Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor Thesis

Practical Aspects of FaaS Applications' Migration

Christian Müller

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h.c. Frank Leymann
Supervisor:	Vladimir Yussupov, M.Sc.
Commenced:	February 8, 2019
Completed:	August 8, 2019

Abstract

With the huge variety of available FaaS platforms in cloud and self-hosted environments the idea of migrating function applications from one provider to another is becoming a important consideration. This work investigates the challenges developers encounter when manually migrating applications between Amazon Web Services, Microsoft Azure and IBM Cloud regarding the efforts needed to migrate the functions and the services. This work also proposes a simple approach to reduce the coupling between the function application and the cloud provider by externalizing the business logic into a serparate, completely vendor independant, package. We see that this approach reduces the efforts needed to migrate the source code to another provider but it does not reduce the effort of migrating the functions configuration and services. We see that the efforts for migration are not only affected by the migration of the source code but also by the migration of the services, especially in self-hosted environments. There developers also have to find a proper substitution of the service for their use-case.

Zusammenfassung

Bei der Vielzahl der verfügbaren FaaS-Plattformen in Cloud- und selbst gehosteten Umgebungen wird die Idee der Migration von Funktionsanwendungen von einem Anbieter zum anderen immer wichtiger. Diese Arbeit untersucht die Herausforderungen, denen Entwickler bei der manuellen Migration von Anwendungen zwischen Amazon Web Services, Microsoft Azure und IBM Cloud hinsichtlich des Aufwands für die Migration der Funktionen und Dienste begegnen. Diese Arbeit schlägt auch einen einfachen Ansatz vor, um die Kopplung zwischen der Funktionsanwendung und dem Cloud-Provider zu reduzieren, indem die Geschäftslogik in ein separates, völlig herstellerunabhängiges Paket ausgelagert wird. Wir sehen, dass dieser Ansatz den Aufwand für die Migration des Quellcodes zu einem anderen Anbieter reduziert, aber nicht den Aufwand für die Migration der Funktionskonfiguration und der Dienste. Wir sehen, dass die Bemühungen um die Migration nicht nur von der Quellcode-Migration, sondern auch von der Migration der Dienste, insbesondere in selbst gehosteten Umgebungen, beeinflusst werden. Dort müssen Entwickler auch einen geeigneten Ersatz für den Dienst in ihren Anwendungsfall finden.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

In the past applications have been developed in a monolithic style, meaning that the whole application is one huge piece of tightly coupled source code. Usualy these applications also ship as one rather large binary. Because these applications can be a nightmare to maintain due to the size of the application, people have started to decompose their applications into smaller services that get composed together to form the complete functionality of the application. These services are often called microservices.

Microservices are independent from the rest of the application, meaning they only interact with other services using a set of specified interfaces, this could for example be done by using an API. Building an application based on microservices has many benefits especially regarding the maintainablity of the application. Since a service is rather small the efforts needed to remimplent the service are also comparably low. The rest of the application is unaffected if the interfaces get implemented as specified. Another huge benefit is the posibility of scaling ouf largely used services independently from other services. When considering a scale out of a monolithic application we always have to launch a new instance of the whole application. This obviously takes up more resources compared to just scaling out a single microservice.

The trend towards mircoservices has also been accellerated with the rise of cloud computing in recent times. With cloud computing the developer does not have to worry about buying new servers on which the applications will run because the expansion of the infrastructure that gets used in the cloud can be set dynamically the pool of servers can be expanded if the application experiences high load once the load is reduced again the cloud resources can also get reduced again.
Of course cloud computing does not end with infrastructure[1]. Many cloud providers also have offerings providing a fully managed platform to run the applications[2], here the infrastructure is completely managed by the cloud provider.

But running microservices on a cloud platform does not seem to be the end of the story. With the announcement of AWS Lambda in 2014 another new approach to develop applications has been presented. Lambda allows the deployment of small, short lived and stateless programms called functions that do exactly one task. Because the cloud provider will schedule the function on shared infrastructure you will only be charged once the function gets called. There is no need to pay for idle times. Scaling also does not have to be considered because the cloud provider will handle the execution of the function. In case of high loads the provider will automatically scale out to fulfil the needs.
In theory a microservice can be decomposed further into a set of functions making it possible to build microservices or even entire applications in a serverless environment [Fro12].

[1]Usualy refered to as Infrastructure as a Service or short IaaS
[2]Usually refered to as Platform as a Service or short PaaS

Because FaaS offerings highly rely on the cloud provider these offerings have a large tendency towards vendor lock-in making the migration to another cloud provider more complex, compared to other cloud deployment models. Because the function and the platforms it runs on have a quite tight coupling moving to another platform might make modifications to the source code of the function necessary. When moving to another IaaS of PaaS provider such modifications are most likely not needed. The only thing that has to get modified there is the configuration of the application [Cla17].

To investigate the complexity of such a migration process from one cloud provider to another one four use-cases will be presented. These use some of the cloud providers other services like databases or message queues. In the next step a initial implementation on Amazon Web Sevices is described. This initial implementation will be used to investigate the efforts needed to migrate the four use-cases over to Microsoft Azure and IBM Cloud. To complete the package we will take a look at the additional problems a developer might encounter when migrating to a self hosted environment.

2 Fundamentals

Due to the widespread definitons of the terms Serverless and FaaS we want to clarify what is understood under these terms in the following chapters.

2.1 Evolution of cloud deployment models

The way applications get deployed and managed has drastically changed in the last decade. Initially everyone running a client-server based application had to maintain their own servers in a datacenter. Expanding the capacities was only possible by purchasing more servers. In many cases it was also necessary to maintain the networking infrastruture in order to ensure that everything works. This approach was rather unflexible because adding servers to the system consumes a significant amount of time [VTT+18].

Nowadays, many companies are moving away from traditional infrastructures due to it being less flexible in comparison with Infrastructure-as-a-Service (IaaS) cloud service model. With IaaS it is no longer necessary to purchase servers because the management of physical hardware is done by cloud providers, which focus on providing virtualized infrastructure to tenants.
Virtualization is done using a hypervisor software installed on the physical machines, also called hosts. Hardware virtualization is usually used because this technique allows very good isolation between virtual machines running on the host. With hardware virtualization the hypervisor virtualizes the hardware of the host meaning the virtual machines running on the host do not depend on any software component of the host. As a result the virtual machine will require the full installation of an operating system.
With IaaS the tenant requests certain resources (servers, IP addresses, etc.) from the cloud provider. On these requested servers the tenant still has to maintain the operating system and the software running on the machines. The networking might have to be configured depending on the needs of the tenant. The great thing about IaaS is that it takes only small amounts of time to provision a new machine. The IaaS resources are usualy billed in a per hour basis allowing tenants to dynamically request more resources based on the expected workloads. If they are not needed anymore the tenants

	Traditional IT	IaaS	PaaS	FaaS
Time to Provision	Days or more	Hours to Minutes	Minutes to Seconds	Milliseconds
Billing granularity	Not applicable	Hours	Hours or Minutes	Milliseconds

Table 2.1: Comparison of the traditonal approach and the three major cloud deployment models in terms of provisioning time and billing granularity [FIMS17].

can just remove them and they do not have to pay for redundant resources. However this scaling process is not great for a unexpected increase of workload because the deployment of new instances is too slow [VTT+18].

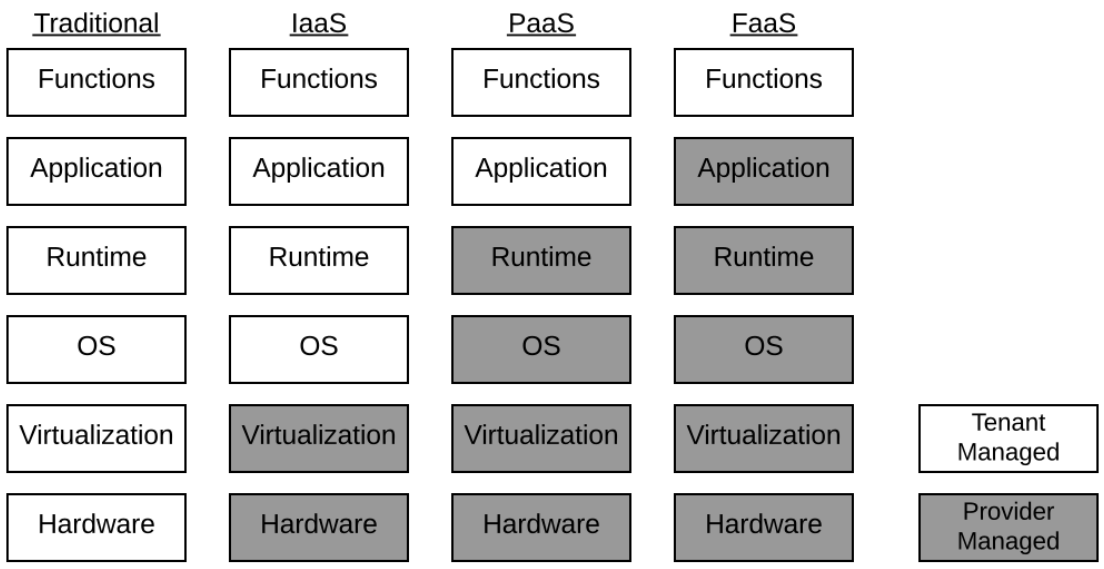


Figure 2.1: Visual comparison of the four (cloud) cloud deployment models [McK16].

With Platform as a Service (PaaS) the tenant does not have to worry about networking tasks regarding the maintenance of the operating systems running on the machines they have deployed. On PaaS the tentant can deploy an application via a container (see Section 2.2) or by uploading a binary into the cloud. PaaS often supports automated scaling based on certain metrics like processor load or requests per second. While scaling is usualy faster than IaaS it can still be measured in minutes, sometimes in seconds. In comparison with IaaS, more responsibilites are handed over to the cloud provider. Biling is usualy also handled in a per hour per instance basis just like IaaS [VTT+18].

With these three approaches you will have to keep at least one instance running at all times. Even if no one is using the application. This means you will have to pay for at least one instance at all times. This is not the case with Functions as a Service (FaaS). It is the newest cloud service in which even more responsiblities are handed over to the cloud provider. FaaS also does not allow the direct deployment of regular applications. In a FaaS environment one can only deploy arbitrary, event-driven code snippets called functions. Billing is very fine grained since developers only have to pay the time the functions have been executed in the cloud. Because of the way functions are designed they can be scaled down to zero instances [Rob18].

As we can see in Table 2.1, the cloud deployment models become more fine grained in terms of billing and scalability. Figure 3.1 also shows that the cloud provider takes more responsibilites depending on the cloud deployment model. This can be a problem because the more responsibilites the cloud provider will handle the higher the risk of introducing a tight-coupling with a particular cloud provider [CNC18].

Of course running functions or applications alone is not really useful because applications typically need to persist state, e.g., to a database. In a traditonal or an IaaS environment this will most likely be done by the tenant. But on PaaS or FaaS envrionments, a managed database service that is either charged on a per hour or a per query basis. This depends on the database and the cloud provider.

2.2 Container virtualization

There are two types of virtualization used widely. Hardware virtualization and container virtualization. In a hardware virtualization environment the hypervisor virtualizes computer hardware to run multiple virtual machines on one host computer. Because of this a complete operating system must be installed in the virtual machine. As a result a host system runs multiple operating system kernels for seperate virtual machines. This can be considered redundant [Man14]. It often also takes minutes to hours to create a new virtual machine [Cha18]
Assuming all virtual machines use the same operating system and even the same kernel another type of virtualization can be considered: operating system virtualization. Instead of virtualizing a whole computer the operating systems kernel is virtualized to produce more lightweight virtual 'machines'. One child category of operating system virtualization is container virtualization. A container is a virtual 'machine' virtualized by the container engine which is the container virtualizations equivalent to a hypervisor. Containers are created from a prebuilt snapshot called a container image that contains all dependencies needed to run one application. This image is packaged during the build process and it ensures the application can be executed anywhere assuming the the type of operating system is identical. This makes containers interesing for PaaS and FaaS environments. In recent times many offerings from cloud providers support the deployment of applications packaged in a container image [Cha18]. A widely used container engine is Docker[1] apart form beeing a container runtime Docker also provides tools to build and ship container images.

Nowadays containers are used widely across the industry because of the wide popularity there was an increasing need to provide a production grade solution to deploy and orchestrate containers at large scale. In this area Kubernetes[2] has become the most popular implementation to perform this. The main tasks of a container orchestrator include 1) deploying containers 2) scaling containers in and out depending on certan conditions 3) exposing endpoints of containers intended for external access 4) managing the configurations of containers [Eld18].

Many FaaS platforms use containers or virtual machines in the background to deploy the environments that are used to execute the functions. But most FaaS platforms do not support the deployment of functions in a container image. This is especially the case with the public cloud offerings like AWS Lambda [WLZ+18]. Many self hosted options do allow the deployment of functions from a container image though.

[1] https://www.docker.com/
[2] https://kubernetes.io/

2.3 FaaS Platform

With the function to execute the platform, sometimes also referred to as FaaS runtime, is the most important component in the Function as a Service environemnt. Looked at it from the outside it has a lot of responsibilities like:

- Retrieving the function from storage
- Provisioning an instance for scheduling a function
- Selecting an instance for function execution
- Shutting down instances that have exceeded a timeout

Of course the implementation as well as the responsibilities differ for every platform but every FaaS environment has such a runtime. It is important that the FaaS runtime is not confused with the runtime used to write the functions in like the JVM or NodeJS.

2.4 Cold and Warm start

A so called cold start occurs when a function gets invoked for the first time or after a longer time period. During the cold start the FaaS runtime will fetch the binary from storage and it will provision a new instance for execution. After successful provisioning the fresh instance is used to execute the function call. After the execution is finished the instance will wait for further function calls. If a function call occurs again, within a time frame defined by the cloud provider, the instance will be reused for execution resulting in lower processing times. This type of invokation is called a warm start. The time difference between cold and warm start is called cold start overhead. If the function is not called within the time frame the FaaS runtime will terminate the instance [Erw18].

There are also scenarios in which cold starts occur even if there are instances running already. For example if all instances are currently busy processing function requests. Then the serverless runtimes scheduler might decide to launch another instance instead.

2.5 Function Orchestrator

There are use-cases in which one function is not sufficent. For example due to the timeout limitations. In such cases function orchestration might be a solution. Generally speaking, function orchestration allows the implementation of workflows in functions. These workflows can include the following features:

- **Chaining** (Composition): This is the simplest concept in function orchestration. The basic idea is to execute one function after the other. Also considering inputs and outputs this means that the output of the first function will be used as the input for the second one. Mathematically speaking with the first function beeing function A and the second one beeing function B and the initial input x this is notated as $B(A(x))$ or $(B \circ A)(x)$.

- **Branching**: While chaining is the core concept behind function orchestration you might also only want to execute certain functions, if the output of the last function matches a certain condition. For example if you are writing an order processing workflow you might only want to execute a function if the order comes from a certain country. This feature also includes error handling because error handling can be considered a subtype of the branching feature.
- **Parallel Execution**: Some workflows contain sections that can be executed independently from each other. These sections can be executed in parallel from a common starting point. Depending on the application, the workflow might wait for the parallel child workflows to finish the execution. After all tasks have finished further functions can be executed.

Function orchestrators can be categorized into two general categories. The first one are function orchestrators based on external scheduling solutions like an external workflow engine. In the second category fit all orchestrators based on functions. This means that the function workflow in itself is considered a function [BCF+17].

3 Presenting the Use-Cases

Before getting into the challenges we have encountered while trying to migrate to other cloud provider we will first take a look at the examples (use-cases) that have been used for migration.

3.1 Thumbnail Generation

In this example application, we refer to one of the classic use cases described in various sources [Ser18a] - generation of a thumbnail. Here, the FaaS-hosted function will resize an image, i.e., create a thumbnail, whenever an image, e.g., a `.png` file, is stored in the object storage.

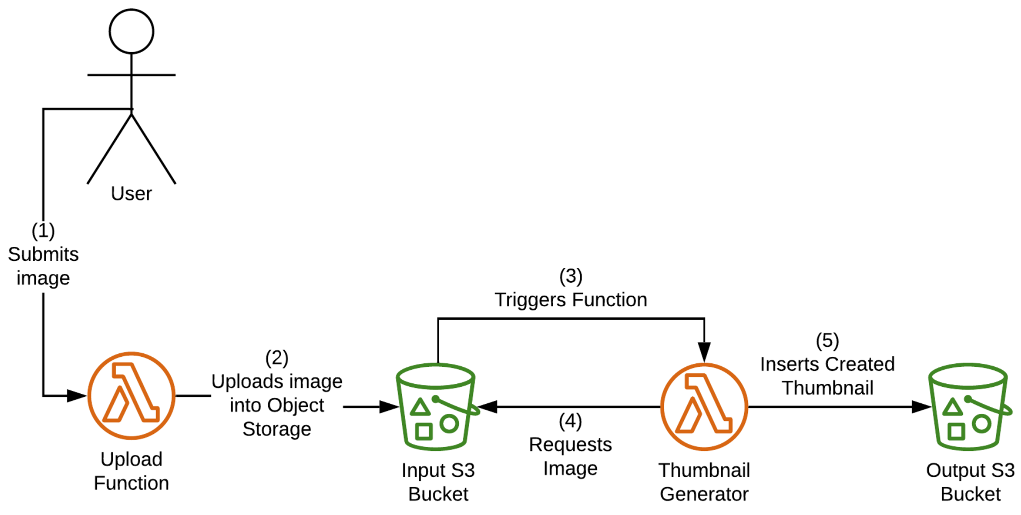


Figure 3.1: The Components (Services) used for the Thumbnail Generator on AWS

The function gets triggered by the object storage (Insertion Trigger). Since, as a baseline we use AWS, we assume that the object storage does support this feature. As shown in Figure 3.1, the user fist uploads an image into the object storage using the `upload` function. As a result the object stroage triggers a function which downloads the stored image and creates the thumbnail from it. The created thumbnail is uploaded into a separate object storage bucket to prevent an infinite loop.

To implement this application, we use Java as a programming language, more specifically, Java 8.

3.2 A simple serverless based API

he possibility to build an entirely serverless application is one of the reasons why FaaS become so popular. One obvious example discussed in multiple sources [BCC+17] is the implementation of a serverless API. In this example, we implement a serverless REST API for a ToDo application the use-case represents a simple HTTP based JSON API for a ToDo application.

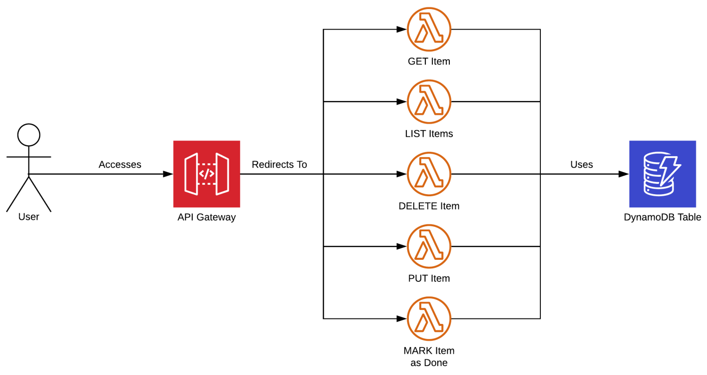


Figure 3.2: The Components (Services) used for the ToDo API on AWS

The ToDo API consists of the following Functions:

- **Put** Create a new ToDo item. This function is mapped to the `/put` path. To create a new entry this must be called through a HTTP POST request.
- **Get** Get one ToDo item based on the ID. This function is mapped to the `/get` path and must be called using a HTTP GET request. The ID is handed over using the `id` query parameter.
- **List** Lists all ToDo items. The user gets access to this function by sending a HTTP GET request to the `/lst` path.
- **Delete** Deletes a ToDo item. In order to access this function the user must send a HTTP POST request to the `/del` path. Since this function also needs an iD parameter to determine what entry should be deleted the ID must be added to the path as a query parameter, just like the **Get** function.
- **Mark as Done** Marks a ToDo item as done. Just like the **Get** and **Delete** functions this also needs the id as a query parameter. The function is called by accessing `/done` using a HTTP POST request.

Since functions are stateless, we need some sort of data store in order to store the ToDo items. For this we use a cloud provider specific datastore mosty targeted to the serverless world like AWS DynamoDB[1] or the Table Storage from the Azure storage account[2]. The initial implementation of this use-case was done in Go.

[1] https://aws.amazon.com/de/dynamodb/

[2] https://docs.microsoft.com/en-us/azure/storage/common/storage-account-overview

3.3 Composed Matrix Multiplication

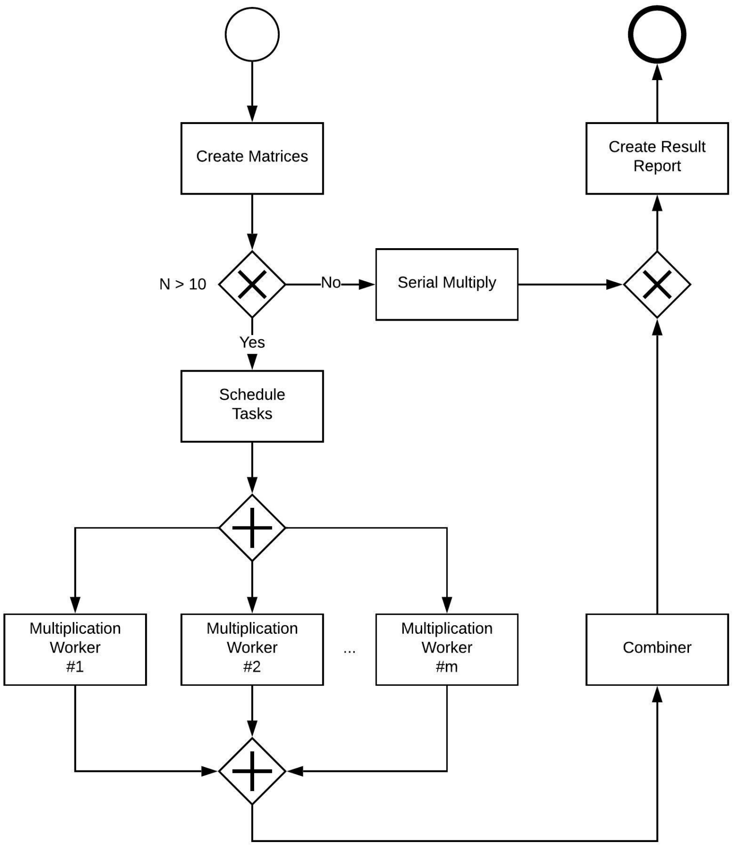


Figure 3.3: BPMN workflow of the Matrix Multiplication composition. *m* represents the worker count.

The main goal of this use case is to demonstrate as many details about function orchestration services as possible. For this we implemente a workflow for matrix multiplication using multiple functions and a workflow utilizing the core concepts behind function orchestration shown in Section 2.5.

The implementation of the use case is completely done using C# and .NET Framework Core (version 2.1).

As seen in Figure 3.3 we generate two $n \times n$ matrices A and B. The size of the matrix (n) is provided to the system by the user. The matrices are generated in the function composition to prevent any issues with parameter size. Because even two quite small matrices (encoded in JSON) could exceed the maximum input of approximately 65.000 characters on the AWS Lambda platform. For simplicity sake we decided to go with this route for every other implementation as well. Due to the limit, we need some kind of cache to store the data of the calculation.

Next, the orchestrator decides how the two matrices should be multiplied. Either using a serial multiplication function or in parallel using 5 workers. The amount of workers can be set in code while it is not possible to modify it on the fly.

The serial multiplication is used in case the order of the matrix is less than ten. In this case the calculation is done in one function that will calculate the result ans will store it in the cache as the result.
If the matrices order is greater or equal to ten the orchestrator will go the parallel route. Here we first have to split the task of the calculation into many sub tasks that will then be executed using the workers. Afterwards, a collection function will collect the results of the calculations done by the workers to build the result matrix.

After the result matrix is generated, a report is created, and the time needed to perfrom the calculation is shown. The report generating function will also handle the cleanup in the cache storage when necessary.

3.4 Event Processing

The main purpose of this use case is to take a look at the complexity of the migration of funtcions that use messaging in the means of a point to point channel and a publish / subscribe channel [HW12].

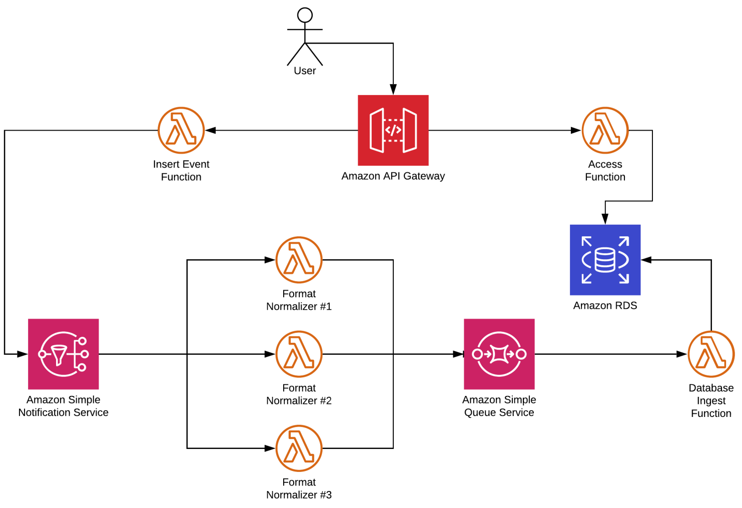


Figure 3.4: The Components (Services) used for the Event Processing use-case on AWS

The following we will describe how these services interact with the functions and what is the purpose of every involved function:

- The user or another application inserts an event using a function via HTTP.
- The *Insert Event Function*, also referred to as `ingest` function is responsible for reading the `type` attribute of the event and publishing it to the pub/sub topic, which represents this event type. There are three different types of events and therefore three topics. The event types are: `temperature` intended to announce the current temperature ate a position, `forecast` beeing intended to perform a forecast for a specific position and `state change` announcing the state of change of a device. Depending on the event the JSON structure differs and in order to

get them into one format `formatter` functions are used as seen in the next step. The main purpose of the insert event function is to route the events to different conversion functions based on the type attribute.

- For every type there will be a conversion function that will normalize the event into a single, simple format. Once these convertion functions have converted an event they will put the converted version into a queue for further processing.
  These functions are called `format` functions
- The last function in the processing chain will store the event in a relational database. The data stored can be viewed by two functions that will expose the contents of the database as a HTTP endpoint. They allow the retrieval of all events inserted as well as the one inserted last.

The use-case is targetted to be implemented in JavaScript. Apart from the functions we have used a relational database to store the processed events, publish / subscribe channels to send raw events from the routing function to the appropriate formating function and a point to point channel to send the formatted event to the database insertion function.

4 Implementation on Amazon Web Services

As a baseline for implementing the use cases described in Chapter 3, we use Amazon Web Services (AWS). We have chosen to use AWS as a reference point because their large marketshare of roughly 75% is very large. However other platforms also see increasing popularities [Hec17]. This might make it interesting to migrate an existing serverless application away from AWS. The following chapter discusses the problems and limitations encountered during implementation. This chapter also covers some technical details on how functions get invoked as well as potential issues when migrating to another platform.
To find the implementations for AWS and all other platforms please take a look at Appendix B.1.

The set of supported langugaes of AWS Lambda is quite large and covers most of the needs. Lambda supports Go, C# (.NET Core), Java, JavaScript (NodeJS), Python, PowerShell and Ruby out of the box [Ser19b]. The set of supported languages can also be expanded by using seperate tools like Apex[1] that adds more languages to the pool of supported languages on Lambda like Rust. The programming languages that have been used to implement the use cases on AWS are summarized in Table 4.1. These programming languages have been chosen because the programming languages or the underlying runtime were fairly common across the industry[2] (Java, C# and JavaScript) and Go was chosen because of the populatiy within the cloud community.

Use Case	Programming Language
Thumbnail Generator	Java
ToDo API	Go(lang)
Matrix Multiplication	C#
Event Processing	JavaScript

Table 4.1: The list of programming languages used for implementation on AWS

Finding a solution for Deploying functions Using Amazons Web UI to deploy more complex serverless applications can be very tedious. This is the main reason why we initialy investigated options to make the deployment more automatic. A great alternative to the Web UI is CloudFormation[3] it uses a domain specific language that is either based on YAML or JSON to define resources on AWS. However, the DSL provided by CloudFormation is very verbose and contains a lot of boilerplate code.

[1] https://apex.run/
[2] https://www.tiobe.com/tiobe-index/
[3] https://aws.amazon.com/cloudformation/

To make the description of serverless cloud resources simpler, we use the Serverless framework[4]. This makes the developing of functions way easier compared to the Web UI or CloudFormation itself. One of the main reasons is the very simplistic CLI that is used to deploy, delete and inspect serverless applications.
In the serverless framework a manifest describing the resources and functions used is written in YAML. Once the function should be deployed the serverless CLI will automatically convert this YAML manifest into a CloudFormation template which gets deployed in the background.
While it seems like the Serverless framework will reduce vendor lock in most of the features used are AWS specific which implies no reduction of vendor lock in [Fra19b]

4.1 Use-Case implementations

This section covers the services used for implementing the use-cases on AWS. Mentioning the chosen implementation approaches as well as the services used for the implmentation of the use-cases.

Thumbnail Generator The **Thumbnail Generation** use-case uses Amazons S3 object storage for storing the input data as well as the output data. For this two buckets are used one is considered the input bucket and the other one is the output bucket storing the generated thumbnails. In order to launch the function every time a new image is inserted into the input bucket. A trigger for the object created event is registered on the input bucket.

ToDo API The **ToDo API** use-case uses Dynamo DB for storing the datasets. In order to make the interaction with Go and DynamoDB easier a mapping library (similar to an ORM) was used to interact with DynamoDB[5].

As shown in Figure 4.1 all of the business logics code was placed into a 'core' library containing the non vendor specific code of all the functions. The only portion that still remains vendor specific is the function invokation and conversion of the input payload into a format that the 'core' library understands. For data interaction a generic interface is defined in the 'core' library that will have to be implemented in the vendor specific portion of the code. The main idea was the prevention of vendor specific code in the business logic which should theoretically help with the migratability of the application.

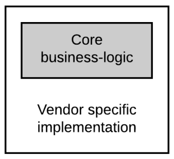


Figure 4.1: Visual representation of the idea behind the `core` library

[4]https://serverless.com/
[5]https://github.com/guregu/dynamo

Matrix Multiplication A core package was also created for the **Matrix-Multiplication** use-case. Due to the implementation complexity of this use-case this was a very good choice. Also because AWS Step Functions only supports a maximum input size of approximately 32.000 characters[6] and the fact that a JSON encoded matrix could easily exceed this limit for larger matrix sizes. A caching mechanism for storing the calculation was needed. For this the S3 object storage was used again. The Step Function object passed through just contains some configuration data as well as a unique identifier to find the data in the object storage. When implementing the *core* package this was taken into account. Therefore the core package uses an interface to retrieve the data using a unique identifier. This decision was made because even if another platform did support larger input payloads the implementation of the interface could be replaced with a very simple in memory solution to ensure no chaching mechanism was used.

Event-Processing While the **Event-Processing** use-case uses the largest number of external services used, the actual functions are very simple. In terms of services we use SNS for publish subscribe messaging, SQS for message queueing and Amazon Aurora (RDS) with MySQL interface as the relational database for storing the processed events.

4.2 Technical details

The configuration on AWS is just as important as the implementation because the triggers of a function are defined through external configurations. This can either be done using the Web UI, through CloudFormation or the Serverless framework as already mentioned in Chapter 4.
When the Lambda runtime triggers a function it passes an event object to the function. This object has a different structure depending on the type of trigger that invoked the function. This event gets unmarshalled either directly by the developer or in the background. This depends on the programming language used and the developers preference. However it is theretically possible to deploy a function with the wrong trigger. For example, a function might be intended to consume an object storage event but the configuration is actually set up for a HTTP trigger [Ser18b].

In Lambda a function is defined by setting its runtime basically telling the platform what programming language is used. To tell the platform what exactly should be executed the handler path has to be set. This path differs for every runtime. For example with Java and C# you define the package path to the class that implemnts the function, with Go you have to put the name of the executable binary here and in JavaScript the path to the function is defined in case of the event processing use case shown in Listing 4.1 the path shown there is built up by $Filename.FunctionName$ citeserverless-aws-functions.

Listing 4.1 also shows how a HTTP Trigger and a SNS (publish / subscribe messaging) trigger is defined. The name of the topic for the SNS trigger gets assigined to the value of a different configurations field in the serverless.yml file. The `${self:path}` notation is used for

[6] https://docs.aws.amazon.com/step-functions/latest/dg/limits.html

Listing 4.1 Partial and simplified trigger definition of the Serverless configuration for the event-processing use-case

```
functions:
  ingest:
    handler: ingest.handleIngest
    events:
      - http:
          path: ingest
          method: POST
  format_temperature:
    handler: formatprocessing.formatTemperatureEvent
    events:
      - sns: ${self:custom.temperature_format_topic}

```

this. The definition of the HTTP trigger should be quite obvious with some basic understanding of the HTTP protocol. By default the URL for a HTTP triggered function looks like this: `https://<random string>.execute-api.<region>.amazonaws.com/<stage>/<function path>` Often, the possible values for stage include dev for development and prod for production stages.

Other parameters for the deployment can either be defined globally or for each function individually. The other parameters include the functions runtime, the memory assigned to the function and the region to deploy the functions on. When deploying on AWS it is also important to assign the functions permissions to perform certain operations like reading the object storage or pushing to a message queue. These so called IAM[7] roles are also set globally in the configuration file for the serverless framework as shown in Listing A.2. To keep the handling of permissions simple a wildcard grant is always assigned for every service. When developing in a production environemnt the roles should be fine grained to only allow the operations that the functions really need to keep the system more secure [Fra18b].

For generating base projects we relied on the templating and boilerplate generation capabilities of the serverless framework. All of the base projects have been created using the CLI of the serverless framework by running `serverless create -t <template> -n <application name>`. Depending on the programming language a differen template was chosen. For example, the Matix Mutliplication use-case was created using the *aws-csharp* template [Fra18a].

Most of the services used in our use-cases can be easily configured using the serverless framework which will also handle the creation of topics, queues or buckets. However databases like RDS and DynamoDB have requried further configuration. By default a RDS instance can only be accessed using a virtual private cloud (VPC) which acts as a private network in the cloud. Only services within the same VPC can access each other. This means that we have to make ensure the functions accessing the RDS gain access to the VPC [Zha18].

[7]Short for: Identity and Access Management

Listing 4.2 Function definition for the Thumbnail Generation function[9]

```java
public class ThumbnailGenerationHandler implements RequestHandler<S3Event, Void> {
    @Override
    public Void handleRequest(S3Event input, Context context) {
        // Handler Code Here
        return null;
    }
}

```

4.2.1 Java - Thumbnail Generator

In order to handle the dependency management we use Apache Maven. However, the serverless framework also supports Gradle by choosing a different template. By default, the serverless framework includes all the necessary libraries in order to allow the implementation of the functions. Functions for Lambda written in Java have to implement an interface called RequestHandler in most cases. Input and return types of the function have to be set through generics as shown in Listing 4.2. The S3Event class shown in this listing is provided by an AWS library[8]containing the event. This class can directly be used for the handling of events from the S3 object storage.

Amazons S3 events only include the path of the files that have been added i.e. the ones the function should generate the thumbnail requiring the developer to handle the download from the object storage within the function. Our first step after the function gets triggered, therefore, is the download of the files that are included in the event. After the download every image gets converted into a thumbnail which will be uploaded to the output bucket in the last step. This makes the function way more complex than the Azure version of the same function. Because most of the tasks we have to implement manually here are done in the background with Azure, we describe the Azure implementation in Section 5.2.1. When using Amazon's libraries to interact with most of their services authorization is not something the developer has to be concerned about, the previously assigned IAM roles grant access to the services without credentials. The only thing developers need is the name of the bucket the function should access. Everything else is handled in the background. These parameters can be set by using environment variables in order to make later modifications simpler. However these libraries usually have to be included as dependencies separately because the boilerplate itself does not include them.

Amazons support for Java Runtime environments is also rather limited. At the time of writing this Lambda only supports Java 8 based on OpenJDK [Ser19d]. Later versions of Java such as Java 11 are not supported yet.

[8]The library that needs to be included is `aws-lambda-java-events` . The initial generated boilerplate does not include this package.

4.2.2 Go - ToDo API

The Go runtime for Lambda is quite different to other runtimes like Java or .NET because this runtime does not really have any special dependency apart from libc because any other dependencies are shipped within the function binaries. Every function is built like a regular Go application wich produces a statically linked, i.e all dependencies are included, binary.
To write a function the developer should use the `aws-lambda-go` [Ser19f] package, which includes the code relevant for the invokation of the function as well as some structs for events like a HTTP event or a S3 event and the `aws-golang-sdk` [10] used to interact with Amazon services apart from lambda. The developer has to implement a handler Function that will implemnt the expected behaviour of the function this handler functions can either use predefined events, like the S3 event or the API gatteway event, or custom data structures with some limitations. As you can see in Listing 4.3 the developer also has to implement a main method within the main package for every function. This implies a problem because Go, like most other programming languages, only allows one `main` method in the `main` package for every binary[11]. Of course the Handler function does not have to be in the same package as shown in this example it could also be in a seperate package as long as its referenced in the `lambda.Start()` operation within the `main` method.
Since the Go runtime can only handle one function per `main` method it is necessary to build one binary for every function [Ser19c]. This results in quite large deployment payloads because a Go binary easily reaches 10 megabytes when including some libraries. Looking at the ToDo API which has five functions. As shown in Figure A.1 every function binary is roughly 12 megabytes in size resulting in 60 megabytes of total deployment payload. This is significantly larger than, e.g., a typical JavaScript deployment bundle size. Even though storage is cheap nowadays this might cause extra cost in comparison to other runtimes in rare cases. Since these binaries get compressed into a ZIP archive a large reduction can be achieved escpecially since these binaries contain a lot of identical sections.

Because Go libraries are usualy shipped as source code we can investigate the way Lambda handles cold start on this runtime. The following steps happen if a Function has to perform a cold start [Ser19f].

1. The binary gets fetched from storage.
2. The Lambda runtime defines an environment variable setting the port on wich the function should listen for invokations.
3. The function binary gets launched and it listens on the defined port.
4. Once the function binary is ready the runtime will fire a RPC request to execute the function.
5. The function will execute and return a result.
6. The function binary will wait for further invocations

[10] https://github.com/aws/aws-sdk-go
[11] In Go the `main` method must always be in the `main` package. For more information see: https://golang.org/doc/code.html

Listing 4.3 Boilerplate code for a Golang based Lambda function for handling a HTTP event

```go
package main

import (
    "context"
    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
)

func Handler(ctx context.Context, bb events.APIGatewayProxyRequest) (events.
APIGatewayProxyResponse, error) {
    // Code ommited for simplicity
    return resp, nil
}

func main() {
    lambda.Start(Handler)
}

```

After the function is finished, it will not directly be terminated. Instead, it waits for more requests. These requests are fired by the AWS Lambda runtime if another call to this function has occured, executing step 4 and 5 again. If no further requests arrive within a certain time frame the function binary will be terminated [Ser19f].

While configuring DynamoDB did not take a great deal of effort, it was necessary to create the table manually using the `resources` section in the `serverless.yml`. When configuring DynamoDB it is required to include the schema of the primary keys. This means that the name and type of the used primary keys have to be configured in the `serverless.yml`. The reference to the database in code is initialized using the name of the table that also has to be set in the configuration file and is passed into the function using an environment variable [Fra17a; Fra17b].

4.2.3 JavaScript - Event-Processing

Writing functions in JavaScript is easier because a function that does not interact with any AWS services does not need a library to define a function. In fact libraries are only needed to actively interact with the services, by pushing or pulling messages within the functions code for example. The libraries used for this purpose were the official Amazon libraries for interaction with SNS and SQS these libraries do not have to be included in the source code uploaded to AWS because the NodeJS runtime used by Lambda does have these libraries preinstalled. To interact with the MySQL database a MySQL driver specificly designed for serverless applications was used[12]. This library wraps the regular MySQL driver to make the use in serverless enviroments easier. Dependencies apart from the NodeJS standard library and the AWS specific libraries are managed by NPM.

Listing 4.4 Portion of the `serverless.yml` used to define a SQS Queue

```
resources:
  Resources:
    SQSQueue:
      Type: "AWS::SQS::Queue"
      Properties:
        QueueName: some_queue_name
```

If a function gets triggered the events data is passed to the function as a parsed object. Since JavaScript is a untyped language you can access the events attributes directly as long as you know the name of the wanted attribute. Even though we do not need a library in this case the function is still highly vendor specific because the schema of the event is tightly coupled to the cloud enviroment.

A example for the definition of the runtime path is `formatprocessing.formatStateChangeEvent` the first portion, left from the `.`, is the name of the source file and the second portion is the name of the function. To ensure the runtime can locate the function the developer must export the function by ensuring `module.exports.<function name>` is pointing to the function. Directly setting this attribute to the function is a very commonly used approach for this as shown in Listing A.3. This listing shows a `format` function of the event-processing use-case.

While the implementation of the functions for this use-case could be considered simple, the configuration of the services is the most complex of the four use-cases. Configuring SNS is relatively easy because you just define the name of the queue or topic and the serverless framework will ensure the topic is created [Fra19c]. However, configuring SQS requries some further adjustments. Instead of just handing the name of the queue over to the trigger the ARN[13]of a SQS queue is needed. Such a ARN is available after the queue is created. Since the serverless framework does not handle the creation of a SQS queue the developer has to define a queue as seen in Listing 4.4 under the *resources* section of the `serverless.yml` to make sure the queue will be created during deployment. After the definition, the ARN can be composed in oder to configure the trigger [Fra19d]. Just like in any other service used, the appropriate IAM permissions have to be set in order to allow the functions to access the resources.

The most effort in terms of configuration is the Amazon Aurora (RDS) database used to store the processed events. By default RDS instances are deployed in a Virtual Private Cloud (VPC) environment to prevent the exposure of the database to the whole internet. In order to access the RDS instance the functions also need to join the VPC. Configuring this was quite tedious. Since the VPC has to be created, the RDS instance needs to join the VPC and lastly the functions need to join the VPC. After ensuring the function joins the VPC the creation of the connection to the database is very simple and familiar the functions connect to the database using credentials handed over to them by enviroment variables. Of course the developer has to make sure the variables are set to the correct values in the `serverless.yml` [Zha18].

[12]https://github.com/jeremydaly/serverless-mysql

[13]Short for Amazon Resource Names. It represents ar unique identifier for a Amazon resource [Ser19a].

4.2.4 C# - Matrix Multiplication

The .NET Core runtime is quite similar to the Java based one. The serverless framework has directly created a project with the required dependencies to get started. For dependency management NuGet which is part of Microsofts development toolchain for .NET Core is used.

In comparison with Java functions get invoked differently. Instead of implementing an interface the developer just writes the function with input and return parameters. These parameters can either be AWS specific events or just simple classes containing information as long as they can be marshalled and unmarshalled by the JSON serializer. To ensure Lambda knows what function it should invoke you have to define the runtime path properly. The runtime path must point to the specific function that should be called. Such a definition looks like this `BTLambda::MatrixMul.Lambda.Handler::CreateMatrix` here the `::` acts as a delimiter. The section before the first `::` (`BTLambda`) is the name of the assembly that is defined in the `.csproj` [14] file. The middle section represents the namespace path to the class implementing the functions[15]. The third portion represents the name of the method (*CreateMatrix*). Apart from that the development experience is very similar to the one with Java.

4.2.5 Function Composition

Function composition on AWS is done using Step Functions. Step functions is a standalone AWS service that allows you to model state machines using a YAML- or JSON-based DSL. an example for such a state machine can be seen on Listing A.5.
Generally function orchestrators can be categorized into two categories, namely: 1) external scheduler based function orchestrators or 2) functions implementing special functionality to orchestrate other functions. Step functions fits in the first categroy because the orchestration engine itself is not a real function and could be considered a specialized workflow engine [BCF+17].

A state machine can be composed of multiple different state types, including:

- **Task**: This state type references to a function that should be executed when in this state [Ser19e].
- **Choice**: This state can be used to decide what the next state should be based on a condition, for example by looking at the JSON object that is passed to this state [Ser19e].
- **Parallel**: Allows the definition of multiple sub state machines that will be executed in paralell. Once all sub state machines have completed a transition to the next state of the Paralell state will occur [Ser19e].
- **Fail and Succeed**: Can be used to stop the execution with either a Fail or Succeed result [Ser19e].
- **Wait**: Waits a certan time until it continues to the next state [Ser19e].

[14]The `.csproj` file defines a C# project its content is comparable to Mavens `pom.xml` . It usualy includes the definition of metadata, like the application type, and the projects dependencies.

[15]Basically identical to a Java classpath.

Listing 4.5 A example of the data object passed through when running a matrix multiplication

```
{
    "MatrixSize": 4000,
    "MaxValue": 1000,
    "CalculationID": "0f976e9a-a97f-4647-82cb-3b5442eea750",
    "WorkerID": null,
    "WorkerCount": "5"
}

```

- **Pass**: Directly passes the input through to the output by default however this can also be used to append certain values to the input object [Ser19e].

By default Step Functions takes the output of the previous state and directly passes it through as the input of the next state. For the transfer of data the JSON format is used. As mentioned previously in Section 4.1 Step Functions only supports rather small data payloads of around 32.000 characters. For larger datasets such as 500×500 matrices, some sort of caching is required. As a result, with caching support added, the object passed to Step Functions can only contain some metadata like the matrix size, the worker id and the matrix id which is used as a reference in object storage. an example of such an object can be seen in Listing 4.5.

Writing State Machines using their DSL is unintuitive, as with the CloudFormation templates. Apart from writing the state machine using the DSL, AWS does also offer a framework that can be used to write state machines using Java[16]. Definitions done with this API will be compiled into the Step Functions DSL in the background. While this feature simplifies the automatic generation of the state machine it does not reduce the effort to write a state machine. Essentially, the serverless framework is not helpful for defining of the state machine. For deployment automation, a separate plugin can be used, which requires a YAML version of the state machine as an input. As a result, to implement this use case we define the state machine manually[17]. The graphical representation of the Step Function state machine for the matrix multiplication use case can be seen in Figure A.2.

With the Matrix multiplication use-case it might be interesing to increase the worker count to potentially get better performance. Doing this is a lot of copy and paste with minimal modifications.

While the configuration of the state machine might be tedious, it comes with some interessting benefits: Every launched workflow can be traced to find out in what state it currently is. You can also take a look at the input and output values for every step that already has been executed.

The simplest way to launch a StepFunction workflow is by a HTTP request. This can be achieved by binding the workflow to an API Gateway. Since this option comes with limitations, an invocation function might be used to prepare the input for the function workflow. This could be useful to ensure the input payload is in the proper format. It is also possible to wrap the payload returned from the API gateway endpoint to launch the function.

[16]https://aws.amazon.com/blogs/developer/stepfunctions-fluent-api/
[17]https://serverless.com/plugins/serverless-step-functions/

The deployment is done using a Step Functions plugin for the Serverless framework. This plugin allows the direct definition of state machines within the `serverless.yml`. It also handles the automated deployment of the state machine [Hor19].

4.2.6 Development Workflow

The development experience with AWS Lambda is consitent across all programming languages. Thanks to the use of deployment automation technologies such as CloudFormation or Serverless framework. Configuring the functions always works the same way no matter what programming language is used. This is one of the huge benefits when decoupling the trigger configuration from the whole function code.

One huge issue with Lambda is the lack of local test runtimes that would make the development experience better. While the serverless framework offers the ability to execute a function locally this will not always work. While most programming languages are supported the execution will fail due to missing access credentials to the services [Fra19a]. Apart from the cost for ever function invokation, this takes way longer than just running the function in a local test environment.

While the Serverless Framework provides a local runtime for the programming languages supported by AWS. As a consequence the execution of functions that do not need any external services works without any issues. However, the whole problem becomes more complex if a function accesses AWS services. Since AWS generally uses IAM roles to grant access to services instead of specific service credentials. It is possible to get access to AWS services using the credentails the AWS and Serverless CLI use to access the services. However this procedure requires further investigation in order to find out what services can be accessed. Since the time benefit from executing the functions locally rather than on was considered lower than the time investment needed to get the IAM credentials to work we decided to deploy every iteration of the function in the cloud [Fra19a].

4.3 Overall impressions

Implementing on AWS with the serverless framework worked quite good. The serverless framework did simplify many processes and it also delivered a consistent development exeperience across all programming languages used. However there were some problems that had to be resolved during the implementations. Of course the problems listed here do not regard the migration because AWS was the initial implementation these problems can be considered more generic regarding the development on AWS in general.

The first problem encountered were the initially missing libraries for service specific events like the S3Event. While the problem can be resolved by just adding the appropricate Maven artifact it was just slowing down the implementation progress.

The biggest problem during the implementation was the limitation of the input in Step Functions. A very intuitive approach to implement the problem would be a Context object containing everything the function needs to perform its task. However this initial approach had to be quickly be changed because of the input limitations. The solution for this was the use of S3 as an external chancing mechanism.

5 Implementation on Microsoft Azure

While Microsoft Azure Functions comes with some great improvements compared to AWS Lambda it also comes with drawbacks. The first one I have encountered while implementing the use-cases on Azure is the rather limited support of programming languages. These beeing 1) .NET based languages (C# and F#) 2) Java, including JVM based languages 3) JavaScript, based on Node.JS including transpiled languages like TypeScript 4) Python 5) PowerShell or the second version of the Functions SDK[Mic19f]:

Due to the fact that Go is not supported by Microsoft Azure Function, we had had to reimplement the whole ToDo API use-case in another programming language. Another problem we encountered is that the source code of all other use cases cannot be migrated as-is and requires severe modifications to make it run on MS AF.
While the vendor specific services might be the main reason why modifications are needed, it is definately not the only huge problem. Without some level of abstraction of the vendor specific interfaces and entrypoints developers will have to rewrite the code mostly or in the worst case completely from scratch. We have encountered this problem with the thumbnail generation use-case. This had to be rewritten almost completely. The only part we were able to keep was the class that actually generates the thumbnail. This was only achieved because we intentionally did not externalize any code into a seperate package as described in Section 4.1 to illustrate this problem.

Another difference to Lambda is the way Azure handles the configuration of triggers and inputs. While Lambda does require develoers to define these factors using the Web UI or through CloudFormation, some programing languages on Azure define the configuration of the functions within the functions source code the programming languages supporting this approach include C# [Mic19b] and Java [Mic19c]. When using other programing languages like JavaScript the configuration of the functions is still done using a configuration file for every function called `function.json` [Mic19b]. While source code configuration approach requires the developer to recompile whole project in order to alter a trigger, subjectively, this approach results in a more pleasant development experience compared to AWS Lambda because you do not need to write a second file describing the cloud archiecture in many cases. For example, this is relevant for cases when developers do not need external services other than a StorageAccount[1], which is created automatically when deploying a function to store its binary contents. This StorageAccount can be used for storage of the functions data at least in development [Pfe18]. When deploying in production this might not be the best idea and a seperate StorageAccount might be needed. The corresponding connection string has to be defined in a configuration file before the deployment is done.
This represents one of the biggest downsides of Azure. While the Azure Resource Manager[2] can be considered Microsofts version of CloudFormation for their platform this work does not investigate

[1]https://azure.microsoft.com/en-us/services/storage/
[2]https://docs.microsoft.com/en-us/azure/azure-resource-manager/

the suitablity of Resouce Manager for serverless applications. The services of use-cases that needed more than just a storage account were created using the Azure CLI and a Bash Script. The other use-cases that just need a storage account did automatically create one when deploying using Maven or JetBrains Rider.

5.1 Use-Case implementations

Thumbnail Generator The **Thumbnail Generator** use-case is based on the Azure Storage Account with two blob containers. One for input and one for output. Compared, to S3 a container can be considered a bucket with the only main difference naming. A bucket name in AWS is global across all with the in S3 and, as a result, the name has to be unique because simple names like `input` or `output` are almost always used. This is not the case with the storage account. Here the container name is only required to be unique within the same storage account. However the name of the storage account has to be unique in a global namespace.
The functions have been ported to Azure Functions in Java. A rewrite in a different programming language was not necessary. However most of the code could not be reused because of the way functions are invoked. More details are described in Section 5.2.1.

Matrix Multiplication The easiest use-case to migrate was the **Matrix Multiplication** use-case. One of the main reasons for this is probably the design of the implementation. Decoupling the business logic from vendor specific APIs using made the migration very easy. Only the Interface for retrieving the cached data had to be replaced. With a simple dummy implementation. The biggest effort for this migration was the migration of the function workflow definition to Azure Durable Functions.

ToDo API The ToDo API use case had to be completely rewritten to make sure it worked with Azure. Apart from that, the Table storage feature of the storage account was used as the storage backend for this use-case. While Microsoft also offers CosmosDB[3] an alternative solution the table storage looked like the perfect fit in replacement for DynamoDB.
Both the storage account and CosmosDB support the same table API, allowing an easy migration between both the table storage and CosmosDB according to Microsoft. Since CosmosDB supports multiple database APIs like Cassandra or MongoDB it becomes clear that these APIs are just wrapping the internal data storage used by CosmosDB [Mic19d].

Event-Processing The event processing use case was the most complicated to migrate considering the amount of external services used. Here, the MySQL based RDS instance was replaced with a *Azure Database for MySQL*[4] instance, which can be considered a suitable replacement. The only thing that is required to migrate this service is the change of the endpoints and credentials. To replace SNS (pub/sub messaging) and SQS (messaging) the *Azure Service Bus*[5] was used. The

[3]https://azure.microsoft.com/en-us/services/cosmos-db/
[4]https://azure.microsoft.com/en-us/services/mysql/
[5]https://azure.microsoft.com/en-us/services/service-bus/

Listing 5.1 Structure of the `local.settings.json` file for the event processing use-case

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "<ConnectionString for the StorageAccount>",
    "ServiceBusConnection": "<ConnectionString for the Service Bus>",
    "DBUsername": "<MySQL Username>",
    "DBPassword": "<MySQL Password>",
    "DBEndpoint": "<MySQL Endpoint>",
    "DBName": "<MySQL Databse Name",
    "FUNCTIONS_WORKER_RUNTIME": "node"
  }
}

```

service bus supports both types of queues out of the box. Migrating to the service bus is not as trivial as a simple replacement. Apart from the implementational aspects pub/sub based messaging requires a subscription for every function that wants to read or write to the topic this meaning that developers have to create at least two subscriptions for every topic.

5.2 Technical Details

The interaction between services is done using connection strings. They basically are a Microsoft specific way to connect to other services like databases or storage accounts comparable to a URL, which contains credentials. These connection strings are also used with Azure Functions to connect to other Azure services.
In order to prevent developers from having hardcoded connection strings the so called *local.settings.json* file is used for configuration of the local development runtime. During deployment a flag can be set to transfer the configuration into the cloud [Mic19h]. The only relevant part for the implementations shown here are the connection strings. These get defined in a string to string map (Name to Connection String). Other parameters like logging and metrics can be configured using another file called `hosts.json` file [Mic19e].

To develop applications on Azure Functions developers have to install the Azure Function Core Tools[6] (`func` command) and the Azure CLI[7] (`az` command). To create resources like a storage account the `az` command is used. To localy execute and deploy functions on Azure the `func` command is used. When working with development tools like Maven or JetBrains Rider[8] these commands are executed in the background. They also configure the `local.settings.json` file with the appropriate credentials and allow the automatic creation of missing resouces as seen in

[6] https://www.npmjs.com/package/azure-functions-core-tools

[7] https://docs.microsoft.com/en-us/cli/azure/install-azure-cli

[8] A small note: Azure might support more Automation features in different development environments like VisualStudio on Windows but these were note used to write the code for this work. A description of the development environment used can be found in Appendix B.2

Listing 5.2 Vendor spectific Java code of the thumbnail generators conversion function

```java
@FunctionName("Create-Thumbnail")
@StorageAccount(Config.STORAGE_ACCOUNT_NAME)
@BlobOutput(name = "$return", path = "output/{name}")
public byte[] generateThumbnail(
        @BlobTrigger(name = "blob", path = "input/{name}")
                byte[] content,
        final ExecutionContext context
) {
    try {
        return Converter.createThumbnail(content);
    } catch (Exception e) {
        e.printStackTrace();
        return content;
    }
}

```

Figure A.3. Since three of our four use-cases only require a storage account that is automatically created when deploying the application Listing 5.1, shows the `local.settings.json` for the event-processing use case. This example shows that all configuration values are placed in the `Values` child object. While all parameters apart from `AzureWebJobsStorage` `FUNCTIONS_WORKER_RUNTIME` are use-case specific these to exist in every `local.settings.json` file that gets used to deploy a function to Azure. The purpose of the `AzureWebJobsStorage` is to define the connection string for storing the applications data and `FUNCTIONS_WORKER_RUNTIME` is used to define the functions runtime environment [Mic19h].

The Azure Functions Core Tools CLI can be used to run the Function Host, an application that allows the local execution of functions. This runtime emulates the whole Azure Function environment if all credentials to the services are supplied [Mic19h].

The Serverless framework does not offer complete support for Azure. In fact it does not even provide templates for Java and C# the only programming language it supports on Azure is JavaScript[9] and since it does not really make the deployment of application easier we have decided to avoid the use of the Serverless Framework on Azure.

5.2.1 Java - Thumbnail Generator

To generate an empty project for the Java runtime Apache Maven[10] is used with a archetype supplied by Microsoft to serve this purpose. Maven in combination with the Azure Functions CLI is used to build the functions, deploy them to Azure or to run them locally.

[9] https://serverless.com/framework/docs/providers/azure/cli-reference/create/
[10] https://maven.apache.org/

As we have already mentioned in the introduction to this chapter most of the function parameters like triggers and function types are defined using annotations. Listing 5.2 and Listing A.6 shows vendor spectific code needed in order to make the thumbnail generator work. The annotations serve the following roles

- **FunctionName** defines a function, i.e. to tell Azure that this method should be represented as a function. The string in the brackets defines the name of the function.
- **StorageAccount** defines the connection string used. At first glance it seems uninutitive how this works, since developers actually have to put the name of the connection string, defined in the `local.settings.json` file, instead of the actual connection string here.
- **BlobOutput**: This annotation is used to get write access to the Blob Storage of the Storage account. It can either be used on the function, as shown in Listing 5.2, to store the returned value at the defined path or it can be used as an output binding as shown in Listing A.6.
- **BlobTrigger**: This annotation is used to define a trigger for the blob storage. The parameters allow developers to set the folders, also called containers, it should listen on. The trigger causes the function to be executed every time a document is inserted or modified.
- **HttpTrigger**: Functions with this trigger can get invoked using HTTP requests. The accepted methods and the authorization level are set as parameters to the annotation. An important note here is the way the HTTP paths are built. they can either be specified explicitly in the annotatation, by setting the value of the `route` parameter, otherwise the function name will be used as default.
- **BindingName**: Retrieves a Parameter from the trigger. In the case of Listing A.6 this retrieves a HTTP query parameter with the name `name`. This value is also used in the blob storage output binding to define the file name in the Blob storage.

During compilation the annotations get processed into configuration files for the function runtime. The developer does not really notice this step, it is handled in the background.

When comparing the Azure Function implementation to AWS it is very obvoius that Azure Functions are way smaller in terms of code size, as shown in Table 5.1. For this comparison, empty lines and log outputs have been removed from both implementations. The mime type check in the upload function has also been removed on AWS lambda because this feature was not implemented in the implementation of the upload function on Azure Functions. The lines counted were the lines needed to explicity tell the FaaS runtime that this is a function. In order to achieve this the code size of the class implementing the function on Lambda was counted. On Azure the Method body with all annotatons was used for counting. Imports, comments and package definitions have always been ingnored.

Function	AWS	Azure
Upload Image	29	18
Generate Thumbnail	52	14

Table 5.1: Comparison of code sizes of the Java functions for AWS Lambda and Azure Functions

Listing 5.3 List function of the ToDo API use-case implemented on C# for Azure

```
    [FunctionName("lst")]
    [StorageAccount("AzureWebJobsStorage")]
    public static async Task<IActionResult> ListItems(
        [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route = null)]
        HttpRequest req,
        [Table("todo")] CloudTable todoTable,
        ILogger log)
    {
        var query = new TableQuery<ToDoItem>().Where(TableQuery.GenerateFilterCondition("
PartitionKey",
            QueryComparisons.Equal,
            "http"));
        var result = await todoTable.ExecuteQuerySegmentedAsync(query, null);
        return new OkObjectResult(result.Results.Select(e => e.ToToDoDTO()).ToArray());
    }

```

Another factor for the increased code size is the rather different approach used by AWS. With AWS it is not possible to receive the contents of an object through the event object itself, as one can see in the example event in Listing A.1 [Ser19g]. The developer has to download the data in code by looking at the data from the event. The upload also has to be handled in the functions code. This is not needed in Azure functions since all these tasks are handled by the runtime resulting in less code.

5.2.2 C# - ToDo API and Matrix Mutliplication

The development experience is very similiar to Java. Functions and their triggers are defined using Attributes (C# equivalent to Annotations in Java). Listing 5.3 shows the `list` function of the ToDo API use-case. This listing shows that the interaction with tables in a storage account is also done using attributes the `Table` attribute the runtime will handle the injection of the corresponding `CloudTable` object that can then be used to interact with the Table [Mic18c].

The development on .NET worked rather good apart from some incompatibilities between library versions. Taking the latest version of the function core library and the storage account library (needed to interact with the table storage) did not work out of the box. Because the version of the storage account library was newer than the core function library the triggers could not get identified properly causing the functions to be ignored during deployment. Reverting to an earlier version using NuGet[11] resolved this issue. We only encountered such pitfalls during the development on C# and also only with Azure Functions.

This section focused on the general implementation aspects on C# as well as the reimplemented ToDo API. A more in depth description of the implementation of the matrix multiplication use-case can be found in Section 5.2.4.

[11]The tool used for dependency management on .NET

Listing 5.4 JavaScriptcode for a `format` function in Azure Functions

```
module.exports.handler = async function (context, item) {
    let tempEvent = item;
    let message = "Message here";
    let evt = {
      type: tempEvent.type,
      source: tempEvent.source,
      timestamp: tempEvent.timestamp,
      formatting_timestamp: getUnixTime(),
      message: message
    };
    let messageString = JSON.stringify(evt);
    context.bindings.queueOutput = messageString;
    return;
};

```

5.2.3 JavaScript - Event-Processing

JavaScript does not support the inline definiton of triggers and outputs like Java or C#. This means that these have to be defined in a separate file called `function.json` for every function. The parameters developers have to define are very simmilar to the ones one has to define in the attributes/annotations because the `function.json` file is required for deploying to Azure Functions. On the other platforms these files are just generated from the annotations during the build process [Mic18a]. Writing the functions configuration can be tedious. Missing code completion in these files makes the implemantation very time intensive because developers always have to look at the documentation in order to find out how they have to specify something. In Java and C# there is code completion for the annotatations making this step way easier. But sometimes it is still necessary to look at the documentation. Subjectively speaking the definiton through annotatations felt way easiser and more intuitive compared to writing the `function.json` file. An example of the `function.json` file can be seen in Listing A.7. The funtion shown here is a very basic HTTP function producing a request from a response with no other service dependencies [Mic18a; Mic18b]. More precisely, this listing shows the `function.json` for the list function. This important because we do not mention the required access to the MySQL database in this definiton. Access to the MySQL database has to be ensured by the developer in code. To get access to the mandatory credentails developers must ensure the credentials for the MySQL database are in the `local.settings.json` , since all values defined in this configuration file are also mounted as environment variables within the functions environment, the configuration of the MySQL client is identical to AWS depending on the names chosen developers might have to change the names of the accessed environment variables [Kos18].

However the great thing about the JavaScript portion is the fact that we do not need to import any vendor specific APIs in order to interact with the topics and queues. This is achieved by just pushing the message into an array, for sending multiple messages or by setting the value of a variable, for one message [Mic18b; Mic19a]. Everything else will be handled by the runtime. But requiring no external libraries does not imply the freedom to execute the function somewhere else of course. Since this the function is still tightly coupled to the runtime environment. However it makes the emulation of the runtime environment easier because mocking the data structures behind the parameters is easier compared to mocking an entire library.

Apart from that Azure does not allow the definiton of multiple functions in one file like it is possible with AWS. Every Function has to be placed in a separate directories with its corresponding function configuration (`function.json`). According to the documentation it is possible to share code through separate folders [Mic19a]. However this was not necessary for our implementation since we did not have complicated any shared methods.

Function	AWS	Azure
Ingest	35	25
Format	21	13
Insert in DB	24	22
List	21	24
Latest	21	24

Table 5.2: Comparison of code sizes of the JavaScript functions for AWS Lambda and Azure Functions

Taking a look at the AWS Implementation of a `format` function in AWS Lambda shown in Listing A.3 we can identify the code shown in Listing A.4 as reusable[12]. This code snippet can also be found in the Azure version of the function shown in Listing 5.4. When comparing the two snippets from Azure and AWS side by side, it can be observed that the implementation on AWS is longer, in terms of lines of code, than the Azure version. This seems to be pretty consistent across functions writing to message queues and topics implemented in JavaScript as shown in Table 5.2. For the other functions the code size of the Azure version can actually be longer than the one on AWS. Due to the very small code size of all the functions implemted for this use case the real program logic is very small and does not really count into the size of the function. For example the only vendor unspecific logic in the `ingest` function is the if statement finding out what type of event was received.

Unlike the previeous use cases this use-case uses more than a storage account. As a result the creation of the resources for the Event-processing use-case have to be created. To make the function executable we must also ensure the credentials are set in the `local.settings.json` to allow local execution and the deployment to Azure. For Azure supports the automated creation of services using the Azure Resource Manager, this work does not use these techniques for Azure. Instead the Azure CLI in combination with Bash is used to create the resources for this use-case. Creating the resources needed and accessing the connection strings is simple using the CLI. The biggest challenge is the automated creation of the `local.settings.json` this procedure was automated using the `jq` [13] utility and the Azure CLI. After the creation of all the resources the functions can be deployed by running `func azure functionapp publish $FUNCTION_APP_NAME -o` where `$FUNCTION_APP_NAME` represents the name of the Function App on Azure. To ensure the `local.settings.json` is uploaded to Azure the `-o` flag must be passed to the Azure Functions Core Tools CLI.

[12]Message concatenation in the code sample has been ommited to improve the readability of the code.

[13]`https://stedolan.github.io/jq/`

Listing 5.5 Part of the orchestration function for the matrix multiplication use-case

```
        [FunctionName("OrchestrateMatrixMultiplication")]
        public static async Task<Matrix> OrchestrateMultiplication(
            [OrchestrationTrigger] DurableOrchestrationContext context
        ) {
            var size = context.GetInput<string>();
            MatrixCalculation c = await context.CallActivityAsync("GenerateMatrix", size);
            Matrix result = null;
            if (int.Parse(size) < 10)
                result = await context.CallActivityAsync<Matrix>("SerialMultiply", c);
            else
                // Has been ommitted
            return result;
        }

```

5.2.4 Function Composition - Matrix Multiplication

Unlike AWS Step Functions Function Workflows are written as code in one of the supported programming languages. A domain specific language is not used. The workflow itself is written in a function too. As a result the function orchestration in itself is a function and it does not rely on an external scheduling solution like AWS Step Functions [BCF+17]. By default all child functions, called activities in Durable Functions, are called asynchronously. If you need to wait for them to complete before the workflow can proceed the `await` functionallity can be used [Mic18d]. This is the native approach of the programming language to asynchronous programming [Mic19g], creating a unified programming experience for writing workflows. To model the workflows C#, F# or JavaScript must be used for the implementation of the functions and the workflow definition [Mic18f]. an example of an orchestrating function is shown in Listing 5.5. The code you write is very similiar to normal programm code. The only main difference is the way activities are called. In comparison to just calling the function like you normally would. It is necessary to use the `DurableOrchestrationContext` to call the function. Invoking the child activity is also not done by the method name, but instead by the name set through the `FunctionName` attribute [Mic18e].

While Step Functions limits the size of the input payload for functions to approximately 32.000 characters, Azure does not come with such a limitation. Instead Azure automatically handles caching of larger payloads in a storage account. To reduce file size all files larger than 60KB get compressed [LSP+18]. As a result there was no need to implement the caching ourselves instead the caching mechanism provided by Azure was used.

Azure Durable Functions categorizes functions involved in a function workflow into three categorizes 1) client functions, i.e., functions that will create, launch and manage the workflow through an external trigger, 2) orchestrator functions, i.e., functions that define the workflow and 3) activity functions, i.e., the functions that perform the actions in the workflow. In our implementation the client function is triggered using a HTTP trigger, this function also checks the data sent to ensure the workflow can process it. To define a client function an external trigged, for example, a HTTP trigger must be bound to the function and the method header definiton must contain a `DurableOrchestrationClient` parameter with the `OrchestrationClient` attribute attached to it, as shown in Listing A.8. Unlike AWS such a client function is mandatory to launch a function, while it was possible to directly bind the AWS Step Functions workflow to an endpoint from the API Gateway. To define an orchestrator function the `DurableOrchestrationContext` must be passed to the method implementing the workflow with the `OrchestrationTrigger` attribute attached to it, as shown in Listing 5.5. Listing A.9, shows the definiton of the `GenerateMatrix` activity function as shown there activity functions need the input object, in this case a string[14], with the `ActivityTrigger` attribute attached to them [Mic18e].

	Functions Size	Repository Size	Workflow definition size
Azure Durable Functions	69	60	44
AWS Step Functions	77	82	125

Table 5.3: Comparison of the code- and workflow definition sizes of matrix multiplication use case on AWS and Azure

As illustrated in Table 5.3[15], the code size and the size of the workflow definition on Azure is smaller than on AWS. The main reason for this is the way simpler implementation of the caching interface. While it is just an in memory storage on Azure, requiring no configuration. The AWS implmentation has to read the environment variables to determine the name of the caching bucket and its region. Also the S3 client must be initialized, requring lines of code to do so. All of these steps are not necessary on Azure Functions reducing code size. The workflow definition is almost three times smaller because it is defined in code rather than using a verbose DSL to define the workflows. For example The parallel portion requires 13 lines per worker resulting in 65 lines to define all five workers. On Azure the definition of the complete parallel multiplication takes 16 lines of code.

Testing a Durable Functions workflow is very easy because the Function Host, part of the Azure Functions Core Tools, can run Durable Functions on the developers machine. Speeding up the development process by a lot. However this feature comes with one downside: it requires a storage account to store the payloads, even when running locally. This feature might support the afforementioned Storage Emulator but this work did not investigate the support for it.

One issue with Azure Durable functions is the lack of tracablility in comparison to AWS Step Functions. On Azure Durable Functions the launch of a workflow using the client function returns a URL, with it one can retrieve the current state of the workflow however this feature is very limited. The current state does not inform the caller about the activites that currently get processed instead the current state just tells the caller whether the workflow is running, has failed, was terminated or did run successfully. Compared to the in depth tracablility available through the AWS web ui on Step Functions. It is important to note that MIcrosoft advertises better diagnosis when combining Durable Functions with Application Insights[16]. However this work does not investigate this further[17].

[14]String was used to pass the size because an integer variable did not work.

[15]For this comparison imports, log outputs, empty lines and comments have been removed on both platforms. The size of the workflow definiton on Azure is the length of the method defining the workflow. On Step Functions the length of the YAML file used to define the state machine was used. On Azure we have ommitted the client function to launch the workflow since it is not included in Step Functions too.

[16]https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview

[17]https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-diagnostics

5.3 Overall impressions

The biggest problem encountered during the migration was the missing support for Go making a complete rewrite of the application in another programming language necessary. While this was reasonably simple for the ToDo use-case it is merely impossible when considering more complex functions regarding code size. Making the migration to Azure impractical. The only real solution for this problem was the rewrite of the complete application.

The unpredictable version issues we have encountered while implementing the ToDo API use case are also something of concern. While the libraries used were official Azure libraries this is probably a general problem with .NET Framework.

While Microsoft offers Azure Resource Manager to automatically create resources based on templates this approach was not taken in pur implementation since we have encountered it after the implementation has been finished. With the assumption that Microsoft does not offer such a service we decided to implement the creation of the resources using BASH and the Azure CLI. Of course this approach also works, but it becomes very tedious when working with a large amounts of service instances. The script we created creates all the services needed to deploy the functions on Azure and it also writes the local configuration to allow further development of the application.

While the Function Core tools CLI allows you to locally run functions with any trigger. The external services still have to be created on Azure. This is still convenient because a restart of the function host is much faster than the redeployment of the complete function.

Generally the development experience feels less consistent compared to AWS. Of course one reason for the consistent development experience is the use of the serverless framework. However Some things are just generally more consistent on AWS like the definition of triggers. On AWS these have to be defined externally while it depends on the programming language when using Azure Functions.

6 Implementation on IBM Cloud Functions

The offering from IBM is different in many regards compared to AWS Lambda and Azure Functions. First of all the complete offering is based Apache Openwhisk, an open source FaaS platform. Secondly IBM Functions has no limits in terms of supported programming languages. Everything that is able to run in a Docker container can be deployed as a function. It also supports JavaScript, Swift, Java, Go, PHP, C# and Ruby directly for better performance [IBM19e].

Functions deployed using a Docker image generally do not have any dependencies. They can also be vendor-agnostic if no provider-specific services are used. The input parameters for the function are passed into the binary by command line arguments as JSON. The output of a function is the last line of console output (UNIX `stdout`) which should obviously also be JSON [Rab17; Rab18].

The wide range of supported programming languages is a huge benefit in comparison with Azure Functions and even AWS Lambda. Both of these services limit the tenant quite seriously on what programming languages he can use to write functions.

When deploying functions to IBM Cloud or OpenWhisk the following terms are frequently encountered in official documentation:

- **Action** is the code that will be deployed on the FaaS paltform, i.e., a function in other FaaS platforms [Des17]. IBM Cloud actions can either be created using the web UI or the `ibmcloud` CLI with the functions plugin which basically wraps and extends the OpenWhisk CLI to allow easy interaction with IBM Cloud Functions.
- To cause functions to be executed due to an external event they have to be triggered. TIn Openwhisk, this is the responsibility of **Triggers**. A trigger itself will not launch an execution this is done by defining a **Rule** connecting the trigger and the action together [Fou19f].
- Many triggers subscribe to a **Feed** of events. A trigger will be fired when a new event is detected on the feed. This mechanism can be used to expand OpenWhisk with custom event sources [Tho16].
- The invocation of a function is refered to as **Activation** [Fou19f].

6.1 Use-Case implementations

Because OpenWhisk supports all programing languages the initial idea was to use the AWS base implementation for every one of the use cases. However due to some limitations of the frameworks used we were not free of rewrites this time either.

Thumbnail Generator For the implementation of the Thumbnail generation use case was done in Java just like the previous implementations on AWS and Azure. *IBM Object Storage*[1] was used to store the input images as well as the generated thumbnails. The object storage offered by IBM provides an API compatible with AWS S3 object storage [IBM19a]. At the time of writing this work, the mandatory trigger for this use-case was only available in one region since it was considered experimental.

ToDo API For the implementation of the ToDo API we have reused the initial implementation for AWS Lambda in Go. For storing the data we use the *IBM Cloudant*[2] database.

Matrix Multiplication This use case was implemented using IBMs Object Storage for caching and *IBM Composer*[3] for function orchestration. The implementation of the multiplication actions was still done using C#. Since IBM Composer only supports workflow definitions in JavaScript we had to write the workflow definition in JavaScript. In addition, we had to use the *Database for Redis*[4] service from IBM that provides a managed instance of Redis, due to the fact that IBM Composer requires a Redis instance for caching when orchestrating functions running in parallel.

Event-Processing The Event-Processing use case was implemented in JavaScript just like before. The relational MySQL database is hosted using the *Compose for MySQL*[5] service. For publish / subscribe messaging *Event Streams*[6] an event bus based on Apache Kafka is used. Because IBM does not offer a queue based trigger the regular queue was also replaced with a publish / subscribe topic based on Event Streams.

6.2 Technical Details

The following sections will cover the technical implementation details of the four use cases.

Working with IBM Cloud Functions was different in many regards compared to the other implementations. During the implementation of the four use-cases we have encountered multiple problems that we had to solve before we were able to continue the development. Some issues are general others only come up in connection with a specific service or programing language.In the next paragraphs, we describe the general problems.

Unlike AWS Lambda and Azure Functions, OpenWhisk and, therefore, IBM Cloud Functions cannot handle any JSON-to-object and object-to-JSON mapping tasks in the background. Therefore, it is not possible to define a custom class as an input or output parameter. This conversion has to be done within the action. Generally OpenWhisk hands over a parsed JSON object that can be accessed

[1]https://www.ibm.com//cloud/object-storage
[2]https://www.ibm.com/cloud/cloudant
[3]https://github.com/apache/openwhisk-composer
[4]https://cloud.ibm.com/catalog/services/databases-for-redis
[5]https://www.ibm.com/cloud/compose/mysql
[6]https://www.ibm.com/cloud/event-streams

just like a dictionary. If this input type should get mapped to a specific class this must be done manually. The exception to this in our small sample of programming languages was Go because it gets compiled into regular binaries and the input payload is passed through as a command line argument. Here the action also has to handle the parsing of the JSON object. But this makes it also possbile to directly deserialize the payload into a typed object. With the output we have a similar situation here the action has to serialize the output object to JSON and it must ensure it is the last line written to standard output [Fou19e].

Actions, Triggers, and Rules in Openwhisk are defined in a function namespace to group them all together. Since IBM Cloud Functions offers two types of namespaces: IAM based namespaces and CloudFoundry based namespaces. We have encountered multiple errors because some operations are not yet supported on IAM based namespaces which we have initialy used for the deployments [IBM19g]. These issues were resolved by using a CloudFoundry namepace instead. The main problem here was the lack of documentation when creating an iAM based namespace no warning has occured that some features are not yet available. We initialy have only found a small side note in the documentation for OpenWhisk Composer on IBM Cloud Functions mentioning that the APIs only work with OpenWhisk-based APIs that don't support IAM based namespaces [IBM19b] further investigation in their documentation has verified this issue [IBM19g]. While the IAM based namespaces can also be used to get different permissions to the actions this generally differs from the IAM roles described in the AWS implementation. For our implementations the IAM based namespace just represented a different type of namespace [Dau19]. The CloudFoundry-based namespaces are clearly older because they still work with the regular OpenWhisk CLI while IAM-based function namespaces require the use of the IBM Cloud CLI with the functions plugin installed.

These two type of namespaces are a common thing across the whole set of offerings from them. Some services such as Event Streams or Cloudant are IAM based whereas the managed MySQL service is CloudFoundry-based. This is one reason why automating the deployments on IBM Cloud can become very cumbersome. In order to get an IAM-based service into a CloudFoundry namespace an alias of that service can be created. To automate the build and deployment process of all four use-cases we have used `Makefiles` from GNU make[7] in combination with the programing language specific build tool, the IBM Cloud CLI[8] (`ibmcloud` command) and the JSON processor `jq` [9].

Genrally IBM does not provide an option to run actions locally. However, local execution can be possible and depends on the programming language developers work with. For example, functions written in Go can be executed locally if a demo payload is passed to them as a command line argument. Another option for local tests is the deployment of an OpenWhisk instance using Docker Compose[10]. Compared to the appraoch of Azure Functions this is not as lightweight and it also does not support all triggers supported on IBM Cloud.

While the Serverless Framework offers support for OpenWhisk and IBM Cloud Functions, we did not choose to work with it. Since it did not seem to be a huge improvement regarding the configuration effort of the functions compared to our manual approach. Other reasons for avoiding the Serverless

[7] https://www.gnu.org/software/make/

[8] https://cloud.ibm.com/docs/cli?topic=cloud-cli-getting-started

[9] https://stedolan.github.io/jq/

[10] http://jamesthom.as/blog/2018/01/19/starting-openwhisk-in-sixty-seconds/

framework on IBM Cloud Functions include: the lacking support for cloud composer[11] and the lacking support for creating resources, apart from OpenWhisk specific ones[12]. In the following sections we will take a look at the implementations on different programming languaes.

6.2.1 Java - Thumbnail Generator

OpenWhisk supports Java through a special runtime providing better performance. As shown in Listing 6.1 the function definition is very similiar to the definition of a regular `implementation` method in Java but it uses different input and output parameter types. However the provided implementation does simplify the acess to attributes of the JSON JsonObject because it has already been unmarshalled. In the end this will also ensure that the proper object type is returned [Fou19c].

The provided runtime does not requrire a normal `main` method because the entrypoint to the function is defined by setting the path to the method during deployment, comparable to the approach of AWS Lambda. But before deployment the source has to be compiled and the resulting files have to be packaged into a JAR archive, mandatory for deployment. This can become very complex. To make the development effort a bit simpler a Maven Archetype[13] was used very similar to the procedure done on Azure Functions. Maven makes compiling and packaging of the JAR archive much more convenient.

In order to implement this use case we had to rely on an experimental feature that is currently only available in one region. To use this feature we had to create a new namespace assigned to that region. During the creation of the buckets it was also important to ensure they were hosted on this specific region. As shown in Listing A.11 the event payload of the object storage trigger is not compatible to the format provided by Amazon S3 which can be seen in Listing A.1 for comparison. In fact, the current implementation IBM provides provide will trigger the function every time something has changed in the object storage causing many unecessary function calls. The task of checking what type of event was received has to be done within a action [Deu19; IBM19f]. However one thing is simmilar to AWS: The trigger does not include the contents of the files themselves they have to be downloaded manually.

All of the configuration steps named here were done using the CLI[14] the only things that we have done using the web UI were the creation of the object storage instance, their buckets and the access credentials needed for the functions to access the object storage.

Overall the development experience was fairly smoth. Apart from some problems with the IBM Cloud CLI. For example, after changing regions, the `ibmcloud fn namespace list` command intended to list all namespaces in the current region still listed the ones from the old region, a attempt to set the namespace also failed. To resolve this we had to create a new namespace using the

[11]We did not find a note in the official documentation of the Serverless Framework mentioning support for Composer. Further investigations have not been done. The documentation can be found under: https://serverless.com/framework/docs/providers/openwhisk/

[12]https://serverless.com/framework/docs/providers/openwhisk/guide/serverless.yml/

[13]https://github.com/apache/incubator-openwhisk-devtools/tree/master/java-action-archetype

[14]For the creation of the trigger the use of the CLI was required.

Listing 6.1 Boilerplate code for defining a Java based action in OpenWhisk

```java
import com.google.gson.JsonObject;
public class MyAction {
    public static JsonObject main(JsonObject args) {
        // Function code here
        return new JsonObject();
    }
}
```

CLI before it listed the proper ones. Another issue was a occasionaly failing region change to resolve this problem we had to log out and log in again. Issues like that are expected when working with experimental software but the parts that caused these issues are not considered experimental.

6.2.2 Go - ToDo API

To implement the ToDO API use case Go was used. Because of the simplistic invocation approach of OpenWhisk we did not need any vendor specific dependencies to implement the functions. Only an iBM Cloudant specific database driver was used to make the connection easier. However this might not be necessary because IBM Cloudant is compatible to Apache CouchDB and it should therefore be possible to use a standard CouchDB database driver.

To ensure API compatibility to the other implementations an OpenWhisk API had to be created. This is very similar to the API Gateway from AWS where HTTP endpoints are mapped to chosen functions. IBM Cloud Functions does support the definition of an API by defining every path manually using the CLI[15]or the web interface. Another option is to use an OpenAPI specification to describe the mappings of the API. Due to the rather small amount of mappings we had to configure we chose the first option. Once an action is used in the API, it automatically becomes a web action. This is a mandatory step to create an API.

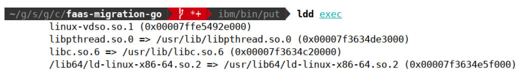
When calling an action through the defined API OpenWhisk does attach the HTTP request data, e.g., HTTP headers as a part of the input payload for the function. It also does some mapping for example HTTP query parameters get directly written into the root of the input payload. For example, a query parameter called `id` can be accessed by retrieving the `id` key from the injected JSON object. an example for the input payload is shown in Listing A.10. The HTTP protocol does not only allow developers to respond developers with a body, it also allows developers to use metadata like the status code or response headers, e.g., the content type of the body or the length of it, to enrich the response. For this the functions result must be in a certain format as seen in Listing 6.2, which comprises the definitions of the statuscode, the response content type and the body. The syntax shown here is only working in conjunction with web actions. Therefore, it is important to set the action as a web action during deployment by setting a corresponding command line flag to true. When a action is deployed as a web action OpenWhisk will expose a default endpoint to the function which is protected using access policies. To properly define the path and the methods supported an API must be created [IBM19c].

[15]Currently only works with CloudFoundry based namespaces.

Listing 6.2 Sample output (raw) JSON of the `put` function.

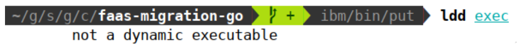
```
{
  "body": {
    "ID": "b2c10bcb83c717b69f13110ae7ba6180",
    "description": "I am a description.",
    "done": false,
    "done_timestamp": -1,
    "insertion_timestamp": 1563991481,
    "title": "Hello World"
  },
  "headers": {
    "Content-Type": "application/json"
  },
  "statuscode": 200
}
```

Deploying functions written in Go is generally quite simple. First developers compile the source code just like every other Go based application using the `go build` command. The output artifact has to be renamed to `exec` to ensure the OpenWhisk runtime knows where to find it. Afterwards the binary is packaged in a ZIP archive. In the next step a action is created using the CLI. At this stage, the function is deployed but triggering it is not yet possible.



```
~/g/s/g/c/faas-migration-go  *+  ibm/bin/put  ldd exec
        linux-vdso.so.1 (0x00007ffe5492e000)
        libpthread.so.0 => /usr/lib/libpthread.so.0 (0x00007f3634de3000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007f3634c20000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007f3634e5f000)
```

(a) `ldd` with CGO enabled



```
~/g/s/g/c/faas-migration-go  +  ibm/bin/put  ldd exec
        not a dynamic executable
```

(b) `ldd` with CGO disabled

Figure 6.1: Output of `ldd` with CGO enabled **(a)** and CGO disabled **(b)**

At this stage the action can be invoked using the CLI, theoretically. During testing we have encountered a problem, any action with the IBM Cloudant library cannot be executed on OpenWhisk because of an error. The error message stating that the executable cannot be found was difficult to debug, because the executable was clearly in the right location[16]. An attempt to deploy a simple `Hello World` function with the same commands did work, which lead us to the conclusion that the problem originates from the executable itself. While Go by default produces binaries with almost no dynamic linking some submodules still rely on dynamically linked libraries. Go by default uses a dynamically linked implementation of the `net` package used for network communication. If any library or even developer's own code uses the `net` package this dependency will occur by default [Aut13]. The resulting binary will run without any issues on most Linux based operating systems. However, some minimalistic container images like `busybox` [17] or `Alpine Linux` [18], which serves as the base image used for executing generic applications in OpenWhisk (`openwhisk/dockerskeleton` [19]), do

[16]Exact error message: 'stdout: No such file or directory: /action/exec'
[17]https://hub.docker.com/_/busybox
[18]https://hub.docker.com/_/alpine
[19]https://hub.docker.com/r/openwhisk/dockerskeleton/dockerfile

Listing 6.3 Makefile section for creating the API bindings

```
deploy_api:
  ; Create API Mappings
  ibmcloud fn api create /todo /lst get todo-lst --response-type http
  ibmcloud fn api create /todo /get get todo-get --response-type http
  ibmcloud fn api create /todo /put post todo-put --response-type http
  ibmcloud fn api create /todo /del post todo-del --response-type http
  ibmcloud fn api create /todo /done post todo-done --response-type http
  ; List all API mappings to retrieve the access links
  ibmcloud fn api list --full
```

not fulfill these dependencies. The result is a binary that does not run in the given environment[20]. In most cases the problem can be resolved by setting an environment variable, to disable the CGO[21] extension, before calling the `go build` command, which was also the case for our implementation. To find out if our implementaion in fact had dynamically linked dependencies we ran the `ldd` [22] command on the binary with CGO still enabled the output of this check can be seen in Figure 6.1a after diabling CGO all the dependencies listed have disappeared, as shown in Figure 6.1b.

With the compilation issues out of the way we can investigate the creation of the web API bindings. OpenWhisk allows the creation of an API that will map certain operations to an action, which is almost identical to defining an HTTP trigger. The definition is either done through the web interface or through the CLI. Both the web interface and the CLI allow the creation of APIs by either importing an OpenAPI specification or by defining every mapping one by one. Because this use case was implemented before working with Composer we were unaware of the issue with the incompatibility of some commands and IAM-based namespaces. Therefore our initial approach running the commands in Listing 6.3 to create the API bindings has failed due to a Authorization error. The solution at that point was the development based on the web interface by manually creating every mapping. However after stumbling accross the incompatibilities to IAM-based namespaces we tried the deployment in a CloudFoundry-based namespace and it worked.

Credentials for the IBM Cloudant database is handed over to the runtime during deployment by setting parameters. These parameters are injected into the input object. Becasue other parameters get mapped into the same JSON object it is possible to inject malicious parameters from outside. To prevent this the `final` option can be used. This will reject every incoming event attempting to override one of the predefined parameters [All17].

The creation of the IBM Cloudant instance also works either through the CLI or through the web interface. In the final implementation we automated the creation of the database using the CLI. With the help of some shell scripting and Makefiles, the creation of the service and the download of the credentials was automated. The creation of the database is done using a `create-database` action that automatically is deployed into the IBM Cloud Functions namespace when linking the

[20]Issue resolved with the help of: https://stackoverflow.com/questions/48897061/how-to-run-a-go-function-with-redis-package-on-openwhisk

[21]CGO is a Go package allowing developers to use C code from within Go. More information can be found under https://golang.org/cmd/cgo/

[22]The `ldd` command is used to print a list of shared (dynamically linked) dependencies on Linux based systems. See http://man7.org/linux/man-pages/man1/ldd.1.html for more information.

IBM Cloudant instance and the namespace. Alternative soltions we have also tried were the manual creation using the web interface and the execution of a create call within the functions themselves if the table did not exist.

While the initial implementation had many issues all of them have been resolved after getting a bit more into the cloud platform, which has severely slowed down the progression. While most of the concepts are fairly simplistic fixing the errors required most of the time especially because most of the error messages were very generic. Another problem that slowed down the progression was the unavailability of helpful documentation to fix the problems we have encountered directly. Most of the fixes were found during the implementation of other use-cases because we have consulted other documentations for them.

6.2.3 Event Processing - JavaScript

The implementation of this use case was rather straightforward in comparison with the previous ones. Just as all other programming languages we have covered in the last sections on IBM Cloud Functions a function consumes an object in JSON format serialized into a dictionary. A function also has to produce the same type of object as a result. JavaScript is no different in that respect, since a function consumes a generic JavaScript object and returns another one as an output [Fou19g].

Unlike Azure, libraries for publishing on Kafka were necessary. However there is no need to use an IBM specific one. There is also an option to publish messages using a preconfigured action comparable to the one used to create the Cloudant Database for the ToDo use-case However this implementation has been marked as deprecated and they do not offer it in every datacenter anymore [IBM19d]. It was also possible to reuse the MySQL driver we have used to interact with the MySQL database previously. The only points we had to change in order to get the functions running on OpenWhisk from the AWS codebase were 1) initializing the Kafka and MySQL client with credentials received through the input payload 2) replacing the SQS and SNS push operations with Kafka based ones 3) updating the input conversion to work properly. The implementation itself was not the biggest issue in terms of time, however, the configuration of the triggers was taking much more time mainly because the message broker had to be configured by creating access credentials and the topics. Next we had to initialize a binding between the Event Streams instance and IBM Cloud Functions. After that we were able to define the four triggers with their corresponding rules. To expose the `ingest` , `list` and `latest` functions, described in Section 3.2, we used a web API wich was created and deployed using the command line interface.

In comparison with the other two platforms the publish / subscribe trigger works differently. While a Function on AWS and Azure was getting triggered for every incoming message, this is not always the case with IBM Cloud Functions. If a function gets invoked on IBM Cloud Functionsit just gets informed that there are new messages available for processing. The function will then fetch the incoming events and process them further [Dan18a; Dan18b].

Of course, just having the actions implemented is only one portion of the migration. The next step was the creation of the services and the creation of the triggers. After the services have been created we had to create a feed on IBM Cloud Functions which connects to the created Event Streams instance. From this feed developers can create triggers for every action subscribing to a topic. In order to define the trigger the name of the corresponding topic must be passed through as a parameter. Lastly we create rules to link the created triggers with the actions [Dan18a].

Listing 6.4 Simplified workflow definition for the Matrix Multiplication on IBM Composer

```
const composer = require("openwhisk-composer");
module.exports = composer.seq(
  "create-matrix",
  composer.if(
    composer.action("choose-type", {
      action: function (data) {
        data.value = data.size >= 10;
        return data;
      }
    }),
    composer.seq(
      composer.action("set-worker-count", {}),
      "distribute-work",
      composer.parallel(
        // Parallel Code has been ommited for simplicity
      ),
      composer.action("cleanup-context", {
        action: function (data) {
          let context = data.value[0];
          return context;
        }
      }),
      "build-result"
    ),
    "serial-multiply"
  ),
  "generate-report"
);
```

6.2.4 C# - Matrix Multiplication

The function orchestrator offered by IBM is called IBM Functions Composer which is now a child project of the Apache Software Foundation. It is similar to Azure Durable Functions in many aspects for example 1) both are based around the idea of functions orchestating other functions 2) both use a programming language to model the workflows by default. However while Azure does not have any limitations regarding the size of input payloads IBM Composer does. The limit for input and output parameters is set to 5 MB [LSP+18]. Therefore a caching mechanism was required. To do this we decided to use IBMs Object Storage offering.

As shown in Listing 6.4, composed workflows are defined using JavaScript code. For the definition we start with a root sequence, which is basically a chain of actions and child sequences. Within the sequence we are able to directly invoke a, previously deployed, action by inserting the name of the action as a parameter. The position of the parameter is important because the first action will be invoked first and its output will be passed into the next. The first action also receives the unchanged version of the input payload. Using the sequence operator with multiple subactions represents the most basic function composition: a chain of functions. As you can see in Listing 6.4 we do not use an action at the second positon instead we use an if condition. This operation has to receive exactly three actions or sequences. The first parameter represents the action used to decide what branch should be invoked. This action has to set the `value` field in the return object to either `true` or `false`, as illustrated in lines 6 to 9 in Listing 6.4. Depending on this value the orchestrator will decide which branch will be executed. The action defined in the second parameter called if the condition is `true` and the one defined as the third parameter if the `value` attribute is `false`. For convenience it is also possible to define actions in line just like we do it for the action deciding to either use serial multiplication or parallel multiplication in line 6 to 9 of Listing 6.4. Of course it is only possible to define actions in JavaScript. The `parallel` feature is used to run multiple functions in parallel once all child sequences have completed this will concatenate all return objects into a array which is set to the `value` parameter again. In order to ensure the following functions work properly we define the `cleanup-context` action. It is responsible for the removal of the four (almost) identical context objects that we do not need. The only differences these return payloads have is the `worker_id` used to tell the parallel worker function which task set it should fetch from the object storage. After these workers have finished their job this value becomes irrelevant because the `build-result` action does not need this information to build the result matrix. It only needs the worker count, included in every payload, to do so.

The use of the parallel feature comes with a additional requirement. It needs a running Redis[23] instance to cache the intermediate results of the functions. Generally this is not a huge issue however configuring the credentials for Redis is very unintuitive. We did not find a viable solution other than the submission of the credentials at invokation [Fou19a].

The deployment of the compositions is done using two command line tools 1) `compose` is used to convert the workflow written in JavaScript into an intermediate, JSON based, format used for deployment 2) the `deploy` command takes this intermediate format and deploys thw workflow and the inline actions on IBM Cloud Functions. Because these commands are based upon regular OpenWhisk APIs there currently is no support for IAM-based namespaces. Requrining a CloudFoundry-based namespace. To ensure everything works properly we had to deploy every action in this namespace [IBM19b].

The changes needed in the C# source code to get everything running are fairly low. We had to implement the Caching interface again because the original one using the AWS S3 client did not work. It was also necessary to modify the vendor specific code portion to also unmarshal the given JSON into the Context object [Fou19b; San19]. Apart from these issues, the vendor specific implementation is very similiar to the one from AWS. The required efforts to migrate the source code and the workflow over from AWS is approximately identical to the efforts needed for a migration from AWS to Azure.

Tracability in Composer is better than the default approach from Azure Durable Functions but not as great as with AWS Step Functions because Compser does not offer a web interface to trace down issues. Instead developers must investigate the lists of activations using the `ibmcloud fn activation list` command. Table 6.1 shows a Simplified version of the output from such a CLI call after running a serial multiplication. Since this command lists all activations this can quickly become very confusing. But it is still possible to trace down a specific invokation. If one requests the logs of the activations of kind `sequence`. If one requests the logs of this activation using the `ibmcloud fn activation logs <activation id>` command one will get a list of all activations,

[23] https://redis.io/

[24] Irrelevant portions like dates, status and namepsace names have been removed. Activation IDs have been truncated.

Datetime	Activation ID	Kind	Start	Duration	Entity
13:46:02	`4093ca92fe...`	`nodejs:10`	warm	1ms	`workflow:0.0.1`
13:46:01	`2abf856e88...`	`dotnet:2.2`	cold	986ms	`generate-report:0.0.1`
13:45:52	`62d91267e7...`	`nodejs:10`	warm	4ms	`workflow:0.0.1`
13:45:51	`c7193a74dc...`	`dotnet:2.2`	cold	1.121s	`serial-multiply:0.0.1`
13:45:43	`69993f324b...`	`nodejs:10`	warm	3ms	`workflow:0.0.1`
13:45:42	`2f9e444d60...`	`nodejs:10`	cold	58ms	`choose-type:0.0.1`
13:45:42	`a1d9f60004...`	`nodejs:10`	warm	4ms	`workflow:0.0.1`
13:45:41	`15bf4e405b...`	`dotnet:2.2`	cold	1.167s	`create-matrix:0.0.1`
13:45:31	`9de806617b...`	`nodejs:10`	cold	218ms	`workflow:0.0.1`
13:45:31	`38b946de76...`	`sequence`	warm	30.81s	`workflow:0.0.1`

Table 6.1: Simplified output of `ibmcloud fn activation list` after running a serial multiplication[24]

by their id, that have occured for this specific activation of the worklfow allowing developers to investigate the logs of one workflow activation. The featureset descibed can further be augmented using external utilities[25] however this work did not investigate this further.

Genrally the migration of this use-case worked without any severe issues, because the documentation has actually mentioned that the OpenWhisk Composer tools actually need a configured `wsk` CLI[26]. However IBM Composer also has downsides like the lack of support for IAM-based namespaces or the occasional fail of a workflow invokation directly after deployment. This could always get resolved by retrying

6.3 Overall impressions

IBM Cloud Functions was the implementation that took the longest time when comparing the three implementations. Most time was not spent on implementing the functions– instead, most of the time was spent on debugging very rudimentary error messages to find their cause. Such problems become less time-consuming with more experience in migrating or implementing functions on IBM Cloud most of these issues will either be resolved very quickly because the solution to fix the issue is already known or it will be avoided completely. The fact that IBM is currently undergoing a migration of their documentation which resulted in Google search results that pointed to nothing while they should have pointed to documentation made this process even more time consuming. The corresponding links were found by manually searching through the IBM Cloud Functions documentation.

The CLI does a good job to automate resource and function creation when used in combination with `GNU make`. While it still comes with some inconsistencies, for example, when creating an event Streams instance developers actually have to create a `messagebus` in the CLI the reason for

[25] `https://thenewstack.io/ibm-composer-provides-way-orchestrate-multiple-serverless-functions/`
[26] `wsk` is the generic OpenWhisk CLI

this is a name change. IBM has renamed Message Bus to Event Streams[27]. During the use of Event Streams we have also encountered another inconsistency: one option to create this service is through the `ibmcloud service create messagebus` command creating an Event Streams instance as a CloudFoundry service. If IBM Cloud Functions are used in the same CloudFoundry namespace the creation of the binding is no problem but the `ibmcloud es` command used to administer the Event Streams instance does not work with this type of service. Instead the resource has to be created using the `ibmcloud resource service instance create` command. This creates the service but does not yet make it a CloudFoundry service to do this we have to create a alias to this service that will then be used to bind to IBM Cloud Functions.

The lack of local execution was not a real problem because the deployment of new actions is done very quickly, while we did not measure the time needed to create ore update an action, we never had to wait for one of these commands to complete. The tasks that take way longer to perform were the compilation and packaging. Which are also unavoidable when working in with a local OpenWhisk instance for development.

IBM Cloud Functions currently only supports the deployment of the Thumbnail Generator use-case in one region because the Object Storage trigger needed for this to function is still experimental.

[27]https://www.ibm.com/cloud/blog/ibm-message-hub-is-now-ibm-event-streams

7 Migration to self-hosted environments

Relying on cloud providers to host services is not always an option in some cases it might be necessary to have a self-maintained environment for hosting applications on-premise. With the increasing popularity of the serverless computing paradigm many self-hosted implementations have been presented.

Some of the popular implementations are:

- **OpenWhisk**: As already mentioned in Chapter 6 this platform is the open source portion of IBM Cloud Functions.
- **OpenFaaS**[1]: A serverless runtime based on Docker that can be hosted on Kubernetes or Docker Swarm. It supports any programming language with support for Linux based operating systems.
- **Kubeless**[2]: A serverless runtime designed to be Kubernetes-native. Supporting Python, Node.js, Ruby, PHP, Golang and C# (.NET) and the ability to implement runtimes for other programming languages.
- **fn Project**[3]: Advertised as a container native serverless runtime. Like OpenFaaS the fn Project also supports anything that can be executed in a Docker container.
- **Fission**[4]: A serverless framework built on top of Kubernetes with support for Python, NodeJS, Go, C#, PHP and the ability to implement custom runtimes for other programming languages.

Apart from completely independent runtimes like the ones listed above there is also **KEDA** or **K**ubernetes-based **E**vent **D**riven **A**utoscaling. A system from RedHat and Microsoft that allows to scale a container down to zero instances while they receive no invokations. At the time of writing this thesis KEDA was still considered experimental, using it in production environments is therefore not recomended. One of the applications for this concept is the execution of Azure Functions in a Kubernetes environment, which basically allows having a self-hosted Azure Functions runtime in a Kubernetes cluster[5] [Mic19i].

Migrating an application requires modifications of the source code on the one hand. On the other hand it is necessary to find replacements for the services used in the cloud environment that can be self hosted. In the next section we will investigate some pitfalls when migrating services.

[1]https://www.openfaas.com/
[2]https://kubeless.io/
[3]https://fnproject.io/
[4]https://github.com/fission/fission
[5]KEDA comes with some limitations, including: some unsupported trigger types like the Blob storage trigger [Mic19i]

7.1 Service interaction

Depending on the chosen runtime, the efforts of migrating might be considerable or, such migration might even be unfeasible.. In the best case functions work out-of-the-box without any modifications. For example, if the function does not interact with provider-specific services, e.g., from IBM Cloud or Microsoft Azure. on the other hand, the biggest problem when migrating to an on-premise environment is the coupling of the FaaS-based application with provider services and, especially, the provider-specific triggers. Migration of databases or services offering a self-hostable implementation like MySQL or Redis is relatively simple. Unfortunately, not every cloud service has a self-hosted version. Like the Azure StorageAccount. Such mandatory parts in many applications based on Azure functions are not available for self-hosting, at least not in production environments[6]. The only options are to either just run the function in a self-hosted environment but still use the StorageAccount in Azure for storage or a self hosted replacement that might not be compatible with Azure's APIs.

Looking at the services used by the four use-cases we can categorize them into three groups. The first group contains the services that can be replaced by just using the self-hosted implementation instead of the cloud provider specific one. Migration in this case is only a matter of changing endpoint and credential parameters. The only service, that fits into this category is the MySQL instance used in the event-processing use-case.
The second group comprises services like a table storage service from the ToDo API use case, which do not have a self-hosted alternative, e.g., when the migration is attempted from AWS or Microsoft Azure to a self-hosted environment. When attempting a migration from IBM there is a potential drop in replacement (CouchDB) if vendor unspecific libraries have been used. In that case this service can be put in the first group too. Otherwise, developers have to find a self-hosted replacement for the services offered by the table storage. After the alternative has been determined all interaction of the functions and the database have to be reimplemented to work with the new database. In the best case, this is done by reimplementing interfaces. However such a migration can be a huge effort especially when working with more complex function-based applications.
The third category contains all other services used e.g., object storage, publish/subscribe messaging and queue based messaging. All of these services rely on event-driven paradigm, and trigger functions if an event has occured in the service. For example, if an object was stored in the object storage. Finding a replacement for such services is more complicatied in comparison to the second category. Migration here is especcially complex because the service must notify the FaaS platform about the event that just happened. If the service does not support notifications, for certain events at all using it is practically impossible[7]. If there is support for notifications it is potentially possible to use the service either directly or through a middleware as we will see in the following secions.
While the migration to services of the first category usually take low efforts, this is completely different for the second and third category. For example, migrating to a different database system, which fits in the second category, can requrire large efforts especcially when working with more complex data models. Similiar efforts might be necessary for category three in the worst case.

[6]Microsoft offers an emulator for the storage account that runs locally but this is only suitable for development or testing.
[7]Theoretically a poll based approach can be used.

	Apache Kafka	NATS	AMQP
OpenWhisk[Fou19d]	Yes	No	No
OpenFaaS [Ope19]	Yes	No	No
Kubeless [Kub19]	Yes	Yes	No
fn Project [Pro19]	No	No	No
Fission [Fis19]	No	Yes	No
KEDA [Mic19i]	Yes	No	No

Table 7.1: A comparison of self hostable serverless platforms in terms of support for triggers from message brokers[13].

7.1.1 Migrating to a self-hosted object storage

The open source object storage MinIO[8] claims to be API compatible with Amazon S3. This is generally true however, it will not work out-of-the-box with the regular library used for interaction because the original AWS library ommits authentication when running in a AWS environment. Making a drop in replacement impossible. However the MinIO server supports notifications based on certain events occuring in the object storage. Currently MinIO supports many different messaging platforms/protocolls like AMQP[9], MQTT[10], NATS[11] or Kafka[12]. Apart from messaging MinIO can also write to several databases including Redis, MySQL and PostgreSQL. But the most interesting option is the support of webhooks. Here MinIO will send a HTTP request to a defined URL. This request includes the S3 compatbile JSON of the event that has occured[Min19]. This is perfect for self-hosted serverless environments because (almost) every self hosted FaaS platform supports a HTTP triggers. Assuming the credential issue with the S3 client can be resolved the migration of the thumbnail generator is very easy.

7.1.2 Migrating to a self-hosted messaging solution

The use of messaging in serverless applications is very interesting. However the configuration of queue- or topic-based triggers in a self hosted environment can get complicated.

There is no generic solution for messaging based triggers. Because not every self hosted platform supports every message broker or protocol in fact the support for certain message brokers is fairly limited. Many self hostable platforms just support one or two messaging protocolls as we can see in Table 7.1. The easiest solution for this issue is to find a serverless platform and a message broker that will suit the developer's requirements. However, this introduces a tight-coupling between a FaaS platform and the chosen message queue, which prevents using other messaging solutions in further modifications of the application. Another approach to resolve this issue is a custom trigger.

[8] `https://min.io/`

[9] Short for: Advanced Message Queueing Protocol for example implemented by RabbitMQ (`https://www.rabbitmq.com/`)

[10] Short for: Message Queuing Telemetry Transport a messaging protocol mostly used in IoT environments (`http://mqtt.org/`)

[11] `https://nats.io/`

[12] `https://kafka.apache.org/`

[13] OpenWhisk, OpenFaaS and Kubeless provide mechanisms to write custom triggers [Kub19; Ope19; Tho16].

This approach is supported by OpenWhisk, OpenFaaS and Kubeless. While a custom trigger solves the problem and reduces coupling the trigger has to be maintained to make sure it works with later versions of the platform. Resulting in additional efforts needed and development cost.

Essentially, it is difficult to give a general reccomendation here because the support for messaging based triggers is very limited as we can see in the compaison in Table 7.1. If we assume that the developer does not want to write custom triggers and the chosen FaaS platform supports triggers from message brokers the developers are limited to a very small set of message brokers. In some scenarios this is a problem. For example, if a developer has a queue he wants to process messages from and that queue is based on a RabbitMQ (AMQP) message broker. No matter what platform you have chosen beforehand this is impossible out of the box. 1) Changing the whole messaging system to a supported broker, 2) using a adapter program that takes messages from the RabbitMQ queue and puts them onto a queue on a supported message broker or 3) writiing a custom trigger are potential solutions. However the first one comes with potentially huge migration efforts. The second one is not generic and it violates the fundamental ideas behind FaaS because this adapter script has to be running at all times and is not hostable in a Function this option also comes with the issues we have discussed before. The third one can be an elegant solution to this issue if the platform supports the implementation of third party triggers. Otherwise a custom trigger can compared to the second option. For example you could write a custom trigger for a generic platform that calls a webhook once it retrieves a message.

7.1.3 Function orchestration

Unlike most other services functon orchestration does not fit into any of the three categories described above. Because the support for function orchestration is highly dependent on the FaaS platform. In case there is support for function orchestration, it is often handled through extensions.

The following list gives a quick overview on wheter function orchestration is supported by the platforms mentioned in the introduction of Chapter 7.

- OpenWhisk Composer supports the orchestration from a self-hosted OpenWhisk instance. The API is identical to the version used with IBM Composer.
- OpenFaaS has an extension that allow function orchestration.faas-flow[14] supports chaining, branching and parallel execution. Workflows are defined in Go code.
- Kubeless does have any extension to handle function orchestration / workflow execution–[Fon19]
- The fn project also has an extension called 'fn flow' that allows the definiton of workflows using a programming language[15].
- Fission does have support for workflow execution using an extension[16]. It therefore also supports function orchestration [Fon19]

[14]https://github.com/s8sg/faas-flow
[15]https://github.com/fnproject/flow
[16]https://docs.fission.io/workflows/

- The documentation for KEDA does not mention support for Durable Functions but also does not explicitly exclude it. A attempt to deploy the Matrix Multiplication use-case on KEDA has failed. Further investigations are necessary to determine if KEDA does support the deployment of Azure Durable Functions workflows.

Apart from direct support for function orchestration a second approach based on an external workflow engine can be chosen to manage the workflow itself. Then only the functions will be executed on the FaaS platform. Some example for such workflow engines are argo[17] a Kubernetes native workflow engine or Apache AirFlow[18].

[17] https://github.com/argoproj/argo

[18] https://airflow.apache.org/

8 Related work

Typically, the choice to migrate an application to another cloud provider is motivated by a combination of various factors. Some reasons might be better availability guarantees, cheaper and more flexible pricing, or better performance. Wang et al. [WLZ+18] compare AWS Lambda, Azure Functions and Google Cloud Functions regarding the performance of these platforms. In addition, authors investigate how the cloud providers provisions virtual machines used to execute the functions internally aa well as the isolation between multiple functions. Similar performance evaluation also exists for self hosted environments by Mohanti et al. [MPD+18] compare OpenFaaS, Fission and Kubeless because these three open source FaaS platforms all use Kubernetes to perform resource orchestration and scaling of the functions. For OpenWhisk, an interesting research paper from Shillaker investigates potential research topics for getting lower latency serverless runtime with the performance of Openwhisk taken as a reference [Shi18]. A similar investigation by Lloyd et al. is concerned with potential factors affecting performance of functions on AWS Lambda and Azure Cloud Functions [LRC+18]

Migrating from one cloud environment to another is one type of migration. But an even more important migration approach is the migration of legacy applications in a cloud environment. Frey et al. [FH11] present a model driven approach to migrate legacy applications to cloud environment.

The scheduling component in FaaS platforms can have a huge impact on the invokation latencies and cold start. This component is responsible for deciding where the incoming function request should be executed, as well as if a warm, i.e., previously used, instance could be reused. Stein has investigates different approaches for the scheduling of functions in a FaaS runtime and proposes an approach that tries to minimize the occurance of cold starts [Ste18].

Many serverless platforms will terminate a function after a predefined timeout, which makes serverless not suitable for long-running tasks. As one of the solutions, a function can be decomposed into several smaller functions that will not exceed the time limits but this cannot always be guanranteed Soltani et al. [SGZ18] propose to resolve this is the migration of a running instance that will soon be terminated into a fresh invokation of the function to extend the timeout period if necessary.

Lopez et al. compare and evaluate the three function orchestrators in depth. Authors consider many aspects including billing and performance [LSP+18]. However some of the results described by the authors are already obsolete, since these platforms are actively beeing improved by the cloud provider. For example, authors mention that IBM Composer (now an OpenWhisk project) does not support the parallel execution of functions, which is no longer true because later versions of IBM composer actually support this feature [Fou19a].

The aforementioned work by Lopez et al. [LSP+18] also compares the three offerings regarding their conformance with the serverless trilemma. The serverless trilemma defines three criteria a function orchestrator should implement to avoid certain issues [BCF+17].

Several research papers have investigate the current state, trends and research as well as applications of the serverless paradigm. Baldini et al. describe the state of serverless computing, the open challenges and trends [BCC+17]. A later work by Castro et al. from UC Berkeley and IBM investigates the same question roughly two years after [CIMS19].

While the serverless paradigm is getting more and more popular it is not a paradigm that can be used everywhere. It also has some drawbacks. A research paper from Hellerstein et al. illustrates problems that the serverless paradigm currently has. They also propose what has to be changed in the future [HFG+18]. A similar investigation from UC Berekley by Jonas et al. also illustrates the current limitations of serverless computing and things that will have to be adressed in the future [JSS+19]

A general overview of the serverless landscape is also given by the Cloud Native Computing Foundations (CNCF) serverless workgroup in their withepaper about the topic [CNC18]. Another whitepaper from Amazon Web Services gives an overview about the capabilities of AWS Lambda [Ser17].

9 Conclusion and Outlook

In this work we analyzed the portability of FaaS-based applications by manually migrating four different use cases to several cloud providers and documenting the required efforts together with encountered challenges. As a starting point we have defined four use cases with the goal to cover many concepts that are often used in serverless applications these concepts include 1) function composition and orchestration 2) Building HTTP applications 3) publish / subscribe messaging as well as queue based messaging for sending messages and for the retrieval in combination with a message queue based trigger 4) interacting with relational and NoSQL databases 5) interfacing with object storage both as an event source and a storage service we implement these four use-cases in four different programming languages namely, Java, Go, C# and JavaScript to investigate the potential challenges and migration efforts required for different programing languages and runtimes. In addition, we investigated a simple abstraction approach to reduce the efforts for a migration. This was achieved by externalizing the complete function in a separate package. This package contains the whole function with generic method headers. This core package also only uses repository interfaces it defines to interact with external resoucres. Therefore the vendor specific implementation just contains input transformation, the vendor specific implementation of the repository interfaces and the call to the core package including the initialization. Other simpler functions were just implemented directy without taking this pattern into account to illustrate the difference of efforts needed for the migration. This has proven to be a simple and effective solution to speed up the migration process because the migration of the implementation consumes significantly less time. However while making the migration of the implementation easier it still requires migrating the configuration of the function application.

More specifically, such configuration includes the functions services like a database, credentials to access the services and the triggers for the function. As a result, it is significantly harder to simplify migrating these details, since all cloud providers handle this differently. For example, to automate the process of creating a service, AWS provides CloudFormation service with a domain specific language that can be used to model cloud infrastructure on Amazon Web Services. While Microsoft Azure and IBM Cloud do offer comparable features the simplest way to automatically create services on these platforms was with the help of shell scripts using the cloud provider specific CLI utility.
The configuration of credentials for services using secrets is only required on Azure and IBM Cloud. These providers require credentials to access their services when the application was deployed in the cloud. This step was not necessary on AWS instead we had to assign IAM role permissions to the functions to access a certain service. If the function attempts to access the service from AWS Lambda no authentication was required. With one exception: a relational database based on MySQL.
The definiton of triggers was relatively inconsistent while it was done through a configuration file on AWS, On Azure the definiton of triggers was depending on the programing language used. With JavaScript the configuration was done in seperate configuration files and on C# and Java the triggers

Use Case	AWS	Azure	IBM Cloud
Thumbnail Generator	Yes	Yes	Experimental
ToDo API	Yes	Different programming language	Yes
Matrix Multiplication	Yes	Yes	Yes
Event Processing	Yes	Yes	Only with pub/sub messaging

Table 9.1: Overview of the support for a migration, with the limitations we encountered.

are defined in the functions code using annonatations on the method header and its input parameters On IBM Cloud we used th command line interface to define the triggers the rules to link them to the functions.
For the function orchestration use-case we also had to redefine the workflow since all three platforms use different approaches to implement function orchestration. With AWS Step Functions the function workflow is defined as a state machine using a YAML or JSON based domain specific language. This is different in comparison with Azure Durable Functions and IBM Cloud Functions Composer these two function orchestration solutions use programming languages to model the workflow. While IBMs offering uses JavaScript to model the workflow it is fairly similar to AWS Step Functions: the workflow gets modelled as a sequence (chain) of functions and child sequences. Composer also provides operations to get branching and parallel sequences. For comparison Azure Durable Functions defines the orchestrator as a function therefore branching and parallel operations are handled using the methods native to the programing language used to write the function (C#).

The supported set of programming languages, the supported trigger types and the services offered by the cloud provider might also intensify the efforts necessary to migrate to another platform. For example, unsupported programming languages require rewriting the functions of an application in a supported programing language. Depending on the size of the functions this can be economically unviable to perform such a migration. In such a case no precausions like the previously introduced core package does not affect the effort needed because the function has to be rewritten completely. We have encountered this problem once when attempting to migrate our ToDo API use case to Azure.

Another problem we have encountered is the lack of support for specific trigger types. Depending on the type of trigger this can make a platform unsuitable since it cannot fulfil the requirements to deploy a certain function application. We encountered this issue when attempting the implementation of the thumbnail generation use-case on IBM Cloud. Since they generally do not provide an option to trigger a function when a change occurs in the object storage. However IBM provides an experimental extension to support this feature. We had a second encounter with such an issue on IBM Cloud due to the lack of support for a pure queue based trigger. For this problem we determined that a seperate publish / subscribe topic will be a workaround for this issue. There is no general solution for this type of problems, and potential alternatives and workarounds have to be investigated in order to see if the platform is suitable for the needs.
During our migration we have not encountered the lack of a specific type of service. But just like before there is no general solution for this problem. Potential alternatives or workarounds have to be investigated. Table 9.1 gives a quick overview of what had to be changed apart form code and configuration changes to perform the migration. In Chapter 3 we describe the idea and general architecture for every use-case shown in the table.

Apart from running in public cloud environments serverless computing is also becoming more popular in private cloud and on premise environments. Therefore we have also investigated potential challenges when migrating from a cloud provider to an on-premise solution. Since the services of an application should also be hosted in an on premise environment these have to migrated as well. This migration procedure differs, depending on the usage. If the service is not used to trigger a function, the migration to a self hostable service does not differ from a regular service migration procedure. If the service triggers functions because of events that occur during the interaction with the service it must be ensured that the replacement used for the on premise implementation supports such a feature.

9.1 Outlook

The migration approaches we have taken all were completely manual requriring a comparably high migration efforts. While many things are handled differently, the core concepts like a trigger always stay the same. As we have seen previously migration on public cloud providers generaly consists of two components. The migration of the source code and the migration of the functions configuration. Since vendor lock in is a huge problem with serverless platforms such migrations should be able to be done automatically or at least with minimal user interaction. For the migration of HTTP trigger based functions this approach seems to be comparably easy because the HTTP protocol itself is not vendor specific at least from a configuration perspective. Further research could investigate the automated migration of HTTP based function applications to another cloud provider. Later iterations might cover different trigger types like the object storage or message queue triggers.
Anohter research direction could involve the avoidance of migrations by building a framework that abstracts away the vendor specific portions of the code by building another layer on top of the FaaS runtimes themselves. This approach could also include a module that will place an abstract layer on top of different vendor specific services.

Bibliography

[All17] R. Allen. *Passing secrets to your OpenWhisk action*. 2017. URL: https://akrabat.com/passing-secrets-to-your-openwhisk-action/ (cit. on p. 59).

[Aut13] T. G. Authors. *Go 1.2 Release Notes*. 2013. URL: https://golang.org/doc/go1.2 (cit. on p. 58).

[BCC+17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, P. Suter. "Serverless Computing: Current Trends and Open Problems". In: *Research Advances in Cloud Computing*. Ed. by S. Chaudhary, G. Somani, R. Buyya. Singapore: Springer Singapore, 2017, pp. 1–20. ISBN: 978-981-10-5026-8. DOI: 10.1007/978-981-10-5026-8_1. URL: https://doi.org/10.1007/978-981-10-5026-8_1 (cit. on pp. 24, 72).

[BCF+17] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, O. Tardieu. "The serverless trilemma: function composition for serverless computing". In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*. ACM Press, 2017. DOI: 10.1145/3133850.3133855 (cit. on pp. 21, 37, 49, 71).

[Cha18] D. Chamberlain. *Containers vs. Virtual Machines (VMs): What's the Difference?* 2018. URL: https://blog.netapp.com/blogs/containers-vs-vms/ (cit. on p. 19).

[CIMS19] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski. "The server is dead, long live the server: Rise of Serverless Computing, Overview of Current State and Future Trends in Research and Industry". In: *arXiv preprint arXiv:1906.02888* (2019) (cit. on p. 72).

[Cla17] T. Claburn. *'Lambda and serverless is one of the worst forms of proprietary lock-in we've ever seen in the history of humanity'*. 2017. URL: https://www.theregister.co.uk/2017/11/06/coreos_kubernetes_v_world/ (cit. on p. 16).

[CNC18] CNCF. "CNCF WG-Serverless Whitepaper v1.0". In: (2018). URL: https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf (cit. on pp. 18, 72).

[Dan18a] J. T. Daniel Krook Olaph Wagoner. *Serverless reference architecture for IBM Event Streams data processing with IBM Cloud Functions*. 2018. URL: https://github.com/IBM/ibm-cloud-functions-refarch-data-processing-message-hub (cit. on p. 60).

[Dan18b] O. W. Daniel Krook. *Deploy serverless functions that respond to messages and handle streams*. 2018. URL: https://developer.ibm.com/patterns/serverless-event-stream-processing/ (cit. on p. 60).

[Dau19] H. Daur. *IBM Cloud Functions is now enabled for Identity and Access Management (IAM), the mechanism used across IBM Cloud to control access to resources by users or applications.* 2019. URL: https://www.ibm.com/cloud/blog/ibm-cloud-functions-is-now-identity-and-access-management-enabled (cit. on p. 55).

[Des17] P. Desai. *Whisk Deploy - Action, Trigger, and Rule*. 2017. URL: https://medium.com/openwhisk/whisk-deploy-action-trigger-and-rule-eed93becbc16 (cit. on p. 53).

[Deu19] M. Deuser. *Listening for changes to a bucket by using the (Experimental) Object Storage events source*. 2019. URL: https://www.ibm.com/cloud/blog/announcements/ibm-cloud-functions-adds-support-for-cloud-object-storage-triggers (cit. on p. 56).

[Eld18] I. Eldridge. *What Is Container Orchestration?* 2018. URL: https://blog.newrelic.com/engineering/container-orchestration-explained/ (cit. on p. 19).

[Erw18] B. L. R. Erwan Alliaume. *Cold start / Warm start with AWS Lambda*. 2018. URL: https://blog.octo.com/en/cold-start-warm-start-with-aws-lambda/ (cit. on p. 20).

[FH11] S. Frey, W. Hasselbring. "The cloudmig approach: Model-based migration of software systems to cloud-optimized applications". In: *International Journal on Advances in Software* 4.3 and 4 (2011), pp. 342–353 (cit. on p. 71).

[FIMS17] G. C. Fox, V. Ishakian, V. Muthusamy, A. Slominski. "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research". In: *CoRR* abs/1708.08028 (2017). arXiv: 1708.08028. URL: http://arxiv.org/abs/1708.08028 (cit. on p. 17).

[Fis19] Fission. *Triggers*. 2019. URL: https://docs.fission.io/usage/trigger/ (cit. on p. 67).

[Fon19] N. Fonseka. *Kubeless vs Fission: The Kubernetes Serverless match up*. 2019. URL: https://medium.com/@natefonseka/kubeless-vs-fission-the-kubernetes-serverless-match-up-41f66611f54d (cit. on p. 68).

[Fou19a] A. S. Foundation. *Apache OpenWhisk Composer - Source Code*. 2019. URL: https://github.com/apache/incubator-openwhisk-composer (cit. on pp. 62, 71).

[Fou19b] A. S. Foundation. *Apache OpenWhisk runtimes for .NET Core*. 2019. URL: https://github.com/apache/incubator-openwhisk-runtime-dotnet (cit. on p. 62).

[Fou19c] A. S. Foundation. *Apache OpenWhisk runtimes for java*. 2019. URL: https://github.com/apache/incubator-openwhisk-runtime-java (cit. on p. 56).

[Fou19d] A. S. Foundation. *Creating triggers and rules*. 2019. URL: https://github.com/apache/openwhisk/blob/master/docs/triggers_rules.md#creating-triggers-and-rules (cit. on p. 67).

[Fou19e] A. S. Foundation. *OpenWhisk - Actions*. 2019. URL: https://github.com/apache/openwhisk/blob/master/docs/actions.md (cit. on p. 55).

[Fou19f] A. S. Foundation. *OpenWhisk Programming Model*. 2019. URL: https://openwhisk.apache.org/documentation.html (cit. on p. 53).

[Fou19g] A. S. Foundation. *OpenWhisk samples*. 2019. URL: https://github.com/apache/incubator-openwhisk/blob/master/docs/samples.md (cit. on p. 60).

[Fra17a] S. Framework. *DynamoDB and Serverless*. 2017. URL: https://serverless.com/dynamodb/ (cit. on p. 35).

[Fra17b] S. Framework. *Serverless REST API*. 2017. URL: https://github.com/serverless/examples/tree/master/aws-node-rest-api-with-dynamodb (cit. on p. 35).

[Fra18a] S. Framework. *AWS - Create*. 2018. URL: https://serverless.com/framework/docs/providers/aws/cli-reference/create/ (cit. on p. 32).

[Fra18b] S. Framework. *IAM*. 2018. URL: https://serverless.com/framework/docs/providers/aws/guide/iam/ (cit. on p. 32).

[Fra19a] S. Framework. *AWS - Invoke Local*. 2019. URL: https://serverless.com/framework/docs/providers/aws/cli-reference/invoke-local/ (cit. on p. 39).

[Fra19b] S. Framework. *Serverless Framework - AWS Provider Documentation*. 2019. URL: https://serverless.com/framework/docs/providers/aws/ (cit. on p. 30).

[Fra19c] S. Framework. *SNS*. 2019. URL: https://serverless.com/framework/docs/providers/aws/events/sns/ (cit. on p. 36).

[Fra19d] S. Framework. *SQS Queues*. 2019. URL: https://serverless.com/framework/docs/providers/aws/events/sqs/ (cit. on p. 36).

[Fro12] K. Fromm. *Why The Future Of Software And Apps Is Serverless*. 2012. URL: https://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless/ (cit. on p. 15).

[Hec17] L. Hecht. *AWS Lambda Still Towers Over the Competition, but for How Much Longer?* 2017. URL: https://thenewstack.io/aws-lambda-still-towers-competition-much-longer/ (cit. on p. 29).

[HFG+18] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, C. Wu. “Serverless computing: One step forward, two steps back”. In: *arXiv preprint arXiv:1812.03651* (2018) (cit. on p. 72).

[Hor19] T. Horike. *How to manage your AWS Step Functions with Serverless*. 2019. URL: https://serverless.com/blog/how-to-manage-your-aws-step-functions-with-serverless/l (cit. on p. 39).

[HW12] G. Hohpe, B. Woolf. *Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions*. 1. Aufl. Amsterdam: Addison-Wesley, 2012. ISBN: 978-0-133-06510-7 (cit. on p. 26).

[IBM19a] IBM. *About the IBM Cloud Object Storage S3 API*. 2019. URL: https://cloud.ibm.com/docs/services/cloud-object-storage?topic=cloud-object-storage-compatibility-api (cit. on p. 54).

[IBM19b] IBM. *Configuring and running compositions in IBM Cloud Functions*. 2019. URL: https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-pkg_composer (cit. on pp. 55, 62).

[IBM19c] IBM. *Creating web actions*. 2019. URL: https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-actions_web (cit. on p. 57).

[IBM19d] IBM. *Event Streams*. 2019. URL: https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-pkg_event_streams (cit. on p. 60).

[IBM19e] IBM. *IBM Cloud Functions - Introduction*. 2019. URL: https://cloud.ibm.com/openwhisk/ (cit. on p. 53).

[IBM19f] IBM. *Listening for changes to a bucket by using the (Experimental) Object Storage events source*. 2019. URL: https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-pkg_obstorage#pkg_obstorage_ev (cit. on p. 56).

[IBM19g] IBM. *Managing namespaces*. 2019. URL: https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-namespaces (cit. on p. 55).

[JSS+19] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. “Cloud programming simplified: a berkeley view on serverless computing”. In: *arXiv preprint arXiv:1902.03383* (2019) (cit. on p. 72).

[Kos18] A. Koskela. *How to access Azure Function App’s settings from C#?* 2018. URL: https://www.koskila.net/how-to-access-azure-function-apps-settings-from-c/ (cit. on p. 47).

[Kub19] Kubeless. *How to add a new event source as Trigger*. 2019. URL: https://kubeless.io/docs/implementing-new-trigger (cit. on p. 67).

[LRC+18] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, S. Pallickara. “Serverless computing: An investigation of factors influencing microservice performance”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2018, pp. 159–169 (cit. on p. 71).

[LSP+18] P. G. López, M. Sánchez-Artigas, G. París, D. B. Pons, Á. R. Ollobarren, D. A. Pinto. “Comparison of Production Serverless Function Orchestration Systems”. In: (July 30, 2018). arXiv: http://arxiv.org/abs/1807.11248v1 [cs.DC] (cit. on pp. 49, 61, 71).

[Man14] P. Mandl. “Betriebssystemvirtualisierung”. In: *Grundkurs Betriebssysteme: Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation, Virtualisierung*. Wiesbaden: Springer Fachmedien Wiesbaden, 2014, pp. 297–318. ISBN: 978-3-658-06218-7. DOI: 10.1007/978-3-658-06218-7_9. URL: https://doi.org/10.1007/978-3-658-06218-7_9 (cit. on p. 19).

[McK16] J. McKim. *Abstracting the Back-end with FaaS*. 2016. URL: https://serverless.zone/abstracting-the-back-end-with-faas-e5e80e837362 (cit. on p. 18).

[Mic18a] Microsoft. *Azure Function Host - Wiki - function.json*. 2018. URL: https://github.com/Azure/azure-functions-host/wiki/function.json (cit. on p. 47).

[Mic18b] Microsoft. *Azure Service Bus bindings for Azure Functions*. 2018. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-service-bus (cit. on p. 47).

[Mic18c] Microsoft. *Azure Table storage bindings for Azure Functions*. 2018. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-table (cit. on p. 46).

[Mic18d] Microsoft. *Durable Functions patterns and technical concepts (Azure Functions)*. 2018. URL: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-concepts (cit. on p. 49).

[Mic18e] Microsoft. *Durable Functions types and features (Azure Functions)*. 2018. URL: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-types-features-overview (cit. on pp. 49, 50).

[Mic18f] Microsoft. *What are Durable Functions?* 2018. URL: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview (cit. on p. 49).

[Mic19a] Microsoft. *Azure Functions JavaScript developer guide*. 2019. URL: https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/azure-functions/functions-reference-node.md (cit. on pp. 47, 48).

[Mic19b] Microsoft. *Azure Functions triggers and bindings concepts*. 2019. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings (cit. on p. 41).

[Mic19c] Microsoft. *Create your first function with Java and Maven*. 2019. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-java-maven (cit. on p. 41).

[Mic19d] Microsoft. *Global data distribution with Azure Cosmos DB - under the hood*. 2019. URL: https://docs.microsoft.com/en-us/azure/cosmos-db/global-dist-under-the-hood (cit. on p. 42).

[Mic19e] Microsoft. *host.json reference for Azure Functions 2.x*. 2019. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json (cit. on p. 43).

[Mic19f] Microsoft. *Supported languages in Azure Functions*. 2019. URL: https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages (cit. on p. 41).

[Mic19g] Microsoft. *The Task asynchronous programming model in C#*. 2019. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/ (cit. on p. 49).

[Mic19h] Microsoft. *Work with Azure Functions Core Tools*. 2019. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local (cit. on pp. 43, 44).

[Mic19i] R. Microsoft. *KEDA - Kubernetes-based Event Driven Autoscaling*. 2019. URL: https://github.com/kedacore/keda (cit. on pp. 65, 67).

[Min19] MinIO. *MinIO Bucket Notification Guide*. 2019. URL: https://docs.min.io/docs/minio-bucket-notification-guide.html (cit. on p. 67).

[MPD+18] S. K. Mohanty, G. Premsankar, M. Di Francesco, et al. “An Evaluation of Open Source Serverless Computing Frameworks.” In: *CloudCom*. 2018, pp. 115–120 (cit. on p. 71).

[Ope19] OpenFaaS. *Triggers*. 2019. URL: https://docs.openfaas.com/reference/triggers/ (cit. on p. 67).

[Pfe18] M. Pfeiffer. *Build Your First Serverless Web Application on Azure*. 2018. URL: https://mikepfeiffer.io/azure-serverless-101.html (cit. on p. 41).

[Pro19] F. Project. *Trigger*. 2019. URL: https://github.com/fnproject/docs/blob/master/fn/develop/triggers.md (cit. on p. 67).

[Rab17] R. Rabbah. *Understanding and using Docker actions in IBM Bluemix OpenWhisk*. 2017. URL: https://www.ibm.com/cloud/blog/docker-bluemix-openwhisk (cit. on p. 53).

[Rab18] R. Rabbah. *Running Go Programs as IBM Cloud Functions*. 2018. URL: https://www.ibm.com/cloud/blog/running-existing-app-ibm-cloud-functions (cit. on p. 53).

[Rob18] M. Roberts. *Serverless architectures*. 2018. URL: https://martinfowler.com/articles/serverless.html (cit. on p. 18).

[San19] C. Santana. *What’s happening with IBM Cloud Functions and .NET Core?* 2019. URL: https://www.ibm.com/cloud/blog/announcements/ibm-cloud-functions-adds-support-for-net-core-2-2 (cit. on p. 62).

[Ser17] A. W. Services. *Serverless Architectureswith AWS Lambda*. 2017. URL: https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf (cit. on p. 72).

[Ser18a] A. W. Services. *Tutorial: Using AWS Lambda with Amazon S3*. 2018. URL: https://docs.aws.amazon.com/lambda/latest/dg/with-s3-example.html (cit. on p. 23).

[Ser18b] A. W. Services. *Using AWS Lambda with Other Services*. 2018. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html (cit. on p. 31).

[Ser19a] A. W. Services. *Amazon Resource Names (ARNs) and AWS Service Namespaces*. 2019. URL: https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html (cit. on p. 36).

[Ser19b] A. W. Services. *AWS Lambda FAQs*. 2019. URL: https://aws.amazon.com/lambda/faqs/ (cit. on p. 29).

[Ser19c] A. W. Services. *AWS Lambda Function Handler in Go*. 2019. URL: https://docs.aws.amazon.com/lambda/latest/dg/go-programming-model-handler-types.html (cit. on p. 34).

[Ser19d] A. W. Services. *AWS Lambda Runtimes*. 2019. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html (cit. on p. 33).

[Ser19e] A. W. Services. *AWS Step Function Documentation - States*. 2019. URL: https://docs.aws.amazon.com/step-functions/latest/dg/concepts-states.html (cit. on pp. 37, 38).

[Ser19f] A. W. Services. *aws-lambda-go Source Code*. 2019. URL: https://github.com/aws/aws-lambda-go (cit. on pp. 34, 35).

[Ser19g] A. W. Services. *Event Message Structure*. 2019. URL: https://docs.aws.amazon.com/AmazonS3/latest/dev/notification-content-structure.html (cit. on p. 46).

[SGZ18] B. Soltani, A. Ghenai, N. Zeghib. “A Migration-based Approach to execute Long-Duration Multi-Cloud Serverless Functions.” In: *ICAASE*. 2018, pp. 42–50 (cit. on p. 71).

[Shi18] S. Shillaker. “A provider-friendly serverless framework for latency-critical applications”. In: *12th Eurosys Doctoral Workshop, Porto, Portugal*. 2018 (cit. on p. 71).

[Ste18] M. Stein. “The Serverless Scheduling Problem and NOAH”. In: *CoRR* abs/1809.06100 (2018). arXiv: 1809.06100. URL: http://arxiv.org/abs/1809.06100 (cit. on p. 71).

[Tho16] J. Thomas. *OpenWhisk and MQTT*. 2016. URL: http://jamesthom.as/blog/2016/06/15/openwhisk-and-mqtt/ (cit. on pp. 53, 67).

[VTT+18] E. Van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, A. Iosup. “Serverless is more: From paas to present cloud computing”. In: *IEEE Internet Computing* 22.5 (2018), pp. 8–17 (cit. on pp. 17, 18).

[WLZ+18] L. Wang, M. Li, Y. Zhang, T. Ristenpart, M. Swift. “Peeking Behind the Curtains of Serverless Platforms”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 133–146. ISBN: 978-1-931971-44-7. URL: https://www.usenix.org/conference/atc18/presentation/wang-liang (cit. on pp. 19, 71).

[Zha18] D. Zhang. *Serverless & RDBS (Part 1) - Set up AWS RDS Aurora and Lambda with serverless*. 2018. URL: https://medium.com/mos-engineering/serverless-rdbs-part-1-set-up-aws-rds-aurora-and-lambda-with-serverless-4c2a5146faf4 (cit. on pp. 32, 36).

All links were last followed on August 5, 2019.

A Appendix

Listing A.1 A example of an event payload for the thumbnail generator function

```
{
  "Records": [
    {
      "eventVersion": "2.1",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "2019-02-11T20:09:49.080Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "A1P82TG7QDFIO2"
      },
      "requestParameters": {
        "sourceIPAddress": "11.22.33.44"
      },
      "responseElements": {
        "x-amz-request-id": "6132F52CB439251C",
        "x-amz-id-2": "DBweqX5cun/rNFiyOw8u+lwSWzgY3pZAnH2o3UeMwVi/
sZNyMejMEC4DEuJoQ6qzStwEYqN6cY8="
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "2c3d3f4d-0243-46e0-b7f3-e4320f027256",
        "bucket": {
          "name": "cmueller-tgen-images",
          "ownerIdentity": {
            "principalId": "A1P82TG7QDFIO2"
          },
          "arn": "arn:aws:s3:::cmueller-tgen-images"
        },
        "object": {
          "key": "rndaj",
          "size": 102400,
          "eTag": "23d761cbed8ad025b5c36f085a1ddf21",
          "sequencer": "005C61D68D0F53063D"
        }
      }
    }
  ]
}
```

Listing A.2 Global IAM role definition for the event-processing use-case in the `serverless.yml`

```
  iamRoleStatements:
  - Effect: "Allow"
    Resource: "*"
    Action:
      - "sns:*"
      - "sqs:*"

```

Listing A.3 JavaScript code for a `format` function function in the event-processing use-case on AWS Lambda

```
module.exports.formatTemperatureEvent = async (event) => {
  let tempEvent = JSON.parse(event.Records[0].Sns.Message);
  let message = "Message here";
  let evt = {
    type: tempEvent.type,
    source: tempEvent.source,
    timestamp: tempEvent.timestamp,
    formatting_timestamp: getUnixTime(),
    message: message
  }
  let messageString = JSON.stringify(evt);
  var params = {
    MessageBody: messageString,
    QueueUrl: getQueueURL()
  };
  sqs.sendMessage(params, function(err, data) {
    if(err != null) {
      console.log(err);
    }
    console.log(JSON.stringify(data));
  });
  return {};
};
```

Listing A.4 Non vendor spectific JavaScript code of a 'format' function in the event-processing use-case

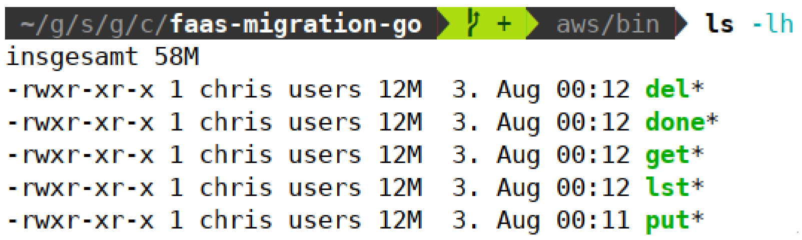
```
  let message = "Message here";
  let evt = {
    type: tempEvent.type,
    source: tempEvent.source,
    timestamp: tempEvent.timestamp,
    formatting_timestamp: getUnixTime(),
    message: message
  }
  let messageString = JSON.stringify(evt);
```

Listing A.5 Simplified version of the Matrix Multiplication state machine modeled in the Step Functions

```
StartAt: CreateMatrix
States:
  CreateMatrix:
    Type: Task
    Resource: "<Create Matrix ARN>"
    TimeoutSeconds: 60
    Next: ChooseVariant
  ChooseVariant:
    Type: Choice
    Choices:
      - Variable: $.MatrixSize
        NumericGreaterThanEquals: 10
        Next: AppendWorkerCount
      - Variable: $.MatrixSize
        NumericLessThan: 10
        Next: SerialMul
    InputPath: $
    OutputPath: $
  SerialMul:
    Type: Task
    Resource: "<Serial Multiplication ARN>"
    Comment: Serial Multiplication Handler
    InputPath: $
    OutputPath: $
    TimeoutSeconds: 300
    Next: GenReport
  GenReport:
    Type: Task
    Resource: "<Generate Report ARN>"
    TimeoutSeconds: 60
    End: true
```

Listing A.6 Vendor spectific Java code of the upload function for the thumbnail generator

```java
@FunctionName("Upload-Image")
@StorageAccount(Config.STORAGE_ACCOUNT_NAME)
public HttpResponseMessage upload(
        @HttpTrigger(name = "req", methods = {HttpMethod.POST}, authLevel = AuthorizationLevel
.ANONYMOUS)
                HttpRequestMessage<String> request,
        @BindingName("name") String fileName,
        @BlobOutput(name = "out", path = "input/{name}")
                OutputBinding<byte[]> blobOutput,
) {
    byte[] data = Base64.getDecoder().decode(request.getBody());
    blobOutput.setValue(data);
    return request.createResponseBuilder(HttpStatus.OK).build();
}
```



```
~/g/s/g/c/faas-migration-go  + aws/bin  ls -lh
insgesamt 58M
-rwxr-xr-x 1 chris users 12M  3. Aug 00:12 del*
-rwxr-xr-x 1 chris users 12M  3. Aug 00:12 done*
-rwxr-xr-x 1 chris users 12M  3. Aug 00:12 get*
-rwxr-xr-x 1 chris users 12M  3. Aug 00:12 lst*
-rwxr-xr-x 1 chris users 12M  3. Aug 00:11 put*
```

Figure A.1: Terminal output of the `ls -lh` command in the binries directory of the ToDo API use-case implemented on AWS

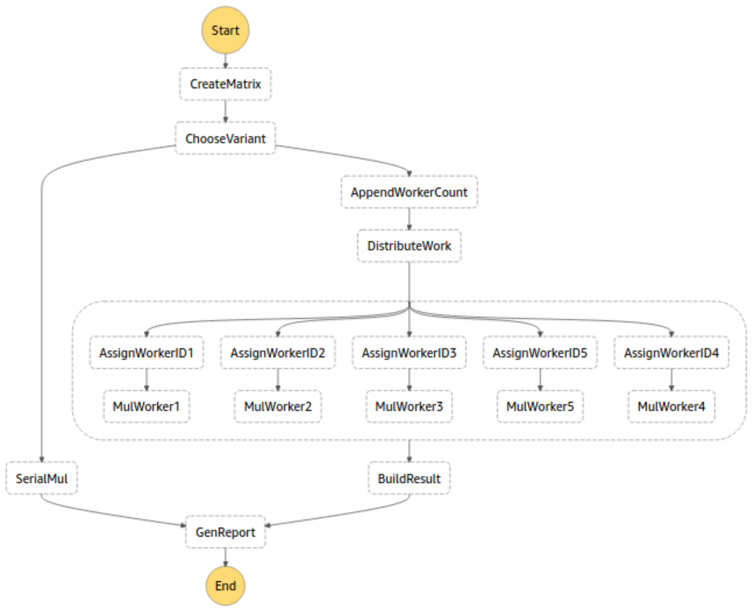
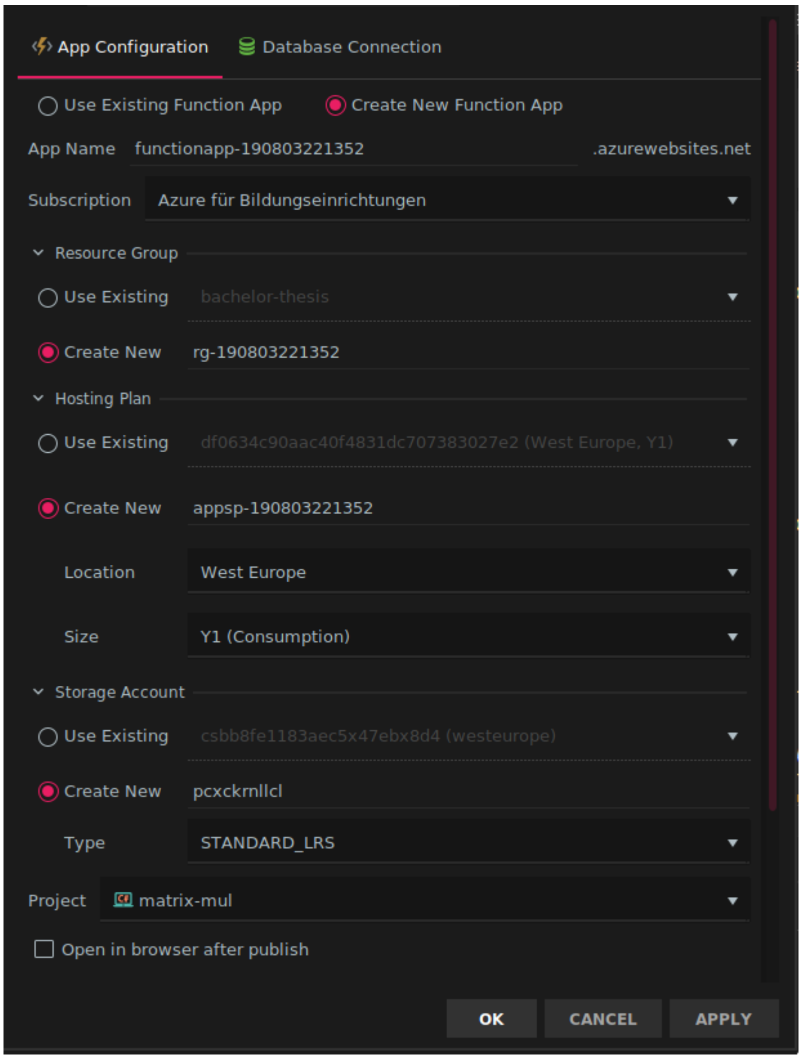


Figure A.2: Visual representation of the AWS Step Functions state machine that implements the matrix multiplication

Listing A.7 `function.json` for the `list` function of the event processing use-case.

```
{
  "bindings": [
    {
      "authLevel": "anonymous",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": ["get"]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "res"
    }
  ]
}
```



Use Existing Function App [✓ Create New Function App]

App Name functionapp-190803221352 .azurewebsites.net

Subscription Azure für Bildungseinrichtungen

Resource Group

Use Existing bachelor-thesis

[✓ Create New] rg-190803221352

Hosting Plan

Use Existing df0634c90aac40f4831dc707383027e2 (West Europe, Y1)

[✓ Create New] appsp-190803221352

Location West Europe

Size Y1 (Consumption)

Storage Account

Use Existing csbb8fe1183aec5x47ebx8d4 (westeurope)

[✓ Create New] pcxckrnllcl

Type STANDARD_LRS

Project matrix-mul

☐ Open in browser after publish

Figure A.3: Deployment dialog for a Function App in JetBrains Rider

Listing A.8 Client function for the matrix multiplication workflow on Azure Durable Functions

```
[FunctionName("TriggerMatrixMultiplication")]
public static async Task<HttpResponseMessage> StartMultiplication(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get")]
    HttpRequestMessage msg,
    HttpRequest req,
    [OrchestrationClient] DurableOrchestrationClient starter,
){
    var ms = 125;
    if (req.Query.ContainsKey("size")) {
        try {
            ms = int.Parse(req.Query["size"]);
        } catch (Exception) {}
    }
    // Function input comes from the request content.
    string iid = await starter.StartNewAsync("OrchestrateMatrixMultiplication", ms.ToString())
;
    return starter.CreateCheckStatusResponse(msg, iid);
}
```

Listing A.9 Activity function (`GenerateMatrix`) for the matrix multiplication workflow on Azure Durable Functions

```
[FunctionName("GenerateMatrix")]
public static MatrixCalculation GenerateMatrix([ActivityTrigger] string size)
{
    var s = int.Parse(size);
    var repo = new InMemoryMatrixMulRepository();
    var hndlr = new FunctionHandler(repo);
    var id = hndlr.CreateMatrix(s, 500);
    return repo.GetCalculation(id);
}
```

Listing A.10 Sample input JSON of an OpenWhisk based API

```
{
  "__ow_headers": {
    "accept": "OMMITED",
    "accept-encoding": "gzip",
    "accept-language": "de-DE, de;q=0.9, en-US;q=0.8, en;q=0.7",
    "cdn-loop": "cloudflare",
    "cf-connecting-ip": "149.81.75.151",
    "cf-ipcountry": "US",
    "cf-ray": "4f421c3cba4b9754-FRA",
    "cf-visitor": "{\"scheme\":\"https\"}",
    "host": "eu-de.functions.cloud.ibm.com",
    "upgrade-insecure-requests": "1",
    "user-agent": "OMMITED",
    "x-forwarded-for": "OMMITED",
    "x-forwarded-host": "eu-de.functions.cloud.ibm.com",
    "x-forwarded-port": "443",
    "x-forwarded-prefix": "/gws/apigateway",
    "x-forwarded-proto": "https",
    "x-forwarded-url": "OMMITED?id=someid",
    "x-global-k8fdic-transaction-id": "d81e936d093212a55399bd8accb9696a",
    "x-real-ip": "162.158.93.17",
    "x-request-id": "d81e936d093212a55399bd8accb9696a",
    "x-require-whisk-auth": "662af5f3-03fe-4283-9d7b-a85566b594f1"
  },
  "__ow_method": "get",
  "__ow_path": "",
  "id": "someid"
}
```

Listing A.11 A example of the payload the action will receive when triggered by an object storage trigger.

```
{
  "bucket": "tgen-input-uss",
  "endpoint": "s3.us-south.cloud-object-storage.appdomain.cloud",
  "file": {
    "ETag": "\"4acc7a5e0b3551f910f0b81866df2f1d\"",
    "Key": "1.bin",
    "LastModified": "2019-07-25T15:13:19.573Z",
    "Owner": {
      "DisplayName": "eabdd2b3-6a5f-4cea-b6ac-61ed7e36a19a",
      "ID": "eabdd2b3-6a5f-4cea-b6ac-61ed7e36a19a"
    },
    "Size": 1024000,
    "StorageClass": "STANDARD"
  },
  "key": "1.bin",
  "status": "added"
}
```

B Notices

B.1 Implementation Source Code

All implementations done in this thesis are available on GitHub. All implementations done in Go can be located in `https://github.com/c-mueller/faas-migration-go` and everything else can be located in `https://github.com/c-mueller/faas-migration`.

B.2 Development Environment

Because the development experience might differ for every development environment I want to describe my development environment used for developing the functions in this section.

In general a Linux based system (Arch Linux) was used. The version of the runtimes used was kept up to date. The last check of the functionality of the functions has happened in the following versions:

- **.NET Framework Core** Version 2.2
- **Node.js** Version 8.16
- **Java** Version 8 (OpenJDK)
- **Golang** Version 1.12

For development the following IDEs have been used:

- **.NET Framework Core** JetBrains Rider - Version 2019.1
- **Node.js** VisualStudio Code - Version 1.34
- **Java** JetBrains IntelliJ IDEA Ultimate - Version 2019.1
- **Golang** JetBrains GoLand - Version 2019.1

The cloud provider specific CLIs had the following versions:

- **Serverless CLI**: Version 1.42.3
- **AWS CLI**: Version 1.16.203
- **Azure CLI**: Version 2.0.65
- **Azure Functions Core Tools**: Version 2.7.1373
- **IBM Cloud CLI**: Version 0.17.0. With the following Versions of extensions:
  - **cloud-databases**: Version 0.6.0
  - **cloud-functions**: Version 1.0.32
  - **cloud-object-storage**: Version 1.1.0
  - **event-streams**: Version 1.0.1

For automation the following tools were used:

- **GNU Make**: Version 4.2.1
- **BASH**: Version 5.0.7
- **jq**: Version 1.6
- **sponge**: `moreutils` Version 0.63

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature