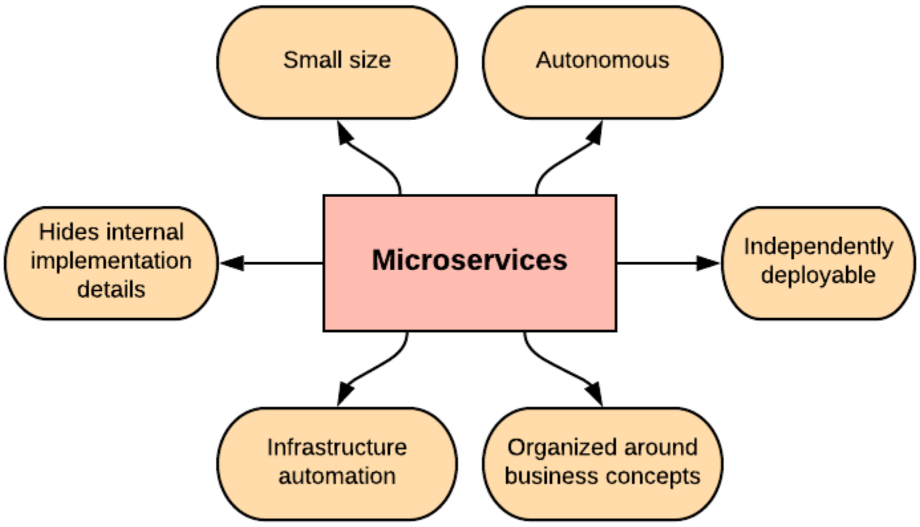

Small size
Autonomous
Hides internal implementation details
Microservices
Independently deployable
Infrastructure automation
Organized around business concepts

Figure 3: Main characteristics of microservices.

In figure 3 the main characteristics of microservices are presented and they are the following:

- autonomous
- hides internal implementation details
- independently deployable
- organized around business concepts
- infrastructure automation
- small size

Next, each of these characteristics are explained broadly in order to give an overview of microservices approach and its benefits for software development.

2.3.1 Autonomous

Microservice is an independent entity that can be deployed as an isolated service (Newman 2015, p. 3) and that is developed by autonomous team (Nadareishvili et al. 2016, p. 5). Decentralization means that there’s no central component that manages and controls the work. Embracing autonomous teams means that development teams need to be trusted more to make their own decisions about implementation. Key benefit here is the increased speed of making changes. Control and decentralization aren’t opposing forces. With microservices approach, services itself get simpler but the overall architecture usually gets more complex – a problem that is tacled with automated processes and tooling (Nadareishvili et al. 2016, p. 8). Nadareishvili et al. (2016) also claims that microservices style is more expensive and organizations must evaluate whether increased system changeability is worth the cost.

2.3.2 Hides internal implementation details

Microservices hide their internal implementation details so they can be implemented in any programming language or framework that best fit the task (Dragoni et al. 2017). This also means that each microservice can be developed and managed by a different team (Lewis & Fowler 2014). Communication between different microservices usually happens using network calls, meaning that they need to expose an API (Newman 2015, p. 3; Dragoni et al. 2017). Different technologies can be used for communication between services, for example REST, XML-RPC and SOAP (Newman 2015, pp. 39–55).

2.3.3 Independently deployable

Independent deployments are considered one of the most important aspects of microservices as it enables the most important benefits microservices has to offer (Nadareishvili et al. 2016, p. 89). Microservices can be deployed independently from the rest of the system and it allows code to get deployed faster. Also if a problem occurs, it can be isolated and thus makes fast rollbacks easier. Getting new functionality for the customers faster is said to be one of the main reasons for companies like Netflix and Amazon to use microservices (Newman 2015, p. 6). Smaller services are faster to deploy and this fact also makes developers deploy more often (Chen 2018).

Independent deployability allows performing selective or on-demand scaling as only the component needing more resources can be scaled, for example by re-deploying it to an environment with more resources. Independent deployability doesn't only offer this kind of operational flexibility but also organizational benefits as it eliminates many cross-team coordination challenges that coupled deployments would cause (Nadareishvili et al. 2016, pp. 89–90).

Microservices offer an interface and sometimes these interfaces may change, enforcing a need to coordinate deployment of multiple services at once. But the aim is to avoid these kinds of dependencies by solid service boundaries and evolution mechanisms in the service contracts that are usually API specifications (Lewis & Fowler 2014). Software architecture can be one of the biggest impediments for enabling Continuous Delivery (CD) approach. Microservices architectural style has enabled increased deployability (Chen 2018).

2.3.4 Organized around business concepts

When talking about microservices, Conway's law is often mentioned, for example by Newman (2015) and Lewis & Fowler (2014). This law is quoted in various forms, Conway himself defining this in 2010 as:

> "Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure."
>
> – Melvin Conway (Conway 2010; original paper: Conway 1968)

So structure of the organization has a strong impact on systems they create. This statement is supported for example by MacCormack et al. (2012), presenting evidence that organization structure tends to get mirrored in the product's architecture and that loosely-coupled, distributed teams end up building more modular and less coupled systems.

If organization has teams built based on technical layers, for example UI (user interface) teams and backend teams, these kind of siloed teams tend to lead into siloed application architecture. In microservices approach the goal is to organize services around business capabilities and have cross-functional teams that include all roles - UI, backend, project management, and so on - needed for development (Lewis & Fowler 2014). Closely related is also the idea of "you build it, you run it", meaning that team is responsible for the service throughout its full lifecycle. Giving developers this kind of operational responsibility and closer contact with the customer has been seen to increase quality of services (Gray 2006).

In his book, Domain-Driven Design, Evans (2004) introduces the term *bounded context* that defines the range of applicability of each model. As an example of a model he uses Charge that might have a different definition in context of billing customers versus paying vendors. Nadareishvili et al. (2016) reminds that the model is just a representation of reality and there might be multiple models, for example a presentation of a bank account is different for a customer and a bank teller. Mogosanu (2012) defines *context* as a responsibility and *bounded context* as a responsibility that is enforced within explicit boundaries.

The model needs to be consistent in its context. Evans argues that even if in an ideal world one would have a single model to be used on the whole enterprise, this kind of total unification of the model is neither feasible nor cost-effective. This is where bounded context comes in place as it defines boundaries and relationships between different models (Evans 2004, pp. 332–333). Domain-Driven Design divides a large system into bounded contexts that use unified model. Different contexts may have different models of common concepts (Fowler 2014). This can be seen in figure 4 where two contexts use the same shared model of a stock item but bounded context of Warehouse also has its own internal, more detailed model of it (Newman 2015).

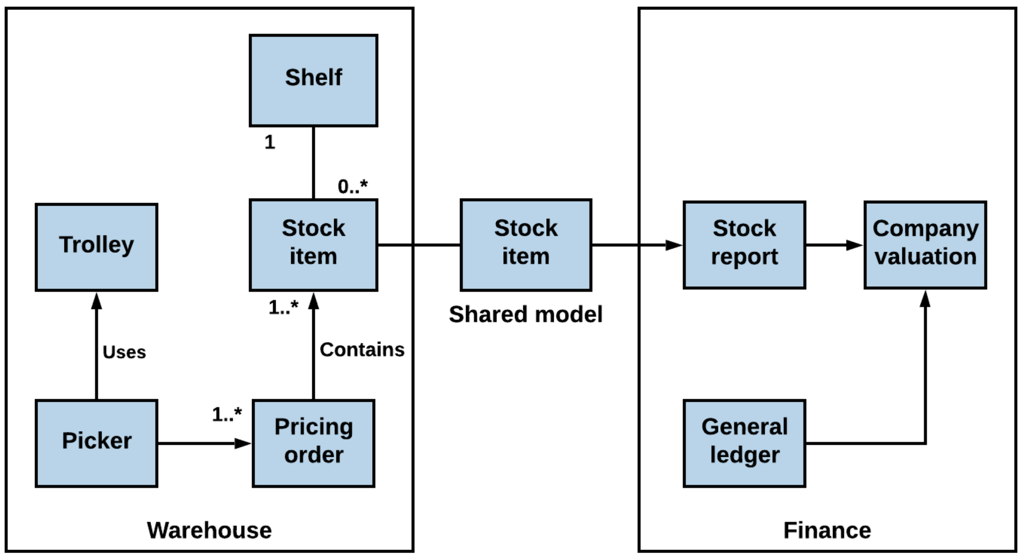


Figure 4: Shared model between bounded contexts (Newman 2015, p. 34).

Defining bounded contexts and relationships between them is useful also for monolithic applications but microservices correlate naturally with bounded contexts and thus help clarify and reinforce the separation of components (Fowler 2014). Newman (2015, p. 34) argues that when definining bounded contexts, one should not think of terms of data that is shared but the capabilities these contexts provide for the rest of the domain. One should first ask "what does this context do?" and only then "what data does it need to do that?". When these concepts are modeled as microservices, these capabilities will be the main operations that are exposed to the rest of the system.

2.3.5 Infrastructure automation

Managing a lot of different microservices will get complex without automation. Managing multiple servers automatically reduces the workload and makes it possible to increase the number of servers without linearly increasing the amount of work needed. Automation also increases productivity of developers (Newman 2015). Dragoni et al. (2017) argues that microservices are meant to be used with continuous delivery and continuous integration and each step in delivery pipeline should be automatic. This will enable fast deployments into production, even in matter of seconds. Newman (2015, pp. 107–108) gives an example of common steps in delivery pipeline that is presented in figure 5.

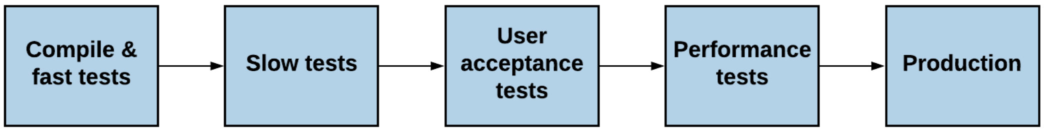


Figure 5: Release process modeled as a build pipeline (Newman 2015, p. 108).

In figure 5, common verification steps in release process modeled as a build pipeline are presented. Code is first compiled and fast tests (usually unit tests) are executed. As code moves through the steps, more time-consuming verifications like integration and performance tests are done. While every step is preferably fully automated, some steps might require manual acceptance, for example user acceptance tests. In final step, new version of the code is deployed to desired environment, in this particular example it is production environment (Newman 2015, pp. 107–108).

2.3.6 Small size

What is considered small? Newman (2015, pp. 2–3) argues a team should be able to rewrite the service in two weeks. He also says that if the codebase is too big to be maintained by one team, it should be considered to break into smaller services. Lewis & Fowler (2014) mentions Two-pizza Team, meaning that no more than dozen people should be in the team responsible for the service.

Nadareishvili et al. (2016) found out that many people use the word “small” as a quality like “reliable” and “coherent” when describing the size of the microservices. They write how most companies focus on quality of each microservice instead of trying to quantify it somehow. Nadareishvili et al. (2016) also identifies the trend of “batch-size reduction”, noticing how methologies like agile, lean and continuous delivery all have principle of size reduction at its core: reducing the size of the problem, reducing the time to finish a task or reducing the size of the deployment unit. Moving from waterfall to agile can be seen as reducing the batch size of development cycles, meaning faster cycles. Term “limited batch size” can be said to be the “micro” part of the microservice (Nadareishvili et al. 2016, pp. 65–66).

2.4 Microservices and managing development work

Microservices approach introduces a new kind of flexibility to arrange work and be more efficient. As Adzic & Chatley (2017) writes, breaking the monolith into more granular units allows the team to work more in parallel and to deploy things independently in comparison to working on the same codebase of the monolith that adds additional work for coordinating deployments and resolving conflicts. Nadareishvili et al. (2016, p. 108) writes that microservices might not be a good approach for project-centric culture where team moves to work on something else when the initial version is built. This is because it’s against the idea of the team being responsible for the service throughout its full lifecycle. Microservices are more suitable for what Narayan (2018) calls product-mode, where instead of projects there are near-permanent teams that own the product over its full lifetime. While there’s no reason that a monolithic application couldn’t be developed in product-mode, both Lewis & Fowler (2014) and Narayan (2018) argue that smaller services make it easier, for example because monolith might be collectively owned. Nadareishvili et al. (2016, pp. 108–109) also mentions outsourced workers and argues that there’s no reason that microservices couldn’t be built by outsourced teams but special care needs to be paid for selecting the correct partners that are amenable for the culture of microservices.

2.5 Summary

In this chapter, overview of microservices was given and main characteristics of these small and autonomous services were examined. There are various ways to deploy microservices as the concept of microservices isn't defined based on how these small services are hosted. They can be hosted in containers, PaaS platform or they can be a part of serverless architecture. In the next chapter, the idea of serverless computing and how it allows to split functionality into even smaller components is introduced. Furthermore, the ways how serverless makes it possible to deploy microservices and how it promises to reduce the amount of work needed for managing infrastructure are presented.

3. Serverless

"Serverless" is a marketing term that refers to the concept of developing and running applications that don't require server management. In serverless approach applications are built as a group of functions that are deployed into platform that handles their execution and scaling. Pricing is based on the actual execution times, meaning that consumer of serverless computing does not pay for idle capacity. It doesn't mean that servers aren't needed for executing services. Also it doesn't mean that operational people like those usually responsible for server management and monitoring aren't needed anymore. But it does mean that in a serverless approach we don't need to think about servers that much anymore as things like server provisioning, updates and scaling are abstracted away from engineers as all these things are handled by the cloud provider (CNCF 2018).

Concept of serverless computing embodies two different areas: BaaS (Backend as a Service) and FaaS (Function as a Service). BaaS refers to third-party API-based cloud services that can be used as a part of the application. Examples of such services are authentication services like Auth0, object storage services like S3 and NoSQL databases like DynamoDB. Usage of BaaS is popular among mobile applications and single-page web apps. Another area of serverless computing is FaaS where server-side logic is implemented as event-triggered, stateless functions that are fully managed by a third-party provider. Serverless BaaS and FaaS components are commonly used together (CNCF 2018; Roberts 2016; Roberts & Chapin 2017). Quickly looking, BaaS and FaaS are quite different as the first one is about outsourcing whole applications and second is a new way to host and run application code, but what they have in common is that neither of them require engineers themself to manage server hosts or server processes (Roberts & Chapin 2017).

Serverless computing has a significant impact on architectural design of applications and as Roberts & Chapin (2017) write, it requires "new design patterns, new tooling, and new approaches to operational management". Especially serverless FaaS, that is ultimately an event-driven model where form of deployments are truly granular and state is handled outside of FaaS components, requires considerably different architecture in comparison to traditional cloud computing with virtual servers (Roberts & Chapin 2017, p. 11).

3.1 From physical servers to containers

Historically applications were hosted on dedicated machines which in turn were hosted in datacenters and these machines would be either purchased or leased for operating systems. Increasing capacity was slow and planning for traffic peaks was challenging and required paying for machines that were under utilised during the time when traffic was normal. Cloud computing and virtual machines allowed to scale resources on-demand much faster and even automatically. Pricing also changed to per-hour resolution. Even if operational costs were reduced, managing virtual servers still require development and operational staff. Various PaaS services like Heroku emerged to ease operational work by adding abstraction layer on top of cloud services, although some flexibility and control is lost while doing this. In serverless model infrastructure provider takes over all the responsibility for receiving and responding to client requests, capacity planning and operational monitoring – meaning that developer only needs to focus on implementation of the function itself. Invoicing is based on CPU time spent on function execution (Adzic & Chatley 2017).

The way virtualization and the pooling of resources is handled has evolved a lot in recent years, as shown in the figure 6.

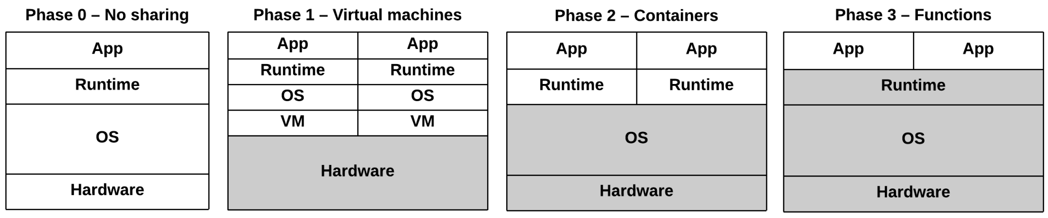


Figure 6: Evolution of sharing (Lynn et al. (2017) who adapted it from Hendrickson et al. (2016)).

Figure 6 shows how sharing has evolved when we have moved from physical servers into containers and now functions. Gray layers in the figure are shared. First, virtualization allowed sharing common hardware, meaning that many virtual machines (VM) could run on the same machine, although each copy of a virtual machine would still run a full copy of an operating system. Later containerization allowed sharing resources through operating system level virtualization where containers would include a small subset of an OS and all the components required to run the specific software. Containers are faster to provision than virtual machines because they have limited access to physical resources and are less resource-intensive. Finally, in FaaS model even the runtime environment is shared (Lynn et al. 2017; Hendrickson et al. 2016).

Figure 7 shows roughly how hosting costs decrease while the amount of control is reduced.

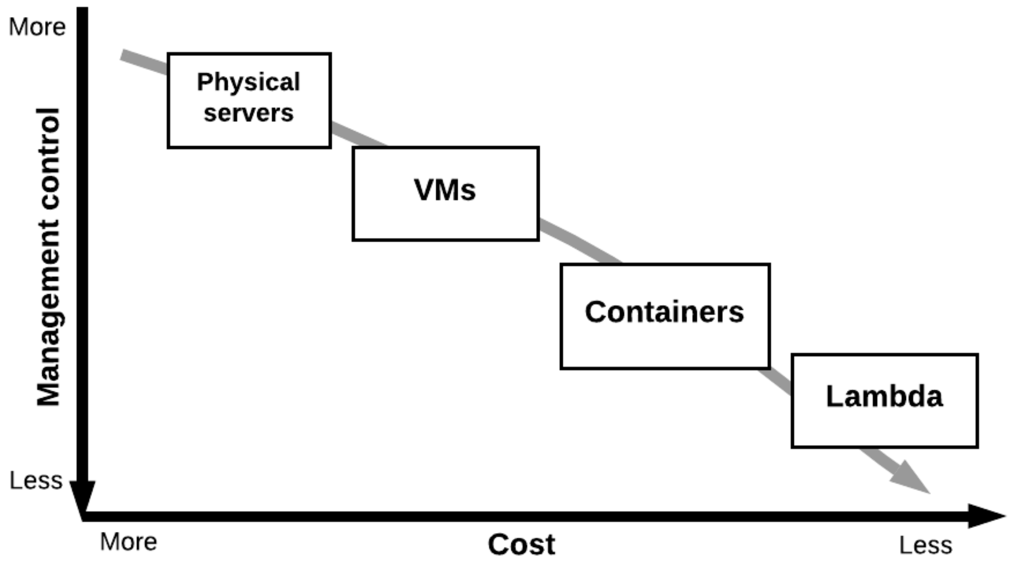


Figure 7: Trend of costs in relation to control with different hosting methods, adapted from Langit (2017).

In figure 7 it can be seen broadly how with physical servers you have the most control but also the costs are the highest. When moving to VMs, containers and finally Lambda (or any other FaaS), the hosting costs are reduced while user also has less control. With FaaS the containers are abstracted away, so in that sense Lambdas are containerless, as Langit (2017) says.

3.2 Functions as a service

In late 2014, Amazon launched AWS Lambda as their FaaS service (AWS 2014). In 2016, other providers followed: IBM with their IBM Cloud Functions, Google with Google Cloud Functions and Microsoft with Azure Functions. Various open-source frameworks that can be run on public cloud or on your own hardware exist as well. Examples of such frameworks are Galactic Fog, IronFunctions and Fission (Roberts & Chapin 2017; CNCF 2018).

A serverless function is a piece of code that gets executed on-demand by an event such as API call, and doesn't consume runtime resources like CPU and memory after that. These stateless functions are run in containers that may last only for one invocation and are scaled automatically without a need for managing servers by the consumer as they are fully managed by cloud provider. Billing is based on actual execution times, meaning that one never pays for idle (CNCF 2018; Roberts 2016; Roberts & Chapin 2017; Eivy 2017).

HTTP API endpoint request is a common event for triggering a function, but the power of serverless lies in the fact that there are various other types of events to act on. For example, a function could react to new file being uploaded to S3 or new object being added to DynamoDB (Eivy 2017). Event sources can be either synchronous or asynchronous. As an example, request via API gateway is a synchronous event whereas object store event or scheduled job similar to cron task are examples of asynchronous events (Roberts & Chapin 2017).

Traditional web application hosting can be shown in figure 8.

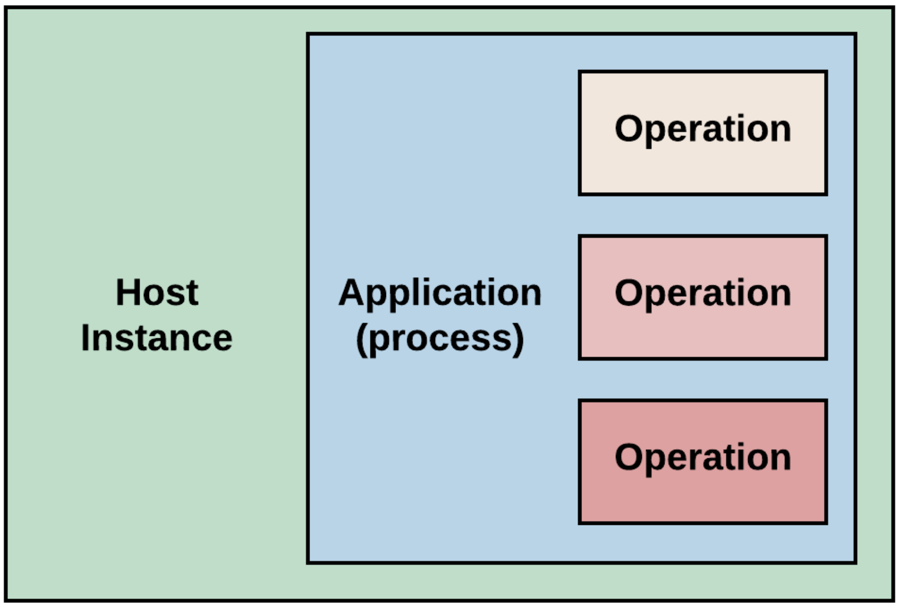


Figure 8: Traditional web application hosting (Roberts & Chapin 2017, p. 8).

Figure 8 shows how in traditional hosting model there's a host instance, usually a virtual machine or a container. Application is deployed within the host and in the case of virtual machine or container, application will be an operating system process. Application code contains functions for several operations that are related (Roberts & Chapin 2017, p. 8). For comparison, hosting model for FaaS is shown in figure 9.

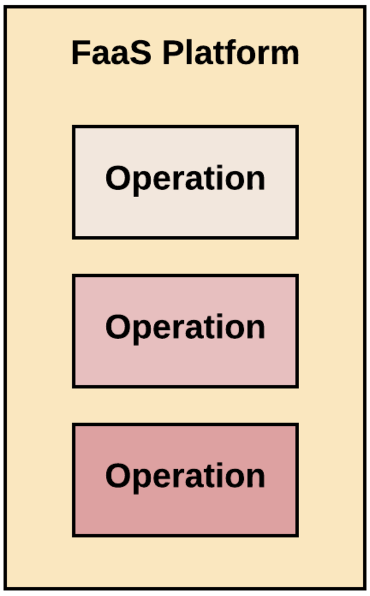


Figure 9: FaaS hosting (Roberts & Chapin 2017, p. 8).

In figure 9 it can seen how in FaaS model both host instance and application process has been stripped away. Functions that were part of application are now deployed individually to cloud-vendor platform and they are not always active in server process (Roberts & Chapin 2017, pp. 8–9).

3.3 Impacts on architecture

Taking a traditional web application for an example, there's a server component responsible for most of the business logic and client application is relatively thin. This server will also act as a gateway between client and other backend resources that require access control, for example a relational database. Simplified illustration of this architecture can be seen in figure 10 by Roberts (2016).

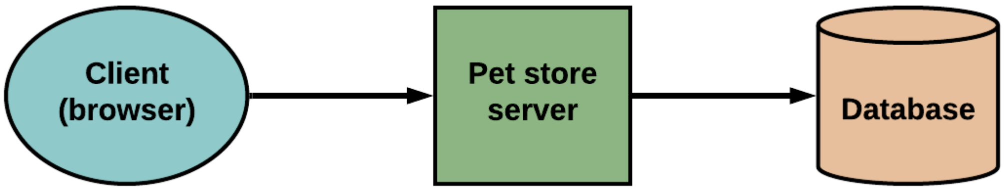


Figure 10: Tradional web application architecture (Roberts 2016).

In traditional client-server architecture server has most of the business logic and as Adzic & Chatley (2017) explain it's important to notice that there's usually also a server process that's typically listening for TCP (Transmission Control Protocol) socket, waiting for client connections and sending responses back. In serverless model developers are no longer responsible for this server process that listens to TCP socket as the serverless platform is responsible for client requests and responding to them (Adzic & Chatley 2017). While this traditional client-server architecture is perfectly functional, Roberts & Chapin (2017) remind that there are many additional responsibilities not directly related to the business logic of the application: server and database need to be configured and maintained, along with taking care of things like security, scalability and high availability. Roberts (2016) argues that in serverless world this same application could have architecture looking something like in figure 11.

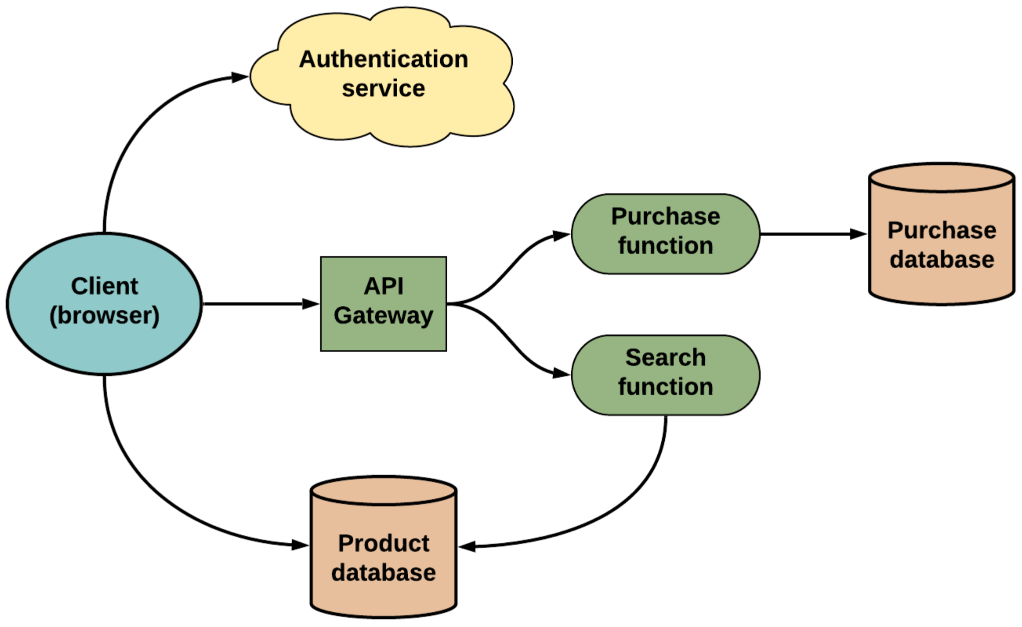


Figure 11: Web app with serverless architecture, adapted from Roberts (2016).

Compared to traditional architecture, it can be seen in figure 11 that client is now allowed to connect directly to backend services like authentication and product database. This also means that client now has some of the business logic that was previously in server component. It's also good to notice that client doesn't have full access to product dataset but limited permissions. Not all functionality is in client, though, but some of them are implemented as FaaS functions that are fronted by API gateway. Good candidates for functions are compute intensive functionality like search or actions that require higher level of security, like purchasing products (Roberts 2016). Both Roberts (2016) and

Adzic & Chatley (2017) make the case that in serverless application client would have more business logic in it when compared to traditional web application. Roberts (2016) reminds that repetition of logic across clients is something that can be considered as a drawback and needs to be taken into consideration when doing technical design on how clients talk with backend services. Roberts & Chapin (2017, p. 17) argue that in comparison to traditional clien-server architecture, serverless version of the application can focus more on the unique business logic instead of infrastructure management and there's less friction of changing functionality as components are decoupled. Security, scalability and high availability are now baked into components, meaning that there's no need to worry about investing into infrastructure if application gains more popularity.

As Adzic & Chatley (2017) explain, in traditional client-server model backend resources implicitly trust the server that acts as a gatekeeper and thus direct access from client to resources like database is considered as a security risk. Without a gatekeeper and due to the fact that two sequentical requests from the same client might end up into separate instances, a distributed request-level authorizations are needed for connections to backend resources. AWS offers services like AWS IAM and AWS Cognito for fine-grained control on what kind of permissions client have on resources (Adzic & Chatley 2017).

Even if serverless computing abstracts away things like load balancing, scaling, availability and security maintenance of infrastructure, there are many operational concerns left to look after, for example making sure that applications are testable, secure and resilient. Versioning and deployment strategy are also needed (Eivy 2017). Roberts (2016) warns about false sense of security and argues that technical education is needed for operating serverless applications as developers have more responsibilities that system administrators used to handle with serverful applications.

3.4 Serverless costs

In traditional cloud computing, billing is based on allocation, not on actual use. Take virtual servers as an example, users are billed for them even if they don't use them at all. This can be compared to the difference between using rental cars and taxi: you are paying for the rental even if you don't use it for a week, unlike a taxi (Eivy 2017). With serverless computing, significant reductions in hosting costs are possible. Villamizar et al. (2016) prove this in laboratory experiment, where as Adzic & Chatley (2017) present real-life case studies where cost savings were achieved.

With serverless computing costs it's important to understand what kind of traffic to expect. For an public API, traffic might be really bursty and thus weekly or monthly average doesn't give enough information for capacity planning. One important measure here is RPS (requests per second). For example an API that receives 60 thousand requests per minute might get 90% of requests during a single 10s period, conducting a burst of 5400 RPS compared to 1000 RPS that flat traffic would account for. Figure 12 illustrates this, showing how differently 60 thousand requests might distribute in one minute time period. Big difference like this is something that engineers need to consider when designing API scaling and capacity needs (Eivy 2017).

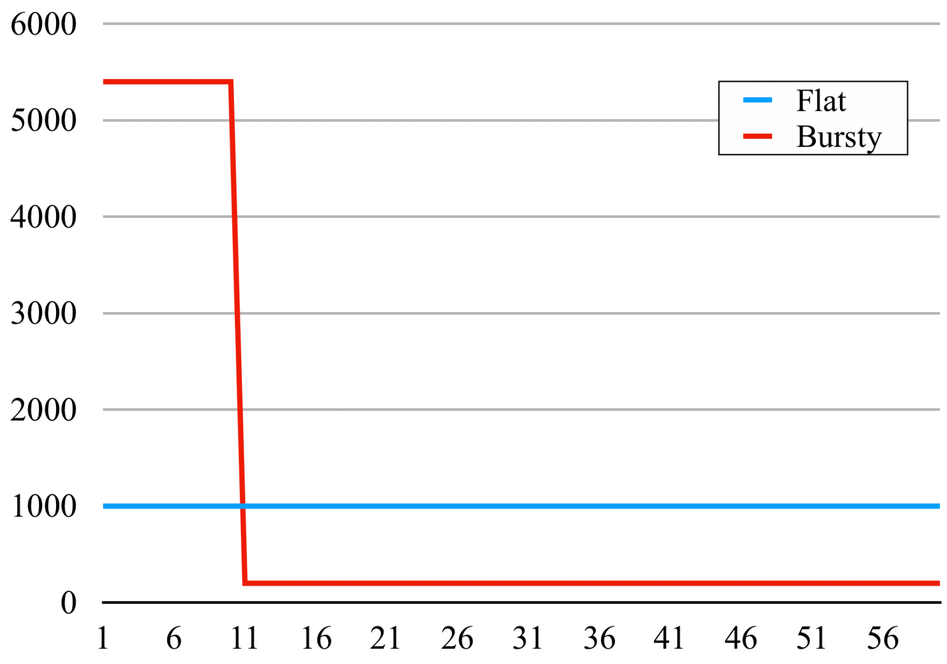


Figure 12: Two different patterns for 60 thousand requests (Eivy 2017).

With public APIs backed by serverless function, one should always use some kind of rate limiting in API gateway as otherwise simple denial-of-service attack may lead into large costs because of auto-scaling (Eivy 2017). These kinds of attacks that aim to create financial costs by taking an advantage of auto-scaling nature of serverless are sometimes referred as denial-of-wallet attacks (Dooley et al. 2019). For example AWS Lambda won't scale infinitely though, as there's a default limit of 1000 concurrent Lambda containers across user's AWS account (Roberts & Chapin 2017, p. 24). Even if serverless providers offer cheap-looking cost per-hit, with large amount of traffic functions might get more expensive than traditional solutions. Eivy (2017) presents an example where API endpoint with 150 RPS was a bit cheaper in Lambda platform in comparison to using classic on-demand instances, but service was expected to reach

30 thousand RPS next year and with such traffic serverless solution would get three times more expensive when average execution time of 50 ms with 128 MB memory allocation were assumed. Eivy (2017) recommends modeling the economic impact of architecture and operational choices beforehand – and while doing it, calculations should also be done for the potential future scale in addition to current traffic levels. With serverless functions modeling should be done using the actual functions in cloud infrastructure instead of local machine to get numbers that really reflect the reality. Eivy (2017) also reminds that even if price per-request might get higher for serverless solution, operational costs might be lower in comparison as autoscale, security patches and loadbalancing are taken care of by the provider.

When optimizing performance, one should remember that many times the provider will round up function execution time. For example with Lambda, this means that user will always pay at least for 100ms of execution time – meaning that optimizing runtime from 50ms to 20ms would be a wasted effort regarding costs (Eivy 2017). With really high RPS it might still make sense to optimize even when being under minimum billable unit as one could avoid hitting concurrent invocation limit.

As Jonas et al. (2019) writes, serverless computing might have somewhat unpredictable costs due to the nature of the pay-as-you-go model and thus it's not that straightforward to estimate future costs. There's no concept of pre-planned capacity on serverless platforms, as CNCF (2018) reminds. This fact might be undesired for some organizations, although as serverless approach is used more, it can be expected that organisations become better in doing estimates based on history (Jonas et al. 2019).

3.5 Benefits of serverless

Roberts & Chapin (2017) list five benefits for infrastructural outsourcing that are also elements of serverless: reduced cost of labor and resource, reduced risk, increased flexibility of scaling and shorter lead time. First four are mainly about cost savings where as the last item, shorter lead time, actually allows developing new functionality faster. Stigler (2018, pp. 6–7) lists similar benefits, highlighting speed of development with less stress about infrastructure management, lower costs and enhanced scalability.

Reduced labor cost is due to the fact that there's no longer a need to maintain your own server process, development team can focus on unique business logic and state of the application instead of looking after operation systems, database version updates and so on. Labor costs are also saved if separate BaaS components are used – for example, if authentication service is provided as BaaS, there is less logic that needs to be developed (Roberts & Chapin 2017, p. 20). Risks are reduced due to the fact that there are less different systems and components to maintain. As outsourced components are usually the ones like NoSQL databases that require less attention and infrequent updates in comparison to core application functionality, when problems occur with such components they are more difficult to solve as team usually isn't as familiar with them as with core business logic – thus risk is reduced when these components are outsourced to serverless platform (Roberts & Chapin 2017, pp. 21–22). Resource costs are saved as there's no need for host management: in serverless approach one does not need to plan, allocate or provision resources. Thus, there's also no need to to prepare for peaks by over-provisioning - that is, having resources available just in case traffic increases suddenly. Automatic scaling is available for non-serverless applications, but it's not as precise as with serverless (Roberts & Chapin 2017, p. 23). More flexible scaling is gained by the fact that scaling is totally automatic: there's no need to set up scaling by configuring auto-scaling groups or monitor processes (Roberts & Chapin 2017, p. 25). Shorter lead time is defined as the time it takes from initial concept to deployment of MVP (minimun viable product) into production enviroment. With serverless model this time is highly reduced as there's a lot of complexity of building, deploying and operating applications that are removed. It should also be remembered that these kinds of MVPs are highly available, scalable and fast to deploy from day one (Roberts & Chapin 2017, pp. 25–26). Also Cockcroft (2017) writes about this, reminding that prototype application on serverless platform is by nature scalable for production use.

3.6 Limitations of serverless

Roberts & Chapin (2017) introduces two kinds of limitations of serverless: inherent limitations and implementation limitations. There's never going to be a way to get totally around the inherent limitations but it might be possible to learn to embrace some of them. Implementation limitations are in fact limitations as of now but as serverless approach matures and ecosystem improves, some rapid improvements are expected.

The most obvious inherent limitation of serverless is that most of the components are stateless. While it makes scaling more straightforward as it can be done just by increasing concurrency it does mean that stateless components are required to communicate with stateful components in order to persist information - which in further will introduce latency and additional complexity (Roberts & Chapin 2017, p. 28; Hellerstein et al. 2018). Current BaaS options for storage are bit lacking, for example autoscaling can be slow and costs high (Jonas et al. 2019). Latency is a concern in two ways: first is the latency between components and the second is the impact of the so called cold starts where new instance is initialized instead of using existing one. As communication between components is usually implemented via HTTP APIs, it's slower than communication inside the same host in non-serverless application that may use faster transports (Roberts & Chapin 2017, p. 29). Local testing of serverless applications is challenging because large part of the infrastructure is abstracted away inside the platform, making it difficult to connect components in the same way as it happens when deployed. These limitations apply to integration and end-to-end testing whereas unit tests can be implemented as with any application (Roberts & Chapin 2017, p. 29). As the BaaS and FaaS platforms are managed by a third party, control over the software stack is given to the provider in various ways. Configuration options are limited or non-existing. For example, there's no control over operating system runtime parameters. Control over performance-related matters is limited as there's no access to operating system or runtime process. Same FaaS function may have different performance characteristics depending on when it's executed. Also response times in BaaS components might be inconsistent and user doesn't really have other options than contacting the platform provider for support. This is actually the only option with any issue other than the ones directly related to serverless application code or configuration. Control is lost also in security-related matters. If AWS API Gateway is used, it's always accessible to public internet, there's no option to limit access for example by IP address (Roberts & Chapin 2017, pp. 30–32).

On implementation limitations, Roberts & Chapin (2017) mention the following: inconsistent and sparingly documented performance characteristics, tooling limitations and vendor lock-in. With FaaS platforms, most common performance issue is the so called cold start. It happens when new container instance is created for the function invocation, for example when function is called for the first time after a change, or after a longer pause or when new concurrent instances are needed for automatic scaling. Once instantiated, subsequent requests don't require running the initilization process, making these requests much faster (Roberts & Chapin 2017, pp. 32–33). Selected runtime makes a big difference on instantiation speed, for example JavaScript and Python are fast where as Java takes significantly longer (Adzic & Chatley 2017). On AWS Lambda platform, containers are said to stay warm for hours (Roberts & Chapin 2017, pp. 32–33), although much shorter times have been also reported (Adzic & Chatley 2017; Vojta 2016). Platform providers like AWS don't reveal any numbers nor do they offer any strict service-level agreements, so performance characteristics are based more or less on observations (Adzic & Chatley 2017).

Tools for deployment, developments and monitoring are still on early stages and limited. Best practices and patterns are still shaping up. As serverless applications consist of many individual components, orchestrating deployment of the whole application can be challenging. FaaS environments have limited execution environment regarding on CPU, memory, disk and I/O (input/output) resources. Execution times are also limited, for example AWS Lambda has maximum execution time of five minutes (Roberts & Chapin 2017, pp. 33–35). Hellerstein et al. (2018) mention these same limitations and argue that current FaaS offering is too restricting for data-centric distributed computing because of limitations in both function lifetimes and I/O bandwidth, along with slow communication between functions and no option to use specialized hardware. Roberts & Chapin (2017, pp. 33–35) continue by writing that monitoring and logging in serverless applications is more complicated as applications consist of multiple individual components, making it harder to track request that is processed by multiple services. This same challenge of distributed monitoring is also present with microservices. Difficulties on local testing were already mentioned, but possibilities for remote testing are also quite limited at the moment. Component-level testing is usually possible, but testing the whole serverless application safely can be complicated. Debugging remote application is currently not possible on most platforms and runtimes.

There’s always some level of vendor lock-in with serverless applications, although different levels of lock-in are enforced by different providers, depending on their APIs, documentation and integration patterns. Then again, using a single vendor is beneficial as components are well integrated, meaning that lock-in is present more in the way how components are tied together than in the components itself (Roberts & Chapin 2017, pp. 35–36). For example, in Lambda environment code that is executed depends little on the environment but as the platform provides various services like authentication, monitoring and configuration management, and the fact that serverless architectures provide incentives let clients communicate directly with backend services, client code may become tighly-coupled with the different services offered by the vendor. Consequently, changing provider would require significant rewrite of client code (Adzic & Chatley 2017). Newman (2017) argues that one should think more about migration costs than the actual vendor lock-in. There is varying difficulty in switching components between vendors in serverless application and it should be considered whether to prepare for that for example by building abstraction layers or accept the risk that some day there might be migration to new vendor.

3.7 Summary

In this chapter, the concept of serverless computing was introduced and defined as a way of developing and running applications that don't require server management and that are billed based on actual execution times. Its impact on software architecture and hosting costs were examined. In addition, known benefits and limitations of serverless were presented. In the next chapter, real world experiences of splitting services into smaller units and into serverless microservices are discussed.

4. Case study

A Finnish video on demand service is used as a case study. Service offers video content both as free and as SVOD (Subscription Video on Demand), meaning that part of the content requires paid subscription. As the service is owned by a TV broadcasting company, it integrates into broadcast TV by making it possible to watch live TV, view airing schedule and watch most of the content aired on TV later on-demand. Service launched in 2009 with a web service, nowadays web client is accompanied with native client applications available for mobile platforms iOS and Android and also on smart TV platforms such as Samsung, LG and Apple TV. In addition to video content, audio-only content is supported as well. Audio content is managed by the same system as video content but separate client applications are used for consuming audio content. Development is done in a multi-vendor environment where in-house engineering and design team is supported by contractors from various vendors. Most of the development teams work in what Narayan (2018) calls product-mode, meaning that instead of projects there are near-permanent teams working on a persistent business issue, such as mobile applications or web client, for long periods of time and there's no separation between build and maintenance teams. In this sense, "you build it, you run it" principle is in use.

The organization that owns the video service has made a strategic decision years ago to host all services in AWS, where applicable. Hence, the video service too has been gradually transforming into using more and more AWS services. Lately also serverless approach has been used in various services. Also, infrastructure as code (IaC) approach is used for configuring all cloud resources in an organised way in definition files that are under version control. The transition process could be described as evolution instead of revolution and modern cloud native architectures and approaches have been taken into use in controlled fashion and step by step instead of big bang rewrites.

4.1 Overview of the system

Overview of the video publishing system is shown in figure 13.

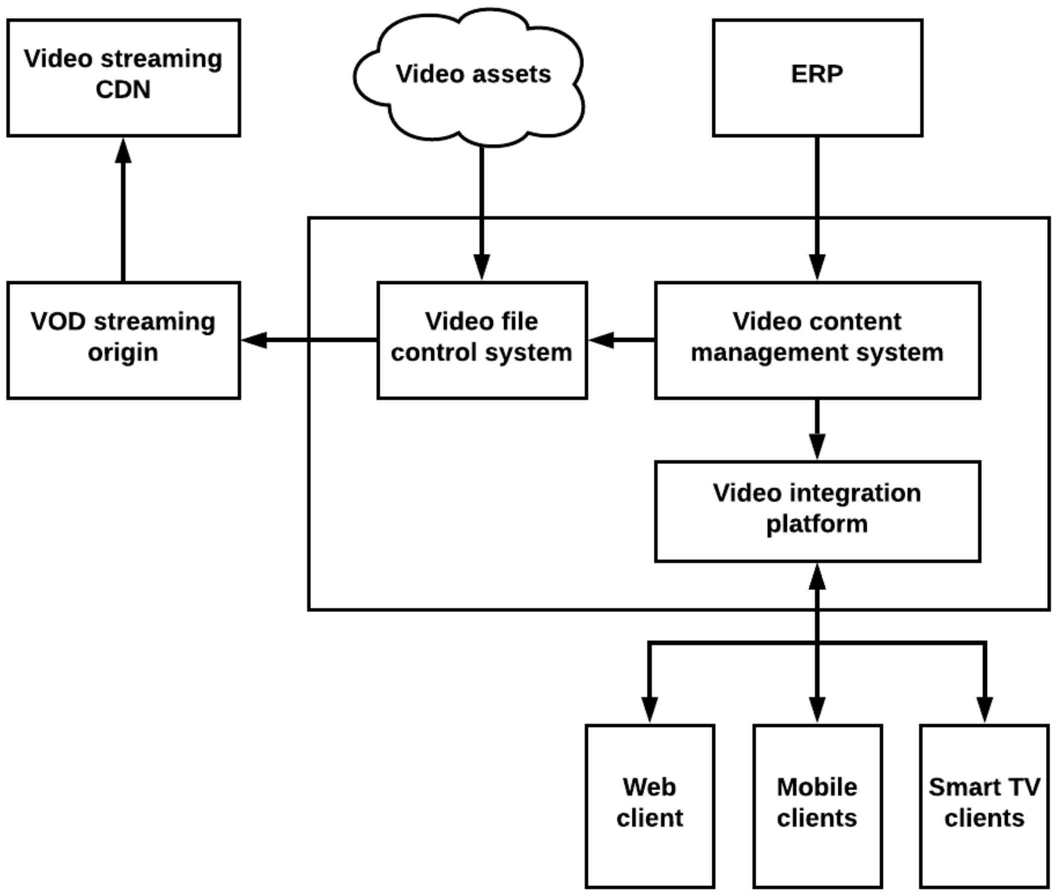


Figure 13: Video publishing platform overview.

In figure 13 main components of the video publishing platform are presented. ERP (Enterprise resource planning) is the system where broadcast airing schedule is managed and information is imported into video content management system (CMS). Most of the meta information originates from ERP, including series, season and video text fields and images. Video files originate from various sources and information about them is automatically sent to CMS. It's also possible for content editors to upload videos directly to CMS. Video files are processed by video file control system and for example different quality versions are generated. Files are then pushed into CDN (Content Delivery Network). Video CMS is a Drupal instance and while great for managing content, in high performance environment it's not suitable for offering APIs due to challenges with scaling, performance and availability. Thus, video integration platform is used where all the content is indexed and then offered in JSON format via various APIs. In addition to content APIs in video integration platform, it also has an API that defines what components different clients should display and how these components should be formatted. Client applications themselves are headless so that content isn't imported but merely cached in them.

In addition to core video system presented in figure 13 there are plenty of other supporting systems as well, for example applications for e-commerce, integrations to single sign-on (SSO), service for authenticating to smart TV via client app, services to manage user entitlements, and so on. As a large ecosystem with over ten years of history these are hosted and built in different ways: some are running on traditional VMs, others are serverful apps in Kubernetes-managed cluster and some are true serverless applications.

4.2 From monoliths to microservices and serverless

Over the years the video publishing platform has steadily moved from monolithic applications towards more decoupled and smaller services, many of them can now be categorized as microservices. For example, web client started as a monolithic Drupal application that provided not only video viewing UI but also e-commerce functionality and it imported all the video content into its own database. In the early days of the service there still wasn't video integration platform APIs available that would have been flexible and performant enough to be used by a headless application. Later the web client has been split into smaller applications, for example e-commerce components that provide checkout pages, voucher redeeming and entitlement updating were split into its own application that while still a monolith, was now separated from the video viewing UI. Later this e-commerce focused monolith application was replaced with API-driven serverful cloud application managed by Kubernetes. As consuming video content is more or less read-only activity, totally different caching patterns were then possible compared to the e-commerce site that involved heavy write requests to various backend services, making performance optimizations much easier. Better and more flexible content APIs were also introduced, making it eventually possible for web client to stop importing all the content into its own database and instead just render content provided by content APIs, on-the-fly.

4.3 Serverless in use

While the organization owning the video publishing platform has step by step moved over to AWS, the first questions when building a new service (or updating the existing one) have been how to best leverage various AWS services and whether serverless approach could be used. At an accelerating pace the answer to the latter question has been yes. Next, various real-life examples of serverless approach are presented and discussed.

4.3.1 New web client and Component API

In 2018, the web client was renewed and implemented with React. As a decoupled application it doesn't have its own database: all the content is loaded from different content APIs. Also the site structure - which components should be shown on which page (or a view in the context of an application) and what kind of content these components should list and how - is defined by a separate service. We call this service Structure API as that's what it does: defines the structure of the service. As a high-level architectural decision it has been decided that only limited amount of business logic should be in clients themselves. Whenever feasible, the goal is to avoid duplicated business logic across different clients. In a sense, this can be seen as a bit of an opposition on common serverless architecture where clients actually would have *more* logic in themselves, communicating directly with backend services and skipping central gateway (a server), as explained in chapter 3.3. So far it has seemed to be more efficient and economical to keep client applications as simple as possible and avoid duplicating business logic whenever possible. As all applications are native, they can't really share any code and thus dedicated developers are required. One could say that we want to avoid updating native applications at all costs because of change in the backend.

Structure API defines what components each web page (or view in applications) should have and what formatter should be used to render it. It also defines parameters that should be used in a request to content API. As a simplified example, it might define that third element on the frontpage should list items from video content API, filter by category to list only movies and sort by weight defined by queue managed by human editor. And it should list these videos using the formatter called "video_thumbnails". There's a detailed design and definition how formatter "video_thumbnails" should look like, but it's up to each client to implement it. As content APIs offer only generic data, it's these formatters that define things such as what date formats to use, whether or not the title should be prefixed with "Movie:" or if only a thumbnail should be shown or title as well. Duplicating these kinds of simple logic across all the clients is not costly, but there are a lot more complex components as well. For example we might want to list series based on the most popular episodes. Here the content API query would be against the video API but rendered items should not be videos but series. Another complexity is the fact that many listings are contextual: on series page list of seasons and episodes should be filtered based on the currently viewed series. Eventually things add up and there's a lot of logic to deal with.

This is where the idea of Component API was born. Some of the logic could be moved to separate service that acts between client application, Structure API and Content APIs. It would process the component definitions from Structure API, load items from content APIs and parse results into JSON that client could simply render without further processing or complex logic to figure out how to merge information from various content APIs. Component API is first used by web client, but later it could be taken into use with other clients as well. Main AWS services of Component API are presented in figure 14.

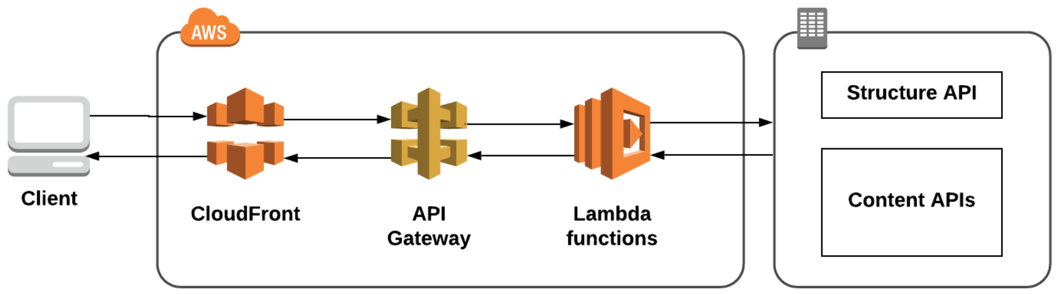
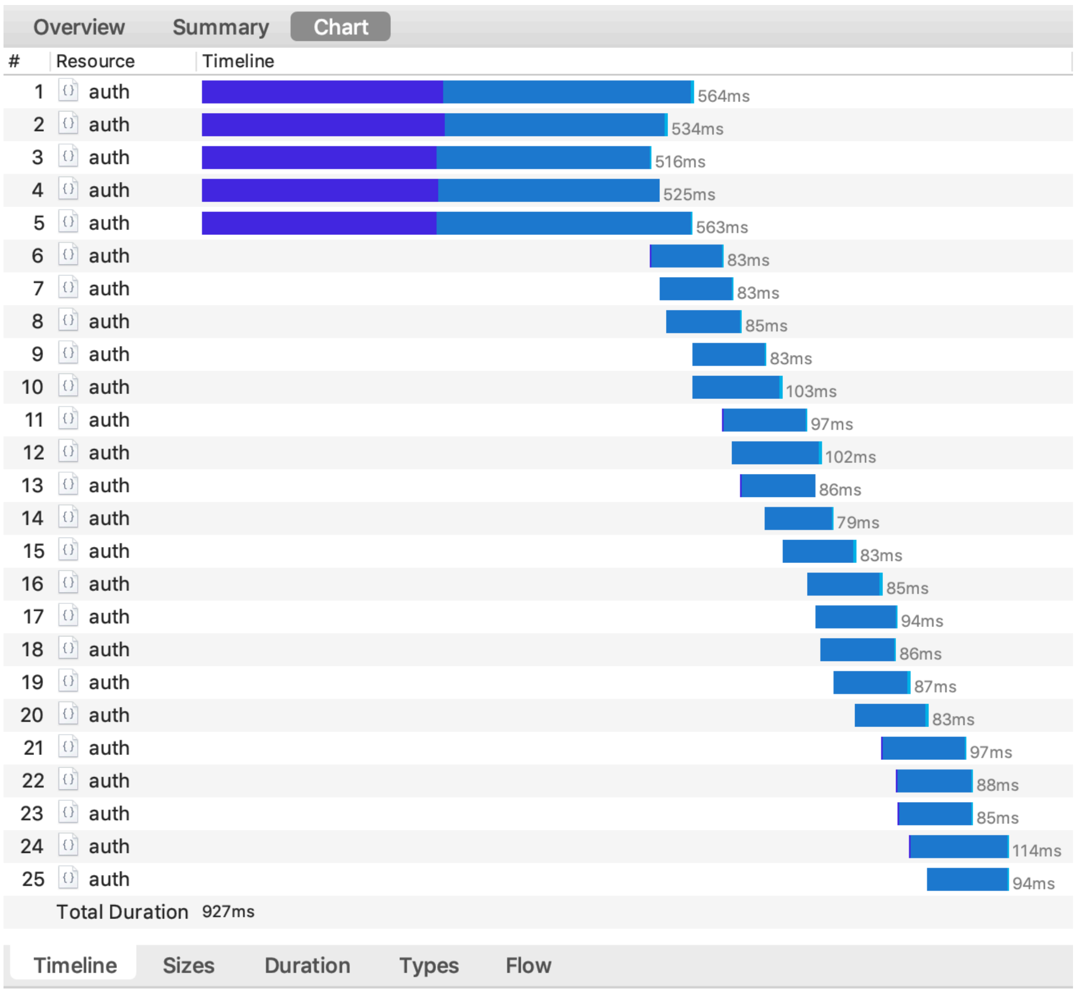


Figure 14: Component API AWS architecture.

In figure 14, AWS services used by Component API are shown. This model is common for applications that expose public HTTP API endpoints to internet. AWS content delivery network (CDN) service CloudFront is in front of API Gateway that is a service to manage and maintain APIs. Lambda functions include the business logic for interpreting the instructions defined in Structure API and using these instructions to fetch content from various APIs for aggregation. Component API is fully stateless and read-only, thus there's no need for storage services like DynamoDB or S3.

4.3.2 Auth API and the issue of cold starts

All clients, including the web client, use the same SSO system. A microservice that we call Auth API is used as kind of a proxy service between web client and the SSO system. Auth API consists of Lambda functions fronted by API Gateway, similarly as in Component API. Authentication process requires saving state: a token generated by the client in the beginning of login or registration process and when user is redirected from the SSO back into client, state is carried from the SSO authentication pages as a parameter. This state is then compared to initial value and only if they match should the user be authenticated. An in-memory storage Redis by AWS Elasticache service is used to store state and possible referrer URL. As AWS invoices for Redis instances even if they aren't used, Auth API can't be considered truly serverless application. Because of Redis, Lambda function needs to be a part of the same subnet as Redis. In AWS, this means using VPC (Virtual Private Cloud) and ENI (Elastic Network Interface). Unfortunately, using VPC drastically increases function cold start times. In our tests we experienced wait times as high as eight seconds for the container initilization, this being in line with observations by Cui (2018). In the case of cold start, end-user would need to wait for over eight seconds for login screen to load in the worst case. So called warmer function could be used to make sure there's one container always available, but we would still have the issue of autoscaling: if multiple requests would hit exactly the same time, then all of these concurrent requests would trigger a cold start, as demonstrated by Cui (2018) and confirmed in my own test as well. Visualization of this test can be seen in figure 15.



#	Resource	Timeline
1	auth	564ms
2	auth	534ms
3	auth	516ms
4	auth	525ms
5	auth	563ms
6	auth	83ms
7	auth	83ms
8	auth	85ms
9	auth	83ms
10	auth	103ms
11	auth	97ms
12	auth	102ms
13	auth	86ms
14	auth	79ms
15	auth	83ms
16	auth	85ms
17	auth	94ms
18	auth	86ms
19	auth	87ms
20	auth	83ms
21	auth	97ms
22	auth	88ms
23	auth	85ms
24	auth	114ms
25	auth	94ms
	Total Duration	927ms

Figure 15: First concurrent requests causing a cold start in Lambda.

In figure 15 screenshot from web proxy debugging application Charles is presented. In the test scenario 25 requests were done with a concurrency of five. We can clearly see how the first five concurrent requests caused a cold start in Lambda and subsequent requests were served from warm containers with significantly faster response times. This is a known limitation of Lambda platform and something users of Lambda platform just need to live with, at least for now. If traffic spikes are to be expected, then various pre-warming methods could be used, although all these kinds of hacks fight against the promise of serverless computing that you would not need to worry about things like scaling. With Auth API, large amount of concurrent requests are luckily rare and none of the authentication related functions take more than 120ms on average to execute, making it quite rare that there isn't a warm container available for the request. Traffic is quite steady around the day and thus there's usually always at least one container warm. However, it would be better to use DynamoDB instead of Redis as DynamoDB would be fast enough and it also supports On-Demand mode nowadays that would allow more efficient auto-scaling and pricing model that would make Auth API truly serverless.

4.3.3 Background tasks

FaaS is a great solution for tasks that are executed periodically and can happen in the background. In such use cases delay caused by a cold start is not an issue as there's no risk of UI interaction being on hold because of container initilization. And with serverless computing, there's no need to pay for idle time between periodically executed tasks.

In the video integration platform, Lambda functions are used to update various fields in the Elasticsearch index. For example, each season of a series has fields for various counters: how many videos there are in total, how many of them can be viewed without paid subscription and so on. Periodically executed Lambda function fetches required information and updates counters. Similar functions are used to update so called premium statuses of series and seasons. One more example is a function that updates weight used for sorting in various content types, for example a series might be scheduled to show up as first item in the listings for certain timeframe.

Web client offers a sitemap in order to help search engine crawlers to be aware of every content URL. A periodically run Lambda function loads content from different content APIs and generates these XML (Extensible Markup Language) files that are then stored in S3 storage. Cloudfront distribution is configured to map sitemap file URLs into S3 origin. Previously sitemap generation process used to be a Drupal module function call triggered by a server process cron task, but with Lambda it's now totally separate component that can be developed, tested and deployed independently from the web application. At the time of writing, Lambda functions still have a 15 minute execution time limit. This is something to keep in mind with long-running tasks like sitemap generation, although luckily most of the current content APIs allow loading paginated results in parallel, thus making it possible to generate sitemap in just under two minutes. In future sitemap generation might need to be split into separate functions, each one taking care of a smaller chunk of the sitemap and then a last function that would build the final sitemap from generated chunks. For example Lambda Step functions could be used to handle this kind of sequal process.

4.3.4 Using FaaS to manage infrastructure

As mentioned by Roberts (2016), one common way to use FaaS is to use it to "glue" code in operations. In our use we have used small functions for example to subscribe new applications to logger function that then submits application logs into Datadog that is an external monitoring and log aggregation service. Another example is a case where team experimented with Lambda function that altered how AWS auto-scales DynamoDB. By default DynamoDB scales up a bit slowly based on consumed read and write capacity units. With Lambda function it would be possible to alter scaling logic and for example count new provisioned capacity by yourself based on request count and thus make it scale up early and then scale down slowly. This would be more suitable for a quick burst of traffic, for example during the couple of minutes before popular ice hockey match when a lot of users are authenticating and creating new subscriptions. FaaS is also a great fit for chatbots and one example of this kind of usage is a serverless application that posts status messages to Slack channels based on open pull requests GitHub, providing useful information for the developers.

4.3.5 Video playback token verification with Lambda@Edge

During video playback the player will periodically make an authorization request to backend. An access token based on JWT (JSON Web Token) standard is used. As verification of JWT is really quick stateless authentication operation, it's a good use-case for Lambda@Edge functions. AWS (2019) defines Lambda@Edge functions as feature of CloudFront CDN and they allow modifying request and response on edge. This function is executed in massive amounts but as average function execution time only 1 ms, it's possible to reach high number of requests per seconds without actually having that many concurrent function invocations. As possible request count per second might reach tens of thousands for this function, calculations were done carefully taking possible traffic bursts and future demand in to account, as Eivy (2017) recommends and was examined in chapter 3.4.

4.3.6 Radio playlist API

There is a web service for audio content that is managed by the video publishing platform. Each radio station has its own section on the web service and these sections contain a list of songs broadcasted on radio channel. A serverless microservice was built to store playlist history and it offers API for saving and retrieving song information. Broadcasting software used by radio stations posts the song to microservice which saves it into DynamoDB database. Microservice offers an API that clients can use to fetch list of played songs for each radio channel. This microservice consists of around 250 lines of TypeScript code for business logic and a bit over 500 lines of Terraform code used for defining various cloud resources like API Gateway, Lambda function, caching rules, DynamoDB table, deployment pipeline and various permission related resources like IAM roles and policies.

4.4 Thoughts on serverless experiences

One of the biggest benefits of serverless approach is that it almost completely removes the burden of capacity forecasting. While traffic for video on demand service is quite predictable as most of the video viewing is done during peaktime at evenings, sudden bursts might still happen anytime of the day due to breaking news, for example. Capacity forecasting is hard: estimate low and customers will face long response times or even downtime, and if estimated too high, you end up paying for idle capacity. Some years ago video service actually was facing downtime as critical DynamoDB table wasn't configured to have enough read capacity. When you can trust the platform to do scaling automatically, a lot of work time can be saved and there's less stress about availability.

With hosting costs there haven't been big savings yet. Sometimes serverless application in AWS was actually more expensive than previous version of the app hosted on VMs. Then again, most of the applications were already quite cheap to host. It's not uncommon that the amount of monthly hosting costs for an application was less than one day of development work invoiced by a contractor. If there are cost savings to see, it's from the decreased amount of development time needed. If one day of development time might cost more than hosting of multiple microservices for a month, it's clear that bigger savings can be done here. Usually smaller codebases are easier to develop with and a lot of development time can be saved when there's less need for implementing and testing capacity scaling. It can be said that serverless approach can save money but with current implementations in video service it's not due to reduced hosting costs but because of less effort is needed for development and infrastructure management.

Video service is developed in a multi-vendor environment. One important benefit of smaller services has been the additional flexibility in resourcing: with smaller services there are less dependencies compared to monolithic application where feature development and deployments require more coordination. Thus, building smaller services have enabled teams to work more in parallel. One goal has been to try to avoid locking developers to work with only one part of the system and thus avoid leaving overall knowledge only to individual developers. This can be seen as a way to do risk management. Developer happiness has also increased as people have been able to work with more varied tasks instead of sticking in their own sandbox. Automatic build processes and deployment pipelines can be seen as an improvement to developer experience as well as it reduces the need to worry about sometimes tedious work of building and deploying. Setting up these deployment workflows seems to still require quite a lot of effort even with serverless applications, but luckily configurations can be re-used in other projects. Smaller codebases make code more readable and secure. More granular services make it also easier for new developers to make changes to code they haven't written themself in the first place. This also means that onboarding new team members is easier.

When an organization has a lot of services and resources in cloud, it's desirable to use IaC approach to have all of these resources defined in code and under version control so it's easy to keep track of resources and their changes. There are languages like AWS's own Cloudformation and HashiCorp's Terraform that are used to define infrastructure as code. While AWS offers visual tool like AWS CloudFormation Designer for defining cloud resources, configuration of cloud resources is still usually manual work of writing definitions for resources by hand. This can be tedious work and as mentioned in chapter 4.3.6, for small application there might be twice as much code for infrastructure configuration than for actual business logic. While it's true that there's less to worry about regarding infrastructure as things like auto-scaling and high-availability are taken care of by the vendor, the developers would still need to spend considerable amount of time with tasks that can't be considered to be directly linked to business logic.

Local development for smaller applications that consist of function or two is efficient. With monolithic Drupal applications, setting up and configuring local development environment with virtual machine was always a bit tricky and time-consuming. Then again, a microservice that depends on PaaS services like DynamoDB can make local setup complex quite quickly and local testing more challenging. Best practices and tooling for local development are changing rapidly and are still somewhat lacking. For example, earlier versions of AWS SAM CLI tool that is used for local development and testing of serverless application crashed when API was hit concurrent requests.

Monitoring serverless microservices is more of a challenge compared to tradional applications, as Newman (2017) writes. For example, tracking a request that is handled by multiple microservices is challenging. While AWS offers services like Cloudwatch, X-Ray and CloudWatch Logs Insight, these still leave a lot to be desired for larger applications. Video services uses monitoring service Datadog with some of its applications and it can show visualizations of Cloudwatch metrics and log messages. Using external service for log aggregation does introduce a slight delay that is a bit problematic if there's a need to get a realtime view of the application status. With serverless applications it also seems to be more likely that one needs to add additional code into application code itself in order to generate more advanced traces, to mention one example. This is something that we would usually want to avoid as it introduces tight coupling to the external monitoring service.

5. Conclusions

The promise of building small serverless applications is the introduction of effiency and cost savings as development team would be able to focus more on the unique business logic instead of managing infrastructure and challenges like scaling, high-availability and server security. Granular pay-as-you-go billing model is said to offer significant savings on hosting costs. The aim of this thesis was to examine the characteristics of microservices and serverless approach, find out what literature has to say about their pros and cons and compare learnings to real-life experiences on video on-demand service. Furthermore, the aim was to gather information on what are the suitable use cases for serverless approach today.

Our experiences have shown that relying on cloud computing and building smaller, loosely coupled applications has had a positive impact on quality and availability of services. Smaller is better, as Nadareishvili et al. (2016, pp. 65–66) writes. In addition, there have been clear benefits on how to manage work: smaller deployable units have enabled to work in parallel easier and when applications have smaller codebases, it's more effortless to onboard new engineers. So even if "you build it, you run it" principle isn't always possible, there's still the benefit of easier onboarding and making it easier for engineers to work on different projects. Smaller services makes it faster to peer review code as well as it's easier to get your head around smaller codebase compared to monolithic application – this is also good for writing secure code.

While the amount of execution time one can get for free is often shouted out loud when talking about serverless, often in real life cost savings in hosting aren't the main driving factor to adapt serverless computing. While costs savings are real – specifically with small applications that experience a lot of idle time, such as cron jobs – the actual savings can be found from the reduced work related to infrastructure management and having more time to dedicate for building actual business logic. It's true that building serverless components removes most of the burden related to auto-scaling and high-availability, but a considerable amount of work is still required for defining cloud resources when infrastructure as code approach is used. Luckily, many of these definitions can be reused between different applications and it was found to be benefial to put some effort for creating templates for commonly needed definitions. On security side of things, it can be argued that many of the easiest attack vectors are blocked as cloud vendor takes care of patching vulnerable OS dependencies on servers and also provides services for coping with denial-of-service attacks. Fine-grained isolation of small functions written in high-level languages on FaaS platform is also benefial for security, but attention is needed in configuration of security policies.

It's easy to agree with Wagner (2018) when he argues that the real value of serverless is in the operating model. Then again, for small businesses such as new startup companies, the pricing of FaaS and BaaS services alone is really attractive and arguably the best way to go when starting from scratch. And as Cockcroft (2017) writes, prototyping a serverless application is fast and the application on serverless platform is automatically scalable for production use, highly available, high utilization and fast to deploy. You may write your prototype for just a few users, but it will scale for millions – a huge benefit when considering things like time-to-market in a competitive environment. This all comes with a price of giving away a lot of control over infrastructure to cloud provider, which migh be even impossible in some cases, for example in public sector.

It was apparent that serverless computing is still a new concept. Tooling is evolving rapidly and patterns on how to bundle functions, how to develop serverless application, manage versions and deployment workflows are still shaping up. This means that one needs to prepare some additional work for setting up tooling, workflows and designing patterns when starting to adapt serverless approach. It's also good to remember that the question of what well-formed serverless application looks like remains still a bit open. Also, for critical services with really high requirements for performance, low latency and availability, serverful application is likely still a better option as FaaS does introduce additional latency and the issue of cold start is very much a reality, not to mention the fact that one could want more control over infrastructure.

It's still early days for serverless computing and things will keep changing fast. However, based on literature and experiences from the case study presented in chapter 4, serverless computing is already widely useful for various kinds of services – especially for services where really low latency isn't required and where things like high-throughput and time-to-market are important. Thus, it's easy to agree with Jonas et al. (2019) and see serverless approach becoming the default paradigm of the cloud era and for large parts replacing the serverful computing.

References

Adzic, G. & Chatley, R., 2017. Serverless computing: Economic and architectural impact. In ACM Press, pp. 884–889. Available at: http://dl.acm.org/citation.cfm?doid=3106237.3117767 [Accessed April 17, 2018].

AWS, 2019. Edge Computing| CDN, Global Serverless Code, Distribution | AWS Lambda@Edge. *Amazon Web Services, Inc.* Available at: https://aws.amazon.com/lambda/edge/ [Accessed February 9, 2019].

AWS, A., 2014. AWS Release Notes. *Amazon Web Services, Inc.* Available at: https://aws.amazon.com/releasenotes/release-aws-lambda-on-2014-11-13/ [Accessed April 17, 2018].

Chen, L., 2018. Microservices: Architecting for Continuous Delivery and DevOps. In IEEE International Conference on Software Architecture. Seattle, USA.

CNCF, 2019. CNCF Cloud Native Definition v1.0. Available at: https://github.com/cncf/toc [Accessed February 20, 2019].

CNCF, 2018. CNCF Serverless Whitepaper v1.0. Available at: https://github.com/cncf/wg-serverless/tree/master/whitepaper [Accessed April 17, 2018].

Cockcroft, A., 2017. Evolution of business logic from monoliths through microservices, to functions. *A Cloud Guru*. Available at: https://read.acloud.guru/evolution-of-business-logic-from-monoliths-through-microservices-to-functions-ff464b95a44d#.bdg75xgb2 [Accessed February 21, 2019].

Conway, M.E., 2010. How Do Committees Invent? Available at: http://www.melconway.com/Home/Committees_Paper.html [Accessed May 1, 2018].

Conway, M.E., 1968. How Do Committees Invent? *Datamation magazine*, 14(4), pp.28–31.

Cui, Y., 2018. I'm afraid you're thinking about AWS Lambda cold starts all wrong. *theburningmonk.com*. Available at: https://theburningmonk.com/2018/01/im-afraid-youre-thinking-about-aws-lambda-cold-starts-all-wrong/ [Accessed August 31, 2018].

Dooley, D., COO & Theorem, D., 2019. Serverless, shadow APIs and Denial of Wallet attacks. *Help Net Security*. Available at: https://www.helpnetsecurity.com/2019/03/29/serverless-challenges/ [Accessed April 24, 2019].

Dragoni, N. et al., 2017. Microservices: Yesterday, Today, and Tomorrow. In M. Mazzara & B. Meyer, eds. *Present and Ulterior Software Engineering*. Cham: Springer International Publishing, pp. 195–216. Available at: http://link.springer.com/10.1007/978-3-319-67425-4_12 [Accessed April 17, 2018].

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

Evans, E., 2004. *Domain-driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley Professional.

Fowler, M., 2014. BoundedContext. *martinfowler.com*. Available at: https://martinfowler.com/bliki/

BoundedContext.html [Accessed May 10, 2018].

Gray, J., 2006. A Conversation with Werner Vogels. *Queue*, 4(4), pp.14:14–14:22. Available at: http://doi.acm.org/10.1145/1142055.1142065 [Accessed May 10, 2018].

Hellerstein, J.M. et al., 2018. Serverless Computing: One Step Forward, Two Steps Back. Available at: http://arxiv.org/abs/1812.03651 [Accessed December 27, 2018].

Hendrickson, S. et al., 2016. Serverless Computation with OpenLambda., p.7.

Jonas, E. et al., 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. Available at: http://arxiv.org/abs/1902.03383 [Accessed February 18, 2019].

Langit, L., 2017. *Serverless - reality or BS - notes from the trenches*, Available at: https://www.youtube.com/watch?v=PgZ2dxnj734 [Accessed April 17, 2018].

Lewis, J. & Fowler, M., 2014. Microservices. *martinfowler.com*. Available at: https://martinfowler.com/articles/microservices.html [Accessed April 17, 2018].

Lynn, T. et al., 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)*. 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). pp. 162–169.

MacCormack, A., Baldwin, C. & Rusnak, J., 2012. Exploring the duality between product and organizational architectures: A test of the “mirroring” hypothesis. *Research Policy*, 41(8), pp.1309–1324. Available at: http://linkinghub.elsevier.com/retrieve/pii/S0048733312001205 [Accessed May 1, 2018].

Mogosanu, M., 2012. DDD - The Bounded Context Explained. *Sapiens Works*. Available at: http://blog.sapiensworks.com/post/2012/04/17/DDD-The-Bounded-Context-Explained.aspx [Accessed May 10, 2018].

Nadareishvili, I. et al., 2016. *Microservice Architecture: Aligning Principles, Practices, and Culture* 1st ed., O’Reilly Media, Inc.

Narayan, S., 2018. Products Over Projects. *martinfowler.com*. Available at: https://martinfowler.com/articles/products-over-projects.html [Accessed February 22, 2019].

Newman, S., 2015. *Building Microservices: Designing Fine-Grained Systems* First Edition., Beijing Sebastopol, CA: O’Reilly Media.

Newman, S., 2017. *Confusion In The Land Of The Serverless - GOTO 2017*, Berlin, Germany. Available at: https://www.youtube.com/watch?v=aZlrv-0PE_c&amp=&t=240s [Accessed September 6, 2018].

Patrizio, A., 2018. What is cloud-native? The modern way to develop applications. *InfoWorld*. Available at: https://www.infoworld.com/article/3281046/cloud-computing/what-is-cloud-native-the-modern-way-to-develop-software.html [Accessed February 20, 2019].

Pollard, T. et al., 2016. *Template for writing a PhD thesis in Markdown*, Zenodo. Available at: https://zenodo.org/record/58490#.WtWV7tNuaRc [Accessed April 17, 2018].

Roberts, M., 2016. Serverless Architectures. *martinfowler.com*. Available at: https://martinfowler.com/articles/serverless.html [Accessed April 17, 2018].

Roberts, M. & Chapin, J., 2017. *What Is Serverless?* First Edition., O'Reilly Media. Available at: https://www.oreilly.com/programming/free/what-is-serverless.csp [Accessed August 27, 2018].

Stigler, M., 2018. Understanding Serverless Computing. In M. Stigler, ed. *Beginning Serverless Computing: Developing with Amazon Web Services, Microsoft Azure, and Google Cloud*. Berkeley, CA: Apress, pp. 1–14. Available at: https://doi.org/10.1007/978-1-4842-3084-8_1 [Accessed February 18, 2019].

Toffetti, G. et al., 2017. Self-managing cloud-native applications: Design, implementation, and experience. *Future Generation Computer Systems*, 72, pp.165–179. Available at: https://linkinghub.elsevier.com/retrieve/pii/S0167739X16302977 [Accessed February 20, 2019].

Villamizar, M. et al., 2016. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). pp. 179–182.

Vojta, R., 2016. AWS journey — API Gateway & Lambda & VPC performance. *Zrzka's adventures*. Available at: https://www.robertvojta.com/aws-journey-api-gateway-lambda-vpc-performance/ [Accessed April 18, 2018].

Wagner, T., 2018. ServerlessConf 2018 Keynote - Debunking Serverless Myths. Available at: https://www.slideshare.net/TimWagner/serverlessconf-2018-keynote-debunking-serverless-myths [Accessed February 19, 2019].

Wang, A. & Tonse, S., 2013. Announcing Ribbon: Tying the Netflix Mid-Tier Services Together. *Medium*. Available at: https://medium.com/netflix-techblog/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62 [Accessed April 21, 2018].

Xiao, Z., Wijegunaratne, I. & Qiang, X., 2016. Reflections on SOA and Microservices. In *2016 4th International Conference on Enterprise Systems (ES)*. 2016 4th International Conference on Enterprise Systems (ES). pp. 60–67.

Yermakov, A., 2018a. Microservices as a self sufficient concept. Available at: https://yermakov.net/microservices-as-a-self-sufficient-concept/ [Accessed April 21, 2018].

Yermakov, A., 2018b. Microservices without fundamentals. Available at: https://yermakov.net/microservices-without-fundamentals/ [Accessed April 21, 2018].