universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Aspects of Migrating Traditional Web Services to Serverless Applications“

verfasst von / submitted by

Bence Farkas

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2022 / Vienna, 2022

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Siegfried Benkner

Contents

Abstract

Serverless platforms are supplying a convenient way to engineers for deploying applications. This innovative approach changes the way how we interact with the cloud. The short-lived functions running in serverless containers are scaling out quickly and the providers only bill us for the execution time of the serverless functions. These are great advantages, but there are some aspects we need to consider before going completely “serverless”.

This thesis focuses on differences between a traditionally hosted web service (VM-based, containerized web server) using Elastic Container Service by Amazon Web Services and a web service hosted on the serverless platform also provided by Amazon Web Services, which is called AWS Lambda. It presents the implementation and hosting details of the traditionally hosted web service, then a migration of this application to the serverless platform. It extensively discusses the artifacts needed to host the web services using the so-called Infrastructure-as-Code approach.

Having the functionally equivalent web services hosted on the two different platforms makes it possible to compare them, focusing on response time, user experience, and code complexity differences. The thesis also presents the evaluation results based on these evaluation criteria.

Abstract (German)

Serverless Platforms bieten Entwicklern einen praktischen Ansatz, Applikationen bereitzustellen. Diese innovative Herangehensweise ändert die Art und Weise wie wir mit der Cloud interagieren. Die kurzlebigen Funktionen, die auf serverlosen Containern laufen, skalieren schnell und Anbieter verrechnen nur für die tatsächliche Ausführungszeit. Das alles hat große Vorteile, aber es gibt auch Aspekte, die beachtet werden müssen, bevor man Serverless implementiert.

Diese Arbeit fokussiert sich auf die Unterschiede im Vergleich zu traditionell gehosteten Web-Services (VM-based, containerized Web Server). Dabei wurden das Elastic Container Service und ein Webservice, gehostet auf der Serverless Platform AWS Lambda, beides von Amazon Web Services, verwendet. Sie demonstriert die Implementierung und Bereitstellung des traditionell gehosteten Webservices, der Migration dieser auf die Serverless Platform und erläutert ausführlich die Artifacts, die zum Hosten mit dem sogenannten Infrastructure-as-code Ansatz gebraucht werden.

Die Verfügbarkeit der gleichen Funktionalität auf beiden Plattformen, ermöglicht den Vergleich mit dem Fokus auf Antwortzeiten, User Experience und Code Komplexität. Diese Arbeit präsentiert auch das Resultat auf Grund dieser Vergleichswerte.

1 Introduction

1.1 Motivation

Since Amazon Web Services (AWS) debuted their Lambda computing platform back in November 2014, hosting applications in a serverless fashion gained increased popularity [1]. Lambda is a platform for hosting serverless applications, which means that the underlying infrastructure and code hosting is managed by the cloud provider entirely. Serverless functions are some executable code pieces with varying sizes executed in the cloud on demand, and the methodology of creating an application combined from serverless functions is called Function-as-a-Service (FaaS) [2]. Mike Roberts describes Function-as-a-Service as the following:

"Fundamentally, FaaS is about running backend code without managing your own server systems or your own long-lived server applications" [3].

Although Amazon Web Services was the first provider to supply the serverless computing model, many other providers (Google Cloud, Microsoft Azure) followed them. It brings a different mindset to the table about how we design our applications when we host them in the cloud.

Hosting an application using the traditional Virtual Machine-based [4] approach will incur costs at a per machine basis, while the provider bills serverless functions based on their execution time. There is clearly a potential for saving costs from the client's side in certain use cases. With their serverless platforms, the cloud providers also have great flexibility when deciding where to execute a function – this information is completely hidden from the client. They can use any of their hardware, even machines with different processors and memory configurations.

There is no general rule for deciding between deploying an application using the traditional Infrastructure-as-a-Service (IaaS) [5] or Platform-as-a-Service (PaaS) [6] approach and deploying and application using the serverless Function-as-a-Service (FaaS) model. Every application has different requirements, the system designers need to evaluate and choose the best possible hosting solution.

However, as in many other fields in Computer Science, we can see that there are general patterns and guidelines that we can follow. These guidelines aim to help us with making hosting and deployment decisions for our applications. There are even suggestions about how to create Function-as-a-Service setups to use all the benefits that this computational model offers [7].

These guidelines come in many forms: Scientific papers, official documentations on the cloud providers' websites, blog posts from experts in the field, etc. Having an active community around this topic is vital to make the serverless platforms better.

A general drawback of current serverless applications that they have high startup latency because of the cold start times of serverless functions. If the web service running in a serverless container takes too long to respond, it may lower the productivity of the user who is interacting with the service. People prefer user applications that are responsive, fast, and dependable. Would it be irritant to use an application that feels slow because of long response times?

1.2 Goal

The main goal of this Master Thesis is to analyze the aspects, such as possibilities and drawbacks, of migrating web services hosted in a traditional fashion to the serverless platform. I am going to present a case study in Section 3 and Section 4 about my experiences with such a migration.

Another goal is to evaluate the web services created in the case study based on some evaluation criteria, such as the response time, user experience, code complexity, or cost differences between the web services, comparing the two hosting solutions. These evaluations can be found in Section 5.

As the last goal of this Master Thesis, I want to identify potential guidelines to engineers who are looking into hosting their application on a serverless platform. There are multiple aspects to consider (for example, serverless web services should be stateless) when migrating or creating an application specifically for a serverless platform. The guidelines are presented in Section 6.5.

1.2.1 Expected Outcome

As part of my Master Thesis, I am going to develop two web services with the same functionality but hosted on the two platforms (traditional and serverless) mentioned in the previous sections, which have different requirements for application hosting and different runtime characteristics (performance, scalability).

Given the functionally equivalent web services running on these platforms, I can make comparisons between them. I want to investigate the differences in terms of development effort (architectural and code changes), I want to evaluate the performance aspects of the two platforms, and I want to see if there are any drawbacks or advantages between the two approaches.

I have a hypothesis that a web service hosted on the serverless platform may negatively affect the user experience of a user-facing application under certain circumstances. My plan is to reject or confirm this hypothesis based on the results of the evaluation results presented in the Thesis.

1.3 Organization

The thesis starts with an introduction, which presents an explanation of technical terms related to the concepts presented throughout the chapters (Section 1.4), and a basic overview of my case study (Section 1.5). As a next step, I am analyzing multiple research papers and studies related to the field (Section 2).

After I have investigated the related works from others, I am carrying out my own case study in which I create a web service and host it on Elastic Container Service [8] provided by Amazon Web Services, then I migrate this web service to Amazon Web Service's Lambda platform [9]. I evaluate and document the steps during the migration (Section 3 and Section 4). When I have the two versions of the same web service at hand, I perform an experimental evaluation in which I test how the different versions perform compared to each other in various aspects (Section 5).

The last section draws conclusions from the previous sections, summarizing their outcome (Section 6). It also contains a small set of guidelines for writing serverless applications based on what I discovered while working on this Master Thesis.

1.4 Technical Terms and Background

In this section, I want to give a quick overview about some of the technical terms that I am going to use throughout the Master Thesis. My goal is to help the understanding the rest of the paper. Note that these terms can have slightly different interpretations in other sources.

1.4.1 Server

In the context of this work, I consider a server to be the combination of a machine that is connected to other machines over the internet and a special application that manages the communication and data serving between these interconnected machines. In the following, when I use the word “server”, I refer to the software part of this architecture.

The server is waiting for incoming requests on a specified port. The ports range from 0 to 65535 and they serve the purpose of enabling multiple servers on the same machine simultaneously. For example, a single machine can run two servers, one listens on the port 3000, the other on 3001. This machine has an address. The clients can reach both servers by using the machine’s address and specifying the right port. Figure 1 gives an overview of this architecture.

The machines that take part in this communication scheme are the clients and the servers, and the system is called the client-server system. The clients are contacting the servers with requests, and the servers are giving responses.

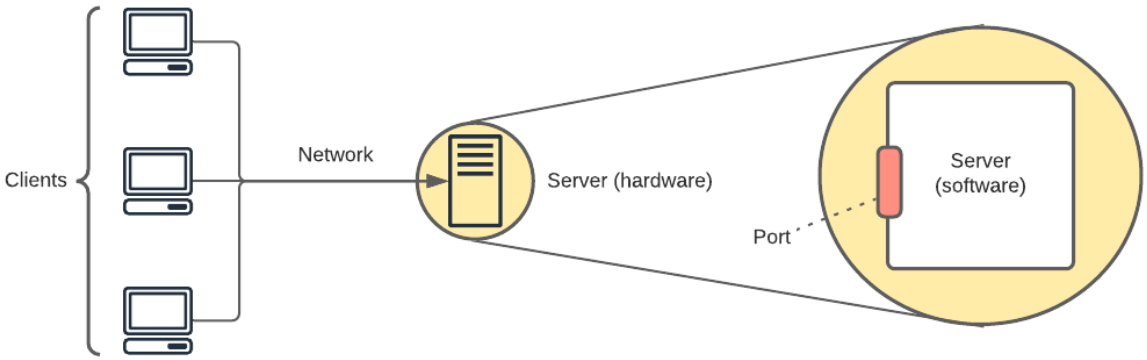


Figure 1: The client-server system. The client machines are making requests to the server machine. On the server machine, there is a software that is listening for the incoming requests on a specified port. This software manages the requests, does some operations, and returns a response to the clients.

The structure and model of the communication between the clients and servers can happen in different formats, these are called protocols. Common communication protocols over the internet include HTTP [10], HTTPS [11], WebSocket [12], FTP [13], etc.

1.4.2 Web Server

Web servers are a subset of regular servers that specialize in serving web content over the internet. They are most often reachable via the HTTP or HTTPS protocols. These types of servers were originally intended to manage static and dynamic web pages, but with technologies such as REST (Section 1.4.3), they are now serving as the basis of general machine-to-machine communication with HTTP/HTTPS.

1.4.3 REST

REST is the acronym for Representational State Transfer [14]. It is an architectural style for designing web servers. It supplies a set of constraints that specify how to reach and manipulate web resources (documents, images, objects). The resources are located by their Uniform Resource Locator (URL) [15] and retrieved or manipulated by standardized HTTP methods [16], for example, GET, POST, or DELETE. RESTful web services are stateless, which means the consecutive requests do not depend on each other.

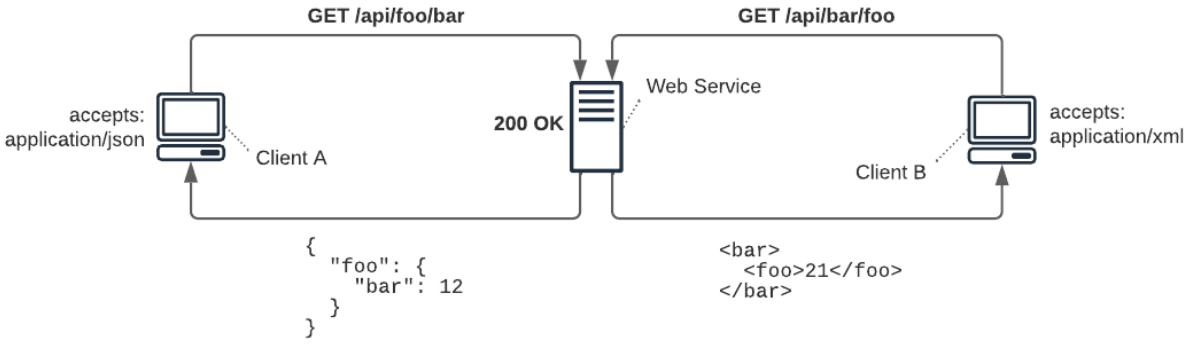


Figure 2: Two clients (Client A and Client B) interacting with a web service. The client on the left expects to receive the requested data in JSON format, while the client on the right expects it in XML format. The web service (in the middle of the figure) supports both formats, and thus the data in the response is serialized as requested.

Figure 1 shows an object retrieved from a web server in a RESTful manner: The resource is identified by its Uniform Resource Locator, the method GET implies that it is a retrieval operation without side effects, the status code 200 OK is a standard HTTP code for signaling the request was well formed and the resource is present in the system. The request contains an “accepts” header with the MIME type [17] in which the client expects to receive the requested resource.

1.4.4 Web Service

Web services are offering one or more services to their clients. The complete definition can be found in [18]. They are giving responses in machine-readable formats (such as XML or JSON), instead of serving HTML content (or other human-readable documents). They are most often built around the REST architectural style (Section 1.4.3), and thus supply a standardized communication interface to the clients.

In my Master Thesis, when I use the term “web service”, I always refer to a RESTful web service that provides a certain set of services related to managing a database (retrieve, create, modify, or delete items), and the communication protocol is HTTP with the data serialized as JSON messages.

1.4.5 Microservice Architectural Style

As opposed to a monolithic approach in which the web service components that cover the business logic typically are tightly coupled (often they reside within the same application), the microservice approach splits a web service into many fine-grained services that are loosely coupled. For example, a web service can manage only a single table in a complex database system, and another service can manage a different table, etc. Figure 3 shows a comparison between the same web service created in a monolithic approach (upper part of the figure) and using the microservice architectural style (lower part of the figure).

Both versions provide the same functionality, but the monolithic approach bundles every part related to the business model into the same application. This is going to make the system scale worse as we cannot scale the modules in the system independently. Using the microservice architectural style results in a system where the modules are more decoupled from each other, and thus it is possible to scale the system more effectively. Note that this architectural style also has a drawback: Having too many web services in the system can highly increase the system's complexity.

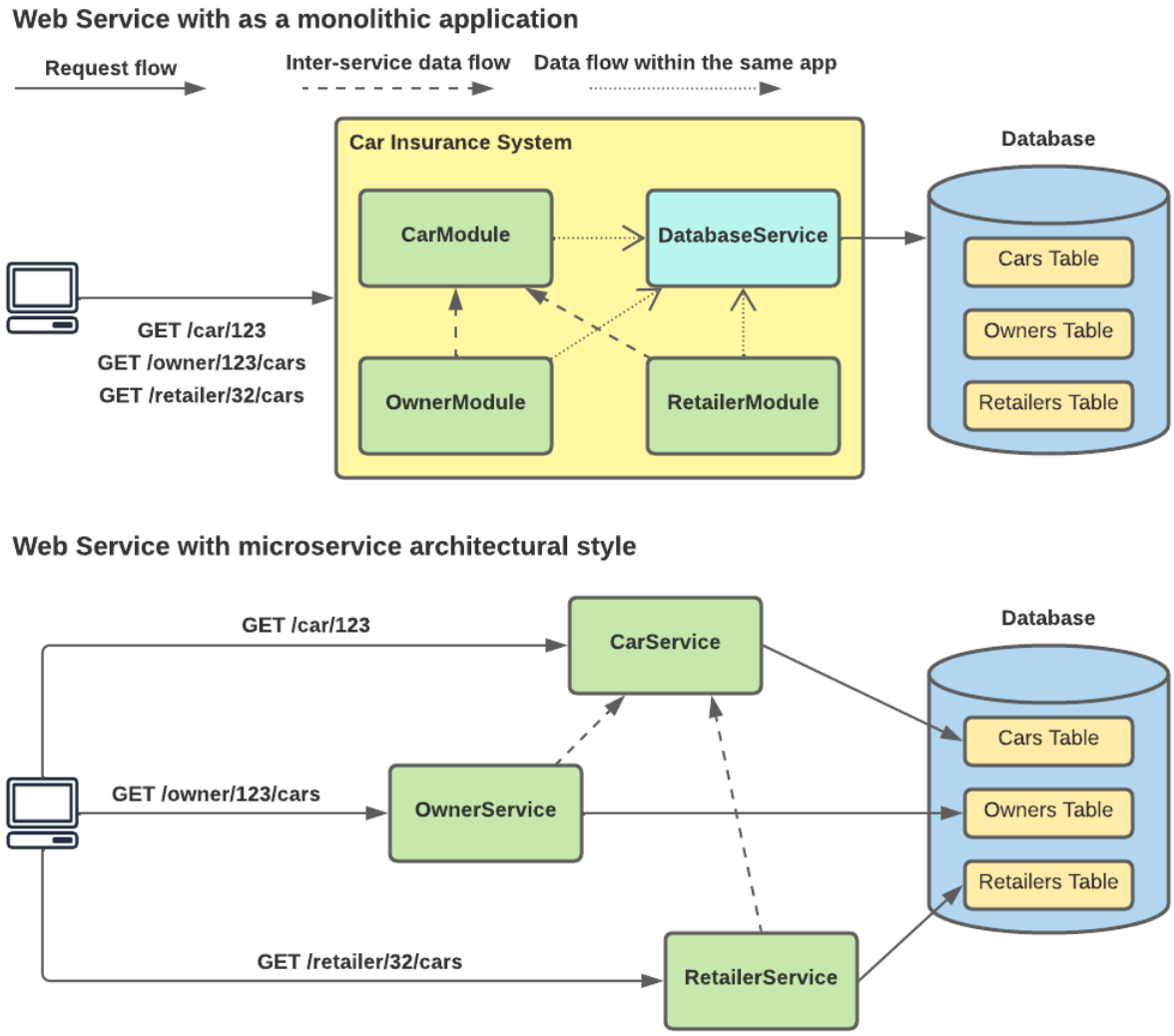


Figure 3: An example system of a car insurance company. On the upper part of the figure, we can see the system as a monolithic web service, that is, the functionality provided by the business is grouped into one application. On the lower part of the figure, we can see the same application organized using the microservice architectural style. The modules from the monolithic setup are replaced with standalone services.

Because of the smaller scope of a single web service, multiple web services need to work together in an application: They are interacting with each other if needed. The combination of multiple web services creates the functionality that covers all the business requirements. The granularity of web services in such a system depends on the architect who designs the architecture.

1.4.6 Web Frameworks and the Request/Response Lifecycle

Web frameworks are a set of web server related software components that enhance the development experience by supplying a high-level abstraction of the underlying networking

features of a given programming language. The level of abstraction varies from framework to framework.

Some examples are Django for Python [19], Ruby on Rails [20], Laravel for PHP [21], Spring for Java [22], and ExpressJS for JavaScript [23].

There are many ways to interpret the request/response lifecycle, usually the interpretation varies per web framework. In the context of this Master Thesis, my own terminology is based on the combination of the terms used in ExpressJS and Hapi for JavaScript [24].

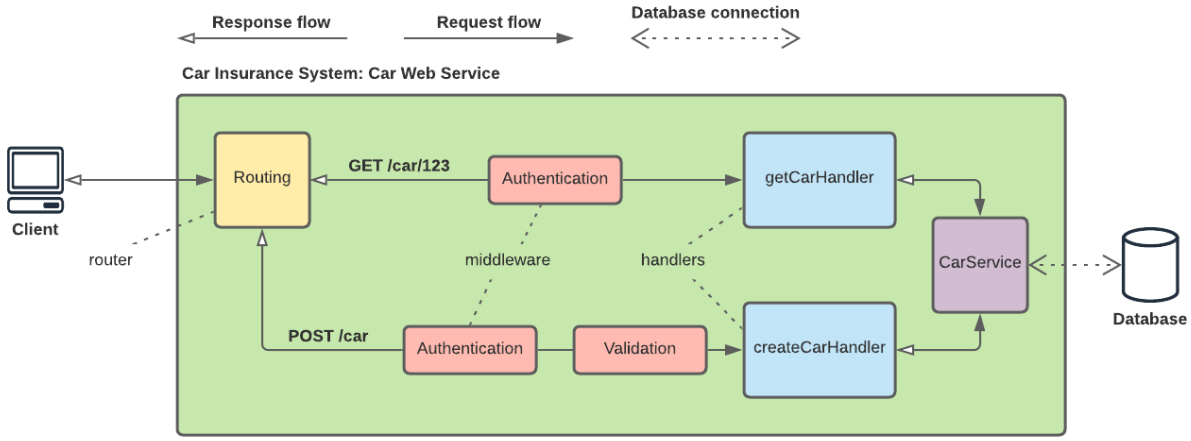


Figure 4: The request/response lifecycle of an example web service. The client makes a request that is routed to the proper handler by the routing mechanism provided by the web framework. The handler invokes the service module that communicates with the database. The response is then formed in the handler and returned to the client.

Routes are defining what the web service is capable of, what operations they provide, what data they manipulate. As soon as a request reaches the server, it is assigned to a route. The corresponding route is selected based on the Unified Resource Locator (URL) [15] of the resource, the standardized HTTP method type [16] from the client, and additional headers on the request (for example, content type [25], or authorization method [26]). After the matching route has been found, the request goes into a pipeline which is optionally built up by one or more middleware [27]. The middleware can validate, modify, extend, filter the request before it reaches the handler.

The handler interprets the input payload and invokes the related service modules that are going to communicate with databases, storage systems, API endpoints, etc. Based on the results of this communication, the handler prepares and sends the response back to the client. Figure 4 gives a visual representation of such a system. The example models a web service handling car related CRUD operations as part of a Car Insurance System.

The first part in the architecture is the routing engine, which is going to pick up the correct handler based on the Unified Resource Locator and HTTP method of the request. The routes in the example are protected by an authentication middleware (for example, using OAuth2.0 [28]). It means that only a certain set of clients have access to the system, with the appropriate permissions. The car creation route also has a validation middleware: The shape of the car object in the input payload must match a certain schema, otherwise it is going to be rejected.

Figure 5 displays the members of this object validation process. The validation engine uses a predefined object schema to validate the incoming object. If the object matches the schema (that is, required attributes of a car object defined in the schema), the object can be safely processed

in the handlers. Otherwise, the validator can reject the request, possibly with pointing out to the client that the input payload is not a valid car object (from the web service's perspective).

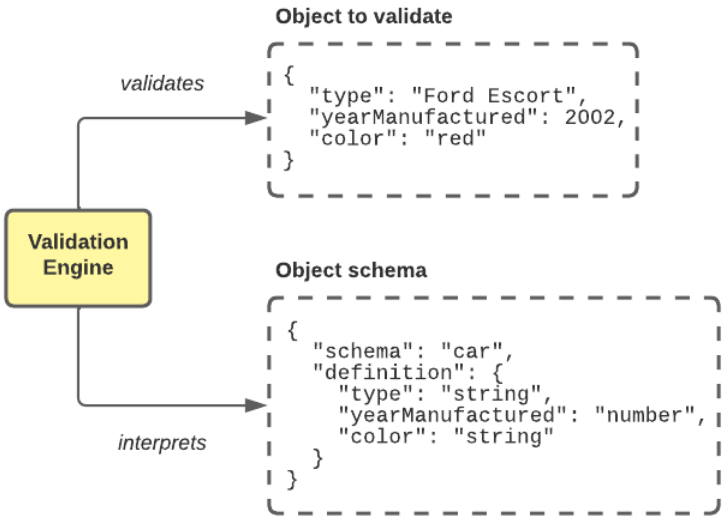


Figure 5: The members of an object (in this case: car) validation process. We have a schema that describes the shape of a certain object. The validation engine can use this schema to see if an object matches the described shape.

1.4.7 Virtual Machines and Containers

Virtual machines [4] are compute resources based on software layers instead of physical computers. These layers can be utilized to run applications like we would run them on physical hardware. The applications that run on a virtual machine execute normally, that is, it does not know that it runs in an emulated environment. We can use Virtual Machines to run multiple operating systems on the same hardware, splitting the hardware resources between them.

There are two types of virtual machines [4], process virtual machines that allow the execution of an application platform-independently (for example, the Java Virtual Machine [29]), and system virtual machines that are fully virtualized to substitutes for a physical machines. The virtual operating systems are executed and managed by a hypervisor [30].

Technologies that support the full virtualization of hardware include Oracle VM VirtualBox [31], or VMware Workstation [32].

Using virtual machines is not the only way to achieve virtualization [33]. Containers [34] are a relatively fresh way to bundle applications and run them in a virtualized environment. Large web servers, data centers, and cloud data centers are using virtualization to supply an additional layer on top of their actual hardware. This provides an efficient way to split up their machines into smaller sub-environments.

The key benefit of using containers instead of virtual machines is that they have less resource overhead [4]. Containers only include the libraries and dependencies required to run an application, while the underlying OS kernel is shared between the containers, which results in smaller virtualized environments (compared to traditional virtual machines). The full comparison between containers and virtual machines can be found in [34].

Containerization [35] is the process of specifying a manifest that describes the environment which is needed to run an application: The operating system, the application runtime, dependencies, etc.

For example, we can specify that a web server written in JavaScript needs the Linux operating system as the basis, the NodeJS runtime and a certain set of dependencies (like a web framework) to run properly. Figure 6 shows the process of creating a container based on a manifest file. The container engine [36] can create the virtual image of a container based in this file, and then it can create multiple container instances based on the virtual image. The containers are using the same kernel (for example, the Linux kernel) provided by the host machine, but they can build upon different operating system libraries (different Linux flavors).

The manifest filed is called a Dockerfile when using the container engine solution provided by Docker [37].

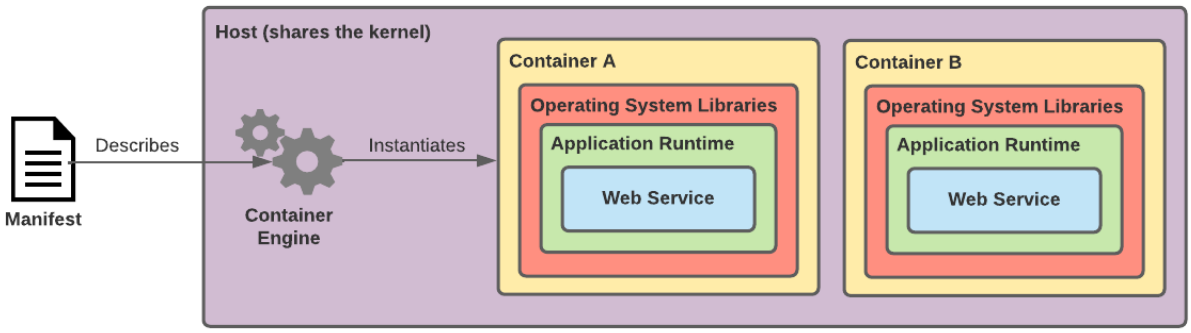


Figure 6: A container setup example. The container engine takes the manifest created by the developers which describes the resources that are needed to run the application. Then, the engine starts the containers based on the manifest file.

1.4.8 Traditionally Hosted Application

In the context of my Master Thesis, I am going to refer to applications that are containerized using a manifest file and hosted in a virtualized environment provided by a cloud provider as "traditionally hosted applications". This deployment solution is still widely used for hosting an application (not just in the cloud, but on custom servers as well), often referred to as Infrastructure-as-a-Service [5].

Example cloud services that support running containerized applications: Elastic Container Service [8] by Amazon Web Services, Container Instances [38] by Microsoft Azure, Droplets [39] by DigitalOcean. There are also Kubernetes-based hosting environments which fall into a similar category, like Elastic Kubernetes Service [40] by Amazon Web Services, or Azure Kubernetes Service [41] by Microsoft Azure. I am going to use Elastic Container Service for the traditional approach in my case study, see Section 3.4.1 for further details.

1.4.9 Serverless Computing

An innovative approach of deploying applications in the cloud is hosting them as serverless functions on a serverless platform. There are still servers involved within this concept, however, the customers of the cloud services do not have to deal with Virtual Machines, networking, and the maintenance of these resources related to their hosted application. The cloud providers take care of the infrastructure that is related to hosting the application, such as the networking resources (load balancers [42], availability zones, subnets, clusters). Example serverless platforms are Amazon Web Service's Lambda [9], Azure Functions [43], or Google Cloud Functions [44]. Throughout my Master Thesis, when I use the word "serverless", I refer to the Function-as-a-Service [2] cloud service model (Section 1.4.11).

Serverless hosting highly differs from the traditional hosting approach. The serverless functions that wrap an application are short-lived, meaning that they are created only when there is a demand to access them, and they are shut down after a brief period of inactivity. This results in a different cost model based on the serverless function's execution time. Serverless functions bring multiple advantages and disadvantages with their approach, see Section 4.2.1.2, Section 4.2.1.3, and Section 5 for a detailed discussion.

1.4.10 Backend-as-a-Service

Backend-as-a-Service [45] (BaaS) is a cloud service model, in which many cloud providers (Amazon Web Services, Microsoft Azure, Google Cloud) offer a set of backend elements as services that engineers can integrate into our applications. These elements include user authentication (like Amazon Web Service's Cognito [46]), storage (like Amazon Web Service's S3 [47]), database (like Amazon Web Service's DynamoDB [48]), message queue (like Amazon Web Service's Simple Queue Service [49]), publish/subscribe (like Amazon Web Service's Simple Notification Service [50]), etc. solutions.

An extensive discussion about the significance and advantages of using Backend-as-a-Service elements can be found in [45]. One drawback is the increased Vendor Lock-In factor with a cloud provider (from which we leverage the BaaS solutions), which is going to be further discussed in Section 2.1.1.

1.4.11 Function-as-a-Service

Function-as-a-Service [2] is a cloud service model in which engineers can organize their applications as a set of serverless functions.

Serverless functions are pieces of code with varying sizes executed on the serverless platform. The function's code is wrapped in packages and executed on-demand.

In context of my Master Thesis, when I am going to mention "function", "serverless function", "Lambda function", or simply "Lambda", I refer to an executable serverless function that is hosted on Amazon Web Service's Lambda platform [9].

1.4.12 Invoking Lambda Serverless Functions

Lambda functions are triggered and executed based on an event. Functions can listen to multiple different events that can come from various sources. For example, in the context of Amazon Web Services, when a Simple Queue Service (SQS) message queue [49] receives a message, it can send the message to a Lambda function by triggering it with the message's content. In this case, the event is called an SQS event. A complete list of Lambda invocation types can be found in [51].

The engineers can decide which invocation approach to use, and it also depends on the use case. A message queue approach might be useful for batched processing of items asynchronously, while an API Gateway [52] (explained in the following section) is useful for building web services.

1.4.12.1 The REST API Gateway

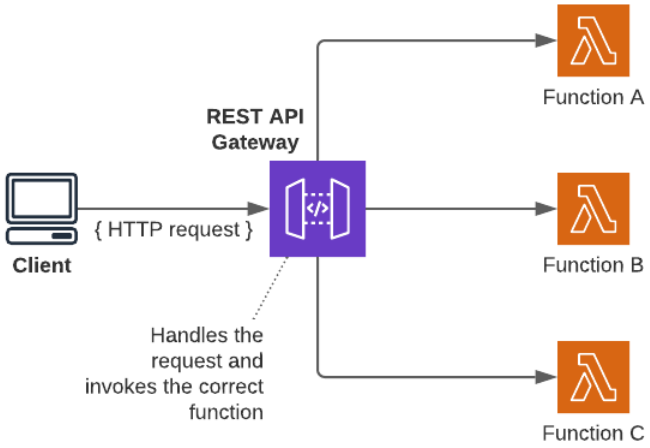


Figure 7: The REST API Gateway connected to three different Lambda functions. The client does not need to know the internal structure of the application to reach a certain function. The API Gateway takes care of the routing in this case.

Another common trigger for a Lambda function is the event from the API Gateway [52], which makes it possible to invoke a function from the internet using HTTP calls. When I mention REST API Gateways or HTTP API Gateways in this Master Thesis, I am always referring to the technologies provided by Amazon Web Services. The API Gateway service can take care of many aspects that we would need from an API that handles HTTP requests. From the official website:

"API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, CORS support, authorization and access control, throttling, monitoring, and API version management" [52].

In case of API Gateways, just as with Lambda functions, the client pays per execution (in this case, per request). We can connect multiple Lambda functions to an API Gateway, which triggers them with the request object that comes from the clients.

This variant of an API Gateway is called the REST API Gateway. It has many built-in features, for example, it is capable of request routing on its own. Figure 7 shows an example serverless architecture that uses the REST API Gateway. The requests from the clients are handled in the Gateway, which invokes the correct Lambda function (based on the HTTP method and the URL of the request).

1.4.12.2 The HTTP API Gateway

The REST API Gateway, as discussed in the previous section, has a built-in way to handle incoming requests with many unique features. However, the customers need to pay for the extra features that REST API Gateways are supplying. Amazon Web Services offers multiple types of API Gateways, not just REST API Gateways.

The HTTP API Gateways do not include as many features as the REST API Gateways [53], for example, they cannot do advanced routing for the incoming requests. They serve only as proxies, which means they forward the incoming requests to a specified Lambda function. This Lambda handles the authorization, routing, and validation of the request. Figure 8 shows the serverless architecture that uses the HTTP API Gateway connected to a Lambda function that handles the incoming request (I call it the server function).

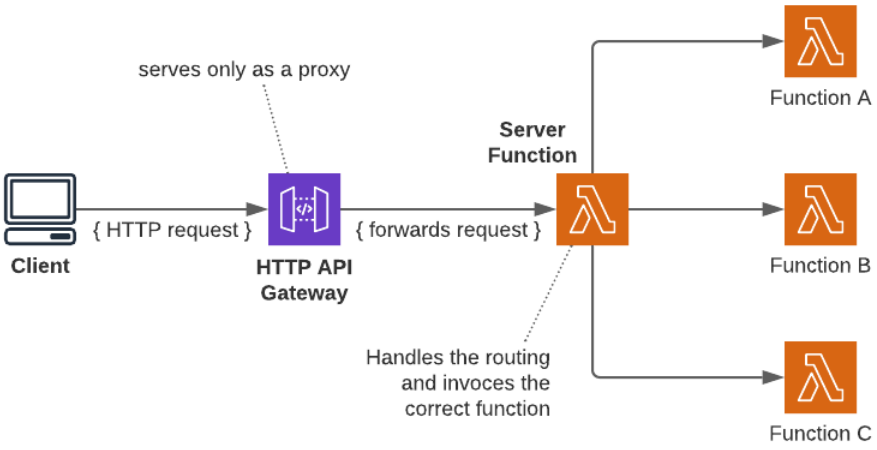


Figure 8: The HTTP API Gateway connected to a server function that handles the request/response lifecycle. The HTTP API only serves as a proxy: It supplies a way for the request and the response to travel between the client and the serverless functions.

Because of the limited functionality, HTTP API Gateways cost less than the REST API Gateways. They are designed for low-latency, cost-effective integrations with Lambda functions (and other services provided by Amazon Web Services).

The HTTP API Gateway approach is useful when migrating traditionally hosted web services to the serverless platform. There is no need to configure the routing and other details of a REST API Gateway, we can reuse the existing server capabilities of our web service, described further in my case study (Section 4.1.1).

1.5 Overview of my Case Study

In what follows, I am going to analyze the differences between two development and hosting strategies that most of the major cloud providers currently support. I call the first approach “traditional” as it was the first way to host applications in the cloud (see Section 1.1 for a brief introduction). This approach requires from developers to provision and manage cloud infrastructure resources needed to run a web service by themselves.

There are many aspects that the engineers need to think about to configure their web services in the best way possible, for example, they need to think about how many resources to allocate for a web service container (in terms of RAM and CPU), how many replicas of the web service to run concurrently to serve the traffic of the web service effectively, what types of virtual machines (which are described in Section 1.4.7) to rent in order to run the web service effectively from a cost perspective, etc.

Serverless platforms provide a way for application hosting that removes the often-tedious resource provisioning work from the engineers. Utilizing the comfortable hosting options from the serverless platforms still come with some configuration requirements (we need to setup and parameterize the serverless functions), however, these configurations are easier to comprehend. There is no need to create the underlying infrastructure that the traditional approach demands, like allocating clusters, virtual machine instances, networking resources, load balancers. Section 3.4 provides an overview of the infrastructural resources that the engineers need to configure for the traditional approach, and Section 4.4 describes the configurations required for the serverless approach.

For discovering the exact differences between the two approaches, I present a web service in my case study hosted with the two different approaches, but with the same business and functional requirements.

1.5.1 Approach of Implementing and Hosting the Web Services

The two different hosting approaches require changes in the codebase and the overall architecture of the application. My approach in the case study is that I implement and host a web service in the traditional sense on Elastic Container Service [8], then I migrate this application to the serverless Lambda platform [9].

Performing this migration helps me with understanding how much effort it takes to transform an already existing, traditionally hosted web service into a web service that runs on the serverless platform. In this sense, I consider "effort" as the following: Required changes in the implementation and hosting configurations, adjustments in the deployment pipeline.

There are two approaches for migrating the traditionally hosted application. I briefly present a minimalistic approach, in which I plan to reuse as much of the existing codebase from the traditional approach as possible, only making the smallest set of changes that is required to run the web service on the serverless platform. After this minimalistic approach, I am going to rework the application to better fit the serverless platform, that is, I am going to increase the serverless function granularity by splitting up the web service into multiple parts. I call this second approach the more realistic one. This splitting comes with advantages and disadvantages that we need to consider.

Note that the solution that I am going to present is only one way of creating and hosting a web service on the Lambda serverless platform. For example, there are multiple programming languages and deployment tools that engineers can utilize when working with serverless applications. My case study aims to give only a general idea about how to implement a web service using the Function-as-a-Service methodology.

Also, I am not covering the case of creating web services from scratch with the idea of hosting them on a serverless platform, I am solely focusing on a migration scenario. The solution that I present here is one viable option, but as of today, cloud providers regularly add new features to their serverless platforms based on previous experiences, which can supply new options/improvements for hosting an application.

1.5.2 The Selected Cloud Provider

I have decided to use the cloud platforms provided by Amazon Web Services for my case study. It supplies all the different types of services that I plan to use in the case study. I am going to host the web service on Elastic Container Service [8] as part of the traditional approach and on Lambda [9] as part of the serverless approach.

1.5.3 The Modelled Web Service

For the case study, I have decided to create a small web service that performs CRUD (create, read, update, and delete) operations on a DynamoDB [48] table. The objects stored in the database are going to be shapes (rectangles, triangles, and circles) with three varied sizes (small, medium, and large) and arbitrary colors (expressed in a hexadecimal format). Figure 9 visualizes a sample set of shapes that the clients can manage by using the shape web service.

Figure 9: Example shapes managed with the web service that I have created for my case study. Left: large, circle, #0062ff. Center: small, rectangle, #ac0712. Right: medium, triangle, #69f20d.

The web service is RESTful (see Section 1.4.3 for a brief introduction), which means that clients can perform the CRUD operations by using standardized HTTP methods, such as POST for creating a shape, GET for retrieving a shape or shapes, PUT to update a shape, and DELETE to remove a shape from the database.

The responses also correspond to the standardized HTTP status codes, for example, 201 Created on a successful shape creation, 400 Bad Request when the client's input payload is invalid, 502 Bad Gateway when there is a communication error with the database, etc. The communication format of the client and the server is application/json.

```
{
  "id": "abbc9fa9-2676-45fc-a879-a52c3a8a49ec",
  "size": "large",
  "color": "#ff00ff",
  "type": "circle"
}
```

Listing 1: An example shape object stored in a JSON format.

Each endpoint of the service is going to be public. In a production grade system, this service could be internal, protected by some form of authorization (like OAuth2.0 [28]). To keep the case study simple, no authorization mechanism is present in the system.

The payload size is going to be small in most cases, that is, we do not have to move large amounts of data over the network since the shape objects are relatively small (approximately 70 bytes). The shapes are stored in a JSON format, Listing 1 shows an example. Note that this influences the evaluations presented in the Thesis, as different payload sizes have different effects on the traditional and serverless hosting approaches. Refer to [54] for a detailed investigation.

1.6 Technologies Used

In this section, I want to list all the services and technologies that I am going to use and refer to throughout the rest of my Master Thesis.

Table 1 shows the list of the cloud services and resources used for hosting my web services, storing the shape objects that the clients create, etc. Table 2 shows the software related resources used while I was implementing the shape web services. It contains references frameworks, most important third-party libraries, and the programming language utilized for the service implementations. Table 3 provides a list of the software tools that I am going to use for various reasons (code deployment, web service evaluations).

Note that the technologies are just listed in these tables, they are going to be further discussed in the sections where I am going to first introduce them.

Name	What I am using it for	Reference
Amazon EC2	Provisioning virtual machine instances for running my traditionally hosted shape web service	[55]
Amazon Elastic Container Service	Executing the containerized shape web service from the traditional approach	[8]
AWS Lambda	Executing the serverless variant of the shape web service (Function-as-a-Service platform)	[9]
Amazon DynamoDB	Storing the shapes that the shape web service handles in NoSQL tables	[48]
AWS CloudFormation	Writing and deploying cloud resources required to my applications using the Infrastructure-as-Code approach	[56]
Amazon API Gateway	Enabling my serverless Lambda functions to be reachable from the internet	[52]
Amazon Elastic Container Registry	Storing the containerized shape web service from the traditional approach (Docker image repository)	[57]
Elastic Load Balancing	Distributing the incoming traffic between multiple shape web service instances in the traditional approach	[58]
Vercel	Hosting the user facing application that I implement as part of my user experience evaluation	[59]

Table 1: The list of different cloud services/resources used as part of my Master Thesis.

Name	What I am using it for	Reference
ExpressJS	Implementing my web services in both the traditional and serverless variants (web framework)	[23]
NextJS	Implementing the frontend application used in the user experience evaluation	[60]
TypeScript	Writing the web services' implementation in both approach (programming language)	[61]
serverless-http	Making the shape web service from the traditional approach executable as a serverless function	[62]

Table 2:The list of the different programming languages, software frameworks and most important third-part libraries used as part of my Master Thesis.

Name	What I am using it for	Reference
AWS CLI	Deploying the cloud resources based on the YAML template definitions in the traditional approach	[63]
SAM CLI	Deploying the cloud resources based on the YAML template definitions in the serverless approach	[64]
Docker	Containerizing the shape web service in the traditional approach	[37]
curl	Measuring the response times from the shape web services during the evaluation	[65]
cloc	Measuring the Lines of Code and number of files of the shape web services during the evaluation	[66]

Table 3: The list of the tools used as part of my Master Thesis.

2 Related Work

In the following section, I am going to present my analysis about related studies and research papers that connect to my Master Thesis in some way.

2.1 Choosing The Cloud Provider and The Programming Language

In this section, I am going to investigate the vendor lock-in phenomenon that makes it difficult to migrate our applications between different cloud providers after we have opted for one of them, then see which cloud provider and programming language is the most popular among developers.

2.1.1 Vendor Lock-In

Since multiple cloud providers have adopted and created their own serverless models, it can be difficult to choose one that suits our needs in the best way possible. The different interpretations of serverless and Function-as-a-Service (Section 1.4.11) are quite similar across all the cloud providers, but there can be subtle differences. The engineers need to choose the provider carefully because of the vendor lock-in issue when using serverless platforms. Vendor lock-in means that the application code depends on details that are related to a specific vendor, for example, Amazon Web Services' Lambda platform has a certain structure of incoming event and context objects, while Google's Cloud Functions have a different one. Thus, engineers need to adjust the code if they want to deploy their application on a different provider's platform.

As part of their serverless ecosystem, cloud providers supply other Backend-as-a-Service (Section 1.4.10) elements that the serverless applications can use and integrate seamlessly. These services are vital parts of serverless applications. The vendor lock-in issue is much larger when using the Function-as-a-Service and Backend-as-a-Service solutions combined from a specific provider. The study in [67] proves that engineers can face dead ends when conducting a migration: If a certain set of Backend-as-a-Service elements are not present with a certain provider, then the migration attempt can result in rewriting the application instead.

The vendor lock-in issue is well analyzed in [67]. In this study, the researchers investigate the migration of four use cases between multiple cloud providers. First, they develop applications for the use cases that they defined. The use cases include many Backend-as-a-Service options to prove that these services are highly affecting the vendor lock-in problem.

The first use case is about a Thumbnail Generation tool, which creates a thumbnail for every uploaded image in a tests system and stores them in a separate storage space.

The second use case is a RESTful web service that supports managing a To-do list. The setup has five serverless functions, each doing a specific task, like To-do creation, update, retrieve, etc. The To-dos are stored in a NoSQL database.

The third use case describes an Event Processing scenario, where the authors simulate sensor input values that need to be aggregated and processed in multiple steps. The architecture includes message queues and a Relational Database Management System.

The last use case is a function orchestration for Matrix Multiplication. The process is done with many serverless functions combined. The function execution is orchestrated by a serverless function orchestrating solution, for example, Step Functions [68] in Amazon Web Services or Durable [69] Functions in Azure.

Initially, the authors host the applications in Amazon Web Services, and then they are porting them to Microsoft Azure and IBM Cloud while documenting the issues they face during the migration process. They assess four main points to consider when doing a migration, like the required changes in the implementation language, or the need of adapting the function code or any of the configurations. They also investigate if it is needed to change the application's architecture, which leads to redesigning the major parts of the application which is being migrated.

The findings prove that the nature of serverless computing amplifies the vendor lock-in issue, since all components (including Backend-as-a-Service elements) are managed by the providers, resulting in multiple dimensions of the vendor lock-in problem.

2.1.2 Research on the Most Popular Programming Language for Serverless

Choosing the most suitable programming language for a given application can be difficult. Compared to traditional hosting in the cloud, where it is possible to use any programming language to create an application (given the flexibility of containerization, which is described in Section 1.4.7), cloud providers only support a certain set of programming languages to use natively for serverless applications.

There are solutions to use any kind of programming language with certain providers, for example, Amazon Web Services supplies an API called the Runtime API [70] which enables engineers to use any arbitrary programming language for their serverless application. However, in such cases there is some overhead that comes with configuring the Runtime API. This goes against one of the advantages of serverless platforms, because the engineers need to do customization that the cloud provider otherwise would cover with the natively supported languages.

A good approach for picking a programming language for a serverless project is to see which language used for serverless functions has the biggest community. A general rule of thumb in Computer Science I have seen so far is that technologies with larger communities tend to outlive the others.

Based on this approach, I consulted [71] to get a better overview about current trends in the field. This study evaluates a huge amount of serverless projects to see how engineers built them, which platform they used. The researchers collected eighty-nine serverless application and investigated their setup: Which programming languages and which cloud providers they are based on, and the trends in using certain Backend-as-a-Service elements (storage, messaging, database).

The study shows that the two most popular programming languages for writing serverless functions are JavaScript (42%) and Python (42%). Every major provider supports these programming languages natively. Based on personal experience, I decided to use JavaScript for the applications in my case study. Figure 10 shows the outcome in [71], showing the most popular deployment platforms and programming languages in percentages.

2.1.3 Research on the Most Popular Cloud Provider

When deciding about the cloud provider for my project, there are multiple effective ways to come to a decision. We can make differences between the services they offer (like the range of Backend-as-a-Service solutions), or we can use the approach presented in Section 2.1.2: See which provider is the most popular among engineers, based on the eighty-nine evaluated services in [71].

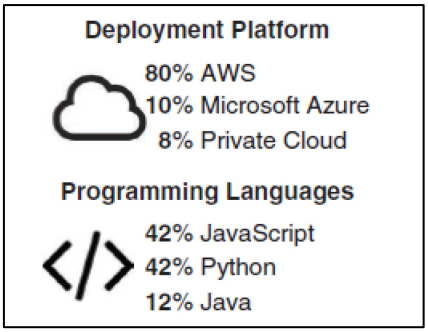


Figure 10: The results of the analyzed 89 applications in [71].

The study has found that Amazon Web Services is by far the most popular platform; I can clearly see that by the numbers. Eighty percent of the observed projects use Amazon Web Services as their cloud provider. Microsoft Azure takes the second place with ten percent. The gap between the first and second place is enormous.

The conclusion is that choosing from these programming languages and selecting the most popular provider will guarantee that if I run into any problems during application development, then I am going to get some form of help from the community members. This can heavily reduce the engineering hours needed to develop a solution, since I do not need to reinvent the wheel each time I face a new problem set during the development process.

2.2 Guidelines for Designing Serverless Applications

In this section, I explore the existing guidelines and programming practices related to the serverless application model.

2.2.1 Discoveries on Serverless Programming Practices

Since the introduction of the serverless model in 2014, engineers are using the serverless platform increasingly to host their applications [71]. Experimenting with this new way of designing and deploying applications yields some patterns and guidelines. The study in [7] raises awareness about a few common bad practices that engineers should avoid when designing their serverless applications. It looks at problems from the software design perspective, giving guidelines about how to design a serverless application's architecture. The guidelines point out different scenarios that can lead to problems, and they give advice on how to avoid them.

The first observation is that asynchronous calls between serverless functions can highly increase the system's complexity. The suggested solution is to use synchronous communication when possible. It is also a problem when functions are calling each other, creating a chain of dependency between each other, which is going to inevitably result in increased costs (when the chain is executed synchronously) (further discussed in Section 4.2.1.2). Avoiding long function chains can resolve this issue.

Shared code is an effective way to remove duplication in the functions, but too much shared code might increase the image size in each of the relevant functions, which will result in longer cold start times. The study suggests writing independent and decoupled functions. Note that using too many external libraries and adopting too many different technologies also increases the image size.

Creating too many serverless functions makes the system hard to maintain. People tend to create new functions instead of reusing existing ones by extending their functionality (since there is no cost penalty for creating new functions). Always consider if there is a need for creating a new function. Group functions together to form complete web services.

The study [7] even discusses some general open issues related to the serverless application design:

- Engineers lack a good understanding of the event-driven paradigm. This paradigm focuses on the data flow in the system and envisions the architecture of an application as a pipeline with multiple stages in the pipe. Based on the guidelines presented in the study, we can conclude that keeping our serverless functions as small as possible comes with many advantages. Combining multiple small functions to form a data pipeline is a clever way to design our systems.
- Confusion between serverless functions and the microservice paradigm. The microservice paradigm is a collection of design patterns, practices about how to design a web service's architecture, while a serverless function is a way to run an application in the cloud. A web service can be composed of one or more functions, but a single function should not be confused with a web service. Engineers can create a "serverless, microservice approach-based web service" by designing and implementing a web service and then hosting it as a combination of serverless functions. Engineers should not confuse an application with a hosting approach.
- Integration and functional testing (see [72]) are difficult with serverless functions since we need to run the functions locally to replicate the actual behavior of the system. Executing multiple functions to test their interaction and mocking the Backend-as-a-Service resources that they are interacting with is complex, also demanding from a hardware perspective.

Based on my personal observations, the tools for productivity and testing are getting better and better each day, the community is solving these issues rapidly.

The guidelines presented in [7] are valid and we should build applications keeping the authors' findings and recommendations in mind. However, these guidelines were describing issues of Function-as-a-Service in general. My Master Thesis is going to discuss aspects to consider when implementing web services specifically as a combination of serverless functions.

2.3 Studies on Differences Between the Traditional and Serverless Approach

In the following section, I am going to present a study on how the response times of a web service are affected by the cold start latency of a serverless function. Also, I am going to look if there are any tools or libraries that can aid the migration of traditionally hosted applications to the serverless platform.

2.3.1 Response Time Measurements

The study [54] does not have a guideline about how to migrate traditionally hosted applications to the serverless platform, but it is a great investigation about the response times and scalability differences of a web service deployed with both traditional and serverless approaches.

The paper takes an employee time-sheet management portal developed in JavaScript. This portal has a web service part that supports CRUD operations in a RESTful way. This is the type of service

that my Master Thesis also investigates. The authors take two hosting strategies, a) the web service in a self-provisioned Virtual Machine (Section 1.4.7), b) the web service as a combination of multiple serverless functions (Section 1.4.11).

For response time and scalability measurements, the study uses the popular k6 [73] load testing software. It has a sophisticated way for defining input payloads, and thus it is possible to pressure test the web service in a way that mimics how the real clients would use it.

The tests and measurements show that there is no general way to decide about where to deploy an application. Both the traditional and the serverless approaches have upsides and downsides. Traditional containers tend to scale out slowly and thus they often cannot account for spikes in the traffic properly. Serverless containers can scale out quickly (measured in seconds) compared to traditional containers (that can have minutes to scale out), but they are performing worse when managing regular traffic patterns, because they are short lived, and engineers need to expect the negative effect cold start latencies in the response times regularly. Figure 11 shows a comparison of response times between the serverless and traditional approach under different types of load patterns.

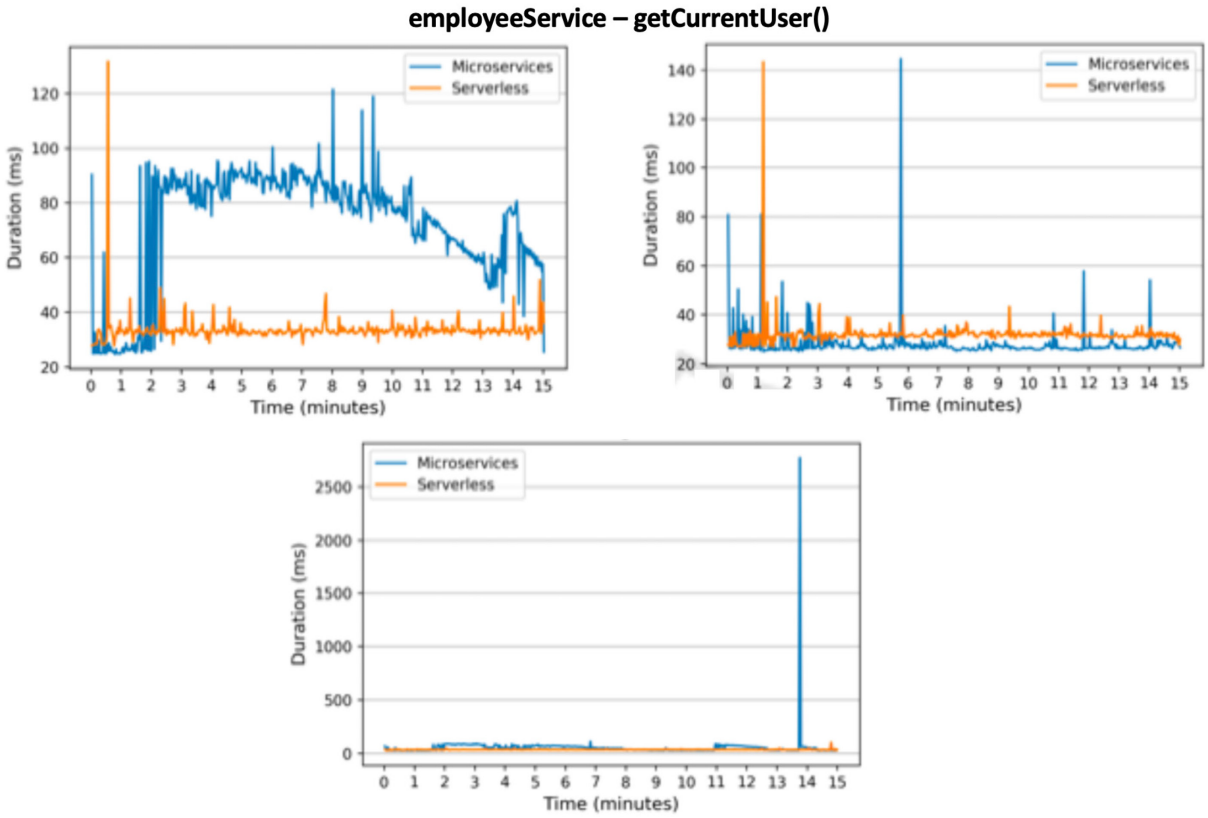


Figure 11: Example timings for a certain function in [54]. Blue: Microservice performance. Orange: Serverless performance. The function is called with three different workload distributions as part of the load testing. The upper left chart shows increasing the load incrementally. The upper right chart shows random pressure increases. The chart on the bottom shows the triangle workload balancing (continuously increasing the load till half time and then continuously decreasing for the remaining time).

In the end, [54] has some key observations and conclusions, such as the response times of an application hosted with the serverless strategy suffers from the cold start latency issue. Self-provisionally hosted strategy, which means that the application is executed in a container on a platform like Amazon Web Services' Elastic Container Service [8], suffers from load balancing and

traffic redistribution problems; however, an advantage is that the self-provisioned variant outperforms the serverless strategy when fetching small-sized objects in repetitive requests. In contrast, serverless deployment is more agile in terms of scalability.

2.3.2 Programming Tools to Aid the Migration

The book [74] gives multiple options and tools that I can leverage when performing the migration from a traditionally hosted application to an application that is hosted on the Lambda platform [9] (in JavaScript).

The book points out solutions for taking any regular JavaScript-based web services (using any of the available and popular HTTP frameworks, like ExpressJS) and converting them into applications that can run as serverless functions. This makes the migration convenient, because I can convert certain web services quickly without changing the original architecture and code base too much. I can keep the existing routing, controllers, schema validations, middleware (see Section 1.4.4 for a description of these elements), and so on. I only need to wrap the original web service into a module like serverless-http [62]. My Master Thesis is going to further describe this process in Section 4.1.1.

It is a crucial factor that existing web servers can be transformed quickly so they become deployable to serverless platforms. It raises the confidence in engineers that they can create serverless web services easily. It also makes fast prototyping possible: Companies can quickly migrate their existing solutions to the serverless platform. If it does not work out because of the increased response times or undesired effects, the engineers can drop the migrated prototype without losing too much work.

2.4 Current Shortcomings of Serverless Computing

In this section, I am going to present studies that investigate how increasing the memory of a serverless function can improve its cold start time, which will consequently result in better response times from web services hosted on the serverless platform. Also, I am going discuss some of the shortcomings of current serverless platforms, grouped by various aspects (like storage options or performance).

2.4.1 Increasing Container Memory

When I am considering web services from a client's perspective, one important metric is the average response time of the web service. Especially when running synchronous operations, that is, when the client invokes an operation of the web service and waits for the results. Even if I consider non time critical applications, I expect my dataflow to be fluent with as minimal delays as possible. As discussed in Section 2.3.1, [54] confirms that cold starts can have a negative effect on response times, but it does not suggest how to improve the latency of serverless functions.

Data that comes slowly when requested can make user-facing applications unresponsive. Responsive-ness is a key metric in such applications. It is not a convenient feature anymore; it is a must-have [75].

In cloud computing, elasticity [76] refers to the dynamic adaptation of capacity to meet the varying resource needs of the clients, for example, by altering the use of cloud computing resources. In case of web services hosted in the cloud (running inside containers), when the number of incoming requests is increasing to a point where the existing number of containers of a web service cannot handle the load properly, the container manager must act and scale out the application by starting

new containers. Also, when the number of requests is decreasing, the additionally created containers can be shut down. This mechanism keeps us from the under- or overprovisioning of cloud resources.

The time for a new cloud resource to become available is called the elasticity factor. As discussed in [77], to find the elasticity factor of a serverless container, engineers need to consider three types of cold starts:

- Provider cold: The very first invocation for a given function (re)deployed in the cloud
- VM cold: The very first invocation of a virtual machine (than can run multiple functions)
- Container cold: The very first invocation made to a container hosting the function code

These cold start times of serverless functions can heavily affect the response time of a web service hosted on the serverless platform. My Master Thesis is going to investigate if I can effectively hide the increased response times of a web service in the client facing applications, such as web, desktop, or mobile apps (refer to Section 5.2). If not, then these apps may become stuttering when talking to web services running on a serverless platform.

Hiding/handling the longer response times is only one way minimize the negative effect of cold start times. As investigated in [77], increasing the memory of an Amazon Web Services Lambda function helps to decrease the cold start time of a function. The lowest available memory configuration is 128MB, it causes a massive 6000ms cold start time on average. The highest memory configuration included in the study is 1536MB, which results in approx. 1700ms cold start time on average. It shows that increasing the memory is a workable solution if the goal is to reduce the response times of a web service, which is, again, influenced by the cold start time of a function. Based on my own evaluation in Section 5.1, I can see that increasing the RAM can also improve the subsequent response times as well (when the serverless functions are in a warm state), although the impact is not as great as with differences that we can see in the cold start times.

Note that the costs of running the serverless functions are going to be higher if I choose larger memory options for them. If the I would decide to increase the memory too much, predictive calculations may show that using an efficient, refined traditional hosting setup can yield smaller costs.

2.4.2 Redesigning the Serverless Model

Another interesting study about improving on cold start latencies is [78]. It focuses on an innovative design of Function-as-a-Service that has a latency in the microseconds range. Their solution would reimagine how serverless functions are working on the lowest level. They aim to restructure the serverless model with low latency as the core metric.

2.4.3 The Berkley View on Serverless Shortcomings

In what follows, I discuss major findings of the well-known Berkeley View on Serverless Computing [1].

2.4.3.1 Inability to Share State between Functions

Cold start times are not the only limiting factors in current serverless applications. [1] has a great collection of issues that may have negative effects on an application or require modifications in the business logic to counterweigh these shortcomings.

Serverless functions are short lived and there is no guarantee that two consecutive executions are going to reuse the same host, and thus they cannot rely on any data hosted in the function used for their execution.

As part of their Backend-as-a-Service lineup, many cloud providers offer different storage services. [1] points out that no of them is perfectly suited to be used for sharing state between serverless functions. Object storage services (like Amazon S3 [47]) are highly scalable, and they are inexpensive for storing objects in the long term, but access costs are too high, and they can have high latency when accessing the data stored in them.

Key-value databases (like Amazon DynamoDB [48]) supply high input/output operations per second (IOPS), but they are not great in terms of scale up speed and they are expensive.

The complete list of the current service lineup and their shortcomings can be found in [1]. The key point is that we are missing a service that is accessible by cloud functions, has transparent provisioning (scales well), has high availability, and a high number of input/output operations per second.

2.4.3.2 Difficulties with Predicting Performance

The study [1] also mentions that it is hard to predict the performance of the serverless functions. This not only includes the unpredictability of cold start latencies, but the fact that choosing the underlying hardware for running a function is decided solely by the cloud provider. A serverless function may run on older hardware which can have a performance impact that can affect the performance predictions negatively.

A possible solution could be if the cloud providers would let the clients choose from different "serverless function tiers". Specifying the exact hardware would neutralize one of the advantages of the serverless platform: The cloud provider deals with the provisioning; we do not need to think about where to host our function. However, if it would be possible, for example, to mark a serverless function for high performance computing or machine learning, the serverless function scheduler could take this into account and choose a suitable hardware for the task at hand.

2.5 Summary of the Related Studies

For deciding about the cloud provider and the programming language of a given serverless function in my case study, I have chosen the approach of using the most popular choices from the community. [71] supplies a study about current trends in serverless development that helps me choosing the best technologies available. Related to these questions, I have investigated the vendor lock-in issues with the help of the study in [67]. Then, I have presented a collection of guidelines in [7], which is a must-have reading for any engineers who plan to use Function-as-a-Service for their application.

I have consulted studies on migrating applications from traditionally hosted environments to the serverless platform, which is a key point of this Master Thesis. I presented that [54] makes a good approach on assessing the same web service deployed on the two different platforms, the traditional self-provisioned and the serverless one. The study concluded that the response time measurements with the k6 pressure testing tool are useful for seeing the characteristics of traditionally hosted and serverless applications under heavy load. The study confirms that every use case is different, there is no general rule about choosing between the two deployment approaches.

I have also seen that the book [74] suggests some useful libraries that I can use for hosting web services on the serverless platform (the examples are for Lambda platform using JavaScript).

As the last part of the section, I have consulted studies on improving the shortcomings of the current serverless models. In the study from [77], the researchers found out that memory configuration of a function has a significant impact on the cold start latency times. I have briefly mentioned that [78] suggests a new model for Function-as-a-Service that would make reduce the cold start latencies to microseconds. Other shortcomings were presented in [1]. This study points out that the serverless functions of today cannot share state between each other without using additional Backend-as-a-Service solutions, and predicting their performance is difficult since we do not know on which hardware the functions are going to run. I have supplied some suggestions to improve on these points.

3 Traditional Approach

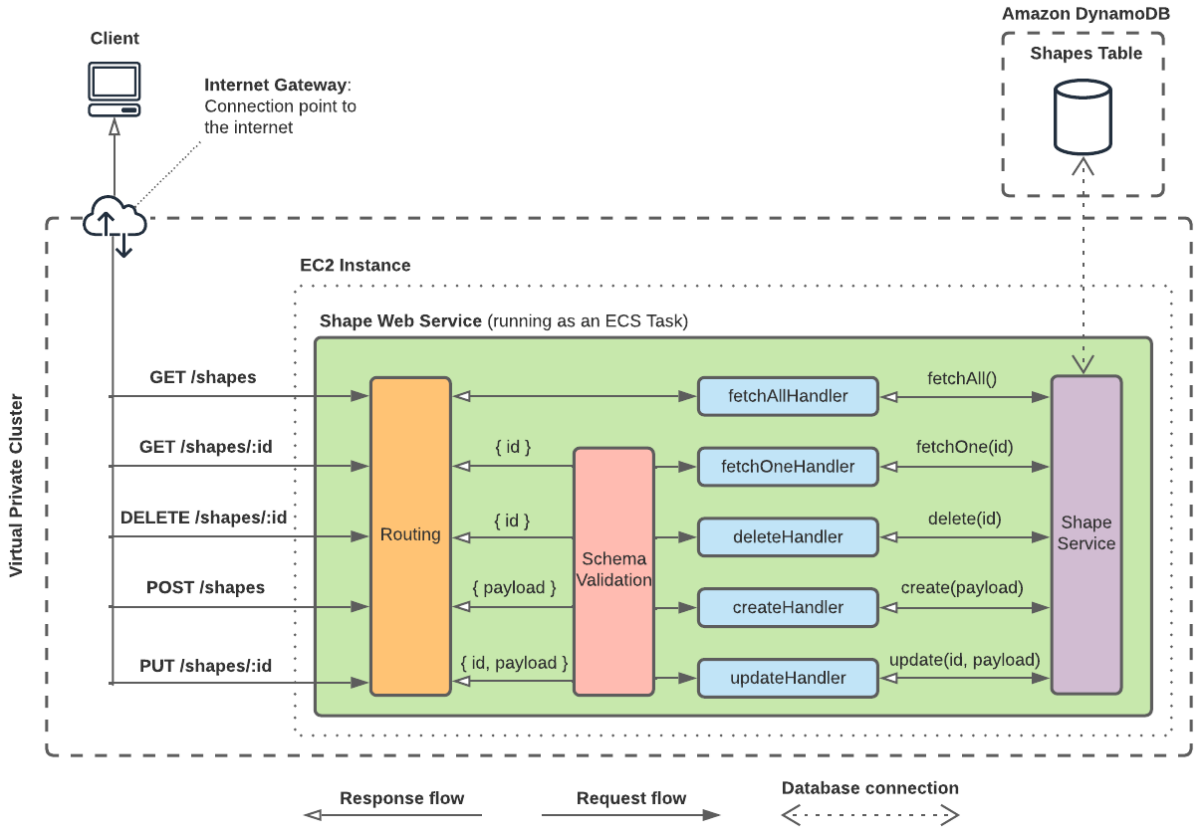


Figure 12: The architecture of the Shape Web Service. The server manages the routing, it forwards the requests to the proper handlers. In some cases, there is a schema validation step involved. The schema validator can validate identifiers (UUIDv4 format strings) and payloads (Shape dimensions: type, size, color). The handlers are using the Shape Service that connects to the database. This flow is described in Section 1.4.6.

3.1 Overview

In this section, I am going to create the web service that provides the functionality presented in Section 1.5. The web service is going to be written in TypeScript [61] and it is going to use the ExpressJS framework [23] which can be leveraged to create web services effectively. My plan is to host the service on the Elastic Container Service [8] provided by Amazon Web Services, which

requires containerizing and publishing the container image of the application before it can be used on the platform.

3.2 Architecture

The traditionally hosted web service is based on a server that listens for incoming requests on a specified port (refer to the technical term introduction in Section 1.4.2 and Section 1.4.4). The requests are connected to their related handlers by using the routing tools provided by the server's web framework (ExpressJS in this case). Before giving the work to the handlers, the server performs the input payload validation on a route level. For example, the server only invokes the creation handler if the request has a valid shape input object. This flow is presented in detail in Section 1.4.6.

The handlers are interacting with the service module which is communicating with the database. Based on the type of the request and the status/outcome of the invoked operation, the response is created and sent back to the client.

Figure 12 shows a visualization of the elements of this architecture and the connections between them. The different parts of this architecture are going to be discussed in the following sections.

3.3 Implementation

In this section, I am going to present the resources that I have created regarding to the traditionally hosted web service. As mentioned in Section 2.1.2, I have originally chosen the JavaScript programming language for writing the codebase in my case study, however, I converted the projects to TypeScript [61] in a later step. TypeScript provides type support for a JavaScript project, which is an important feature for static code checks. Describing TypeScript in detail is out of scope of this Master Thesis, refer to [61] for additional details on the programming language. I am going to introduce only the topics that are necessary to understand the application structures presented in the upcoming sections.

Note that TypeScript is transpiled to JavaScript after compilation. TypeScript is a subset of JavaScript which enhances the language by adding type support. When I refer to the "compilation step" in the following sections, I refer to the process of translating TypeScript code into JavaScript.

3.3.1 Organizing the Code in TypeScript

The code pieces are built-up by modules in TypeScript. It is a bit different from what is seen in strictly object-oriented languages like Java [79] or C# [80]. It is possible to create classes in TypeScript to mimic the behavior of object-oriented languages, but programmers have more freedom in designing their functions, data structures, etc.

Modules can have variables and functions that are only local to the certain modules themselves, but they can be exported by using the export keyword in TypeScript. After marked with the export keyword, the exported variables, functions, and data structures from the modules can be imported and used in other modules, like what programmers would expect from the package system in the Java language.

Let us consider a small example in TypeScript: I want to build a tiny command-line interface software that accepts a sentence as an input parameter, then translates this sentence to another language. The entry point of this application is a function that is invoked when the application is started. Then, while the application is running, it invokes several other functions to carry out its

work, for example, a handler function from an “input” module to parse the input arguments, a translate function from a module called “translator”, etc. Note that I am not describing this process in terms of objects: I am not creating an input handler object that has the function for translating an object, and I am not creating a translator object that has a translate method to translate something. I am talking in terms of functions; it does not matter where they come from. This is a key difference between the two approaches, and it requires a change in my mindset about how I organize my codebases.

This way of looking at my code fits perfectly with serverless functions. I can specify a top-level module that exports a function which is going to be the entry point of a serverless function. When an application is combined by functions, it only needs an appropriate template (discussed in Section 4.4), then it is ready to be deployed on a serverless platform like Lambda [9].

3.3.2 Application Structure

After I have discussed what modules are in TypeScript, I can present the code structure for the traditional approach, shown in Figure 13. The src folder contains the actual code of the web service implementation, the other folders and files are used in managing the cloud infrastructure, the TypeScript compiler, etc. Note that every file included in this figure was created by me. I have not used any code generation tools or other aids. The only exceptions are the files in the types folder, which is going to be discussed later in this section.

```
TRADITIONAL APPROACH CODEBASE I.
├── aws
│   └── templates
│       ├── infrastructure.yml
│       └── service.yml
└── scripts
    ├── create_and_push_image.js
    ├── create_and_push_image.sh
    ├── create_app.js
    ├── create_app.sh
    ├── create_infrastructure.js
    ├── create_infrastructure.sh
    ├── create_repository.js
    ├── create_repository.sh
    ├── delete_app.js
    ├── delete_app.sh
    ├── delete_infrastructure.js
    ├── delete_infrastructure.sh
    ├── delete_repository.js
    ├── delete_repository.sh
    └── typegen.ts
```

```
TRADITIONAL APPROACH CODEBASE II.
├── src
│   ├── clients
│   │   └── dynamoClient.ts
│   ├── config
│   │   ├── default.ts
│   │   └── types.ts
│   ├── handlers
│   │   └── index.ts
│   ├── services
│   │   └── index.ts
│   ├── routes
│   │   └── index.ts
│   ├── schemas
│   │   ├── IdentifierSchema.ts
│   │   └── ShapeSchema.ts
│   ├── types
│   │   ├── Shape.ts
│   │   └── index.ts
│   ├── index.ts
│   └── server.ts
├── Dockerfile
├── LICENSE
├── README.md
├── tsconfig.json
├── package.json
└── yarn.lock
```

Figure 13: Folder structure of the codebase for the traditional approach. I have split the view in two to save space. The aws and src folders are on the same level. Every file listed on the figure was created by me.

The aws folder contains the templates that serve as the input for the tools that can manage my cloud infrastructure (specifically for this web service). For example, the infrastructure.yml file describes in YAML [81] the cloud resources that are needed to host the web service as a task in in Elastic Container Service [8]: Clusters, instances, subnets, load balancers, etc. Section 3.4 describes these resources in more detail. This template file can be then used with a tool like AWS CLI [63], which is a command-line tool that has the capabilities to manage cloud resources. Describing cloud resources with template files is a major part of the Infrastructure-as-Code approach [82]. The technology provided by Amazon Web Services that handles Infrastructure-as-Code templates is

called CloudFormation [56]. When I refer to templates, I always refer to YAML files that contain a set of resource configurations, and these files can be processed by CloudFormation.

The scripts folder contains script files that help me with certain operations that I want to execute during application deployment. The operations include creating and pushing the containerized application to the cloud, creating/removing the networking infrastructure connected to the web service, or the application stack, which contains the Elastic Container Service task definitions.

These script files can be used in automated deployment pipelines and in local development environments as well. For my case study, I made them platform agnostic, that is, they can be executed both on Linux and Windows operating systems (when the required tools, like AWS CLI, are available on the machine that executes the script files). An example shell script content is shown in Listing 2. It shows how the Docker images (see Section 1.4.7) are created and pushed to the Elastic Container Registry [57] (ECR) provided by Amazon Web Services.

The steps in the script file include getting the login credentials of my Amazon Web Services account using the AWS CLI tool, then configure the Docker CLI tool to use these credentials when publishing the image. After Docker is properly configured, I can build the image using the Dockerfile that I created for the web service (discussed in Section 3.4.1). I need to name the output image (tagging) in an Elastic Container Registry specific format, and then I can push it to the target Elastic Container Registry repository which were created by me using the create_repository.sh shell script. I am using dynamic variables to specify some of the Amazon Web Service specific details in the shell script, so it can be easily reused in other projects. When there is a reference like $1 or $2, it points to a dynamic variable that I specify from the command-line when using the scripts.

```
#!/bin/sh
set -euo

aws ecr get-login-password --region eu-central-1 | docker login --username AWS --
password-stdin 537258071742.dkr.ecr.eu-central-1.amazonaws.com

docker build \
  --build-arg AWS_REGION=$1 \
  --build-arg AWS_ACCESS_KEY_ID=$2 \
  --build-arg AWS_SECRET_ACCESS_KEY=$3 \
  -t 537258071742.dkr.ecr.$1.amazonaws.com/master-thesis-traditional:latest \
  .
docker push 537258071742.dkr.ecr.$1.amazonaws.com/master-thesis-traditional:latest
```

Listing 2: The content of the create_and_push_image.sh shell script. It is used for grouping multiple steps together, like configuring Docker to use the credentials for my Amazon Web Services Elastic Container Registry repository (which is used to store Docker image files in Amazon Web Services), bundling the application using Docker, and publishing the image to Elastic Container Registry.

The tsconfig.json file contains the compiler settings used for this project. The TypeScript compiler has a huge set of parameters that programmers can configure, for example, the compiler has multiple levels of strictness for the type checking process [83]. This strictness setting controls what kinds of type violations I consider as errors or warnings in my codebase. I can also specify the output JavaScript format/standard. Listing 3 shows the compiler configuration for the traditionally hosted application.

```
{
  "compilerOptions": {
    "target": "ESNext",
    "module": "CommonJS",
    "moduleResolution": "Node",
    "resolveJsonModule": true,
    "allowJs": true,
    "esModuleInterop": true,
    "strict": true,
    "outDir": "dist",
    "rootDir": "src",
    "baseUrl": "src"
  },
  "exclude": ["node_modules", "dist", "scripts"]
}
```

Listing 3: The content of the tsconfig.json file for the web service used in the traditionally hosted application. There are multiple parameters to specify concerning the output JavaScript code (which TypeScript compiles to), which types of files can we directly import in our source code, what are the excluded directories, etc. For a complete reference of a TypeScript configuration file, refer to the TypeScript documentation [61].

The package.json file is the metadata description of the project. It contains all the dependencies required to run the application, that is, the third-party libraries used in the project, and the name of the project, the authors, compatibility versions, etc.

The src folder contains the modules that build up the web service. Within this directory, the clients folder contains the code that is needed to interact with external services. For example, in my application dynamoClient (see Listing 7) provides a low-level layer that enables me to execute commands to an Amazon DynamoDB [48] table which is connected to the web service.

The config module contains some variables that are used throughout the application based on the execution environment. I can have an arbitrary number of environments, one for local development, sandbox, production, etc. For my case study, I have only used a single environment since this web service only serves as a prototype and not as a production-grade application. Listing 4 shows the content of my configuration used in the traditionally hosted service. It contains the region from Amazon Web Services in which the web service is hosted, and the name of the table that I use for the application.

```
import type { Config } from './types';

const config: Config = {
  aws: {
    region: 'eu-central-1',
    dynamodb: {
      shapes: 'mt-shapes-table',
    },
  },
};

export = config;
```

Listing 4: The configuration file (src/config/default.ts) used for the traditionally hosted web service. It contains the Amazon Web Services region in which the application is hosted and the name of the DynamoDB table used for storing the shape objects.

The handlers module contains the functions that handle the incoming requests and then send the appropriate responses. Usually, they are only used for organizing the request/response and data flow, and not for doing complex operations by themselves. It means that, for example, if I need to store a shape object in my database, then the handler functions are going to invoke the service

module (introduced later) responsible for communicating with the database, but they are not going access the database by themselves. An example handler function is presented in Listing 5. It fetches a shape with the matching identifier from the database using a service function. When the shape does not exist, it responds with 404 Not Found.

Note that the handler on the previous listing is using the ShapeService object exported from the services module to handle the communication with the database. Listing 6 shows the ShapeService, which is a thin layer in the shape web service between the handlers and the database. It contains each operation that the clients can perform on the shape database.

The asynchronous functions that the methods in the service module are calling come from the DynamoDB client module, which provides low-level layer for communicating with the DynamoDB table connected to the shape web service. It contains generic methods that can be used to perform CRUD operations on any type of data (not just shape objects, so this client can be reused in other applications). Listing 7 shows the fetch method on the client that is responsible for retrieving an object by its identifier. In case there is no object with the specified identifier, it returns undefined (nothing).

```
export const fetchOneHandler = async ({ params: { id } }: Request, res: Response) => {
  try {
    const shape = await ShapeService.fetchOne(id as string);
    if (shape) {
      return res.status(StatusCodes.OK).json({ shape });
    }
    return res.status(StatusCodes.NOT_FOUND)
        .json({ message: `There is no shape with id: ${id}` });
  } catch (err) {
    return res.status(StatusCodes.BAD_GATEWAY)
        .json({ message: (err as Error).message });
  }
};
```

Listing 5: An example handler function (src/handlers/index.ts) that fetches one shape from the database and returns it to the client. It receives the id parameter from the request, then invokes the fetch service function (see Listing 6) to retrieve the requested shape object. If it is not found or some error happens during the operation, it gives back a negative response.

```
export const ShapeService = {
  fetchAll: async () => fetchAll<Shape>(),
  fetchOne: async (id: string) => fetch<Shape>(id),
  create: async (shape: Shape) => put<Shape>(shape),
  update: async (shape: Shape, id: string) => put<Shape>({ ...shape, id }),
  delete: async (id: string) => deleteOne(id),
};
```

Listing 6: The shape service module (src/services) implemented by me, which is a thin layer between the handlers and the DynamoDB table.

The routes module contains all the available route configurations for the traditional application, such as fetching one shape, creating a shape, deleting a shape, etc. The routes are configuring how to validate the input/output payloads, and which handlers to invoke. They can also specify a chain of middleware to perform pre- and post-operations on the data objects that flow in the system. For example, a middleware can check for a unique identifier on the input shape object during a shape creation process, and if it is not there, it can generate a new one.

```
export const fetch = async <T>(id: string): Promise<T | undefined> => {
  const { Item } = await getDynamoClient().send(
    new GetItemCommand({
      TableName: shapes,
      Key: {
        id: { S: id },
      },
    })
  );

  if (Item) {
    return <T>unmarshall(Item);
  }
  return undefined;
};
```

Listing 7: Fetching an item from the DynamoDB table using a low-level, generic method. The method is part of the src/clients/dynamoClient.ts file. T is a generic type parameter [84] which is used to make functions or classes reusable. For example, if I specify T as Shape, TypeScript knows that it is dealing with objects that have the Shape type or interface. The Promise type suggests that this is an asynchronous function; more details can be found in [85].

Listing 8 shows the routing configuration for the shape web service. I need to specify the path and the handler, and optionally a list of middleware. In the case of the shape web service, I only have a validator middleware that can reject or pass through the request object after performing a validation against a certain schema. I am using the route parameter (the identifier) and the payload (the shape object) validators in this case.

```
export const router = Router();

router.get('/shapes', fetchAllHandler);

router.get('/shapes/:id', validator.params(IdentifierSchema), fetchOneHandler);

router.post('/shapes', validator.body(ShapeSchema), createHandler);

router.delete('/shapes/:id', validator.params(IdentifierSchema), deleteHandler);

router.put(
  '/shapes/:id',
  validator.params(IdentifierSchema),
  validator.body(ShapeSchema),
  updateHandler
);
```

Listing 8: The routing configuration (src/routes) for the shape web service implemented by me. I specify the HTTP method, the path, the middleware, and the handler function for each route that the web service has.

Schemas are describing how the data objects in the system must look like (already mentioned in Section 1.4.6). I can check the input payloads against my schemas. When the data specified in the payload differs from what I have defined in the schema, then the schema validator rejects the request. There are many libraries for schema validation in JavaScript, my application uses Joi [86]. An example Joi schema that describes a shape object which the web service accepts can be found in Listing 9.

```
const ALLOWED_SHAPE_TYPES = ['circle', 'triangle', 'rectangle'];
const ALLOWED_SIZES = ['small', 'medium', 'large'];

export const ShapeSchema = Joi.object({
  id: Joi.string().uuid({ version: 'uuidv4' }).optional(),
  type: Joi.string()
    .valid(...ALLOWED_SHAPE_TYPES)
    .required(),
  color: Joi.string()
    .regex(/^#(?:[0-9a-fA-F]{3}){1,2}$/)
    .required(),
  size: Joi.string()
    .valid(...ALLOWED_SIZES)
    .required(),
}).meta({ className: 'Shape' });
```

Listing 9: The Joi schema that describes the shape objects (src/schemas/ShapeSchema.ts). I can mark attributes as optional or required. Joi provides many built-in types and type constraints. For example, I can match strings against regexes, or specify upper and lower boundaries for integers. I could even do more advanced assertions, for example, if I would only accept triangles for the size "medium", I could make this conditional check part of the schema.

The types folder contains the TypeScript type definitions for the data objects in the application. I do not need to create the types manually when I am using a schema definition tool like Joi. After I have created my Joi schemas, I was running a tool called joi-to-typescript that converts them into TypeScript definitions. The output from the shapes schema in Listing 9 is shown in Listing 10.

```
export type Shape {
  color: string;
  id?: string;
  size: 'small' | 'medium' | 'large';
  type: 'circle' | 'triangle' | 'rectangle';
}
```

Listing 10: The TypeScript type definition of a shape object created by the tool joi-to-typescript, based on the schema specified in Listing 9. The question mark means that a certain attribute is optional. The "|" operator serves as OR when listing the types for an attribute. In this example, it specifies that the size can be small, medium, and large, but nothing else.

```
(async () => {
  try {
    await application.listen(3000, '0.0.0.0', () => {
      console.log('Server is listening on 0.0.0.0:3000');
    });
  } catch (err) {
    console.error(err);
    process.exit(1);
  }
})();
```

Listing 11: The content of the index.ts file. It defines a function in place and calls it immediately. The function contains the code required to start the web service written using ExpressJS.

Finally, the server.ts file defines methods that are responsible for creating the server object, and the index.ts file creates and starts the server instance. Listing 11 shows how the top-level file in TypeScript looks like. There is an asynchronous function that is defined in place and executed immediately. In the case of an ExpressJS server, this function starts the web service.

3.4 Hosting

3.4.1 Target Platform

As discussed before, the platform where I want to host the application is the Elastic Container Service (ECS) [8] platform from Amazon Web Services. Elastic Container Service enables me to run my containerized (see Section 1.4.7) web service in the cloud, and with the appropriate configuration, I have the possibility to reach it from the internet. Elastic Container Service is organized in the following way: I can define clusters, the clusters can have an arbitrary number of services, which can have an arbitrary number of tasks. The tasks are containerized applications inside the services.

```
FROM node:alpine

ARG AWS_REGION
ARG AWS_ACCESS_KEY_ID
ARG AWS_SECRET_ACCESS_KEY
ENV AWS_REGION=$AWS_REGION
ENV AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID
ENV AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY

EXPOSE 3000

# Include missing dependencies that can cause issues when missing
RUN apk add --update --no-cache libc6-compat

# Copy application builder directory
RUN mkdir /build
WORKDIR /build
COPY . .

# Compile the application
RUN yarn install --frozen-lockfile && yarn compile

# Create application runner directory
RUN mkdir /app
WORKDIR /app
COPY package.json yarn.lock ./

# Install PRODUCTION dependencies
RUN yarn install --production --prefer-offline --frozen-lockfile

# Copy the built application
RUN cp -r /build/dist/. ./

# Cleanup
RUN rm -rf /build && yarn cache clean --all

CMD ["yarn", "start"]
```

Listing 12: The Dockerfile manifest used for creating a container image for the shape web service, written by me. It specifies a set of Amazon Web Services related environment variables that are necessary to host the container in Elastic Container Service. It also installs additional Linux packages, compiles the application, installs the dependencies of the web service, and does some cleanup in the end. The yarn start command is executed when creating a container from this manifest file, which starts the web service listening on port 3000.

Before being able to run it in Elastic Container Service, I need to containerize the web service using a manifest file, which is called the Dockerfile in case of Docker.

The Dockerfile of the shape web service that I have created uses an image with the Alpine Linux operating system as the basis, which already has the NodeJS runtime preinstalled as part of an

image layer [87]. The commands include adding missing Linux packages that I need to safely run the web service, running the TypeScript compilation step, installing the dependencies, etc. The entry point of the container is the command that starts the (compiled) web server.

The reason for choosing Alpine Linux as the basis for my image is that it is extremely lightweight, it is approximately 5MB in size. With all the required resources to run our web service, our final Docker image takes up approximately 120MB of space. We should aim to create as small images as possible, since they are faster to transfer over the network and they cost less to store on the cloud.

The Dockerfile that I have written for the shape web service can be seen in Listing 12. During the build process, I can create the Docker virtual image using the manifest file. An image has all the required resources bundled together, that is, the operating system files, the dependencies, the used resources.

Elastic Container Service works well with the Docker image registry provided by Amazon Web Services, which is called Elastic Container Registry (ECR) [57]. I can use Docker to build and push my virtual images into an Elastic Container Service registry, then I can use the images from the registry as the basis for my Elastic Container Service tasks. The Elastic Container Service task definition also contains the parameters that I can adjust to specify how many resources I want to allocate for my container: The amount of CPU and RAM values. CPU is measured in a unit called vCPU that stands for virtual-CPU. The smallest unit of vCPU is 256, which we can think of as a single core in a 4-core CPU. 1024 vCPU is equal to one physical CPU. The unit for specifying the RAM is represented in MB.

```yaml
TaskDefinition:
  Type: AWS::ECS::TaskDefinition
  Properties:
    Family: mt-shape-service
    Cpu: 256
    Memory: 1024
    NetworkMode: bridge
    RequiresCompatibilities:
      - EC2
    ExecutionRoleArn:
      # Loaded from a different template, which is going to be described later
      Fn::ImportValue:
        !Join [':', [!Ref InfrastructureStackName, ECSTaskExecutionRole]]
    ContainerDefinitions:
      - Name: mt-shape-service
        Cpu: 256
        Memory: 1024
        Image: 537258071742.dkr.ecr.$1.amazonaws.com/master-thesis-traditional:latest
        PortMappings:
          - ContainerPort: 3000
```

Listing 13: The task definition the CloudFormation template for the web service's container. There are mandatory fields I need to specify, such as the launch type, the CPU and RAM configurations, and the container definitions, which specify the Docker containers that the task is going to execute.

The task configuration of shape web service can be seen in Listing 13. The Family gives a name to the tasks that are going to be created from this definition, for example, a certain task is going to be called mt-shape-service-xyz, where xyz is automatically assigned by Elastic Container Service. The Cpu and Memory specifies an upper limit on the resources that the task can take.

The NetworkMode attribute specifies the Docker networking mode used by the task. For a complete specification of available network modes, refer to [88]. RequiresCompabilities points out explicitly that the task is going to be executed on an EC2 instance (discussed in Section 3.4.2), and thus the task must be capable to run on such an instance. The ContainerDefinitions attribute describes the Docker containers to be loaded for this task. Note that a single task can run multiple Docker containers, but in this case, I only needed one definition for the traditional web service (published in Elastic Container Registry).

For the shape web service, I have configured 512MB RAM and 256 vCPU. Different configurations can lead to different performances. This Master Thesis does not include optimizing the resource allocations for the tasks (and the containers inside the tasks), which varies between use cases. The selected configuration is feasible enough to perform the tests I plan to carry out in Section 5.

3.4.2 Running the Elastic Container Registry Tasks

Running the Elastic Container Registry tasks requires virtual machines where it is possible to execute them. In case of Amazon Web Services, the virtual machines are called the EC2 instances [55]. I need to provision the EC2 instances used in the case study by myself. Listing 14 shows an example instance definition that I have created for hosting the shape web service. The instance type specifies the number of resources to provision for the virtual machine. In this case, the instance type t3.small means 2 vCPUs, 2GB of RAM, 30GB of storage, and network speed up to 5Gbps. For a complete list of instance types, refer to [89].

```
EC2InstanceA:
  Type: AWS::EC2::Instance
  DependsOn: ECSCluster
  Properties:
    InstanceType: 't3.small'
    SecurityGroupIds:
      - !GetAtt ECSSecurityGroup.GroupId
      - !GetAtt SSHSecurityGroup.GroupId
    ImageId: 'ami-036be9830ccba80a8'
    SubnetId: !Ref SubnetA
    KeyName: 'admin-key'
    IamInstanceProfile: !Ref EC2InstanceProfile
    UserData:
      Fn::Base64:
        !Sub |
          #cloud-boothook
          #!/bin/bash
          mkdir -p /etc/ecs/
          touch /etc/ecs/ecs.config
          echo ECS_CLUSTER=${ECSCluster} >> /etc/ecs/ecs.config
```

Listing 14: An example EC2 instance definition (from the file aws/templates/infrastructure.yml).

The security groups are enabling access to the instance, ECSSecurityGroup makes it possible for Elastic Container Service to reach use the instance, and the SSHSecurityGroup gives me the ability to connect to the instance using Secure Shell Protocol [90] (SSH).

The ImageId specifies the Amazon Machine Image (AMI) [91] to be used with the instance. An AMI provides the operating system for a given instance, as well as preinstalled applications and configurations. The AMI ID used for my instance is pointing to a freely available AMI made by Amazon Web Services, running Amazon's custom Linux operating system (Amazon Linux 2 [92]). It has the capabilities by default that are needed to connect the instance to an Elastic Container Service cluster.

The SubnetId connects the instance to a subnet within the Virtual Private Cluster provisioned for my web service. The KeyName property points to an SSH key that I have previously generated for my Amazon Web Services account. I can use this key to access the instance from my own computer (I have used it extensively for debugging reasons). The IamInstanceProfile contains a set of security rules related to the instance (what services or who can access the instances, what operations can be executed that involve the instances).

The UserData attribute contains a short shell script that runs when the instance starts up. In this case, I am writing the name of the Elastic Container Service cluster created for my web service into a configuration file, from which the Elastic Container Service system service running on the instance will know which Elastic Container Service cluster to connect to, that is, which Elastic Container Service cluster is going to use the instance to run its tasks.

3.4.3 Networking Resources and Configurations

Specifying the task definition and the EC2 instance is not enough to host the web service with Elastic Container Service. I also need to create a service definition that specifies the following properties: The target count related settings, the network configurations (so the application is reachable from the internet), and the launch type of the containers. I show the service definition of the shape web service in Listing 15. The different parts of this configuration, and the possible uses cases for load balancing, are going to be described in the following sections.

```
Service:
  Type: AWS::ECS::Service
  Properties:
    ServiceName: mt-shapes-service
    Cluster: ... # imported from the networking stack
    LaunchType: EC2
    DeploymentConfiguration:
      MaximumPercent: 200
      MinimumHealthyPercent: 50
    DesiredCount: 2 # = target count
    NetworkConfiguration:
      AwsvpcConfiguration:
        AssignPublicIp: ENABLED
        SecurityGroups: ... # imported from the networking stack
        Subnets: ... # imported from the networking stack
    TaskDefinition: !Ref TaskDefinition # see Listing XXX
    LoadBalancers: ... # connect the service to our load balancer
```

Listing 15: The service definition of the shape web service. The launch type is discussed in Section 3.4.3.3. The desired count (target count) and deployment configuration parameters are discussed in Section 3.4.3.1.

3.4.3.1 Target Count

I can run multiple instances of the same web service and balance the load between them, in other words, I can distribute the work between them. It is a good practice to have at least two instances of the same application always running, possibly in multiple availability zones, for stability and update reasons. The number of concurrently running containers time is called the target count.

I can also define a limit on the maximum number of instances. Note that he maximum number of instances is usually different from the target count. This is a percentage value, for example, if the shape service configuration has 200% specified for the maximum containers and 2 as the target count, then the maximum number of running containers is going to be 4.

Let us consider two scenarios where having multiple instances can help. If my application instances are hosted in two availability zones, for example, Germany and Ireland, and the traffic is distributed between them, then the system can handle a power outage in Germany by routing all the traffic to the other instance running in Ireland. To prevent the Ireland instance from collapsing by the suddenly increased traffic pressure, the service can scale out by starting new instances in that zone until the power outage issue is dealt with in German cloud center.

The second example scenario is about updating an application to a new version/revision. If I would have a new version of the shape web service that I want to publish, I can utilize the load balancer to apply the new version without downtime, that is, without disabling the service for the time of the update. With the 200% maximum containers and 2 target count specification, I have setup an environment that can carry out the update workflow. First, Elastic Container Service is going to create 2 containers with the new web service versions, then it redistributes the traffic from the old containers to the new ones using the load balancer. Finally, it is going to shut down and remove the old versions.

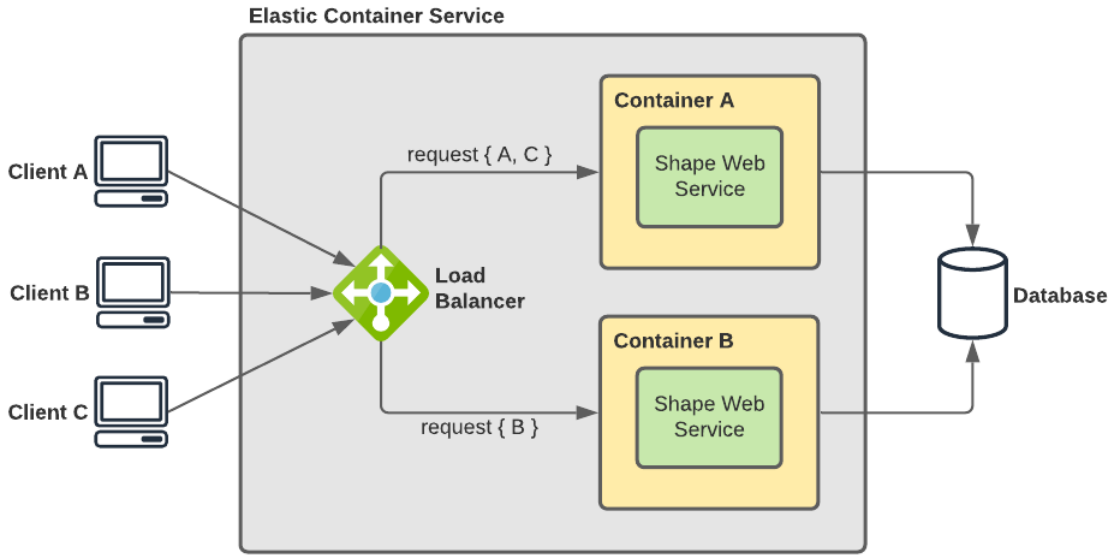


Figure 14: An example load balancing setup. The containerized web service runs in multiple instances inside AWS Elastic Container Service. There is a load balancer that distributes the traffic between the instances, so the input load is split in two. The requests from Client A & B go to Container A and the request from Client C goes to Container B. Note that this figure, for the sake of simplicity, leaves out many other networking resources that are needed to make this communication possible.

Load balancing has a benefit of dynamically scaling up/down the web service instances under certain circumstances (like the unexpected traffic increases). One drawback is that using a load balancer in Amazon Web Services comes with additional costs [93]. Amazon Web Services provides different types of load balancers that I can utilize [94] for my web service.

Figure 14 shows an example layout of a load balancing setup. The requests from the clients are distributed to two different instances of the shape web service. Amazon Web Services provides different types of web services that can distribute the traffic in different ways. For a complete explanation and comparison of Load Balancers in Amazon Web Services, refer to [95]. Note that using a load balancer as part of the traditional approach is optional.

3.4.3.2 Networking

If I want to make my web service reachable from the internet, I need to create an environment from multiple infrastructure resources (discussed later in this section) that makes it possible to connect to the web service. I have already mentioned that a load balancer that plays a key role in this setup: It has its own DNS name that the clients can use to reach my web service instances. However, the load balancer is not the only element in the networking infrastructure; I need to deal with other internal elements that make it possible for the load balancer to reach the web service instances.

The networking resources that I need to configure for the traditional setup include a Virtual Private Cloud network, which is going to be the environment for all the other components. Then I need subnets (possibly in different availability zones), an internet gateway, a routing table, any type of load balancer, and all the security configurations for these resources (roles and permissions to specify and limit what the certain resources can do). This Master Thesis is not going to describe each of these resources in detail. The key point is that I must create and maintain a tedious setup of networking related resources if I want to make the shape web service accessible for the clients. Creating an optimal configuration is not straightforward, it takes time and possibly multiple iterations/adjustments.

Fortunately, the networking part can be separated from an application in a standalone template. This is the reason why I have two templates (as discussed in Section 3.3.2): One template that represents the service configuration (service.yml), and another that contains the networking resources (infrastructure.yml). If I would want to create a different service (other than the shape web service presented in this case study), I could either reuse the already existing networking infrastructure, or I could easily create a new one by copying (and slightly modifying) this infrastructure template.

3.4.3.3 Launch Types

There are two major ways that engineers can choose to run their Elastic Container Service tasks: Launching them as EC2 [55] or Fargate [96] instances. Fargate is a new approach from Amazon Web Services to run containerized applications, referred to as Container-as-a-Service [97]. For a complete comparison between the two launch types, refer to [98].

In my case study, I have decided to use EC2, since it is the Infrastructure-as-a-Service solution provided by Amazon Web Services (which I planned to compare with Function-as-a-Service). Fargate can be considered a serverless solution on its own, but it has many differences from the Lambda platform [99]. Discussing Fargate and other Container-as-a-Service solutions in detail is not part of this Master Thesis.

3.5 Deployment

After I have the source code of the web service, the hosting related templates, and the script files that help me with the build process, I can deploy the web service in Amazon Web Services. I use the AWS CLI tool [63] provided by Amazon Web Services, so that I can manage my cloud infrastructure from the command line, which is a simple way to interact with the cloud. I could easily automatize the build process using a deployment pipeline (no need to manual configurations on a GUI). Note that I have not built an automated deployment pipeline in my case study since there was no need to automatize it: I have only created and teared down the infrastructure a few times.

As part of the deployment process, I need containerize the project using the Docker CLI tool: It takes the Dockerfile manifest that we have previously created (see Section 3.4.1), and it creates a Docker virtual image using the shape web service's source code. After this step, I need to make sure that I have an Elastic Container Registry [57] repository where I can store this image file. I can programmatically create such a repository using one of my script files, and then I can publish the image file into it. The image in the registry becomes ready to be cloned and executed in the Elastic Container Service tasks.

Next, I need to create the network infrastructure. I can load and parse the networking template with the AWS CLI tool, which is then going to instantiate the resource creation process. Note that there is a dependency between the networking and the application stacks: The infrastructure must be created first, since we reference these resources in the service template.

After the networking resources are ready, I can load the service template to create the instances of the shape web service. When the process completes, the service becomes available and reachable for the clients by using the load balancer's DNS name.

3.6 Summary for the Traditional Approach

In this chapter, I have discussed the different aspects of the traditionally hosted solution. I have briefly looked on the application structure: I have discussed the different parts of a web service written in TypeScript, the script files that enhance the application management process, the CloudFormation [56] templates for the networking infrastructure and the web service itself. I have presented the requirements of hosting an application in Elastic Container Service, and I have introduced the steps of carrying out a deployment process of the traditionally hosted web service.

It is difficult to draw conclusions on the web service implementation part (how difficult was it to implement the web service), since it depends on the engineer's experience who creates it.

However, I can reflect on the work required to host the application in Elastic Container Service. Once I have CloudFormation templates that I can reuse in other projects (get inspiration from them), it is much easier to create an initial deployment, but this time I needed a lot of effort to create these templates by scratch. There are many resources that I needed to consider, which are is tedious to maintain.

In the next chapter, I am going to investigate how different it is to use a serverless hosting approach instead.

4 Serverless Approach

As I have mentioned in Section 1.2 where I have established my goals, my plan is to migrate the shape web service presented in Section 3 from Elastic Container Service [8] to the Lambda platform [9]. Lambda is the Function-as-a-Service (see Section 1.4.11) solution provided by Amazon Web Services.

In what follows, I am going to describe the migration of the web service. This includes the reflection on the required infrastructure and source code changes, and the hosting and deployment differences between the two approaches.

4.1 Overview

To host the shape web service on the Lambda platform, I need to redesign it into a set of serverless functions. This requires source code and infrastructure code changes, and it already gives me a set of constraints about how to organize the application structure.

Before carrying out the migration, I have familiarized myself with the Lambda platform, such as concepts, guidelines, design patterns (like the ones presented in Section 2.2). After this step, I have taken the shape web service from the traditional approach, and I have designed two types of Function-as-a-Service architectures that I can implement by "migrating" the traditional approach's source code, that is, by creating something new based on what I have implemented already.

I have created a minimalistic and a more realistic serverless application as part of my case study. With the minimalistic approach, I wanted to reuse as much of the traditional version of the web service as possible with the least number of changes to the underlying architecture and source code. This represents a "prototyping scenario" that I can use if I want to simply check how a web service performs on the Lambda platform. The second, more realistic approach changes the application structure to make it more feasible for the serverless platform, including applying best practices and design patterns of serverless application development.

Since I have used Elastic Container Service from Amazon Web Services in the traditional approach, I am not going to consider any serverless platforms other than Amazon Web Services' Lambda for hosting my serverless application, since I want to model a migration between two platforms supplied by the same provider. For the challenges that can come up when migrating between different providers, refer to [67].

4.1.1 Migration Approach

The easiest way for the migration is taking the existing server and wrapping it in a module that makes it usable as a serverless function, visualized on Figure 15. Such a wrapper provides an interface between the cloud provider's system and the web service written using a web framework (which is ExpressJS in my case study).

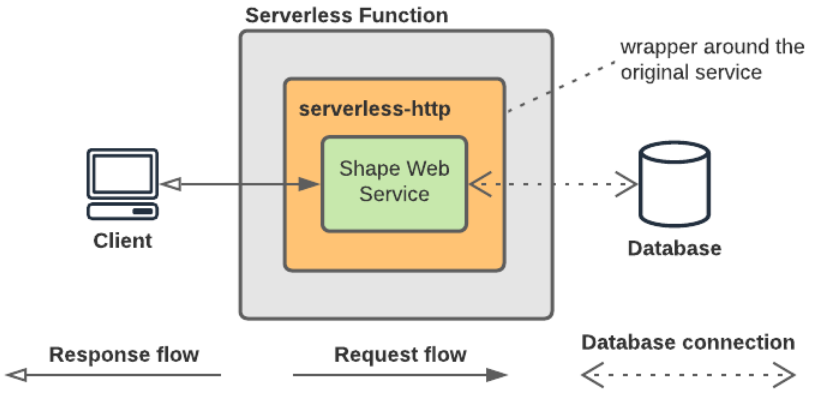


Figure 15: The web service created in Section 3 is reusable when wrapped inside a module like serverless-http. Figure 16 shows what the wrapper provides.

The request and response structure of the API Gateway (presented in Section 1.4.12) is not fully compatible with the expectations of the traditional service, so the wrapper transforms them accordingly. Figure 16 demonstrates this transformation process visually. The API Gateway and

the web service will not notice anything: They can do their tasks without knowing that there is a data transformation step in between them. The request/response format from an HTTP API Gateway (which I am using in my case study, described in Section 1.4.12.2) can be found in [100], and the ExpressJS related format in [101].

There are many libraries that support this type of wrapping approach. I decided to use the serverless-http wrapper [62] because it is lightweight (the bundle size is only 22kb), and it can be used with many different web frameworks, and thus it gives great flexibility. For example, a different tool called serverless-express [102] is created specifically for the Express web framework, while serverless-http supports other JavaScript web frameworks like Koa [103], Hapi [24], or the increasingly popular Fastify [104]. I prefer this kind of flexibility in my applications. For example, if ExpressJS would become deprecated, there would be one less step required in migrating to a different web framework.

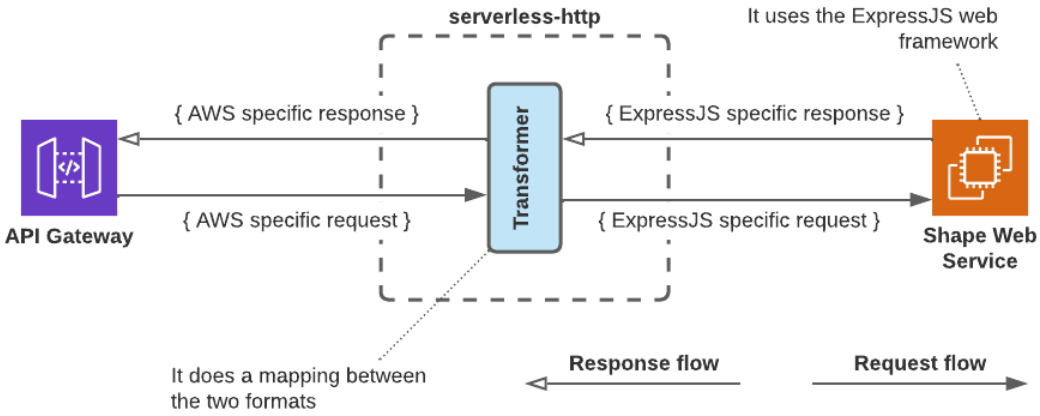


Figure 16: The serverless-http transformer in action. It is capable of translating/mapping requests/responses from the API Gateway into a ExpressJS format, and vice-versa.

The original server wrapped in serverless-http is going to be the function that the HTTP API Gateway invokes with the incoming request. The biggest advantage of using a wrapper like serverless-http is that it does not require modifications in the original web service.

After the original web service is wrapped in the serverless-http wrapper, I can directly use it as the entry point of the serverless function which is then going to be invoked by the HTTP API Gateway. Listing 16 shows the method of wrapping the ExpressJS application using serverless-http.

```
import serverless from 'serverless-http';
import express from 'express';

application = express();
// Setup routes, middleware, etc...

export const ExpressServerless = serverless(application);
```

Listing 16: Wrapping an ExpressJS application with serverless-http. It requires only one line of code from the developers to perform the wrapping.

4.2 Architecture

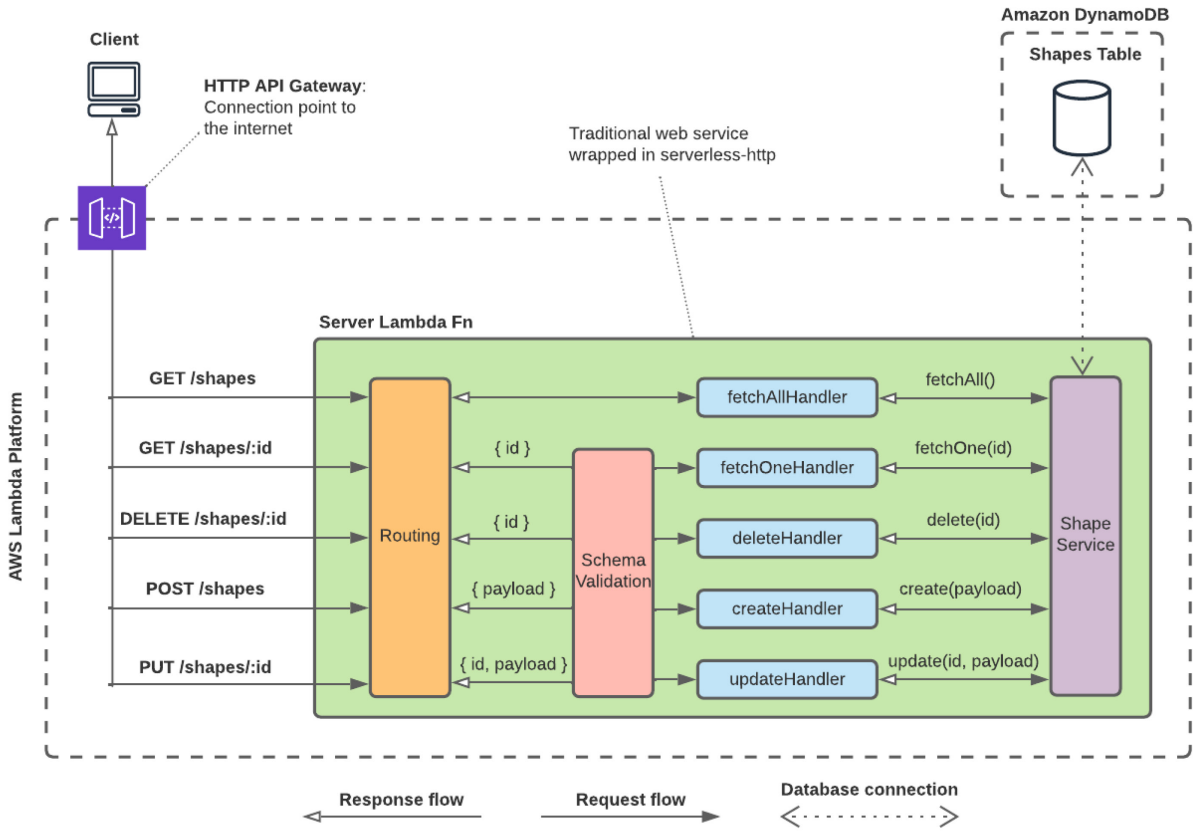


Figure 17: The serverless architecture after the first iteration. The traditional server is wrapped in serverless-http and is being used as a serverless function. Only few changes were required to the original web service's code, which are going to be described in later sections.

As mentioned in Section 4.1, I have tested out two approaches for creating a serverless application, a minimalistic and a more realistic one. The minimalistic variant is not going to be discussed in detail. Although it is a working approach, it goes against the idea of Function-as-a-Service since it puts every functionality provided by the web service into a single serverless function. The more realistic variant is composed by multiple serverless functions, which comes with advantages and drawbacks, described in later sections (Section 4.2.1.2 and Section 4.2.1.3).

Even though the minimalistic approach is not discussed in detail, I want to give a brief overview about its architecture and the general idea behind it. Figure 17 shows the architecture diagram of this version.

In this minimalistic approach, I have put all my web service-related code into one serverless function. This is not a bad approach in the sense that the system fulfills all the requirements from the client's perspective. The CRUD operations are still present, the communication is still based on standardized HTTP methods. However, the general idea of a serverless application is that we compose it from multiple serverless functions, each doing only a small set of tasks. Wrapping the functionality of a complete web service into a single function goes against this point of view.

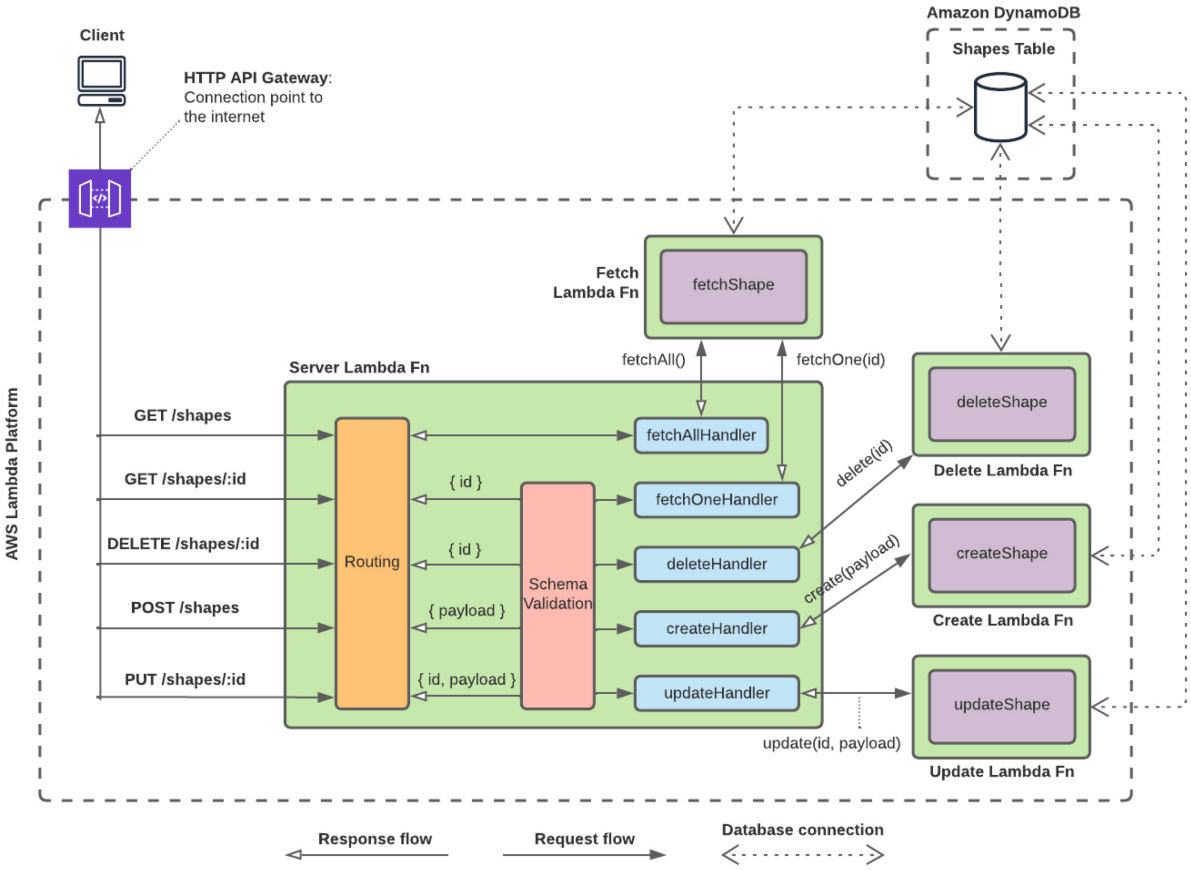


Figure 18: The architecture after the second iteration. The functionality that was not tightly related to the core server functionality (the service operations) were moved into separate functions.

The reason why I have opted for the HTTP API Gateway is that I already have the request/response lifecycle management features implemented in the original web service using ExpressJS, it would not make sense to use routing options of a REST API Gateway. What I need in the case of an existing web service is a proxy, and that is one of the reasons why Amazon Web Services created the HTTP Gateway resource. It is going to cost less, and it is easy to setup and integrate with my exist application. The database is going to be the same DynamoDB table for the shapes, like what I had for the traditional approach as well.

The more realistic approach composes the shape web service by using multiple serverless functions, while still aiming to keep the original codebase intact as much as possible. Figure 18 shows an overview of the system after the second iteration. In the following sections, I am going to explain what resulted in these changes and what the different functions do.

4.2.1.1 Increasing the Function Granularity

Although migrating the original application to the serverless platform was not hard with a wrapper like serverless-http, I did an experiment with increasing the function granularity by decomposing some parts of the original application into other serverless functions. This may come with many benefits, which are going to be further discussed in Section 4.2.1.3.

After I made the decision to increase the serverless function granularity in the architecture, I had to come up with plan about which parts of the original ExpressJS application to extract and how. I can put the modules of the original web service into two separate groups, one with modules that

are tightly coupled with the ExpressJS web framework itself, and one with items that are loosely connected to it. The first group has modules related to routing, managing the requests and responses, and loading plugins, middleware, and configurations. The second group has the service functions (shape creation, deletion, etc.) interacting with the DynamoDB table.

I can make distinct serverless functions based on these two groups. I have further increased the function granularity in the second group by creating separate functions for each method provided by the service object (originally, this object was shown on Listing 6).

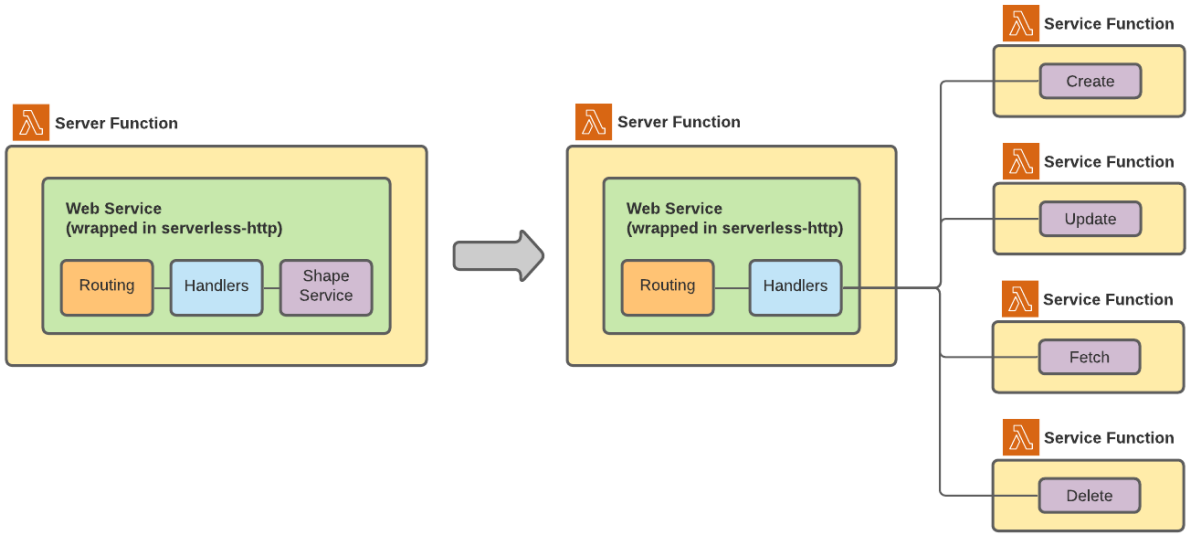


Figure 19: Decomposing the single serverless function into multiple serverless functions by extracting the functionality provided by the service module. After the transformation, the server function is only responsible for managing the request/response lifecycle and the service functions are responsible for handling the communication with DynamoDB.

Figure 19 shows the process of how I decomposed the loosely connected parts from the core server function. The server function remained responsible for handling the request itself, and the service functions remained responsible for communicating with the shape database. For example, if a client invokes a client operation, the request validation, routing, and request/response lifecycle handling (described in Section 1.4.6) happens in the server serverless function, but it propagates the data to the corresponding service function without accessing the database by itself.

4.2.1.2 Disadvantages of the Increased Granularity

Programmers need to keep in mind the guidelines presented in [7]. Chaining too many serverless functions is going to result in increased costs (when the functions are executed sequentially). The impact of the cost increase caused by the function chain depends on the length of the function chain.

In the case of the refined shape web service, the server Lambda function needs to wait for the results of the service Lambda functions, and thus this communication is sequential. It will result in increased execution costs for the time while one of the service functions runs, since a) the server function is waiting for the results (it keeps running), b) the service function is doing its computation. The issue is visualized on Figure 20. Note that the negative effect caused by function chains depends on the cold/warm start of the serverless functions involved in a chain. If the invoked service function is in a warm state, the server function is going to wait for a shorter amount of time.

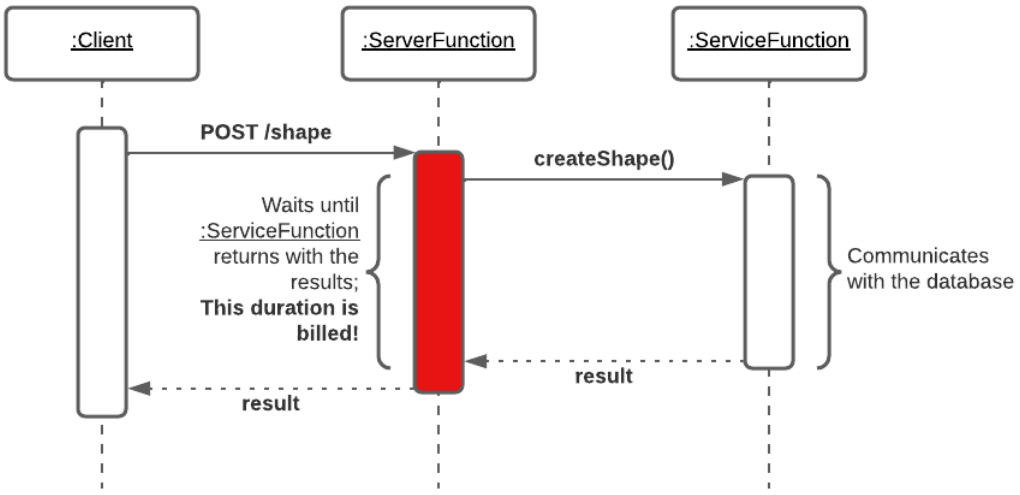


Figure 20: The negative effect of serverless function chains. The server Lambda function (marked with red) assigns the work to the service Lambda function and waits for the results. This waiting is resulting in additional costs since Lambda functions are billed based on their execution time, and the server Lambda keeps running until it has the results from the service function.

4.2.1.3 Advantages of the Increased Granularity

There are multiple advantages that come with the increased serverless function granularity. One important aspect is the scalability of the application. This is best presented by an example: When most of the traffic (of the incoming requests) goes to the shape fetch route, then only the server and the fetch service Lambdas need to scale out horizontally, the other service Lambdas (create, update, delete) can stay on a lower amount of running instances. The system only scales out the code parts that are being used. With the minimalistic approach, when I had the complete business logic in a single function, then the whole application scaled out regardless of the request route being taken.

Another aspect is the better maintainability. If there would be a situation when one of the service functions requires an update (changes from the business perspective or a simple bug fix), then it is possible to update only that single Lambda function, and the others will not be redeployed. This makes the update release process a lot more efficient, since a) the update takes less time to complete, b) the warm instances of the other Lambdas can keep running (an update in a Lambda function invalidates and terminates all the running instances of that function [105]).

Increasing the serverless function granularity also helps with the security of the system. Each service Lambda has only a small set of rights to access the database, for example, the create service function has the write permission to the database, but it cannot read the shape objects in the system and cannot delete them.

Considering the architecture of the minimalistic approach, having the complete web service in a single serverless function seems like a terrible idea from a security perspective, since that single serverless function would have complete access to the DynamoDB table, and it would be directly exposed to the internet through the HTTP API Gateway (without considering the security layers in the networking infrastructure that connects it to the internet).

4.3 Implementation

In this section, I am going to present the implementation details of the more realistic version of the serverless application. I have already mentioned in Section 3.3.1 that functions are the basic building blocks of TypeScript (and JavaScript) application, and they are organized into modules alongside with variables. I have also mentioned that thinking in terms of composing functions to form an application is helpful for thinking about how serverless functions should behave, how they should interconnect. The serverless variant of the shape web service is also written using TypeScript.

4.3.1 Application Structure

The organization of the source code has changed significantly, but it is important that these changes are due to differences between the two ways of designing the application. The serverless approach requires different hosting and deployment related artifacts than the traditional approach. I am going to reflect on the code complexity differences during the evaluation of the two approaches in Section 5.4. The code that represents/implements the web service is just slightly modified compared to the original version. Figure 21 shows an overview of the structure of the serverless code repository.

Note that every artifact shown on the figure was created by me, these are artifacts that the programmer needs provide for an application hosted on the Lambda platform [9].

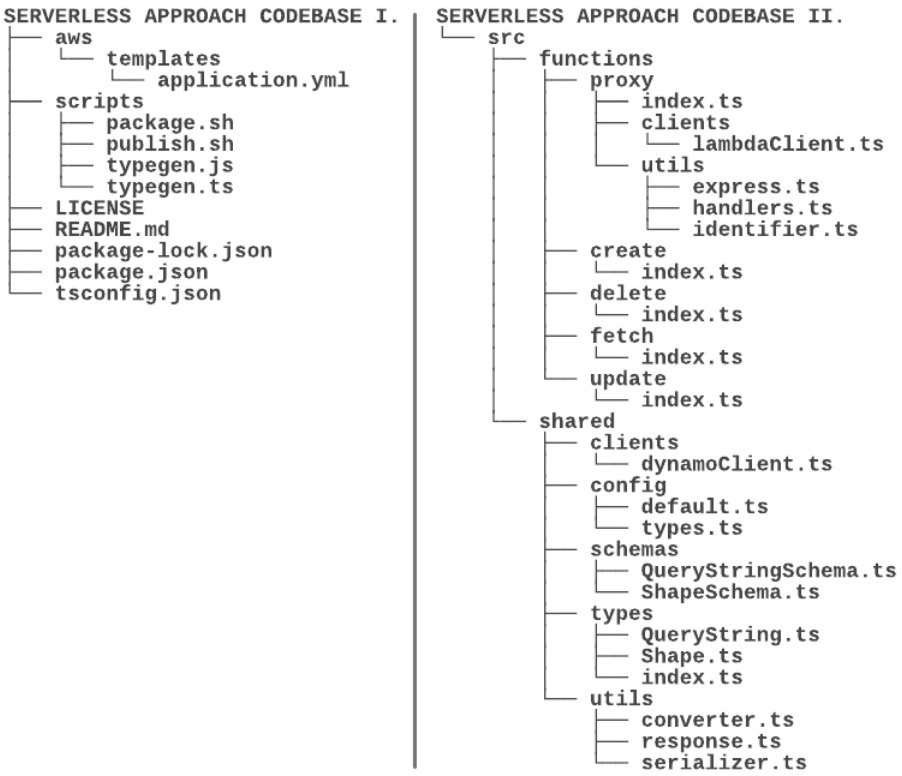


Figure 21: Folder structure of the codebase for the serverless approach. Each artifact was implemented by me. I have split the view in two to save space. The aws and src folders are on the same level.

The folder structure, other than the internals of the src directory, is similar to what I have presented for the traditional approach on Figure 13. The aws folder contains the templates that are describing the serverless infrastructure. Note that I only need one YAML template this time,

since I do not have to deal with the networking infrastructure – this is done by the cloud provider. I am going to reflect on the content of this template in Section 4.4.1.

The scripts folder is present here as well, containing only two shell script files this time. The package.sh is used to bundle the serverless function into a zip file, which is one way to package the serverless functions for the Lambda platform (making it ready for deployment). Amazon Web Services recently introduced a way to create Lambda functions from Docker images, but that solution is relatively fresh and is not considered in this Thesis. For more information, refer to [106].

Listing 17 shows the content of the package.sh script. First, I need to compile the application from TypeScript to JavaScript, then there is a folder and source code organization step to prepare the Lambda layers [107] (explained in Section 4.3.1.1) and the different functions. I put everything that is going to be used be my serverless functions into the package folder, from which the YAML template is going to reference code pieces.

This packaging step is relatively short compared to the previous build script for the traditional approach. I do not have to login into the Elastic Container Registry, I do not have to build a Docker image from a Dockerfile, and I do not need to push the image in an Elastic Container Registry repository. I just need to make sure that the application code is compiled to plain JavaScript, and that I organize the output code and the project dependencies in a way that I can upload it to the Lambda platform.

```
#!/bin/sh

set -euo

# Cleanup existing dist and package folders
npm run clean

# Compile the TypeScript code to JavaScript
npm run compile

# Organize layers and functions into the package directory
mkdir -p package/dependencies/nodejs
mkdir -p package/shared/nodejs
cp -R dist/functions package/
cp -R dist/shared/ package/shared/nodejs/

# Install the dependencies for the dependency layer
cp package.json package-lock.json package/dependencies/nodejs
npm --prefix ./package/dependencies/nodejs ci --only=production \
    ./package/dependencies/nodejs
```

Listing 17: Packaging my Lambda functions, preparing them for deployment (contents of package.sh). I need to clean up the previous resources that I have built (it removes an existing package folder if there is one), then I need to compile the application from TypeScript to JavaScript. As the last step, I copy the project files into the output folder, and I install the required third-party dependencies used by the functions.

Listing 18 shows the publish.sh script for packaged project. Note that I am using a different tool this time for the deployment: AWS SAM [64], which stands for Serverless Application Model. In the traditional approach, I used the AWS CLI tool. The two tools are similar in terms of what they do, but they are doing it differently. The different between them is going to be further explained in Section 4.5.

```
#!/bin/sh

set -euo

sam deploy \
  --template-file aws/templates/application.yml \
  --s3-bucket bfrks-sam-code-bucket \
  --stack-name mt-shapes-app \
  --capabilities CAPABILITY_IAM \
  --region eu-central-1 \
  --no-fail-on-empty-changeset
```

Listing 18: The publish.sh shell script for our packaged Lambda application.

The other files in the root folder are also like the files from the traditional approach: The compiler configuration, the project metadata, etc. These files only have minor differences and adjustments, for example, the compiler configuration is different since the project's folder structure is not the same. Consider these files as the same in both versions.

The src folder has a different structure this time. The remaining files in this folder are split into two different sections: shared code and function specific resources.

4.3.1.1 Shared Code

The shared folder contains modules that I want to share between the different functions. When I have pieces of code that more than one function uses, I can put it in the shared folder. Organizing shared code into a separate folder comes with an advantage: I can create a Lambda layer [107] from it. The code in a Lambda layer can be uploaded to Amazon Web Services separated from the serverless functions themselves, that is, I do not need to include these files in every Lambda package where I want to use them, instead I can reference them in the Lambda functions where the shared code is used. This results in smaller Lambda packages, which are desirable for reducing the cold start times (see Section 2.2 which points out that having smaller Lambda packages is a good choice). Programmers should aim for keeping the serverless function packages as small as possible.

The third-party dependencies of the shape web service (like the ExpressJS framework) can also be part of this Lambda layer of the shared code, or they can be separated into a different layer; this decision is up to the engineers. Lambda functions can reference up to five layers [108]. Note that having too many Lambda layers increases the overall complexity of the system, because it quickly becomes confusing where the different modules are loaded from. Also, based on my personal experience, I can say that writing integration and functional tests [72] for Lambda functions that are using multiple layers is a demanding task. For this case study, I have opted for a two-layer setup, one layer for the third-party dependencies and one for the shared code.

The config directory stores the application configurations for the different environments (suitable for local development, sandbox, production, but I only use one this time), using the same configuration management tool that I have used for the traditional setup.

The shared folder contains modules that were present in the traditional approach as well, these can be used from the original implementation without little to no modifications (clients, schemas, types). There are also changes and new files for the serverless approach.

The utils are a bit different this time. Since I have the server and multiple service Lambda functions, I need to deal with additional requirements regarding to the communication between these Lambda functions. The data I want to send between them must be converted into a JSON format.

The data conversion to JSON comes with further challenges because plain JavaScript objects (that I use in the application) can have references to themselves (circular dependencies), and object references not supported in JSON directly [109]. Thus, I need to encode and decode the objects carefully. Listing 19 shows the implementation of the utility functions for decoding and encoding objects safely, stored in the serializer.ts file. This module uses a third-party module called CircularJSON [110] that can encode/decode the circular dependencies in a custom format.

```
export const serialize = (item: any): string => CircularJSON.stringify(item, null);

export const deserialize = (item: string): any => CircularJSON.parse(item);
```

Listing 19: Serializing JavaScript objects in a safe way using a module called CircularJSON (utils/serializer.ts).

Another challenge is that the JSON strings must be converted to unsigned byte arrays, this is currently the only way to share data between two Lambda functions that call each other. I can use the built-in TextEncoder and TextDecoder JavaScript modules for this. The converter.ts file has the utility functions to support the conversion, as shown in Listing 20.

```
export const encode = (item: string) => getTextEncoder().encode(item);

export const decode = (item: Uint8Array | undefined)
        => (item ? getTextDecoder().decode(item) : '');
```

Listing 20: Utility functions (from utils/converter.ts) to encode strings into byte arrays, and to decode byte arrays into UTF-8 strings. The decode function is prepared to occasionally handle inputs that are undefined (they are nothing), which can happen in case an invoked Lambda function gives back an empty data object. It returns an empty string in that case.

I have covered the means of handling the communication between two Lambda functions, but I have not mentioned yet that the API Gateway connected to the server Lambda function also expects a certain format for the response object that I want to send back to the client. If I do not fulfill these requirements, the response will result in a 502 Bad Gateway error.

I have created utility functions to create responses in the required format, they are stored in the response.ts file, shown in Listing 21. The format needs a status code number and the body as a string. If I want to transfer objects or arrays, I need to use the serializer function that I have already presented in Listing 19.

There is an additional method (createErrorResponse) for creating an error response from a JavaScript error object, as these cannot be converted into JSON, since this conversion is only possible on data structures that have enumerable properties (like simple objects with their key-value pairs). The internal representation of Error in JavaScript differs from a simple object, and thus I need to explicitly pick a few properties that I want to include in the response (like the name and the message).

```
export type SimpleResponse = {
  statusCode: number;
  body: string;
};

export const createSimpleResponse = (
  statusCode: number | undefined,
  body: string
): SimpleResponse => ({
  statusCode: statusCode || StatusCodes.OK,
  body,
});

export const createErrorResponse = (err: Error): SimpleResponse =>
  createSimpleResponse(
    StatusCodes.BAD_GATEWAY,
    serialize({ name: err.name, message: err.message })
  );
```

Listing 21: The utility functions used for creating a response from the server Lambda function to the API Gateway (utils/response.ts). These functions are creating objects that are matching the format requirements from the API Gateway, so the responses organized this way can be transferred to the clients successfully.

4.3.1.2 Function Specific Resources

In the last directory that I have not discussed yet, I have subfolders containing code pieces that are related to only one serverless function. These source code files are only included in their corresponding Lambda packages, and thus they are not available in others. For example, if I put a module into the directory of the server Lambda function, that module will not be available for the other serverless functions to use.

The first function I want to describe is the server Lambda function. As I have mentioned it already, this function is connected to the API Gateway, it handles the incoming request (routing, validation, parsing) and the outgoing response (encoding), and it invokes the service functions for database related operations.

The code in this Lambda from the original implementation is heavily reused from the traditional approach. Configuring the ExpressJS web framework is the same, so I can reuse the route configurations with the handlers and validators presented in Section 3.3.2. I only need to call the serverless-http (see Section 4.1.1) wrapper to wrap the server object.

There is an additional file that contains a client that is used in the server function called lambdaClient.ts. It is used for handling the invocation of the service Lambdas. Listing 22 shows how the invocation mechanism is implemented. Note that this solution is specific to AWS JavaScript SDK V3 [111], that is, previous versions of the AWS SDK use a different approach to carry out operations like invoking a Lambda function or inserting an object into a DynamoDB table.

In this version of the SDK, I need two things to perform an action: A client that is connected to an Amazon Web Services account and a command that I want to execute using the client. In this case, the client is connected to my developer account that hosts the service Lambda functions (which I want to invoke), and the command is an invocation command of a Lambda function. Note that Listing 22 uses the utility functions presented before to encode/decode the messages between the functions.

```
export const invoke = async (targetFn: string, payload: unknown) => {
  const command = new InvokeCommand({
    FunctionName: targetFn,
    InvocationType: 'RequestResponse',
    Payload: encode(serialize(payload)),
  });

  try {
    const response = await getLambdaClient().send(command);
    const payload = <SimpleResponse>deserialize(decode(response.Payload));
    return createSimpleResponse(payload.statusCode, deserialize(payload.body));
  } catch (err) {
    return createErrorResponse(err as Error);
  }
};
```

Listing 22: Invoking a Lambda function programmatically (server/clients/lambdaClient.ts). First, I need to create an "invocation command" that is executed via the Lambda client provided by the Amazon Web Services SDK. I can expect a response from the invoked Lambda function or an exception if something fails. I can utilize the response creation utility functions presented in Listing 21 to wrap the response in a format that I can send directly to the API Gateway later.

The interesting part is the content of the invocation command. I can specify the name of the Lambda function that I want to invoke (for example, the create service function), the type of the invocation, and the payload, which is the data that I want to send. There are multiple invocation types for the Lambda functions. RequestResponse results in a synchronous invocation (the calling Lambda function waits for the results from the invoked one), Event would mean a "fire-and-forget" (asynchronous invocation) approach, and DryRun would mean that I am not performing a real execution, I just want to check if the invoker function has the correct permissions to invoke the target function.

The lambdaClient is used in the slightly modified handlers. In the original implementation, the service functions were stored in an object (Listing 6) within the same application, but they are now separate Lambda functions, so I need to adjust the handlers accordingly. Listing 23 shows the delete handler as an example to demonstrate how I can use the invoke method presented in Listing 22.

```
export const deleteHandler = async ({ params: { id } }: Request, res: Response) => {
  const response = await invoke(deleteFn, { id });
  res.status(response.statusCode).json(response.body);
};
```

Listing 23: The handler that is responsible for the shape deletion (from proxy/utils/handlers.ts). It takes the identifier of the shape that I want to delete, then uses the invoke method presented in Listing 22 to invoke the Lambda function responsible for the shape deletion.

After I have discussed how the server Lambda function (the entry point of the system) works, I can present one of the service Lambda functions. These Lambdas are quite similar to each other, and they are much more simplistic compared to the server Lambda. Their only task is to perform a certain CRUD operation on the shape DynamoDB table. Listing 24 shows the internals of the create Lambda to illustrate how a service function look like.

Note that createFn uses the put method to insert the shape object into the database. This method is imported from the dynamoClient module that is shared across the service functions, and it is very similar to the Lambda client and the invocation command discussed recently. Listing 25 shows the put method in detail. The dynamoClient is completely reused from the traditional implementation.

```
export const createFn = async (shape: Shape) => {
  if (shape) {
    try {
      await put<Shape>(shape);
      return createSimpleResponse(StatusCodes.CREATED, serialize(shape));
    } catch (err) {
      return createErrorResponse(err as Error);
    }
  }
  return createSimpleResponse(
        StatusCodes.BAD_REQUEST,
        serialize({ message: 'Invalid request' })
  );
};
```

Listing 24: The internals of the create service Lambda function (create/index.ts). It is responsible for inserting the input shape object into the DynamoDB table.

Note that the put method uses the marshall method from the Amazon Web Services SDK on the shape that I want to store in DynamoDB. This step is necessary, since DynamoDB has a custom JSON format [112] for storing the objects that is not fully compatible with the structure of plain JavaScript objects, and thus I need perform a conversion. The marshall function has a counterpart called unmarshall, which converts a DynamoDB JSON object into a plan JavaScript object.

```
export const put = async <T>(item: T) =>
  getDynamoClient().send(
    new PutItemCommand({
      TableName: shapes,
      Item: marshall<T>(item),
    })
  );
```

Listing 25: The put method in the dynamoClient module. It uses the marshall method from the Amazon Web Services SDK to convert the shape object into a format that is feasible for DynamoDB.

After discussing how the implementation looks like for the serverless approach, I can proceed to show what is required to host these server=less functions on the Lambda platform.

4.4 Hosting

In the previous section, I have presented the resources used in implementing my serverless functions. In what follows, I am going to briefly present the templates that I need to create (and maintain) to host these serverless functions on the Lambda platform from Amazon Web Services.

4.4.1 Serverless Resources

4.4.1.1 DynamoDB Definition

The DynamoDB table definition is only a few lines long when using a Serverless Application Model (SAM) template [64], shown in Listing 26. In case of a CloudFormation template [56], I would have to use the resource type AWS::DynamoDB::Table that requires many more details to be specified (see [113] for the complete list). Therefore, using a SAM template is useful: It provides a way for engineers to write concise resource definitions in their templates.

```
ShapesTable:
  Type: 'AWS::Serverless::SimpleTable'
  Properties:
    TableName: 'mt-shapes-table'
```

Listing 26: Defining the shapes table (application.yml). It only requires specifying the resource type and the name of the table, other attributes (like [113] shows) are automatically generated, saving the programmers from writing boilerplate code.

4.4.1.2 Serverless Function Definitions

Using SAM, I do not have to explicitly specify a resource for the HTTP API Gateway in the template, I can simply specify it on the server Lambda function which is going to be responsible for handling the requests.

Listing 27 shows the resource definition of the server Lambda. The Events attribute specifies which events can trigger the Lambda function (explained in Section 1.4.12). The ApiEvent means that the function can be invoked by an API Gateway. Specifying the Type as HttpApi and leaving out every other attribute related to an HTTP API Gateway resource is a shorthand for a) creating an HTTP API Gateway, b) connecting it to the Lambda function, c) specifying that every request to the API Gateway (from the clients) should be forwarded to the server Lambda function as-is, the HTTP API Gateway should not handle the requests. It is going to serve only as a proxy.

```
ServerFn:
  Type: 'AWS::Serverless::Function'
  Properties:
    FunctionName: 'mt-shapes-server'
    CodeUri: '../../package/functions/server'
    Handler: index.serverFn
    Policies:
      - LambdaInvokePolicy:
          FunctionName: !Ref CreateFn
      - LambdaInvokePolicy:
          FunctionName: !Ref DeleteFn
      - LambdaInvokePolicy:
          FunctionName: !Ref UpdateFn
      - LambdaInvokePolicy:
          FunctionName: !Ref FetchFn
    Environment:
      Variables:
        CREATE_FN: !Ref CreateFn
        DELETE_FN: !Ref DeleteFn
        UPDATE_FN: !Ref UpdateFn
        FETCH_FN: !Ref FetchFn
        NODE_CONFIG_DIR: '/opt/nodejs/config'
    Events:
      ApiEvent:
        Type: 'HttpApi'
```

Listing 27: The resource definition of the server Lambda function that is directly connected to the API Gateway.

This is another example of how SAM is reducing boilerplate code. In CloudFormation, I would need to define a separate API Gateway resource [114] with multiple required attributes, then connect it to the Lambda function, and finally specify that it should forward the requests to the function as-is. The SAM template is much more concise in terms of describing the infrastructure.

The CodeUri parameter tells the Lambda platform where to look for the entry point of the serverless function (in which folder, which in this case is going to be the package folder created by

using the shell script presented in Listing 17). The Handler points to the exported function in the compiled TypeScript module that I have created for the server Lambda function.

Note that we I also specify list of policies. The server Lambda function must invoke the service functions to carry out the CRUD operations, and thus I need to give access to these other functions. I pass the names of the functions as environment variables, so I do not have to hardcode these Lambda function names into the server Lambda function.

The service Lambda function definitions are similar to the definition of the server Lambda, but they do not have an input event (they are going to be programmatically invoked from the server Lambda), and they need different policies (enabling them to perform CRUD operations on the DynamoDB table). Listing 28 shows an example service Lambda resource definition.

```
UpdateFn:
  Type: 'AWS::Serverless::Function'
  Properties:
    FunctionName: 'mt-shapes-update'
    CodeUri: '../../package/functions/update'
    Handler: index.updateFn
    Policies:
      - DynamoDBWritePolicy:
          TableName: !Ref ShapesTable
    Environment:
      Variables:
        SHAPES_TABLE: !Ref ShapesTable
        NODE_CONFIG_DIR: '/opt/nodejs/config'
```

Listing 28: The resource definition of the update service Lambda function with similar properties defined on Listing 27.

4.4.1.3 Global Resources and Lambda Layers

Note that the resource definitions presented in Listing 27 and Listing 28 do not contain attributes about the RAM configurations or the Lambda layers (discussed in Section 4.3.1.1). The reason is that I can define global properties for all serverless functions within the same template. This technique is used for removing code duplication, since I do not have to repeat these properties repeatedly for each function. Listing 29 shows the global attributes defined in case of the shape application.

```
Globals:
  Function:
    Runtime: 'nodejs14.x'
    Timeout: 15
    MemorySize: 256
    Layers:
      - !Ref DependenciesLayer
      - !Ref SharedLayer
```

Listing 29: The shared attributes between all functions defined within the same template. In this case, it specifies the runtime required for the function execution, the timeout in seconds for a Lambda execution, the memory size of the functions, and the referenced Lambda layers.

The last missing piece of the setup is the definition of the Lambda layers. Listing 30 shows the definition of the shared layer resource (the dependency layer's definition is the same, just with a different name and code directory). It specifies the name of the layer, the folder in which I store the code that is going to be part of the layer, and the compatible runtimes. This runtime version must match the runtime version of the serverless functions (where it is referenced).

```
SharedLayer:
  Type: AWS::Serverless::LayerVersion
  Properties:
    LayerName: 'sharedLayer'
    ContentUri: '../../package/shared'
    CompatibleRuntimes:
      - "nodejs14.x"
```

Listing 30: Resource definition for the shared Lambda layer. This code specified in this layer can be used in every Lambda function defined within the same template.

4.5 Deployment

In the previous sections, I have presented the files related to the serverless web service and the hosting it on the Lambda platform. The last step to make the shape web service available to our clients is deploying it using the previously defined resources.

I have shown the two scripts file that we can utilize for building and publishing the project (in Section 3.3.2), package.sh and publish.sh. These shell scripts can be part of an automated build pipeline, or I can execute them manually (which I did for the case study, I have not created a deployment pipeline). Since I am working with SAM templates this time (compared to CloudFormation templates in the traditional approach), I am using the CLI tool part of SAM to do the deployment. It performs the transformation of the SAM template into a matching CloudFormation template, and it creates the changeset that it must execute to create/update/delete the resources on the Lambda platform.

In case of an update, SAM is going to update only the resources that were touched, which can speed up the deployment process. After the process is finished, the clients can use the DNS name of the HTTP API Gateway created for my application to reach the shape web service.

4.6 Summary for the Serverless Approach

In this section, I have presented the migration of the shape web service introduced in Section 3 to the Lambda platform from Amazon Web Services.

I have discussed the changes required in the architecture and in the underlying source and infrastructure code, the tools used for the deployment, and some Lambda platform specific optimizations (like sharing source code between serverless functions using Lambda layers, shown in Section 4.3.1.1). For the code complexity differences between the two approaches (traditional and serverless), refer to Section 5.4. I am discussing my conclusions about my case study in Section 6.1.

5 Evaluation

In the following section, I am going to evaluate the applications that I have created in Section 3 and Section 4. The most important metric that I am interested in is the response times from the web services, but I am going to follow a set of evaluation criteria that enables me to do a comparison of the two approaches based on multiple aspects. My evaluation criteria between differently hosted services are the following:

1. Response time differences.
2. User experience differences.

3. Complexity of infrastructure differences.
4. Cost differences.

Note that there are other differences that I could reveal by additional evaluations, for example, scalability differences under certain traffic size scenarios. This scalability aspect is not discussed in this Thesis since there is already a great study that reflects on these differences: As I have pointed out in Section 2.3.1, [54] provides extensive research about the scalability differences of an application hosted traditionally and on the serverless platform.

5.1 Response Time Evaluation

In what follows, I am going to manually evaluate and compare the response time differences between the traditionally hosted and serverless applications that I created in my case study.

5.1.1 Time Measurement Approach

5.1.1.1 Testing Plan

My plan is to evaluate the response times from the traditionally hosted application first, then the application hosted on the Lambda platform, for each endpoint that the web services provide. I am going to visualize the results to ease the understanding of the differences between the two variants.

Note that the results depend on the service configurations (RAM and vCPU in case of the traditional, RAM in case of the serverless approach), the EC2 instance type (see Section 3.4.2), and my networking conditions. I am going to send multiple requests and calculate the minimum, maximum, and average values of the response times.

5.1.1.2 Execution Environment and Measurement Tools

For testing and measuring the response times of the web services, I am going to use the curl [65] command-line tool. It is a powerful tool for transferring data from the command-line. The version of curl used in the evaluation is 7.77.0. Listing 31 shows an example usage and output of curl.

```
curl -X GET https://y203aq4qol.execute-api.eu-central-1.amazonaws.com/shapes \
  -w %{time_total} -s -o /dev/null
0.418009
```

Listing 31: An example execution of the curl tool and the result of the request. -X specifies the HTTP method to be executed, then comes the target URL. On the next line, -w tells curl to extract the total time from request to response, and the additional -s and -o flags hide the exact response returned from the service (since I am only interested in the timings).

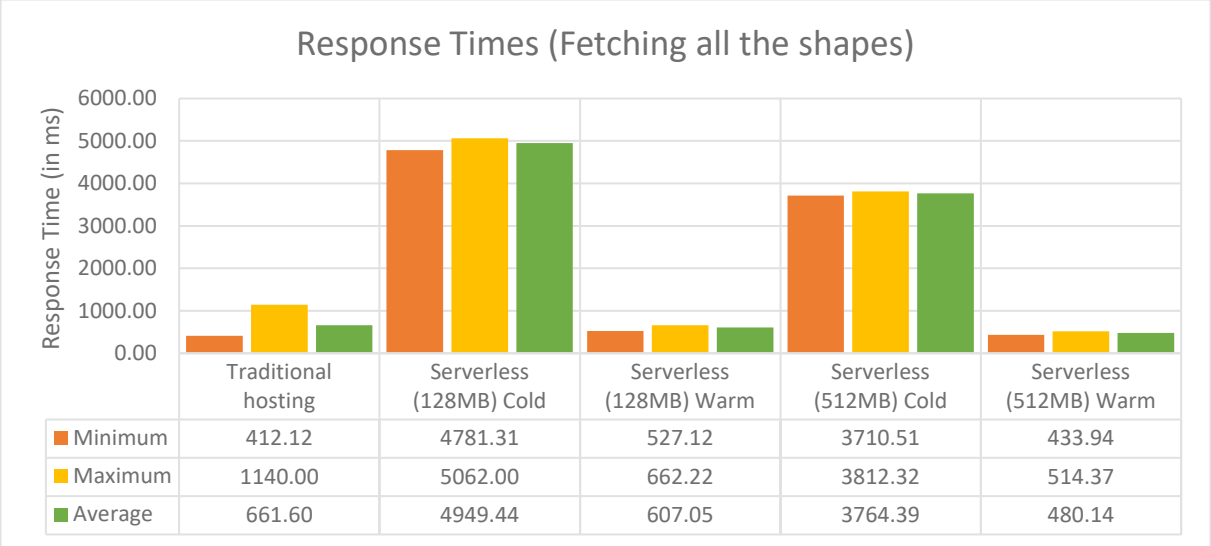
5.1.1.3 Testing the Response Times with Increased RAM in the Lambda Functions

Based on the results presented in [77], there is a confirmed hypothesis that increasing the RAM for a Lambda function is going to result in lower cold start times for that function. I want to take the chance to test this hypothesis myself. I am going to run the response time measurements for two different serverless function setups, one where the functions have 126MB of RAM and the other where they have 512MB configured. I expect that the increased RAM is going to result lower response times.

The traditionally hosted application is configured with 512MB RAM and 256 units of (virtual) CPU, where 1024 units of CPU means that the service would have 1 dedicated CPU to itself. 256 units of CPU is like a single core from a four-core processor.

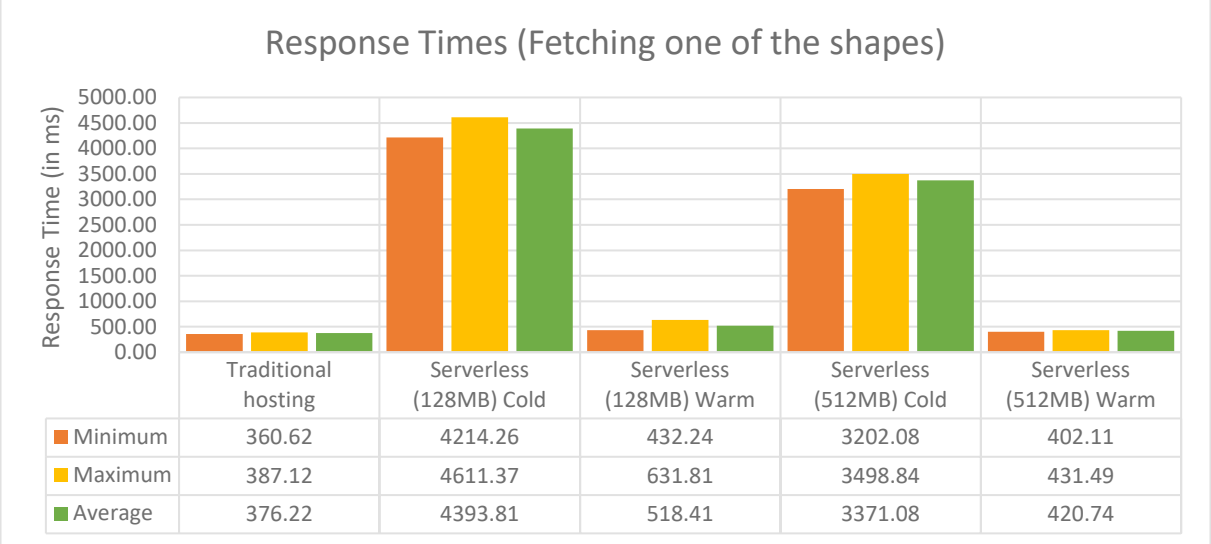
5.1.2 Measurement Results

Figure 22, Figure 23, Figure 24, Figure 25, and Figure 26 shows the results of fetching all, fetching one, creating, updating, and deleting shapes, respectively. The results are further discussed in Section 6.2.



	Traditional hosting	Serverless (128MB) Cold	Serverless (128MB) Warm	Serverless (512MB) Cold	Serverless (512MB) Warm
Minimum	412.12	4781.31	527.12	3710.51	433.94
Maximum	1140.00	5062.00	662.22	3812.32	514.37
Average	661.60	4949.44	607.05	3764.39	480.14

Figure 22: The response time measurement results of fetching all the shapes currently in the database (23 shapes). The numbers show that even the traditional hosting can give relatively long response times (but usually it is stable). The response times if the serverless functions are taking long when the serverless functions are in a cold state. After that, they also give stable response times. The serverless function with the increased RAM nearly matches the response times from the traditional approach.



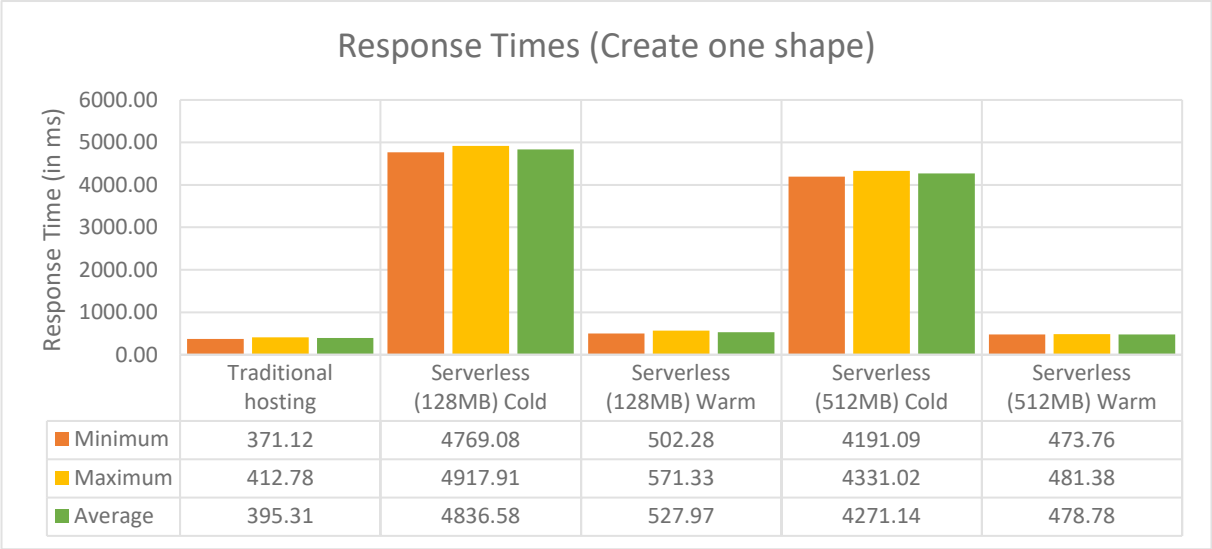
	Traditional hosting	Serverless (128MB) Cold	Serverless (128MB) Warm	Serverless (512MB) Cold	Serverless (512MB) Warm
Minimum	360.62	4214.26	432.24	3202.08	402.11
Maximum	387.12	4611.37	631.81	3498.84	431.49
Average	376.22	4393.81	518.41	3371.08	420.74

Figure 23: The response time measurement results of fetching one of the shapes. The patterns are similar to what I have measured and shown in Figure 22. The traditional approach has stable response times as well as the serverless functions when they are in a warm state. The cold start times increase the response times.

The results show that the cold start times in case of the serverless functions indeed impact the response times in a negative way. Note that these cold start times do not happen too often when the web service is constantly used by the clients. If the service is accessed regularly, the clients can expect the response times shown in the “Serverless Warm” columns.

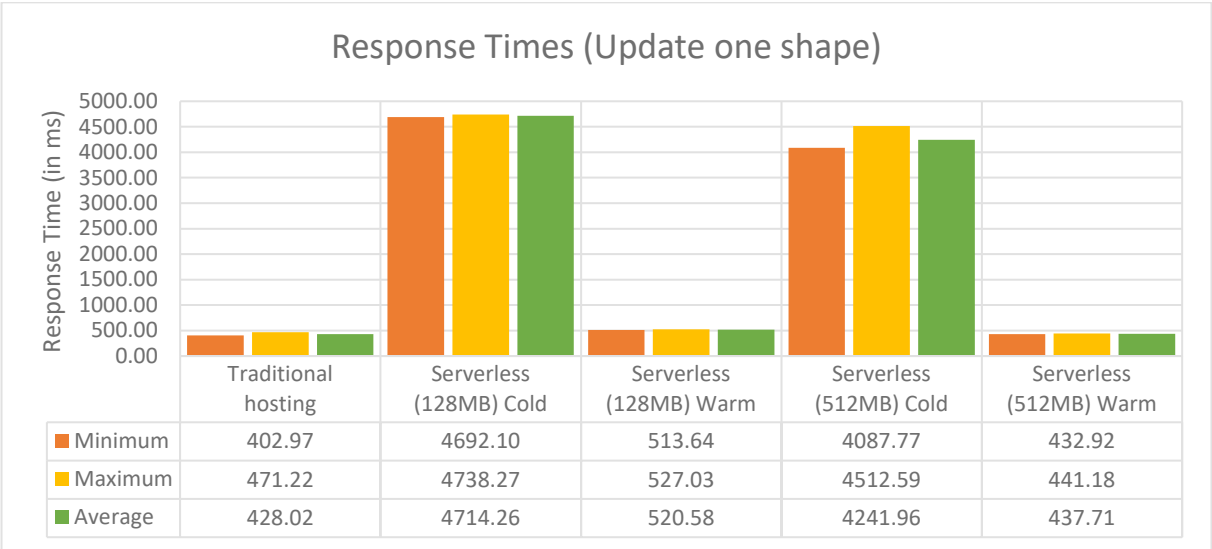
When the serverless functions are in a warm state, the response times are at nearly the same the I measured for the traditional approach, that is, they are short and steady. Note that there were no large differences between the response times of the operations provided by service (fetch, create, update, delete), this is since the payload size is similar in each case. For example, if I would have multiple hundreds of shapes in the database, the web service would return a larger payload which is affecting the response times.

The results are promising for the serverless functions. Considering that cold start times happen only occasionally, they can keep up with the performance provided by the traditional approach.



	Traditional hosting	Serverless (128MB) Cold	Serverless (128MB) Warm	Serverless (512MB) Cold	Serverless (512MB) Warm
Minimum	371.12	4769.08	502.28	4191.09	473.76
Maximum	412.78	4917.91	571.33	4331.02	481.38
Average	395.31	4836.58	527.97	4271.14	478.78

Figure 24: The response time measurement results of creating a shape. The pattern is similar to what I have measured and presented in Figure 22 and Figure 23. After the serverless functions get into a warm state, the response times from them match the response times from the traditional approach.



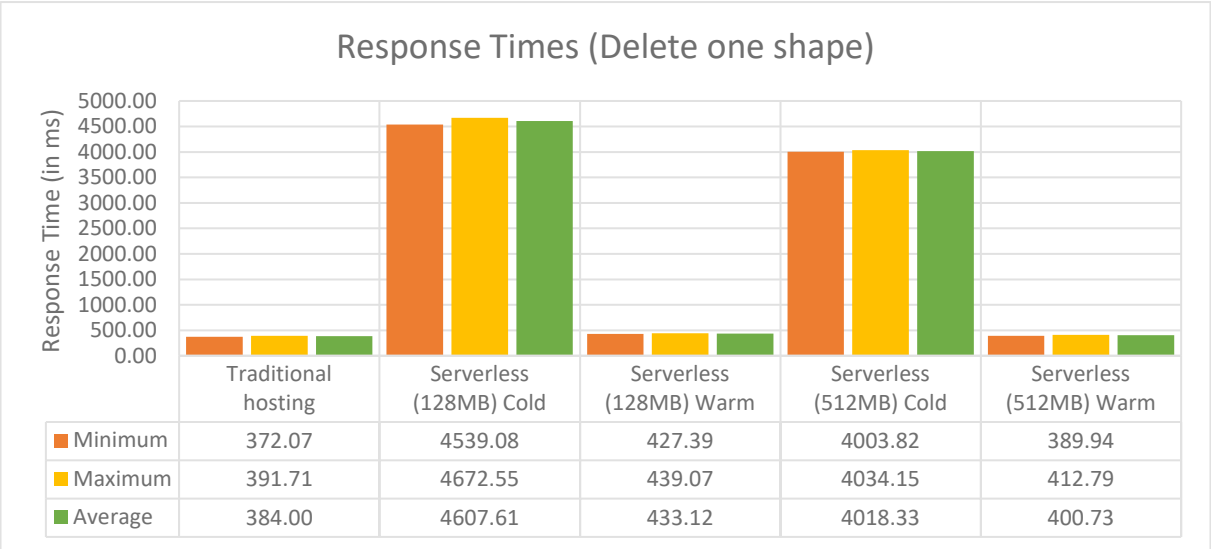
	Traditional hosting	Serverless (128MB) Cold	Serverless (128MB) Warm	Serverless (512MB) Cold	Serverless (512MB) Warm
Minimum	402.97	4692.10	513.64	4087.77	432.92
Maximum	471.22	4738.27	527.03	4512.59	441.18
Average	428.02	4714.26	520.58	4241.96	437.71

Figure 25: The response time measurement results of updating a shape. The pattern matches what I have measured and presented in Figure 22, Figure 23, and Figure 24. Only the serverless functions with cold start times have long and unsteady response times.

5.2 User Experience Evaluation

In this section, I am going to evaluate the user experience differences between the web services from the traditional and serverless approaches by creating a user-facing application that is connected to them.

I expect that the cold start times of the Lambda functions are going to negatively affect the overall user experience of the web application that I am going to create and present for the evaluation. Also, I want to investigate if I can effectively hide the lengthened response times by asynchronously loading some parts of the application.



	Traditional hosting	Serverless (128MB) Cold	Serverless (128MB) Warm	Serverless (512MB) Cold	Serverless (512MB) Warm
Minimum	372.07	4539.08	427.39	4003.82	389.94
Maximum	391.71	4672.55	439.07	4034.15	412.79
Average	384.00	4607.61	433.12	4018.33	400.73

Figure 26: The response time measurement results of deleting a shape. Deletion is a relatively fast operation for both the traditional and serverless approaches. The pattern matches what I have measured and presented in Figure 22, Figure 23, Figure 24, and Figure 25. After the initial cold start times of the serverless functions, they give steady and low response times that are very close to the numbers from the traditional approach.

I have a hypothesis that if I can load the page quickly and then show a loading indicator until the shapes are fetched from the backend, the negative effect on the user experience can be minimized. I plan to confirm or reject this hypothesis by getting feedback regarding the overall experience from a few preselected users. Note that this hypothesis is depending on the web application that I create for the evaluation. The results would probably differ with a different user interface, another programming language, another framework, etc.

5.2.1 Provided Functionality

The test web application enables users to interact with the shapes in the system. They can see the available shapes in the database, modify them (size, type, color), delete them, or create new ones.

For implementing the web application, I have used the NextJS framework [60] (discussed in the following section). The applications created with this framework can be considered as a mixture of traditional and single-page web applications (SPAs) [115] [116]. SPAs use only minimalistic HTML documents and a rich set of JavaScript code to populate this document with further content in the client's browser. They heavily rely on DOM [117] manipulation [118], that is, they execute JavaScript functions (such as API calls to retrieve some form of data) resulting in updates in the DOM of the webpage. JavaScript controls everything in this case on the client side, even the switch

between the pages provided by a web application (like a homepage and a purchase page in case of a web shop).

In comparison, traditional web applications prepare and serve complete HTML documents on a request, rendered on the web server that serves the requested webpage (details in Section 5.2.3). Switching to a different page in a traditional web application is going to result in a call to the server that controls the application.

My web application created using NextJS is a mixture of the two approaches, since it uses ReactJS [119] (an SPA framework) under the hood but uses a custom routing (page changing) solution instead of performing the routing on the client side. This way, my application leverages the advantages of both approaches: The application is going to be split into multiple source files based on the pages (which has a huge impact on the page load speeds, as the multiple source files are loaded on-demand), and the pages are populated using JavaScript.

5.2.2 ReactJS and NextJS

An effective way to build a web application nowadays is by using any of the available JavaScript frameworks for frontend development. These frameworks can differ from each other in many areas, but the goal is the same: Give the ability to developers to create applications with high performance and promote reusability by enforcing a certain way of code organization.

The most popular frontend framework [120] [121] is ReactJS from Facebook [119]. It is a component-based framework, which means that the user interface is built up by a component-tree. Figure 27 shows an example design of a web application. The top-level component is the webpage itself, then the page has subcomponents, like the header, footer, and the main area. These components can also have subcomponents, and so on.

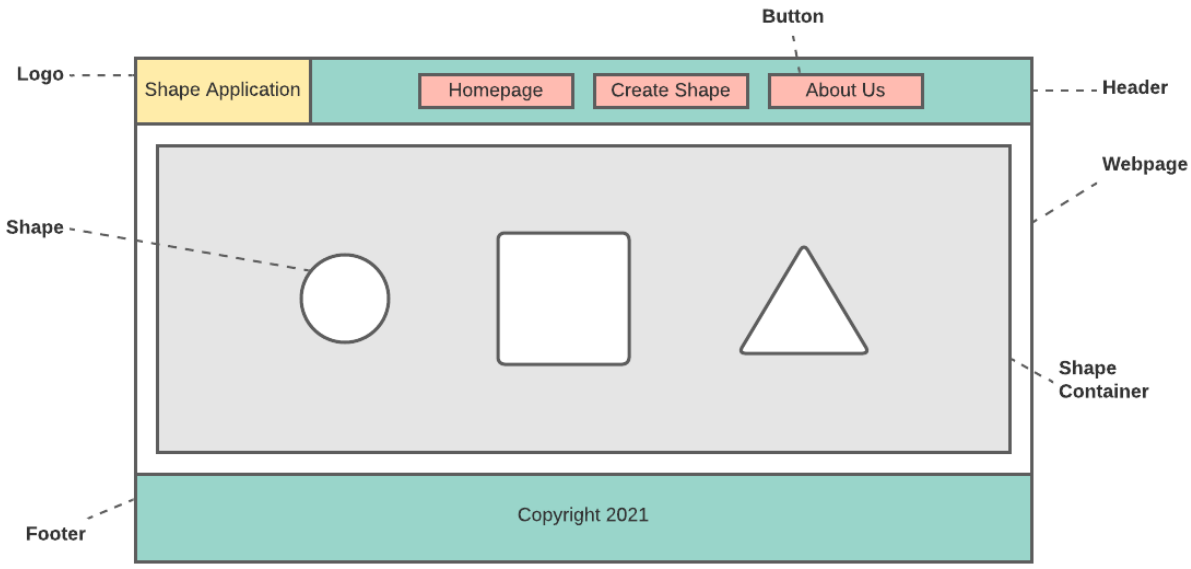


Figure 27: An example webpage that is built up by components. Each component can contain subcomponents. The largest component is the webpage itself, and the smallest components are the atoms (button, shape, text).

The components are described in ReactJS's custom JavaScript syntax called JSX [122]. It provides an HTML-like syntax to create a component, which ultimately becomes plain JavaScript code after compilation.

The root component of the application (that is, the largest component) hooks to a element in an HTML document and renders the application when a client application loads this document (this is how single-page applications initialize themselves).

There are other web frameworks that build on top of ReactJS, providing additional tools and plugins which further enhance the development experience. One of these frameworks is NextJS [60] by Vercel. In my personal experience, this framework is the most convenient to use when creating ReactJS applications, because it is easy to get comfortable with, has a great library of plugins, and Vercel provides a powerful cloud platform for hosting web applications [59], which is easy to setup with a NextJS application. I have chosen the NextJS framework to develop the web application used for the experimental evaluation.

5.2.3 Server-side Rendering

ReactJS supports two types of rendering approaches: Rendering on the client device and rendering on a web server. Client-side rendering means that a given component's JavaScript code is executed on the client's device (for example, in a web browser), which will result in some HTML output that is applied to the Document Object Model [117] of the page. Figure 28 shows an example HTML document that is already rendered and shows which part in the rendered document connects to a component.

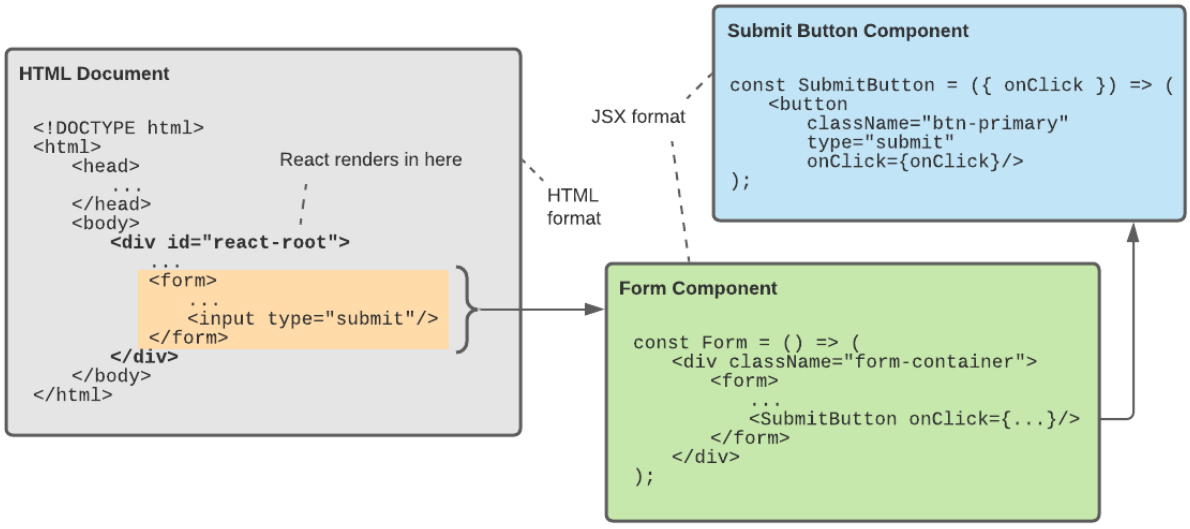


Figure 28: The connection between the ReactJS components and the rendered webpage. The form in the HTML is produced by rendering a component called Form, written using the JSX syntax. The Form component is using another component called the Submit Button component. The Button component takes a mouse click event handler.

Rendering all the components on the client-side is not an efficient approach, since large web applications can put much computing load on the client devices. Also, the speed of processing JavaScript code can differ per device, that is, a smartphone from 2012 might struggle with building a largely interactive web application. Figure 29 shows an example server-side rendering setup, in which the client receives an already rendered webpage.

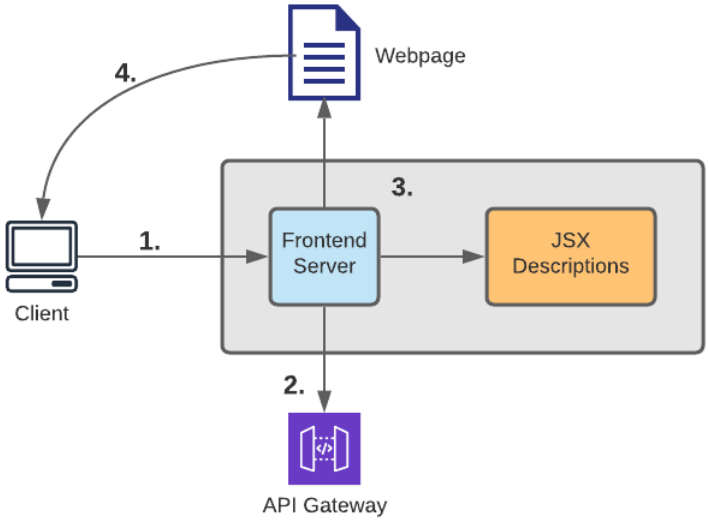


Figure 29: An example server-side rendering setup. 1. The client accesses a webpage, the request for the content is sent to the server that controls the frontend application. 2. The frontend server requests and receives the data that is required to render the page to the user. 3. The frontend server uses the JSX files and the now available data to render the HTML content. 4. The rendered HTML webpage is sent back to the client.

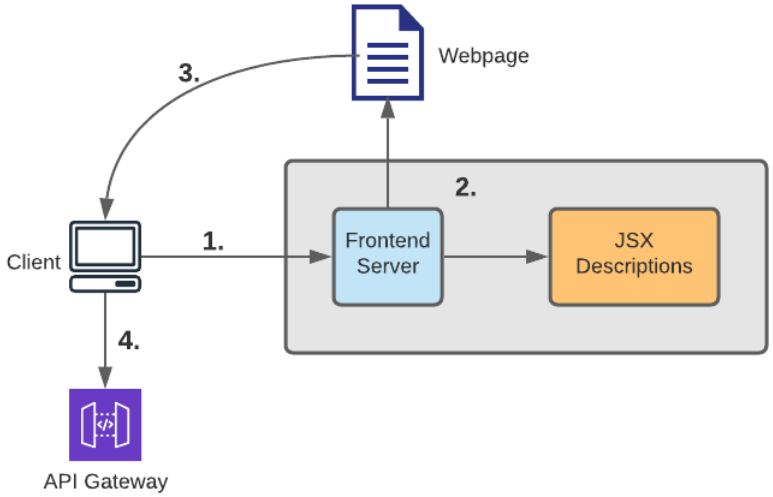


Figure 30: An example hybrid-rendering setup. 1. The client accesses a webpage, the request for the content is sent to the server that controls the frontend application. 2. The frontend server takes the JSX files and renders some parts of the HTML page. 3. The client receives the partially rendered webpage and displays the sections that are not dynamically loaded. 4. The client requests and receives the missing data from the page. As soon as the data arrives, the client renders the remaining components.

Server-side rendering takes the burden of rendering the pages off from the clients. In a modern web application setup, we have a web server that takes requests from the clients (when accessing a page, for example, from the web browser) and renders the ReactJS components before giving back the response. The client receives an already rendered HTML page and the related JavaScript code, and thus the client device does not need to perform the rendering step (or just needs to render some of the not rendered parts partially).

Although server-side rendering is a powerful tool, there are cases when programmers want to render a component on the client side. For example, in the shapes web application used for the experimental evaluation, I want to load and render the list of the existing shapes asynchronously. This is going to result in partially rendered webpages from the web server, and the client must do

some additional rendering after the data becomes available. Figure 30 shows an example hybrid-rendering setup, in which the client must perform rendering on its own (only parts of the webpage).

In the web development field, every website's use case is different. Engineers need decide where to perform rendering parts of the application to achieve the best performance and user experience. There is a fine balance between server-side and client-side rendering for every frontend use case.

5.2.4 User Interface

In this section, I am going to share three screenshots of the client application that I have created for the experimental evaluation. The homepage of the application contains four buttons which take the user to the page variants I have created for the experiments:

- Serverless Lazy: The page is connected to the backend on the serverless platform. The list of shapes is loaded asynchronously. While the shapes are loading, it shows a loading indicator.
- Serverless Regular: The page is connected to the backend on the serverless platform. The whole page is server-side rendered, that is, we do not show anything until the shapes have been fetched from the backend.
- Traditional Lazy: Similar to Serverless Lazy, but the page is connected to the backend that is hosted traditionally.
- Traditional Regular: Similar to Serverless Regular, but the page is connected to the backend that is hosted traditionally.

Figure 31, Figure 32, and Figure 33 are showing screenshots of the application that I have created for the evaluation.

5.3 Feedback From Users

5.3.1 Purpose of Gathering Feedback

As mentioned in the beginning of Section 5.2, I decided to contact a few non-technical persons and ask them to interact with the application. I want to see if they notice the delays caused by the cold start times from the web service on the Lambda platform (presented in Section 4).

I also want to see if the asynchronous shape loading can hide the long response times efficiently. My hypothesis is that showing a loading screen while fetching some parts of the application is better than waiting for a page load that takes seconds.

Note that the amount of feedback is going to be too small to draw a conclusion about the topic at hand in general. Also, the response times are varying based on architectural decisions, technical details, configurations, the web application used in the testing, etc. Nevertheless, it should be a good approach for investigate my hypothesis about this certain use case presented in the Master Thesis.

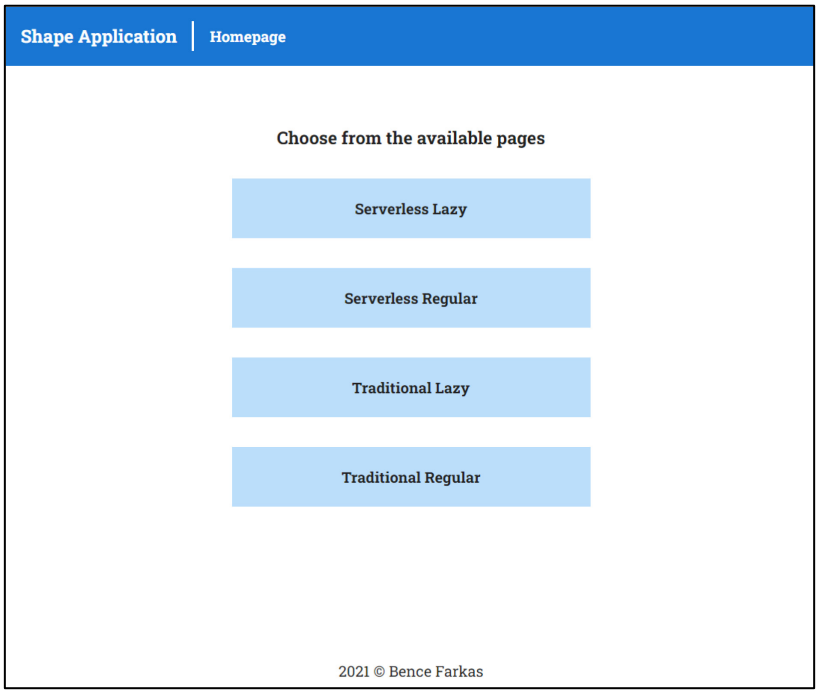


Figure 31: The homepage of the web application. The users can choose from four different variants of the same page. Regular means that the complete shape page is server side rendered, while Lazy means that the shapes are asynchronously loaded from the client-side.

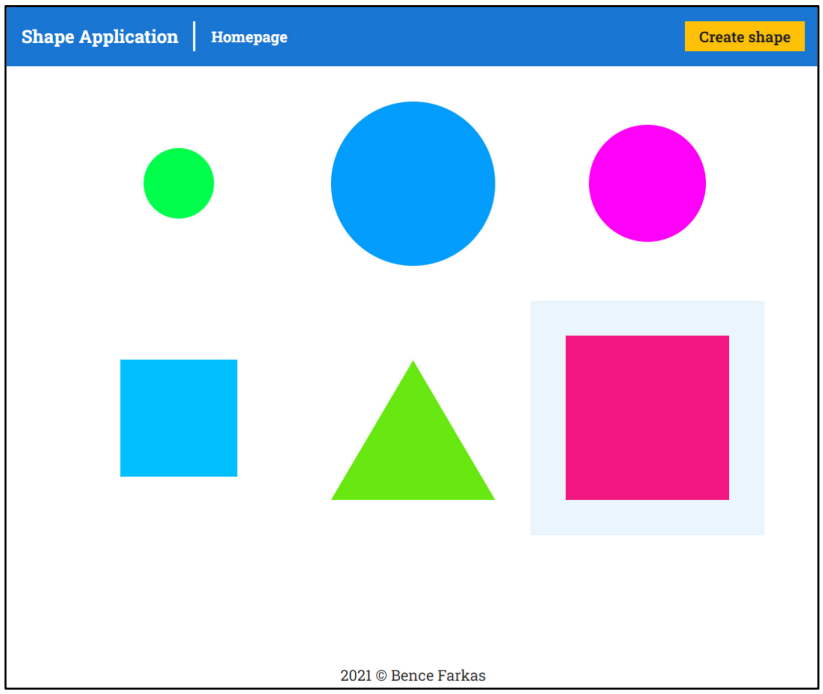


Figure 32: The shapes page. The user can navigate back to the homepage by using the Homepage link. The Create shape button opens a shape creation modal. Double-clicking a shape opens a shape modification modal.

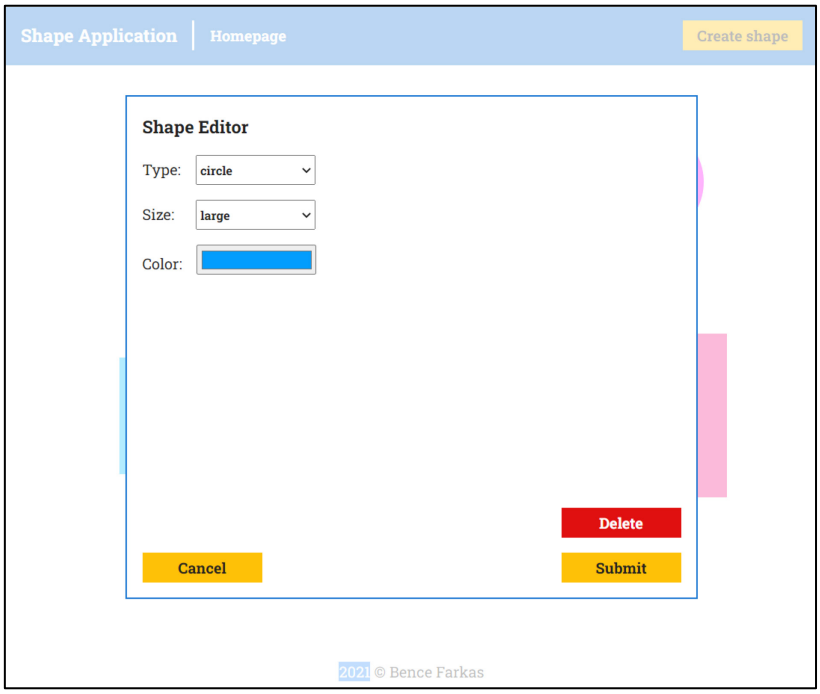


Figure 33: The shape modification modal. The users can delete or modify a certain shape in this view.

5.3.2 The Approach for Getting Feedbacks

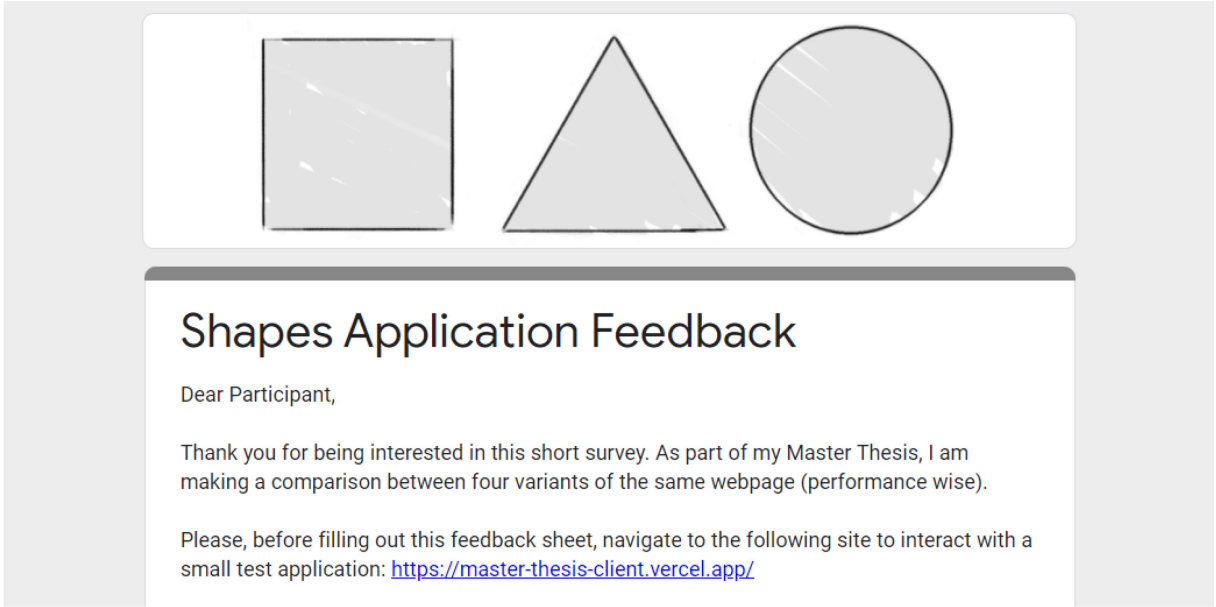


Figure 34: The entry point of the form. It starts with an explanation of the application and some basic instructions about how to interact with it, what to look for.

The form starts with explaining the reasons of the test itself without going into too much technical details. The users are asked to interact with each variant of the application and look for performance trends and anomalies. I ask the users to do the experimenting two times and wait five minutes between the two test runs (to cause another cold start in the serverless functions). I also point out that if they have executed the variants from top to bottom the first time, then do it the other way around the second time, that is, from bottom to top. This way, the cold start times should hit differently.

The four variants (serverless lazy loaded, serverless server-side rendered, traditional lazy loaded, traditional server-side rendered) are named after fruits as Apple, Lemon, Cherry, and Peach. I decided to use fruit names because it is more memorable to non-technical users than phrases like "Serverless with Lazily Loaded Shapes".

After the introduction part in the survey, there are four sections for the four variants with the same set of questions:

- How quick was the page load time for you in case of the XYZ variant?
- How quick were the other interactions in case of the XYZ variant?
- How would you rate the overall experience of the XYZ variant?

The users can also provide textual feedback about a certain variant, but this is only optional.

There is a final section that asks the users to pick a variant as the worst experience and a variant as the best experience. Figure 35 shows an example set of questions for a certain variant.

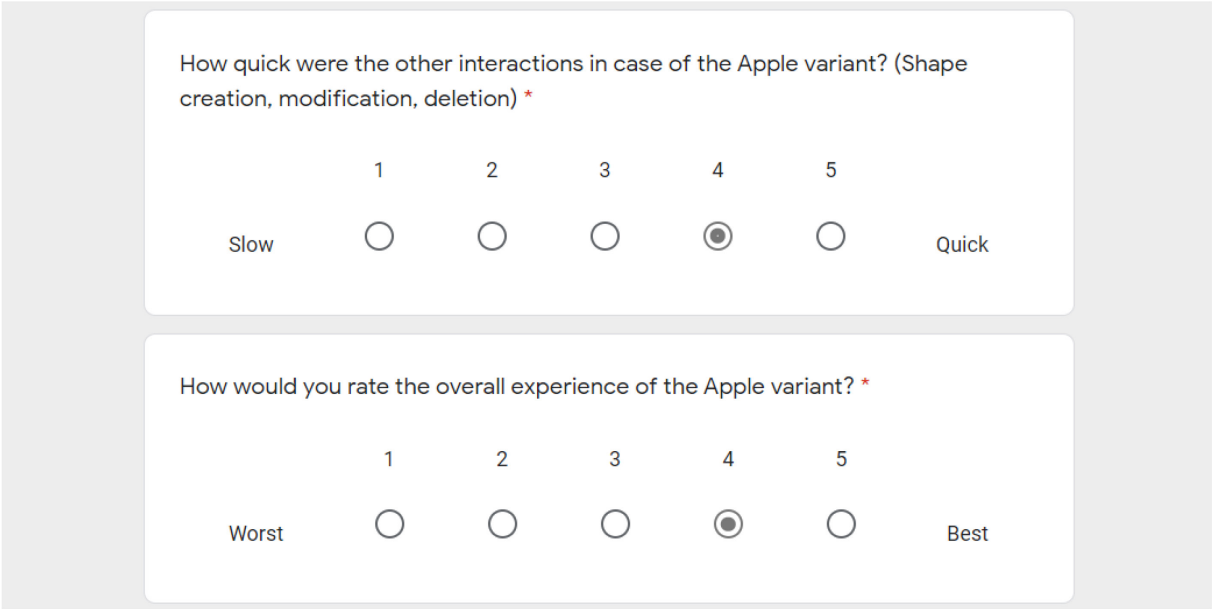


Figure 35: An example set of questions for the Apple variant. There are four sections that contain these types of questions for each variant, so the users can review them individually.

When all the target users completed the survey, I am going to perform a manual evaluation of the results to see their feedback about the experience they had with the different variants.

5.3.3 Evaluating the Feedback from the Users

During a three days long testing period, eleven people have taken the time to interact with the application and fill out the survey.

The results are confirming what I have expected before conducting the tests. Figure 36 shows the sum of the points that each variant received, where the possible maximum were 55 points.

I have also received two textual feedbacks per page, which are further confirming my assumptions:

1. Serverless Lazy:
    1.1. “When first opened the page, the first actions took a little bit more time, but after going through the functionality, it felt more consistent and faster”.
    1.2. “Some lagging when saving the settings of a shape (e.g., changing color or size)”.
2. Serverless SSR:
    2.1. “This was nearly the same as with the Apple [Serverless Lazy] "mode", but I felt it a little bit slower (5-10%) after the initial load time”.
    2.2. “Better than Apple [Serverless Lazy], but it takes at least a few seconds before the page with the shapes is opened after clicking the Lemon [Serverless SSR] button. Processing of requests was a bit faster than in Apple”.
3. Traditional Lazy
    3.1. “This felt quicker right from the start. All the actions were quick and convenient”.
    3.2. “Cherry [Traditional Lazy] gave me the best user experience. Quick opening of page and processing of user requests. Hardly any visible glitch or lagging”.
4. Traditional SSR
    4.1. “Felt performant, could use every feature fast. Same as at Cherry [Traditional Lazy]”.
    4.2. “Peach [Traditional SSR] was second best after Cherry [Traditional Lazy]. Some visible glitches or lagging, but only for fractions of seconds probably. Nearly the same user experience as Cherry [Traditional Lazy]”.

The scores and the textual feedbacks combined show that these eleven persons liked the pages that are connected to the traditionally hosted backend the most, but they have also found the serverless variants usable, especially after the serverless functions were in a warm state.

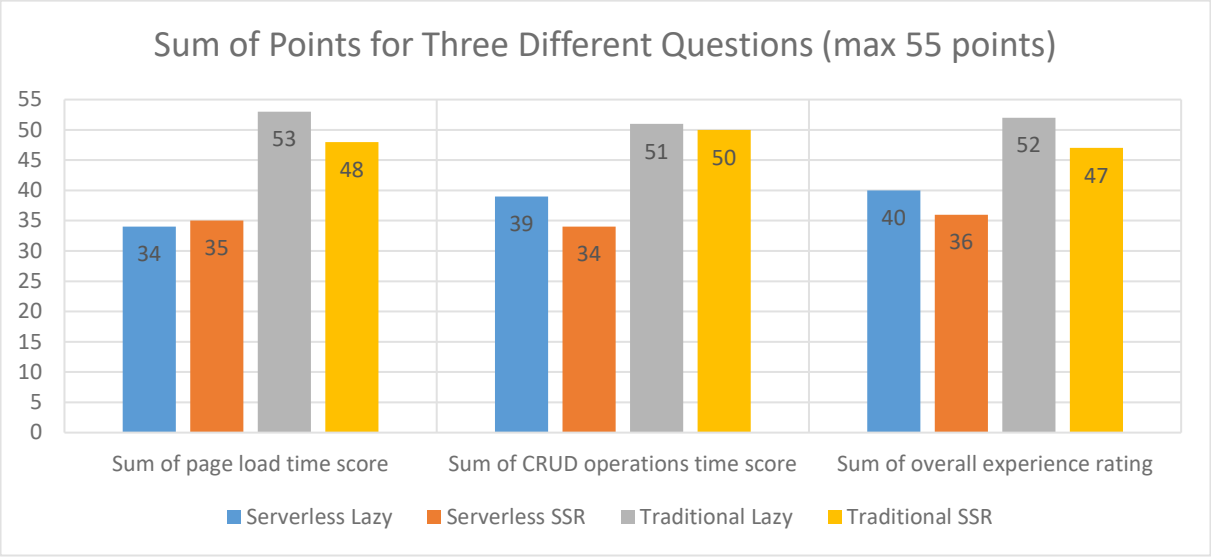


Figure 36: Sum of the points for each page after the testing phase was closed. We can see that the pages that were connected to the traditionally hosted backend received much better scores overall. The Serverless Lazy and Serverless SSR variants received the same score for the page load time test.

At the end of the survey, I have also asked to select the variant with the worst and best user experience. Figure 37 shows the percentage of the variants marked as the “worst experience” and “best experience”.

The textual feedbacks might reflect that people did not like the pages connected to the serverless variant, but this can be misleading. If I look at Figure 36, it shows that the overall points for the serverless pages are quite high. Putting the two variants side by side and making sure that the users run into cold start times made a huge impact on the users. In a real-world scenario, webpages are usually not connected two different types of web services providing the same functionality. My assumption is that if I would have created the frontend application that would have used the serverless approach only, the users would have liked it, nonetheless.

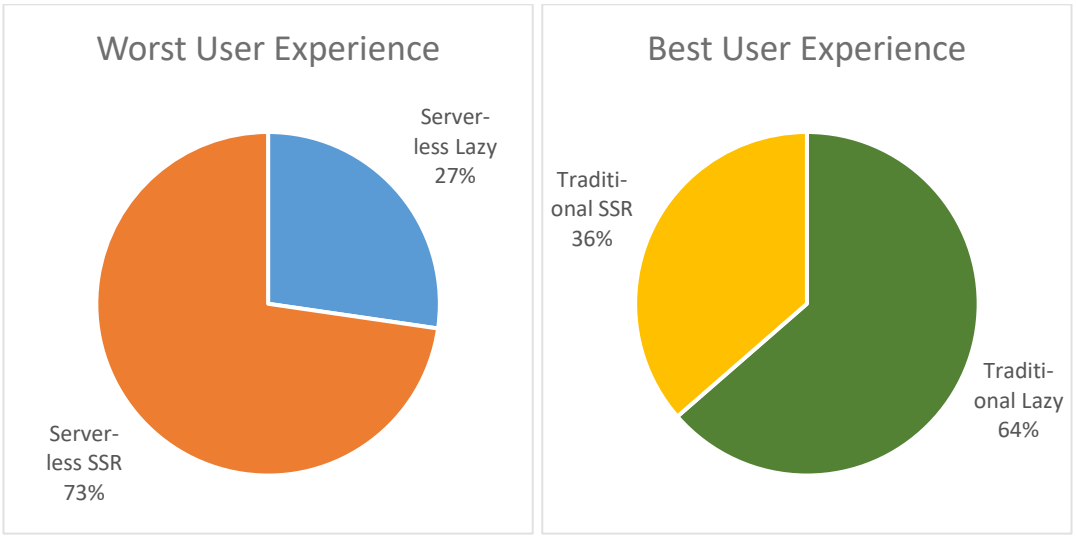


Figure 37: Percentage distribution of the variants for the questions “worst” and “best experience” questions. The “worst experience” variants were always the serverless options, while people always enjoyed using the traditional variants.

5.4 Complexity of Infrastructure Evaluation

5.4.1 Tool Used in the Evaluation

To gain insights in the differences between the codebase of the traditional and the serverless approach, I am using the command-line tool called cloc (Count Lines of Code) [66], version 1.90. This tool is capable of counting files, blank, comment, and code lines, it can determine the types of the source code artifacts used in the project, etc. By using this tool, I can make a comparison between the two variants of the shape web service.

Listing 32 shows an example output of the cloc command, used on the src directory of the traditional application (refer to Figure 13 for the application structure).

```
% cloc src
14 text files.
14 unique files.
0 files ignored.

github.com/AlDanial/cloc v 1.90  T=0.03 s (556.0 files/s, 10324.8 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
TypeScript                      14             33              8            219
-------------------------------------------------------------------------------
SUM:                            14             33              8            219
-------------------------------------------------------------------------------
```

Listing 32: Example output from the cloc tool. The tool can identify the programming languages used in the project, the unique files. The number of lines in the files are summed up and grouped by the type of the line (blank, comment, code).

5.4.2 Source Code Analyzation

In case of the shape web service from my case study, I consider the changes to the original source code as comprehendible. Table 4 shows the statistics for both the traditional and serverless approaches.

The functionality provided by the service module in the original application (which handles the communication to the database) was split up into different serverless functions (create, delete, update, and fetch), and I had to change the way we invoke these functions (see Section 4.2.1.1 for the related discussion). Instead of simply importing the service module and calling the operations directly, I had to introduce a client that makes it possible to invoke Lambda functions synchronously. This resulted in more files and more lines of code in the serverless approach, but the difference in the lines of code between the variants is still not significant.

	Files	Lines		
	TypeScript	Blank	Comment	Code
Traditional	14	33	8	219
Serverless	20	52	12	372

Table 4: The results of file number and line of code analyzation of the source code in both approaches. These are the results from running the cloc tool on the src folders (Figure 13 and Figure 21). The Lines of Code are quite close to each other when comparing the two variants. This is because I have reused as much code from the traditional implementation as possible. Increasing the serverless function granularity in the more realistic approach (see Section 4.2.1.1) resulted in a slight increase in the number of files and the Line of Code.

The number of files and lines of code differences are relatively close to each other. This relates to my approach of implementing the serverless solution: I have reused as much of the original codebase as possible (as described in Section 4.1.1). Since I have increased the serverless function granularity (see Section 4.2.1.1) for my “more realistic” Function-as-a-Service approach, I had to introduce some additional source files, for example, the one that contains the Lambda client module that gives me the ability to invoke one Lambda function from another one. Other than these few additions, the code complexity can be considered equal between the traditional and serverless approaches.

5.4.3 Infrastructure Code Analyzation

The changes in other resources (templates, scripts) were significant between the traditional and serverless approaches. Instead of using two CloudFormation templates [56] to describe the service

and the infrastructure in the traditional approach, I introduced a Serverless Application Model (SAM) template [64] for the serverless web service that reduces boilerplate configurations and describes the functions and related resources in a concise manner.

There was a change also in the bundling of the web service: The traditional solution used a Dockerfile to create an image from which we can start a container in Elastic Container Service, and the serverless approach only requires to move the compiled source code into a certain directory structure, from which the SAM CLI tool could create the appropriate Lambda layers and functions.

The serverless hosting has less requirements that the traditional hosting, most importantly, I did not have to create and manage the networking infrastructure. I did not need to provision my cluster, the internet gateway, the load balancer, the subnets, etc. A huge advantage of the serverless platform is that this part of the infrastructure is completely hidden from the programmers, in a sense that they do not need to deal with it themselves, the cloud provider takes care of the provisioning.

I would not say it is black box architecture either, that is, if programmers have questions about how the architecture works, they can find materials on it online.

Leaving out the infrastructure part is already a great amount of help, but Amazon Web Services even eases the way of configuring the resources related to the web service itself. The traditional approach demands to specify an EC2 instance, a target count of the service, a percentage of maximum containers, the amount of RAM and CPU, the load balancer to use for reaching the containers, etc. There are many parameters that programmers need to finetune (and play around with) when they want to find the balance between the business needs and the costs of their system. In contrast, I could specify less details for my serverless setup before it became executable in the cloud. Table 5 shows the infrastructure code analyzation results of both approaches.

	Files						**Lines**		
	YAML	**JSON**	**Dockerfile**	**Shell Script**	**JavaScript**	**TypeScript**	**Blank**	**Comment**	**Code**
Traditional	2	2	1	7	7	1	63	29	575
Serverless	1	2	0	2	1	1	17	10	228

Table 5: The results of file number and line of code analyzation of the infrastructure code in both approaches. These are the results from running the cloc tool on the aws, scripts folders and other infrastructure related files (Dockerfile, package.json, tsconfig.json) (Figure 13 and Figure 21). The results show that there is a large difference between the number of files and the Lines of Code in the two approaches. The serverless approach required less than half number of lines to handle the infrastructure code.

The results show the importance of Function-as-a-Service. The infrastructure related code is much less in this serverless approach, it has half of the number of lines compared to the traditional approach. I did not have to deal with provisioning the different networking and computing resources described in Section 3.4. The code complexity measurement confirms with numbers that I have felt during the implementation of the services in the case study: Using the Lambda [9] platform is much more convenient than the Infrastructure-as-a-Service approach. There are less resources to configure, and thus there are less resources to maintain. This is a huge advantage of serverless platforms.

5.4.4 Differences in Tooling

The serverless approach is different in a way that I could use Serverless Application Model (SAM) [64] templates instead of CloudFormation ones. SAM is helping programmers with reducing boilerplate code in the resource templates. SAM has a template transformator part, and a SAM template is going to become a CloudFormation template during the deployment process; think of it as a concise way of describing the CloudFormation resources. A valid SAM template can be translated into a valid CloudFormation template, but not vice-versa.

5.5 Cost Evaluation

In the section, I am going to perform a brief cost analysis to compare the costs of the traditional and serverless approach based on the input traffic. Note that these calculations are based on hypothetic scenario inspired by the two approaches presented in my case study.

I am going to show by numbers that function chains already mentioned in Section 4.2.1.2 have a negative impact on the serverless costs at the times of this writing.

5.5.1 Cost Differences Between the Serverless and Traditional Approaches

One advantage of serverless functions is that they are billed based on their execution time and configured size of memory. In contrast, traditionally hosted applications are billed based on the number of resources they are configured with and the time while the applications are running.

In the following, I am going to do a calculation of cost approximations for the two types of hosting, based on the traffic density (number of requests in a month) as the input parameter.

The serverless setup contains an HTTP API Gateway (see Section 1.4.12.2) and a single server function that handles the incoming requests. There are no provisioned functions in the system for keeping the calculations simple (provisioned functions are discussed in Section 6.5.1.2). Note that the cold start latencies are not considered as function execution times, they are not billed, this is going to be an important aspect when dealing with function chains. The billed duration of the Lambda functions is going to be 500 milliseconds and the configured RAM is 512MB for the server Lambda function. AWS Free Tier [123] is not considered in the calculations.

As for the traditional approach, I am going to calculate with an Elastic Container Service cluster with two instances of the shape web service. The configured resources per task are 512MB RAM and 0.25 vCPU. Note that running these tasks and the Elastic Container Service cluster does not generate additional costs, we only need to pay for the EC2 instances on which the tasks are running.

The traffic is distributed via an Elastic Load Balancer [58]. The launch type of the applications is EC2 [96], and the instance type is t2.medium on-demand (2 vCPU, 4GB RAM, with 30GB SSD storage per instance). There are two instances in two availability zones (because of the load balancer), and the instances are running Amazon Linux 2 [124] as the operating system. These on-demand instances are running constantly (no scaling up or down during a day).

The input payload is going to have the size of 512KB in each call.

The additional costs of the Backend-as-a-Service solutions used by the web service is not considered, as these costs should be the same regardless of the hosting type. The calculations are done with the official pricing calculator [125] provided by Amazon Web Services, and thus the

costs used in these calculations are the actual costs of the involved AWS services at the time of this writing.

5.5.2 Applications with Low Traffic

For the low traffic use case, let us consider a web service that receives three million requests per month.

I can predict the costs of the serverless setup as the following:

- There are going to be 3.000.000 HTTP API Gateway accesses in a month, that costs 3.6USD.
- There are going to be 3.000.000 Lambda accesses in a month, that costs 13.1USD.
- The total costs for the month are 16.7USD.

I can predict the costs of the traditional setup as the following:

- The monthly cost for the EC2 instances: 67.74USD for the t2.medium instance type and 6USD for the 30GB storage, that is 73.74USD per month.
- The fixed monthly cost of the Application Load Balancer is 19.71USD.
- Based on the 512KB input payload size and the 3.000.000 requests, the Load Balancer Capacity Units also add 12.29USD monthly (1536GB of data).
- The total costs for the month are 105.74USD.

If I do not consider any cost optimization possibilities to the traditional approach (that add further complexities to the budget planning), the serverless setup can be third of the costs of a traditional setup. If I would increase the number of running tasks or the memory/CPU configuration of an individual instance in the traditional setup, the costs can increase to even higher numbers.

5.5.3 Applications with High Traffic

For the high traffic use case, let use consider a web service that receives fifty million requests per month.

I can predict the costs of the serverless setup as the following:

- There are going to be 50.000.000 HTTP API Gateway accesses in a month, that costs 60USD.
- There are going to be 50.000.000 Lambda accesses in a month, that costs 218.33USD.
- The total costs for the month are 278.33USD.

I can predict the costs of the traditional setup as the following:

- The monthly cost for the EC2 instances: 67.74USD for the t2.medium instance type and 6USD for the 30GB storage, that is 73.74USD per month.
- The fixed monthly cost of the Application Load Balancer is 19.71USD.
- Based on the 512KB input payload size and the 50.000.000 requests, the Load Balancer Capacity Units also add 224.51USD monthly (25600GB of data).
- The total costs of the month are 317.96USD.

The traditional setup has nearly turned the balance. If I keep increasing the traffic, there is going to be a turning point where the traditional setup costs equal or less to the serverless setup. An important thing to notice is that the cost of the EC2 instances did not change, it is separated from the traffic – although it might be necessary to increase the instance count or the resource configurations of the instances to handle the increased amount of requests.

5.5.4 The Effect of Function Chains

I have already presented the phenomenon of function chains in Section 4.2.1.2. Sequentially chained serverless functions are going to yield increased costs. As I have already pointed out in Section 2.4.3, the current serverless platforms have a few shortcomings that keep them from becoming the go-to ways of application hosting in the cloud. One of the shortcomings is that there is no way for serverless functions to signal each other when they have data to share between themselves. This shortcoming is discussed in [1].

For example, if function A depends on data from function B, there is no efficient mechanism for B to signal A when the data is ready. In the case of sequential Lambda chains, one serverless function must wait for the results from another serverless function before it can return a response to the client, when the communication between the serverless functions is synchronous.

If the invoking serverless function could step into a "sleeping state" in which it would stop generating costs until the results from invoked serverless function are ready, it would solve the cost issues. But with the current serverless platforms, there are no such solutions, and thus there are no ways to solve the function chains.

Let us consider a small example to simulate how the function chains increase the costs of a web service, similar to what I have presented in my case study. I have predicted in Section 5.5.2 that a serverless setup with an HTTP API Gateway and a server Lambda function costs three times lower than a traditional setup with custom-managed resources. Figure 38 provides an overview of an example setup for which I am going to perform a cost prediction now.

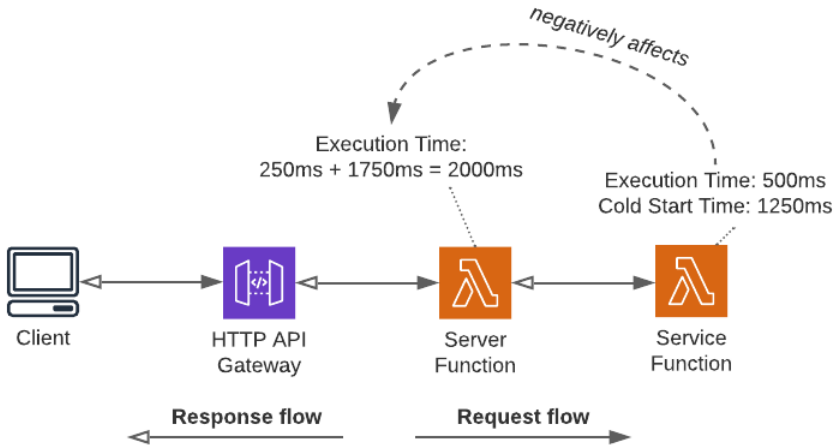


Figure 38: A serverless function setup which contains a function chain. The server Lambda function itself has an execution time of 250ms, but this is increased with 500ms at minimum, since this Lambda invokes the service Lambda synchronously. In the worst case, when the service Lambda function is not in a warm state, the execution time of the server Lambda is increased by 1750ms.

To perform calculations again with the input numbers presented in Section 5.5.2, this time with function chain included, I need to add an approximation to represent the number of cold starts

that I am going to run into with the service Lambda function (from Figure 38) in a month. Let this number be the 30% of all requests in the month, that is, 900.000 requests. When I adjust the calculations of the low traffic use case based on what I presented in Figure 38, then these are the results:

- There are going to be 3.000.000 HTTP API Gateway accesses in a month, that costs 3.6USD.
- There are going to be 3.000.000 Lambda accesses in a month for the server Lambda.
    - 900.000 requests are going to run into cold starts of the service function, resulting in 2000ms total execution times, which costs 50.60USD for the month.
    - 2.100.000 requests are going to run into warm starts of the service function, resulting in 750ms execution times, which costs 19.35USD for the month.
- There are going to be 3.000.000 service Lambda accesses in a month, that costs 13.10USD.
- The total costs for the month are 86.65USD.

Although this estimated monthly cost is based on rough estimations, function chains have huge impacts on the overall costs. If engineers want to keep even the smallest chains in their setups (for architectural reasons), they should consider using provisioned functions at least for the serverless functions that are invoked synchronously. Provisioned functions are discussed in Section 6.5.1.2.

Function-as-a-Service clearly has huge potential in it. In certain cases, or with certain architecture setups, using serverless functions is already more cost efficient than then constantly running computing units and clusters in the traditional approach.

6 Conclusion

6.1 Migration Between the Platforms, Architecture and Source Code Differences

My conclusion is that moving the shape web service to the serverless platform came with low efforts. This statement might be a bit biased since I already had experience with the Lambda platform. It may take a bit longer when the engineer who is carrying out the migration never tried encountered Function-as-a-Service solutions, but after the initial learning curve, the migration can be delivered quickly.

If there is uncertainty in a company about moving a traditionally hosted web service to a serverless platform, my advice would be to test it out: Create a prototype as quick as possible with the least amount of changes to the original service, like my “minimalistic approach” in Section 4.1.1, and then evaluate the performance, the costs of the system. If it turns out to be a good decision, then there are many possibilities to move forward: Either refining the serverless setup approach further (like my “more realistic” approach, or rewriting the application to use a REST API Gateway (see Section 1.4.12.1) instead and getting rid of the custom web service implementation (using the capabilities of REST API Gateways), etc.

6.1.1 Improvements Compared to the Traditional Approach

In my opinion, the most significant improvement compared to the traditional approach is the way I could describe my infrastructure, the resources I needed to create to host the web service. Since

the cloud provider takes care of the infrastructure in case of the serverless functions, there are significantly less infrastructure resources that I need to care about. From the hardware perspective, I only needed to adjust the RAM to increase the capabilities of a single function, and I did not have to deal with EC2 instance types, RAM, and CPU configurations for the Elastic Container Service tasks.

The main promise of serverless is that engineers can host their applications without thinking in terms of servers; There is no need to create EC2 instances, Elastic Container Service clusters, Internet Gateways, etc. They can instead focus on delivering the functionality that matters to their clients. This promise is fulfilled, based on the experience that I have gained, I can safely say that creating and maintaining a Function-as-a-Service-based application is a lot less tedious than maintaining a traditional Infrastructure-as-a-Service approach.

From the source code perspective, I think it does not make sense to think in terms of improvement. Writing serverless functions requires a slight change in the programmers' mindset, since they need to think in terms of functions and data-driven architectures. It is different, but not necessarily better. For programmers who are coming from a JavaScript or Python background, it can be easier to understand this type of architecture, since these languages are built around functions and function compositions, but programmers with object-oriented skills may need to adjust their mindset a bit to leverage the strengths that the platform provides.

The code complexity evaluation in Section 5.4 shows that the source code became a bit larger in case of the serverless application, but this relates to my personal coding style of writing serverless functions. The key point is the large difference in the infrastructure code, both in the number of files and in the lines of code. Using the Lambda platform and the SAM template enables engineers to write short and concise infrastructure definitions.

If I would have to pick between the two approaches, I would choose the serverless approach. I enjoy using the Lambda platform. I can second to what the researchers in [1] predict: The serverless approach is the future of the cloud. If the cloud and system engineers can resolve the shortcomings of the current system (described in Section 2.4), it is going to be the go-to way of interacting with the cloud.

6.2 Response Time Differences

In Section 5.1, I have evaluated the response time differences between the two web services. In case of cold function starts, the response times are prolonged. When the functions are in a warm state, the serverless response times are matching the values from the traditional approach, which is in my opinion, is a great result.

The more client uses the web services, the more functions are going to be in a warm state. As we can see from the study in [54], the serverless applications scale well: They can quickly react to an increase or decrease in the traffic. As a comparison, the tasks running in Elastic Container Service take time to scale out and scale down, and thus there can be a situation in which the clients are facing longer response times (or no responses at all if every running instance of the web service is overwhelmed with handling other the requests).

It is not possible to draw general conclusions based on the web service that I have created in my case study because of the limited scope. However, at least for web services in my case study, I can say that other than the occasionally longer response times caused by cold start times in serverless functions, the serverless application is the better choice in my opinion. Since I cannot give a

general conclusion, I would recommend doing a use case analysis before moving a web service to the serverless platform, which should contain traffic, cost, and resource analysis.

6.3 User Experience Evaluation

In this section, I have presented a small frontend application that provides a user interface connected to the two web services. This interface enabled me to evaluate how the response times from the differently hosted services affect the user experience of a front-facing application. Note that this evaluation heavily depends on the user facing application itself. I have kept this in mind when doing this part of the evaluation, and my hypothesis is only valid for this single use case. Different frontend applications might handle the communication to the web services differently, they can have different user interfaces, etc.

For the user experience evaluation, I have selected eleven users to interact with my application and I have collected their feedbacks using a short survey. The users were asked to participate in the survey after one another, because I especially wanted them to run into cold start times at least once. The results are promising for the serverless application. As I have expected, running into cold starts can cause a bad experience for the users, but after the functions get into a warm state, the user experience feels like with the traditionally hosted web service (which gave consistent results).

My hypothesis about that the cold start times cause a slightly worse experience is confirmed (for my use case and setup), but the scoring from the survey shows that it only has a minor impact. My conclusion is that using Function-as-a-Service based web services can be great choice even when directly connected to user-facing applications. I think if I would have presented a frontend application to the users which would have been only connected to a serverless API (living out the traditionally hosted service), the users would have enjoyed using the application nonetheless, regardless of the cold start times (but this is just an assumption).

6.4 Cost Differences

In Section 5.5, I have done a hypothetical, brief cost analysis of two different setups. If engineers want to optimize their budget, it is always a good idea to do traffic and cost analyzations of their applications. Applications that have low to medium traffic can cost less when hosted on the serverless platform since the billing for the serverless functions is per execution-based. Applications that have medium to high traffic are may be suited with a traditional hosting approach, but as I mentioned, it depends on the use case. There are many variables that effect the costs, like payload sizes, number of requests.

When companies have already existing web services hosted traditionally, I would suggest looking into cost optimization possibilities [126] with the traditional approach before planning a migration to the serverless platform (solely for cost reasons).

6.5 Guidelines

In this section, I am going to discuss a few guidelines that I have created based on the experience that I have gained while working on the applications presented in my Master Thesis.

6.5.1 Considerations

6.5.1.1 Limitations Of Serverless Applications

Although it is a good architectural decision to create web stateless web services, that is, where the sequential requests do not depend on the outcome of each other, you might have an existing application that holds a state.

When having a temporary data store is a must-have in your application, you should consider leveraging a data storage option that your cloud provider supplies. For example, Amazon S3 [47] is a good option for storing your data when writing Lambda functions. Unfortunately, there is no best solution for data storage for now, all the available data storage options come with drawbacks. For a detailed analysis of this problem, refer to [1].

Note that this consideration only applies to your data objects. It is a nice pattern to keep parts of your applications running between invocations, for example, the client objects that handle communication to external services (like DynamoDB). There is no need to instantiate these clients on every invocation since they do not hold intermediate data, they are only used for managing your requests.

Listing 33 shows a JavaScript example about using the singleton pattern for the DynamoDB client object. This object is going to be instantiated and configured only once per Lambda function execution.

```
let dynamoClient: DynamoDBClient;
const getDynamoClient = () => {
  if (!dynamoClient) {
    dynamoClient = new DynamoDBClient({ region });
  }
  return dynamoClient;
};
```

Listing 33: The DynamoDB client used in the serverless approach is defined only once per serverless functions. Using the singleton pattern, we do not need to reconfigure the client on every Lambda invocation.

When you have a server that stores files on the local filesystem of your server, it can also be considered as a stateful behavior, which is not suitable for serverless applications as already discussed in the previous section. Use storage systems like Amazon S3 for storing your files in long-term.

There is an exception: If you need to store a file temporarily while processing a request, you can use the file system of the machine that executes the serverless function. For example, if you have an application that provides an image upload feature which applies filters to an image, you can use the file system directly to handle your multipart upload request. The image is going to be available at least until the request/response cycle ends.

However, this can be generally considered as bad practice with serverless applications. As suggested before, if you need to handle some form of file management, you should check the data storage options supplied by your cloud provider. Only use the local filesystem when there is no other (efficient) way to handle your file-related operations.

You might have an ecosystem that cannot has a web service that can occasionally give responses after multiple seconds. This can be any time-critical application, for example, systems used by railway companies. I would recommend avoiding purely serverless based solutions for these use cases.

6.5.1.2 Provisioned Functions

Amazon Web Services presented a new feature called Provisioned Functions in 2019 [127]. With this feature, you have the possibility to keep a certain amount of your Lambdas in a "warm" state, that is, you can make sure you will not run into long response times caused by the cold start latencies.

The price for the provisioned function depends on three components, a) the time duration that you enable it for a certain function, b) the memory configured for the target function, and c) the amount of concurrency of the target function.

For example, the Lambda functions presented in Section 4 can be configured to run with 512MB RAM. We can keep five containers running from 8:00 to 17:00 by enabling the Provisioned Functions feature for the function to handle an increased payload during office hours.

This feature might be a good option for your use case if you need to keep your response time low for every request. I recommend doing an evaluation of your traffic for one or two days, then you can come up with a feasible configuration for Provisioned Functions. Note that this Master Thesis is not giving more attention to this feature as it reduces the advantages of serverless computing in general. If you decide to use Provisioned Functions, you need to do additional planning and you are no longer pay per execution only – Provisioned Functions are generating costs while they are active, which makes it a bit like the traditional hosting approach (in terms of the cost model).

6.5.2 Leverage Backend-as-a-Service Options

From my personal experience, if you decide to host your application on a serverless platform like Lambda [9], you might want to use other Backend-as-a-Service solutions from the cloud provider's supply. I have already discussed in Section 2.1.1 the vendor lock-in issue, which means that when you may want to switch your cloud provider for some reasons, it comes with the need of adjusting your codebase to fit the environment of the new provider.

Even though I am aware of the vendor lock-in issue, I prefer using the Backend-as-a-Service solutions from Amazon Web Services whenever I need to work with the Lambda platform, because the pieces in their ecosystem fit together perfectly. For example, when designing a serverless application that needs to store data in some way, I could choose from an enormous set of possible technologies to do so. However, this set becomes smaller if we think only about the solutions that a certain cloud provider supply. If there is a decision that the application is going to be hosted in Amazon Web Services, then we should choose from Amazon DynamoDB [48], Amazon RDS [128], or any similar database technology that Amazon Web Services supports.

We already have strong integration possibilities with these services, and the providers are constantly working on making their integrations even better. The conclusion is that if you decide to create a serverless application, actively use the Backend-as-a-Service options of the provider to achieve an efficient outcome.

6.6 Summary

6.6.1 Wrap Up

In my Master Thesis, I have investigated various aspects of migrating traditionally hosted web applications to a serverless platform. There are multiple options to carry out such a migration, and I have picked one that is relatively quick and efficient to perform. I have discussed my workflow

and pointed out important details along the way. As part of the outcome of the Thesis, I have two working web services running in two types of platforms, one hosted in Elastic Container Service from Amazon Web Services, and one hosted on the Amazon Web Services' Lambda platform. I also have a user-facing application to support evaluations and measurements related to the two backend services.

I have started my work with an introduction to the topic which can be found in Section 1. It describes my motivation for the topic: Serverless platforms are becoming increasingly popular because their convenient hosting options, price models, and execution schemes. It looks tempting to grab our existing web services that are hosted in a different way and opt-in for Function-as-a-Service approach, but I assumed that there are some pitfalls or shortcomings of these platforms that makes it hard or impossible to port certain applications. The introduction section also gives an overview of my case study presented in this Master Thesis, and it also gives a brief introduction to technical terms used in the later sections.

In Section 2, I presented several scientific research papers related to my topic. After evaluating the related works, I have built a web service in Section 3. I have implemented the web service in TypeScript using the ExpressJS web framework, containerized it with Docker, and hosted it in Amazon Web Service's Elastic Container Service. I considered this the traditional approach of application hosting.

In the following Section 4, I took the codebase of this web service and migrated it to Amazon Web Service's Lambda platform. I documented the steps along the way, and I have reflected on source and infrastructural code changes. I wanted to make the web service more appropriate to the Lambda platform and refactored out some parts of the original web service into separate serverless functions. This comes with the disadvantage of sequential Lambda chains (which can result in increased costs), but this way the architecture fits more into the event-driven pattern that current serverless platforms represent.

After I had the same web services hosted on two different platforms, I was ready to make comparisons between them. In Section 5, I have presented four different evaluation criteria, and I have evaluated the two web services based on these criteria. I have investigated their response time differences, what differences they have in terms of code complexity, etc. As part of this experimental evaluation, I have created a quick survey and gathered feedback from a set of users who interacted with a user-facing application that I have created and connected to the web services. In Section 6, I have drawn a few conclusions related to my case study and evaluations and discussed these conclusions briefly.

6.6.2 Lookout for the Future

Overall, I can second to what the study in [1] predicts: The serverless platform (especially the Function-as-a-Service methodology) is going to be the default way to interact with the cloud. Based on my case study, I can say that the Lambda platform from Amazon Web Services is already an efficient way to host applications, and I can recommend using it for any existing or upcoming software projects. Although there are some shortcomings of the platform (pointed out in Section 2.4 of my Thesis), these issues seem minor compared to the advantages that the Lambda platform provides: Heavily reducing the infrastructure code needs, scaling in an efficient way, introducing a new cost model based on execution times. I am looking forward to seeing improvements in these shortcomings and to seeing Function-as-a-Service becoming the de facto standard of cloud computing.

Bibliography

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. M. Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," EECS Department, University of California, Berkeley, Berkeley, 2019.

[2] IBM Cloud Education, "FaaS (Function-as-a-Service)," 31 July 2019. [Online]. Available: https://www.ibm.com/cloud/learn/faas. [Accessed 1 March 2022].

[3] M. Roberts, "Serverless Architectures," 22 May 2018. [Online]. Available: https://martinfowler.com/articles/serverless.html#unpacking-faas. [Accessed 28 February 2022].

[4] VMware Inc, "What is a virtual machine?," [Online]. Available: https://www.vmware.com/topics/glossary/content/virtual-machine.html. [Accessed 21 February 2022].

[5] IBM Cloud Education, "IaaS (Infrastructure-as-a-Service)," 12 July 2019. [Online]. Available: https://www.ibm.com/cloud/learn/iaas. [Accessed 1 March 2022].

[6] IBM Cloud Education, "PaaS (Platform-as-a-Service)," 14 July 2021. [Online]. Available: https://www.ibm.com/cloud/learn/paas. [Accessed 1 March 2022].

[7] J. Nupponen and D. Taibi, "Serverless: What it Is, What to Do and What Not to Do," in *IEEE International Conference on Software Architecture Companion (ICSA-C)*, Salvador, 2020.

[8] Amazon Web Services Inc, "Amazon Elastic Container Service - Fully Managed Container Solution," [Online]. Available: https://aws.amazon.com/ecs/. [Accessed 11 February 2022].

[9] Amazon Web Services Inc, "AWS Lambda - Run code without thinking about servers or clusters," [Online]. Available: https://aws.amazon.com/lambda/. [Accessed 11 February 2022].

[10] The Internet Society, " Hypertext Transfer Protocol -- HTTP/1.1," June 1999. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2616. [Accessed 1 March 2022].

[11] The Internet Society, "HTTP Over TLS," May 2000. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2818. [Accessed 1 March 2022].

[12] Internet Engineering Task Force (IETF), "The WebSocket Protocol," December 2011. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6455. [Accessed 1 March 2022].

[13] Network Working Group, "Fiel Transfer Protocol (FTP)," October 1985. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc959. [Accessed 1 March 2022].

[14] S. Hawke, "REST," W3C, 16 December 2011. [Online]. Available: https://www.w3.org/2001/sw/wiki/REST. [Accessed 11 February 2022].

[15] The Internet Society, "Uniform Resource Identifier (URI): Generic Syntax," January 2005. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3986. [Accessed 21 February 2022].

[16] Network Working Group, "HTTP Method Definitions," June 1999. [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html. [Accessed 21 February 2022].

[17] Mozilla Inc, "MIME types (IANA media types)," 1 February 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types. [Accessed 11 February 2022].

[18] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris and D. Orchard, "Web Services Architecture," 11 February 2004. [Online]. Available: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/. [Accessed 11 December 2021].

[19] Django, "Django - The web framework for perfectionists with deadlines," [Online]. Available: https://www.djangoproject.com/. [Accessed 11 February 2022].

[20] Ruby on Rails Team, "Ruby on Rails - Compress the complexity of modern web apps," [Online]. Available: https://rubyonrails.org/. [Accessed 11 February 2022].

[21] Laravel LLC, "Laravel - The PHP Framework for Web Artisans," [Online]. Available: https://laravel.com/. [Accessed 11 February 2022].

[22] VMware Inc, "Spring - Spring makes Java simple," [Online]. Available: https://spring.io/. [Accessed 11 February 2022].

[23] OpenJS Foundation, "Express - Fast, unopinionated, minimalist web framework for Node.js," [Online]. Available: https://expressjs.com/. [Accessed 11 February 2022].

[24] Sideway Inc, "hapi - The Simple, Secure Framework Developers Trust," [Online]. Available: https://hapi.dev/. [Accessed 11 February 2022].

[25] Mozilla Inc, "HTTP Headers - Content-Type," 18 February 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Type. [Accessed 1 March 2022].

[26] Mozilla Inc, "HTTP Headers - Authorization," 4 February 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization. [Accessed 1 March 2022].

[27] OpenJS Foundation, "ExpressJS - Using middleware," [Online]. Available: https://expressjs.com/en/guide/using-middleware.html. [Accessed 1 March 2022].

[28] Internet Engineering Task Force (IETF), "The OAuth 2.0 Authorization Framework," October 2012. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6749. [Accessed 12 February 2022].

[29] IBM Cloud Education, "JVM vs. JRE vs. JDK: What's the Difference?," 30 June 2021. [Online]. Available: https://www.ibm.com/cloud/blog/jvm-vs-jre-vs-jdk. [Accessed 21 February 2022].

[30] VMware Inc, "What is a hypervisor?," [Online]. Available: https://www.vmware.com/topics/glossary/content/hypervisor.html. [Accessed 21 February 2022].

[31] Oracle Corporation, "Oracle VM VirtualBox," [Online]. Available: https://www.virtualbox.org/. [Accessed 21 February 2022].

[32] VMware Inc, "VMware Workstation Pro," [Online]. Available: https://www.vmware.com/products/workstation-pro.html. [Accessed 21 February 2022].

[33] IBM Cloud Education, "Virtualization: A complete guide," 10 June 2019. [Online]. Available: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide. [Accessed 11 February 2022].

[34] IBM Cloud Education, "Containers," 23 June 2021. [Online]. Available: https://www.ibm.com/cloud/learn/containers. [Accessed 21 February 2022].

[35] IBM Cloud Education, "Containerization," 23 June 2021. [Online]. Available: https://www.ibm.com/cloud/learn/containerization. [Accessed 21 February 2022].

[36] S. McCarty, "A Practical Introduction to Container Terminology," 22 February 2018. [Online]. Available: https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction#. [Accessed 28 February 2022].

[37] Docker Inc, "Docker - Empowering App Development for Developers," [Online]. Available: https://www.docker.com/. [Accessed 11 February 2022].

[38] Microsoft Inc, "Azure - Container Instances," [Online]. Available: https://azure.microsoft.com/en-us/services/container-instances/. [Accessed 11 February 2022].

[39] DigitalOcean LLC, "Droplets - DigitalOcean's Scalable Virtual Machines," [Online]. Available: https://www.digitalocean.com/products/droplets. [Accessed 11 February 2022].

[40] Amazon Web Services Inc, "Amazon Elastic Kubernetes Service - The most trusted way to start, run, and scale Kubernetes," [Online]. Available: https://aws.amazon.com/eks/. [Accessed 11 February 2022].

[41] Microsoft Inc, "Azure - Kubernetes Service," [Online]. Available: https://azure.microsoft.com/en-us/services/kubernetes-service/. [Accessed 11 February 2022].

[42] IBM Cloud Education, "Load Balancing," 10 June 2019. [Online]. Available: https://www.ibm.com/cloud/learn/load-balancing. [Accessed 1 March 2022].

[43] Microsoft Inc, "Azure Functions - Execute event-driven serverless code with an end-to-end development experience," [Online]. Available: https://azure.microsoft.com/en-us/services/functions/. [Accessed 11 February 2022].

[44] Google LLC, "Cloud Functions - Scalable pay-as-you-go functions as a service (FaaS) to run your code with zero server management," [Online]. Available: https://cloud.google.com/functions. [Accessed 11 February 2022].

[45] Cloudflare Inc, "What is BaaS? Backend-as-a-Service vs. serverless," [Online]. Available: https://www.cloudflare.com/en-gb/learning/serverless/glossary/backend-as-a-service-baas/. [Accessed 25 February 2022].

[46] Amazon Web Services Inc, "Amazon Cognito - Simple and Secure User Sign-Up, Sign-In, and Access Control," [Online]. Available: https://aws.amazon.com/cognito/. [Accessed 25 February 2022].

[47] Amazon Web Services Inc, "Amazon S3 - Object storage built to retrieve any amount of data from anywhere," [Online]. Available: https://aws.amazon.com/s3/. [Accessed 25 February 2022].

[48] Amazon Web Services Inc, "Amazon DynamoDB - Fast, flexible NoSQL database service for single-digit millisecond performance at any scale," [Online]. Available: https://aws.amazon.com/dynamodb/. [Accessed 25 February 2022].

[49] Amazon Web Services Inc, "Amazon Simple Queue Service - Fully managed message queues for microservices, distributed systems, and serverless applications," [Online]. Available: https://aws.amazon.com/sqs/. [Accessed 11 February 2022].

[50] Amazon Web Services Inc, "Amazon Simple Notification Service - Fully managed pub/sub messaging, SMS, email, and mobile push notifications," [Online]. Available: https://aws.amazon.com/sns/. [Accessed 12 Ferbruary 2022].

[51] G. Mao, "Understanding the Different Ways to Invoke Lambda Functions," 2 July 2019. [Online]. Available: https://aws.amazon.com/blogs/architecture/understanding-the-different-ways-to-invoke-lambda-functions/. [Accessed 1 March 2022].

[52] Amazon Web Services Inc, "Amazon API Gateway - Create, maintain, and secure APIs at any scale," [Online]. Available: https://aws.amazon.com/api-gateway/. [Accessed 11 February 2022].

[53] Amazon Web Services Inc, "Choosing between HTTP APIs and REST APIs," [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-vs-rest.html. [Accessed 11 February 2022].

[54] C.-F. Fan, A. Jindal and M. Gerndt, "Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, Online Streaming, 2020.

[55] Amazon Web Services Inc, "Amazon EC2 - Secure and resizable compute capacity for virtually any workload," [Online]. Available: https://aws.amazon.com/ec2/. [Accessed 25 February 2022].

[56] Amazon Web Services Inc, "AWS CloudFormation - Speed up cloud provisioning with infrastructure as code," [Online]. Available: https://aws.amazon.com/cloudformation/. [Accessed 2 March 2022].

[57] Amazon Web Services Inc, "Elastic Container Registry - Easily store, share, and deploy your container software anywhere," [Online]. Available: https://aws.amazon.com/ecr/. [Accessed 12 February 2022].

[58] Amazon Web Services Inc, "Elastic Load Balancing - Distribute network traffic to improve application scalability," [Online]. Available: https://aws.amazon.com/elasticloadbalancing/. [Accessed 11 February 2022].

[59] Vercel Inc, "Vercel - Develop. Preview. Ship. For the best frontend teams," [Online]. Available: https://vercel.com/. [Accessed 11 February 2022].

[60] Vercel Inc, "NextJS - The React Framework for Production," [Online]. Available: https://nextjs.org/. [Accessed 11 February 2022].

[61] Microsoft Inc, "TypeScript Documentation," [Online]. Available: https://www.typescriptlang.org/docs/. [Accessed 12 February 2022].

[62] D. Moscrop, "serverless-http," [Online]. Available: https://github.com/dougmoscrop/serverless-http. [Accessed 11 February 2022].

[63] Amazon Web Services Inc, "AWS Command Line Interface Documentation," [Online]. Available: https://docs.aws.amazon.com/cli/index.html. [Accessed 12 February 2022].

[64] Amazon Web Services Inc, "What is the AWS Serverless Application Model (AWS SAM)?," [Online]. Available: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-sam.html. [Accessed 12 February 2022].

[65] Open Collective, "curl - command line tool and library for transferring data with URLs," [Online]. Available: https://curl.se/. [Accessed 28 February 2022].

[66] A. Danial, "cloc - Count Lines of Code," [Online]. Available: https://github.com/AlDanial/cloc. [Accessed 28 February 2022].

[67] V. Yussupov, U. Breitenbücher, F. Leymann and C. Müller, "Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2019)*, ACM, 2019, pp. 273-283.

[68] Amazon Web Services Inc, "AWS Step Functions - Visual workflows for modern applications," [Online]. Available: https://aws.amazon.com/step-functions/?step-

functions.sort-by=item.additionalFields.postDateTime&step-functions.sort-order=desc. [Accessed 11 February 2022].

[69] Microsoft Inc, "Durable Functions Overview," 15 January 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp. [Accessed 11 February 2022].

[70] Amazon Web Services Inc, "AWS Lambda runtime API," [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/runtimes-api.html. [Accessed 11 February 2022].

[71] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad and A. Iosup, "Serverless Applications: Why, When, and How?," *IEEE Software,* vol. 38, no. 1, pp. 32-39, 2020.

[72] S. Pittet, "The different types of software testing," [Online]. Available: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing. [Accessed 2 March 2022].

[73] Grafana Labs, "k6 - Load testing for engineering teams," [Online]. Available: https://k6.io/. [Accessed 11 February 2022].

[74] T. Sallai, "Javascript on AWS Lambda: How to use Node.js in a serverless architecture," Advanced Web Machinery, 2020, pp. 78-91.

[75] UX Planet, "How page speed affects Web User Experience," 29 December 2019. [Online]. Available: https://uxplanet.org/how-page-speed-affects-web-user-experience-83b6d6b1d7d7. [Accessed 2 March 2022].

[76] VMware Inc, "What is Cloud Elasticity?," [Online]. Available: https://www.vmware.com/topics/glossary/content/cloud-elasticity.html. [Accessed 2 March 2022].

[77] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly and S. Pallickara, "Serverless Computing: An Investigation of Factors," in *IEEE International Conference on Cloud Engineering*, Orlando, 2018.

[78] S. Boucher, A. Kalia, D. G. Andersen and M. Kaminsky, "Putting the “Micro” Back in Microservice," in *USENIX Annual Technical Conference (USENIX ATC ’18)*, Boston, 2018.

[79] Oracle Corporation, "Overview of Java," [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/12.2/jjdev/Java-overview.html#GUID-17B81887-C338-4489-924D-FDDF2468DEA7. [Accessed 2 March 2022].

[80] Microsoft Inc, "A tour of the C# language," 30 November 2021. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/. [Accessed 2 March 2022].

[81] The YAML Project, "YAML Ain't Markup Language," [Online]. Available: https://yaml.org/. [Accessed 2 March 2022].

[82] Red Hat Inc, "What is Infrastructure as Code (IaC)?," 1 December 2020. [Online]. Available: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac. [Accessed 12 February 2022].

[83] C. Rippon, "Controlling Type Checking Strictness in TypeScript," 28 July 2020. [Online]. Available: https://www.carlrippon.com/controlling-type-checking-strictness-in-typescript/. [Accessed 12 February 2022].

[84] Microsoft Inc, "TypeScript Documentation - Generics," [Online]. Available: https://www.typescriptlang.org/docs/handbook/2/generics.html. [Accessed 2 March 2022].

[85] Mozilla Inc, "Promise," 18 February 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Accessed 25 February 2022].

[86] Sideway Inc, "joi - The most powerful schema description language and data validator for JavaScript," [Online]. Available: https://github.com/sideway/joi. [Accessed 25 February 2022].

[87] Docker Inc, "About storage drivers - Images and Layers," [Online]. Available: https://docs.docker.com/storage/storagedriver/#images-and-layers. [Accessed 25 February 2022].

[88] Amazon Web Services Inc, "Amazon ECS task networking," [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-networking.html. [Accessed 1 March 2022].

[89] Amazon Web Services Inc, "Amazon EC2 Instance Types," [Online]. Available: https://aws.amazon.com/ec2/instance-types/. [Accessed 1 March 2022].

[90] SSH.COM, "SSH Protocol – Secure Remote Login and File Transfer," [Online]. Available: https://www.ssh.com/academy/ssh/protocol. [Accessed 2 March 2022].

[91] Amazon Web Services Inc, "Amazon Machine Images (AMI)," [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html. [Accessed 1 March 2022].

[92] Amazon Web Services Inc, "Amazon Linux 2," [Online]. Available: https://aws.amazon.com/amazon-linux-2/?amazon-linux-whats-new.sort-by=item.additionalFields.postDateTime&amazon-linux-whats-new.sort-order=desc. [Accessed 1 March 2022].

[93] Amazon Web Services Inc, "Elastic Load Balancing pricing," [Online]. Available: https://aws.amazon.com/elasticloadbalancing/pricing/. [Accessed 25 February 2022].

[94] Amazon Web Services Inc, "Load balancer types," [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/load-balancer-types.html. [Accessed 12 February 2022].

[95] Amazon Web Services Inc, "How Elastic Load Balancing works," [Online]. Available: https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/how-elastic-load-balancing-works.html. [Accessed 25 February 2022].

[96] Amazon Web Services Inc, "AWS Fargate - Serverless compute for containers," [Online]. Available: https://aws.amazon.com/fargate/. [Accessed 11 February 2022].

[97] I. Buchanan, "What is containers as a service?," [Online]. Available: https://www.atlassian.com/continuous-delivery/microservices/containers-as-a-service. [Accessed 25 February 2022].

[98] Amazon Web Services Inc, "EC2 or AWS Fargate?," [Online]. Available: https://containersonaws.com/introduction/ec2-or-aws-fargate/. [Accessed 12 February 2022].

[99] N. Pantic, "Serverless Showdown: Fargate vs. Lambda," 3 May 2021. [Online]. Available: https://www.serverlessguru.com/blog/serverless-showdown-fargate-vs-lambda. [Accessed 25 February 2022].

[100] Amazon Web Services Inc, "Working with AWS Lambda proxy integrations for HTTP APIs," [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-develop-integrations-lambda.html. [Accessed 25 February 2022].

[101] OpenJS Foundation, "ExpressJS API," [Online]. Available: https://expressjs.com/en/api.html. [Accessed 25 February 2022].

[102] Vendia Org, "Serverless Express," [Online]. Available: https://github.com/vendia/serverless-express. [Accessed 11 February 2022].

[103] OpenJS Foundation, "koa - next generation framework for node.js," [Online]. Available: https://koajs.com/. [Accessed 11 February 2022].

[104] OpenJS Foundation, "Fastify - Fast and low overhead web framework, for Node.js," [Online]. Available: https://www.fastify.io/. [Accessed 11 February 2022].

[105] J. Beswick, "Operating Lambda: Performance optimization – Part 1," 26 April 2021. [Online]. Available: https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/. [Accessed 2 March 2022].

[106] Amazon Web Services Inc, "Creating Lambda container images," [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/images-create.html. [Accessed 12 February 2022].

[107] Amazon Web Services Inc, "Creating and sharing Lambda layers," [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html. [Accessed 12 February 2022].

[108] Amazon Web Services Inc, "Lambda quotas," [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. [Accessed 3 March 2022].

[109] Mozilla Inc, "TypeError: cyclic object value," 20 July 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors/Cyclic_object_value. [Accessed 12 February 2022].

[110] A. Giammarchi, "CircularJSON," [Online]. Available: https://github.com/WebReflection/circular-json. [Accessed 3 March 2022].

[111] Amazon Web Services Inc, "AWS SDK for JavaScript v3," [Online]. Available: https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/index.html. [Accessed 3 March 2022].

[112] Amazon Web Services Inc, "DynamoDB Low-Level API," [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors. [Accessed 3 March 2022].

[113] Amazon Web Services Inc, "AWS::DynamoDB::Table," [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-dynamodb-table.html. [Accessed 12 February 2022].

[114] Amazon Web Services Inc, "AWS::ApiGatewayV2::Api," [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-apigatewayv2-api.html. [Accessed 3 March 2022].

[115] Mozilla Inc, "SPA (Single-page application)," 8 October 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/SPA. [Accessed 28 February 2022].

[116] Microsoft Inc, "Choose Between Traditional Web Apps and Single Page Apps (SPAs)," 16 December 2021. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps. [Accessed 28 February 2022].

[117] Mozilla Inc, "Introduction to the DOM," 14 September 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. [Accessed 11 February 2022].

[18] Mozilla Inc, "Manipulating documents," 2 February 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents. [Accessed 28 February 2022].

[19] Meta Platforms Inc, "React - A JavaScript library for building user interfaces," [Online]. Available: https://reactjs.org/. [Accessed 11 February 2022].

[20] Prosus N.V., "Stackoverflow Insights - Web Frameworks," 2020. [Online]. Available: https://insights.stackoverflow.com/survey/2020#technology-web-frameworks. [Accessed 11 February 2022].

[21] S. Greif, "State of JS - Front-end Frameworks," 2020. [Online]. Available: https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/. [Accessed 11 February 2022].

[22] Meta Platforms Inc, "Introducing JSX," [Online]. Available: https://reactjs.org/docs/introducing-jsx.html. [Accessed 11 February 2022].

[23] Amazon Web Services Inc, "AWS Free Tier - Free Cloud Computing Services," [Online]. Available: https://aws.amazon.com/free/. [Accessed 11 February 2022].

[24] Amazon Web Services Inc, "Amazon Linux 2," [Online]. Available: https://aws.amazon.com/amazon-linux-2/?amazon-linux-whats-new.sort-by=item.additionalFields.postDateTime&amazon-linux-whats-new.sort-order=desc. [Accessed 28 February 2022].

[25] Amazon Web Services Inc, "AWS Pricing Calculator - Estimate the cost for your architecture solution," [Online]. Available: https://calculator.aws/#/. [Accessed 3 March 2022].

[26] Amazon Web Services Inc, "Amazon EC2 Cost and Capacity Optimization," [Online]. Available: https://aws.amazon.com/ec2/cost-and-capacity/. [Accessed 11 February 2022].

[27] D. Poccia, "AWS News Blog: Provisioned Concurrency for Lambda Functions," 3 12 2019. [Online]. Available: https://aws.amazon.com/blogs/aws/new-provisioned-concurrency-for-lambda-functions/. [Accessed 31 12 2021].

[28] Amazon Web Services Inc, "Amazon Relational Database Service (RDS)," [Online]. Available: https://aws.amazon.com/rds/. [Accessed 3 March 2022].

[29] J. Beck, T. Le Moullec, K. Polossat and S. Sanders, "Amazon Web Services Blogs," 15 February 2021. [Online]. Available: https://aws.amazon.com/blogs/containers/theoretical-cost-optimization-by-amazon-ecs-launch-type-fargate-vs-ec2/. [Accessed 11 December 2021].

[130] Amazon Web Services Inc, "Amazon EC2 Reserved Instances," [Online]. Available: https://aws.amazon.com/ec2/pricing/reserved-instances/. [Accessed 11 February 2022].

[131] Amazon Web Services Inc, "Amazon EC2 Spot Instances," [Online]. Available: https://aws.amazon.com/ec2/spot/. [Accessed 11 February 2022].