

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Riku Aapakallio-Autio

From Elastic Beanstalk to Lambda: A comparative case study on the AWS tools

Master's Thesis
Espoo, July 28, 2021

Supervisor: Professor Antti Ylä-Jääski, Aalto University
Advisor: Peter Karjalainen M.Sc. (Tech.)

Aalto University
 School of Science

 Master's Programme in Computer, Communication and
 Information Sciences

 ABSTRACT OF
 MASTER'S THESIS

Author:	Riku Aapakallio-Autio	
Title:	From Elastic Beanstalk to Lambda: A comparative case study on the AWS tools	
Date:	July 28, 2021	Pages: 55
Major:	Computer Science	Code: SCI3042
Supervisor:	Professor Antti Ylä-Jääski	
Advisor:	Peter Karjalainen M.Sc. (Tech.)	
	<p>Cloud computing is a rapidly growing paradigm that offers out-sourcing hosting capabilities to a cloud provider. Due to its rapid development, it requires some effort to keep up with the current trends. In this thesis we implement a contemporary serverless REST API for the Iltalehti news media through transformation from a more traditional web server version.</p> <p>Nowadays, it is almost essential to be proficient in the cloud to be able to provide web services efficiently. Thus, we first examined cloud computing and serverless technologies closely and identified their capabilities and limitations. Recently, serverless computing and the microservices model have taken over monolith applications to provide more granular testing and development.</p> <p>Secondly, we implemented the actual transformation and compared this new implementation to our previous, more traditional approach. The main finding of this study was that it is quite simple to lift and shift existing services to serverless infrastructure. Due to the fact that Elastic Beanstalk and AWS Lambda are two different types of service, their pricing models are quite different. In our use case, the proposed solution would reduce expected costs by 99 %.</p>	
Keywords:	serverless, cloud computing, aws, microservices	
Language:	English	

Tekijä:	Riku Aapakallio-Autio		
Työn nimi:	Elastic Beanstalk ja Lambda: Vertaileva tutkimus AWS-työkaluista		
Päiväys:	28. heinäkuuta 2021	Sivumäärä:	55
Pääaine:	Tietotekniikka	Koodi:	SCI3042
Valvoja:	Professori Antti Ylä-Jääski		
Ohjaaja:	Diplomi-insinööri Peter Karjalainen		
<p>Pilvilaskenta on nopeasti kasvava paradigma, joka tarjoaa verkkopalveluiden tarjoajalle mahdollisuuden luopua omasta fyysisestä infrastruktuurista ja siirtää vastuun isännöinnistä pilvipalveluntarjoajalle. Nopean kehityksensä myötä, vaatii se alan toimijoita pitämään aktiivisesti yllä tietämystään nykyhetken trendeistä. Tässä diplomityössä konvertoimme perinteisen palvelinmallin nykyaikaiseksi serverless REST-rajapinnaksi Iltalehdelle.</p> <p>Verkkopalveluiden tarjoaminen tehokkaasti vaatii nykypäivänä pätevyyttä pilvipalveluiden infrastruktuureissa. Aluksi tutustuimme tarkasti pilvipalveluihin ja serverless-teknologiaan, sekä niiden ominaisuuksiin ja rajoituksiin. Viime vuosina serverless-teknologia ja mikropalveluajattelu ovat yhdessä korvanneet suurempia yhtenäisiä järjestelmäkokonaisuuksia pienempinä paloina toteutettuihin sovelluksiin. Hienojakoisuus helpottaa sovelluksen testaamista ja kehittämistä.</p> <p>Seuraavaksi toteutimme sovelluksen transformaation ja esittelimme käyttämämme mallit, sekä vertasimme toteutuksemme suorituskykyä ja hintaa aikaisempaan, perinteisempään ratkaisuun. Huomasimme työssä, että sovelluksen siirtäminen Elastic Beanstalk -alustalta AWS Lambdaan on hyvin toteutettavissa. Nämä kaksi erilaista pilvipalvelumallia ovat kuitenkin perustaltaan hyvin erilaisia ja niiden hinnoittelu eroaa suuresti. Tutkimuksessa kohteena olleen sovelluksen tapauksessa ehdottamamme ratkaisu laskisi kustannuksia noin 99 prosentilla.</p>			
Asiasanat:	serverless, pilvipalvelut, pilvilaskenta, aws, mikropalvelut		
Kieli:	Englanti		

Acknowledgements

I wish to thank my supervisor Antti Ylä-Jääski for taking the time to supervise my thesis. I would also like to thank my advisor and hands-on teacher Peter Karjalainen for valuable help and input before and during the writing of this thesis. I want to also mention my colleague Joni Väyrynen for constructive feedback. Most of all, I would like to thank my wife Minna Aapakallio-Autio for always believing in me and pushing me forward.

Espoo, July 28, 2021

Riku Aapakallio-Autio

Abbreviations and Acronyms

AMI	Amazon Machine Image
API	Application Programming Interface
AWS	Amazon Web Services
BaaS	Backend as a Service
CDN	Content Delivery Network
CI/CD	Continuous Integration / Continuous Delivery
DDoS	Distributed Denial of Service
EC2	Elastic Compute Cloud
FaaS	Function as a Service
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
REST	Representational State Transfer
SaaS	Software as a Service
S3	Simple Storage Service
VPC	Virtual Private Cloud

Contents

Abbreviations and Acronyms	5
1 Introduction	8
1.1 Problem Statement	8
1.2 Research Questions	9
1.3 Outline	10
2 Background	11
2.1 Cloud Computing	11
2.2 Monoliths & Microservices	13
2.3 Serverless	14
2.4 AWS Products	16
2.5 REST	19
3 Environment	21
3.1 Organization	21
3.2 Technologies	22
3.3 Motivation	23
3.4 The Application	25
4 Implementation	26
4.1 Objectives	26
4.2 Runtime	27
4.3 Infrastructure	31
4.4 Deployment	35
5 Results and analysis	38
5.1 Actualized runtime and cost	38
5.2 Performance	39
5.3 Deployment & Logging	43
5.4 Reliability & Transferability	44

6	Discussion	47
6.1	RQ1: What is serverless and where is it useful?	47
6.2	RQ2: How to transform a PaaS to serverless?	48
6.3	RQ3: How do the two different methods compare to one another?	48
7	Conclusions	50

Chapter 1

Introduction

Serverless cloud computing is becoming a staple in modern web services, conjointly with the transition towards microservices from classic monolithic web applications. The cloud computing paradigm promises rapid elasticity and provisioning at its core definition [1]. However, the cloud is evolving rapidly, and the recent development has opened even more granular provisioning compared to just a few years prior. Thus, we should not let conformity keep us stuck in the past in a fast-moving industry. Keeping up with the current trends further lessens the risk of provisioning too much or too little resources to an application, thus maximizing reliability whilst keeping costs to a minimum.

Amazon Web Services (AWS) offers a multitude of cloud services. Among these, are the Elastic Beanstalk and AWS Lambda. Elastic Beanstalk offers a platform, on which the user can run arbitrary applications on virtual machines with highly elastic provisioning. This presents the modern cloud alternative to local physical server machines. Whereas Lambda is a serverless computing service, that offers a much simpler platform where a user can execute event-driven function code. The two services compare like apples to oranges, yet similar types of applications are being ran on both.

1.1 Problem Statement

The cloud is evolving quickly and the evolution is being led by the industry [2]. This is due to the fact how it allows streamlining hosting related costs by utilizing the granularity of cloud computing. Consequently, current infrastructural choices might not stay optimal for long periods due to new options and functionalities being made available constantly. Not keeping up with the current trends might result in over-provisioning cloud computing resources

and therefore excessive costs. Along with these, there is also a potential threat for technical debt and falling out of the loop. If, for example, the future of the REST API lies in serverless containers, the surrounding support will most likely grow and simultaneously decrease for other solutions.

In this thesis, we will focus on the infrastructure of hosting a REST API web service in the environment of the news media Iltalehti, one of the two biggest online newspapers in Finland [3]. At Iltalehti, we are used to hosting different kinds of web applications on the Elastic Beanstalk Platform-as-a-Service. However, serverless computing is emerging as a new compelling paradigm for the industry, due to the transition towards microservices [4]. Event-driven by nature [5], serverless computing also maps to the REST API use case quite linearly.

Currently, we are hosting some REST API applications with irregular load on the Elastic Beanstalk. This means that for the most part, they are sitting idly and waiting for requests. Meanwhile, cloud computing has evolved to the point that these applications could benefit from the serverless paradigm and only be running for the time they are actually executing program code. Whereas the Elastic Beanstalk is rather difficult to provision just right, Lambda offers exactly that. Consequently, we are interested in adopting this new type of infrastructure where applicable. However, it is essential to bear in mind that these two types of infrastructure are innately different and we might face some new challenges with this adoption. Elastic Beanstalk offers a platform for virtual machine images imitating the classic physical infrastructure whereas Lambda exposes simple function containers.

The difference in Elastic Beanstalk and the Lambda produces a need for us to take a closer look on what this means for the end product. We are interested in the maximum granularity when it comes to costs and provisioning, as offered by the Lambda. Yet we do not want to increase response time noticeably or compromise reliability, of the levels that we are used to with the Elastic Beanstalk. With the adoption of our implementation, our baseline cost for a service will most likely reduce significantly. However, we also need to take into account how to manage sudden spikes in load.

1.2 Research Questions

In this thesis, we will present some background for cloud computing and examine the serverless computing paradigm in detail. After laying the foundation, we will implement the serverless infrastructure and fit our existing application in it. Finally, we will perform a comparison on the two methods. We summarize the research questions as:

- RQ1: What is serverless and where is it useful?
- RQ2: How to transform a PaaS to serverless?
- RQ3: How do the two different methods compare to one another?

1.3 Outline

The remainder of the thesis is structured as follows. Chapter 2 explains the background and the underlying concepts that we will be using in our implementation and discussing in this thesis. Chapter 3 introduces the working environment and describes the background of the Iltalehti news media. The motivation to this study for our environment is also discussed. Chapter 4 explains our chosen function runtime and presents our proposed implementation. In Chapter 5 the results are analyzed and compared to the previous implementations along with reliability and possibility to transfer the implementation to similar applications. In Chapter 6 we take a retrospective look at our research questions and summarize answers to them. Finally, in Chapter 7 we sum up the thesis.

Chapter 2

Background

This chapter provides background for the concepts to be discussed. Section 2.1 describes cloud computing to create a foundation for the technologies discussed in this thesis. Section 2.2 explains the traditional monolithic applications and their more contemporary counterpart, microservices. To add to the definitions of the cloud, Section 2.3 introduces serverless computing. Section 2.4 introduces the different cloud products provided by AWS referenced in this thesis. Finally, Section 2.5 explains the definition of Representational State Transfer, since it is the type of web application we will be working with in this thesis.

2.1 Cloud Computing

Traditionally, computing took place on physical, local machines and, in the case of online web services, in server rooms consisting of multiple computers. As early as 1961, computer scientist John McCarthy originated the idea that the computer could become the basis of a new and important industry, referring to the sharing of its computational resources [6]. This is considered to be the first step in the history of computation towards cloud computing. The actualization of this underlying concept did not take place until early 2000s.

In 2009, Armbrust et al. conducted the "Berkeley View of Cloud Computing" [7] to explain the rising popularity behind this paradigm. They identified and named six features for the cloud:

- Seemingly infinite computing resources on demand
- Incremental cost, as opposed to paying for physical machines up front
- Allow short-term use of computing resources

- Significantly sized data centers allow scaling with reduced costs
- Increased utilization and simpler operation via virtualization of resources
- Maximizing utilization of hardware due to organizations sharing data center resources

They viewed the cloud as being something that has been dreamed about for long and that is finally emerging. It seemed highly compelling for the cloud providers side due to the possibility of building very large shared data centers at low cost sites with on demand rental computing power. Meanwhile, for the cloud user, it opened up new possibilities. For example, start ups would have an easier time at getting up and running without having to build their own data center.

Amazon Web Services (AWS) originates from a need to create internal Infrastructure as a Service (IaaS) to speed up own development, which soon scaled to commercial use. In August 2006, AWS introduced Elastic Compute Cloud (EC2) [8] and in 2010 Amazon had transferred their retail web services into the AWS cloud [9]. Around this time, Google and Microsoft had joined the competition with Google Cloud and Azure, respectively. In 2020 the cloud industry was a 129 billion dollar industry and was led by AWS with a 32-34 % share of the market followed by Microsoft Azure with a 20 % share [10]. Furthermore, Eismann et al. [11] performed a survey study of serverless use cases and AWS was chosen by 80% of the use cases presented.

The National Institute of Standards and Technology provides a definition for cloud computing in the form of essential characteristics: [1]

- On-demand self-service: the ability to provision computing capabilities when needed.
- Broad network access: capabilities are widely available on different platforms.
- Resource pooling: physical and virtual resources of the provider are assigned and reassigned on-demand.
- Rapid elasticity: capabilities are elastic in the sense that they can be scaled quickly according to demand, even automatically.
- Measured service: strict monitoring and control to provide transparency between provider and customer.

The standard also introduces three service models commonly used in cloud computing: [1]

- Software as a Service (SaaS): the provider offers software solutions the consumer can use over the web on different interfaces such as a web-based email application. The consumer is not responsible for any of the underlying cloud infrastructure.
- Platform as a Service (PaaS): the consumer can deploy applications of their own in provided cloud platforms with no control or responsibility of the rest of the infrastructure.
- Infrastructure as a Service (IaaS): the capability of deploying arbitrary software such as operating systems on the cloud platform of the provider.

The first two mentions, SaaS and PaaS, leave quite a substantial gap in between them. SaaS usually depicts a ready-made service for the consumer who then becomes the end user. PaaS, on the other hand, provides an instance where the consumer can run their own custom applications on the cloud and requires the consumer to have knowledge of building software.

2.2 Monoliths & Microservices

Traditionally, applications have been built as significantly sized monoliths. A monolithic application stands for an application with a large, shared, single codebase that serves a multitude of services with different types of interfaces, such as HTML pages and REST API end-points [12]. However successful these monolithic applications can prove to be, multiple problems arise for development and scaling [13]. Whenever a developer in a development team makes changes to a single part of the monolith, they must be certain of the integrity of the whole system. Even then, a restart for all the services provided by the system might be necessary. Furthermore, the different services in the monolith require distinct levels of computing power and thus, correct provisioning of cloud resources proves to be difficult.

Along with the cloud, the term microservices also gained popularity in the recent decade. A microservice is a single application that can be used, tested and scaled independently [14]. It has a single purpose and responsibility and is simple to understand. For example, a contemporary web service might consist of a front-end application to depict how the web site looks like, a back-end service for authentication and another back-end service to serve

content of some kind. Traditionally, this whole web service could have been built as one big application but with the microservices way of thinking, it could be three or more separate applications. In this manner, it is easier to deploy the separate parts of an application and test them.

However, microservices generate certain new challenges for development as described by Jamshidi et al. [15]. As the popularity of microservices rise, they might be used in situations where it would be quicker or more cost-effective to choose the monolithic approach. Level of granularity opposes another challenge. Although the term proposes as fine-grained modularity as possible, developers have different visions on how small the modules should be in practice. Additionally, companies often have several development teams working separately on the same microservices. Consequently, the flow of information requires special attention, oftentimes in the form of concrete actions such as cross-team discussions.

2.3 Serverless

Van Eyk et al. [5] adds serverless computing, also called Function as a Service (FaaS) to the definitions of the cloud described in Section 2.1. They provide three main characteristics to define serverless architectures:

- Granular billing: the consumer is charged only for the actual run time of the application itself.
- No operational logic: the cloud provider has almost complete responsibility of the underlying infrastructure including auto-scaling and resource management.
- Event-driven: the infrastructure deploys short-living serverless applications in response to events defined by the consumer.

Looking at these characteristics, FaaS could be placed in the previous list of service models in the gap between PaaS and SaaS. Serverless computing also maps naturally to a microservice software architecture due to its maximum level of granularity.

In 2019, Jonas et al. added on to the previously mentioned "Berkeley View of Cloud Computing" [7] by conducting the "Berkeley View on Serverless Computing" [16]. They believe that the serverless paradigm is more attractive for new customers than previous infrastructure implementations and is likely to draw in more users for the cloud providers. The attractiveness comes from the simplification, since with serverless, users can now

deploy from as little as a single function all the way up to whole applications. Additionally, many security concerns are also outsourced to the cloud provider due to the high level of programming abstraction and fine-grained isolation. They also predict that eventually, serverless options will always cost equally as much or less than the so called serverful alternative.

Backend as a service, or BaaS, is a model that became common with mobile applications. The idea behind it is to replace all server-side components with ready-made applications and interfaces, bearing similarities to SaaS [17]. Google Firebase is one example, where a mobile application developer can utilize backend services from the cloud provider without having to code the components by hand. These services include authentication, user management and simple database access. Amazon Cognito and Auth0 are also similar authentication services provided as BaaS. Jonas et al. [16] define serverless as FaaS + BaaS. This stands due to their definition of serverless: it scales automatically and is billed purely based on usage. They also predict that the future of serverful computing is to facilitate BaaS services that are difficult to build on serverless infrastructure, such as queue structures and various online transaction systems.

Baldini et al. [18] mention the stateless nature of serverless computing. This means that a serverless function is also naturally unaware of the current state of actions. Upon triggering the function, it is woken up and if it requires information that is not passed to it via the parameters of the triggering event, this information needs to be fetched from outside storage. After the execution is finished, the state and memory will be wiped, and must be saved elsewhere if required at a later time. This fact highlights the simple nature of serverless functions, meanwhile raising the need for surrounding ecosystem. In the case of Lambda, it often uses services like Simple Storage Service (S3) for state storage and API Gateway to create API end-points [19]. Simplicity comes at the expense of evident vendor lock-in when additional services are required, typically from the same cloud provider.

The key driver of serverless computing is the promise of more value than other cloud operations [2]. This equates to equal or better performance meanwhile reducing costs. Multiple studies have discovered cost savings ranging from 60% to 90% by adapting to serverless computing instead of other types of cloud infrastructure [13, 20]. Consequently, the aspect of lower costs has led industry, rather than academia, to be the driving force in the adoption and development of serverless computing. Traditionally, other cloud infrastructure such as PaaS products have suffered from more coarse-grained billing due to under- and over-provisioning issues. Together with the fine-grained costs, serverless computing abstracts almost all customer-end operational concern out of traditional cloud computing [18].

However, given the manner in which serverless functions are executed in their own containers, a concern raises for so called cold starts [21]. Cold starts occur when the serverless function has not been called for around 15 minutes and the cloud provider must create a new container for the function. Yet, as serverless computing is growing fast, the serverless platforms are also evolving rather quickly to tackle this issue. According to the analysis performed by Muller et al. [21] the additional time due to a cold-start can be up to 17 % of the round-trip-time. The cloud provider has idle virtual machines with the run times readily installed to mitigate these cold starts. Consequently, choosing a popular run time might lead to a greater pool of idle machines and provide the consumer of the serverless platform with lesser cold start times.

Although rapidly evolving, serverless is not quite complete yet. Nupponen et al. [22] list some bad practices and open issues with it. One big open issue is the lack of understanding the event-driven paradigm. This is common with developers that are used to traditional infrastructure and technology and something we aim to help bridge in this thesis. Another open issue concerns integration testing and system-level testing, which becomes harder with the high level of abstraction in serverless functions. Identified bad practices include:

- Asynchronous calls: increase complexity
- Functions calling other functions: hinder debugging
- Shared code between functions: changes might break functions
- Too many libraries: increase space taken and warm-up time

2.4 AWS Products

In this section, we will cover the basics and pricing of the AWS cloud products that are relevant to this thesis. These include Elastic Compute Cloud, Elastic Beanstalk, Lambda, API Gateway, CloudFormation, Simple Storage Service and CloudFront.

Elastic Compute Cloud (EC2) is a lower level cloud service that allows the consumer to rent virtual computer partitions where any custom software can be deployed. AWS provides the consumer with a partition of a physical computer running an operating system of their liking. This happens on a so called Amazon Machine Image (AMI) that configures a virtual server instance. The amount of these instances can be scaled easily, hence the word elastic.

Elastic Beanstalk is a commercial PaaS offered by AWS. It provides the consumer with a platform that runs on one or more EC2 virtual server instances. Elastic Beanstalk adds orchestration and scalability in the form of autoscaling and load-balancing on top of the low level EC2 service. Additionally, it writes logs and metrics to the CloudWatch logging service. The consumer can configure the minimum and maximum amount of EC2 instances to be used and when the workload passes a set threshold, the amount of virtual server instances is scaled up and down automatically. This infrastructure provides great availability for all kinds of online services but it comes with a price. For a service that has less constant workloads it seems unnecessary to pay a constant price for simply standing by. In business, cutting costs is key. Therefore we should look at Lambda due to the aforementioned granular billing [5].

Lambda is the commercial name for the serverless compute service, or FaaS, provided by AWS. Whereas the combination of Elastic Beanstalk and EC2 provide a platform, Lambda cuts out the operational logic and provides fine-grained billing where the consumer pays only for the duration of the run time of the code. This Lambda code can then be executed in response to an event or regularly at certain intervals. AWS have set certain upper limits for the usage of Lambda. For example, the run time of Lambda code times out after 15 minutes of execution and the invocation payload can contain a body of 6 MB at maximum [23]. The workflow of Lambda is fairly straightforward. The service creates a container that runs application code provided by the customer, usually stored in an S3 bucket as a zip package. The application code must contain a function that is defined as the entry point, called the lambda handler. This handler function receives information concerning the event that triggered this application code execution and defines how to proceed with this information.

AWS describes the EC2 instances sized t3.medium to possess 2.5 GHz Intel Scalable Processors with 2 vCPU's [24]. In the study conducted by Muller et al. [21] the authors reverse-engineered Lambda by measurement functions that logged information concerning the virtual machines. Out of over 300 deployed functions, two functions ran on instances with 3 GHz of processing power. All the rest had the same specifications as the ones listed for t3.medium.

Pricing of the Lambda are shown in Table 2.1. When calculating the cost of an application, the memory required by it must be taken into account. Lambda supports memory allocation of 128 MB onwards up to 10,240 MB in one megabyte increments [25].

API Gateway is an AWS tool to create an API interface for another service. This tool can be used to create a simple API layer with some end-

	Price
Requests	0.20 \$ per 1M requests
Duration	0.0000166667 \$ per GB-second

Table 2.1: Lambda pricing (Ireland)

points that can be accessed from other services. These end points trigger a certain event that requests action from the Lambda and it can validate the incoming requests without additional cost. API Gateway pricing per requests can be seen in Table 2.2 [26].

Number of requests	Price per million
First 333 million	3.50 \$
Next 667 million	3.19 \$
Next 19 billion	2.71 \$
Over 20 billion	1.72 \$

Table 2.2: API Gateway pricing (Ireland)

Simple Storage Service (S3) is the data storage solution of AWS. It is a highly scalable and reliable system that can be used to store personal data, easily distribute content to the public or as a storage component in a web service [27]. In the previous section, we mentioned that S3 can be used to provide state information to a Lambda function [19]. However, often, the zip packed Lambda function code itself is also hosted in an S3 bucket. An exception to this is that some short functions can be placed inline straight into a CloudFormation stack configuration template file. In most, smaller storage solutions, S3 pricing is rather negligible. For the first 50TB of data stored in an S3 bucket, the cost of one gigabyte is 0.023\$ monthly [28].

CloudFormation is a tool that combines these pieces together in templates that can be used to deploy resources needed easily and repeatably, as well as managed in version control systems. There is no extra cost for using CloudFormation, the only cost involved is of the services deployed using it [29].

In the latter parts we will also mention CloudFront. CloudFront is a content delivery network (CDN) that can be used to cache the data at edge regions. In practice, this means that if our AWS region is in Ireland and an user accesses our website from Finland, the response they receive is stored in an edge cache at Helsinki for a set amount of time, such as one minute, for all

other requests. When the next request in a similar location is fired, the CDN delivers it quickly from the edge cache without disturbing the application. We will not be looking at the price of CloudFront in this thesis, as we will not be using it in our suggested implementation. It is mentioned here for the purpose of possible protection against Distributed Denial of Service (DDoS) attacks.

2.5 REST

Representational State Transfer (REST) is a term formulated in 2000 by Roy Fielding in his dissertation [30]. Fielding describes REST as a hybrid architectural style for distributed hypermedia systems. Nowadays, it is a common style for back-end services and utilized by web applications widely. It is built around six architectural constraints:

- **Client-Server:** The user interface is separated from the data storage concerns. This allows portability for the client such as desktop and mobile user interfaces. Furthermore, these components can then evolve individually.
- **Statelessness:** The server does not store any state context information. Instead, it should reside client-side and be passed to the server with requests when needed. The increased simplicity helps with visibility, reliability and scalability.
- **Cache:** Data returned by a response to a request should be labeled as cacheable or non-cacheable. Thus, the client knows whether the received data can be reused instead of sending excessive requests.
- **Uniform interface:** All components should share a uniform standardized interface. This provides simplified and highly visible interactions between the client and the server. The trade-off is high granularity with the data. Standardization might result in excessive data in a request, for example.
- **Layered system:** The system can consist of hierarchical layers that are encapsulated within their layer. This provides safety and bounds the system complexity by restricting the knowledge of a component.
- **Code-on-Demand:** The server can extend the client side code on demand with downloadable scripts or applets.

In the upcoming chapters, we will be describing our implementation, a transformation job from the Elastic Beanstalk to the serverless Lambda. The application that will undergo the transformation will be a REST API. Worth noting is that the REST constraints show cohesion with both the serverless paradigm and the microservices way of thinking. Splitting the client and server into separate components is typical to microservices and statelessness is also core to the concept of serverless [18].

Chapter 3

Environment

This chapter introduces the environment of Iltalehti and the choices of technological methods used to implement the environment. First, Section 3.1 briefly describes the organization and history of Iltalehti. Section 3.2 presents the choices of technologies used in the environment and Section 3.3 explains the motivation from the organization for this study and experiment.

3.1 Organization

Iltalehti is a leading Finnish news media and a multifaceted lifestyle media owned by Alma Media Oyj. It is among the two largest online news media sites in Finland [3]. According to the Amazon powered Alexa, it is among the top 10 most popular websites in Finland [31].

Iltalehti has been a printed tabloid since 1980 and the first version of its web site was made public in 1995. Originally, the online version of Iltalehti consisted of static HTML files that were served from a self managed, on premises datacenter. Earlier versions of the servers included a Sun Solaris server and slightly later, rack server computers. In-between the physical computers and the current cloud provider were VMware machines that allowed us to imitate virtual computers with operating systems in an early interpretation of the cloud. As of the summer of 2021, all of the hosting has been moved under AWS.

The current responsive web site has been operative since May 2018 after a significant rework introducing contemporary technologies. Covered in further detail in the next section, these technologies are more dynamic rather than static such as the original website implementation. Naturally, as the cloud market emerged, along with the compatibility with the developing web technologies, Iltalehti moved into the cloud. There is far less risk of

over-provisioning and an opportunity to significant cost savings over the traditional self hosted servers.

3.2 Technologies

The Iltalehti environment is built of microservices. These include the front-end web application and different API services to serve content such as articles, pictures, statistics, discount vouchers and betting odds, to name a few. These separate applications are using contemporary web technologies including React, TypeScript in the front-end and Ruby, Elixir and Go in the back-end. For the purpose of development and ease of testing and safe deployment, we have two identical, separate environments. One for development, and the other for production. Both of these environments consist of all the same microservices with access to their respective databases. The use of these two mirrored environments help us with testing and demonstrating new implementations before actually deploying changes to production. All of these web services are hosted on the AWS cloud platform.

Commonly, the separate microservices are deployed to the AWS cloud using CloudFormation and similar YAML base templates that are known to provide safe and reliably working Elastic Beanstalk environments. However, the PaaS environment created by these templates might be extensive for certain services and use cases, where we might take a look at using a Lambda function instead. Currently, we are not deeply familiar in using Lambda to host entire applications.

It is worth mentioning Elastic Beanstalk and Lambda are highly dissimilar in nature and thus difficult to compare. They are different types of infrastructure that serve different purposes, at least originally. Recently, however, using the serverless technologies, such as Lambda, has become the epitome of microservices. Consequently, even traditional HTTP REST API web servers have been implemented using event-driven serverless solutions [2]. Microservices and serverless computing share a similar fine-grained view of dividing applications into small-scale chunks. As we are already utilizing microservices, it is easy to look at the amount of traffic our different applications handle and pick one with suitable load to try our transformation for the scope of this thesis. A service that processes heavy data loads often such as images or videos would not be our first choice, due to the limitations of Lambda, such as the maximum payload size [23].

We have identified a number of our own back end API services that are not under constant workload but have been deployed to the Elastic Beanstalk using the common, safe deploy pipeline and templates. We chose to proceed

in this study with one that is in response of sending push notification content to Twitter and mobile applications. Given our isolated development and production environments, this application and its workload is simple to test on our development environment, since both environments handle the same amount of traffic. Obviously enough, the one in production has more mobile devices to send push notifications to, but that is not a concern for this service, because the actual sending of concrete single notifications is done outside it's scope.

3.3 Motivation

The first two characteristics of serverless technologies listed in Section 2.1 are granular billing and the need for no operational logic. In our case, these two are of great interest. From a business stand point, there is financial motivation in the form of savings to be made by looking at using Lambda instead of our current solutions and providing proper analysis on the differences in performance and cost. The current architecture provides high availability and it is in our interest not to risk increasing potential down time or response times but we hypothesize there is potential upside in Lambda when it comes to simplicity and cost.

As mentioned previously, we will target our push content API as the service to be converted. It is currently utilizing two EC2 instances sized t3.medium and has an average work load of approximately 10,000 requests per month and an average response time of 330 milliseconds. This workload is relatively small and inconsistent for the current architecture it is running on and will provide an interesting comparison between the new proposed method and the current one. The current on-demand price for a t3.medium EC2 instance is 33.228 \$ per 30 days [32], for two instances this equates to 66.456 \$. This is an on-demand price that varies to some extent and can be lowered by using reservations and so called spot instances. We are utilizing both reservations and spot instances to decrease the price. The price of spot instances varies greatly and for further simplicity, we will perform our comparisons using the listed on-demand prices.

To begin our comparison, we need to perform some calculations to reach an approximation for the costs of the proposed solution. Lambda is priced at the Ireland region at 0.00001667 \$ per GB-s and 0.20 \$ per million requests [25]. It comes with 400,000 GB-s of free compute time and one million free requests monthly but we will not be taking these into account, as they are consumed across all Lambda functions in a single account. Lambda costs per the amount of requests monthly summed with the compute time charges. As

for the requests, we will stay safely under one million requests and thus can use the given 0.20 \$ for our monthly request costs. The price estimate calculation for Lambda compute charges is as follows:

$$MonthlyCost = ComputeSeconds * ComputeGB * GBsPrice$$

In our case, we will gain a rough estimate of our compute seconds using the aforementioned average response time of 330 milliseconds. This is not accurate by any means, as this is the response time of an application that is already actively running in a dedicated Elastic Beanstalk. However, in our proposed solution, the application would first be started in a serverless container before serving our request. Nevertheless, we will obtain an estimate and compare these calculations to actualized run times later on in Chapter 5. We can compute the approximation with:

$$ComputeSeconds = 10,000 (reqs) * 0.330 s = 3,300 s$$

And we will be using the smallest Lambda size of 128 megabytes and thus our compute gigabytes will be

$$ComputeGB = \frac{128}{1,024} GB$$

giving us a monthly cost of

$$MonthlyCost = 3,300 s * \frac{128}{1,024} GB * 0.00001667 \frac{\$}{GB * s} = 0.0068 \$$$

Adding this to the one million request baseline, our monthly cost will be at 0.21 \$.

Lastly, we also need to take a look at AWS API Gateway pricing [26]. The price of REST API requests is 3.5 \$ per a million monthly requests for the first 333 million. Our use case will only have ten thousand requests resulting in a price of 0.035 \$ a month. Adding this to our Lambda cost estimate, we get 0.25 \$ a month, rounded up. This is less than one percent of our current monthly costs using EC2. This would lead to a yearly saving of approximately 800 \$, when compared to the EC2 on-demand instance prices. If such measures could be safely transferred to other similar services, the potential for cost saving is substantial.

It is worth noting, that the load faced by this application in question is relatively low and irregular. The EC2 instances used by this application could probably handle a constant load at least ten times the current amount for the same cost. Meanwhile, the Lambda costs scale linearly along with the amount of requests.

3.4 The Application

The targeted service in this study is called the push content API. It is a Ruby web service using Sinatra. Sinatra is a minimalistic domain-specific language that can be used to build web services such as HTTP REST APIs on top of the Ruby language. It abstracts Rack, a modular interface library underlying in most contemporary Ruby web frameworks [33]. Interestingly for our use case, Sinatra can also be deployed in to a Lambda function to handle the requests proxied by our API Gateway. This is covered in more detail in Chapter 4. This is helpful, since we have multiple similarly built applications that could benefit from the switch to Lambda.

The push content API is a small back end service that waits for an event to trigger it with an article id. Once triggered, it looks for information in the database concerning the given article and sends Twitter and mobile push notifications to subscribed readers. Usually, this happens after a journalist writes an article in the writer software and chooses to send notifications about the given publication. Such a workflow seems pretty fitting for the event-driven serverless infrastructure.

Chapter 4

Implementation

In this chapter we discuss the methods used and our proposed implementation. The main objective for the study is to recreate a current online service using an alternate method in Lambda and identifying whether or not this is profitable. Section 4.1 revisits the research questions and objectives of the study and Section 4.2 reveals our choice of Lambda runtime. We dive further into the infrastructure setup in Section 4.3 and finally, we examine the steps of deployment in Section 4.4.

4.1 Objectives

Our objective in this study is to get familiar with contemporary serverless methodologies and how to apply them to classic web services. Meanwhile, cost savings can be made for the hosting of these services. This study also attempts to establish said methodologies in a manner such that they can be applied to other similar services created by our team. Our research questions were defined as follows:

- RQ1: What is serverless and where is it useful?
- RQ2: How to transform a PaaS to serverless?
- RQ3: How do the two different methods compare to one another?

We have discussed RQ1 and the differences between the PaaS and FaaS methodologies in Chapter 2. Given what we covered so far, this Chapter aims to answer RQ2 and we will go into deeper detail concerning RQ3 in the next Chapter.

4.2 Runtime

Previously, we identified that our problem is hosting a number of web applications that are generally idle for a majority of the time. For the purpose of this study, we are looking at transferring an existing Elastic Beanstalk web service into one that runs in a serverless fashion on Lambda. We explained the differences between the two cloud services in Section 2.4. For the implementation, we needed to also decide the Lambda function run time, that is, the programming language in which the serverless function is to be implemented. AWS supports the following runtimes: [34]

- Node.js
- Python
- Ruby
- Go
- .NET
- Custom

Node.js is the most common among them and if we were to implement something from scratch, it would probably be our choice. Python is most suitable for data handling and machine learning purposes. Ruby is something we are already familiar with concerning our back-end web services. Other types of runtimes are also supported, but they are not so typical amongst the technologies used in our organization. As mentioned previously in Section 3.4, we identified that it is possible to pack our existing Sinatra application code to a Lambda container and connect it to AWS API Gateway. For this study, we are going to examine this possibility further, because if viable, we could be able to utilize this implementation in other similar services.

This implementation is divided into three main parts. **AWS API Gateway** creates the outer layer that is exposed into the internet. It accepts requests and proxies them onward to the application. The application resides in our **Lambda**, that creates a container for running code triggered by the API Gateway event. Inside the container lies our Ruby **Sinatra** application that handles the requests accordingly. As stated previously, Sinatra is originally purpose-built to handle requests as a web server. In our implementation, it is merely code in a zip package that takes the requests as function parameters.

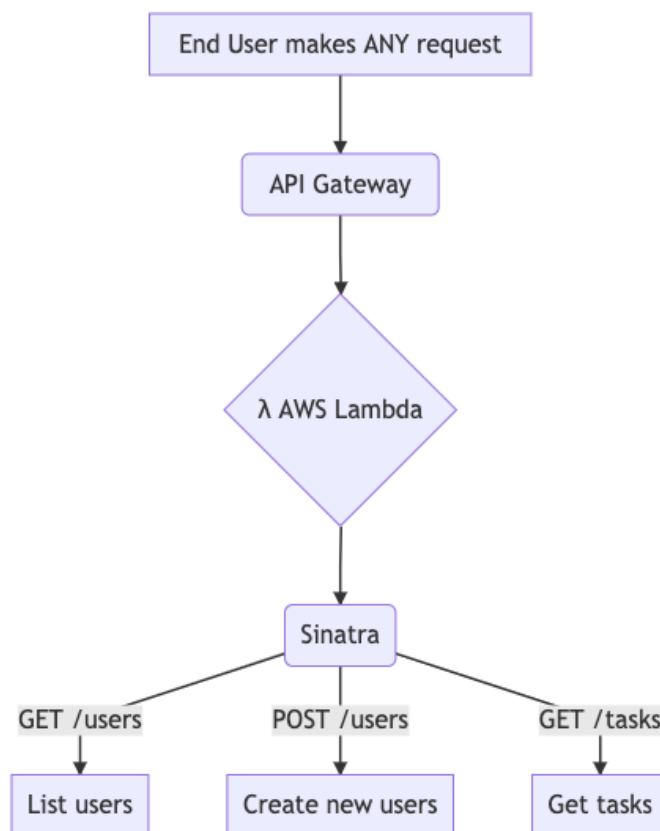


Figure 4.1: API Gateway, Lambda and Sinatra example

Figure 4.1 shows a graph of how API Gateway, Lambda and Sinatra would work together as an example serverless back-end API. API Gateway creates an URL that can be accessed from a web browser or by another application. Let us consider an end user that would like to get a list of all the users in the system. The end user would send a GET request to the url of the API Gateway appended with the example path of /users. API Gateway is set up to proxy all requests forward to our application along with information of the event, including the given path /users and the method used GET. Lambda gets triggered and executes our Ruby application that receives the event as a parameter to the handler function as explained in Section 2.4. The handler function shown in Listing 4.1 is at the very center of our RQ2 and in connecting the Sinatra application to Lambda. It is a slightly modified version of the one included in the GitHub repository `aws-samples/serverless-sinatra-example` [35].

```
1 require 'json'
2 require 'rack'
3 require 'base64'
4
5 $app ||= Rack::Builder.parse_file("#{__dir__}/config.ru").
  first
6
7 def handler(event:, context:)
8   body =
9     if event['isBase64Encoded']
10       Base64.decode64 event['body']
11     else
12       event['body']
13     end || ''
14
15   headers = (event['headers'] || {})
16
17   env = {
18     'REQUEST_METHOD' => event.fetch('httpMethod'),
19     'SCRIPT_NAME' => '',
20     'PATH_INFO' => event.fetch('path', ''),
21     'QUERY_STRING' => Rack::Utils.build_query(event['
  queryStringParameters'] || {}),
22     'SERVER_NAME' => headers.fetch('Host', 'localhost'),
23     'SERVER_PORT' => headers.fetch('X-Forwarded-Port', 443).
  to_s,
24
25     'rack.version' => Rack::VERSION,
26     'rack.url_scheme' => headers.fetch('CloudFront-Forwarded-
  Proto') { headers.fetch('X-Forwarded-Proto', 'https') },
27     'rack.input' => StringIO.new(body),
28     'rack.errors' => $stderr
29   }
30
31   headers.each_pair do |key, value|
32     # Content-Type and Content-Length are handled specially
  by Rack
33     name = key.upcase.gsub '-', '_'
34     header =
35       case name
36       when 'CONTENT_TYPE', 'CONTENT_LENGTH'
37         name
38       else
39         "HTTP_#{name}"
40       end
41     env[header] = value.to_s
42   end
43
44   begin
```

```
45 # Response from Rack must have status, headers and body
46 status, headers, body = $app.call env
47
48 # body array into string
49 body_content = ''
50 body.each do |item|
51   body_content += item.to_s
52 end
53
54 # We return the structure required by AWS API Gateway
55 response = {
56   'statusCode' => status,
57   'headers' => headers,
58   'body' => body_content
59 }
60 rescue Exception => exception
61   # Handle any exception
62   response = {
63     'statusCode' => 500,
64     'body' => exception.message
65   }
66 end
67
68 # By default, the response serializer will call to_json for
69 # us
70 response
```

Listing 4.1: Handler function

The handler itself is fairly minimal and contains only essential handling and pre-processing of the event information required by our Sinatra application. Worth noting is that we have placed the definition of our Rack application before the handler function and outside of its scope. This is due to the fact, that when Lambda is executed, it creates a container with the application code. After running the code, in response to the given request, the container is dismissed. In the case that another request is received before the container is dismissed, we can re-utilize code that has been defined and executed outside the scope of the handler.

Inside the scope of the handler function, we first make sure that the body of the incoming request is not Base64 encoded. Next, we set the environment for Rack and Sinatra. Most importantly, we set the request method and path info from the event along with other crucial information such as the logging destination for possible errors of Rack. Then, we pre-process the headers of the request to the correct format and execute our Sinatra application with the prepared environment. Lastly, we format our application response as per

the requirements of API Gateway and return it from the handler function.

Given this relatively generic handler function, it should be reasonably straightforward to connect any existing Sinatra application to the code for serverless deployment. The code package should consist of the proposed handler function file and a minimal config file that launches the Sinatra application. Along with these, the Sinatra application that consists of path routing and the web service logic itself should be bundled for deployment and compressed in a zip package. For our Lambda to have easy access to this package, it should be uploaded into an S3 bucket with versioning enabled.

4.3 Infrastructure

We use CloudFormation to build pieces of cloud infrastructure in a repeatable manner that is also easily readable with version control technologies. Listing 4.2 presents our implementation of a CloudFormation stack that puts together the required infrastructure and permissions required by the Lambda and API Gateway. An S3 bucket is a pre-requisite that can not be created via the same template.

```
1 AWSTemplateFormatVersion: '2010-09-09'
2
3 Description: <Name of your application>
4
5 Parameters:
6   VersionId:
7     Type: String
8     Description: Version id of the latest zip
9
10 Resources:
11
12   ApiGatewayRestApi:
13     Type: AWS::ApiGateway::RestApi
14     Properties:
15       ApiKeySourceType: HEADER
16       Description: An API Gateway for a sample API
17       EndpointConfiguration:
18         Types:
19           - REGIONAL
20       Name: Sample API
21
22   ApiGatewayResource:
23     Type: AWS::ApiGateway::Resource
24     Properties:
25       ParentId: !GetAtt ApiGatewayRestApi.RootResourceId
26       PathPart: '{proxy+}'
```

```

27     RestApiId: !Ref ApiGatewayRestApi
28
29   ApiGatewayMethod:
30     Type: AWS::ApiGateway::Method
31     Properties:
32       ApiKeyRequired: false
33       AuthorizationType: NONE
34       HttpMethod: ANY
35       Integration:
36         ConnectionType: INTERNET
37         Credentials: !GetAtt ApiGatewayIamRole.Arn
38         IntegrationHttpMethod: POST
39         Type: AWS_PROXY
40         Uri: !Sub 'arn:aws:apigateway:${AWS::Region}:lambda:
path/2015-03-31/functions/${LambdaFunction.Arn}/
invocations '
41       ResourceId: !Ref ApiGatewayResource
42       RestApiId: !Ref ApiGatewayRestApi
43
44   ApiGatewayModel:
45     Type: AWS::ApiGateway::Model
46     Properties:
47       ContentType: 'application/json'
48       RestApiId: !Ref ApiGatewayRestApi
49       Schema: {}
50
51   ApiGatewayStage:
52     Type: AWS::ApiGateway::Stage
53     Properties:
54       DeploymentId: !Ref ApiGatewayDeployment
55       Description: Lambda API Stage v0
56       RestApiId: !Ref ApiGatewayRestApi
57       StageName: 'v0'
58       MethodSettings:
59         - ResourcePath: "/*"
60           HttpMethod: "*"
61           LoggingLevel: INFO
62
63   ApiGatewayDeployment:
64     Type: AWS::ApiGateway::Deployment
65     DependsOn: ApiGatewayMethod
66     Properties:
67       Description: Lambda API Deployment
68       RestApiId: !Ref ApiGatewayRestApi
69
70   ApiGatewayIamRole:
71     Type: AWS::IAM::Role
72     Properties:
73       AssumeRolePolicyDocument:

```

```
74     Version: '2012-10-17'
75     Statement:
76     - Sid: ''
77       Effect: 'Allow'
78       Principal:
79         Service:
80         - 'apigateway.amazonaws.com'
81       Action:
82         - 'sts:AssumeRole'
83     Path: '/'
84     Policies:
85     - PolicyName: LambdaAccess
86       PolicyDocument:
87         Version: '2012-10-17'
88         Statement:
89         - Effect: 'Allow'
90           Action: 'lambda:*'
91           Resource: !GetAtt LambdaFunction.Arn
92
93     LambdaFunction:
94       Type: AWS::Lambda::Function
95       Properties:
96         Code:
97           S3Bucket: <name of your S3 bucket>
98           S3Key: <name of your code zip>.zip
99           S3ObjectVersion: !Sub ${VersionId}
100        Description: AWS Lambda function
101        FunctionName: <name of your application>
102        Handler: app.handler
103        MemorySize: 128
104        Role: !GetAtt LambdaIamRole.Arn
105        Runtime: ruby2.7
106        Timeout: 60
107
108     LambdaIamRole:
109       Type: AWS::IAM::Role
110       Properties:
111         AssumeRolePolicyDocument:
112         Version: '2012-10-17'
113         Statement:
114         - Effect: 'Allow'
115           Principal:
116             Service:
117             - 'lambda.amazonaws.com'
118           Action:
119             - 'sts:AssumeRole'
120         ManagedPolicyArns:
121         - "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"
```

```
122 Path: '/'
```

Listing 4.2: Lambda & API Gateway Cloudformation YML

Listing 4.3 depicts a minimal way to create the bucket to host our application code. This CloudFormation template creates a bucket with all public access restricted as proposed by AWS. However, for simplicity, it currently allows all principals access to get objects from this bucket. To restrict access further, the bucket policy can be removed and the Lambda function can be given access to this bucket, for example.

```

1 AWSTemplateFormatVersion: '2010-09-09'
2
3 Description: Bucket to host Lambda code
4
5 Resources:
6
7   BucketPolicy:
8     Properties:
9       Bucket: !Ref LambdaBucket
10      PolicyDocument:
11        Statement:
12          - Sid: AddCannedAcl
13            Effect: Allow
14            Principal: '*'
15            Action: 's3:GetObject'
16            Resource: !Join
17              - ''
18              - - 'arn:aws:s3:::'
19                - !Ref LambdaBucket
20                - /<name of your zip>.zip
21      Type: 'AWS::S3::BucketPolicy'
22
23   LambdaBucket:
24     Type: AWS::S3::Bucket
25     Properties:
26       AccessControl: BucketOwnerFullControl
27       BucketEncryption:
28         ServerSideEncryptionConfiguration:
29           - ServerSideEncryptionByDefault:
30             SSEAlgorithm: AES256
31       BucketName: <name of your S3 bucket>
32       PublicAccessBlockConfiguration:
33         BlockPublicAcls: true
34         BlockPublicPolicy: true
35         IgnorePublicAcls: true
36         RestrictPublicBuckets: true
37       VersioningConfiguration:

```

```
38 Status: Enabled
```

Listing 4.3: S3 Bucket Cloudformation YML

4.4 Deployment

To initially deploy the application, three steps are required. First, we create the S3 bucket using CloudFormation and the template presented in Listing 4.3. The shell script that can be used to create this bucket is depicted in Listing 4.4. Second, we copy our application code to the bucket we just created. If our application requires some additional Ruby gems, we prepare by building our application for deployment beforehand. In order for our Lambda container to communicate with the gems we built, we make sure to build the application with the same Ruby version as the Lambda run time we have selected. Here, Docker can be utilized to build with another Ruby version than the one installed locally on one's machine. After the build has been done, our application code and the built gems should be packed in a zip package and copied into the S3 bucket. Third, and finally, we create the API Gateway and Lambda stack with the CloudFormation template shown in Listing 4.2. This template takes the version id of our Lambda function as a parameter and can be fetched with the AWS CLI or the web console. After this, our application is ready to use with the url found in the API Gateway AWS console.

```
1 #!/bin/bash
2 set -euo pipefail
3
4 aws cloudformation deploy \
5     --capabilities "CAPABILITY_NAMED_IAM" \
6     --stack-name "<S3 Bucket stack name>" \
7     --template-file "lambda-s3.yml" \
8     --no-fail-on-empty-changeset
```

Listing 4.4: S3 deployment script

For further deployments, we can skip the first step as we have already created the S3 bucket. The latter steps can be quite simply integrated into common CI/CD pipelines. After making code changes:

- If changes in Gemfile, build application
- Zip application
- Upload application code into S3

- Fetch version id of previous upload
- Deploy CloudFormation

Listing 4.5 depicts an example shell script that runs the above listed steps given the assumption that the application code including a Gemfile and the main CloudFormation template, called `cloudformation.yml`, reside in the same directory as said script.

```

1 #!/bin/bash
2 set -euo pipefail
3
4 # Build the gems using Docker to ensure Ruby 2.7
5 docker run -v 'pwd': 'pwd' -w 'pwd' -i -t lambci/lambda:build-
   ruby2.7 bundle install --path vendor/bundle
6
7 # Zip application
8 zip -r app.zip app.rb vendor # + Other folders / files
9
10 # Copy to s3
11 aws s3 cp \
12     "./app.zip" \
13     "s3://<bucket name>/"
14
15 # Fetch version id
16 version_id=$(aws s3api list-object-versions \
17     --bucket "<bucket name>" \
18     --prefix "app.zip" | \
19     jq '.Versions[] | select(.IsLatest == true)'
   | jq --raw-output '.VersionId')
20
21 # Deploy CloudFormation
22 aws cloudformation deploy \
23     --capabilities "CAPABILITY_NAMED_IAM" \
24     --stack-name "<CloudFormation stack name>" \
25     --template-file "cloudformation.yml" \
26     --no-fail-on-empty-changeset \
27     --parameter-overrides \
28     "VersionId=$version_id"

```

Listing 4.5: Deployment shell script

Now we have an up and running Lambda HTTP REST API with API Gateway and a Ruby Sinatra application serving requests. Our API endpoint URL can be retrieved from our AWS API Gateway console. Alternatively, it can be constructed using:

```

1 https://<api_id>.execute-api.<region>.amazonaws.com/<stage>

```

Where api id can be fetched using AWS CLI v2 [36] and the following command:

```
1 aws apigateway get-rest-apis
```

Our region is eu-west-1 and we set the name of the stage in Listing 4.2 on line 57 to be v0.

Chapter 5

Results and analysis

In this chapter, we look at how the implemented methods compare to existing services. In Section 5.1 we will compare our previous calculations to actualized numbers. Next, to compare our previous solution and the one suggested in this thesis, Section 5.2 will examine the differences in performance while Section 5.3 will cover the visible changes in deployment speed and logging. Finally, in Section 5.4 we will consider how reliable this solution is in comparison to other similar Elastic Beanstalk applications and examine its transferability.

5.1 Actualized runtime and cost

In Section 3.3 we performed an estimation of the costs of this Lambda function. We established that with 3,300 monthly compute seconds our monthly cost of the Lambda function alone would be:

$$EstimatedMonthlyCost = 3,300 \text{ s} * \frac{128}{1,024} \text{ GB} * 0.00001667 \frac{\$}{\text{GB} * \text{s}} = 0.0068 \$$$

Now, the estimate of 3,300 monthly compute seconds was somewhat optimistic, since it was the request response time of a dedicated web server. To gain actual statistics of our Lambda function run times, we deployed the application and its new serverless infrastructure to our development environment. In this environment, it processed traffic identical to our production environment without any risk. Calculating the average over one week gave us an average for the runtime for a single call of the Lambda function of 911 ms. The amount of average daily function invocations was 260. This leads to a daily function runtime of

$$0.911 \text{ s} * 260 \text{ (Invocations)} = 236.86 \text{ s}$$

To get monthly numbers we multiply this by 30 and round it to 7,105.8 monthly compute seconds. That would make our actual monthly cost as follows:

$$\text{ActualizedMonthlyCost} = 7,105.8 \text{ s} * \frac{128}{1,024} \text{ GB} * 0.00001667 \frac{\$}{\text{GB} * \text{s}} = 0.0148 \$$$

The actualized cost is around double our estimate, but the total cost is still in the same order of magnitude as our total cost estimate, under 0.25 \$. It is important to note, that the load of the push content API in question is quite light and irregular. These calculations could prove to be much greater for more frequently utilized applications.

5.2 Performance

As mentioned previously, Elastic Beanstalk and Lambda create a problematic comparison. Largely due to the fact that the first is a PaaS, while the other is a FaaS. In this section, we take a closer look at the performance differences between the two applications now that we have them doing work side by side. To help visualize the load handling, Gatling [37] was used to produce artificial load to the applications.

Elastic Beanstalk allows you to run applications in the cloud in a similar manner to running them on local physical computers. This commonly means a server listening actively to requests. The cloud adds monitoring and scalability to the mix, so that when the load of the machine partition in the cloud rises, more computational power is deployed automatically. Elastic Beanstalks are thus highly elastic and hard to break, even with intensive sudden workloads such as Distributed Denial of Service (DDoS) attacks [38]. In the face of an Elastic Beanstalk service, an upper limit is set for the amount of EC2 instances deployed at maximum. In the scenario that the upper limit is met and the load still manages to overload the application, it might face trouble. This can quite easily be further prevented with CloudFront caching for example. With CloudFront set up, the sudden spike in requests simply end up loading a cached response and never even touch the application itself. However, if the requests differ from one another, CloudFront can not help. Furthermore, the reaction time of the elasticity is relatively slow as it can take minutes for additional EC2 instances to be online following breach detection and deployment.

There is little to none when it comes to infrastructure surrounding Lambda, meaning it is noticeably simpler. As we mentioned previously in our calculations, more requests and/or longer runtime mean more cost. This leads to

the fact that our baseline price is extremely small, but surprising high spikes in traffic could lead to significant rise in cost. The application itself should not ever face downtime since responsibility of that is outsourced to the cloud provider.

Gatling was used to produce constant load for both applications. It was set up to produce 10 users per second for 3 minutes and then 18 users per second for 7 minutes. Both applications would face 37,440 requests in a time span of 10 minutes. The Elastic Beanstalk was configured to a single initial EC2 instance and to scale up to two instances after breaching 85 % CPU load for a minute. Figure 5.1 shows the requests that the application failed to respond to in red. With this set up, it failed 5,054 requests in the approximately 3 minutes and 45 seconds it took to identify the breach and scale up. Figure 5.2 shows the Lambda side that failed one request in total.

Looking at what we, at Italehti have faced before, the order of magnitude of DDoS attacks are from a couple million to 15 millions of requests in a matter of minutes. The additional base price for a million requests to a public API hosted in the manner presented by this thesis would be 3.5 \$ for API Gateway, another 0.2 \$ for the base requests of Lambda. In addition to these base costs, one million requests for our runtime average and function size would cost:

$$0.911 \text{ ms} * 10^6 * \frac{128}{1024} \text{ GB} * 0.00001667 \frac{\$}{\text{GB} * \text{s}} \approx 1.9 \$$$

This sums up to 5.4 \$ per a million additional requests. Thus, a few million requests are still negligible in comparison to the base cost of an EC2 instance.

We mentioned facing an attack of 15 million requests earlier, which would amount to:

$$5.4 \frac{\$}{M \text{ Requests}} * 15 M \text{ Requests} = 81 \$$$

However, currently the push content API with two t3.medium EC2 instances can not handle as much requests. To determine the amount of requests it can properly serve with the current Elastic Beanstalk infrastructure, Gatling was used to stress test the application. To get a clear picture, only one instance of t3.medium was used for this Gatling stress test.

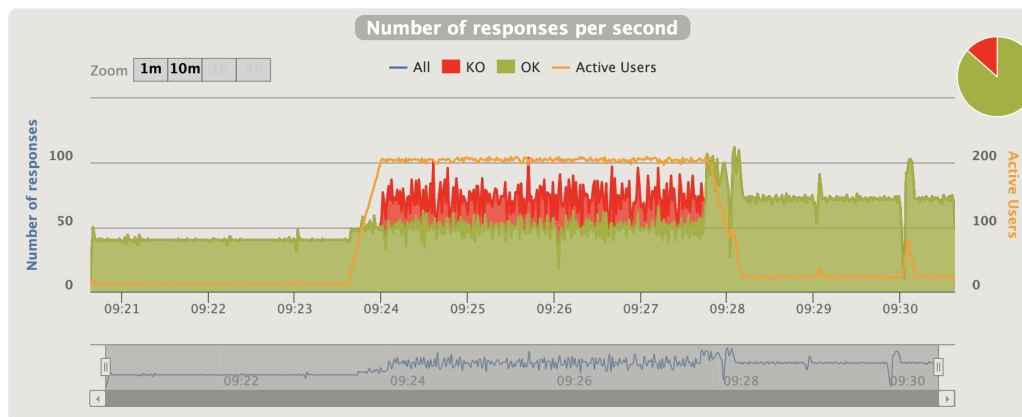


Figure 5.1: Elastic Beanstalk load handling

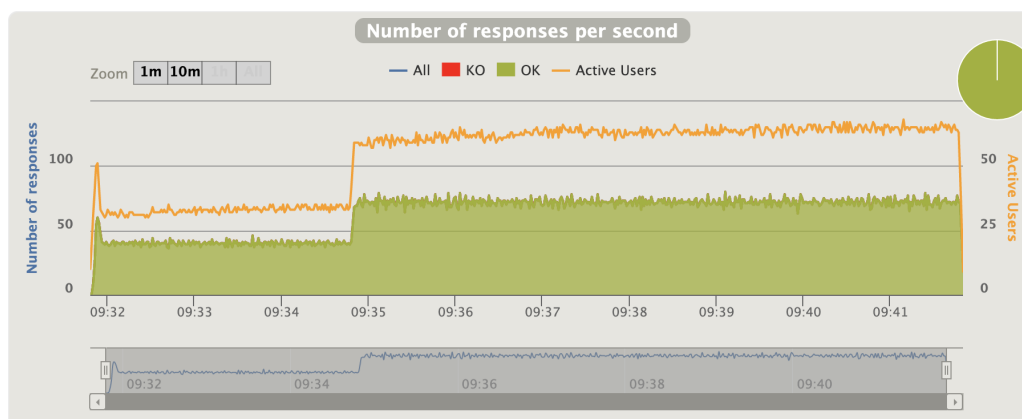


Figure 5.2: Lambda load handling

```

1 -----
2 Global Information -----
3 > request count 31452 (OK=7835 KO=23617 )
4 > min response time 1 (OK=2 KO=1 )
5 > max response time 6071 (OK=6071 KO=84 )
6 > mean response time 855 (OK=3421 KO=4 )
7 > std deviation 1652 (OK=1480 KO=3 )
8 > response time 50th percentile 4 (OK=3763 KO=2 )
9 > response time 75th percentile 12 (OK=4547 KO=5 )
10 > response time 95th percentile 4644 (OK=5092 KO=8 )
11 > response time 99th percentile 5137 (OK=5416 KO=12 )
12 > mean requests/sec 179.726 (OK=44.771 KO=134.954)
13 -----
14 Response Time Distribution -----
15 > t < 800 ms 915 ( 3%)
16 > 800 ms < t < 1200 ms 61 ( 0%)
17 > t > 1200 ms 6859 ( 22%)
18 > failed 23617 ( 75%)
19 -----
20 Errors -----
21 > status.find.in(200,201,202,203,204,205,206,207,208,209,304), f 23617 (100.0%)
22 found 502

```

Listing 5.1: Gatling report

Listing 5.1 shows the amount of requests sent during the Gatling test run. Over 30,000 requests were sent with a mean rate of nearly 180 requests per second. These requests happened over a time-span of approximately three minutes. 7,835 requests were successful out of the 31,452. From these results, we can deduct that this application and one t3.medium instance can serve a spike of 7,835 requests. As two such instances are used in production, we multiply that amount by two to get 15,670 requests for two instances. Using the previously defined average Lambda runtime of 911 ms, the additional price for 15,670 Lambda requests would be:

$$SpikeCost = 0.911 \text{ s} * 15,670 \text{ (reqs)} * \frac{128}{1,024} \text{ GB} * 0.00001667 \frac{\$}{\text{GB} * \text{s}} \approx 0.03 \text{ \$}$$

API Gateway would cost an additional 0.055 \$ on top of the request price. Thus, a load spike of this size could be handled for less than 0.1 \$.

We can also define the amount of requests our Lambda application can serve for the same price as the application with two on-demand t3.medium instances:

$$66.456 \text{ \$} / 5.4 \frac{\$}{M \text{ requests}} \approx 12.3 \text{ M requests}$$

This, to some degree, highlights the level of over provisioning, since our current application is only serving around 10,000 requests monthly.

To ease up the amount traffic to our Lambda, there are a couple of ways including:

- Access restriction
- API Gateway limitations
- Web Application Firewall (WAF)

The easiest one is limiting access to the API itself. This would probably be the most obvious one for us, since the application in question is only used by our own services internally. Thus, we could set the API to be accessible only from within our own Virtual Private Cloud (VPC) and we would never face an attack. API Gateway can also be set to limit the amount of traffic, for example, to only accept one request per second. Thirdly, AWS also offers Web Application Firewall (WAF), that allows for protection against many types of attacks. The focus of WAF is in injection attacks and other attacks that do not typically create extensive load to an application, but it can still be used to protect against some cases with rate-limits on subsequent requests from the same IP address [39].

5.3 Deployment & Logging

Another aspect that differs is the application deployment. Due to the fact that the infrastructure of our Lambda implementation is much simpler, the deployment becomes noticeably quicker than the more classic approach. Deploying to Elastic Beanstalk creates a new EC2 instance and after starting up the operating system, it executes the application inside. Depending on the application set up and the deployment pipeline, it takes at least a couple of minutes to have a newly deployed application version up and running. Due to the application server constantly running, this also leads to a certain level of need for monitoring the said application. The simple, serverless nature of Lambda cuts out the need to configure an operating system in the deployment process, since that is a concern for the cloud provider. We only upload the function code into our S3 bucket and update the CloudFormation stack with the new function version id so that the Lambda container fetches the up-to-date application when executed. This takes no longer than seconds on top of the time necessary to upload the code, which sums up to arguably a fraction of the deployment time in comparison to the classic infrastructure.

Logging is another dimension of difference between these two implementations. When considering the Elastic Beanstalk application, it resides in an EC2 instance that imitates an actual computer. Therefore, it produces logs of everything and stores them in a log folder located on the instance. Thus, if there is an error with the instance or the instance is updated and rebuilt, the logs are lost. Furthermore, reading the logs requires one to download them from the AWS web console as text files or connecting to the EC2 instance directly. On the other hand, Lambda logs everything rather neatly to the AWS monitoring and logging service called CloudWatch that can be queried with ease and requires no additional setting up.

Figure 5.3 shows an example graph of the number of invocations of our function for one week. Figure 5.4 presents the average, maximum and minimum duration that the function ran for in one graph and Figure 5.5 shows error count and success rate. A sample of logs for one function invocation can be seen in Figure 5.6. All the executed log commands written in the function code can be found there in different levels, I stands for Info and E for Error. Finally, after the execution has finished, the system automatically logs the billed duration and memory usage information.

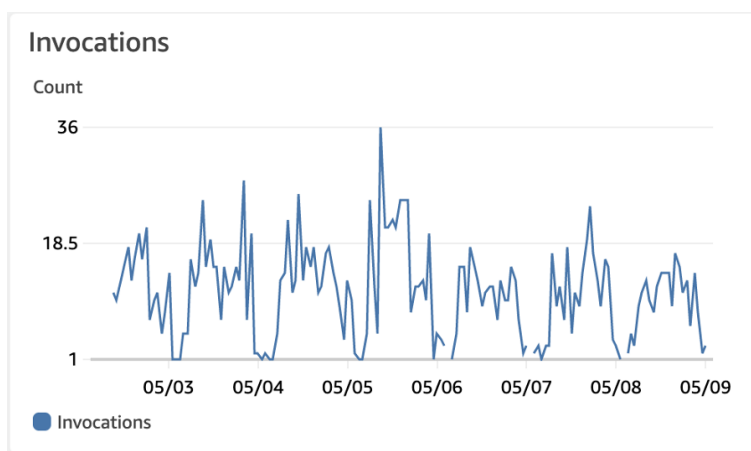


Figure 5.3: Number of invocations

5.4 Reliability & Transferability

The application chosen for this study was our push content API. When it comes to push content for a news media platform, quickness is a priority. After a reporter publishes an article, the information flow to the reader must be swift. Consequently, it is in our interest to not have a middleman API service as a bottleneck for said flow. As mentioned in Section 2.1 cold starts can add up to 17% to the round-trip-time of a request. However, after testing the response times with the unix curl command, the cold starts seemed to blend in with natural variance of the response time. During testing, single cold starts were forced by uploading a new version of the application code with no actual changes.

The application has a health check path that simply checks whether the application is in working condition and if so, responds with a 200 response code and the message OK. We tested and timed one thousand requests to this health path in both applications, one running in our current Elastic Beanstalk infrastructure and one in the serverless Lambda infrastructure proposed by this thesis. The average round-trip-times presented in Table 5.1 show that there is a small, although quite insignificant, difference in favor of the Elastic Beanstalk application. This is probably due to a difference in the provisioning of the Lambda versus the over-provisioned Beanstalk instance. As mentioned in Section 2.4 the underlying virtual machines are almost identical when it comes to processing power.

In case of the push content API, we found no noticeable difference in the times the notifications and tweets were delivered. Furthermore, the serverless

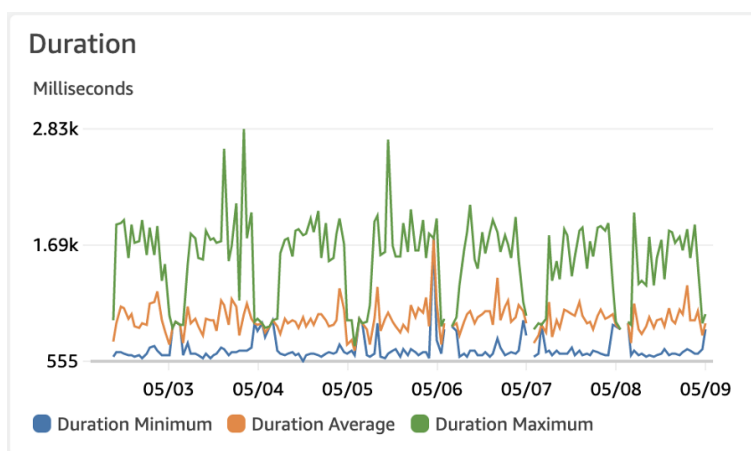


Figure 5.4: Function run durations

Platform	Average round-trip-time
Elastic Beanstalk	0.22514s
Lambda	0.25294s

Table 5.1: Health check path response times

application faced no faults in two months. The handler code and infrastructure configuration presented in Chapter 4 are extremely general to a Sinatra application. Therefore, this transformation should be applicable to almost any Sinatra application with only fine-tuning the way parameters are received and possible Ruby version issues. However, it is important to recall the limitations of Lambda: 15 minutes maximum execution time and 6 MB maximum invocation payload. This means, that for example handling files sized over 6 megabytes in a file upload application will be a problem and require some additional attention. Other than that, it seems like it is worth considering Lambda for REST API services.

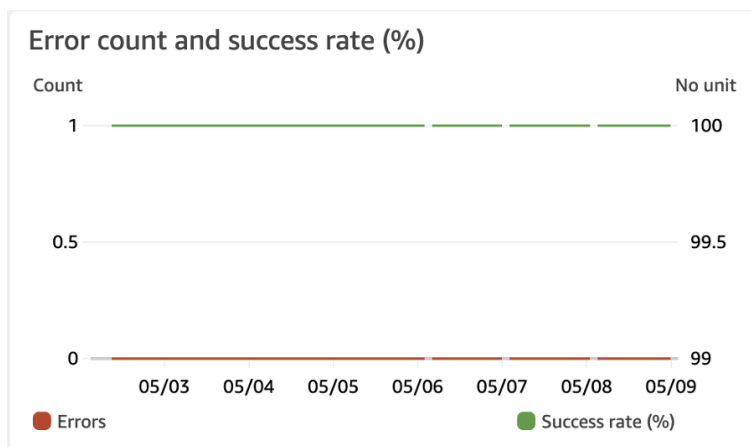


Figure 5.5: Errors

▶	2021-07-08T07:46:01.003+03:00	START RequestId: d4aa7af3-28d4-49a3-8229-65dfe4f46255 Version...
▶	2021-07-08T07:46:01.518+03:00	I, [2021-07-08T04:46:01.518644 #8] INFO d4aa7af3-28d4-49a3-82...
▶	2021-07-08T07:46:01.572+03:00	I, [2021-07-08T04:46:01.572551 #8] INFO d4aa7af3-28d4-49a3-82...
▶	2021-07-08T07:46:01.895+03:00	I, [2021-07-08T04:46:01.882296 #8] INFO d4aa7af3-28d4-49a3-82...
▶	2021-07-08T07:46:02.235+03:00	E, [2021-07-08T04:46:02.226577 #8] ERROR d4aa7af3-28d4-49a3-8...
▶	2021-07-08T07:46:02.277+03:00	END RequestId: d4aa7af3-28d4-49a3-8229-65dfe4f46255
▼	2021-07-08T07:46:02.277+03:00	REPORT RequestId: d4aa7af3-28d4-49a3-8229-65dfe4f46255 Durati...
REPORT RequestId: d4aa7af3-28d4-49a3-8229-65dfe4f46255 Duration: 1272.75 ms Billed Duration: 1273 ms Memory Size: 128 MB Max Memory Used: 96 MB Init Duration: 813.45 ms		

Figure 5.6: Sample logs

Chapter 6

Discussion

Now that we have evaluated the performance of our proposed implementation and compared it to the current working version, we take a retrospective look at the research questions we set back in Chapter 1.

6.1 RQ1: What is serverless and where is it useful?

Serverless is a rapidly evolving cloud computing paradigm that originates from single, short functions. Along with its evolvment, it has begun overtaking whole applications, that have previously ran on dedicated server machines. It is offered by AWS under the name Lambda, along with Azure Functions and Google Cloud Functions being its biggest competitors. These services provide an easy way to upload function code into the cloud that can then be triggered via events.

The three main characteristics of serverless we identified in Section 2.1 are granular billing, lack of operational logic and that it is event-driven. Serverless continues along the path of cloud computing, but provides even higher granularity when it comes to provisioning and billing. Single function calls cost only for the time the application actually runs. This has obviously sparked interest in the industry and consequently, the development in serverless computing is being led by the industry. The lack of operational logic relieves pressure off of the customer of the cloud service. In short, the customer does not have to configure auto-scaling limits or think about resource management. Serverless outsources these concerns to the cloud service provider. At its core, serverless is event-driven. The functions are deployed as separate containers in response to events defined by the customer. For example, these events can mean timed jobs, such as every day at a specific

time or in response to an API call as presented by the implementation in this study.

Originally, serverless use cases included small tasks such as downloading and caching something at specific intervals or handling data coming and going to and from a database. Nowadays, the technology has evolved to include HTTP REST APIs such as the one implemented in this study and scalable chat bots [40], for example. It still has limitations when it comes to the payload size and maximum running times and it might be expensive if you have high loads using great amounts of memory.

6.2 RQ2: How to transform a PaaS to serverless?

The implementation proposed in Chapter 4 explains how to transform a Ruby Sinatra application that is currently used by Iltalehti in an Elastic Beanstalk PaaS environment to be operated in Lambda serverless architecture. In this particular case, the transform process was relatively straightforward. However, it should be mentioned that this shift is somewhat gimmicky, due to the fact that Sinatra and Rack are traditionally utilized as servers constantly in stand-by for requests. Whereas here, the application is facing a single API call as just a serverless function. For an application of this relatively small size, this implementation seems to work fairly well. Generally, the optimal solution in terms of performance would most likely be a rewrite of the application to be transformed. However, the amount of working hours necessary for the rewrite would most likely be excessive in comparison to the benefits.

To apply a transformation such as the one proposed, but for another technology stack, it would require customized handler code and possibly changes to the CloudFormation stack as well. In the scope of this thesis, Ruby Sinatra was chosen as it represents multiple similar applications in our environment and is therefore within our interest.

6.3 RQ3: How do the two different methods compare to one another?

First, it is essential to recall that, in the original sense of PaaS and serverless, we are comparing two highly dissimilar types of infrastructure. One of them is a platform service, whereas the other is an application container. However, as serverless functions continue development, the line between the

two thins. What we found during this study is that the comparison seems to favor serverless quite heavily. In our use case, serverless would cost less and generate more easily readable logs and allow quicker deployments.

What comes to cost, applying our proposed implementation to the application in question would lessen the cloud hosting cost of that application by approximately 99 %. While that sounds considerable, it is worth mentioning that for this single service the costs are relatively low to begin with, in the magnitude of 66.5 \$ per month and with our implementation, 0.3 \$ per month. Nonetheless, the percentages should spike interest in a bigger picture with multiple applications that would lead to greater accumulated savings. Yet, unexpectedly large loads could generate significant costs. Therefore, some level of protection against DDoS attacks could be rational. Applications that could be made private and accessible only from inside the virtual private cloud would be easiest to protect and consequently ensure the low cost. Still, for this application, we could serve 12.3 million additional requests on top of the expected ten thousand baseline to match the price of two current EC2 instances.

Chapter 7

Conclusions

For Iltalehti and the industry in general, streamlining costs is of great interest. Having evolved hand-in-hand with the needs of the industry, cloud computing has driven forth the granularity of billing in hosting web applications. The granularity along with the ease of development and testing have lead to the rise of popularity in microservices and, consequently, serverless computing. At Iltalehti, we have traditionally created REST API services utilizing the Elastic Beanstalk platform. Recently, however, we have discussed about trying out serverless versions of the applications in question.

In this thesis, we transformed an existing HTTP REST API web service running in the AWS PaaS Elastic Beanstalk to run on the serverless Lambda infrastructure instead. The application we chose to transform is the push content API, responsible of sending push notifications to mobile applications and Twitter. The transformed application was originally built as a web server utilizing the Ruby language and Sinatra domain-specific language. Concerning our implementation, we decided to only apply minimal changes to the application code itself to maintain our ability to compare the two infrastructures effectively. Thus, we break the serverless pattern to some extent, since we are shifting a web server application to reside in a serverless container. In this manner, we proxy all of the API requests to the application code where Sinatra decides which actions to take given the URL and request parameters.

In the original sense of serverless, we are using it rather improperly. Nevertheless, the implementation works rather well. We created a serverless application that reduces the regular monthly cost of this application from approximately 66.5 \$ to 0.3 \$. In both infrastructures, these costs scale with the workload the application faces. For the implementation proposed by this thesis, the application can face a spike of 12.3 million requests once a month in addition to the baseline average of 10,000 monthly requests to stay within

the cost of the Elastic Beanstalk version baseline.

We documented the process to be as repeatable and generic as possible for transferring Ruby Sinatra applications to Lambda. It remains to be seen if this implementation will see wide use within our development team, but it is certainly something worth considering.

Bibliography

- [1] Peter Mell, Tim Grance, et al. The NIST definition of cloud computing. 2011.
- [2] Erwin Van Eyk, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uță, and Alexandru Iosup. Serverless is more: From paas to present cloud computing. *IEEE Internet Computing*, 22(5):8–17, 2018.
- [3] Finnish Internet Audience Measurement. FIAM top lists, 2021. WWW: <https://fiam.fi/tulokset/>. Accessed 8 July 2021.
- [4] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The server is dead, long live the server: Rise of serverless computing, overview of current state and future trends in research and industry. *arXiv preprint arXiv:1906.02888*, 2019.
- [5] Erwin Van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. The SPEC cloud group’s research vision on FaaS and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 1–4, 2017.
- [6] Ling Qian, Zhiguo Luo, Yujian Du, and Leitao Guo. Cloud computing: An overview. In *IEEE International Conference on Cloud Computing*, pages 626–631. Springer, 2009.
- [7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. Technical report, Technical Report UCB/EECS-2009-28, EECS Department, University of California . . . , 2009.
- [8] AWS. Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta, 2006. WWW: <https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2--beta/>. Accessed 12 Mar 2021.

- [9] AWS. 2011 AWS Tour Australia, Closing Keynote: How Amazon.com migrated to AWS, by Jon Jenkins, 2011. WWW: <https://www.slideshare.net/AmazonWebServices/2011-aws-tour-australia-closing-keynote-how-amazoncom-migrated-to-aws-by-jon-jenkins/>. Accessed 12 Mar 2021.
- [10] Synergy Research Group. Cloud Market Ends 2020 on a High while Microsoft Continues to Gain Ground on Amazon, 2021. WWW: <https://www.srgresearch.com/articles/cloud-market-ends-2020-high-while-microsoft-continues-gain-ground-amazon>. Accessed 12 Mar 2021.
- [11] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. A review of serverless use cases and their characteristics. *arXiv preprint arXiv:2008.11110*, 2020.
- [12] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590. IEEE, 2015.
- [13] Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, et al. 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)*, pages 179–182. IEEE, 2016.
- [14] Johannes Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.
- [15] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.
- [16] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.
- [17] Michael Roberts and John Chapin. *What Is Serverless?* O’Reilly Media, Incorporated, 2017.

- [18] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.
- [19] Matthias Jung, S Malleri, P Dalbhanjan, P Chapman, and C Kassen. Microservices on aws. *Amazon Web Services, Inc., New York, NY, USA, Tech. Rep*, 2016.
- [20] Gojko Adzic and Robert Chatley. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 884–889, 2017.
- [21] Lisa Muller, Christos Chrysoulas, Nikolaos Pitropakis, and Peter J Barclay. A traffic analysis on serverless computing based on the example of a file upload stream on aws lambda. *Big Data and Cognitive Computing*, 4(4):38, 2020.
- [22] Jussi Nupponen and Davide Taibi. Serverless: What it is, what to do and what not to do. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 49–50. IEEE, 2020.
- [23] AWS. Lambda quotas, 2021. WWW: <https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html>. Accessed 12 Mar 2021.
- [24] AWS. Amazon EC2 Instance Types, 2021. WWW: <https://aws.amazon.com/ec2/instance-types/>. Accessed 25 May 2021.
- [25] AWS. Lambda pricing, 2021. WWW: <https://aws.amazon.com/lambda/pricing/>. Accessed 19 Mar 2021.
- [26] AWS. EC2 pricing, 2021. WWW: <https://aws.amazon.com/api-gateway/pricing/>. Accessed 19 Mar 2021.
- [27] James Murty. *Programming amazon web services: S3, EC2, SQS, FPS, and SimpleDB*. ” O’Reilly Media, Inc.”, 2008.
- [28] AWS. S3 pricing, 2021. WWW: <https://aws.amazon.com/s3/pricing/>. Accessed 23 Apr 2021.
- [29] AWS. S3 pricing, 2021. WWW: <https://aws.amazon.com/cloudformation/pricing/>. Accessed 23 Apr 2021.

- [30] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.
- [31] Amazon. Alexa siteinfo - Iltalehti, 2021. WWW: <https://www.alexa.com/siteinfo/iltalehti.fi>. Accessed 6 Mar 2021.
- [32] Vantage. EC2instances.info, 2021. WWW: <https://instances.vantage.sh/>. Accessed 12 July 2021.
- [33] Alan Harris and Konstantin Haase. *Sinatra: Up and Running: Ruby for the Web, Simply*. ” O’Reilly Media, Inc.”, 2011.
- [34] AWS. Lambda runtimes, 2021. WWW: <https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html>. Accessed 9 Apr 2021.
- [35] GitHub. serverless-sinatra-example, 2019. WWW: <https://github.com/aws-samples/serverless-sinatra-sample>. Accessed 16 Apr 2021.
- [36] AWS. AWS Command Line Interface, 2021. WWW: <https://aws.amazon.com/cli/>. Accessed 27 July 2021.
- [37] Gatling. Gatling.io, 2021. WWW: <https://gatling.io/>. Accessed 16 July 2021.
- [38] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004.
- [39] AWS. Using AWS WAF to protect your APIs, 2021. WWW: <https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-control-access-aws-waf.html>. Accessed 21 May 2021.
- [40] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, pages 1–4, 2016.