Utilising machine and transfer learning for an effective FaaS implementation on the edge-cloud continuum

Thomas Adam Coward

BSc. (Hons) Computer Science
School of Computing and Communications
Lancaster University

March 24, 2023

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation.

Regarding the electronically submitted work, I consent to this being stored electronically and copied for assessment purposes, including the School's use of plagiarism detection systems in order to check the integrity of assessed work.

I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Name: Thomas Adam Coward
Date: March 24, 2023

Abstract

Function-as-a-Service (FaaS), following its initial widescale release as AWS Lambda, has rapidly gained popularity in recent years as a quick and easy model for application developers to deploy their applications upon, taking advantage of other recent developments such as event driven architectures and cloud computing. This rise coincides with the equally promiment rise in proliferation of IoT and other edge devices, many of which are heavily reliant on real-time data and increasingly making use of intense AI and analysis workloads currently predominantly exclusively executed in the edge or cloud. This project aims to explore how a FaaS architecture, currently primarily adopted in relatively homogeneous cloud environments, can be deployed across a heterogeneous network consisting of both edge and cloud infrastructure, by dynamically orchestrating requests between the edge and cloud with the help of machine and transfer learning. The project's results show promise in the field and present some interesting findings regarding the handling of data in environments as dynamic as an edge-cloud FaaS infrastructure.

Contents

List of Figures

Chapter 1

Introduction

This chapter will introduce the background of the project and FaaS, before laying-out the high level aims of the project.

1.1 Background

Function-as-a-Service (FaaS) has been rapidly growing in prominence in recent years as a method of deploying applications to be highly-available, elastic/scalable and efficient with low deployment and development costs [1].

FaaS' fundamental premise is to abstract many low-level details surrounding the underlying infrastructure that application code is executed on by providing simple APIs to quickly deploy applications formed of many standalone single-purpose stateless "functions". These functions are placed within an event-driven architecture (events are often HTTP requests, but could also be published by another function to form a workflow of several chained functions) and allocated resources only as they're requested.

The scale-to-zero [2] ability of FaaS environments means that resources are only assigned to functions when they're actually invoked, with the application developer only incurring costs as such.

These features of FaaS allow for application developers to focus on business logic as opposed to managing infrastructure, whilst also minimising monetary running costs and realising improved efficiency. The fiscal benefits of adopting FaaS could be considered twofold by application developers, with the combined cost reductions of diminished development and maintenance time and efficiency savings at runtime.

This application architecture is in contrast to other "as-a-service" abstractions [3] also offered by cloud providers, which nowadays with the mass adoption of virtualisation and cloud computing are commonly defined as:

- infrastructure-as-a-service (IaaS), which provides virtual machines upon which to run applications, but requires the application developer to install and manage supporting software (for instance, a web server or even an OS) themselves;
- container-as-a-service (CaaS), which provides containers (for instance, Docker containers) upon which to run applications in;
- platform-as-a-service (PaaS), which provides a platform upon which application code can be deployed to (the key difference to FaaS being that the application is constantly running).

Each of these offerings, culminating with FaaS, offer gradually higher level abstractions over the complexities and therefore costs of running operating environments for applications.

Commonly cited advantages of FaaS over such alternative offerings include lower costs, with ”pay-as-you-go” pricing models from cloud providers [4] offering sometimes drastically lower costs than provisioning infrastructure to constantly run and acquire costs, increased elasticity, with individual functions representing aspects of the application able to be scaled automatically independently of each-other (as opposed to an entire application being scaled), and an improved developer experience, with no need to manage infrastructure and there being a straightforward deployment process involving minimal configuration.

FaaS lies within the growing serverless computing space, and is most commonly offered by cloud providers to allow for fully managed infrastructure to be offered. Given its many advantages to alternative cloud offerings, FaaS is becoming an increasingly popular model across a range of infrastructures [5], with [6] claiming that ”serverless [of which FaaS is the primary offering] will dominate the future of cloud computing”.

Despite the wide adoption of FaaS, there remain some challenges that have prevented the adoption of FaaS, as well as cloud computing more broadly, in some use cases. Notably, latency is a challenge [7] currently being attempted to be addressed through efficiency optimisations; FaaS is susceptible to higher latency owing to it being sensitive to network latency (having many interdependent components networking with each-other) and function invocations requiring complex scheduling across their shared resources. Higher latency is of course undesirable for any application, however it is a particularly critical issue for latency sensitive real-time applications which are becoming increasingly common largely due to the ongoing rise in pervasiveness of IoT devices [8]. IoT, despite being such a fast growing domain, is reported to have constituted around only 10% of serverless technology usage by 2020 [9], indicating that there are barriers to its adoption for IoT use cases in particular.

One potential method of reducing the higher latency encountered by FaaS is to utilise edge computing. At the edge FaaS brings similar benefits to those enjoyed in the cloud, albeit while also introducing some new challenges [8] such as a lack of managed infrastructure, generally less available computing resource and added complexity given the highly-heterogeneous nature (in terms of hardware and software) of edge infrastructure, also bringing reduced elasticity. However, edge brings a key benefit in that owing to its closer geographical proximity to end user clients, it enjoys lower network latency.

Some of these challenges specific to FaaS deployed at the edge can be, and have been, negated by operating a FaaS platform across a cloud and edge environment simultaneously (sometimes referred to as the "edge-cloud continuum") [10]. This solution considers available resources across cloud and edge environments when invoking functions in order to take advantage of the strengths of both - the higher available compute power in the cloud and lower network latency at the edge. With this, the new challenge becomes how to efficiently orchestrate the FaaS solution across both cloud and edge infrastructures.

A key requirement of effective orchestration between the edge and cloud is being able to establish latency and execution times, among several other factors, in order to efficiently distribute requests across all available resource. This is a perfect application for machine learning, which provides the ability, based on prior acquired knowledge, to make predictions on key variables such as latency using models. Additionally, transfer learning provides the ability to easily adapt acquired knowledge within one machine learning model to another model, assisting in satisfying the main requisite to effective machine learning, high quality data.

1.2 Aims and Objectives

This project will attempt to address the edge-cloud orchestration problem by exploring how machine and transfer learning techniques can be utilised to provide effective per-request orchestration within a FaaS system operating across a cloud and edge environment.

Machine learning will be utilised to predict execution times of functions based on variable features such as input size, with training data being provided by past execution history.

Transfer learning will be utilised to adapt existing machine learning models from functions classified as being similar using function classification techniques, thus avoiding lengthy and high cost training of a new model upon a new function being provisioned if a similar one is already provisioned and collecting execution data.

The aim will be to provide lower overall function latency (factoring network

latency and execution time) than both previously proposed edge-cloud solutions and executing on the edge/cloud environments respectively without any orchestration.

My solution is branded as ”Heterogeneous FaaS”, and will be referred to as such throughout the paper.
This objective can be further split into the following more granular objectives:

1. implementation of a ”Resource Manager”, which consists of a HTTP server to allow:
   (a) application developers to create a new function;
   (b) clients to invoke a function.
2. implementation of a ”Learning Manager”, which will provide predictions of a function’s execution time on each of the available environments considering a number of variable features, such as the function’s input size; this will inform the Resource Manager’s orchestration decision;
3. implementation of transfer learning within the Learning Manager, to allow for already-established machine learning models from other functions to be applied to newly added functions. This will also involve a function classification process to identity similar functions by their features;
4. a full evaluation of the machine and transfer learning aspects of the solution, as well as the overall solution benchmarked against an alternative solution and the cloud/edge directly.

1.3 Structure

The remainder of this paper is structured as follows:

2 Related Work a summary of previous research and solutions within the edge-cloud continuum area, and a problem statement outlining the intended contributions of this project;

3 Architecture an overview of the solution’s technical architecture and components;

4 Implementation a detailed technical overview of the solution’s functionality;

5 Evaluation an evaluation of the solution’s results, benchmarked against a similar alternative solution;

6 Conclusion a review of the project, its contributions and potential future research.

Chapter 2

Related Work

This chapter will review works on the existing state of FaaS (particularly deployed at the edge-cloud), as well as function comparison/similarity finding solutions to inform the transfer learning aspects of the project. The intended contributions of this project will then be detailed in a problem statement.

2.1 FaaS offerings

[11] presents a review and comparison of FaaS offerings in cloud computing, discussing the technical challenges currently encountered, both specific to FaaS and cloud computing in general. Some key issues identified are allowing ”non-cloud systems utilising the serverless architectures” and ”managing the hybrid cloud”, as well as ”supporting heterogeneous hardware of vendors”, presenting a picture of a space currently dominated by cloud solutions with little provision for FaaS architectures outside of the cloud (i.e. at the edge), as well as alternating architectures even within the cloud.

There are both academic and industry solutions which provide FaaS infrastructure either exclusively to edge or cloud environments, or across both in a hybrid implementation (sometimes referred to as the ”edge-cloud continuum”).

The FaaS architecture was initially designed and implemented as a cloud solution with AWS Lambda [12]. Following AWS' initial success with Lambda, a number of alternative cloud computing providers introduced competitor products, notably Microsoft with Azure Functions [13] and Google with Cloud Functions [14]. Each of these provide similar functionality, allowing an application developer to simply upload their function source code and enjoy fully managed invocations of the function from the cloud provider. Each of the cloud providers also provide easy integration with other offerings within their ecosystem, for instance AWS Lambda supports direct integration with AWS API Gateway, which offers

a frontend HTTP server to functions to act as an event trigger for function invocations by end users. All of these offerings are proprietary, with each requiring varying source code schema in order for functions to be compatible, forming a barrier to application developers easily deploying across multiple providers without maintaining several codebases and resulting in platform lock-in [15].

There are also a number of open source FaaS platforms, notably Apache OpenWhisk [16], which is offered in the cloud by IBM Cloud Functions [17], as well as several solutions, notably OpenFaaS [18] and Knative [19], which make use of Kubernetes [20], which has in recent years become an industry standard for container orchestration, of which FaaS platforms are heavily reliant. As well as being deployed in the cloud, these solutions given they're open source are able to be deployed hypothetically anywhere.

There is also active research exploring FaaS in edge-cloud hybrid environments.

[21] proposes an interesting solution whereby FaaS functions are, using machine learning, segregated into several smaller workload functions, which are then distributed across the edge and cloud environments at a more granular scale, allowing for the cloud resources to be utilised only for the highest workload operations and the advantages of low latency otherwise enjoyed by running lower workload tasks at the edge. The notable limitation of this work is that, even with the splitting of functions into smaller tasks, the total latency of a function execution is still bottle-necked by the completion of the highest workload task and therefore network latency to the cloud (if executed on the cloud), meaning actual performance gains can be limited. Its performance is also heavily reliant on the dynamic splitting of tasks into several smaller workloads, meaning that it's best suited to high-workload functions with longer execution times and largely unsuitable for smaller functions, which form a sizeable proportion of workloads deployed to FaaS environments.

[22] proposes a solution which "dynamically determines where [(AWS Lambda in the cloud or AWS Greengrass at the edge)] to execute serverless functions so as to optimise developer-specified performance criteria" using prediction models, with a focus on accommodating high-workload functions for purposes such as data processing or machine learning. Said performance criteria is primarily latency, but an advantage of this work is that it also considers cost in its orchestration decisions. One limitation of this work is that it disregards the heterogeneous nature of edge infrastructure to an extent, for example by imposing the same memory limits on all edge workers.

[23] presents a platform to distribute user requests to functions to workers deployed across an edge and cloud environment, somewhat building on the contributions of [22] by further considering the heterogeneity of edge infrastructure and introducing optimisations such as data caching. Linear regression machine

learning models, trained by historical execution history across multiple workers in the edge and cloud, are utilised to predict execution times of a function on each available worker group (workers are classified by resource specification); these predictions are then used to determine which available workers are suitable to execute a function based on a provided quality-of-service metric (the maximum permissible function latency). Evaluated using a video surveillance application, a 30% cost reduction was achieved by reducing the number of function executions in the cloud, alongside a 25% performance gain across all function invocations. The primary limitation of this work is that, given its reliance on machine learning models to provide accurate predictions, the creation of new functions within the solution requires cost, in terms of both time and resource usage, collection of training data.

2.2 Function comparison/similarity finding

Works involving algorithm (function) comparison, which detects similarities in applications (in our case functions) in order to classify similar functions together, have also been found.

[24] and [25] compare the performance of a number of SLAM algorithms, which are complex algorithms used for robot positioning, based on properties such as CPU and memory usage (the average of both throughout the lifetime of an algorithm), which are extracted from said algorithms via. profiling. [24] builds on other works by combining the accuracy of the algorithms with performance metrics, with all collected metrics then analysed further using a Plackett–Burman process, which aims to mathematically determine any dependence between different variables (in this case the collected metrics). Although this work concerns the comparing of algorithms much larger than what are likely to be deployed to a FaaS environment, the process and successful findings from extracting CPU and memory usage as well as the combining of a number of metrics are valuable.

Another example of algorithm comparison, again with SLAM algorithms, is [26]. This work considers performance primarily based on execution time, as opposed to considering CPU and memory usage as in similar works, or computational complexity. This is performed by providing a number of linear inputs to the algorithms, starting with simple inputs expected to result in lower execution times, followed by incrementally larger inputs ensuing larger workloads. The work was able to successfully model the performance of each of the algorithms, and demonstrates how simple execution time metrics can also be used to model performance somewhat effectively without any reliance on metrics concerning the algorithm's resource usage or complexity.

Although there is research into the area of algorithm comparison and many

examples of the performance of algorithms/applications being compared, no solution which fed the found metrics into machine learning could be found.

2.3 Problem statement

Existing solutions allow for a FaaS infrastructure to be deployed across edge and cloud environments. Works such as [23] utilise machine learning to aid in the effective orchestration between the available workers to minimise network latency and conform to high standards of quality-of-service requirements made achievable by edge environments.

However, many of these solutions are rendered impractical due to the high cost of adding new functions (any FaaS solution, even if only used as part of a single application, could be expected to house many functions) brought about by training new machine learning models upon a previously unseen new function being added. This training process involves repeatedly executing a function in order to generate training data, which is by nature high cost and inefficient, consuming considerable time and resources.

Heterogeneous FaaS aims to address this problem by utilising transfer learning alongside the clustering of functions based on their similar features to remove the need of said high cost training of machine learning models, and instead adopt models from similar functions identified through algorithm comparison, improving upon the models via. online machine learning (incremental learning) as function execution history is naturally generated from genuine function invocations by end users. This, in turn, reduces the cost of adding new functions to the FaaS solution considerably and frees valuable resource to be used for genuine user requests, improving throughput and the overall performance of the solution.

Chapter 3

Architecture

This chapter will describe the adopted architecture of Heterogeneous FaaS and its rationale.

3.1 Design principles

A number of design principles informed by key technical requirements had to be considered when designing the architecture of Heterogeneous FaaS. Some fundamental design principles of the implemented architecture include:

- offering elasticity to cater for fluctuations in load, to avoid the system becoming a bottleneck in function invocation requests;
- maintaining low network latency throughout the solution (i.e. by minimising network requests), to similarly avoid the system becoming a bottleneck;
- allowing for the entire function invocation workflow to be executed at the edge, and only utilising the cloud when necessary. The ultimate test of this is that, should the cloud become unavailable, the solution should still be fully functional;
- to provide an accessible and low-maintenance interface for application developers to create new functions and end users to invoke functions, to match the functionality of existing FaaS solutions;

3.2 Overview

The architecture of Heterogeneous FaaS, as shown in Figure 3.1, can be broadly split into the following elements:

- **Resource Manager** a frontend HTTP server to provide an interface for both application developers (creating new functions) and end users (invoking functions). Also responsible for orchestrating all other elements of the solution;
- **Learning Manager** trains machine learning models and performs function clustering to allow for transfer learning to be performed, then utilises the models to provide predictions to the Resource Manager;
- **Apache Cassandra (database)** an Apache Cassandra database used as a persistent data store. The database stores three entities; functions, function execution logs and machine learning models;
- **AWS Lambda (cloud worker)** acts as a worker node for functions to be executed in the cloud;
- **Kubernetes/Knative (edge worker)** is a Kubernetes cluster deployed at the edge running Knative, which acts as a worker node for functions to be executed at the edge.

The solution intentionally consists of as few individual elements as possible, thus minimising overall latency throughout the life-cycle (as per an aforementioned requirement) of function invocation requests by reducing network latency and the potential for bottlenecks brought about by requiring communication over a network between separated elements of the solution. However, to allow for compute resources to be efficiently allocated and isolated to specific elements of the system (for instance, to allocate higher compute resource to the machine and transfer learning elements which are likely to carry out larger workloads), some elements have been separated. These elements communicate via. HTTP requests. This has also conveniently allowed for different elements to be written in different languages (utilising Java for the Resource Manager to take advantage of its faster execution times relative to alternative languages and high quality Apache Cassandra, AWS SDK and Kubernetes/Knative client libraries, and utilising Python for the Learning Manager to take advantage of high quality and well-supported machine/transfer learning libraries).

Each function deployed to Heterogeneous FaaS is a Docker image (defined by a Dockerfile which implements the AWS Lambda base container image [27] of the function's chosen runtime language; Heterogeneous FaaS only currently supports Python functions, but extension to other languages would be relatively simple given the range of language runtimes supported by Lambda [28]. This allows for the container images to be passed to the AWS Lambda API and compiled automatically into Lambda functions ready to be invoked in the cloud, and also deployed

and executed as an executable Docker container within a Kubernetes cluster (as a Knative Serving service [29]) via. the Lambda Runtime Interface Client (RIC), which can be used to emulate the Lambda Runtime API used in the cloud within a Docker container running in Kubernetes, thus providing a consistent runtime environment between cloud and edge environments. This allows for application developers to write their functions following a single schema (the popularised AWS Lambda function handler schema) and deploy to both AWS Lambda and the provided edge FaaS implementation, abiding by the design principle of providing a simple interface for function creation.

3.3 Key assumptions

There have been several key assumptions made regarding the architecture of Heterogeneous FaaS, as follows:

- the nodes of the system, including the cloud nodes (i.e. the AWS region) and edge nodes (crucially the nodes housing the Resource Manager and Kubernetes cluster) are static and therefore will experience relatively consistent network latency. If any of these nodes were liable to movement, the predictions provided by the Learning Manager based on past execution data would be unreliable due to the high likelihood of network latency differing to that of the Learning Manager's training data;
- function execution time is defined as the latency from the point of a worker being invoked by the Resource Manager, to a response being received from the worker by the Resource Manager;
- functions are entirely stateless, meaning they have no external dependencies for data persistence which would influence the total latency of each execution. Each function takes input data as part of its invocation event and returns output data, without any further querying or persistence of data.

3.4 Resource Manager

The Resource Manager is a Java application which accepts HTTP requests from both application developers (to create a function) and end users (to invoke a function). Its purpose is to manage functions deployed to the edge and cloud environments and orchestrate their invocations, using machine learning models to aid in determining to which environment a function invocation request should be directed and ultimately executed.

The Resource Manager's work can be defined by two workflows: creating a function and invoking a function.

3.4.1 Creating a function

The Resource Manager, referencing the AWS Elastic Container Registry [30] (a private Docker image registry hosted in the cloud) URI provided by the application developer within the HTTP request body, creates an AWS Lambda function via. the AWS SDK (API) and a new Kubernetes Knative service; each of these accepts a Docker container image to build its runtime.

Once the function has been initialised and is ready to be invoked in both environments, the training workflow begins, consisting of the following steps:

1. **collect minimal training data** the Resource Manager will initially collect a minimal amount of training data, as this will be required even if transfer learning is performed. Functions are invoked on each worker (cloud and edge) at input size intervals of 100 (i.e. 1, 100, 200, etc. inputs) up to 1000 inputs.

2. **attempt to train models via. transfer learning** the Resource Manager will initially initiate an attempt at using transfer learning (to avoid the need to generate new training data and instead use function execution data from the similar function) via. HTTP request to the Learning Manager. If the Learning Manager is unable to identify a similar function, the process moves onto step 2.

3. **collect full training data** *if no similar function can be found, new training data will need to be generated to train an entirely new model*. This involves repeatedly invoking the function in either environment 100 times, each time with an increased input size, with each invocation being logged as function execution history stored in the Cassandra database (as would be the case with any function invocation). This step provides real-world training data for a range of function input sizes across both environments (edge and cloud);

4. **train the ML model** this involves calling the Learning Manager, via. HTTP POST request, to begin the process of training a machine learning model for the function (a model for each environment). This is performed asynchronously to avoid HTTP request timeouts to the application developer while waiting for lengthy training workloads to complete.

3.4.2 Invoking a function

The Resource Manager acts as middleware between the end user client and the worker node tasked with executing the function. Upon a function invocation HTTP request being received from an end user, the Resource Manager queries the Cassandra database to fetch details of the function, and then the Learning Manager to fetch how long execution is predicted to take on the edge and cloud environments - this then aids the Resource Manager in deciding which worker (Lambda or Knative) is to be tasked with executing the function (following the simple policy of shortest predicted execution time wins, provided that worker is available). Once a response is received from the worker, it's returned to the end user as a response to their HTTP request. The function invocation is then logged in the Cassandra database as function execution history; this is performed asynchronously so as to avoid bottle-necking the return of a response to the end user and prevent blocking any other latency-sensitive tasks.

The input size of a function is defined as the number of elements passed in its input array. Some inputs may be larger than others, but this provides an accurate representation of input size within the context of the function. The duration is, as per the key assumptions, defined as the latency from the Resource Manager invoking a worker to the Resource Manager receiving a response from said worker.

3.5 Learning Manager

The Learning Manager is a Python application which accepts HTTP requests (via. a Flask server) from the Resource Manager. Its purpose is to manage all of the machine and transfer learning functions of the solution to aid the Resource Manager in its decision making (i.e. where to execute a function).

The Learning Manager's work can be defined by four workflows; training a new function's model using new training data gathered by the Resource Manager (when a similar function to the unseen one being created couldn't be found), training a new function's models via. transfer learning (taking weights from a function identified as being similar), performing incremental learning upon new function executions to incrementally improve a function's models, and using a function's models to provide predicted execution times on each available worker.

3.5.1 Training models using new training data

When no similar functions to the unseen one being added are identified, it's necessary for the Resource Manager to gather new training data, which is then stored in the Cassandra database to be queried by the Learning Manager; this data is then

cleaned and used to train new machine learning models (one for each worker - i.e. one for cloud AWS Lambda, one for edge Kubernetes).

The collected training data is then fitted into a simple linear regression machine learning model, with the single input variable (feature) being input size, and the output variable being duration (function execution time). This model can then be queried to provide predicted function execution times given an input size.

Linear regression has been selected as the machine learning model as the collected function execution data from the Resource Manager across both cloud and edge workers has been found to be relatively linear in nature, with the assumption being that increased input sizes generally corresponding to increased function execution times (durations).

3.5.2 Performing incremental learning

Incremental learning is a form of online machine learning which gradually trains the model upon new data (in this case function execution data) being collected.

This is a key function of the Learning Manager, as it allows for models to be enriched by real-world user data and constantly improve the predictions provided and consequently improve the orchestration decisions made by the Resource Manager.

This process is initiated by the Learning Manager upon an asynchronous HTTP request being received by the Resource Manager, indicating that a function execution has been completed. The new function execution record is then queried from the Cassandra database and used to further train the model of the execution's function and worker.

3.5.3 Predicting function execution times

The Learning Manager's ultimate purpose is to provide accurate predictions of how long a function will take to execute on each worker, given its input size.

This is performed by querying the models for each of the function's workers, with the results then being returned to the Resource Manager to decide which worker will be selected to execute on (i.e. the worker with the shortest predicted execution time).

3.5.4 Clustering similar functions

The clustering of functions based on similar features will be performed using a K-means clustering algorithm. This will facilitate the transfer learning capabilities of the solution detailed in Section 3.5.5 by providing clusters of functions with similar features (i.e. execution times and time/memory complexity).

The clustering features will consist of metrics gathered through profiling (CPU and memory usage), as shown in [24] and [25], in addition to collecting execution times as shown in [26].

3.5.5 Training models via. transfer learning

Training models for new previously unseen functions being created by an application developer via. transfer learning removes the need to perform the high cost and resource intensive gathering of new training data by repeatedly invoking the function, and is the primary novel contribution of Heterogeneous FaaS. Transfer learning also takes advantage of learning acquired from the many executions of other functions (including real world requests from end users) to be applied to the new function, with the aim of improving its predictions as well as reducing training time.

The Learning Manager only currently supports performing transfer learning on Python functions due to methods of identifying similar functions, particularly profiling to collect CPU and memory usage data, being specific to a function's language.

3.6 Apache Cassandra (database)

An Apache Cassandra database is deployed at the edge (so as to avoid network latency when queried by the Resource Manager and Learning Manager, which are also deployed at the edge). For development and evaluation purposes the Cassandra server is initially deployed using a simple deployment strategy, but configuration options provide the ability to easily later distribute it across multiple nodes to assist with maintaining low query latency if doing so becomes necessary under higher workloads.

The database stores three types of entity; functions (metadata of the function such as its AWS ARN identifier and its Knative ID and namespace), function execution logs (one per invocation, attributed to a function with key items being the execution time and input size) and built machine learning models from the Learning Engine.

The database is written to and queried by both the Resource Manager and Learning Manager. Its schema is created and managed solely by the Resource Manager.

Cassandra was chosen as, owing to its NoSQL architecture, it has the ability to be distributed (accommodating the nature of edge environments, which are often spread across many nodes) and supports big data to support the machine and transfer learning functions of the solution.

Bottlenecks upon other elements of the solution querying the Cassandra database are avoided by a combination of deployment at the edge, minimising network latency when being queried by the Resource Manager or Learning Manager, and indexing at the database level allowing for faster querying. The number of queries required at the point of function invocation has also been consciously kept as low as possible to avoid overhead, with the only query required being to get the function's two machine learning models (one for each environment).

3.7 AWS Lambda (cloud worker)

AWS Lambda provides a cloud environment to execute functions on.

The rationale for selecting AWS Lambda to be the cloud worker is:

- AWS remains the most prominent of the cloud providers, and as of early 2022 held the highest market share of all cloud providers globally [31];
- AWS Lambda base images [27] allow for an AWS Lambda environment to be replicated within any Docker container, allowing for a consistent runtime and function schema across the cloud and edge;
- AWS provides a Lambda API and AWS Java SDK [32] which allows for Lambda functions to easily be created and managed.

Being a managed cloud FaaS solution, the underlying infrastructure of AWS Lambda is almost entirely abstracted, with minimal configuration extending to defining the max memory a function should be assigned; this is kept at the default of 128MB.

Upon a HTTP request being received by the Resource Manager from an application developer creating a new function, a new Lambda function is created by providing the function's container image URI (hosted on AWS Elastic Container Registry, a private Docker container registry) to the Lambda API via. the SDK.

The Lambda function can then be invoked once its initialisation process is complete. If an early invocation request is made, for instance if the new function needs to be invoked to collect training data, it's likely that the initialisation process will not yet be complete and therefore the Resource Manager will poll the Lambda API until it's indicated that the Lambda function is ready before retrying an invocation request.

All Lambda functions are built and executed within the ARM64 CPU architecture, both to allow for portability with ARM edge environments, of which many IoT and other fog devices are, and take advantage of improved efficiency and lower costs in Lambda compared to x64/x86.

Should, for whatever reason, AWS become unavailable, any invocations routed to it by the Resource Manager will be re-routed to an alternative worker (i.e. the edge). This prevents AWS and the cloud from becoming a dependency of the Resource Manager and the operation of the entire solution.

3.8 Kubernetes/Knative (edge worker)

A Kubernetes cluster running Knative is employed across any edge node(s) to provide an edge environment to execute functions on.

The rationale for deploying Kubernetes and Knative at the edge is:

- it's an increasingly popular method of managing containerized applications, of which Heterogeneous FaaS is one (with functions being stored and executed within containers), and boasts significant community support;
- Knative provides many abstractions over a standard Kubernetes cluster that allow for serverless and event-driven applications to be deployed easily with minimal setup, which Heterogeneous FaaS takes huge advantage of. Knative Serving provides a runtime environment consistent of AWS Lambda's when used alongside AWS base container images, and this combination is used to store and execute all functions housed at the edge, with autoscaling and scale-to-zero functionality also providing key properties of FaaS and aiding in minimising resource utilisation whilst still providing elasticity, which is of course crucial at the edge given the limited resource available;
- Kubernetes and Knative offer an officially supported API with an associated Java SDK which allows for the cluster to be interfaced with easily by the Resource Manager;
- There is active research such as [33], and industry solutions such as KubeEdge [34] which particularly cater Kubernetes to edge environments, considering issues such as resource limitations (particularly memory footprint) and the heterogeneity of infrastructure (for instance, supporting a range of CPU architectures).

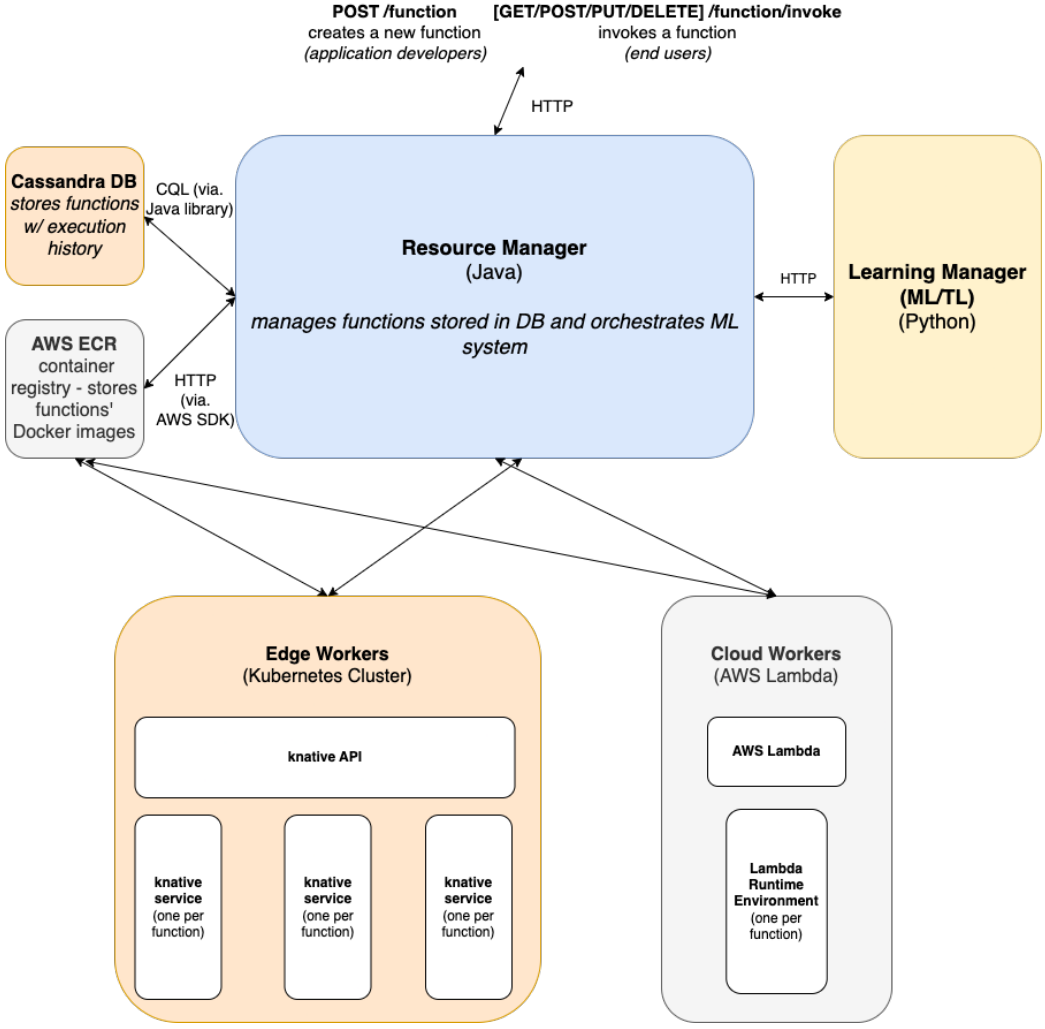
A function is defined as a Knative Serving [29] service, which allows for a Docker container (in this case the AWS base container image for the function's runtime language packaged with the function source code) to be provided via. AWS ECR [30] and a Knative service created. This is the only point at which communication outside of the edge environment (i.e. to AWS ECR) is required throughout the lifecycle of a function. Here, the reliance on an external dependency has been traded-off for the convenience of application developers only

needing to push their function images to a single registry, with cloud connectivity being required during the function creation process when it's required anyway to create AWS Lambda function(s).

Once created, the Knative service will run within Kubernetes pod(s) and can be invoked via. HTTP request. The Knative service's pods will be automatically scaled up and down (including to zero if there is no usage) based on current usage; this is the key difference between a standard Kubernetes service [35] and a Knative service, with Knative providing its own Kubernetes services in addition to the created one housing the function to automatically manage scaling and otherwise maintain function services. This allows for the Resource Manager to pass responsibility of maintaining much of the FaaS infrastructure to Knative, which has better visibility of the state of the Kubernetes cluster and its activity at any one time.

If the Kubernetes worker is selected by the Resource Manager as the preferred worker (i.e. it has a lower predicted execution time), it will only be selected if the Kubernetes' cluster is not at max resource utilisation; this is defined as 95% CPU or memory usage across the cluster. This is to prevent a Knative service from being invoked and suffering from a prolonged cold start whilst it waits for a pod to become available. This provides provision to route all invocations to the cloud when the edge is at capacity, providing elasticity within the solution.

Figure 3.1: An architecture diagram of Heterogeneous FaaS



Chapter 4

Implementation

This chapter will detail each stage of implementation, and how the architecture detailed in Section 3 has been implemented at a lower level.

The implementation phase of the project can be broadly split into the following stages, in order of their completion:

- implementation of function containers, which are used as an execution environment for function code;
- implementation of the Resource Manager, to manage orchestration of functions and their executions, and the persisting data of both in the database;
- implementation of the Learning Manager, which manages all of the machine and transfer learning aspects of the solution:
  - machine learning to build models from new training data;
  - incremental learning to allow for machine learning models to be incrementally improved upon new function executions;
  - prediction querying from built models to provide predicted execution times for a function across all workers given an input size;
  - function clustering to identity similar functions based on their profiles;
  - transfer learning to build models from the data of an existing model.

Many of these stages are prerequisites to others; for instance, function comparison must be achieved to allow for similar functions to be identified before transfer learning can be performed.

4.1 Implementation of function containers

The prerequisite to all further steps of implementation is to establish an runtime environment for function code to run on.

As discussed in Section 3, AWS base container images [27] are used to provide said environment. These are a Docker container image extending the AWS Lambda base image [27] in the ARM64 CPU architecture for the function's runtime language is pushed to the Lambda API, with the initialisation of the function on the AWS side then being a fully managed process.

An example of a Dockerfile defining a function's container image, in this case for a Python 3.8 function, is:

```
FROM public.ecr.aws/lambda/python:3.8-arm64

# Copy application code into the container
COPY . ${LAMBDA_TASK_ROOT}

# Set the command to be run when the container starts
CMD ["main.handler"]
```

The base image for the function's language, which provides the Lambda Runtime API running on Amazon Linux as is the case in the cloud Lambda environment, is extended with any files comprising the function and an entrypoint to the function (known as the function handler) defined, which is in this case the *handler* function within the *main.py* file.

4.2 Implementation of the Resource Manager

The Resource Manager is a Java Gradle [36] application used to orchestrate all elements of the solution, handling the workflow of functions during their creation by application developers and invocation by end users.

The Resource Manager has been written in Java owing to its persistent popularity, fast execution times due to its compiled nature and well-supported libraries, particularly for Apache Cassandra [37] and Docker [38]. Potential alternatives with similar levels of library support for other components include Go, however this language is less commonly used within the research and open-source community in this context, making Java more favourable.

Standard best practices, such as domain driven design, interfaces and code re-usability, has been enforced throughout the development of the Resource Manager. This is particularly important as it allows for implementations of alternative

components (for instance, replacing Cassandra with another database engine) to be done easily without requiring a large code refactor.

The Gradle [36] package manager is used to easily manage packages from the popular Maven Central repository, with the benefit of lower build and run times than alternatives, such as Maven itself.

Logging is provided by the native *java.util.logging* library, as this implementation is extensible by the majority of available Java libraries and therefore allows for logs to be written to a range of different sources as is desired.

4.2.1 Undertow HTTP server

The *Undertow* package is used as a HTTP server to provide a frontend API for application developers and end users to call when creating or invoking functions respectively.

For the purposes of the Resource Manager and the wider solution, an application developer is defined as somebody creating functions and managing the overall solution, and an end user is somebody who invokes functions.

The Resource Manager initially used Flask as a frontend HTTP server, however owing to its support for asynchronous processing of requests, Flask was removed in favour of Undertow [39]. Each request is handled asynchronously on its own threads, where blocking IO (receiving the request from the client and then returning a response) operations can be performed without blocking the main thread of the application, therefore allowing for many requests to be received and processed concurrently; this is clearly crucial functionality of the Resource Manager, which is expected to scale to handle many requests at any one time.

The following API endpoints are provided by the HTTP server:

- **POST /function (create function)** called by an application developer, creates a function in Kubernetes and AWS (if both are selected- it's possible to declare *isCloudSupported* or *isEdgeSupported* as true/false in the request body);
- **POST /function/invoke (invoke function)** called by an end user, invokes a function on the worker selected by the Resource Manager and returns the response of the execution;
- **PUT /credentials (set credentials)** called by an application developer to set AWS API credentials.

Each of these routes is served by its own *HttpHandler* class, which handles the complete workflow of the request, from the receipt of the request to the response being returned to the client.

Google's Gson [40] library is used to deserialize the JSON provided in the request body before the necessary processing occurs. Gson is also used to serialize any responses to be returned as a HTTP response; common actions like these are performed by the *HttpHelper* class, which contains a number of common actions utilised by the multiple *HttpHandler* classes.

4.2.2 Apache Cassandra

It was decided to use Apache Cassandra as a persistent data store, with the rationale for doing so discussed in section 3.4; key highlights are that this allows for fast read and writes of data which cater for the latency-sensitive nature of the solution, as well as elasticity and the ability to distribute the database server nodes across multiple machines.

Cassandra is interfaced with by the Resource Manager using the DataStax Java Driver for Cassandra [37], which provides convenient capabilities such as mapping of database entities to Java objects. This mapping is achieved using an *Entity* class, which is the Java object for a Cassandra table to be mapped to; for instance, the *function* Cassandra table is mapped to the Function entity class in Java.

The Entity class is then used within a *DAO (data access object)* class to provide querying ability, for example here with *FunctionsDao* providing select and insert database queries on the *function* table:

```
@Dao
public interface FunctionsDao {
    @Select
    Function get(String functionName);

    @Insert
    void create(Function function);
}
```

This class provides methods to select and insert a Function record from/to the database.

A *CassandraClient* class within which the Cassandra CQL session is stored (to ensure there's only a single session throughout the application) and all database operations are performed has been implemented. This is alongside the implementation of a repository pattern, with repository classes consisting of common database operations such as "creating a function", has been written around the base DataStax library to further abstract low level details of writing to and querying the database and limit code duplication. For instance, all database queries

are executed via. the *execute* method in *CassandraClient*, which provides an abstraction over what would otherwise be a somewhat verbose library call due to the CQL session being stored within DBClient.

The *CassandraClient* also handles the creation of the *heterogeneous_faas* Cassandra namespace and tables for each of the entities (functions, function executions, workers, ML models and clusters) if they don't yet exist (i.e. on first run of the Resource Manager). The automation of this process assists in providing a simpler setup process for application developers.

CassandraClient implements the *IDBClient* interface, and so it's very easy and requires no other code changes to replace Cassandra with an alternative database engine by simply implementing the interface.

4.2.3 Docker and AWS ECR

To create a new function, the application developer provides the following in a HTTP POST request to the Resource Manager */function* API endpoint:

```
"function": {
    "name": "[unique function name]",
    "source_code": "[source code in string format]",
    "edge_supported": true,
    "cloud_supported": true,
    "example_inputs": [JSON array of string inputs]
}
```

Upon receiving a request, the Resource Manager will initially build a Docker image using a predefined *Dockerfile* (i.e. the Python AWS Lambda base image as shown in Section 4.1) and the provided source code written to a *main.py* file, both of which are written to a temporary directory. Following authentication between Docker and AWS ECR via. the AWS SDK, the Docker API equivalent to the *docker build* and *docker push* (to AWS ECR) CLI commands are ran against the temporary directory using the *com.github.dockerjava* library [38].

Here, the Resource Manager communicates with the Docker API running on a local Docker instance via. the library's HTTP Docker client.

4.2.4 AWS IAM

AWS Identity and Access Management (IAM), AWS' authentication and identity service used to manage access to an account's infrastructure, is also managed automatically by the Resource Manager.

An IAM role with an associated IAM policy allowing it access to the Lambda service is automatically created upon the first request being made to AWS by the Resource Manager. The IAM policy's document is as follows:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

4.2.5 AWS Lambda

AWS Lambda was identified as the cloud worker for functions, with the rationale for doing so discussed in section 3.5. Lambda is interfaced with using the Lambda API via. the AWS SDK for Java [32].

Provided an application developer has defined their function as cloud-enabled, the function will be created in AWS Lambda immediately after its Docker image is built and pushed.

The URI of the function's uploaded AWS ECR container image, which was persisted against the function's object during the Docker build and push process, is passed to the Lambda function creation API, making sure to define the architecture as ARM64. Being a managed service, the Lambda API then handles the further setup of the function for it to be ready to invoke.

Invoking of functions on an AWS worker (Lambda) is also performed via. the AWS SDK, with the end user's function input (request body) passed to the function as part of its invocation event to be retrieved as input by the function's handler logic. The AWS ARN, queried from the function's Cassandra record, is used to identify the function.

The invocation response is then returned as a JSON string in the HTTP response to the end user.

It's possible to enforce memory constraints on Lambda functions, however in Heterogeneous FaaS the default limit of 128MB is maintained. Lambda functions are also bound to being invoked for a maximum 15 minutes (after which a cold start will be required if not kept "warm" by invocations). Both of these limits

should be more than sufficient for a standard FaaS function as is being evaluated in this project, but if the memory limit was required to be lifted for a particular workload, this could be easily achieved within the Resource Manager on a function-by-function basis.

4.2.6 Kubernetes/Knative

Kubernetes, providing with minimal configuration a versatile edge environment which can be deployed across multiple nodes (a key requirement of distributed edge networks), with additional benefits brought by the inclusion of Knative on-top to provide a serverless environment to run function containers on.

Kubernetes is interfaced with via. the third-party fabric8io Kubernetes library [41], which provides an SDK to the Kubernetes native API, as well as the Knative API via. an extension. This allows for functions to be created easily by initialising a new Knative service. The API also allows for virtual memory and CPU limits [42] to be set on the pods (virtual machines) the service will execute the function's container on, although the current Heterogeneous FaaS implementation leaves these unspecified to allow for Kubernetes to manage its resources with full visibility of the entire environment and current workload; the solution could however easily be extended to utilise resource constraints to enforce heterogeneity of the edge environment and provide a choice of different containers with differing resources if so desired. If a Knative service doesn't have explicit resource limits set, any redundant resource on a Kubernetes node (the entire edge cluster) can be assigned to a function pod at-will while it remains available; this maximises resource utilisation at the edge.

Knative services can also be declared within many namespaces if so desired, for instance to split the functions of several application developers, however Heterogeneous FaaS in this instance creates all services within the *default* namespace.

As the container images of all functions at point of creation are stored in a private AWS ECR registry, Knative requires access to pull said images. To allow for Knative to access the private registry, a Knative secret containing AWS Docker login details is created upon the Resource Manager interacting with Knative for the first time, with this secret then being attached to the *default* service account which owns and manages all Knative services.

Provided the application developer has defined their function as edge-enabled, a Knative service for a function will be created immediately following the AWS Lambda creation process.

Creating a function requires for its container image to be pulled from the AWS ECR private registry hosted in the cloud, with Knative then creating the necessary Kubernetes services and pods to run the container's service.

Here, the Knative service name is being defined as the function's name, followed by the container being defined by its AWS ECR URI, and the Service created via. the SDK/Knative API.

The invocation URL of the created Service is then formed by combining the created Knative Service URL (returned by the Knative API in the format "[service-name].[namespace].[cluster-IP].sslip.io") with the standard Lambda API endpoint dictated by the Lambda container base image ("2015-03-31/functions/function/invocations"). *sslip.io* is a service which allows for an IP address to be passed as a subdomain to the URL and the URL resolve to that IP address, allowing for further subdomains on the URL which would not otherwise be permissible if using a raw IP address. As it's immutable, this URL is compiled upon a function being created and persisted in Cassandra, thus removing the need for this to be done repeatedly at the point of invocation at cost.

Invoking a function on a Kubernetes worker simply involves sending a raw HTTP request containing the end user's payload (passed in the request body) to the invocation URL.

A raw HTTP request is sent with its body consisting of the function payload in string format, via. the native *java.net.http* library. If the HTTP response's status code indicates a successful invocation (i.e. a 200 or 204 status code), the response body is returned to the end user in response to their initial HTTP request to the Resource Manager.

If the response status code indicates an error, two possible flows are followed. If the status code is 404, indicating the function couldn't be found, the request is retried after 1 second to account for periods in which scaling or initialisation may render the function temporarily unavailable; it's assumed that any 404s will be temporary hence the constant retry logic. Otherwise, the response body containing details of the error (which could, for instance, be a 400 Bad Request error indicating an issue with the end user's input) will be returned to the end user.

The Metrics API [43] is installed on the Kubernetes cluster and used by the Resource Manager to query the CPU and memory utilisation of the cluster at any one time to aid its orchestration decisions.

4.3 Implementation of the Learning Manager

The Learning Manager, a Python Quart [44] app, is interfaced with by the Resource Manager via. a HTTP API.

The implementation of the Learning Manager is more investigative in nature than that of the Resource Manager, due to it encompassing many of the experimental research aspects of the project.

4.3.1 Quart HTTP server

Quart has been used to provide a HTTP server as it's asynchronous, allowing for many requests from the Resource Manager to be processed concurrently (preventing the Learning Manager from becoming a bottleneck to the Resource Manager), and lightweight, offering low level control of the handling of HTTP requests.

The following HTTP API endpoints are provided by the Learning Manager:

- **PUT /train/{functionName}** triggers the training of models (one model per worker; i.e. one cloud and one edge model) for a function;
- **PUT /train/incremental/{functionName}** triggers the incremental learning of an existing function-worker model using a newly recorded function execution;
- **GET /predictions/{functionName}** retrieves predicted execution times per worker for a function, given its input size;
- **PUT /cluster** places all functions currently created within Heterogeneous FaaS into clusters;
- **PUT /transfer/{functionName}** triggers an attempt to find a similar function to the one to be trained and perform transfer learning.

4.3.2 Training models using new training data

The training of machine learning models by the Learning Manager is performed within a *train* Python module, which is referenced by the Quart server on the */train* route. The *scikit-learn* [45] library is used to train models and *numpy* [46] is used to prepare data.

The first stage of the training of the models is to query the training data (function executions) collected by the Resource Manager. Only successful executions, as flagged by the Resource Manager, are queried. These provide function execution times (durations) based on a number of input sizes.

The distribution of collected function execution times based on input size for each worker, as shown in Figures 4.1 and 4.2, shows that there are clear outliers in the data preventing any clear trends from being visible. These outliers are likely a result of the cold start problem [47], which occurs when the initial invocations of a function takes significantly longer than subsequent ones which are "warm started" due to the environment needing to be initialised to a state whereby it's ready to execute the function. Outliers can also be present due to other factors, such as network issues causing a spike in latency. As these executions are not representative of the overall data and skew the data, they need to be removed.

This can achieved by the use of a Z-score, whereby the mean and standard deviation of the training data's durations (input variables) is used to calculate a Z-score following the formula:

$$Z = (x - mean)/standarddeviation$$

With x being the input variable, in this case an array of duration integers.

This Z-score represents how far an individual duration deviates from the mean (average) of the data, thereby highlighting any outliers. Durations with a Z-score higher than the specified threshold (3 has been found to be the optimum in this case) are removed from the training data and not fitted to the model.

Figures 4.3 and 4.4 show the distribution of data following the removal of outliers using the Z-score method. It can now be seen that the most severe of outliers have been removed, however the distribution of the data, especially in the case of the Kubernetes worker shown in Figure 4.4, is still non-normal and non-linear.

An alternative to using Z-score to distinguish outlier data is using the interquartile range (IQR). This involves calculating the 25 (Q1) and 75 (Q3) percentiles of the data to find where the majority of the data lies, as well as the interquartile range. A threshold value (in this case the optimum was found to be 1) is then used in the following formulae to find a lower and upper bound of values:

$$lowerbound = Q1 - threshold * IQR$$

$$upperbound = Q3 + threshold * IQR$$

The results of employing IQR to establish and remove outliers are shown in Figures 4.5 and 4.6.

Although there remain issues with the data with some noticeable outliers still present, the extent of outliers present has been drastically cut, meaning the data is less noisy and distributed across a smaller range of durations when employing the IQR method over the Z-score method.

It is possible to employ both methods alongside eachother, with a benefit being that outliers missed by one may be caught by another. In this case, both methods applied together didn't heed results any more favourable than IQR alone given its success at removing outliers at a low threshold, and therefore only the interquartile range method has been employed within the final solution.

Clear and relatively linear relationships are now visible in the data following the removal of outliers, with a more normal (Gaussian) distribution being followed. This means that the data can now be considered cleaned and ready to be fitted to models.

The cleaned training data is used to train a simple (uni-variate; with only a sin-

gle input variable) ordinary least squares linear regression model. The *scikit-learn* library's *SGDRegressor*, which utilises the stochastic gradient descent method to fit data to the model, has been utilised to fit the data to the models.

The created models are shown in Figures 4.7 and 4.8; the red plot lines show the predictions returned for each input size by the model. The AWS model has a coefficient score of 0.6095158671735333 (the closer the value is to 1, the better) and the Kubernetes model as score of 0.21351908398277097. Due to the nature of the Kubernetes model predictions are likely to be less accurate than those of AWS, as will be explored during evaluation.

Linear regression models offer a number of parameters which can be used to influence its derived linear regression function, used to establish the output variable (predicted duration) based on the input variable (input size). In this case, a low alpha value of *0.0001* is used as a higher value would result in increased regularization to prevent overfitting, whereby the model becomes too fitted to the training data provided (in this instance up to an input size of 1000). Overfitting, whilst to be avoided, is of less concern in this instance as the input sizes of function executions are unlikely to be significantly higher than those included in training data.

Once the model has been created and fitted, it is persisted to the Cassandra database as a bytes string, which is formed from the created model using the *pickle* library.

4.3.3 Performing incremental learning

Upon each function execution, incremental learning (otherwise known as online machine learning) is performed with the aim of improving the model of the function and the worker it was executed on, and in turn provide more accurate predicted execution times for future invocations.

Incremental learning is also particularly useful to further fit the models of functions adapted from an identified similar function based on execution data from the function itself as it's collected from real-world usage by end users.

Upon a function invocation being completed by the Resource Manager and a function execution being recorded in Cassandra, the Resource Manager will inform the Learning Manager by HTTP request to the Learning Manager's */train/incremental* API.

It's possible to achieve incremental learning by either fully fitting the model, as performed in Section 4.3.2 when training the model initially, or by partially fitting the newly collected data to the existing model. Partially fitting with a small dataset (i.e. data from a single execution) is likely to have an adverse effect on the reliability of the function, and when attempted was found to massively skew the model resulting in highly inaccurate predictions being returned.

Therefore, with each execution the model of the recent execution's function and worker is fully fitted including the newly collected data from the recent execution, as well as all previous successful execution data. Outliers are removed from the dataset as normal.

The new model, once fitted, is then persisted back into Cassandra, overwriting the previous model for the function and worker combination.

4.3.4 Querying predictions from models

Once a model has been built, it can be queried for predictions via. the Learning Manager's */predictions* API endpoint. This API endpoint is queried by the Resource Manager as part of the function invocation process to inform its decision of which worker to execution a function on.

The function's machine learning models are queried from Cassandra and loaded using the *pickle* library. The loaded models (one per worker for the queried function) are then iterated through and queried with the current execution's input size, as passed by the Resource Manager.

The queried predictions, one per worker, are then returned to the Quart server, which in turn returns them to the Resource Manager via. HTTP.

4.3.5 Clustering functions

The ability of the solution to perform transfer learning during the function models training process is facilitated by the clustering of functions.

Function clustering is triggered asynchronously via. the Learning Manager's */cluster* API by the Resource Manager upon a new function being created, so that the new function is added to a cluster. An alternative to this approach would've been to implement a queue, however this was deemed unnecessary with clustering operations being performed asynchronously within both the Resource Manager and Learning Manager and therefore not blocking any client requests.

A number of properties of each function with minimum variability have been extracted from each function and provided as input variables to a K-means clustering model. Some experimentation has been required to establish the best features to extract from each function, with the aim of determining which features best represent how long a function is likely to take to execute; in this context this is the most important metric, as if two functions take similar lengths of time to execute their prediction models will be similar.

One of the most representative features of a function's execution profile is how long a function takes to execute given a specific input size. To gather such data, the Learning Manager, via. the Resource Manager, invokes each function being clustered on both edge and cloud workers with the input sizes 1, 100, 200, 500,

750 and 1000. This provides execution times across a range of input sizes to provide an impression of the time complexity of the function (how its execution time scales as its input data scales) without needing to perform any source code level analysis. Each worker being executed on has had one execution ran on it prior to the analysis executions being performed to reduce the likelihood of cold starts, which will have an impact on execution times and provide inaccurate feature data.

In addition to execution times at workers, further metrics can be extracted from the function via. profiling. This allows for execution time locally within the Learning Manager to be collected, providing an accurate depiction of execution time without factors such as network latency and cold starts, as well as cumulative memory usage to provide insight into the memory footprint of the function, a factor closely associated with execution times. This data is extracted by profiling the function's source code, queried from Cassandra, using the *cProfiler* [48] library.

These input variables, once collected for each function, are each passed to the *scikit-learn* library's *KMeans* function, which builds a K-means clustering model and fits the input variables to it, assigning each to a cluster. The optimum number of clusters in the model has been found to be 8, meaning that a minimum of 8 functions are required to permit clustering (with so few functions clustering would be unreliable anyway, as each would simply consume its own cluster).

Once built and fitted, the cluster is persisted to Cassandra as a bytes string using the *pickle* library. The index (ID) of the cluster each function belongs to is also persisted in Cassandra against each function's record in the *function* table; the functions belonging to a cluster wouldn't otherwise be able to be identified due to the cluster itself providing no labelling of each cluster.

The built cluster can now return predicted clusters, and thereby identity functions with similar properties, based on the features of unseen functions as they're created within Heterogeneous FaaS.

Upon a prediction being queried from the clustering model for the purpose of finding a similar function, the difference between the cluster's centroid value (the mean of all feature vectors within the cluster) and the cumulative value of the new function's feature vector (the sum of all of its feature values) is calculated. If the difference exceeds a defined threshold, for which the optimum value has been found to be 30, the cluster prediction will be considered unreliable and transfer learning will not be performed. Otherwise, if the difference is below the threshold, transfer learning will be performed, as detailed in Section 4.3.6.

4.3.6 Training models by transfer learning

If a suitable cluster is found for a function, the models (one for each worker) of a function within the cluster will be used in transfer learning to build a model for the new function.

There are two means by which transfer learning can be performed on linear regression models; just the coefficient (weights) and intercept (bias) of the similar function's model can be applied to the new model and small amounts of newly collected training data applied to it, or the entire model, including its input and output variables, can be taken and copied into the new function's model. It's been opted to apply the latter in this solution as genuine function data can be easily collected via. the Resource Manager, and so there is little benefit, in terms of time gains, to applying data from another function.

A minimal amount (10 executions) of training data has been collected prior to the Resource Manager triggering the Learning Manager to attempt transfer learning. If transfer learning is performed, the Learning Manager will query the collected training data (10 executions for each worker) and fit this to the new function's model, which has already had its coefficient and intercept previously defined, following the same process as incremental learning as detailed in Section 4.3.3. This data will then make any necessary adjustments to said coefficient and intercept, tailoring the newly created model to the new function whilst also accounting for learning from the other function's additional executions.

Once the models of the new function have been transferred and fitted, they are persisted to Cassandra using *pickle* following an identical process to if the model was trained using new training data.

Figure 4.1: Execution times of a function on AWS Lambda

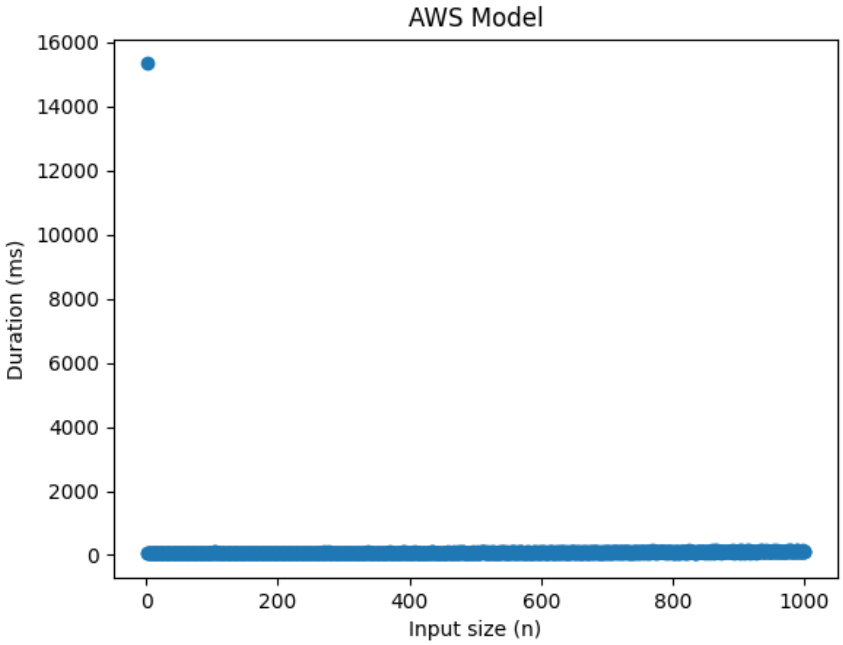


Figure 4.2: Execution times of a function on Kubernetes

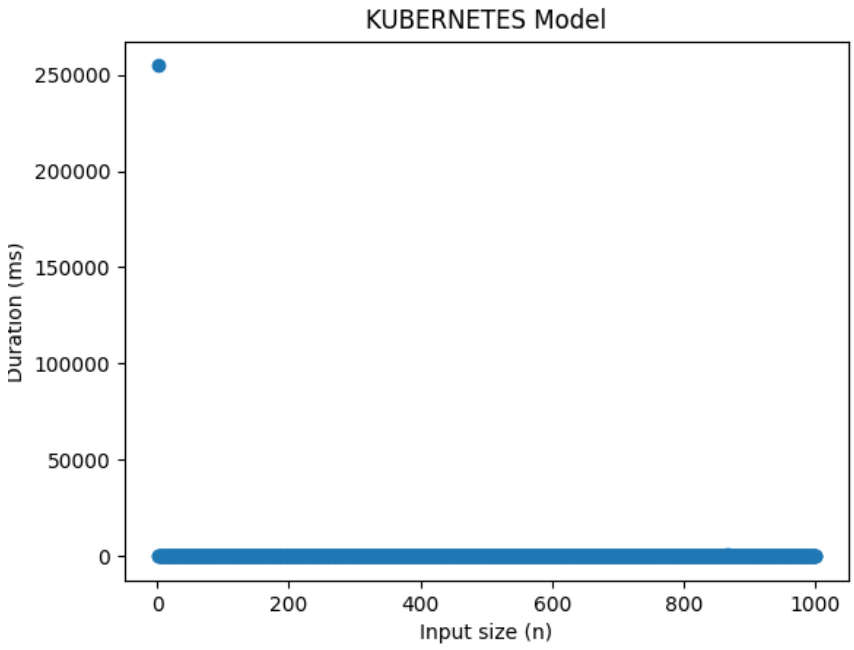


Figure 4.3: Execution times of a function on AWS Lambda (with Z-score outliers removed)

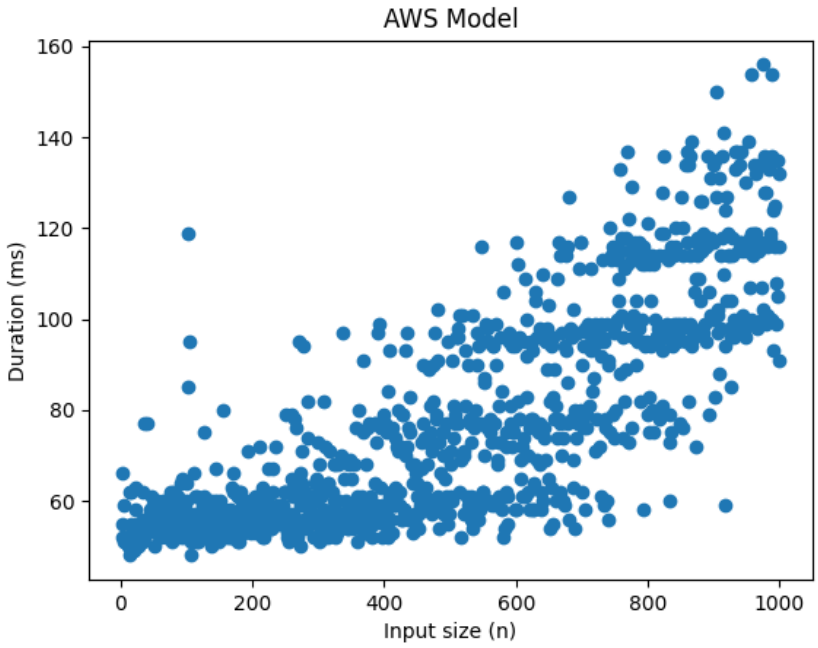


Figure 4.4: Execution times of a function on Kubernetes (with Z-score outliers removed)

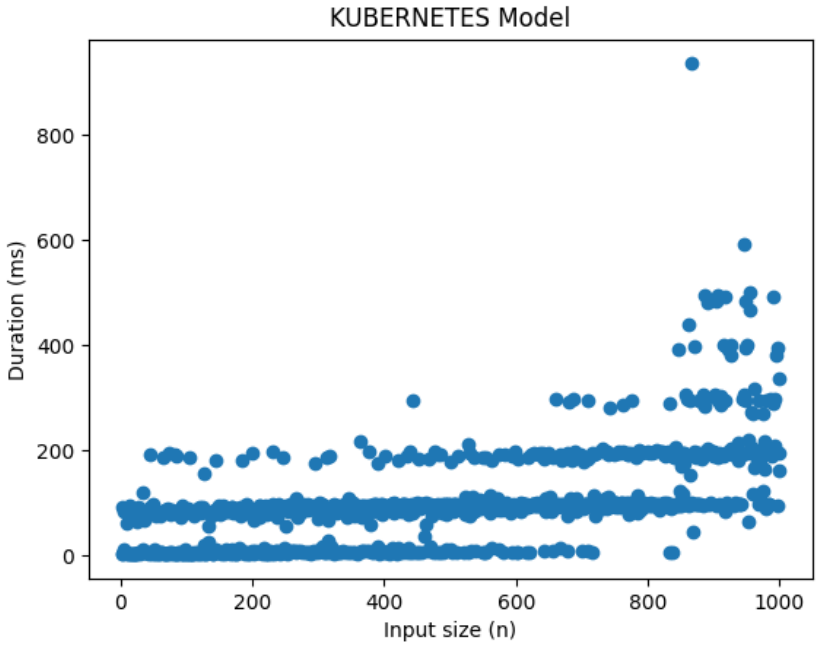


Figure 4.5: Execution times of a function on AWS Lambda (with IQR outliers removed)

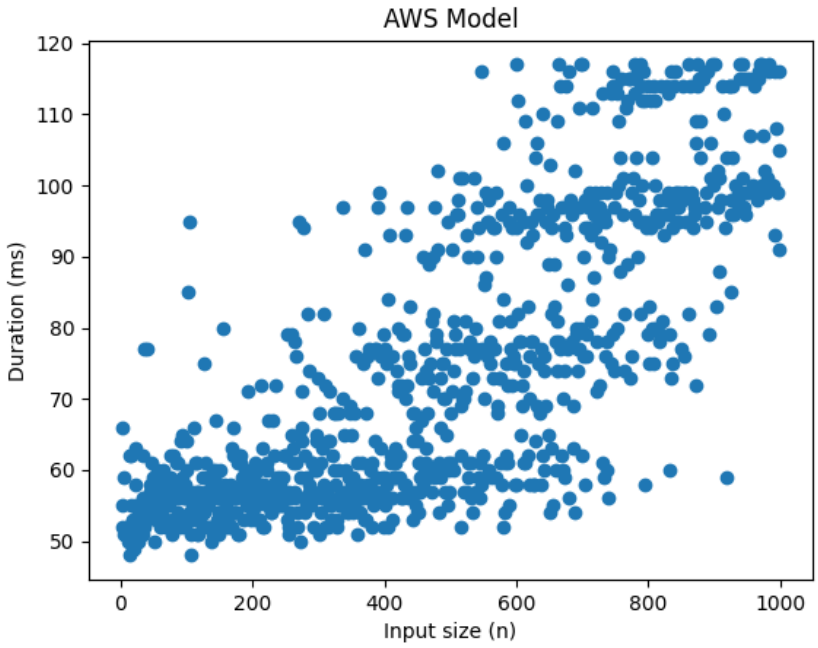


Figure 4.6: Execution times of a function on Kubernetes (with IQR outliers removed)

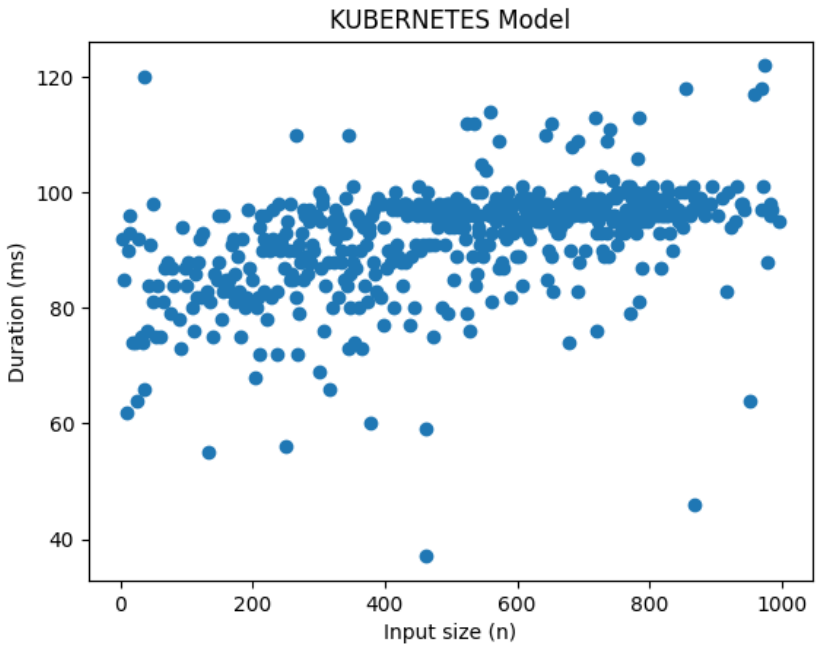


Figure 4.7: Linear regression model of a function on the AWS worker

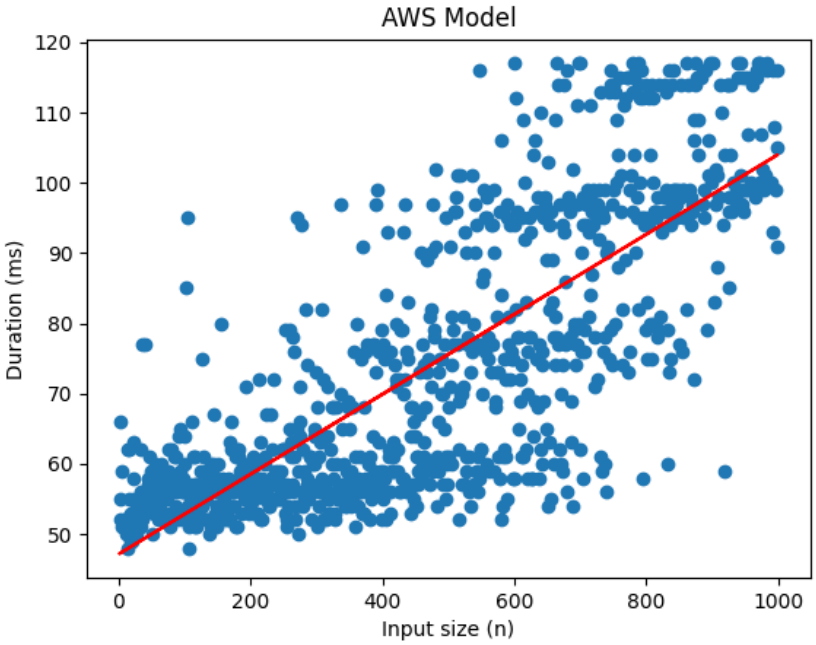
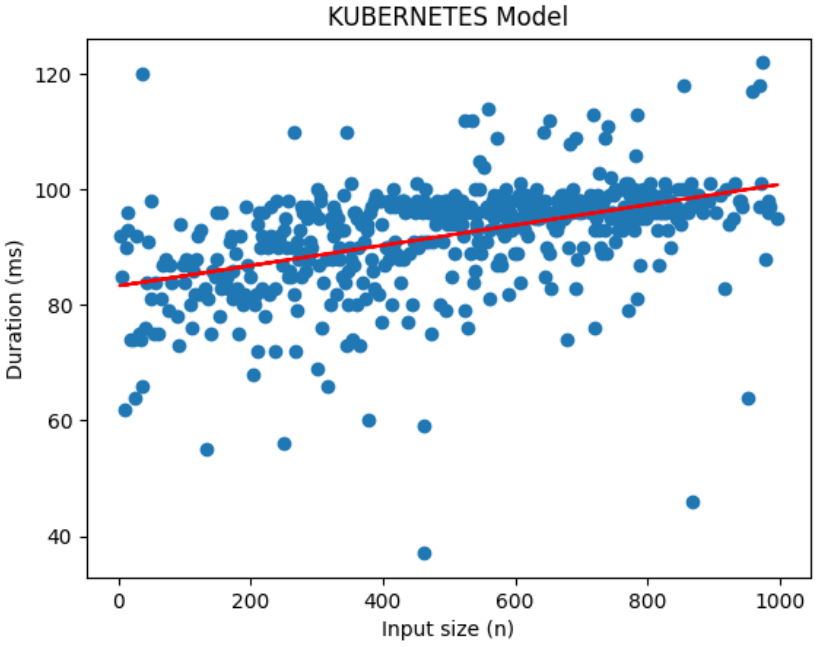


Figure 4.8: Linear regression model of a function on the Kubernetes worker



Chapter 5

Evaluation

This chapter will evaluate the solution's overall performance and discuss key findings.

5.1 Experimental setup

Each element of the solution was ran on the following hardware/with the following configuration during the evaluation phase:

- **Resource Manager** running natively as a Java app on a MacBook Pro (M1 Pro, 16GB memory);
- **Learning Manager** running natively as a Python app on a MacBook Pro (M1 Pro, 16GB memory);
- **AWS Lambda** running within the *EU-West-1* (Ireland) region, with each Lambda function being assigned 128MB of memory (as per default config);
- **Kubernetes/Knative** with a single Kubernetes node, running on Docker container using Kind [49].

Apache JMeter [50] will be used to facilitate load testing of the solution. JMeter allows for many HTTP requests to be sent to the Resource Manager simultaneously, allowing for many concurrent function invocations at once to test the scalability of the solution, emulating an evaluation environment as close to a real-world environment as possible.

Given the small and low workload nature of the functions to be evaluated, the Heterogeneous FaaS default configuration has been modified during the evaluation phase. A 50MHz CPU and 128MB memory limit has been imposed on the

Kubernetes pods housing function's Knative Services to emulate an edge environment with resource constraints relative to the cloud with more available resource. The resource configuration in the cloud (AWS Lambda) will remain as default, with the selected AWS region (*EU-West-1* (Ireland)) being located geographically further away and conversely suffering from higher network latency than the edge environment, which is running locally and therefore theoretically suffers from no network latency. These software level bounds are intended to emulate a real-world environment whereby a Kubernetes cluster would run on an edge environment separate to the Resource/Learning Manager, which owing to a lack of hardware access hasn't been possible as part of this project.

For the purpose of the evaluation, each function invocation has been performed as a "warm start", meaning that in every case the worker being executed on has been initialised to avoid any execution bottlenecks from a cold start [47], which would have a significant impact on results. This has been achieved in AWS by invoking the Lambda function immediately prior to the evaluation (Lambda functions remain invoked for 15 minutes), and in Knative by disabling the scale-to-zero functionality, meaning that at least one Kubernetes pod is always running for each function.

5.2 Test function preparation

IBM's Project CodeNet [51] dataset of function source code will be used to provide a collection of experimental functions, each of which is a function submitted to an online code judging site whereby programmers submit their coded solutions to problems, similar to Leetcode. Although such functions may not be fully representative of applications typically deployed to FaaS environments, the use of this dataset provides a range of authentic functions written by several developers with varying code styles.

A subset of Project CodeNet problems has been selected based on the profile of a problem's input data - notably its format, constraints and how it scales. For instance, one selected problem takes two random integers as an input, sums both together and then returns the number of digits in the answer; this problem has been selected as it allows for its input to be scaled easily to provide training data and can have many solutions providing a solution in different ways, providing similar functions which can also be used to evaluate Heterogeneous FaaS' function clustering capabilities.

For each selected problem, a random subset of "accepted" (valid) solutions has been selected of functions written in Python.

Python functions have been selected to be used during evaluation as, besides Python being the only language supported by the Learning Manager to perform

transfer learning, they provide an accurate representation of function execution times as they suffer minimally from the cold start problem [47], which is particularly prevalent in compiled languages executing on virtual machine based runtimes, such as C# and Java, but much less so in Python being an interpreted language and therefore having a lower startup time. It is worth, however, noting that Python functions do still suffer from cold starts, with for instance one initial "cold start" invocation of a Python function having a duration of 822.00ms, in contrast to the following "warm start" invocation immediately after taking only 146.00ms.

The selected solutions, once randomly selected, have been manually refactored (whilst ensuring that their complexity and original makeup is maintained as much as possible to maintain authenticity) to conform to the AWS Lambda function handler schema. Once refactored, the function's source code and example inputs are passed to the Resource Manager within a *POST /function* HTTP request to create the functions within Heterogeneous FaaS.

An example of an original function prior to refactor is:

```
import sys

def digit_check(n):
    digit = 1

    while int(n/(10**digit)) != 0:
        digit += 1

    return digit

def main():
    for line in sys.stdin:
        ls = list(map(int, line.split(' ')))
        print(digit_check(ls[0]+ls[1]))

if __name__ == '__main__':
    main()
```

Compared to the same function post-refactor to a Lambda function, with some modifications to accept a Lambda event input as opposed to using OS IO as the function's input:

```
def digit_check(n):
    digit = 1

    while int(n/(10**digit)) != 0:
```

```
            digit += 1

        return digit

    def main(input):
        results = []
        for line in input:
            ls = list(map(int, line.split(' ')))
            results.append(digit_check(ls[0]+ls[1]))

        return results

    def handler(event, context):
        return main(event)
```

Each solution also requires test input data. For each function, a set of 1000 random input values have been generated and provided to the Resource Manager upon function creation. These inputs are then, if transfer learning is not being used, used to generate training data with an incrementally larger subset of the provided inputs (i.e. one more input per invocation).

5.3 Evaluation objectives

The following evaluation objectives have been identified to determine the effectiveness of the Heterogeneous FaaS solution:

1. evaluation of machine learning
    (a) how well do models' predicted execution times compare to actual execution times of functions with both seen and (crucially) unseen input sizes?
2. evaluation of transfer learning
    (a) how does the latency of invoking functions created via. transfer learning compare to the latency of the same functions created by collecting their own training data?

5.4 Evaluation of machine learning

In order to evaluate objective 1(a), it's necessary to evaluate the accuracy of Heterogeneous FaaS' machine learning models' predictions against actual results.

The accuracy of the Learning Manager's regression models predictions have been evaluated by invoking functions with the seen (from training data) input sizes of 10, 50, 500, 1000, and then unseen input sizes of 1001, 1100 and 1250 and 2000. It's important to test on both seen and unseen input sizes, as there are many variable factors which can impact the execution time of a function at any one time and it's important that predictions are able to withstand such variance at any input size. At each input size, each function has been passed identical inputs.

For each invocation, the predicted execution time returned by the Learning Manager for each worker (AWS and Kubernetes) has been compared to the actual invocation time.

Each function has been invoked 100 times for each input size using Apache JMeter [50]. Invocations have been carried out incrementally as part of this evaluation to isolate the evaluation to the effectiveness of the machine learning alone. The evaluation has also been carried out firstly with incremental learning disabled, followed by the same evaluation with incremental learning enabled to review the impact incremental learning has on the accuracy of predictions across a wider set of input sizes.

Figure 5.1 visualises the accuracy of predictions on seen input sizes on AWS, and Figure 5.2 the accuracy of predictions on unseen input sizes on AWS. All of said data has been collected with no incremental learning performed. Each function is represented by its own colour key.

It can be seen from the range of actual durations present in all data, but more so in the case with unseen input sizes, that the range of execution times is in some cases considerable. This presents a clear challenge to effective machine learning, especially linear models such as linear regression, which is reliant on output variables (i.e. actual durations) remaining similar to allow for accurate predictions. An observable trend is that as predicted duration (and it can be assumed input size) increases, so does the range of actual durations. This is an interesting finding, as it demonstrates that with larger function inputs not only does execution time increase, but it also becomes more liable to change per invocation; network latency could also potentially be a factor at play, with larger input sizes resulting in larger HTTP request payloads, however at this scale it's unlikely that the differences in transmission sizes will have any significant impact.

There are also clear outliers skewing the scale of the data, with some executions taking significantly longer than the majority; this is likely due to the cold start problem. Outliers like this can be considered somewhat acceptable given the context of a FaaS environment, whereby there are many variable factors on execution time, such as network latency and resource utilisation on a worker.

Despite the range of durations, predicted durations, especially on seen input sizes, have been found to be roughly in-line with actual durations, excluding outliers. The median actual durations are roughly in line with predicted durations in

the majority of cases, especially on seen input sizes.

On unseen input sizes, the accuracy of predictions is noticeably worse than with seen input sizes, suggesting potential overfitting to the training data. Similarly to seen input sizes, the median actual durations is roughly in-line with predicted durations at smaller input sizes, with this increasingly less the case as input sizes increase. The range of actual durations is larger than those found on seen input sizes, supporting the previous finding that as input sizes increase execution times become more liable to change on an invocation-by-invocation basis. This is an issue that incremental learning will assist in negating.

Figures 5.3 and 5.4 show execution times on Kubernetes with seen and unseen input sizes respectively. This data has been collected in the same way as the AWS data.

5.5 Evaluation of transfer learning

In order to evaluate objective 2(a), it's necessary to compare the prediction accuracy of models trained by collecting their own new data with models trained via. transfer learning.

This has been done by training a function by collecting a full training dataset, followed by training the same function but training it via. transfer learning using a function identified as being similar by function clustering. This will provide an indication of the difference in accuracy between the two models, as well as indirectly evaluate the reliability of function clustering in identifying similar functions.

Within this evaluation the AWS worker has been used as it provides the most reliable machine learning models.

Figure 5.5 shows the accuracy of the function's model trained following the standard flow of collecting a full training data, followed by Figure 5.6 which shows the same function with a model transferred from a function identified as being similar (a function belonging to the same cluster as the function).

Another important measure of the effectiveness of transfer learning is its speed. Transfer learning only becomes useful when its training time is noticeably lower than the standard machine learning workflow of collecting a full dataset of training data. This is by far the case, with the collection of training data by the Resource Manager alone taking an average of around 2 minutes to complete on both workers, and transfer learning taking seconds.

5.6 Discussion of findings

Overall, machine learning has been found to be somewhat effective, moreso on AWS where the data has shown more normal (Gaussian) distributions and therefore heeded more accurate predictions. The data of Kubernetes is an issue which, once addressed, as discussed in Section 6, will greatly improve its model's accuracy to at least be similar to AWS' accuracy. There is, however, further work to be done on the models of both workers, especially around accommodating a range of different functions with different execution profiles.

Transfer learning has been found to be similarly somewhat effective. Of course the reliability of transfer learning is heavily dependent on the effectiveness of function clustering and the reliability of the feature vectors used to do so in identifying the execution profile and complexity of functions.

Figure 5.1: AWS prediction accuracy on seen input sizes (predicted duration vs. actual duration) with no incremental learning

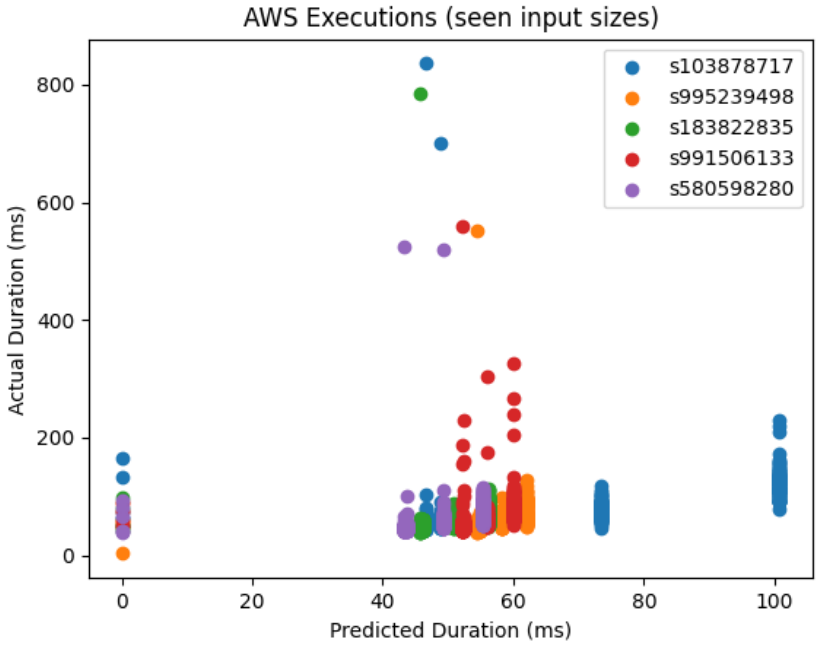


Figure 5.2: AWS prediction accuracy on unseen input sizes (predicted duration vs. actual duration) with no incremental learning

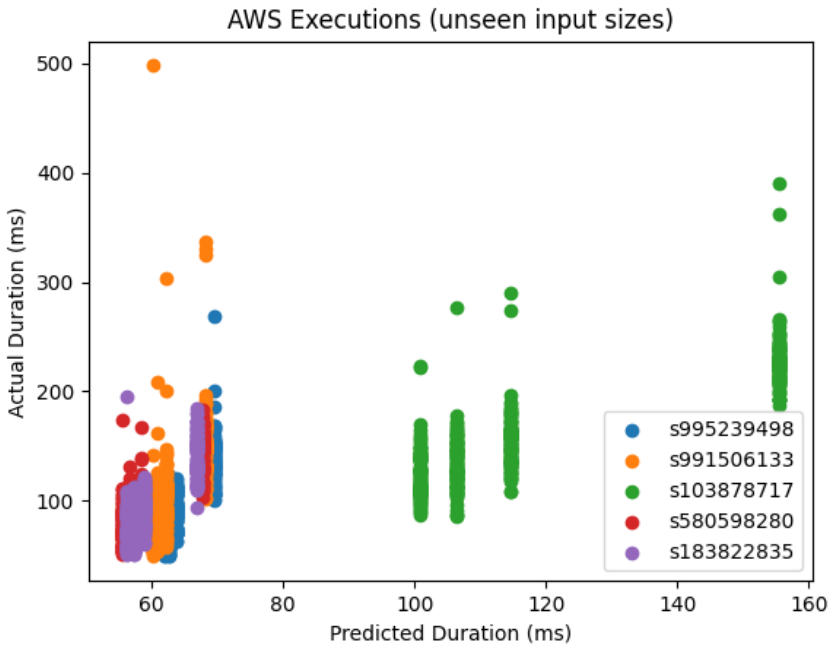


Figure 5.3: Kubernetes prediction accuracy on seen input sizes (predicted duration vs. actual duration) with no incremental learning

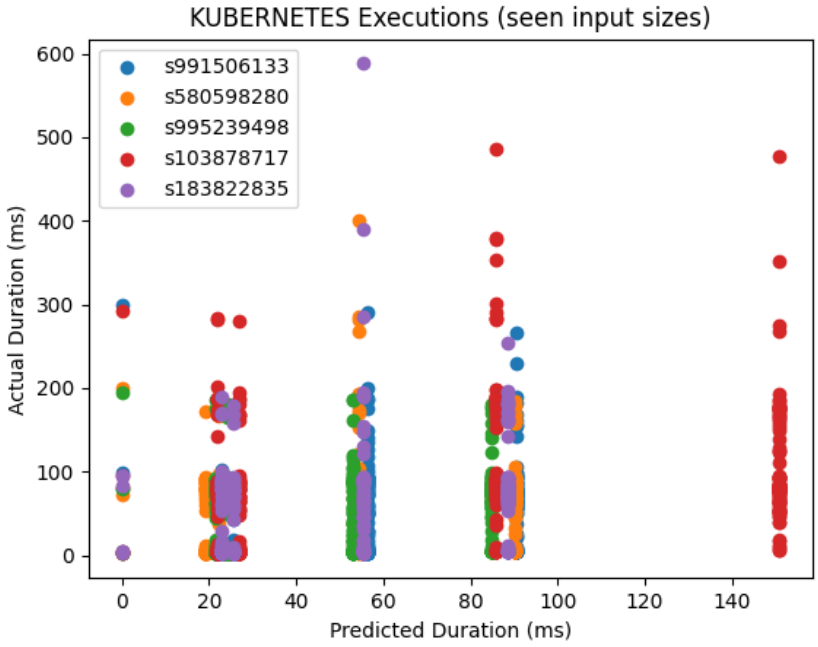


Figure 5.4: Kubernetes prediction accuracy on unseen input sizes (predicted duration vs. actual duration) with no incremental learning

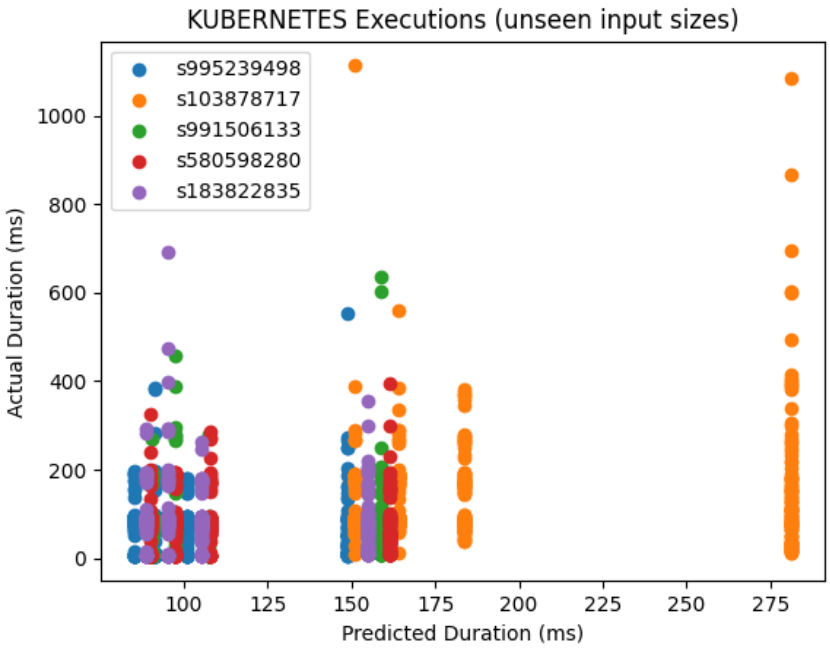


Figure 5.5: AWS prediction accuracy on seen input sizes (predicted duration vs. actual duration) on own training data

Figure 5.6: AWS prediction accuracy on unseen input sizes (predicted duration vs. actual duration) on own training data

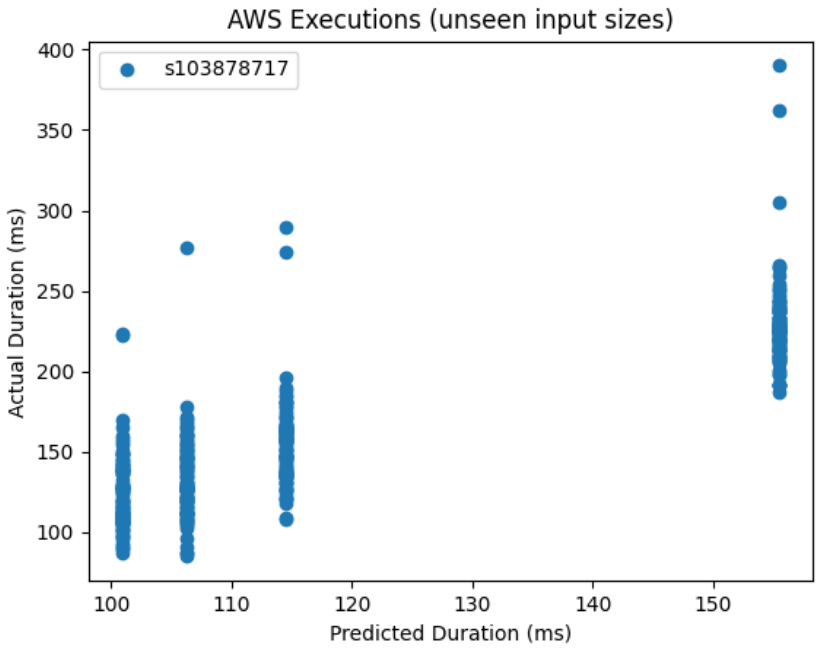


Figure 5.7: AWS prediction accuracy on seen input sizes (predicted duration vs. actual duration) after transfer learning

Figure 5.8: AWS prediction accuracy on unseen input sizes (predicted duration vs. actual duration) after transfer learning

Chapter 6

Conclusion

This chapter will retrospectively review the project, its achievements and key findings. It will then lay out any possible future directions to further the contribution of this work, as well as related works. Finally, the overall outcome of the project will be discussed as final remarks.

6.1 Review of aims

In this section, I will refer back to the initial aims and objectives of the project, as set-out in Section 1.2, and review challenges in achieving each, as well as how well each has been achieved at the conclusion of the project.

6.1.1 Implementation of the Resource Manager

Objective 1 is concerned with the implementation of the ”Resource Manager”, the details of which can be found in Section 4.2.

The Resource Manager references many external dependencies, and is therefore heavily reliant on the use of APIs and their associated libraries. Choosing Java was a wise decision given its popularity, especially in the area of the project, and the wide availability of suitable libraries that comes with that. However, as the project progressed I found that libraries covering key components such as Docker suffered from a lack of official libraries, meaning that some unofficial libraries could lag behind changes to the Docker API itself. This, alongside the early design decision to avoid using a Java framework such as Spring [52] with the aim of ensuring a lightweight solution minimising bloat also potentially brought more drawbacks than benefits during development, with the development of, for instance, the HTTP API to be asynchronous and the handling of database mapping,

while necessary, detracting valuable development time from other more research-oriented elements of the solution.

Overall, the end result of the Resource Manager is a performant, elastic, highly extensible orchestration manager which is able to handle load well within the solution and satisfies its initial objectives.

6.1.2 Implementation of machine learning (Learning Manager)

Objective 2 is concerned with the implementation of machine learning within the ”Learning Manager”, the details of which can be found in Sections 4.3.3 and 4.3.4.

The Learning Manager has been successfully implemented.

6.1.3 Implementation of function clustering and transfer learning (Learning Manager)

Objective 3 is concerned with the implementation of function comparison (or clustering) and transfer learning within the ”Learning Manager”, the details of which can be found in Sections 4.3.5 and 4.3.6.

Function clustering has been successfully implemented, as has transfer learning.

6.1.4 Evaluation of the solution

Objective 4 is concerned with the evaluation of the solution and each of its key aspects, being machine learning, transfer learning and the elasticity and performance of the overall solution.

Although the Project CodeNet functions used for evaluation provide benefit in that they are authentic functions written by a range of developers, their small workload nature has meant that in some cases effective evaluation hasn't been possible. These functions are also very stateless in nature in that they don't read or write any data from external sources, which isn't necessarily representative of the functions deployed to FaaS in a real-world scenario; this is however a limitation of the solution as a whole, supporting only stateless functions, as well as the CodeNet dataset.

Overall, despite dataset challenges, evaluation of all aspects of the solution has been performed and has returned results which present interesting findings to be discussed.

6.2 Future directions

Heterogenous FaaS has laid the groundwork for an effective orchestration system utilising machine and transfer learning to augment its decision making. With further research into the best methods of data collection and parameters for the machine and transfer learning aspects of the solution, I believe that a further effective solution is in close sight.

Many of the issues encountered with data stem from a lack of consistency in execution times, particularly within Kubernetes. Effective machine learning, especially regression, is heavily reliant on normally distributed data. Despite measures being available and taken as part of the machine learning data cleaning process to ensure such data, in the cases of some function's models the data was so random in nature that these measures proved ineffective. Any future work should address this at root by further exploring in more detail what influences said execution times, besides issues such as the cold start problem which have been discussed and mitigated within this project. If Kubernetes execution times can be made more consistent, of course without any significant cost in terms of added latency added, the machine learning elements of the solution will become far more accurate in serving predictions.

The transfer learning aspect of the solution is also heavily influenced by data, in this case the features provided during function clustering. Following on from related works in the area of algorithm comparison, CPU and memory usage as well as raw execution times were utilised within this project, however I believe interesting findings lie within source code analysis, which involves exploring the time and memory complexity among other properties at source code level, as opposed to only at runtime level. This work would be novel in nature, especially within a FaaS context, and has the scope to provide a great contribution upon this work and the area in general.

6.3 Final remarks

Overall, despite some downfalls in data, the project has been a success and has presented some findings. A novel implementation of transfer learning has been successfully implemented, and a basis for improved accuracy of execution time predictions to aid effective orchestration of a FaaS environment across the edge and cloud using a combination of this and machine learning has been laid out.

The project has also been of great personal benefit to me, granting me valuable insight into exactly what a research project entails. I'd say my key takeaway is to plan, plan, plan, as I'm sure my supervisor could've advised me (and probably did) right from the beginning. Looking back it's also apparent how broad the project

has become with many elements involved, and thus enforcing a more narrowed scope may have heeded improved results overall. If I was to undertake such a project again (after a long rest) I would take an entirely different approach, which I believe is testament to the valuable experience I've gained while undertaking the project.

I hope you've enjoyed reviewing my project as much as I (think) I've enjoyed undertaking it.

References

[1] J. Nupponen and D. Taibi, "Serverless: What it is, what to do and what not to do," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 49–50. DOI: 10.1109/ICSA-C50368.2020.00016.

[2] V. Yussupov, U. Breitenbücher, F. Leymann, and M. Wurster, "A systematic mapping study on engineering function-as-a-service platforms and tools," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC'19, Auckland, New Zealand: Association for Computing Machinery, 2019, pp. 229–240, ISBN: 9781450368940. DOI: 10.1145/3344341.3368803. [Online]. Available: https://doi.org/10.1145/3344341.3368803.

[3] S. K. R and J. Lakshmi, "Qos aware faas platform," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021, pp. 812–819. DOI: 10.1109/CCGrid51090.2021.00099.

[4] A. W. Services. "Aws lambda pricing." (2023), [Online]. Available: https://aws.amazon.com/lambda/pricing (visited on 02/24/2023).

[5] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, IEEE, 2017, pp. 405–410.

[6] E. Jonas, J. Schleier-Smith, V. Sreekanti, *et al.*, *Cloud programming simplified: A berkeley view on serverless computing*, 2019. DOI: 10.48550/ARXIV.1902.03383. [Online]. Available: https://arxiv.org/abs/1902.03383.

[7] Y. Li, Y. Lin, Y. Wang, K. Ye, and C.-Z. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," *IEEE Transactions on Services Computing*, pp. 1–1, 2022. DOI: 10.1109/TSC.2022.3166553.

[8] R. Mahmud, K. Ramamohanarao, and R. Buyya, "Application management in fog computing environments: A taxonomy, review and future directions," *ACM Comput. Surv.*, vol. 53, no. 4, Jul. 2020, ISSN: 0360-0300. DOI: 10.1145/3403955. [Online]. Available: https://doi.org/10.1145/3403955.

[9] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, *et al.*, "Serverless edge computing: Vision and challenges," in *2021 Australasian Computer Science Week Multiconference*, ser. ACSW '21, Dunedin, New Zealand: Association for Computing Machinery, 2021, ISBN: 9781450389563. DOI: 10.1145/3437378.3444367. [Online]. Available: https://doi.org/10.1145/3437378.3444367.

[10] G. Kousiouris and D. Kyriazis, "Functionalities, challenges and enablers for a generalized faas based architecture as the realizer of cloud/edge continuum interplay.," in *CLOSER*, 2021, pp. 199–206.

[11] A. P. Rajan, "A review on serverless architectures-function as a service (faas) in cloud computing," *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, vol. 18, no. 1, pp. 530–537, 2020.

[12] A. W. Services. "Serverless architectures with aws lambda." (2014), [Online]. Available: https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf (visited on 02/20/2023).

[13] Microsoft. "Introduction to azure functions." (2016), [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview (visited on 02/20/2023).

[14] Google. "Google cloud functions." (2016), [Online]. Available: https://cloud.google.com/functions (visited on 02/20/2023).

[15] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, and F. Leymann, "Faasten your decisions: A classification framework and technology review of function-as-a-service platforms," *Journal of Systems and Software*, vol. 175, p. 110 906, 2021, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2021.110906. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221000030.

[16] A. S. Foundation. "Apache openwhisk." (2023), [Online]. Available: https://openwhisk.apache.org (visited on 03/23/2023).

[17] IBM. "Ibm cloud functions." (2023), [Online]. Available: https://cloud.ibm.com/functions (visited on 03/23/2023).

[18] OpenFaaS. "Openfaas." (2023), [Online]. Available: https://www.openfaas.com (visited on 03/23/2023).

[19] T. K. Authors. “Knative.” (2023), [Online]. Available: https://knative.dev (visited on 03/23/2023).

[20] T. K. Authors. “Kubernetes.” (2023), [Online]. Available: https://kubernetes.io (visited on 03/23/2023).

[21] H. Korala, D. Georgakopoulos, P. P. Jayaraman, and A. Yavari, “Managing time-sensitive iot applications via dynamic application task distribution and adaptation,” *Remote Sensing*, vol. 13, no. 20, 2021, ISSN: 2072-4292. DOI: 10.3390/rs13204148. [Online]. Available: https://www.mdpi.com/2072-4292/13/20/4148.

[22] A. Das, S. Imai, S. Patterson, and M. P. Wittie, “Performance optimization for edge-cloud serverless platforms via dynamic task placement,” in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 41–50. DOI: 10.1109/CCGrid49817.2020.00-89.

[23] K. R. Sheshadri and J. Lakshmi, “Qos aware faas for heterogeneous edge-cloud continuum,” in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, 2022, pp. 70–80. DOI: 10.1109/CLOUD55607.2022.00023.

[24] K. Trejos, L. Rincón, M. Bolaños, J. Fallas, and L. Marın, “2d slam algorithms characterization, calibration, and comparison considering pose error, map accuracy as well as cpu and memory usage,” *Sensors*, vol. 22, no. 18, p. 6903, 2022.

[25] D.-T. Ngo and H.-A. Pham, “Towards a framework for slam performance investigation on mobile robots,” in *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, 2020, pp. 110–115. DOI: 10.1109/ICTC49870.2020.9289428.

[26] Y. Zhang, T. Zhang, and S. Huang, “Comparison of ekf based slam and optimization based slam algorithms,” in *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, 2018, pp. 1308–1313. DOI: 10.1109/ICIEA.2018.8397911.

[27] A. W. Services. “Aws lambda base container images.” (2020), [Online]. Available: https://github.com/aws/aws-lambda-base-images (visited on 02/24/2023).

[28] A. W. Services. “Lambda runtimes.” (2023), [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html (visited on 03/10/2023).

[29] Knative. “Knative serving overview.” (2023), [Online]. Available: https://knative.dev/docs/serving (visited on 02/24/2023).

[30] A. W. Services. "Amazon elastic container registry." (2023), [Online]. Available: https://aws.amazon.com/ecr (visited on 02/24/2023).

[31] Statista. "Global quarterly market share of cloud infrastructure services from 2017 to 2022, by vendor." (2022), [Online]. Available: https://www.statista.com/statistics/477277/cloud-infrastructure-services-market-share (visited on 03/01/2023).

[32] A. W. Services. "Aws sdk for java." (2023), [Online]. Available: https://aws.amazon.com/sdk-for-java (visited on 03/06/2023).

[33] A. Jeffery, H. Howard, and R. Mortier, "Rearchitecting kubernetes for the edge," in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, ser. EdgeSys '21, Online, United Kingdom: Association for Computing Machinery, 2021, pp. 7–12, ISBN: 9781450382915. DOI: 10.1145/3434770.3459730. [Online]. Available: https://doi.org/10.1145/3434770.3459730.

[34] C. N. C. Foundation. "Kubeedge." (2023), [Online]. Available: https://kubeedge.io/en (visited on 03/01/2023).

[35] T. K. Authors. "Kubernetes services." (2023), [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service (visited on 03/01/2023).

[36] G. Inc. "Gradle build tool." (2023), [Online]. Available: https://gradle.org (visited on 03/10/2023).

[37] DataStax. "Datastax java driver for apache cassandra." (2023), [Online]. Available: https://github.com/datastax/java-driver (visited on 03/06/2023).

[38] docker-java. "Docker-java/docker-java." (2023), [Online]. Available: https://github.com/docker-java/docker-java (visited on 03/15/2023).

[39] R. JBoss Community. "Undertow." (2023), [Online]. Available: https://undertow.io (visited on 03/20/2023).

[40] Google. "Gson: A java serialization/deserialization library to convert java objects into json and back." (2023), [Online]. Available: https://github.com/google/gson (visited on 03/10/2023).

[41] fabric8io. "Kubernetes and openshift java client." (2023), [Online]. Available: https://github.com/fabric8io/kubernetes-client (visited on 03/06/2023).

[42] T. K. Authors. “Resource management for pods and containers - kubernetes.” (2023), [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers (visited on 03/14/2023).

[43] T. K. Authors. “Kubernetes metrics api.” (2023), [Online]. Available: https://github.com/kubernetes/metrics (visited on 03/22/2023).

[44] Pallets. “Quart.” (2023), [Online]. Available: https://github.com/pallets/quart (visited on 03/21/2023).

[45] scikit-learn. “Scikit-learn.” (2023), [Online]. Available: https://scikit-learn.org (visited on 03/22/2023).

[46] NumPy. “Numpy.” (2023), [Online]. Available: https://numpy.org (visited on 03/22/2023).

[47] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, “Cold start influencing factors in function as a service,” in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 181–188. DOI: 10.1109/UCC-Companion.2018.00054.

[48] P. S. Foundation. “The python profilers.” (2023), [Online]. Available: https://docs.python.org/3/library/profile.html#module-cProfile (visited on 03/24/2023).

[49] T. K. Authors. “Kind.” (2023), [Online]. Available: https://kind.sigs.k8s.io (visited on 03/02/2023).

[50] A. S. Foundation. “Apache jmeter.” (2023), [Online]. Available: https://jmeter.apache.org (visited on 03/19/2023).

[51] IBM. “Project codenet.” (2023), [Online]. Available: https://github.com/IBM/Project_CodeNet (visited on 03/09/2023).

[52] V. Tanzu. “Spring.” (2023), [Online]. Available: https://spring.io (visited on 03/24/2023).

Appendix A

Project Proposal

A.1 Project outline (from supervisor)

Serverless computing, also known as Function as a Service (FaaS), is a new cloud computing paradigm that is increasingly getting a momentum in the cloud. The paradigm shifts the burden of server management from developers to provider platforms. Developers just need to develop their functions and submit them to the cloud to be managed in an elastic way to deal with changes in workload at runtime. Edge computing is a paradigm where compute nodes are moved in close proximity to the users. The advantage is to bring computation closer to where data is being generated and thus reducing latency and bandwidth usage; compared to sending data for processing in the cloud. However, the computational capacity of the edge is limited and, in some cases, is not able to satisfy the requirements of certain computation tasks. This project is about investigating the advancement of the above in a serverless edge. This would provide the advantages of reducing latencies for FaaS applications in addition to reducing bandwidth usage. The project is specifically to develop a method for switching between the edge computing nodes and the cloud FaaS platforms.

A.2 Introduction

A new server architecture popularised as "serverless" has emerged in recent years, primarily through large providers such as Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft Azure [1] One type of cloud offering from these providers is FaaS (function as a service), which allows application developers to write their code as functions in a number of popular languages and simply deploy it, without concerning themselves with the infrastructure it'll be hosted on as they, or a DevOps/ITOps team, would previously. These functions are then

deployed to a highly available but generalised cloud environment which can scale and descale in real-time based on current load, with infrastructure only running when the function is actually being executed, and the application developer being billed as such. Many businesses are now switching to this method of hosting their applications due to the lowered need for ITOps teams to manage infrastructure, and an even further lowered cost from only being billed for a function's execution time, compared to having to constantly run a server (using either PaaS or on-premise servers) or a container (CaaS) previously.

There are, however, some barriers to businesses adopting cloud offerings such as FaaS; with one of the most noticeable in 2022 being a lack of integration with other application dependencies [2] This issue can be tackled by adopting a solution labelled as the "hybrid cloud" [3], which typically involves one application being deployed across several cloud providers – this could however be extended to include a "hybrid" of edge and cloud servers, thus allowing the low latency (due to their geographic proximity) and more specialised edge environments to be utilised when it makes technical sense to do so, and the lower cost, more scalable and highly available cloud environments being used otherwise [4]. The barriers to adopting a hybrid edge-cloud solution like this can be heavily reduced by utilising commonly used developer tools, such as IaC tools, to abstract away much of the complexity and providing a means of automatically orchestrating, with minimal overhead, the distribution of requests between edge servers and cloud providers.

The edge-cloud hybrid (sometimes called the "serverless edge" [5]) could be used in a number of different domains, but is particularly tailored to Internet of Things (IoT) devices, which often perform operations which are very latency-sensitive [5] and will therefore take advantage of edge servers. One common service architecture is providing "background as a service" (BaaS) to IoT devices, with edge servers utilising their higher computational power for higher workload tasks, and then taking advantage of low latency to quickly return the results to the IoT device [6]. There are also energy-efficiency benefits to using edge devices with lower power consumption as opposed to cloud services in large datacentres.

A.3 Related work

There have been a number of academic research papers which have addressed extending serverless solutions to the edge in recent years, with a majority being focused on providing solutions for IoT devices. "Toward Distributed Computing Environments with Serverless Solutions in Edge Systems" [7] addresses the challenges of implementing a FaaS infrastructure in an edge environment, and how to best orchestrate each request to the optimal worker (where the function is to be executed). It compares the performance and suitability of a number of assignment

methods and architectures for serverless edge environments, but doesn't consider at all using the cloud as a fallback to the edge infrastructure; neither does it consider the requirements of persisting/receiving data to/from a remote data source (such as a database in the cloud) and assumes that each function will be stateless, which may not always be the case for data-dependent IoT applications.

"Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement" [8] aims to minimise costs of running functions on AWS Lambda by reducing execution time and memory usage, with one of its methods of doing so being to utilise AWS Greengrass to run Lambda functions on an edge (IoT) device and split functions between these and the cloud. This paper offers interesting insight to an example of a "hybrid" edge-cloud architecture in AWS and considerations such as communication between edge servers and cloud services such as S3 for object storage, but its focus is primarily on reducing cost rather than maximising performance, and is very specific to AWS – it doesn't provide a way of dynamically allocating functions between the edge and cloud and results in vendor lock-in to AWS.

"Orchestration of Optical Networks and Cloud/Edge Computing for IoT Services" [9] proposes an architecture focused on processing analytics data from IoT devices that takes advantage of low-latency edge servers, only sending the necessary data (which will require higher workload processing) to cloud servers for additional processing. This implementation is very tailored towards analytics operations, which by their asynchronous and incremental nature can perform some of an operation on the edge and then handover larger tasks to the cloud – this may not always be the case for operations outside of an analytics scope.

In an older paper from when edge (and fog) computing was in its early stages, "Fog computing — Network based cloud computing" [10], lays out an attempt at implementing a fog computing architecture using a Raspberry Pi as an edge server which processes "tagged" packets at the edge and "untagged" packets at the cloud as usual. Here, the distribution of requests between edge and cloud is dictated by the client (by whether it "tags" a packet or not), and there is no centralised orchestration of request distribution.

Another older paper, "Vehicle control system coordinated between cloud and mobile edge computing" [11], presents a system whereby a vehicle controller is migrated between edge and cloud servers without loss of service to the client (in this case an autonomous car). In the presented architecture, the controller running in the edge orchestrates if the edge or cloud should be used based on current network conditions (i.e. the latency between the edge and cloud servers) – if the cloud is selected the edge server will simply act as a proxy for any packets, forwarding client requests to the cloud and then server responses back to the client. This paper offers a solution for a microservice architecture, whereby there's a single application that can be replicated and both constantly running on both the

edge and cloud, however doesn't support stateless functions that can vary like those accommodated for in a FaaS system.

There are also industry solutions readily available, such as AWS Greengrass [12], that allow for FaaS functions (in this case branded as Lambda functions) to be executed on an IoT device itself whilst still being able to interact with AWS cloud services such as S3 (for static object storage); this offering is different in scope to what I'm trying to achieve as it doesn't provide a mechanism to offload from the edge to the cloud (or vice versa), and relies on the device(s) itself being the edge infrastructure, rather than a separate edge infrastructure being in place between the IoT client devices and the cloud datacentre. It does however provide the ability for edge devices to interact with cloud services for data storage etc.

None of the prototypes or solutions found utilise infrastructure-as-code (IaC) tools (with the exception of AWS Greengrass, which has Terraform and Serverless framework support) or any other means of allowing for easy declaration and management of infrastructure by the application developer.

I have laid out a table of limitations to the related work reviewed: Related work Positive findings Limitations Toward Distributed Computing Environments with Serverless Solutions in Edge Systems • Optimal performance of edge FaaS infrastructure • No edge-cloud migration Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement • Edge-cloud migration (between Greengrass device and Lambda in cloud), but static • Minimising costs • Strong vendor lock- in to AWS • Little focus on performance • No dynamic live edge-cloud migration Orchestration of Optical Networks and Cloud/Edge Computing for IoT Services • Edge-cloud migration (within operations) • Heavily tailored to async operations (such as analytics gathering) • No support for stateless FaaS functions

Fog computing — Network based cloud computing • Distribution of requests (like FaaS functions) between edge and cloud servers • Client-dictated request distribution between edge and cloud (not centralised) Vehicle control system coordinated between cloud and mobile edge computing • Controller-based centralised orchestration of requests (based on network conditions) • Tailored to microservice architecture, not FaaS – only works for applications deployed in advance to edge and cloud Of the works reviewed, each provides different aspects of a desired system, however none provide a complete version of a centrally orchestrated infrastructure distributed across the edge and cloud that supports stateless functions, as is required in FaaS, as I propose. I was also unable to find any examples of an IaC tool being used to manage the infrastructure – many IaC tools, for instance Terraform, provide support for multiple providers (of infrastructure) so this could be utilised.

A.4 Project aims and objectives

The overall aim of my proposed project is to provide a seamless way of setting up and dynamically orchestrating a "serverless edge" setup to application developers, allowing functions to be deployed to an open source FaaS setup on the edge (and theoretically on "fog" servers as well) and cloud simultaneously. Developing the implementation of my project once benchmark metrics have been defined will involve developing software to run on a collection edge servers, which allows the servers, running open source FaaS software such as OpenFaaS or Apache Openwhisk (part of the project will be comparing and assessing the best open source solution), to switch between themselves and a cloud provider with higher computational power if the workload is too much for the edge server to handle, or another factor such as little need for low latency means the workload would be better suited to being handled on a cloud provider with higher availability, rather than unnecessarily hogging resources on a less readily-available edge server.

Once this has been achieved, the aim will be to build a plugin for IaC tools (starting with Serverless and potentially later also adopting Terraform) to offer a single provider that's capable of running function code on the edge servers as well as a cloud provider (such as AWS), with the application developer only needing to define infrastructure once by abstracting details of cloud implementations exclusive to a specific cloud provider from the application developer wherever possible.

A.5 Project plan and methodology

The project will need to be undertaken in a step-by-step linear fashion as each step will be a prerequisite to its successive step. I will adopt an agile method of managing the project, following the wider objectives laid out as follows, splitting each into more granular tasks (similar to a Scrum sprint) as I commence work on each larger step.

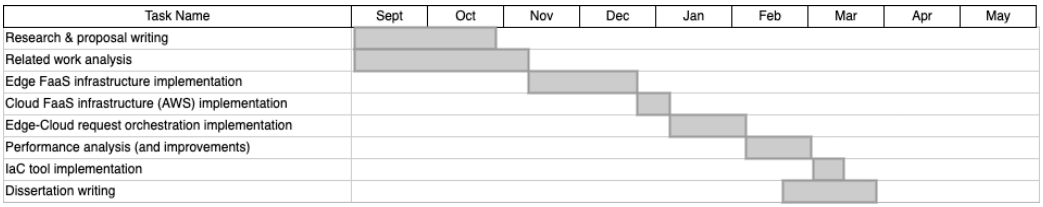
I will first further analyse any related works to find their limitations in terms of performance, such as network latency, energy consumption, costing, level of availability/scalability (mainly of the edge infrastructure) to define metrics to benchmark my solution(s) against. This will also involve exploring potential architectures and assessing the advantages and disadvantages of each. The software to be used to operate the FaaS infrastructure in the edge environment (which will consist of two Raspberry Pi Model Bs) will then need to be established; this will need the ability to containerize each instance of a function running and manage each on each edge server, and likely provide an API to an orchestration service to allow it to spin up/destroy function instances; compatibility with the runtimes of cloud services such as AWS will also need to be carefully considered.

A prototype of a working edge FaaS environment will then need to be developed, with the Raspberry Pi devices acting as a pair of edge servers in the chosen architecture. At this stage considerations such as supported network protocols etc. will need to be made.

Once the edge infrastructure is running, the ability for the orchestrator service to instead direct requests to the cloud will be introduced. It's at this stage that optimisation and distribution policies and algorithms will need to be considered and developed for the orchestrator to follow; it's possible that several algorithms can be implemented to compare their performance. Once the edge-cloud infrastructure is fully operational, its performance in comparison to just using an edge or cloud implementation can be assessed and changes made if necessary to improve results.

Once the implementation of the edge-cloud infrastructure is satisfactory and has been measured, the possibility of developing an extension to an IaC tool (likely Terraform or Serverless) can be explored. The aim here will be to develop a tool which allows for an application developer to define their infrastructure to be deployed on the edge and cloud only once, and it be deployed to both ready to be called (i.e. a function will only need to be declared once and deployed to the edge infrastructure and configured cloud provider).

I have laid out a rough idea of the timings of these overriding tasks in a Gantt chart. The Gantt chart is a indicative estimate of timings and is likely to be deviated from somewhat. For instance, I plan to write parts of my final dissertation incrementally from the beginning of the first project phase (in week 4).



A.6 References

1. IBM State of Multicloud, page 36 (IBM, 2022):
   https://www.ibm.com/downloads/cas/L9K1MK1Y

2. IBM State of Multicloud, page 38 (IBM, 2022):
   https://www.ibm.com/downloads/cas/L9K1MK1Y

3. IBM Cloud Hub, "What is the hybrid cloud?" (IBM, 2021):
   https://www.ibm.com/cloud/learn/hybrid-cloud

4. Edge server architecture: How edge and cloud fit together (The Enterprisers Project, 2021): https://enterprisersproject.com/article/2021/3/edge-server-architecture- how-edge-and-cloud-fit-together

5. Performance optimization of serverless edge computing function offloading based on deep reinforcement learning (Xuyi Yao, Ningjiang Chen, Xuemei Yuan and Pingjie Ou, 2023): https://www.sciencedirect.com/science/article/pii/S0167739X2200293X

6. In-Network Computing With Function as a Service at the Edge (C. Cicconetti, M. Conti and A. Passarella, 2022): https://ieeexplore.ieee.org/document/9869606

7. Toward Distributed Computing Environments with Serverless Solutions in Edge Systems (C. Cicconetti, M. Conti, A. Passarella and D. Sabella, 2020): https://ieeexplore.ieee.org/document/9040261

8. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement (T. Elgamal, A. Sandur, K. Nahrstedt and G. Agha, 2018) https://ieeexplore.ieee.org/document/8567674

9. Orchestration of Optical Networks and Cloud/Edge Computing for IoT Services (R. Muñoz et al., 2019): https://ieeexplore.ieee.org/document/8817786

10. Fog computing — Network based cloud computing (Y. N. Krishnan, C. N. Bhagwat and A. P. Utpat, 2015): https://ieeexplore.ieee.org/abstract/document/7124902

11. Vehicle control system coordinated between cloud and mobile edge computing (K. Sasaki, N. Suzuki, S. Makido and A. Nakao, 2016): https://ieeexplore.ieee.org/document/7749210

12. AWS Greengrass (Amazon Web Services, 2022): https://aws.amazon.com/greengrass/