university of groningen

faculty of science and engineering

SG Papertronics

Dynamic Vertical Serverless Function Deployment on the Edge

Master Thesis

Author:
Job Heersink

Supervisors:
Viktoriya Degeler
Mostafa Hadadian
David Bor

University of Groningen
The Netherlands

August 16, 2022

Abstract

Internet of Things (IoT) devices are becoming more and more common in our environment, yet the process of developing programs for these devices remains a challenge. To ease this experience, many research papers have focused on introducing Function as a Service (FaaS), also known as serverless functions, on these edge devices. With FaaS, the developers only need to concern themselves with the core functionality. Everything related to scalability, orchestration and the operating system is taken care of by the FaaS platform. A common issue with FaaS platforms on the edge is the Quality of Service. Services on the cloud can be scaled according to demand, but the edge does not have this flexibility due to limited resources. In this paper, we explore function offloading as a solution to this problem and develop a vertical dynamic serverless function scheduler that will move function execution from the edge to the cloud based on available resources, price and user-defined constraints. By testing the scheduler on a real-world deployment and comparing the results to the static or random placement of functions on the edge and the cloud, we measured a reduction in latency between 13%-20% or a reduction in cost up to 28%.

Contents

1 Introduction **3**

2 Approach **6**
2.1 Scheduler 6
2.2 Automated deployment 7
2.3 Usecase 7

3 Related work **9**
3.1 Challenges in serverless edge computing 9
3.2 Serverless function offloading techniques 10
3.3 Serverless edge deployment platforms 13
3.4 AWS Greengrass 15

4 System Design **17**
4.1 Inside the scheduler 18
4.2 Outside the scheduler 21
4.3 Papertronics deployment 22
4.4 Configuration 24
4.4.1 Lambda configuration 24
4.4.2 Resource intensive configuration 25

5 Methodology **26**
5.1 Data 26
5.1.1 price 28
5.2 Machine learning ensemble 29
5.2.1 Random Forest 30
5.2.2 Extra Trees 30
5.2.3 Gradient Boosting 30
5.2.4 The best regression method 33
5.3 Automatic function deployment infrastructure 34

6 Results **35**
6.1 Pipeline data 35
6.1.1 Original Pipeline 37
6.1.2 Resource intensive pipeline 42
6.2 Machine learning ensemble comparison 48
6.3 Gradient Boosting method metrics 52
6.3.1 Overall Performance 52

6.3.2 Residuals vs fit plot . . . 53
6.3.3 Feature importance . . . 54
6.4 Scheduler performance . . . 57
6.4.1 Original pipeline . . . 57
6.4.2 Resource intensive pipeline . . . 58
6.5 Scheduler overhead . . . 60
6.6 Automated deployment script . . . 61

7 Discussion **63**
7.1 Pipeline . . . 63
7.2 Machine learning model . . . 64
7.3 Scheduler . . . 65
7.4 Automated deployment script . . . 68

8 Future work **70**

9 Conclusion **72**

Appendices **77**

A deployment configuration **78**
A.1 Configuration grammar . . . 78
A.2 Example configuration . . . 80
A.3 Finalizer and scheduler configuration . . . 82

B Machine learning methods performances **84**
B.1 Residuals vs fit plot . . . 84
B.1.1 Original pipeline . . . 84
B.1.2 Resource intensive pipeline . . . 86

C Pipeline data visualized **89**
C.1 Original pipeline . . . 89
C.1.1 resources and message size vs duration . . . 89
C.1.2 network bandwidth, available resources and message size vs transfer times . . . 91
C.1.3 input features vs total pipeline duration . . . 93
C.2 Resource intensive pipeline . . . 94
C.2.1 resources and message size vs duration . . . 94
C.2.2 network bandwidth and message size vs transfer times . . . 96
C.2.3 input features vs total pipeline duration . . . 98

Chapter 1

Introduction

The arrival of cloud computing proved to be fruitful for a great number of industries. It has introduced the concept of "anything" as a service (XaaS) [1], which is an abstraction that allows users to think of computation hardware, infrastructure and software as a utility that can be employed with a subscription. Platform as a Service is still one of the most popular products for developers, which allows them to effortlessly host and publish their applications. However, there are still some nuisances with this product. Namely, not all aspects related to the infrastructure are abstracted to a satisfactory extent and developers still need to provide the entire software package.

To solve this problem, the Function as a Service (FaaS) product, also called serverless functions, was introduced. With FaaS, developers do not need to worry about server management. They do not have to manage underlying operating systems, software and scalability, as this is the responsibility of the FaaS platform. FaaS ensures that these functions automatically scale to zero when they are no longer needed, thus avoiding wasted resources.

With the increasing number of services running in data centers and an increase in the number of devices communicating with these services, an enormous strain is put on the boundary of communication networks, making it more challenging to provide low latency communication to central cloud computing systems [2]. This issue is even more underlined by the fact that many services are latency sensitive, and require results to be processed near real-time.

To address these challenges, the concept of edge computing was brought to light. Edge computing tries to minimize data processing latency by leveraging available computational power close to where the data are generated, sometimes at the source itself [3]. These edge devices can, for example, be used to preprocess or aggregate the data, before the data are sent to the cloud. In some cases, edge computing also offers the benefit of cheaper execution compared to the cloud [4]. With new services like AWS Greengrass [5], it is now already possible for developers to migrate a part of their cloud based serverless functions to the edge. This new technology makes edge computing a viable option to potentially decrease latency or reduce cost for most serverless applications, where latency is measured in terms of the time from the ingestion of the input to the storage of the results in the cloud [6].

To realize this latency reduction, edge devices should be able to process the data in real-time. However, for some applications and workloads, this is not completely possible due to the resource constrained nature of these edge devices. Depending on the situation and available resources, the duration of execution might be shorter when some tasks are dynamically offloaded to the cloud. So the problem is; **How do we determine what tasks should be offloaded to the cloud and what should remain on the edge?**

Another advantage of edge computing is that the execution can be cheaper than on the cloud. For example, AWS Greengrass only charges its users per device, rather than per service invocation. The cost is fixed, no matter how many processes are running. However, in the decision to offload applications to the cloud, the potential increase in cost should also be taken into account. To be able to make the right decisions based on the user's preference, the user should be able to set a constraint on either cost or latency, for which the scheduler will then try to optimize the opposed value. For example, if a user defines a cost constraint, then the scheduler will try to optimize latency and if the user defines a deadline constraint, then the scheduler will try to optimize cost. Therefore we can reformulate our research question as: **How do we determine what tasks should be offloaded to the cloud and what should remain on the edge based on a cost or deadline constraint?**

One disadvantage of edge cloud computing is the fact that setting up a cloud edge infrastructure can be a time-consuming tasks, since a lot of manual work is still required in order to configure and run serverless functions on the edge [7, 8, 9, 10]. With the addition of dynamic function offloading, the time needed to configure a deployment may increase even further. We therefore also try to answer the question: **How do we minimize the time-consuming process of configuring and deploying serverless functions to the cloud and to the edge?**

In summary, in this paper we try to answer the following research questions:

1. How do we determine what tasks should be offloaded to the cloud and what should remain on the edge based on a cost or deadline constraint?
2. How do we minimize the time-consuming process of configuring and deploying serverless functions to the cloud and to the edge?

To answer these research questions, we propose a dynamic vertical serverless edge scheduler that will schedule tasks to be executed on the edge or in the cloud based on predefined user constraints for either duration or cost. In addition to that, we provide a serverless edge-cloud configuration program that will automatically create and update serverless functions with minimal intervention from the developer. Our scheduler is dynamic in the sense that scheduling choices are made as messages come in, instead of when the pipeline is created. In addition to that, our scheduler is vertical in the sense that we only focus on edge to cloud function offloading [7]. Horizontal offloading, or in other words offloading from one edge device to other edge devices, is not discussed in this paper.

We implement our scheduler and automatic function deployer on Amazon Web Services [11] (AWS). More specifically, we create the scheduler as an AWS Greengrass [5] component and run our computing services as AWS Lambda [12] instances. AWS Lambda is Amazon's implementation of a Function as a Service infrastructure and allows developers to easily deploy computing services without great amounts of necessary configuration.

AWS Greengrass on the other hand is an Amazon service that can be installed and run on a range of IoT devices, like a raspberry pi [13], and provides an environment for AWS Lambda functions to be executed. It also provides the ability to securely communicate with other lambda functions running on the edge or other services running in the cloud.

We evaluate and test our scheduler on a deployment for a real-world use case: the Beer-O-Meter [14]. This is a device, created by the company SG Papertronics [15], which is capable of retrieving the properties and characteristics of beer by capturing and analyzing an image of a colorimetric sample of the beer. They can potentially have a great number of these devices scattered over several locations and all the images from these devices need to be analyzed and eventually stored in the cloud. A serverless edge implementation with dynamic function placement is advantageous in this use case.

The remainder of this paper is structured as follows: In Chapter 2 we dive deeper into the problem, the proposed solution and the approach. In chapter 3, we describe the related work of this research, including their findings and open issues. In chapter 4, the system design of our scheduler and Beer-O-Meter deployment is described. In Chapter 5, we describe the methodology of our research. In Chapter 6, the results of our experiments are described. In Chapter 7, these results are discussed and evaluated. In chapter 8, we mention some remaining challenges in the fields and some problems our solution was not able to solve. Finally, in chapter 9, we conclude this paper.

Chapter 2

Approach

We propose to create a vertical dynamic serverless function scheduling system that decreases the cost or latency of cloud/edge based systems by scheduling tasks based on user-defined cost or deadline constraints.

2.1 Scheduler

The proposed scheduler consists of an ensemble machine learning method that predicts the individual lambda functions duration, transfer time to the lambda function and total remaining pipeline duration for both the edge and the cloud. This machine learning implementation is trained on incoming message size, available resources like CPU and memory, and current network bandwidth.

Based on these predicted values, the scheduler will decide where to execute the function based on a deadline or cost constraint. If a deadline constraint is set, the scheduler will try to minimize cost while keeping execution duration under the deadline. If a cost constraint is set, the scheduler will try to minimize latency while keeping the cost under the specified constraint.

The machine learning model needs to train itself on a number of data points before it can provide accurate results. In order to do so, we utilize an exploration vs exploitation strategy [16]: When the machine learning model is untrained or performs poorly, more exploration moves are made by randomly scheduling tasks and analyzing the performance. The machine learning model will then be retrained on this data. When the machine learning model is trained and is performing relatively accurately, the exploitation move is made and tasks are scheduled according to the output of the model.

To evaluate the model itself, we gather around 500 data points of performance statistics of the scheduler and the pipeline it is applied to. We train the model on 75% of this data and evaluate the score of the model on the remaining 25%.

To improve fault tolerance and improve the QoS, function execution will automatically be scheduled on the cloud if not enough resources, like CPU and ram, are present on the device. In addition to that, If no or limited internet is available, the function execution will automatically be scheduled on the device itself, regardless of the output of the machine learning model.

Our implementation of the scheduler is written on top of AWS Greengrass [5] and therefore, the deployment used for evaluating the scheduler is created in the Amazon Web Services environment [11].

2.2 Automated deployment

This scheduler comes with a program that should be able to automatically create and update functions on both the cloud and the edge with minimal intervention from the developer. When developing an application or when porting an existing application to work on the edge, the developer will only need to take care of the following:

First of all, the developer should put the existing function in our predefined python wrapper, by adding the line **@serverless_decorator** above their serverless function as can be seen in the following code snippet:

```
@serverless_decorator
def handle(event, context):
    print(f"incoming message {event}")
    return "message received"
```

This wrapper will take care of the differences in communication channels in the cloud and the edge and use the corresponding message channel relative to the placement of the serverless function. This way, the developer does not need to care about where the function is going to be deployed when writing the code. The wrapper also facilitates direct communication with the scheduler if the function is run on the edge, so the developer will not have to configure this communication manually.

Second of all, the developer should provide a configuration of the entire pipeline and the individual lambda functions to create the functions correctly and to inform the scheduler about the structure of the pipeline. We provide a .yaml configuration file for the developer where the configuration can be easily defined and imported by the deployment program. An example of this .yaml file can be seen in Appendix A. Some of the mandatory requirements that need to be set are: The correct AWS credentials, the id of the group of devices to deploy to, a bucket to store the code in, a region to deploy to and a lambda role. In addition to that, the cost or deadline constraint for the scheduler will need to be set.

Each individual lambda function must have one or multiple ”destinations” defined if the lambda function is expected to communicate with other services. When the lambda function has finished its task then it will send the result to the defined destination. Furthermore, the developer can constrain the lambda function on its place of execution by inserting a ”placement” field in the lambda configuration. For example, a lambda function can be deployed on only the edge by setting the placement field to ”edge_only”.

2.3 Usecase

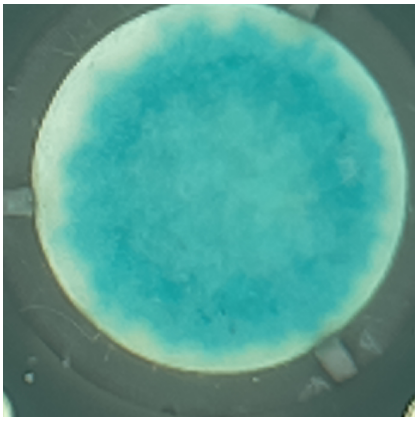
We created a deployment for a real-world use case scenario in the chemistry sector. This deployment is used to test and evaluate the scheduler. It was created for the company SG Papertronics [15] and is tied to their product the Beer-O-Meter [14].

This company is a small startup consisting of about 6-8 people. It was founded in July 2016 and its first product, the Beer-O-Meter, has recently hit the shelves. This device is an encased Raspberrypi 3B [13] with 1 Gigabyte of memory equipped with a camera and LED's, along with some other additional hardware. An illustration of the current state of the device can be seen in figure 2.1a. It has an opening at the top of the case, where a user can insert a "pod" containing their sample of the beer on a piece of specialized paper made for colorimetric tests. The station will proceed to take pictures of this colorimetric sample over time and see how the colors have transposed. An example of such an image taken by the station can be seen in figure 2.1b. After the images have been taken, they will be analyzed and the resulting data will be sent back to the user. Using this approach, the device is able to measure, among other things, the pH, bitterness, calcium, magnesium, chlorium and alcohol percentage in beer.

The main purpose of the Beer-O-Meter is to provide small beer brewers the ability to test their beer on the spot and get their test results back in minutes time. Without the Beer-O-Meter, small brewers would have to send their beer samples to a lab, potentially somewhere across the country, and wait days for the result to come back.

The device in its current state takes pictures of beer samples and sends them to the cloud for processing. Our deployment moves this processing stage to the device itself and only offload this stage to the cloud if it is beneficial for latency or cost. To adhere to the FaaS infrastructure of AWS and keep the deployment consistent, all the functionality of the deployment is implemented as serverless functions.

(a) The Beer-O-Meter



(b) a colorimetry sample

Figure 2.1

Chapter 3

Related work

In this chapter, we list some relevant works related to serverless edge scheduling and some technologies and platforms used to implement and test the scheduling algorithms. For each related work, we provide a short description of their paper, what they tried to solve, what insights they have made and what challenges still remain to be solved. We also provide an explanation of how each work influenced our research and how we incorporated or improved on their research.

A great number of papers have been published concerning the area of serverless (edge) computing in the past years and it still continues to be an active area of research. Many research papers have been trying to solve existing challenges in the area of serverless computing, like the cold-start problem, maintaining the quality of service and lack of debugging tools [17]. Others concerned themselves with creating new frameworks and architectures to optimize the serverless edge platform for certain use cases [7].

3.1 Challenges in serverless edge computing

This section describes some of the open challenges within serverless edge computing, how relevant research tried to solve them and how we plan to tackle or note these challenges.

One of the most prominent challenges in the area of serverless computing is the warm/cold start problem [17]. A warm start of a serverless function implies that the function was already loaded into memory and the environment was already set up, likely due to a recent previous invocation, resulting in relatively low latency. A cold start of a serverless function implies that a new instance of the function needs to be created and this can result in a higher startup time and thus higher latency. A number of researchers have tried to analyze the cold startup problem and proposed solutions to minimize the startup time by reducing the size of the function and its resources or by trying to predict when a function might be invoked so that the cloud provider can start them up beforehand [17, 18, 19, 20].

In addition to that, some papers on serverless edge offloading techniques tried to predict warm and cold starts by analyzing the frequency and interval between requests from a single edge device, and then using this information in the decision of whether or not to offload the function [6]. This technique works well when only one edge device is

operating in the deployment, but when multiple devices operate in the same deployment, the accuracy goes down. This happens because the edge device can only see its own requests made to the cloud. Although some papers show promising results, this matter is still considered an open challenge [17, 7].

Another challenge is maintaining the quality of service of serverless functions on cloud edge deployments [7]. This challenge aims to ensure the proper execution and stability of serverless functions under the most diverse scenarios and mission-critical applications. This challenge is also deeply intertwined with the cold/warm start problem mentioned before. Serverless functions in the cloud try to tackle this challenge by allocating enough resources according to the demand. This helps with handling all requests and prevents system slowdown. However, this is not possible on edge devices, since there are often not enough resources available to scale to. A solution to this would be to offload the serverless functions to the cloud when demand gets too high. In this paper, we try to overcome this QoS challenge by moving the execution of serverless functions to the cloud when the load gets too high or when function offloading is beneficial for latency, according to some predefined user constraint.

Some research papers mentioned a lack of debugging and development tools, as a current challenge of serverless edge computing. As stated by Gustavo Cassel et al. [7]: "It may be tedious or impractical to manually deploy a new version of a function on every single device, depending on the total amount of devices. Simplifying the deployment process is crucial to popularize serverless IoT applications.". In our research, we try to overcome this challenge by automating the deployment process as much as possible for the scheduler as well as the individual serverless functions.

3.2 Serverless function offloading techniques

In this section, we list some relevant papers that also try to implement vertical serverless function offloading techniques. We point out the general techniques used in scheduling the serverless functions, several shortcomings and future work. We also describe how our solution may circumvent the currently existing shortcomings in the implementations presented by the recent literature. To give the most relevant view of the latest research, we only included articles that are not older than 3 years.

Anirban et al. [6] developed a solution for dynamic task placement on the edge. They propose a technique that automatically determines whether to execute the serverless function on the edge device or on the cloud by predicting the execution time of both scenarios on the edge device itself. They propose a scheduler that is located on the edge and takes execution time as well as cost into account. Their implementation is created in AWS [11], but could theoretically be applied to any cloud provider. Although this research has some promising elements, their deployment only works with one single serverless function. In its current state, it does not work with complex chained function deployments, since their scheduling method is implemented inside the single serverless function. This is a problem for several reasons in a deployment with multiple serverless functions:

First of all, each function would have to carry about 200 Mb of additional libraries due to the needed scikit-learn python package, leaving almost no room for additional packages due to the serverless function size limit of 250Mb AWS puts on the entire ZIP archive. Second of all, this method of scheduling can be a strain on memory and storage, since each

function would need to store all the additional libraries for each function as well as load them into memory each time the function is invoked on a cold-start. This can also increase cold-startup times. Last of all, since the code for the scheduler is inside the function codebase, this would mean the scheduling code is also present in the lambda function on the cloud, including the large libraries. This would mean that lambdas running in the cloud could have the same problems with storage and memory as lambda functions running on the edge. A way to solve this would be to create a "cloud" version and an "edge" version of the lambda function, but this would put unnecessary strain on the developer to create two different versions of what is essentially the same functionality.

Another limitation of this research is their proposed scheduling method. They utilize a pre-trained machine learning approach that they create by training on a custom-made data set for every single deployment. This can be rather impractical in a production environment, since the scheduler cannot always be retrained every time the deployment changes. The authors mention improvements to this scheduling method as future work

In this paper, we take the underlying idea of using a random forest-like machine learning algorithm to schedule serverless functions either on the cloud or on the edge, but we propose to detach the scheduling element away from the serverless functions so that the individual functions stay lightweight and maintainable. Instead, we propose to create a separate component for the scheduling functionality. Because of this change, we also change the features and output of the model to be able to predict the total remaining duration as well as single execution time, since predicting only a single lambda duration will not give enough information for scheduling a multi-stage pipeline. To solve the machine learning implementation shortcomings, we retrain the network after a certain amount of messages have passed through the network or if an accuracy lower threshold has been reached, to ensure that the machine learning algorithm stays accurate, even when the deployment or functions changes.

Tarek Elgamal et al. [21] focuses on reducing execution cost by function offloading and function fusion, where they researched the performance increase of placing the function either in the cloud or on the edge as well as combining two functions together. Combining two functions together implies that two functions are deployed as one, and therefore the communication overhead between two functions no longer exist. Just as with many other research papers listed here, this implementation was created for AWS as well, but could theoretically be applied in any cloud environment. Function fusion showed a great reduction in cost: They were able to decrease cost by 37% to 57%, however, the latency also slightly increased by 5%. Although these are promising results, the authors stated that the current implementation is not at all maintainable, since functions need to be manually combined and this can put a strain on the development process. We therefore chose not to pursue function fusion in this research, since we are looking for a maintainable, developer-friendly solution that is able to handle a changing deployment.

István Pelle et al. [10] provides another scheduling technique implemented in AWS and is tested for hybrid cloud as well. The tactic proposed in this paper is not to predict where to place the serverless functions, but to run the deployment first and then evaluate what the most optimal placement would be. If the deployment is not satisfying the specific requirements made by the developer, the deployment is automatically reconfigured.

The authors mentioned the following shortcomings however: The technique uses Amazon cloudwatch to retrieve information about the performance data of edge devices. This can

hinder the proper management of applications requiring low latency, since communication with the cloud is required. A decentralized alternative to this cloudwatch solution would be able to solve this problem. As future work, they mention that replacing CloudWatch with a different option and placing it closer to the edge resource can significantly speed up the process of component offloading.

Duarte Pinto et al. [16] also proposes a function offloading technique as a decentralized solution implemented in AWS. It uses a Bayesian Upper Confidence Bounds algorithm to find the most optimal placement of serverless functions. However, in the placement of these serverless functions, they only consider execution time. Parameters like cost, CPU power, available memory and network bandwidth are completely ignored. They mention in their future work that other parameters, apart from time, may improve the accuracy in the decision process of placing the functions and result in better performance.

George et al. [22] presents an optimized FaaS platform that can be deployed on extremely resourced constrained hardware like embedded systems. They also use storage systems like AWS Simple Storage Service or Google Cloud Storage in an interesting way. They store the binary code of the serverless functions inside a bucket in the cloud, that is dynamically downloaded on the edge device when functions are triggered. This is great for embedded devices that have little to no persistent storage for functions like this, but it can also be a bottleneck to have to download the bytecode of the function every time the function is called. For this reason, we decide to store the latest function on the device and only download the bytecode when the function is updated, rather than pull it from the cloud every function invocation. We assume that the device has enough storage to store the serverless functions, since a serverless function is only a couple of hundred megabytes large and a few gigabytes of storage on persistent storage is relatively inexpensive. This method of function deployment is however a good option for extremely small and resource-constrained IoT devices.

Because of their design for very small IoT devices, they used a relatively simple execution scheduler. This scheduler does not take into account device capability, energy use, battery life, networking or cost. Taking these parameters into account could potentially improve performance. The function code on the edge also had its problems. Some of the python scripts did not work, since a selection of the libraries were written in C and compiled on different hardware. The code and necessary libraries were recompiled and directly retrieved from cloud storage, which means that some libraries would not work on the edge device. A solution would be to either upload multiple versions of the code for different hardware specs or compile the libraries on the device itself when the pipeline is instantiated. For our deployment, we use the latter solution to circumvent this problem.

Chunglae Cho et al. [23] uses reinforcement learning to decide whether to put the function on the cloud or on the edge. They use a decentralized approach where, after the edge device has made a decision to place the function on the edge or the cloud and the function has been executed, a cloud component will evaluate the choice and send feedback back to the edge device to improve future predictions.

3.3 Serverless edge deployment platforms

In this section, we list some existing serverless edge deployment platforms, list their upsides and downsides and elaborate on the decision why AWS Greengrass was selected for our research. The serverless edge frameworks of the most popular cloud service providers among relevant research have been included in this list. An overview of all serverless edge deployment platforms mentioned in this section can be seen in table 3.1. Note that neither Google Cloud functions nor IBM Cloud functions provide the functionality needed to execute their functions on edge nodes [10], therefore these cloud service providers are not included in this list. Also note that, although IBM cloud functions is OpenWhisk based, it does not provide integration with user-managed OpenWhisk deployments and therefore its serverless functions can not be deployed on the edge.

AWS Greengrass
AWS IoT Greengrass [5] is a service provided by Amazon Web Services [11]. Greengrass is by far the most popular option among relevant research, with approximately 7 out of 10 papers utilizing at least AWS Greengrass to test their edge deployments [16, 10, 21, 6, 24, 23, 25]. Greengrass allows the developer to connect their IoT device directly to the AWS cloud infrastructure with little to no configuration. It will, for example, allow for the execution of AWS lambda functions directly on the hardware of the device or in a container without the need for reconfiguration or changes to the code.

Furthermore, since AWS Greengrass version 2.0, the code of the component running on the edge device has been made completely open source. Amazon even allows developers to modify and extend this source code to meet the developers specific software and hardware needs.

There are however still some disadvantages of AWS, as mentioned in the article by Pelle et al. [10]: "While AWS CloudFormation excels at resource setup, it lacks a high-level interface for specifying application components and their intended setup. While monitoring data can be collected at the same centralized location in the cloud, monitoring deployed FaaS code, especially in hybrid edge cloud scenarios is still cumbersome, thus it is also a weakness of the platform."

Fogflow
Fogflow [26] is a completely Opensource edge computing framework capable of running on any Kubernetes-based environment. With the addition of fog functions [27], the platform is also able to schedule and place serverless functions themselves and only requires the developer to provide an initial service topology. The main benefit of Fogflow over any other platform listed here is that is it independent of any service provider like AWS or Google cloud. Since it is purely kubernetes-based, it could be deployed anywhere.

The fact that Fogflow is not tied to any vendor makes it ideal for a hybrid cloud scenario, but also makes it less attractive as a serverless edge solution for industry applications. One of the main advantages of serverless is that little to no configuration should be required from the developer, and only the code should have to be provided. On the other hand, fogflow requires the developer to set up its own Kubernetes cluster, configure each edge component manually and configure the security settings correctly. Furthermore, the developer is required to create its own container for the serverless function, Something that AWS Greengrass and Azure IoT edge already do for the developer.

	Amazon Greengrass	**Azure IoT Edge**	**FogFlow**	**OpenWisk-Lite**
Availability	Made for AWS, but is Open-source	Made for Azure, but is Open-source	Opensource	Opensource
Deployment	Easy, using AWS.	Easy, using Azure	Difficult, using helm+kubernetes.	Difficult, using helm+kubernetes.
Distribute function changes automatically	yes, using a push model	yes, using a push model	yes, using a push model	yes, using a pull model
popularity	+++	+	++	+
cost	via vendor	via vendor	free, but requires kubernetes	free, but requires kubernetes
access functionality via	CLI or UI	UI, CLI or visual studio	UI or API	API or CLI
other		max 50 functions per deployment	has 2 single point of failures [26]: Discovery and Orchastrator component	Still in Beta version

Table 3.1: Serverless edge deployment platforms overview

Another shortcoming of fogflow is the discovery and orchestrator component, which acts as the serverless function scheduler. Currently, these components are not replicated and placed in the central location, resulting in a single point of failure. The article on fog functions even lists this issue as a future work: "The current approach is scalable with hundreds of fog nodes, but it is necessary to decentralize the discovery and orchestration for a much larger scale." [27].

Azure IoT edge
Azure IoT edge [28] is the serverless edge framework provided by Azure. It provides many of the features that AWS IoT Greengrass does, but is more limited in terms of cloud to edge data transfer and scalability [10]. For example, azure only allows for the deployment of a maximum of 60 different functions.

OpenWisk
OpenWisk [29] is originally a serverless platform for cloud-based applications, but with the new extension of OpenWisk lite, is now also able to be deployed on the edge. Just like with fogflow, OpenWisk is open-source and can be deployed on any device that supports Kubernetes. It is still undergoing development and is currently still marked as an early prototype and not ready for production use.

Note that IBM Cloud function is OpenWisk based, however it is not able to deploy functions to the edge because the developer is not able to directly modify the openwisk deployment.

Because of its frequent use in relevant research, available services and support for edge execution, AWS Greengrass has been chosen as our serverless framework for the edge. Although the implementation itself will therefore be tied to only a single cloud provider, the machine learning algorithm and architecture could theoretically also be applied to other cloud vendors as well.

3.4 AWS Greengrass

Now that we established why Greengrass is the most suitable serverless edge framework available at the moment for our use case, we can dive deeper into the architecture of AWS Greengrass itself and how it works. The relevant architecture of AWS Greengrass can be seen in figure 3.1.

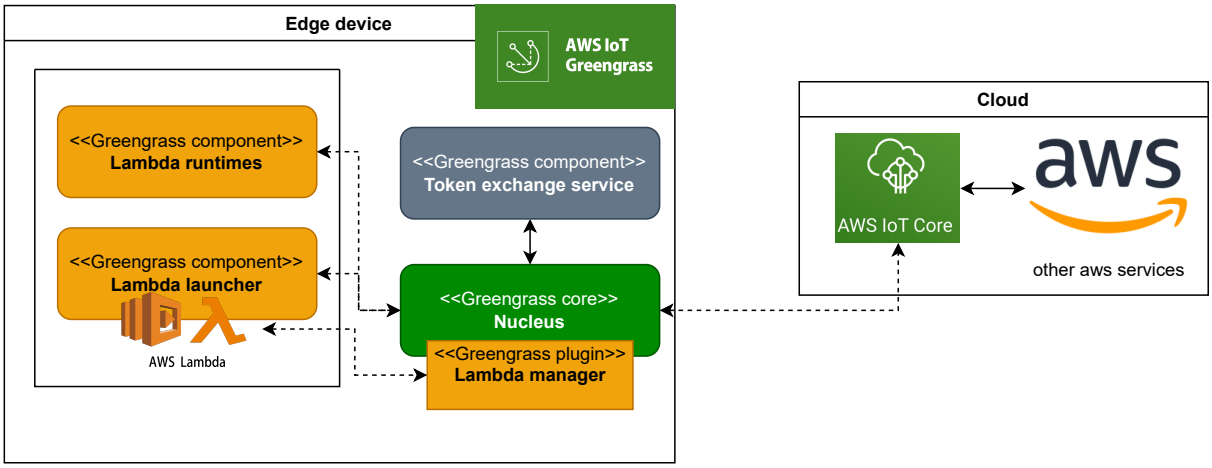


Figure 3.1: The architecture of Greengrass

The architecture of Greengrass consists of a collection of individual isolated processes, also referred to as Greengrass components or simply components, that each has a simple task to perform. The most important component within Greengrass is the Nucleus. The nucleus component is a mandatory component of Greengrass and the minimum requirement to run the AWS IoT Greengrass Core software on a device. This component instantiates and controls all the other components and handles all the interprocess communication between them. One of the inter-process communication channels provided by the nucleus and open to use by any component is a simple publish-subscribe queue. In this paper, we refer to this queue as the local pub/sub queue. The nucleus also provides a method for other components to communicate with the cloud via IoT MQTT. Although with the use of the Token exchange service component, the components themselves are also able to establish a connection with other services of AWS, Like the simple storage service or AWS lambda located in the cloud.

Following the nucleus, there are several components and plugins necessary to run lambda functions on the edge. These are the lambda manager plugin, the lambda runtimes component and the lambda launcher component. The fact that the lambda manager is a plugin and not a component means that this instance runs directly inside the same Java Virtual Machine (JVM) as the nucleus. When a lambda function is deployed to Greengrass, the

lambda manager, lambda runtimes and lambda launcher are automatically installed on the edge device.

The Lambda manager is responsible for porting the interprocess communication and scaling of the Lambda functions to the existing functionality of the nucleus component. It provides a simple communication layer that is able to invoke or scale lambda functions as calls come in.

The Lambda runtimes and lambda launcher components are responsible for setting up the serverless environment. Lambda runtimes provide a set of artifacts to run serverless functions for different runtimes. Current artifacts available are created for python, javascript, java, Go, Ruby and C. The lambda launcher is responsible for creating and maintaining the individual lambda functions as a process and its respective environment configuration. Together with the lambda manager plugin and the Greengrass nucleus, they are the only components required to run lambda functions on the edge.

Apart from these mandatory components to run lambda functions, a number of other optional AWS provided components can also be specified for Greengrass. One such component is the Token exchange service, which is also included in this diagram. This component provides AWS credentials that the developer can use to interact with AWS services. If the Token exchange service is added as a dependency to a component, then this component becomes authenticated to perform certain tasks on the cloud. For example, storing an image in AWS Simple Storage Service (S3).

A range of other optional components created by AWS exists as well, like AWS IoT Device Defender which gives the developer insight into security vulnerabilities of the application, Docker application manager which enables AWS IoT Greengrass to download Docker images from Docker Hub or Greengrass CLI which provides a command-line interface that a developer could use to create deployments on Greengrass and interact with the components. Moreover, the developer is even able to create its own custom components and deploy them within Greengrass.

To deploy a lambda function into Greengrass, the developer will have to create a Greengrass component out of a lambda function. This is not as complicated as it sounds, since AWS does most of the configuration itself. The developer will only have to specify the name of the lambda function and if necessary some optional configuration about the event topics or environment. Once the component is created, it can be attached to a deployment scheme. Once this deployment is applied to an edge device, the device will automatically create the component and start it up.

Apart from the functionality explained above, Greengrass also supports device shadows, a way of managing a devices state through the cloud or the edge, and data streams, a method of streaming high volume data from local sources to the cloud, among other things. This functionality is all optional, requires seperate Greengrass components and is not necessarily part of the core Greengrass functionality. Since we do not utilize these Greengrass services in this paper, they will not be further elaborated and are therefore not included in figure 3.1.

Chapter 4

System Design

In this chapter, we go more in-depth into the system design of the dynamic serverless edge scheduler, and the implementation we use to test the scheduler.

Our implementation consists of 2 main custom components: A scheduler running on each edge device and a finalizer running in the cloud. The scheduler is a custom Greengrass component created to run alongside the serverless functions on Greengrass and the finalizer is a custom lambda function running alongside other lambda functions in the cloud. A diagram of this design can be seen in figure 4.1.

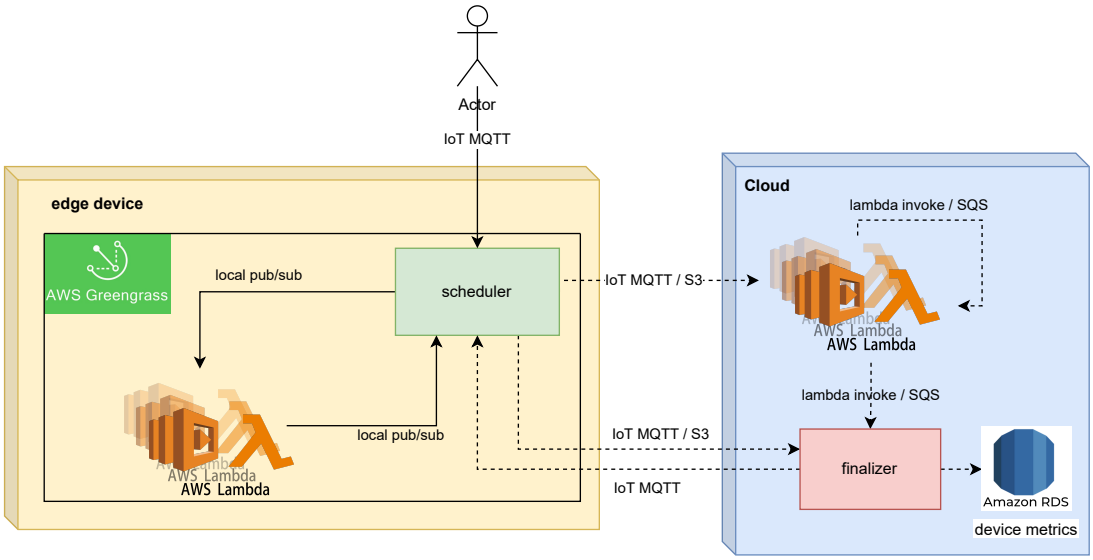


Figure 4.1: Scheduler system design

The main purpose of the scheduler is to receive all incoming and internal messages and decide where to send and handle these messages. For example, when a message is sent to an edge device or an internally running Lamba function publishes a message, the scheduler will have to decide to handle this incoming message internally or send it to the cloud so that it can be handled there. If it chooses to handle it internally, the message will be published to an internal Greengrass pub/sub queue and one of the lambda functions running on the edge device will start working on it. If the scheduler decides on a cloud execution, the message is sent to one of the lambda functions running in the cloud via

the AWS IoT MQTT queue. From there on all further executions are performed without interaction with the scheduler in the cloud.

The main purpose of the finalizer is simply to receive the last message in the pipeline, extract the execution statistics embedded in the message and store these in a database. When needed, the finalizer has the option to retrieve the statistics back from the database and train a new machine learning model on this data. This newly trained model will be stored in AWS S3 and the edge devices can retrieve the new models at their earliest convenience.

We decided to implement the scheduler as a separate component, rather than implementing it inside the individual serverless functions themselves, like what has been done in relevant research [6]. The main advantage of this is: lower startup times for serverless functions and less memory consumption, since each serverless function does not have to load in unnecessary libraries used for the scheduler. In addition to that, this method doesn't clutter the limited lambda size. lambdas can only be 250 MB in size and the required packages for the scheduler will take up 75% of that space, leaving no space for other packages.

There is however one downside to this separation: The scheduler will need to be invoked via an internal queue for each message going from one lambda function to another on the edge, which can result in communication overhead between the functions. After running some experiments, we see that the communication overhead is about 0.2 seconds for messages smaller than 250KB and 0.7-3 seconds for messages between 2 and 10 MB. In most cases, this means that the communication overhead is increased by 10%, compared to if no scheduler would be present. Although this difference is not negligible, it will still perform better and use fewer resources than the previously mentioned alternative [6].

Note that in this design, all lambda functions are present on both the cloud and on each edge device. However, since these are serverless functions, not all of them are running at the same time, the code and configuration of each lambda function simply exist on the device itself, ready to be executed. An alternative solution exists where lambda functions are retrieved from the cloud only when they are needed, but this has a negative impact on performance and is only a viable solution for edge devices with extremely limited storage capabilities [8].

4.1 Inside the scheduler

The scheduler is the main contribution of this research paper, in this section, we therefore look a bit deeper into the inner workings of the scheduler. A diagram of the design of the scheduler can be seen in figure 4.2.

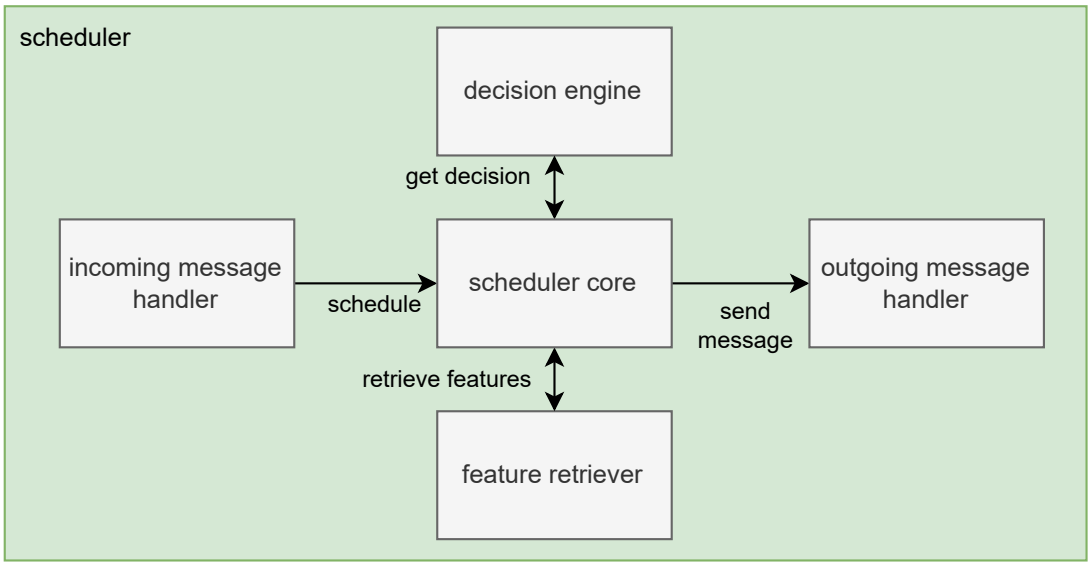


Figure 4.2: Inside the scheduler

The design of the scheduler can be divided into 5 separate "handlers". First of all, we have the **incoming message handler**. This handler is responsible for receiving and reformatting incoming messages. This handler subscribes to a number of AWS IoT MQTT and local pub/sub topics and directs the message to the scheduler core. A full list of the topics the scheduler subscribes to and their purpose can be seen in table 4.1.

topic	**type**	**description**
{s_id}/+	IoT MQTT	Used to invoke any lambda function on the device from the outside. In our case, we only use this endpoint in the form *{s_id}/station_lambda*, since this is the starting point in our pipeline
scheduler/{s_id}/+	IoT MQTT	Used for specific scheduler commands. The '+' sign can be replaced with any of the following options: *type* and *get_model*, used for setting a scheduling method and retrieving the updated models from the cloud respectively.
out/+	Local pub-/sub	Is an internal topic used by the scheduler to receive messages from the internally running lambda functions. The '+' sign can be replaced with any lambda function's name present on the device.

Table 4.1: Scheduler IoT MQTT and local pub/sub topics, where {s_id} is the identification tag of an edge device.

Second of all, we have the **scheduler core**, which acts as an intermediary between the feature retriever and the decision engine. When a message comes into the scheduler, the incoming message handler calls the scheduler core. This component in turn calls the feature retriever first, to retrieve the performance statistics and message characteristics, and then calls the decision engine with the retrieved features to generate an offloading

decision. The incoming message and the offloading decision are then passed onto the outgoing message handler.

Third of all, we have the **feature retriever**. After a message has been received, the scheduler core calls this handler first. The feature retriever fetches up-to-date device metrics like CPU usage, memory usage and network speed. The feature retriever also fetches the size of the incoming message. After the features have been retrieved, they are returned back to the scheduler core.

Fourth of all, the **decision engine** is called given the retrieved features. It applies a machine learning model on the given features and predicts the individual lambda duration for the edge d_e and the cloud d_c, transfer time from the scheduler to the lambda function running on the edge t_e or running in the cloud t_c and total remaining pipeline duration for both the edge r_e and cloud r_c. The input features and output labels of the model are described in detail in section 5.2. Using these predicted values, this handler then calculates the expected price p of executing the remainder of the pipeline in the cloud using equation (5.4). Now the decision engine has the required information to make a scheduling decision based on some user-defined constraint c. The decision engine supports 2 scheduling methods: latency optimization under cost constraint and cost optimization under deadline constraint.

The formula for determining the offload decision for latency optimization according to cost constraint ($) can be seen in equation (4.1). This equation can be interpreted as follows: We offload the lambda function to the cloud if the calculated price is less or equal to the constraint c and either the individual lambdas execution is slower on the edge or the total remaining pipelines duration is slower on the edge. This formula works under the assumption that if the individual lambdas cloud execution + transfer time is faster than edge execution, than the entire pipeline duration on the cloud will be faster as well. Namely, more resources are available on the cloud and large messages will no longer have to be transferred over the internet.

$$\text{offload} \Longleftarrow p \leq c \wedge (d_e + t_e \geq d_c + t_c \vee r_e + t_e \geq r_c + t_c) \tag{4.1}$$

The formula for determining the offload decision for cost optimization according to deadline constraint (sec) can be seen in equation (4.2). Here s is the elapsed time since the start of the request. This equation can be interpreted as follows: We offload the lambda function to the cloud if edge execution no longer satisfies the deadline constraint and the cloud does or individual cloud duration is shorter than edge.

$$\text{offload} \Longleftarrow r_e + t_e \geq c - s \wedge (r_c + t_c \leq c - s \vee d_e + t_e \geq d_c + t_c) \tag{4.2}$$

The decision engine determines the offloading decision based on one of these formulas, depending on what optimization the user has defined beforehand, and returns this decision back to the scheduler core handler. The scheduler core then calls the outgoing message handler and gives it the message and the offloading decision.

Last of all, the **outgoing message handler** is called and sends the messages to either internal lambda functions or lambda functions running in the cloud based on the given offloading decision. The outgoing message handler can do this by either publishing

the message to an internal topic "in/{lambda function name}" or an IoT MQTT topic "cloud/{lambda function name}". Lambda functions on the edge are listening on the former topic and cloud lambda functions on the latter.

4.2 Outside the scheduler

Apart from the scheduler component itself, we also use a number of other AWS resources outside of the scheduler to make it perform as intended. These resources mainly contribute to providing an execution environment for serverless functions on the edge and the cloud, providing communication channels to lambda functions in different locations and providing storage capabilities for device metrics. These services and their relations can be seen in figure 4.1.

The most important service utilized by the scheduler is most likely the AWS Greengrass service [5]. This service is responsible for providing an execution environment for lambda functions on the edge and providing inter-process communication channels on the edge device as well as communication channels to the cloud. It is the host to the scheduler itself and the lambda functions running on the edge. Greengrass provides interprocess edge to edge communication via a local pub/sub queue. Note that with edge to edge communication, we imply the communication of one Greengrass component to another Greengrass component running on the same device. The scheduler does not support the communication channels for lambda functions across different edge devices natively. More details about the architecture and functionality of Greengrass can be found in section 3.4.

To run serverless functions on the cloud, the AWS Lambda [12] service is used. AWS Lambda is the FaaS implementation of Amazon. AWS Lambda is an event-driven serverless compute service that lets a developer run code for a wide range of runtimes without provisioning or managing servers.

For communication from the edge to the cloud, the scheduler can utilize both AWS IoT MQTT [30] or AWS Simple Storage Service (S3) [11]. AWS IoT MQTT is used in almost all cases for any communication to or from the edge device. It is comparable to a publish subscribe queue where a message is published on a certain topic and a subscriber will receive this message, which can be both a lambda function running on the edge, the scheduler itself, or a lambda function running in the cloud.

AWS S3 is used only for messages larger than 128Kb, since IoT MQTT does not support messages larger than that. AWS S3 is a cloud storage solution that allows users to upload or download byte like objects to or from a "bucket". Certain hooks can be configured for this bucket such that a lambda function is invoked when a new item has been uploaded to the bucket. Because of this, S3 can be utilized as a message channel for exceedingly large messages.

For cloud to cloud communication, simple AWS lambda invocations are used. This means that the lambda functions are directly invoked from other lambda functions without any other means of communication in between. This is a simple and cheap solution, but lambda invocations have a limited message buffer. This can be a bottleneck for large high throughput pipelines. To satisfy these cases, communication via AWS Simple Queue Service (SQS) is also supported. AWS SQS is a publish subscribe service for

communication in the cloud and offers more scalability and a bigger buffer than direct lambda invocation for a higher price.

To store the device metrics in the cloud, so that the machine learning model in the scheduler can be trained on them, we use AWS Relational Database Service. More specifically, a PostgreSQL instance of AWS RDS, although any SQL flavor could be used for a database instance. We utilize a db.t3.micro database instance with 1GB of ram and 20GB of storage in order to remain in the free tier.

4.3 Papertronics deployment

To evaluate the scheduler we apply it to a real-world use case in the industry. More specifically, we create an edge/cloud deployment for the Beer-O-Meter [14]; a beer tester created by SG Papertronics [15].

As previously mentioned in section 2.3, the Beer-O-Meter is a portable beer testing station that is able to test the quality of beer by taking pictures of a colorimetric sample with beer on it. An early render of the device can be seen in figure 2.1a and an example of a colorimetric sample can be seen in figure 2.1b. In its current state, these images are sent directly to the cloud for analysis and processing. The station is only responsible for controlling the hardware and taking the pictures.

We create a new backend for this Beer-O-Meter to test and evaluate the scheduler. We move the image analyses process to the edge and only offload the processing tasks if the scheduler deems it more efficient with regard to cost or latency. A diagram of how this deployment would fit into our proposed design can be seen in figure 4.3. One should be able to see the resemblance with the scheduler design in figure 4.1. Each outgoing arrow is marked with a number, indicating the chronological order of interaction between components. The following enumeration describes the process of a test performed with the Beer-O-Meter, where each step number corresponds to a step in the diagram in figure 4.3:

1. The user sends a test request via Bluetooth or via AWS IoT Core to the Beer-O-Meter station. This request contains all the necessary instructions for the station, like the test type, amount of images to take and the amount of time to wait in between the captures of images. The station then proceeds to inform the user about the necessary steps he or she should take before the station can start taking pictures, like opening the cap, inserting a beer sample in a pod, inserting the pod into the station and closing the cap again.

2. When the user has finished all the necessary steps, the station automatically starts taking pictures. The number of pictures taken can range from 2 to 10, depending on the test type. Next, the images should be processed, such that only the relevant features are extracted. This is done by applying a mask to every image to isolate the paper and then extracting the average color. The resulting data are then sent to a specific component corresponding to the test type, which analyzes this data and returns a value corresponding to the measured pH, alcohol or bitterness level. Depending on the type of test, these values are calculated using linear regression or polynomial regression.

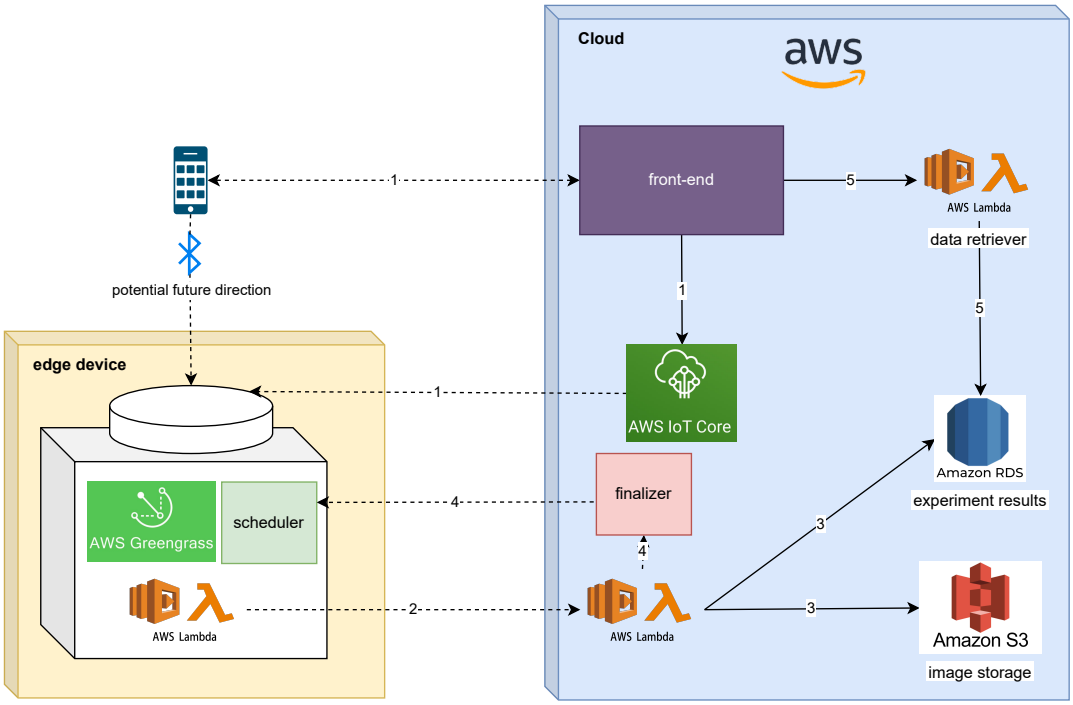


Figure 4.3: System design

These steps are split up into several different lambda functions, which can be executed on the edge or on the cloud, depending on the decision of the scheduler. An overview of the lambda function pipeline can be seen in figure 4.4. The pipeline starts with the station lambda, which is a lambda function that is configured to only run on the edge device, and is responsible for taking the pictures, setting up the LED's and interacting with the other hardware components on the device. After that comes the image processor, which is responsible for extracting the relevant features from the given images. Then we have a collection of lambda components responsible for different tests. The result of the image processing lambda is sent to the corresponding lambda depending on the test types set in the message metadata. Each test may perform a different operation on the data to retrieve the relevant information. Lastly, each test lambda sends the result to the collector lambda which stores the result in a database (next step) and sends it to the user.

3. The resulting data are stored in an SQL database and the images are saved in S3 cloud storage [11].

4. The measured statistics of the pipeline stored in the metadata of the final message is sent to the finalizer lambda component, which retrains the model on this collected data after a certain amount of messages have passed through or if retraining is requested. If the model is retrained, the edge device is notified of the update.

5. Finally, the data are retrieved from the database and presented to the user.

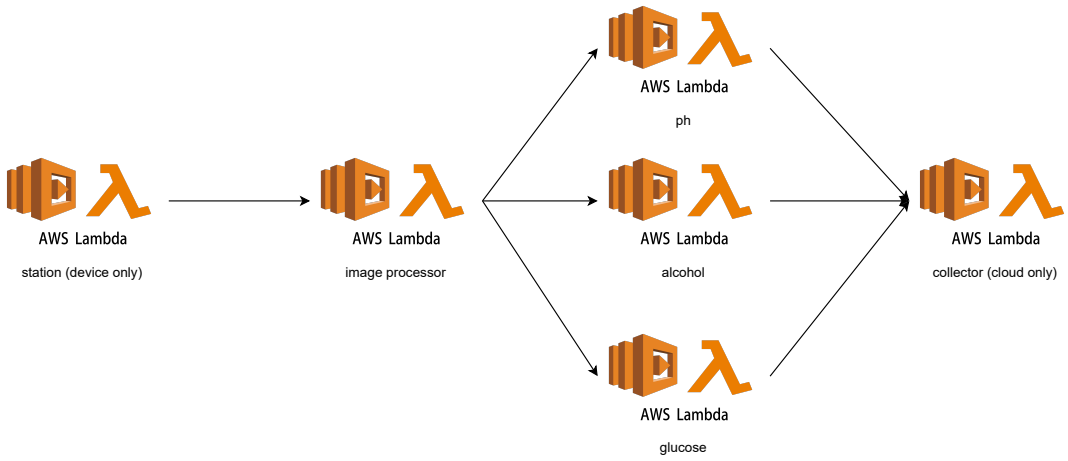


Figure 4.4: Lambda deployment

4.4 Configuration

In this section, we explain the configuration and settings we used for our experiments. Note that the configuration of a lambda function can be different on the edge and on the cloud, because of the differences in hardware limitations and platform settings. For example, Greengrass does not provide the ability to set a memory limit for non-dockerized lambda functions. All the listed parameters can be tweaked to the needs of the developer, we simply choose the values corresponding to the observed behavior of the individual lambda functions. For example, the image processing lambda has a longer duration than the default timeout in the cloud, and we therefore had to increase this value.

4.4.1 Lambda configuration

The settings for the cloud and edge deployment can be seen in table 4.2. Here we list the allocated memory in Megabytes, temporary storage in MegaBytes and maximum function duration in seconds in the cloud, as well as the maximum number of allowed instances, queue size in bytes, idle time before terminating the function in seconds and whether or not to keep the function running indefinitely. Note that we chose to set the station lambda to 1 maximum instance which is always running, since it will need sole access to the hardware (camera and LED's). If this would not be the case, then one instance could change the LED while another instance was running an experiment, making the experiment faulty.

Note that we do not have the option to set a CPU as a resource. That is because, in AWS, CPU resources are assigned according to the amount of memory assigned to the function [11]. So if one were to increase the memory resources of a serverless function, the CPU resources automatically scale with it. This also implies that serverless functions with more memory might, in the end, be cheaper to run than functions with less memory, because more CPU power may result in lower execution time and therefore a lower billed function duration.

	Image capture	**Image processing**	**ph**	**glucose**	**alchohol**	**finalizer**
cloud						
memory (MB)	-	256	128	128	128	128
ephemeral storage (MB)	-	666	512	512	512	512
timeout (sec)	-	900	20	20	20	20
edge						
max instance count	1	3	3	3	3	-
max queue size	1000	1000	1000	1000	1000	-
max idle time (sec)	240	60	60	60	60	-
warm start	True	False	False	False	False	-

Table 4.2: lambda function settings

4.4.2 Resource intensive configuration

To gain deeper insight into the performance of the scheduler for a more resource intensive deployment, we adjust this deployment to be slightly less efficient. To be more precise, we change one line of code used to filter the image:

```
filter = np.sum(array, axis=1) > 0
```

To a slightly less optimized one:

```
filter = [np.sum(a) > 0 for a in array]
```

The two lines of code do essentially the exact same, however the former is optimized by the use of NumPy and can therefore run faster and use fewer resources over time. The latter uses python list comprehension and can be 10 times slower than its NumPy counterpart. Numpy arrays are homogeneous and stored in continuous memory while python arrays are heterogeneous and stores pointers to different data types, this and the fact that NumPy uses C for its calculations makes NumPy the faster alternative [31].

This change should give us insight into how well the scheduler is able to adapt to a changing deployment, as well as give us insight into how well the scheduler performs with resource intensive serverless functions.

Chapter 5

Methodology

In this section, we describe the methods we use to create the dynamic serverless scheduler. In addition to that, we also describe how we tested the implementation.

To host our serverless functions in the cloud, we use the Amazon Web Service [11]. More specifically, we use AWS Lambda [12] for hosting the serverless functions on the cloud and AWS Greengrass [5] for hosting the serverless functions and the scheduler on the devices. We decided to use this platform over other alternatives, like Azure and Google cloud, since it is considered to be the most versatile and feature-rich service amongst its competitors in the area of edge/cloud computing [10].

We use AWS Relation Database Service to store the device statistics which the model will be trained on later. For communication between lambda functions on the edge to lambda functions on the cloud, we use AWS IoT [30] or AWS Simple Storage Service. AWS IoT is intended to transfer small messages of a maximum of 128 Kb in size. Any messages larger than that, like the images generated by SG Papertronics Beer-O-Meter, will have to be transferred via AWS Simple Storage Service. For communication between lambda functions within the cloud, we can use either AWS Simple Queue Service or direct lambda invocation. The scheduler supports both, however, we chose to only use direct lambda invocation, since no extra costs are tied to this method. For communication within the edge device itself, the local pub/sub queue of Greengrass is utilized.

All the serverless functions and the scheduler are written in the programming language Python [32]. Although not the fastest existing programming language, we have chosen this language in particular because of its maintainability, readability and support for machine learning applications. In addition to that, python was one of the 3 programs that the Greengrass software development kit supported, next to Java and C++.

5.1 Data

To train and evaluate the machine learning model inside the scheduler, around 500 messages were constructed and run through the original version of the pipeline and the resource-intensive version with a random scheduler in place. Then for each component, we collected the duration, transfer time, message size, estimated additional cost and device statistics like CPU usage and temperature, memory usage and network bandwidth.

	values
nr of images	2, 4, 6
wait time (sec)	0, 30, 120
tasks	(pH), (glucose), (alcohol), (pH, glucose), (pH, alcohol), (glucose, alcohol), (pH, glucose, alcohol)
internet speed (Kbps)	500, 1000, 5000, 50000

Table 5.1: values used to customize messages for data gathering

To add variety to the data set, both the network bandwidth, experiment settings, task type and message load are adjusted. The exact parameters used to tweak the variety can be seen in table 5.1.

The number of images field controls how many images the station should take for a single experiment. This setting should have a direct impact on the duration of most lambda components and on the size of the message sent from the first lambda component to the image processing component. We chose to tweak these values in order to evaluate how well the scheduler is able to predict the difference in execution time by just looking at the message size, CPU usage and memory usage, etc. without looking at the content of the message.

The wait time setting indicates how long to wait before sending the next message. This indicates the load we put on the station. Note that the actual time between messages send is calculated as follows: $n \cdot 5 + w$, where n is the number of images, 5 is the interval between images captured and w is the wait time. This setting is necessary, since the first lambda function, also called station lambda, can only process one single experiment at a time.

The task setting represents the set of tests that are performed in a single experiment. For example, with the task set (pH), only the Ph lambda function is executed, but with the task set (pH, glucose, alcohol), all task components are executed in parallel. This setting is tweaked in order to evaluate how well the scheduler is able to differentiate between tasks selected, without looking at the content of the message.

The internet speed setting represents the maximum amount of Kb/s that the station is able to download/upload. The internet speed was tweaked using the wondershaper Unix tool [33] in order to see if there would be a difference in execution time for different internet speeds and if so, if the scheduler would be able to distinguish those cases.

The data itself was gathered with a random scheduling method. That means that the recorded times do not include the duration it would take to apply a machine learning model to the data and predict the correct scheduling placement. However, the time the scheduler needs to apply the machine learning model to the data and generate some results is about 0.0005 seconds, which we consider to be a low enough difference to neglect it.

5.1.1 PRICE

In order for the scheduler to make a decision based on both the price and the execution time, the price corresponding to the execution needs to be calculated. In order to do that, we retrieve the latest pricing information for the AWS pricing API and then proceed to calculate the price given a predefined formula. Note that we only calculate the variable prices, that is the prices that can be varied by executing a lambda function either on the cloud or on the edge. We therefore only consider lambda invocation and execution costs, and MQTT and s3 message transfer costs. The prices are calculated as follows, using the formulas given by Amazon Web Services[11]:

$$L_{total} = r \cdot d \cdot L_{execution} + L_{request} + s \cdot d \cdot L_{storage} \tag{5.1}$$

Where r is the amount of memory assigned to a lambda function in the cloud in GB/s, d is the number of seconds the lambda function was running and s is the amount of ephemeral storage assigned to the lambda function. $L_{execution}$, $L_{request}$ and $L_{storage}$ are the latest prices retrieved from AWS for lambda executions, requests to the lambda function and ephemeral storage for the lambda function in dollars respectively. Finally, L_{total} is the total cost of a single lambda execution on the cloud. Note that there is no CPU entry in this calculation, since AWS automatically assigns more CPU power corresponding to the amount of memory assigned to the lambda function.

$$S3_{total} = S3_{put} + S3_{get} \tag{5.2}$$

Here $S3_{put}$ and $S3_{get}$ are the prices for posting and retrieving a single object from S3. Note that the storage price is not included in the calculation, since the messages will only be preserved for a short duration and deleted automatically. $S3_{total}$ denotes the total price for sending a message via S3.

$$MQTT_{total} = \cdot(m_s/8000) \cdot MQTT_{message} + MQTT_{rule} \tag{5.3}$$

Here m_s is the size of the message in bytes. This value is divided by 8000, since AWS IoT MQTT messages are sent in blocks of 8KB [5]. $MQTT_{message}$ and $MQTT_{rule}$ are the prices for sending a message via MQTT and redirecting it to a lambda function respectively. $MQTT_{total}$ is the total price for sending a message via MQTT.

Note that for some sub-regions, for example, eu-west-2 or eu-east-1, no pricing information was available. During calculations, we therefore assumed the pricing within the same continent to be identical and retrieved the pricing information from the first EU region found. At the time of writing this article, there is indeed no difference between pricing within a continent for the selected services, so this method will not result in any problems during the experiments. However, there is no guarantee that this will also be the case in the future.

Given these formulates, the price for a single lambda cloud execution is then calculated as follows:

$$P_{cloud} = \begin{cases} L_{total} + S3_{total}, & \text{if message is sent via S3} \\ L_{total} + MQTT_{total}, & \text{if message is sent via mqtt} \\ L_{total}, & \text{otherwise via direct lambda invoke} \end{cases} \tag{5.4}$$

If a component is executed on the edge, no additional costs are billed by AWS. The execution on the edge device and message transfer between components on the same device is free. AWS only charges users on a per device basis, and since this cost does not change relative to the amount or location of functions executed, we do not take this cost into account.

Furthermore, the cost of the consumed energy by the IoT devices is negligible compared to the cloud execution costs. Edge devices are usually resource constraint small IoT devices, meaning they have a relatively small energy requirement. For the raspberry pi 3B that we use, the energy consumption is at 1.4W on idle and 3.7W on full load. Assuming the IoT device will be turned on indefinitely, the extra wattage consumed by our processes running on the edge device would be no more than $3.7 - 1.4 = 2.3W$. Say a process would run for 5 minutes on the edge and the cost for 1 kWh would be \$0.331, than a single execution would cost $\frac{2.3W*(\frac{5}{60})}{1000} \cdot \$0.331 = \$0.0000634$. A process with similar runtime executing in the cloud with memory set to 256MB and a computation cost of \$0.000013 per GB/s, would cost $0.25GB \cdot 300 \text{ seconds} \cdot \$0.000013 = \$0.000975$, more than 10 times the price of edge execution. Because of this large difference in cost and to maintain some simplicity, we set the edge cost to 0. Anirban et al. [6] proposed the same strategy for simplicity's sake.

$$P_{edge} = 0 \tag{5.5}$$

Note that the price for the entire cloud execution is calculated by taking the total cloud duration and average lambda settings for the remaining lambdas. This may lead to inaccuracies if the lambdas in the cloud have a very varying memory configuration and average execution time. However, it is not possible to calculate the price without knowing the approximate duration beforehand. We need the message size to predict the duration in the cloud, and since we do not have that, this approximation should suffice.

5.2 Machine learning ensemble

For the scheduler to make accurate predictions based on available resources and message size, we use a machine learning algorithm. More specifically, we use a random forest regressor, or another variation of random forest, as our machine learning method. We choose a random forest like regressor, since they are known to be less likely to overfit and can work relatively well with a small data set.

The main candidates for our machine learning models are: Random Forest Regressor (RFR) [34], Extra Trees Regressor (ETR) [35] and Gradient Boosting Regressor [36] (GBR). In chapter 6 we evaluate the performance of all 3 methods with varying parameters and chose the optimal one for our scheduler.

5.2.1 Random Forest

Random forest [34] is an ensemble learning method that can be used for both regression and classification. Random forest works by generating a collection of decision trees. These decision trees are applied to the data to predict a certain value and an average prediction of these decision trees is returned. Due to the fact that Random forest constructs multiple decision trees, it is often less vulnerable to overfitting than a single decision tree.

The random forest algorithm uses the "bagging" principle and it works as follows:

1. k number of random subsamples are taken from the entire data set.
2. for each random subsample, an individual decision tree is constructed.
3. each decision tree generates an output
4. the final output is calculated by averaging the individual outputs of the decision trees.

5.2.2 Extra Trees

The Extra Trees [35] method is a variation of the Random forest method. They are similar in the fact that both are an ensemble of individual decision trees, however, they differ in two ways:

First of all, each decision tree in Extra Trees is trained using the entire data set, instead of a bootstrap sample. Second of all, Random Forest chooses the optimum split in decision trees while Extra trees chooses it Randomly. However, after the split points are determined, both methods choose the best one between all the features. Therefore, Extra Trees still has optimization.

The main argument behind this change is the reduction of both bias and variance. Using the whole data set instead of a bootstrap sample will reduce bias and determining the split point randomly will reduce variance. In addition to that, since split points are determined randomly instead of being calculated, the Extra trees algorithm is generally faster.

5.2.3 Gradient Boosting

Gradient Boosting [36] is also similar to Random Forest. The main difference between the two lies in how the decision trees are created and combined. In Random forest, the individual decision trees are created independent of one another, while in Gradient Boosting, the decision trees are created one after another. The method that Random Forest utilizes is also referred to as "bagging", while Gradient Boosting uses the "boosting" method to create and combine the trees.

Instead of creating k independent decision trees and simply taking the average of their output, we combine each tree sequentially and train each tree on the remaining error of the former tree to reduce the error of the entire model. More specifically, this algorithm works as follows:

1. Create a training set $\{(x_i, y_i)\}_{i=1}^{n}$ and determine a loss function $L(y, F(x))$:

2. Initialize the model with a value α for which the sum of the loss function applied to the training set is minimal:

$$F_0(x) = arg_\alpha min \sum_{i=1}^{n} L(y_i, \alpha)$$

3. for $m = 1$ to a given maximum number of iterations M

   (a) Compute pseudo-residuals on which the new decision tree should be trained:

$$r_{i,m} = -\left[\frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}\right] \text{ for } i = 1, ..., n$$

   (b) Fit a decision tree on the pseudo residuals. In other words, train a decision tree on the training set $\{(x_i, r_{i,m})\}_{i=1}^{n}$

   (c) Compute the multiplier α_m by solving the following equation:

$$\alpha_m = arg_\alpha min \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \alpha h_m(x_i))$$

   (d) Update the model by adding the trained decision tree:

$$F_m(x) = F_{m-1}(x) + \alpha_m h_m(x)$$

4. Return $F_m(x)$

This method of gradient boosting usually outperforms random forest [37], mainly because we train the trees to correct each other's errors. With this method, they are capable of capturing complex patterns in the data. There is one downside to gradient boosting however, if the data are too noisy, the boosted trees may overfit and start modeling the noise.

To find the most optimal machine learning model we evaluate the performance of several multi-output ensemble regression models with several different configurations and evaluate their performance against the collected data points.

The model will take the device statistics, like available CPU, memory, upload speed and message size as input and return the expected duration of cloud execution and edge execution of a lambda function. The exact features and their description can be seen in tables 5.2 and 5.3.

We can expect a decent score for the prediction of partial duration and transfer times for both the cloud and edge models, since the input features may be correlated with these values. However, the total remaining edge and cloud duration are not or barely influenced by these factors, since the duration depends on the performance of all next components in the pipeline as well. In addition to that, the total remaining edge duration is also influenced by any future scheduling decision.

Although we may expect a relatively high error for these specific output labels, a rough approximation of the total remaining duration should be sufficient to achieve decent performance from the scheduler. The scheduler will mostly rely on the predicted individual

	Unit	Description
CPU	percent (%)	The percentage of cpu that is not available at the start of execution
Memory	percent (%)	The percentage of memory that is not available at the start of execution
Network up	Megabit/s (Mb/s)	Network upload speed
Network down	Megabit/s (Mb/s)	Network download speed
Message size	Bytes	Incoming message size

Table 5.2: input features for the machine learning model

	Unit	Description
Individual duration edge	seconds	The duration of execution of the lambda function if this function were to be executed on the edge
Total remaining duration edge	seconds	The approximate remaining duration of the entire pipeline if this function were to be executed on the edge
Transfer time edge	seconds	The duration of the message transfer between the scheduler and the lambda function if this function were to be executed on the edge
Individual duration cloud	seconds	The duration of the lambda function if this function were to be executed on the cloud
Total remaining duration cloud	seconds	The approximate remaining duration of the entire pipeline if this function were to be executed on the cloud
Transfer time cloud	seconds	The duration of the message transfer between the scheduler and the lambda function if this function were to be executed on the cloud

Table 5.3: output labels for the machine learning model

	RFR	**ETR**	**GBR**
loss function	n/a	n/a	squared_error
learning rate	n/a	n/a	0.4,0.1
number of estimates	10,100,1000	10,100,1000	10,100,1000
subsample	0.9, 0.7, 0.5	0.9, 0.7, 0.5	1, 0.7, 0.5
criterion	"absolute_error", "squared_error", "poisson"	"absolute_error", "squared_error"	"friedman_mse"
max depth	2,3,4,None	2,3,4,None	3
min samples split	2,20	2,20	2,20
min samples leaf	10	1,10	1,10
max features	"auto"	"auto"	"auto"

Table 5.4: parameters for the different ensembles

lambda duration and the predicted transfer time. It will only use the predicted total remaining duration to calculate the price for a cloud execution and to validate that the user-defined constraint is still met.

5.2.4 The best regression method

To evaluate the different regression methods, we run all of them multiple times, with varying parameters, on the generated data set. The exact set of parameters we tested with can be seen in table 5.4. Note that each entry refers to a parameter that can be tweaked in the scikit-learn [38] implementation of the model. If there is an entry n/a in the table for a specific parameter, that means that this parameter could not be tweaked for this particular regression method.

For each ensemble and each possible combination of parameters, the model is trained at least once on 75% of the entire data set. The remaining 25% is used to validate the performance of the ensembles and their configurations. The quality of each model has been quantified by a coefficient of determination R^2. We evaluate each method using this coefficient, as well as its approximate memory usage and training time. R^2 is calculated by subtracting the residual sum of squares divided by the total sum of squares from 1. The exact formula of the coefficient of determination R^2 can be seen in equation (5.6). Here R^2 is the coefficient of determination, RSS the residual sum of squares, TSS the total sum of squares, y the true values, y' the predicted values and N the number of elements in y.

$$R^2 = 1 - \frac{RSS}{TSS} \tag{5.6}$$

$$RSS = \sum_{i}^{N} (y_i - y_i')^2 \tag{5.7}$$

$$TSS = \sum_{i}^{N} (y_i - \bar{y})^2 \tag{5.8}$$

5.3 Automatic function deployment infrastructure

To remove some load on the developer when developing and (re-)deploying serverless functions, we automated the process of deploying serverless functions on the edge and cloud, as well as automatically instantiating communication infrastructure. The developer will only have to provide a .yaml file containing the configuration of the pipeline and the infrastructure will be automatically set up. An example of this .yaml file can be seen in Appendix A.

In this file, the developer will have to specify the scheduler constraint, like a deadline constraint or cost constraint, and the complete configuration of all lambda functions. Developers are able to tweak the resource settings like memory and timeout for both cloud and edge instances in one single file and are also able to set environment variables this way, for both the edge and cloud environment.

The developer is also able to orchestrate communication between serverless functions by setting the "destinations" key. This key lets the developer specify what lambda functions the result should go to. The developer is also able to set custom protocols for edge-edge, edge-cloud and cloud-cloud communication. Current supported communication channels are lambda invoke and AWS SQS for cloud to cloud communication, AWS IoT MQTT and AWS S3 for edge cloud communication and Greengrass local pub/sub for local communication. Most of the time the developer will not need to set these protocols, since the scheduler will automatically pick the most suited one based on message size. For example, AWS IoT MQTT is picked when the message is below the maximum allowed size of 128KB and the slower AWS S3 alternative is picked if the message is over this maximum threshold.

Do note though that the automatic function deployment program only creates the edge/-cloud infrastructure. Any other services an application may require, like an SQL Database, cloud storage or other non-lambda compute instance like AWS EC2, will still need to be manually created by the developer.

Chapter 6

Results

In this chapter, we determine what regression method is most suitable as our machine learning algorithm for the scheduler. We also show the score of the machine learning algorithm for different lambda functions and output labels, as well as show the performance of the scheduler on the Papertronics deployment for different cost and deadline constraints. We will also determine possible overhead caused by the scheduler and gather some metrics on the automated deployment script. Then, in chapter 7, we discuss these results.

6.1 Pipeline data

In this section, we present the characteristics and overall performance of the created pipeline. We mainly look at the performance of the pipeline and the relation between the output labels and the input features of the model in the gathered training data. The input features and output labels of the model can be found in tables 5.2 and 5.3. We created 2 versions of the pipeline, one which we refer to as the "original" and one we refer to as the "resource intensive". The two versions only differ in one line of code, which is explained in section 4.4.2. We gathered around 500 data points of the pipeline performance for both cases with random scheduling and we try to find some relation between the input features of the model and the output labels by plotting this data. An overview of what experiment number corresponds to what experiment configuration can be seen in figure 6.1. The features adjusted correspond to the values listed in table 5.1.

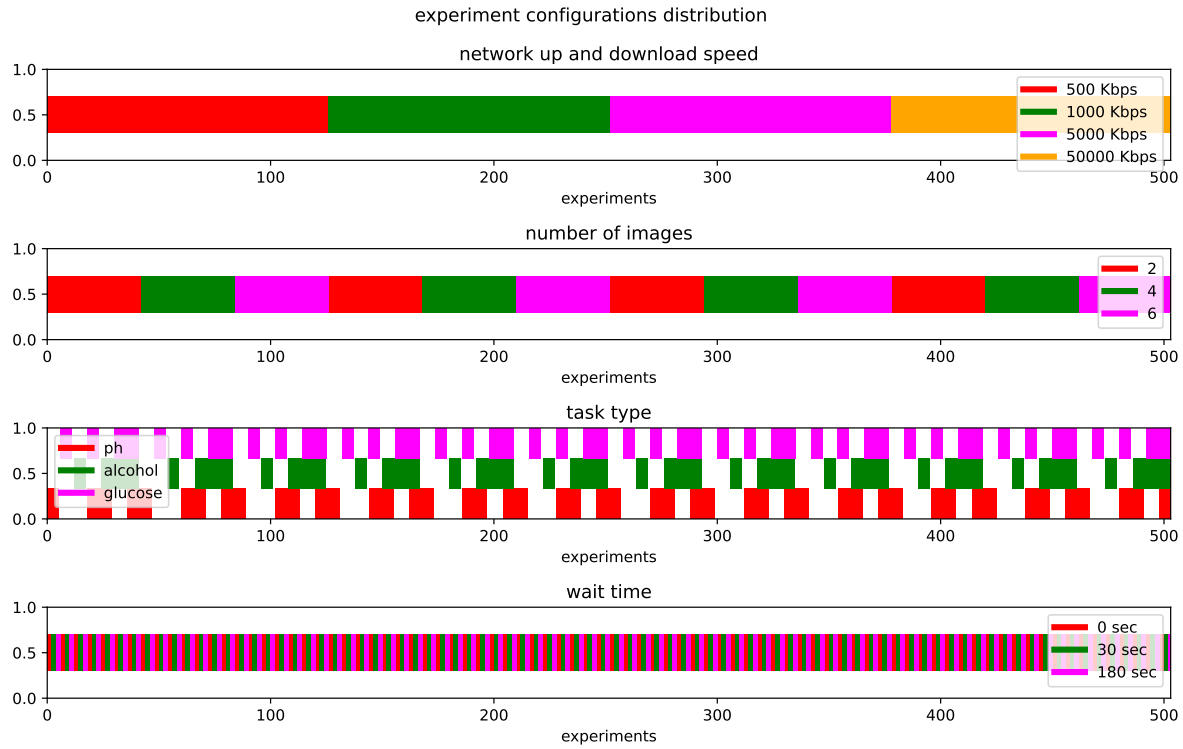


Figure 6.1: The distribution of settings over the experiments.

Each colored rectangle in this figure corresponds to a value that has been set for the experiments. For example, it can be seen from this figure that the network up and download speed has been set to 500 Kbps in the first 125 experiments and 1000 Kbps in experiment number 125 to 250. Some settings can overlap as well, like the task type setting. The task type can be set to ph, glucose and alcohol, but also to a multiple like (ph, glucose), (glucose, alcohol) or (glucose, alcohol, ph). The combination of these settings have been visualized in the figure as overlapping rectangles.

Note that the wait time setting changes every 2 experiments, which makes the exact wait time per experiment hard to see. In figure 6.2 a better readable cutout of the wait time setting per experiment number can be seen, which makes the mentioned pattern distinguishable.

This figure is not particularly useful on its own, but it can be used to identify and classify patterns in later figures presented in this chapter. For example, this figure can be paired with figure 6.3 to help identify what changes to the experiment configuration result in peaks in the duration.

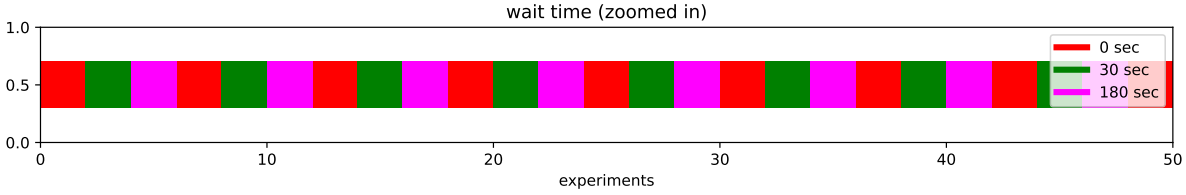


Figure 6.2: The distribution of wait time setting over the experiments zoomed in.

6.1.1 Original Pipeline

The individual duration and transfer time of each individual lambda function in the original pipeline over several experiments with varying internet speed and experiment settings can be seen in figure 6.3. Note that the station lambda and collector lambda are not included in these results. These functions are not included, since their placement is fixed and they cannot be scheduled. In figure 6.3 we can see that the individual duration of lambda functions on the edge and on the cloud seem to be relatively stable, apart from the image processing lambda. The image processing lambda processes and filters the images belonging to an experiment, and the duration of the lambda function is therefore dependent on how many images were taken by the station. The more images were taken, the longer the duration of the image processing lambda. At each peak, the experiment contained around 6 images and at each valley, the experiment contained only around 2 images, as can be seen if we cross-reference the peaks with figure 6.1. These constant and dependent processing times are a good indication that the machine learning model is able to predict the individual lambda duration.

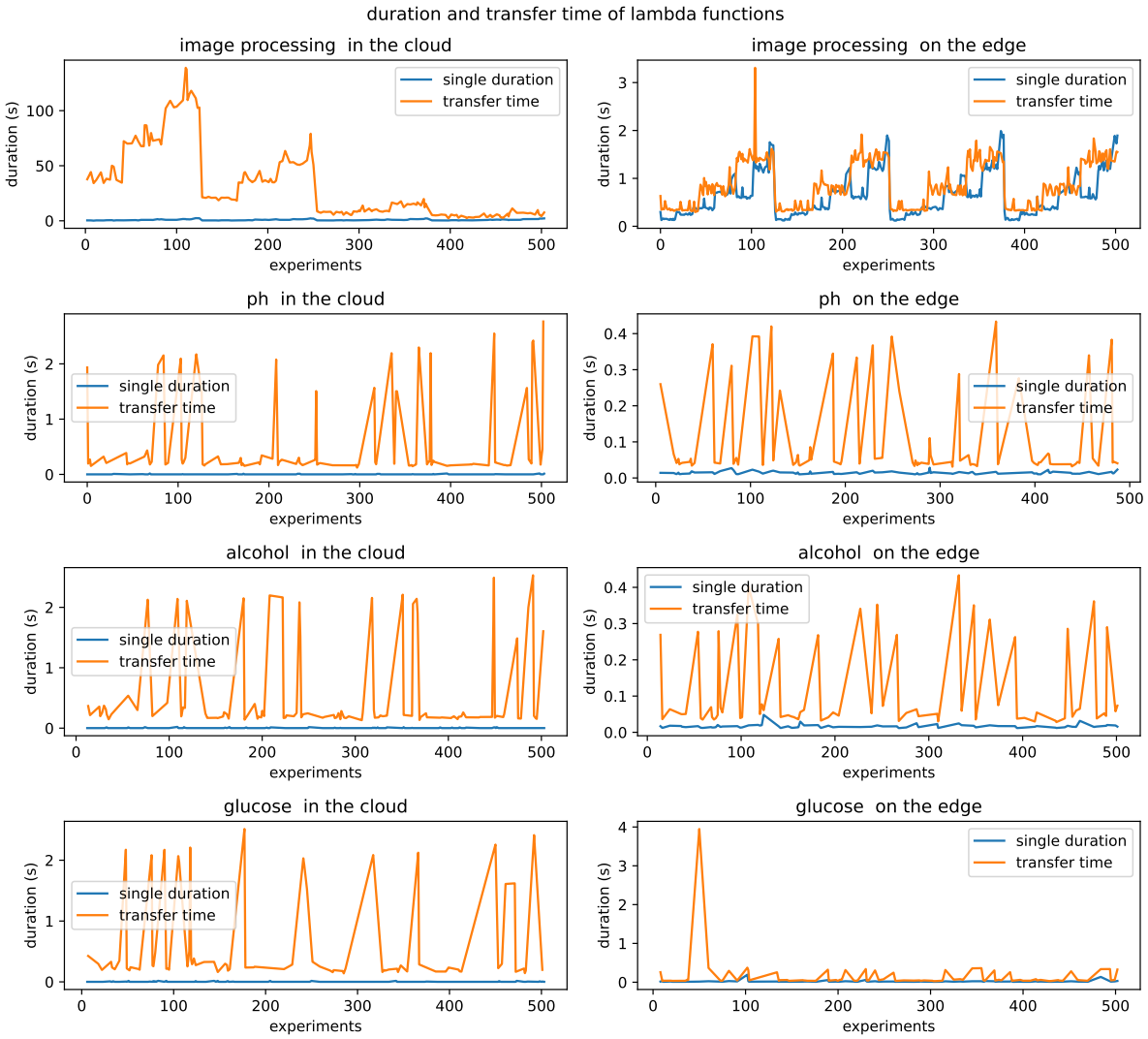


Figure 6.3: duration and transfer time of lambda functions over experiments

The transfer time on the other hand seems to vary a bit more per experiment. For the image processing lambda, once again a pattern can be observed. Just as with the duration, transfer time is also related to the incoming images, and therefore the incoming message size. In addition to that, for the cloud execution, the transfer time is also related to the network speed. Namely, for the first 125 experiments or so, the network speed has been set to a max of 500 Kb per second and increased after that, as can be seen from figure 6.1. This explains the peaks at the beginning for the image processing lambda.

The variation in transfer times for the other lambdas is not as easily explained. So far, no real pattern can be seen in the figure. The variation can be a number of things: The infamous cold/warm start problem [17] explained in section 3.1, a byproduct of sending messages over the internet for the cloud execution or available resources for edge execution.

In this figure, we can see that for the original pipeline implementation, edge execution is almost always the optimal one. Compared to the cloud, the overall execution and transfer times are much lower. Although this characteristic of the pipeline will not give us much information on how well the scheduler is able to adapt its decision based on parameter settings, it could provide us with some information on how well the scheduler is able to perform knowing the optimal scheduling decision beforehand. Now that we know that edge scheduling is always optimal, we can determine the error of the scheduler based on the number of tasks that were not scheduled on the edge. To still be able to evaluate the ability of the scheduler to adapt its decision to the changing environment, we also train the scheduler on a more resource-intensive version of the pipeline.

RESOURCES AND MESSAGE SIZE VS DURATION

The relation between the available resources and individual lambda duration for the image processing lambda can be seen in figures 6.4 and 6.5. More detailed plots of the generated data can be seen in Appendix C. More specifically, the relation between the available resources and individual lambda duration for all lambdas can be seen in figures C.1 and C.2.

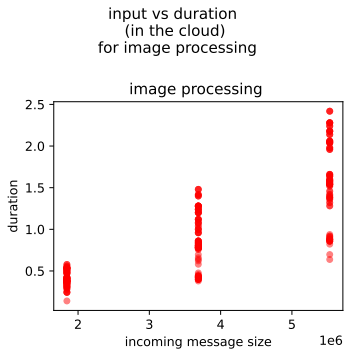


Figure 6.4: Incoming message size versus the duration of the image processing lambda if this lambda is executed in the cloud.

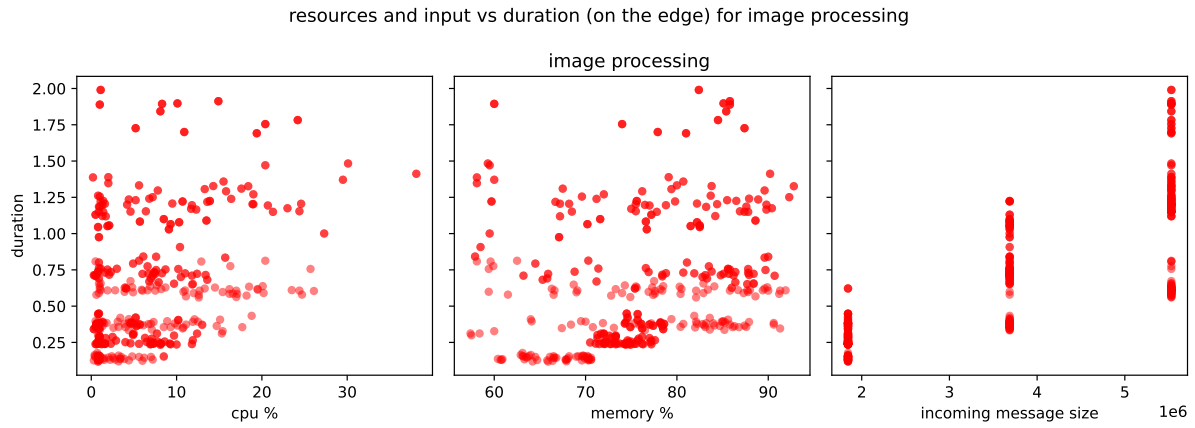


Figure 6.5: Available resources and incoming message size versus the duration of the image processing lambda if this lambda is executed on the edge.

In figures 6.4 and 6.5 we can see a relation between the duration of the image processing lambda function and the size of the incoming message for both a cloud and an edge deployment. This relation most likely exists because this particular lambda needs to scan and filter each incoming image and its duration is therefore directly related to how many images come in. In turn, the message size is a strong indication of how many images it contains.

Other factors that influence the duration of the image processing lambda are the number of tests (ph, glucose or alcohol) to perform next. This explains the variation still present in the incoming message size vs duration plot for the image processing lambda. In the case that multiple tests are selected for one invocation, the colorimetry paper will be divided into multiple sections and the image processing lambda will need to retrieve features for each section, therefore increasing the duration. The scheduler does not have direct access to the number of tests that need to be performed, since this is hidden within the context of the message. This characteristic may make it difficult for the scheduler to predict the total duration.

A slight pattern can also be observed in the CPU vs duration and memory vs duration plot for the image processing lambda in figure 6.5. At a low CPU and memory percentage, implying a higher availability of CPU and ram resources, the duration seems to be evenly distributed. However, when the available resources become more scares, the minimal duration also seems to increase. This relation should help the scheduler in predicting the individual duration of this lambda function.

For the other short-running lambda functions, like pH, alcohol and glucose visible in figures C.1 and C.2, these relations seem less obvious. This is because these lambda functions only run for about 0.05 to 0.2 seconds, and because of this, there is an increase in variance within the duration caused by decisions made by the operating system. This should not be a problem however, if the scheduler is able to predict the average duration of these lambda functions over the data set. the scheduler should still be able to make an accurate offloading decision based on that.

NETWORK BANDWIDTH AND MESSAGE SIZE VS TRANSFER TIMES

Now that we have discussed the relation between the available resources, message size and individual lambda duration, we can look into the relation between network bandwidth, available resources, message size and transfer times between lambda functions. The relation between the available resources on the device, the network speed, the message size and the transfer time to the image processing lambda can be seen in figure 6.6. The relations between these values for all lambda functions for both cloud and edge execution can be seen in figures C.3 and C.4 in the Appendix.

Looking at both figure 6.6, we can see that for image processing lambda the transfer time is also strongly related to message size. For the edge, this relation is clearly visible. The bigger a message is, the more load is put on the station and the longer it will take to send it to the next process. For sending to cloud, this relation is visible, but sometimes for big messages, the transfer time can also be low. This is because cloud transfer times of the image processing lambda are not only dependent on message size, but also on network speed. Compared to the cloud, the transfer duration on the edge itself is around 40x faster than sending the message to the cloud.

This relation between network speed and transfer time does not exist in internal edge communication, since no network bandwidth is required there. Messages are only communicated between processes running on the station itself. What does seem to influence this duration are the CPU and memory resources of the station. For image processing lambda the minimal transfer time seems to increase with a higher CPU and memory occupancy, meaning fewer resources available, and even for the other lambdas the transfer times lasting longer than 0.2 seem to be influenced by memory and CPU usage.

Looking at figures C.3 and C.4, we can see that message size and network bandwidth do not seem to have any influence on the other lambda functions, most likely since the message size for these functions is rather small.

INPUT FEATURES VS TOTAL REMAINING PIPELINE DURATION

The final output label we discuss is the total pipeline duration. We do not expect to find many relations here, since the total remaining pipeline duration should be influenced by more factors than just the available resources at some point in the pipeline. We therefore only look at the message itself and its size. The plot visualizing this relation for the image processing lambda can be seen in figure 6.7. The plot visualizing this relation for all the lambda functions can be seen in figure C.5 in the appendix.

As expected, no direct relation can be observed for the image processing lambda, or any other lambda for that matter. We do not expect the model to perform well on this output label, however we want an R^2 score of 0 such that the model is at least able to return the average duration over the data set. This should be enough for the scheduler to perform decent, since the total duration is mostly used for calculating the price and checking if the user-defined constraints are met.

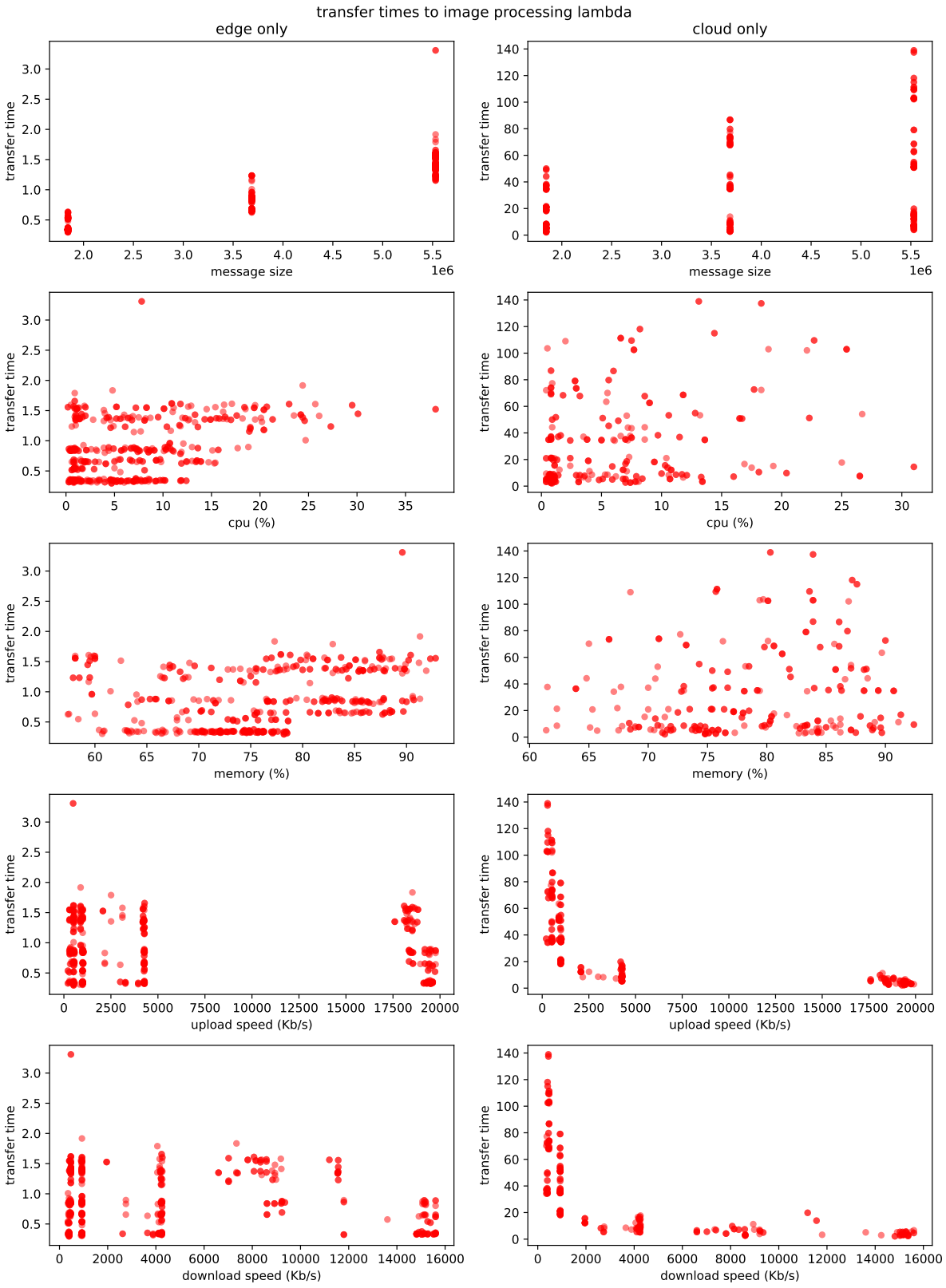


Figure 6.6: Incoming message size versus the transfer time to the image processing lambda

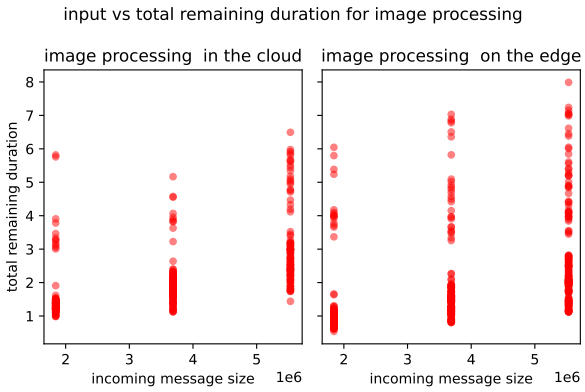


Figure 6.7: Incoming message size versus the total remaining pipeline duration of the image processing lambda and all following lambdas.

6.1.2 Resource intensive pipeline

Many of the same patterns observed for the original pipeline can be seen in the resource-intensive pipeline as well, as can be seen in figure 6.8. We can now see however that the duration of each lambda function on both the edge and the cloud has increased drastically for the image processing lambda. The duration of all the other lambda functions seem to be relatively unaffected by the change.

Another notable change is the increase in transfer time variation for image processing on the edge. Since only the optimization has been changed, the content and the size of the message being transferred to the image processing lambda have not changed, so the message size cannot be a factor in this new behavior. The increase in variation of the transfer time is likely caused by the increase in used resources on the edge. Since the image processing lambda needs about 100x more time to process the images, it is more likely that multiple image processing lambdas are being executed at the same time. If the station does not have enough resources, then the station will have trouble starting up another image processing instance or relaying the message to an idle one.

This characteristic of the pipeline might be beneficial for evaluating the effectiveness of the scheduler. In reality, we would not want a message to be stuck in a queue for around 400 seconds before the edge device has enough resources to process it. We would more likely prefer a cloud execution in this case. In the following sections, we can evaluate the scheduler on how well it is able to predict these spikes in transfer time and avoid edge execution in those cases.

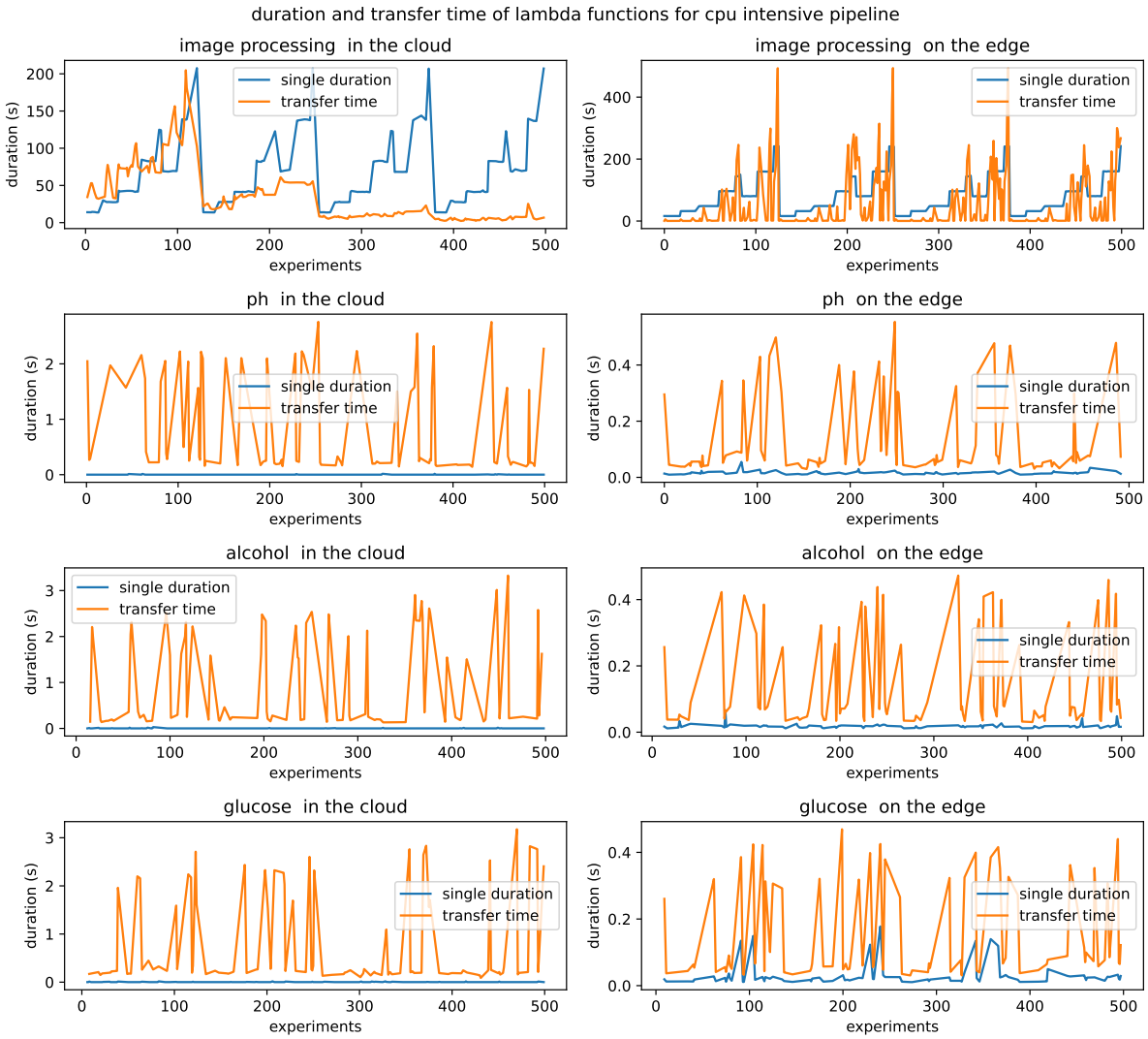


Figure 6.8: duration and transfer time of lambda functions over experiments for resource-intensive pipeline

RESOURCES AND MESSAGE SIZE VS DURATION

The relation between the available resources and individual lambda duration for the image processing lambda for a resource-intensive pipeline can be seen in figures 6.9 and 6.10. More detailed plots of the generated data can be seen in Appendix C. More specifically, the relation between the available resources and individual lambda duration for all lambdas can be seen in figures C.6 and C.7.

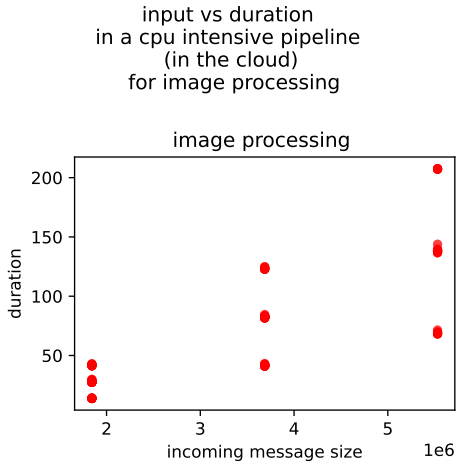


Figure 6.9: Incoming message size versus the duration of the image processing lambda if this lambda is executed in the cloud for a resource-intensive pipeline.

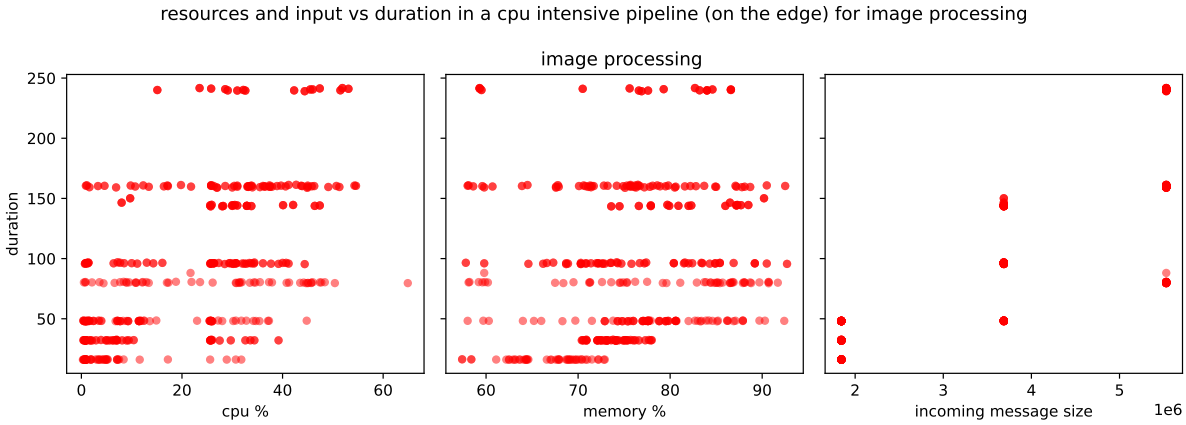


Figure 6.10: Available resources and incoming message size versus the duration of the image processing lambda if this lambda is executed on the edge for a resource-intensive pipeline.

As you can see, most of the relations that were present for the original pipeline still remain for the resource-intensive pipeline. The duration of the image processing lambda still heavily relies on the incoming message size for both an edge and cloud execution, CPU and memory availability still influences the minimal possible duration and the data of the other lambda functions remains completely unchanged.

There are two notable differences for the image processing lambda however. First of all, there seems to be a clear gap between durations. The reason for this gap is not deducible

from the plots, but as explained earlier, the image processing lambda duration not only relies on incoming message size, CPU and memory, but also on the number of tests (pH, glucose or alcohol) that need to be performed. Now that the lambda is less optimized, the difference between processing the image only once, twice or thrice is more notable in the duration of the lambda function. Thus causing the gaps in duration.

Second of all, the duration of the image processing lambda is now 100 times slower. This characteristic was to be expected from the changes made to the pipeline. We can use this characteristic to evaluate the performance of the scheduler on longer running, more resource-intensive, tasks.

NETWORK BANDWIDTH AND MESSAGE SIZE VS TRANSFER TIMES

Next, we look into the relation between the available resources, network bandwidth message size and transfer time. These relations can be seen in figure 6.11 for the image processing lambda. The relations for all lambda functions can be seen in figures C.8 and C.9 in Appendix C.

An interesting development can be seen in figure 6.11. The transfer time has increased 50 times on the edge compared to the original pipeline. This is not a direct cause of our change to the pipeline. The messages being produced by the lambda functions are still exactly the same, and can therefore not influence the transfer time. Rather, this behavior is the result of a longer running image processing lambda instance. It is now more likely that multiple messages come in when the station is still processing previous images. The resources on the station are limited, meaning that if the station is busy processing multiple other requests, it will have more trouble starting up a new lambda instance and running it. This causes messages to be stuck in a queue for a while, therefore increasing the transfer time.

This explanation is underlined by the CPU and memory plot in figure 6.11. For low CPU and memory usage, the transfer time seems to be low, since in this case, the station can process the messages immediately. However when the station seems to be busy, and the CPU usage is up, the transfer times go way up. This implies that if the station is already processing something, it will take longer to transfer and invoke the next lambda function.

The other lambda functions are not as much influenced by this fact. This is likely due to a combination of a much smaller message size that needs to be transferred and the fact that fewer new instances need to be created because some are already running, which is the case for image processing lambda.

There is not much difference between the original pipeline and this resource-intensive one for the transfer time to the cloud, as can be seen in figures 6.11 and C.8. The transfer time for image processing lambda still seems to be highly influenced by incoming message size and network bandwidth. The transfer times of the other lambda functions still seem to be rather unaffected by the variations in resources and incoming message size.

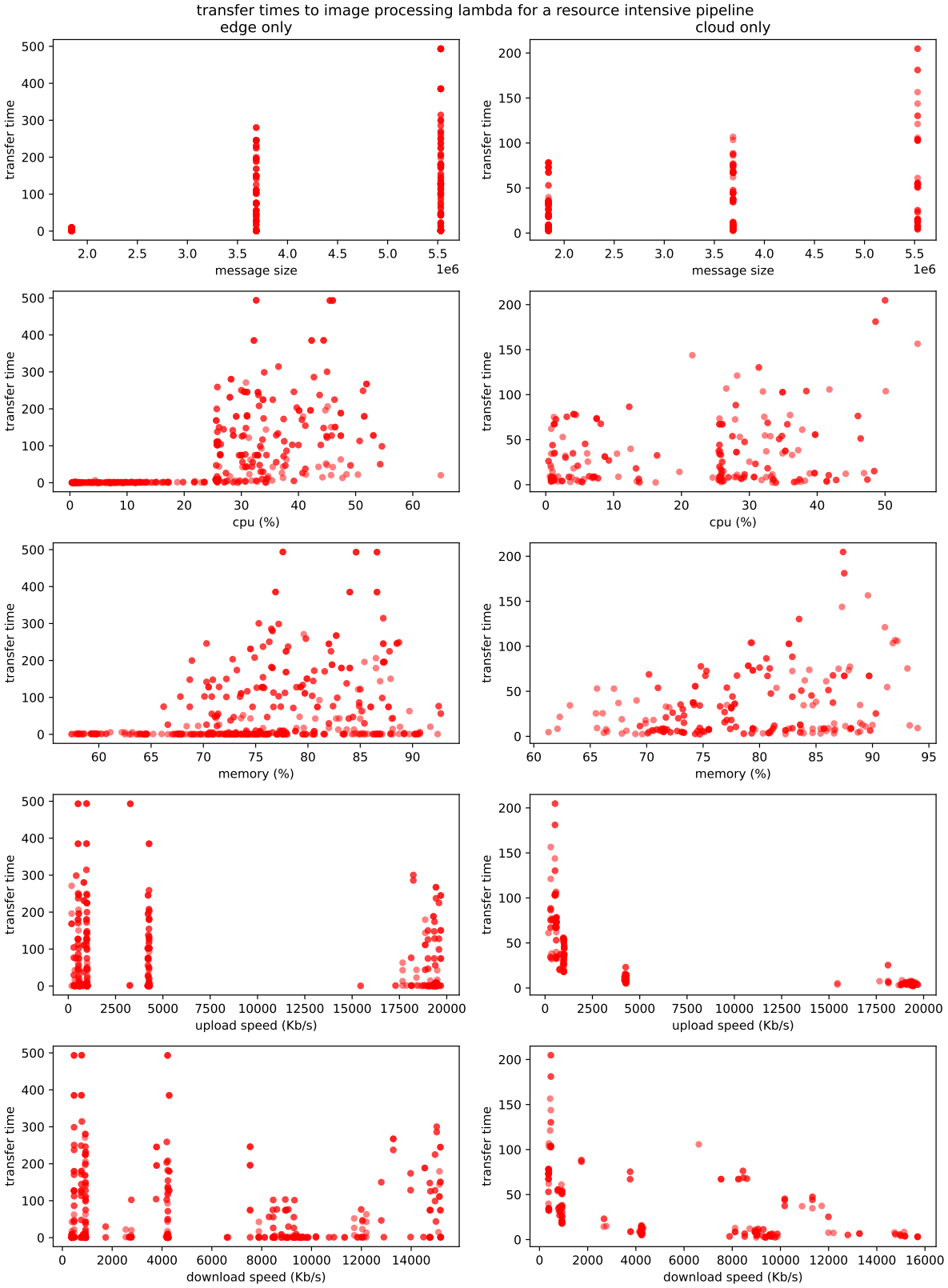


Figure 6.11: Incoming message size versus the transfer time to the image processing lambda for a resource-intensive pipeline

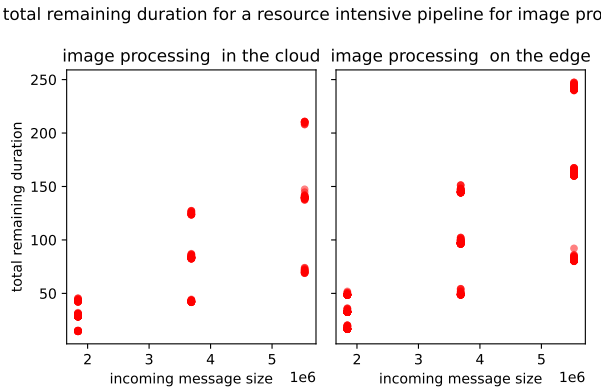


Figure 6.12: Incoming message size versus the total remaining pipeline duration of the image processing lambda and all following lambdas for a resource-intensive pipeline.

INPUT FEATURES VS TOTAL REMAINING PIPELINE DURATION

Just as with the original pipeline, the final output label we discuss is the total pipeline duration. The plot visualizing this relation for the image processing lambda can be seen in figure 6.12. The plot visualizing this relation for all the lambda functions can be seen in figure C.10 in the appendix.

Contrary to figure 6.7 of the original pipeline, figure 6.12 does show some relation between the incoming message size and the total remaining pipeline duration. This is likely the case due to the fact that the duration of the image processing lambda overshadows all the other ones. The duration of the image processing lambda is somewhere between 50 and 200 seconds for large message sizes and the other lambda functions only have a duration of a few seconds. This is good news for the model, since it should now be much easier to predict the remaining pipeline duration.

6.2 Machine learning ensemble comparison

To find the most suitable machine learning algorithm for the scheduler, we train and test the Random Forest method, Extra Trees Method and Gradient Boosting method with a number of different configurations on the previously mentioned data set. We then quantify the performance of each model according to the R^2 measurement in equation (5.6).

The performances and model size in memory of each method for different parameter configurations can be seen in figures 6.13, 6.14 and 6.15. From this data, we then pick the configuration of parameters for each model for which the R^2 score is overall the highest for each label and lambda function and for which the memory size of the model is the lowest. The best configuration that was picked is marked by a vertical black line in the respective figures. These picked configurations can be found in table 6.1.

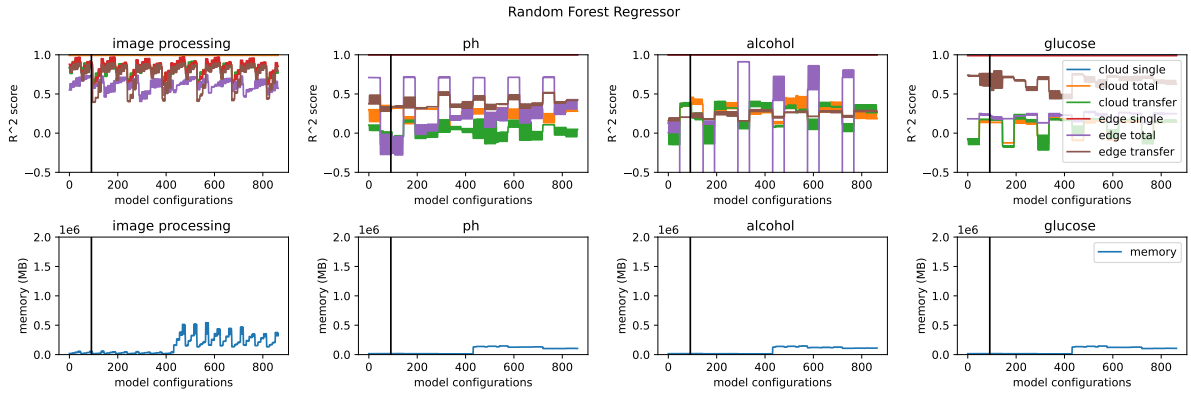


Figure 6.13: This figure indicates the performance measured by the R^2 metric in equation (5.6) and the model memory size of the Random forest method for different parameter configurations. The yellow line indicates the parameter configuration chosen as the best based on overall performance and size of the model in memory.

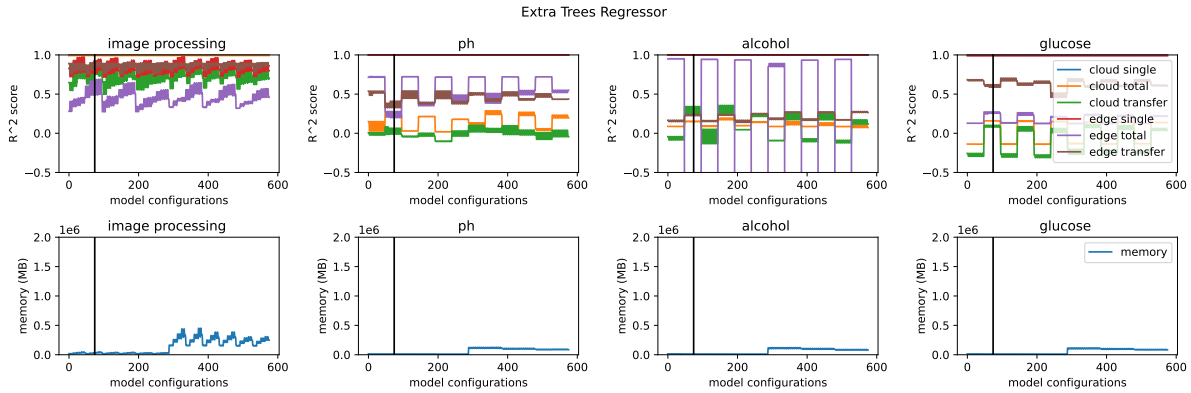


Figure 6.14: This figure indicates the performance measured by the R^2 metric in equation (5.6) and the model memory size of the Extra trees method for different parameter configurations. The yellow line indicates the parameter configuration chosen as the best based on overall performance and size of the model in memory.

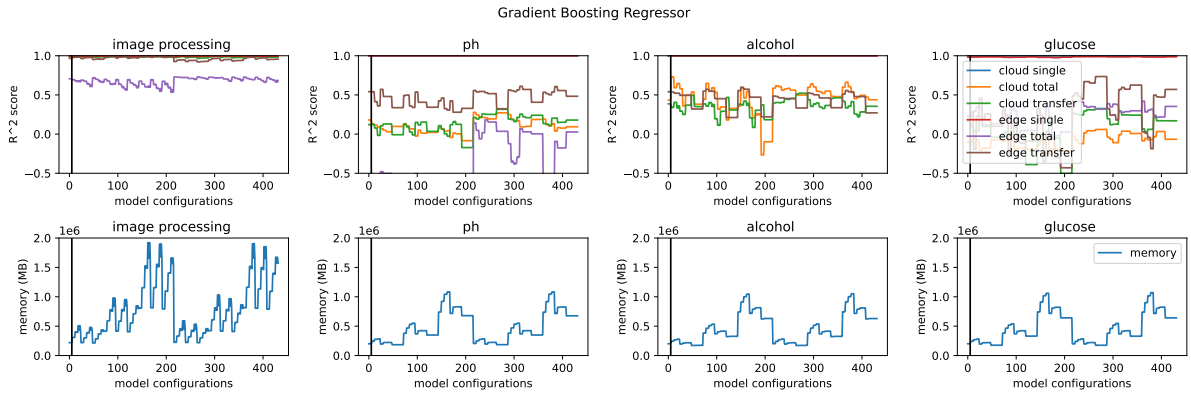


Figure 6.15: This figure indicates the performance measured by the R^2 metric in equation (5.6) and the model memory size of the Gradient boosting method for different parameter configurations. The yellow line indicates the parameter configuration chosen as the best based on overall performance and size of the model in memory.

Parameter	GB	ET	RF
loss function	squared_error	n/a	n/a
learning rate	0.4	n/a	n/a
n_estimators	50	10	10
subsample	1	0.9	0.9
criterion	friedman_mse	squared_error	squared_error
max depth	2	4	None
min samples split	20	2	20
min samples leaf	10	10	10
max features	auto	auto	auto

Table 6.1: Parameter values for scikit-learns Gradient Boosting (GB), Extra Trees (ET) and Random Forest (RF)

The R^2 scores of the best parameter configuration of the 3 models can be seen in figure 6.16, 6.17 and 6.18. Since the model is placed on a resource constraint IoT device, the size of the model in memory is also of importance. The size of the chosen models in memory can be seen in table 6.2.

	RFR	ETR	GBR
size of the model in memory (KB)	98.9	69.8	789.59

Table 6.2: The size of the chosen models in memory.

To evaluate the models, we calculated the average scores (weighted by the duration of each lambda function). These calculated scores can be seen in table 6.3. Although the scores are rather similar, the approximate best performing ensemble was Gradient Boosting. We therefore pick Gradient Boosting as our machine learning algorithm for the scheduler. The configuration corresponding to the picked machine learning model can be seen in table 6.4.

Apart from the performance score, we also care about the memory footprint. Since we use this model on a resource-constrained device, running in the background, we want a

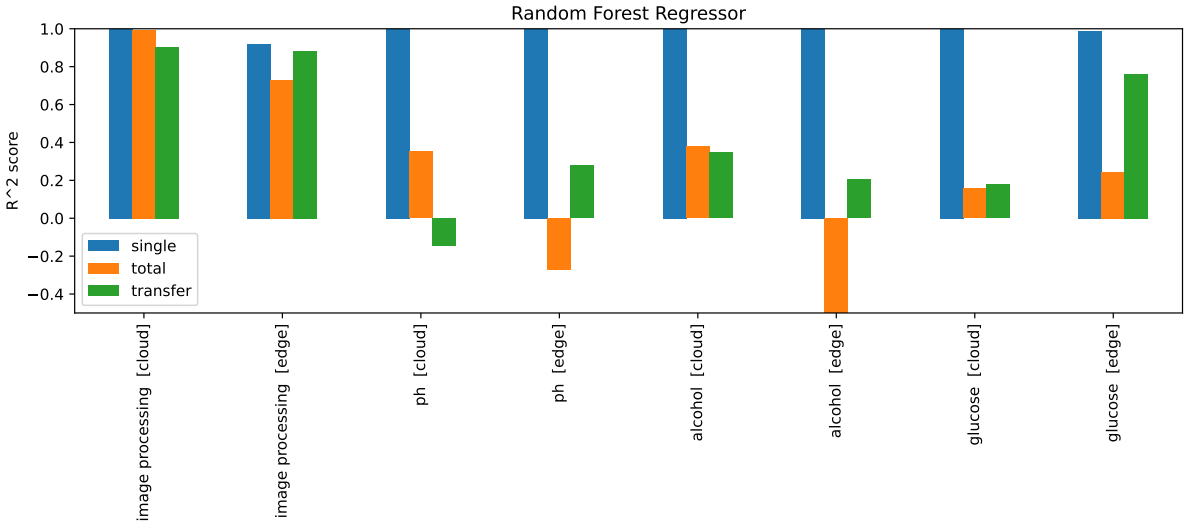


Figure 6.16: Random Forest score with best measured configuration

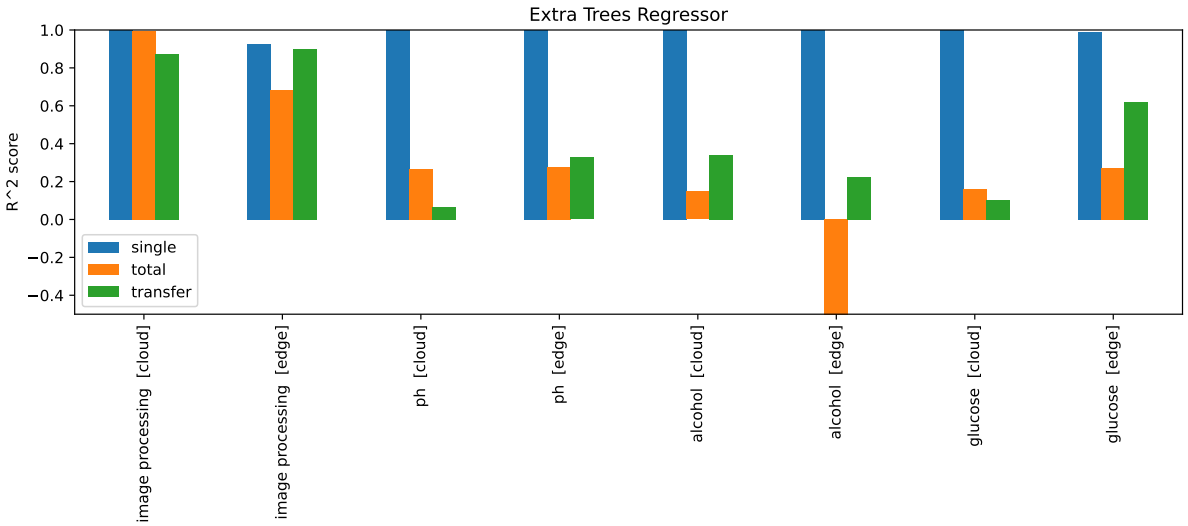


Figure 6.17: Extra Trees score with best measured configuration

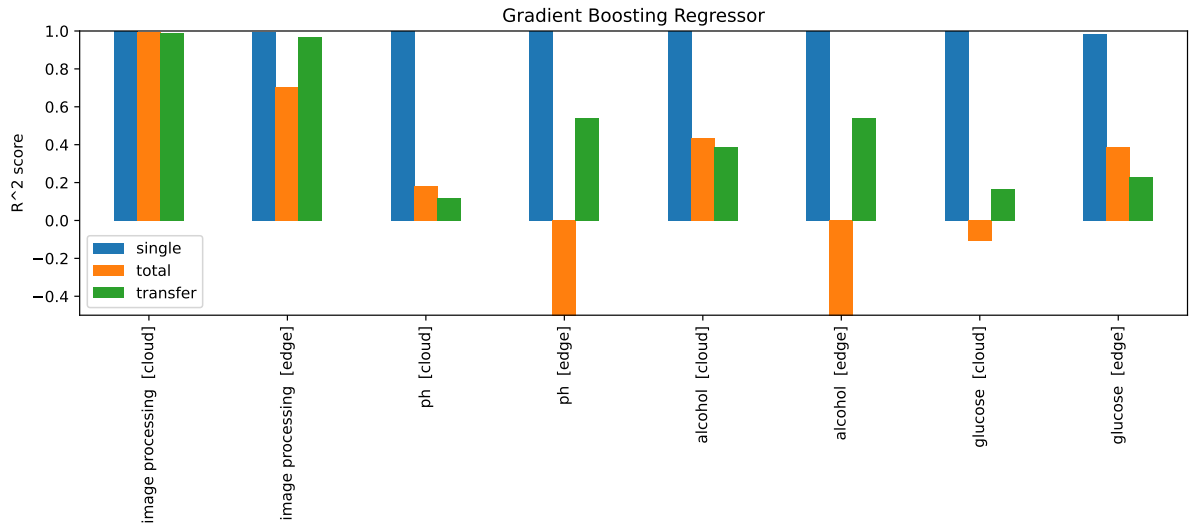


Figure 6.18: Gradient Boosting score with best measured configuration

	RFR	ETR	GBR
average R^2 scores	0.627	0.635	0.638

Table 6.3: The average score of each machine learning method over all lambda functions, placement (edge or cloud) and output labels

lightweight model with decent prediction power. Although the selected Gradient Boosting configuration needs a bit more memory than the other 2 chosen configurations for Random forest and Extra trees, we choose a slightly higher memory footprint to get a slightly higher score.

For this scheduler, we choose Gradient Boosting because of its overall performance over the other alternatives. Although the memory requirements are a bit higher than the alternatives, it should still be within the capabilities of most IoT devices. Future research may look into the performance of a regression method like Random Forest or Extra Trees with a lower memory footprint on extremely resource constraint IoT devices.

Parameter	value
loss function	squared_error
learning rate	0.4
n_estimators	50
subsample	1
criterion	friedman_mse
max depth	2
min samples split	20
min samples leaf	10
max features	auto

Table 6.4: Parameter values for scikit-learns Gradient Boosting Regressor

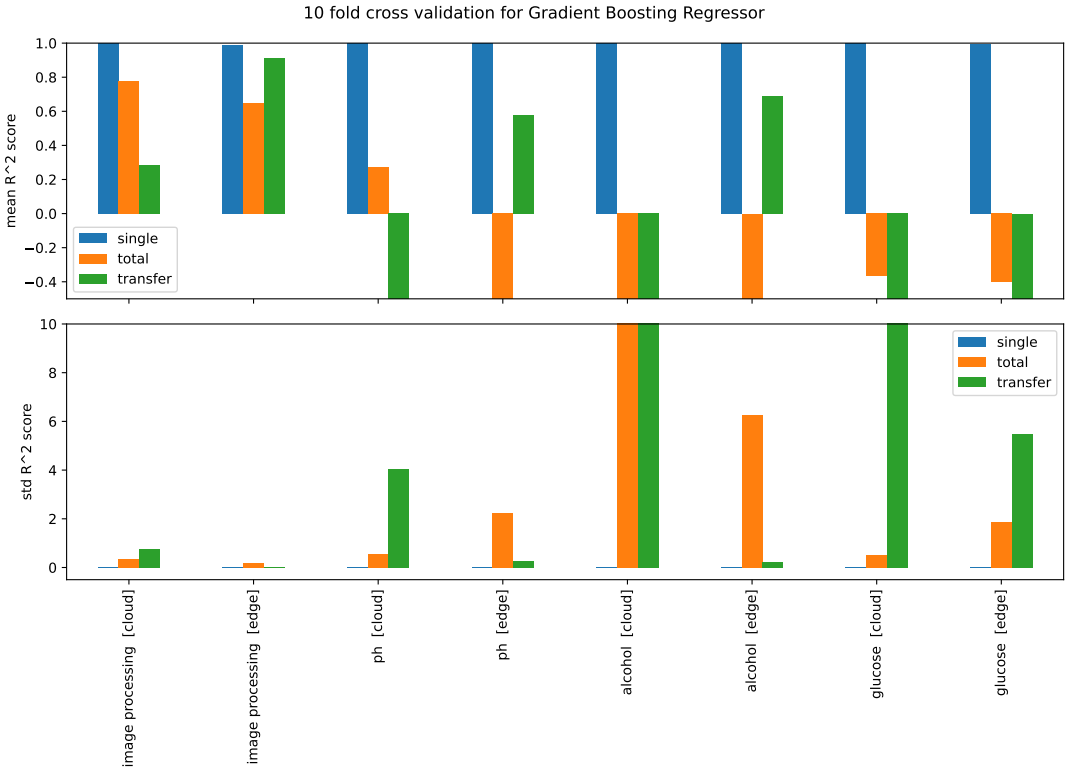


Figure 6.19: 10-fold cross-validation for the Gradient Boosting Regressor

6.3 Gradient Boosting method metrics

In this section, we analyze the chosen machine learning method, the gradient boosting regressor, in more detail. Among other things, we look into the overall performance, the importance of the features and the residual vs fit plot.

6.3.1 Overall Performance

To more accurately evaluate the overall performance of the model, we apply 10-fold cross-validation to the model and evaluate how well it performs on different subsets of the data set. With 10-fold cross-validation, we shuffle the sample data, split it into 10 groups and then for each unique group we: Take one group as our test set, use the remaining groups as our training data and fit the model using this training data. We then evaluate the model on the remaining test data. In the end, we should have 10 performance statistics for each "fold" of the data set. The mean and standard deviation of the scores from the 10-fold cross-validation test for the original and resource-intensive pipeline can be seen in figures 6.19 and 6.20 respectively.

The first thing we can see from both figures 6.19 and 6.20 is that the individual lambda duration can be accurately predicted for all lambda functions. This is likely due to the lack of variation in the duration for ph, glucose and alcohol and the fact that the duration for image processing is influenced directly by message size.

The second score we can see is the total duration. The total duration scores much better than anticipated for image processing lambda. Previously we were not able to identify any relation between the features and the total transfer time for the image processing lambda

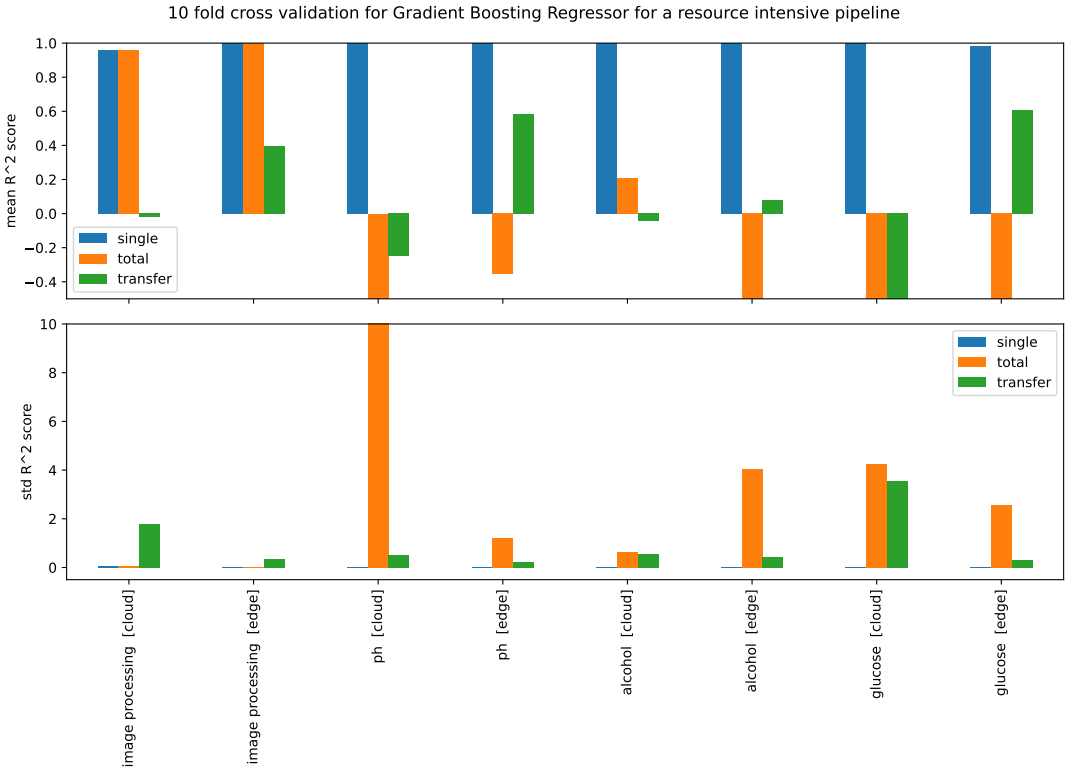


Figure 6.20: 10-fold cross-validation for the Gradient Boosting Regressor for a resource-intensive pipeline

in figures 6.7 and 6.12. Especially for the original pipeline. It seems that the model is able to identify some relation between the total duration and the input features. This score does unfortunately not reappear in the other lambda functions. The model is not able to predict the total remaining duration for those lambda functions well. However, the total remaining duration for those lambda functions is rather small, so it shouldn't influence the performance of the scheduler too much.

The last score visible is from the transfer time label. The score for the transfer time to the image processing lambda on the edge is relatively decent, with an R^2 score between 0.4 and 0.9. The transfer time to the cloud is slightly less, with an R^2 score between 0 and 0.3 for the original and resource-intensive pipeline. The transfer time for the other lambda functions does not seem to score as well. Edge transfer time is relatively good, but the transfer time to the cloud contains too much variation that cannot be explained. This is likely due to the smaller size of the message being transferred.

6.3.2 Residuals vs fit plot

To evaluate the chosen model, one of the characteristics we can look at is the residuals. Residuals are the difference between true values and predicted values. We can visualize these residuals using a residual vs fit plot, which can be seen in figures B.1, B.2, B.3 and B.4 in section B.1.1 in the Appendix. The residuals versus fit plot of the resource-intensive pipeline can be seen in figures B.5, B.6, B.7 and B.8 in section B.1.2 in the Appendix.

A residuals versus fit plot is a scatter plot of residuals on the vertical axis and estimations

of the model on the horizontal axis. This visualization can be used to detect non-linearity, outliers or unequal error variances. If the points are evenly distributed around the horizontal 0 line, then this implies that the model is a good fit. If not, then the model might have some problems.

For the original pipeline, we can see that for the individual lambda duration and transfer time, the mean of the dots are roughly centered around 0 and there is there is no nonlineariy visible. For the total duration in the cloud, this is also roughly the case. however, for the edge, the points are not evenly distributed around the 0 axis. This might indicate Heteroskedasticity, where the variance of the residuals is unequal over a range of measured values.

For the other lambda functions, there are little to no problems predicting the individual lambda duration. However, a slanted relation can be observed for the transfer time and total duration. This implies that the model not doing a particularly good job of predicting those values for these lambda functions. However, what can also be observed is that the difference in residuals is nearly always less than a second. So the problems with the model to predict small variations in short running tasks should not be a big problem in the overall performance of the scheduler.

For the resource-intensive pipeline, we can roughly see the same statistic. For image processing, everything seems to be evenly distributed around the horizontal 0 line, but for the other short running lambdas the same issue can be observed.

6.3.3 Feature importance

In this section, we look into the importance of each input feature for the machine learning model. An overview of the features and labels of this model can be found in tables 5.2 and 5.3 in section 5.2. The importance of each input feature relative to each output label for each lambda function can be seen in figures 6.21 and 6.22. Here edge single, edge total and edge transfer are the individual lambda duration, total remaining pipeline duration and transfer time to the lambda function for an edge execution respectively. Cloud single, cloud total and cloud transfer imply the single duration, total pipeline duration and transfer time to the lambda function if the respective lambda function were to be executed on the cloud.

The feature importance is equal to the probability of reaching a node weighted by the decrease in the node's impurity for each feature [38]. The probability of a node is calculated by counting the number of samples that reach the node and dividing it by the total number of samples. The node impurity is calculated using mean absolute error. The higher the value the more important the feature.

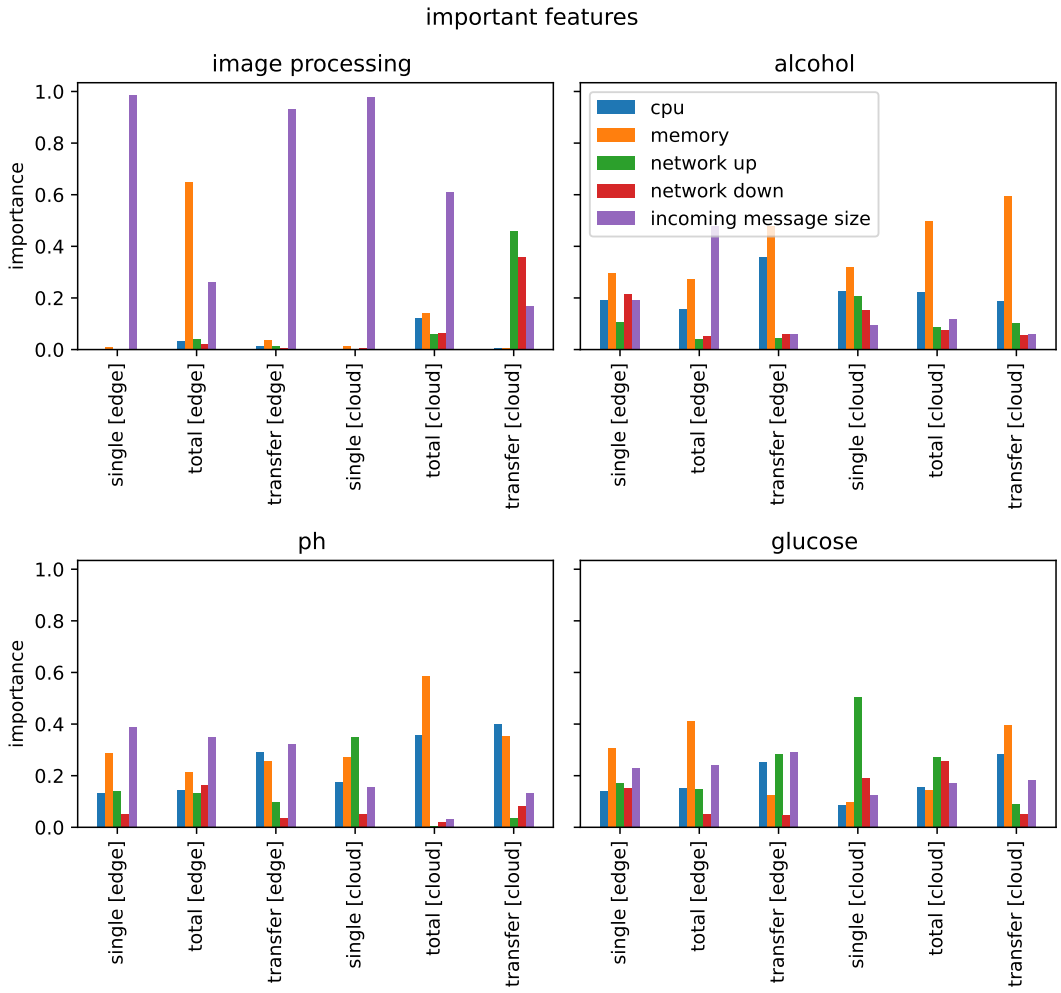


Figure 6.21: Importance of features for each output label per lambda function

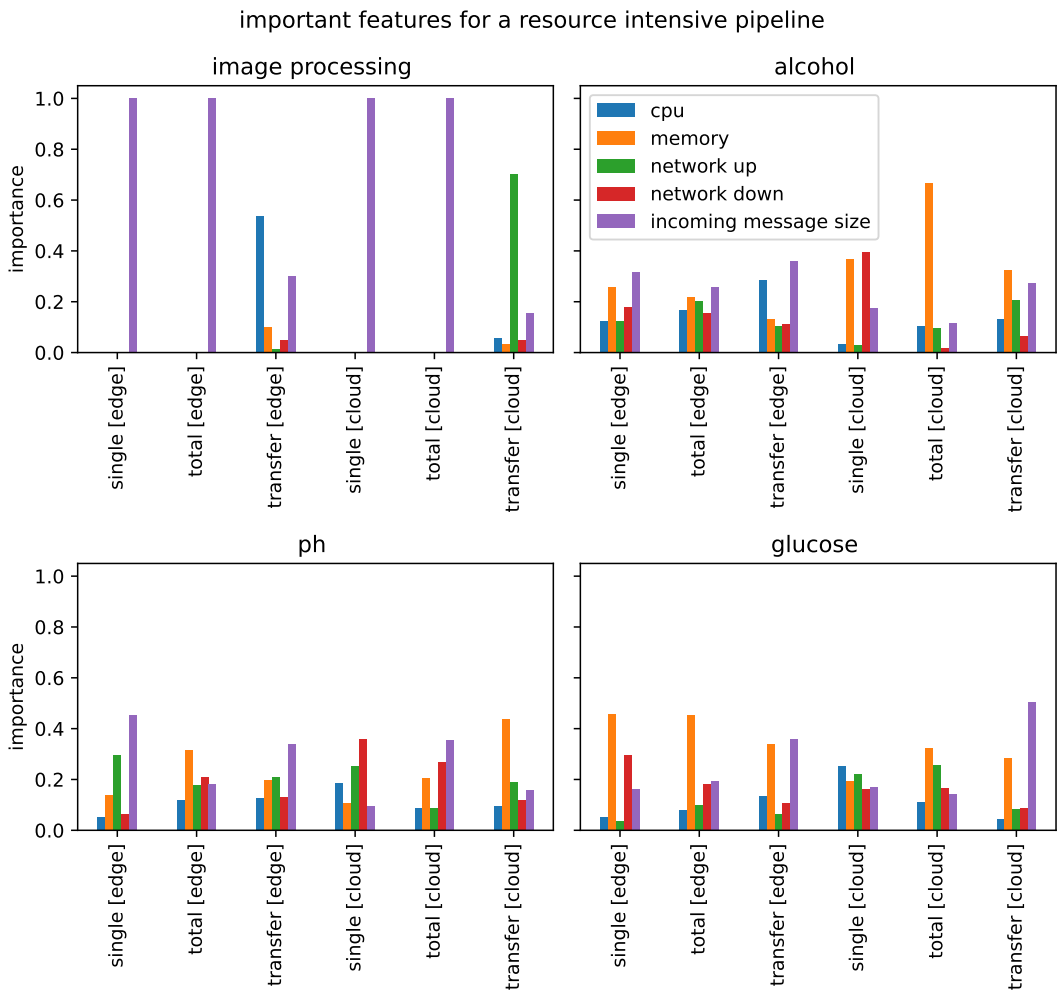


Figure 6.22: Importance of features for each output label per lambda function for a resource-intensive pipeline

From figure 6.21 we can see that the duration of the image processing lambda and the transfer time on the edge is directly influenced by the incoming message size. This relation was to be expected, since the messages being sent to the image processing lambda are rather big, ranging from 1 to 10MB in size, and the duration of the image processing lambda is directly tied to how many images the message contains. For the transfer time to the cloud, the network bandwidth also seems to play a role. This behavior falls in line with what we expected to happen in section 6.1.1.

We can see the same patterns in 6.22 for image processing lambda in a resource-intensive pipeline. However, in this case, CPU is an important metric in the prediction of transfer time within the edge device. This is in line with the observed patterns in figure 6.11 described in section 6.1.2. Namely, if the edge device is not processing anything, than the transfer time is next to nothing, however when the edge device is already busy, then it has trouble starting up another instance of the lambda function to process the data and the transfer time increases.

For the other lambda functions in this figure we can see that memory is overall relatively important for predicting the input features. However, distinct relations are less visible for these other lambda functions.

6.4 Scheduler performance

To evaluate the performance of the scheduler, we created 9 different configurations for the scheduler and measured the duration and cost of around 500 invocations. The internet speed, experiment type and number of images taken within those 500 invocations are altered the same way as with random scheduling, which can be seen in table 5.1. The different values chosen as constraints can be seen in table 6.5. These values are based on the maximum and minimum measured duration and cost for the pipeline. For example, in the original pipeline, the maximum total duration measured was 178 seconds and the minimum 12.8 seconds. We chose the constraint values for that pipeline to be approximately in that range, but never lower than the minimum duration.

Scheduling type	**value type**	**original**	**resource intensive**
cost optimization under deadline constraint	deadline (sec)	250	1000
cost optimization under deadline constraint	deadline (sec)	125	600
cost optimization under deadline constraint	deadline (sec)	60	200
cost optimization under deadline constraint	deadline (sec)	30	50
latency optimization under cost constraint	max cost ($)	0.012	1
latency optimization under cost constraint	max cost ($)	0.007	0.6
latency optimization under cost constraint	max cost ($)	0.002	0.2
latency optimization under cost constraint	max cost ($)	0.0002	0.01
random	-	-	

Table 6.5: configurations for the scheduler experiment

6.4.1 Original pipeline

We first test the performance of the scheduler on the original pipeline. The mean duration and cost over the 500 invocations for each constraint value can be seen in figure 6.23. The frequencies of a specific lambda function being offloaded to the cloud for each constraint value can be seen in figure 6.24.

A peculiar pattern can be observed in the scheduler performance graph for both latency and cost optimization. The determined optimal duration and cost seem to be unchanging and unrelated to the constraint value. This apparently happens, because there is one scheduling choice for this pipeline that is both the fastest and cheapest compared to the other alternatives: edge only scheduling.

Knowing that edge only is always the fastest and cheapest option, as previously determined in section 6.1.1, we can use this figure to evaluate the error of the model. From

figure 6.24 we can see that cost optimization perfectly scheduled everything on the edge, but scheduling with latency optimization brings with it some errors. We can see that latency optimization made the correct choice of keeping the image processing lambda on the edge, but it still chooses to place the ph, alcohol and glucose lambdas in the cloud for around 200 occurrences. Most likely the scheduler incorrectly predicted the duration of the lambda functions by a few hundred milliseconds. This error is not disastrous however, since the scheduler with latency optimization is only about 2% slower than the optimal alternative.

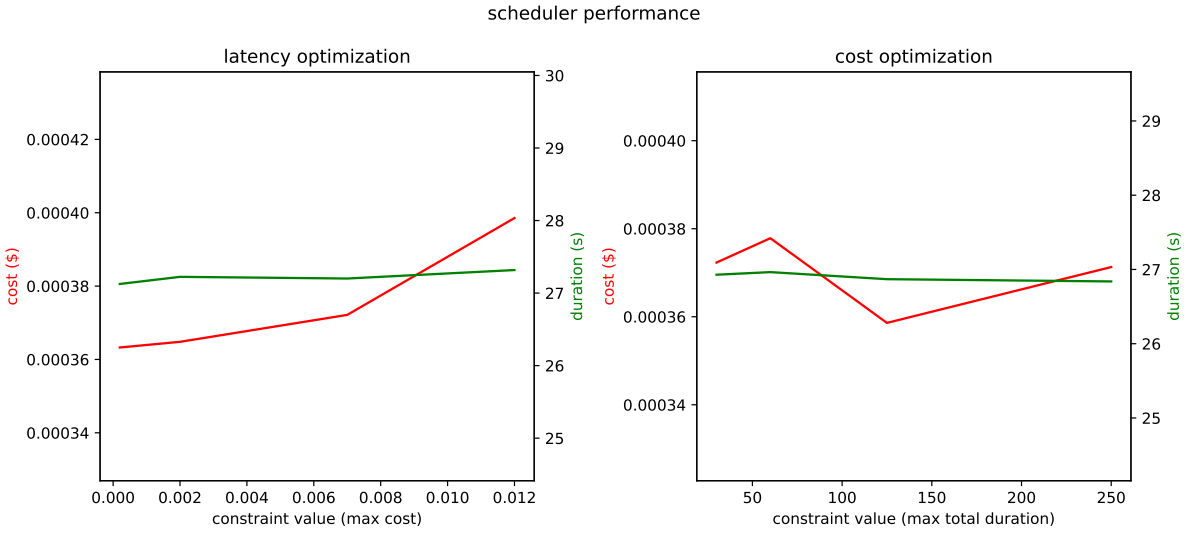


Figure 6.23: Performance of the scheduler.

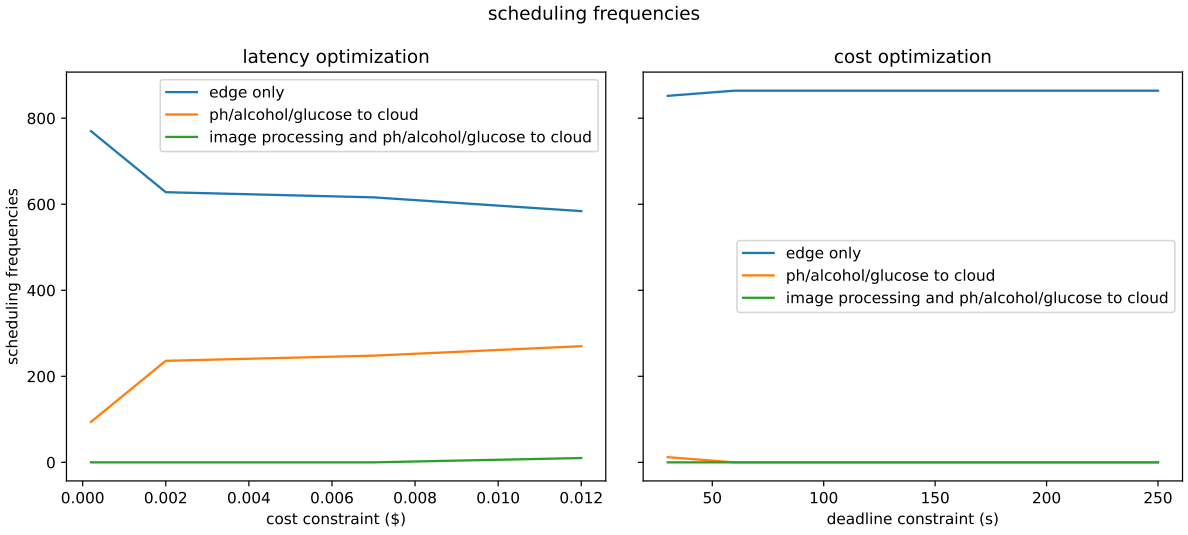


Figure 6.24: Frequencies of lambdas offloaded to the cloud.

6.4.2 Resource intensive pipeline

Following the original pipeline, we also test the performance of the scheduler on a resource-intensive pipeline. The differences between the two pipelines have been described in

section 4.4.2. The mean duration and cost over the 500 invocations for each constraint value can be seen in figure 6.25 and table 6.6. The frequencies of a specific lambda function being offloaded to the cloud for each constraint value can be seen in figure 6.26.

We can see that for this resource-intensive pipeline, the scheduler performs much better. For latency optimization, the scheduler is able to correctly decrease latency with a higher constraint value and for cost optimization, the scheduler is able to correctly decrease cost for a higher constraint value.

From figure 6.26 we can see that not one single offloading decision is picked for all constraints, but the decisions depend approximately on the given constraint value. This indicates that the scheduling decisions do depend on the available resources on the station and message size.

For example, for latency optimization with a cost constraint of \$1, we can see that sometimes cloud only scheduling is chosen and sometimes edge only scheduling is chosen, which results in an average duration of 137.3 seconds. For latency optimization with a cost constraint of \$0.01, we can see that if we were to schedule everything on the edge, the duration would be much higher: Around 196.2 seconds.

As a reference, cloud only scheduling for a resource-intensive pipeline would take approximately 152.7 seconds and random scheduling 178.3 seconds. This implies that this scheduler is able to increase performance by approximately 10,1% compared to cloud only scheduling, 30,0% compared to edge only scheduling and 23% compared to random scheduling by dynamically scheduling the lambda functions.

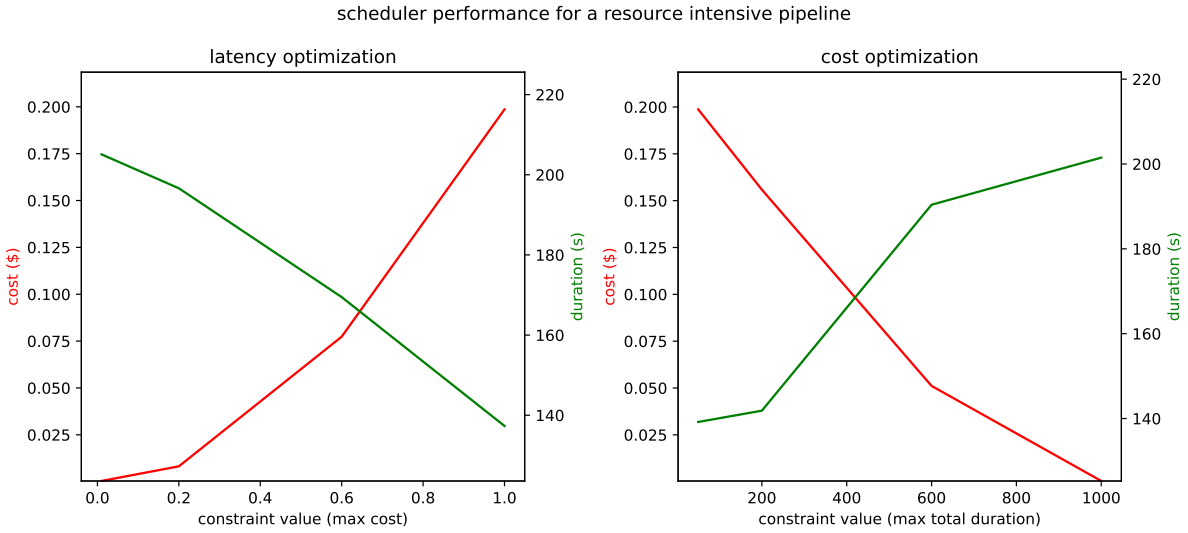


Figure 6.25: Performance of the scheduler in a resource-intensive pipeline

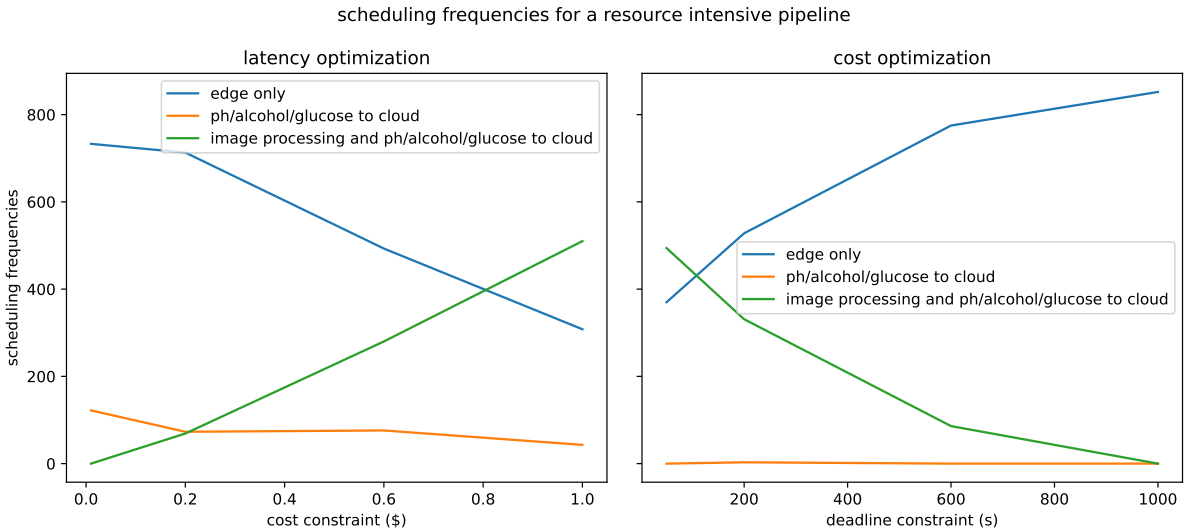


Figure 6.26: Frequencies of lambdas offloaded to the cloud in a resource-intensive pipeline.

scheduling type	constraint	duration	cost
random		178.3 sec	$0.0748
edge only		196.2 sec	$0.0004
cloud only		152.7 sec	$0.2767
latency optimization	$1	137.3 sec	$0.1986
	$0.6	169.5 sec	$0.0773
	$0.2	196.6 sec	$0.0082
	$0.01	205.1 sec	$0.0004
cost optimization	1000 sec	201.5 sec	$0.0004
	600 sec	190.4 sec	$0.0511
	200 sec	141.9 sec	$0.1559
	50 sec	139.2 sec	$0.1987

Table 6.6: Results overview for a resource intensive pipeline

6.5 Scheduler overhead

In this section we look into the overhead the scheduler adds to the deployment that is not directly visible in the results. We mainly look into the latency overhead of passing messages through the scheduler and cost overhead due to the existence of the finalizer lambda in the cloud, which collects the device performance data and retrains the models.

The added latency for sending messages to the scheduler first for each lambda function in the original pipeline can be seen in table 6.7. This table contains the average scheduling overhead caused by the duration of sending a message to the scheduler and the scheduling process itself of each lambda function, as well as the standard deviation of this measured average value, the average total transfer time to this function and the fraction size of the overhead compared to the total transfer time. From this table, we can see that the scheduling overhead is between 10% and 20% for each lambda function, and about 10.6% for the entire pipeline. This means that the scheduler should improve the average latency

	mean overhead	std overhead	mean total transfer time	fraction of mean total transfer time
to image processing	1.484 sec	0.699 sec	13.828 sec	10.7 %
to ph	0.109 sec	0.033 sec	0.476 sec	22.9 %
to glucose	0.115 sec	0.037 sec	0.553 sec	19.3 %
to alcohol	0.117 sec	0.037 sec	0.596 sec	21.1 %
to collector	0.054 sec	0.035 sec	0.459 sec	11.8 %
total duration	1.573 sec	0.720 sec	14.851 sec	10.6 %

Table 6.7: scheduling overhead versus total transfer time for lambda function in the original pipeline

by more than 10.6% over any static lambda placement strategy before it can be classified as more efficient.

The finalizer lambda is a required component for the scheduler to operate. It is responsible for collecting the device metrics and storing them in a database, as well as retraining the model when necessary. The finalizer has an average duration of 0.016 seconds, which according to equation (5.4) with a memory configuration of 128Mb and ephemeral storage of 512Mb is roughly equal to $ 0.00011 per request.

The cost of writing data to storage should also be accounted for. Per request about 1Kb worth of data are stored in an SQL server. According to AWS RDS pricing model [11], storing 1Kb of data in the database would increase the monthly cost by $ $0.133 \cdot 10^{-6}$. If only the last thousand data points were to be kept for training the model, then storage would cost around $0.000133 a month. This is around the price of a single pipeline invocation, and we can therefore disregard the storage price if old data is occasionally purged from the database.

This means that the scheduler has to optimize the price such that it saves more than approximately $ 0.00011 per request. if not, then the scheduler is not able to optimize the price sufficiently.

Apart from these costs, there is also the cost of retraining the model. This is however an operation that only occurs periodically, if at all. Moreover, if the deployment does not change, the model doesn't have to be retrained at all. We therefore disregard the cost associated with this operation.

6.6 Automated deployment script

To quantify the optimization of the development process that the automated development script offers, we conducted two experiments where we measured the approximate time necessary to deploy the original pipeline for the first time or update the functions of an existing pipeline using the automated deployment script or AWS provided alternatives. Without the automated deployment script, these alternatives would likely be the AWS User Interface (UI) or the AWS Command Line Interface (CLI). For these experiments we assumed that the (updated) code for the lambda functions is already written, a rela-

tion database instance already exists and all the security credentials have already been configured. An overview of these results can be seen in table 6.8.

	initiating deployment	updating existing deployment
AWS UI	51 min	15 min
AWS CLI	32 min	10 min
Automated script	22 min	8 min

Table 6.8: The approximate duration of each lambda function deployment method for initiating a new deployment or updating an existing one.

The duration for initiating the deployment for the first time was measured by creating 5 serverless functions, identical to the image processing, pH, glucose, alcohol and collector lambda, deploying them on the cloud and edge and configuring their communication channels. The duration of updating lambda code was measured by updating the code of only the image processing and pH lambda and deploying them to the cloud and edge. The experiments were undertaken with knowledge of the necessary steps beforehand and for AWS CLI a portion of the needed commands was available for a copy-paste action.

From table 6.8 we can see that the automated script always beats the other two deployment strategies in duration. Note however, that the values in this table are the duration of the initiation or update process from start to finish. Meaning that a large portion of this duration for the automated script requires no developer to be present and essentially happens in the background. For initiating the deployment with the automated script, only the initial configuration needs to be written, which in this case took approximately 10 minutes and for updating an existing deployment, the script only needs to be started, which takes roughly 3 seconds at most. For the two other alternatives, constant developer interaction is required over the entire duration.

What is more, the duration of AWS UI and CLI might in reality be even higher, since these alternatives are more prone to human error. This increases the odds of the deployment process needing to be restarted again. These alternative deployment methods also have a higher learning curve for deploying together with the scheduler, since the developer needs to know what steps to perform and how to configure the scheduler. This might make the initial development time significantly longer using AWS UI and AWS CLI.

Chapter 7

Discussion

7.1 Pipeline

Before we analyze the performance of the scheduler, it is important to understand the characteristics of the pipeline it is measured on. Certain characteristics of a pipeline might make it easier for the scheduler to schedule lambda functions and display good performance, while the absence of these characteristics might make the scheduler perform poorly. In section 6.1 we describe the relations of lambda functions and the characteristics of the pipeline with respect to the input features and output labels of the model for both the original and resource-intensive pipeline.

The first important characteristic of our pipeline is the fact that our longest running lambda function duration is dependent on message size. This resulted in the machine learning model being able to predict the image processing lambda duration relatively well and will therefore increase the performance of the scheduler. Other pipelines, however, with lambdas not depending on message size, but rather some value hidden within the context of the message, may not perform as well in combination with the scheduler.

second of all, our longest running lambda function is first in the pipeline. This means that the total remaining duration can be predicted with a relatively decent score and even if the scheduler predicts it incorrectly, it will have little impact on the final scheduling decision.

However, if the longest running lambda function were to be in the middle or the end of the pipeline, the machine learning model will likely have much more trouble predicting the total remaining duration. In this case, the total remaining duration will be highly influenced by one lambda function somewhere in the pipeline, for which we do not know the available CPU, memory or network bandwidth nor the incoming message size. So the model may not have enough information to make an accurate prediction. Future research may focus on testing the scheduler on a pipeline with this characteristic to determine if the scheduler can also optimize such a pipeline.

Last of all, for the resource-intensive pipeline specifically, the difference in duration for our lambda functions is rather significant. The duration of the image processing lambda is about 100 times longer than that of the ph, glucose or alcohol lambda. This means that the scheduling of image processing lambda will have significantly more influence on the

final performance optimization of the scheduler than ph, glucose or alcohol. Moreover, if ph, alcohol or glucose is incorrectly scheduled, it will likely not be noticeable, since this will only influence the total duration by a few hundred microseconds at most. This implies that our results may not indicate similar results for a more balanced multi-stage pipeline.

Because of these characteristics, the measured performance of the scheduler may not apply to all pipelines. The scheduler may completely fail to optimize a pipeline if the input features do not imply the individual function duration, transfer time or total remaining duration of a pipeline. To increase the confidence in the performance of the scheduler, it will need to be tested on other pipelines with other characteristics in future work.

7.2 Machine learning model

In this section, the performance of the machine learning model for both the original and resource-intensive pipeline is discussed. We look at the quality of the model and discuss if it is a good fit. We can see the 10-fold cross-validation scores for the gradient boosting models per lambda function in figures 6.19 and 6.20.

One thing we can see is that, for both figures, the individual lambda duration can be accurately predicted for all lambda functions. The reason for this is a lack of variation in the duration for ph, glucose and alcohol and the fact that the duration for image processing is directly influenced by message size. The transfer time on the other hand seems to be a bit harder to predict. The score for the transfer time to the image processing lambda on the edge is relatively decent, with an R^2 score between 0.4 and 0.9, while the transfer time to the cloud is slightly less, with an R^2 score between 0 and 0.3, for the original and resource intensive pipeline. Note that a low R^2 score value doesn't necessarily mean that the model is bad. Any R^2 score higher than 0 indicates that the model is better than taking the average from the data set. A score lower than 0 implies you might as well just take the average of the data set. So with the given scores the scheduler should roughly still be able to make correct decisions, although some error can be expected.

The transfer time for the other lambda functions does not seem to score as well. Edge transfer time is overall relatively good, but the transfer time to the cloud contains to much variation that cannot be explained. This is likely due to the smaller size of the message being transferred compared to the messages coming into the image processing lambda. In any case, this will likely not be a problem for the scheduler, since the transfer time to these lambda functions only varies by a few seconds.

The total duration scores much better than anticipated for image processing lambda. This is likely due to the fact that the longest running lambda function is first in the pipeline. The image processing lambda's total remaining duration is including its own duration, meaning that the few seconds in difference in total duration caused by the ph, glucose or alcohol lambda is negligible in the end result. This is even more visible for the resource-intensive pipeline, where image processing lambda is 100x slower than ph, glucose or alcohol. The total duration for the other lambda functions does not score well at all, likely due to the variation added by transfer time from ph to the collector lambda and the cold/warm start problem.

We can see that the model is not perfect, but considering the characteristics of the lambda

functions and the corresponding scores. The scheduler should be able to make decent decisions based on the predictions made by the model. Future research could look into the performance of the machine learning model on other pipelines, with different characteristics.

7.3 Scheduler

In this section, the performance of the scheduler for both pipeline versions is discussed. For the main part, we try to answer the research question stated in chapter 1: *How do we determine what tasks should be offloaded to the cloud and what should remain on the edge based on a cost or deadline constraint?*

In this paper, we try to answer this research question by constructing a machine learning model that predicts the individual lambda duration, total remaining pipeline duration and transfer time to the lambda function based on available resources and message size and we either pick the option with the lowest duration or cost based on an optimization strategy and a specified constraint. To determine whether or not this solution answers the research question, we have to evaluate whether or not the tasks are offloaded to the cloud or the edge based on the given constraint and whether this decision improves either cost or latency relative to the optimization strategy for an increasing constraint value.

For the original pipeline, we can see that this is simply not the case. In figure 6.23 and figure 6.24 we can see that the scheduling decision does not change according to the constraint value, nor do the average cost and duration change over constraint values. This is because this pipeline has only one optimal scheduling decision: edge only. Both the cost and latency are optimal for the placement of all functions on the edge. The affinity with the edge is likely due to the short running nature of the pipeline. The current device has enough resources available to accommodate for edge only execution.

Furthermore, one could argue that the scheduler may even increase latency and cost. The added overhead of the scheduler counts as roughly 10% of the total duration and the finalizer lambda costs about $0.00011 per invocation. Since the scheduler is not able to optimize performance or cost, the extra overhead and finalizer cost will only make the pipeline slower and costlier than it needs to be.

This does not mean that the scheduler is performing poorly however, this simply means that there is no need for a dynamic scheduler in this pipeline. The scheduler could still be used for this pipeline as an analysis tool. Developers could test their pipeline using the scheduler, and evaluate what static placement is most optimal in their use case. When that has been determined, the developer can remove the scheduler from the pipeline and implement the static placement manually. The scheduler could still improve the performance of the pipeline this way, even though the functions are not dynamically scheduled.

To evaluate the scheduler on a pipeline that does not only have an affinity with the edge, we slightly increase the resources required to run the pipeline by removing some optimization in the code. The exact changes can be seen in section 4.4.2. For this resource-intensive pipeline, we see more promising results. In figure 6.25 and figure 6.26 we can at least see that the cost and latency are much more dependent on the constraint value, as is the placement of the lambda functions. Furthermore, for latency optimization

we can see that the duration is correctly decreasing for an increasing constraint value of maximum invocation cost and for cost optimization we can see that the cost is correctly decreasing for an increasing constraint value of maximum total duration. This implies that the scheduler is able to correctly schedule functions for latency and cost optimization under some predefined constraint.

This is a good indication of the performance of the scheduler, but we still need to determine whether the scheduling decisions are approximately optimal. To determine this, we can compare the result of scheduling with latency optimization and cost optimization with random, edge only and cloud only scheduling. The improvement per optimization strategy and constraint value can be seen in tables 7.1 and 7.2. In these tables a positive value represents an overall improvement and a negative value represents that the optimization strategy is performing worse in that respect.

The performance of cost optimization is easily checked: If the scheduler is scheduling every lambda function on the edge then cost should be minimal. From the figures we can indeed see that for a constraint value of 1000 seconds for cost optimization, which is much higher than any recorded pipeline duration and thus implies that the cost is optimized regardless of pipeline duration, the scheduling decision is indeed edge only, and therefore the cost is minimal and thus optimized.

In addition to that, the latency should improve with a decreasing constraint value. From table 7.1 we can indeed see that this is the case. The scheduler is even able to slightly optimize latency given a very small constraint value.

To evaluate the optimization of latency, we need to compare the results to edge only, cloud only and random scheduling. For latency optimization with a constraint value of \$1 , which is higher than any recorded invocation cost and will therefore optimize latency regardless of cost, we measure a duration of 137.3 seconds. Edge only scheduling has an average duration of 196.2, cloud only scheduling an average duration of 152.7 and random an average duration of 178.3. This implies that this scheduler is able to decrease latency with approximately 10,1% compared to cloud only scheduling, 30,0% compared to edge only scheduling and 23% compared to random scheduling by dynamically scheduling the lambda functions, as can be seen in table 7.2.

	improvement on latency			**improvement on cost**		
constraint	random	edge only	cloud only	random	edge only	cloud only
1000 sec	-13.0%	-2.7%	-31.9%	99.5%	0.0%	99.9%
600 sec	-6.8%	3.0%	-24.7%	31.7%	$-1.3 \cdot 10^4$%	81.5%
200 sec	20.4%	27.7%	7.1%	-108.4%	$-3.9 \cdot 10^4$%	43.7%
50 sec	21.9%	29.1%	8.9%	-165.6%	$-5.0 \cdot 10^4$%	28.2%

Table 7.1: Improvements with cost optimization

	improvement on latency			**improvement on cost**		
constraint	random	edge only	cloud only	random	edge only	cloud only
\$1	23.0%	30.0%	10.1%	-165.5%	$-5.0 \cdot 10^4$%	28.2%
\$0.6	4.9%	13.6%	-11.0%	-3.3%	$-1.9 \cdot 10^4$%	72.1%
\$0.2	-10.3%	-0.2%	-28.7%	89.0%	$-2.0 \cdot 10^2$%	97.0%
\$0.01	-15.0%	-4.5%	-34.3%	99.5%	0.0%	99.9%

Table 7.2: Improvements with latency optimization

These numbers are an excellent indication that the scheduler is able to improve performance over statically placed functions, however we need to evaluate if the scheduler is able to improve performance regardless of the scheduling overhead. Since the measured duration for latency optimization is faster than edge only and random scheduling by more than 10%, we can say that the scheduler is able to optimize the performance even with the scheduling overhead taken into account compared to those static scheduling methods. Compared to cloud only scheduling, the optimization is roughly equal to the overhead caused by the scheduler. However, cloud-only scheduling has an average cost of \$0.2767 and latency optimization with high constraint a cost of \$0.1986. So the scheduler is able to optimize cost by 28% with regard to cloud only placement, even with the finalizer lambda cost overhead taken into account. So this means that:

- Compared to cloud only scheduling, the scheduler with latency optimization, highest constraint and scheduling overhead taken into account, is able to keep the same latency and optimize cost with 28%.
- Compared to edge only scheduling, the scheduler with latency optimization, highest constraint and scheduling overhead taken into account, is able to optimize the latency by 20%. account.
- Compared to random scheduling, the scheduler with latency optimization, highest constraint and scheduling overhead taken into account, is able to optimize the latency by 13%.
- Compared to cloud only scheduling, the scheduler with cost optimization, highest constraint and cost overhead taken into account, is able to reduce cost by 99.9%.
- Compared to edge only scheduling, the scheduler with cost optimization, highest constraint and cost overhead taken into account, is able to roughly maintain the same cost and latency.
- Compared to random scheduling, the scheduler with cost optimization, highest constraint and cost overhead taken into account, is able to optimize the cost by 99.5%.

So the scheduler is able to optimize latency with **13%-20%** or reduce cost with **28%**, with the scheduling overhead taken into account.

We can conclude from these results that the scheduler does not work on every pipeline. If a static placement of function is both optimal in cost and latency, then using a scheduler will only decrease performance. This is often the case for short running and lightweight applications where the edge device has enough resources to run them. Or when there is

a unschedulable lambda function, like our station lambda function, that bottlenecks the deployment such that it is unlikely for the edge device to be overloaded with requests.

However, for longer running resource-intensive pipelines, the scheduler is able to schedule the lambda functions correctly according to the chosen optimization strategy and given cost or deadline constraints. It is even able to optimize the latency and cost with the scheduling overhead taken into account.

7.4 Automated deployment script

In this section, our implementation of an automated deployment script for the scheduler and lambda functions is discussed. We shortly reiterate how we refined the development process and argue why it is an improvement over the alternative. We also discuss some shortcomings which may be able to be fixed in the future. Mainly, we try to answer the research question: *How do we minimize the time consuming process of configuring and deploying serverless functions to the cloud and to the edge?*

To minimize the time consuming development process, we implemented an automatic deployment script that configures the scheduler, the lambda function and the communication channels according to the specifications given by the developer in a .yaml file. The grammar and an example of the configuration file can be found in Appendix A. The only thing the developer will have to define is the scheduling optimization strategy and constraint, assigned resources, environment and message channels for the lambda functions and security credentials. All the deployment steps are automatically performed by the script.

The first difference between the creation of lambda functions via the AWS CLI or UI and the .yaml configuration lies in the first time setup of the entire pipeline. In both cases, quite some initial configuration is required to set up the pipeline. The only difference is that via the AWS CLI or UI, the lambda function first has to manually be configured for the cloud and then on a separate interface, it needs to be configured for the edge. This means manually running through commands or pages to complete a task that could be completed at once. Our deployment script provides one central file in which everything can be defined.

The second difference is where the deployment script really shines. That is the automatic updating of serverless functions. If certain code changes in the pipeline, the developer normally will have to manually identify what lambda functions are influenced, upload the code to AWS manually and configure the lambda functions in the cloud and on the edge separately. Depending on the changes to the pipeline, the developer will have to perform these actions for every single lambda function. On the other hand, for our deployment script, none of these steps are necessary. The only thing the developer will need to do is specify the names of the lambda functions to update and run the script. Uploading the code, configuring the cloud, configuring the edge and updating the scheduler is all performed automatically.

From the comparison with AWS CLI and the AWS UI in section 6.6 we noticed that the automated development script is always faster for creating an edge cloud deployment. The data of this comparison can be seen in table 6.8. Furthermore, the data gathered here is the total duration from start to finish. Meaning that a large portion of the

automated script duration requires no developer to be present and essentially happens in the background. For the two other alternatives, constant developer interaction is required over the entire duration. In addition to that, AWS CLI and AWS UI have a greater learning curve for creating an edge cloud deployment, since knowledge of all the manual steps required is necessary. This fact can slow down the initial development process and introduce more room for human error.

There are however some shortcomings to our implementation. First of all, the creation of the relational database for the finalizer is not created automatically. We made the choice to not include this in the automated deployment script, since a lot of customization can go into creating a database and it is likely to be used for other ends as well, not just for storing the pipeline metrics. Rather, we let the user create the database via the AWS portal and let them insert the database connection info as a secret.json file. Furthermore, creating a database via .yaml configuration instead of an interactive frontend would likely not have been beneficial for the development process.

Second of all, the current automated deployment script only supports a subsample of the services that AWS offers. The automated deployment script could be extended to support communication channels like: AWS SNS, AWS IoT Greengrass stream manager and HTTPS. It could also be extended to include the configuration of non-lambda compute instances in the cloud or on the edge, although this would require substantial modification of the deployment script, the scheduler and possibly Greengrass itself.

Third of all, AWS stepfunctions or another graph-based interface could be utilized to visualize the relation between lambda components. Currently, the developer is only able to configure the deployment via a configuration lambda file. A friendly user interface can be utilized to optimize the development experience by making it faster and possibly more pleasant.

Chapter 8

Future work

In this section we list some of the challenges and open questions that still remain within the research area of serverless edge computing. We describe how our implementation tries to tackle them and what future research may focus on to solve these issues.

First of all, the scheduler should be tested on more than one pipeline. We tested our scheduler on one specific pipeline, or to be more precise, a regular and a resource intensive version of one specific pipeline. This gives us some indication of the performance of the scheduler, but to get a better picture of the general performance of the scheduler, it will need to be tested on a wider verity of different pipelines. Future research can focus on creating multiple deployments and pipelines to test the scheduler. Furthermore, more focus can be put into making a universal set of deployments and pipelines for which multiple serverless edge schedulers in the literature can be tested, so that they can be reliably and accurately compared with one another.

Second of all, future research can focus on improving the machine learning model by implementing content aware scheduling. For example, our model was able to predict the individual lambda duration with a near 99% score. This was due to the fact that the duration was either constant or directly related to message size. However, in most other systems this does not necessarily have to be the case. If the duration of a lambda function is not related to message size, or any of the other input features of the model, the model will presumably not be able to predict the individual lambda duration well. The duration may not be directly tied to message size, but to the message content, like a parameter settings or values in a JSON message. One could let the developer define important parameter values in the message, which the model should pay attention to, to increase the model score. However, this would increase the strain on the development process significantly, which reintroduces a problem serverless functions tries to solve. Future research could focus on implementing a combination of text analysis and machine learning to retrieve relevant features from the incoming message and predict the individual lambda durations, transfer times and total remaining execution time. This could potentially increase the score of predicting the individual lambda durations in those specific pipelines, and would not put any further strain on the development process of the pipeline.

Third of all, future research could focus on solving the cold/warm start problem. In this research we did not specifically try to solve this problem, rather we would point out its existence and evaluate the ability of the scheduler to incorporate warm and cold starts in

its decision making process. Currently, AWS provides no endpoints to retrieve information about the running state of a function and whether it is already running or still needs to start up. In addition to that, AWS does not provide the ability to customize the time a serverless function can idle in a warm state, apart from an " always-on" setting. If AWS were to provide this functionality in the future, then future research could focus on incorporating the available information into the model and potentially increasing the ability of the model to predict a cold/warm start in the cloud.

Fourth of all, to alleviate the strain of developing an edge cloud deployment, a graphical user interface can be used. We created an .yaml configuration file in which the developer can easily define its entire pipeline and all the other configuration and creation steps will be completed automatically. Although this feature may help, the development process could be improved by introducing a graphical interface to create the configuration, like the AWS stepfunctions [11] service. AWS stepfunctions allows the developer to define the pipeline as a directional graph, where the nodes are the serverless functions and the lines represent the communication paths. AWS stepfunctions currently does not support support the right functionality to be used in edge cloud computing, since the graph can only be executed from the starting node, meaning we cannot run only half of the graph if we chose to do the first on the edge. AWS stepfunctions also does not allow for setting custom configurations necessary for edge execution. Future research can focus on creating a graphical user interface to alleviate the developing strain even more, or if AWS stepfunctions becomes more customizable, implement it there.

Last of all, the scheduler can be written in another language to improve overall performance. Currently, the scheduler itself is written in Python. Although python is a very flexible and maintainable programming language, it is not know for its performance. The memory consumption can be as much as twice as high and the run time 5 to 10 times as long compared to its C and C++ counterpart [39]. To reduce the scheduling overhead in a pipeline, future research can focus on rewriting the scheduler in C++.

Chapter 9

Conclusion

This paper introduces a dynamic serverless edge scheduler that optimizes either cost or latency according to a predefined deadline or cost constraint. This scheduler is evaluated on a real word use case; an image processing pipeline for a company called SG Papertronics. In the next paragraphs, each of the research questions from chapter 1 is discussed.

The first research question we presented was: **How do we determine what tasks should be offloaded to the cloud and what should remain on the edge based on a cost or deadline constraint?** To answer this research question we created a dynamic scheduler that predicts the total remaining pipeline duration, individual lambda duration and transfer time using the available resources on the device, the network bandwidth and the message size, in order to make a decision whether or not to offload a lambda function to the cloud. We then evaluated this scheduler on two versions, the original and a resource-intensive version, of an image processing pipeline.

From this evaluation, we noted that the scheduler was able to optimize both cost and latency for a resource-intensive pipeline. The latency was improved by 10%-30% compared to the situation if the same lambda functions were to be placed statically on the cloud or statically on the edge. Considering the scheduling overhead is about 10% of the entire pipeline, the scheduler is able to improve latency by 0%-20% compared to a pipeline where no scheduler is present. Compared to cloud only static placement, the scheduler is able to reduce cost by 28% while maintaining an approximate equal duration.

We also determined that the scheduler is not able to optimize every pipeline. If a static placement of function is both optimal in cost and latency, then using a scheduler will only decrease performance. This was the case for our original pipeline, where the placement of functions on the edge was always optimal in cost and latency. In those cases, the scheduler could still prove beneficial as an analysis tool, to determine what static placement is most optimal for the lambda functions. The developer can then manually place them on either the cloud or the edge, without including the overhead of dynamically scheduling those functions.

The second research question was: **How do we minimize the time-consuming process of configuring and deploying serverless functions to the cloud and to the edge?** To answer this research question, we created an automated deployment script for deploying serverless functions to the edge and cloud. This script is able to optimize the deployment process by removing a great portion of manual steps necessary in the initial stage of development and any future stage for updating the deployment. We discussed that the automated deployment script is able to minimize the time-consuming development process compared to other alternatives like AWS CLI and AWS UI. However, we also noted that our deployment script has a few shortcomings, like the lack of a visual tool and limited support for a number of services within AWS.

Bibliography

[1] Yucong Duan, Qiang Duan, Xiaobing Sun, Guohua Fu, Nanjangud C. Narendra, Nianjun Zhou, Bo Hu, and Zhangbing Zhou. Everything as a service (xaas) on the cloud: origins, current and future trends. *Services Transactions on Cloud Computing*, 4(2), 2016.

[2] Peter Middleton, T Tully, J Hines, Thilo Koslowski, Bettina Tratz-Ryan, K Brant, Eric Goodness, Angela McIntyre, and Anurag Gupta. Forecast: Internet of things-endpoints and associated services, worldwide, 2015. *Gartner Inc., Stamford, CT, USA, Tech. Rep. G*, 290510:57, 2015.

[3] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[4] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE Access*, 8:85714–85728, 2020.

[5] Agus Kurniawan. *Learning AWS IoT: Effectively manage connected devices on the AWS cloud using services such as AWS Greengrass, AWS button, predictive analytics and machine learning.* Packt publishing, 2018.

[6] Anirban Das, Shigeru Imai, Mike P. Wittie, and Stacy Patterson. Performance optimization for edge-cloud serverless platforms via dynamic task placement. *CoRR*, abs/2003.01310, 2020.

[7] Gustavo André Setti Cassel, Vinicius Facco Rodrigues, Rodrigo da Rosa Righi, Marta Rosecler Bez, Andressa Cruz Nepomuceno, and Cristiano André da Costa. Serverless computing for internet of things: A systematic literature review. *Future Generation Computer Systems*, 128:299–316, 2022.

[8] Trang Quang and Yang Peng. Device-driven on-demand deployment of serverless computing functions. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 1–6, 2020.

[9] Saqib Rasool Chaudhry, Andrei Palade, Aqeel Kazmi, and Siobhán Clarke. Improved qos at the edge using serverless computing to deploy virtual network functions. *IEEE Internet of Things Journal*, 7(10):10673–10683, 2020.

[10] István Pelle, Francesco Paolucci, Balázs Sonkoly, and Filippo Cugini. Latency-sensitive edge/cloud serverless dynamic deployment over telemetry-based packet-optical network. *IEEE Journal on Selected Areas in Communications*, 39(9):2849–2863, 2021.

[11] Andreas Wittig Michael Wittig. *Amazon Web Services in Action*. Simon and Schuster, 2018.

[12] Danilo Poccia. *AWS Lambda in Action: Event-driven serverless applications*. Simon and Schuster, 2016.

[13] Warren Gay. *Raspberry Pi Hardware Reference*. Apress, USA, 1st edition, 2014.

[14] SG Papertronics. Beer-o-meter, 2021.

[15] SG Papertronics. Sg papertronics, 2021.

[16] Duarte Pinto, João Pedro Dias, and Hugo Sereno Ferreira. Dynamic allocation of serverless functions in iot environments. *CoRR*, abs/1807.03755, 2018.

[17] Hassan B. Hassan, Saman A. Barakat, and Qusay I. Sarhan. Survey on serverless computing. *Journal of Cloud Computing*, 10(1):39, Jul 2021.

[18] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Sock: Rapid task provisioning with serverless-optimized containers. In *USENIX Annual Technical Conference*, 2018.

[19] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 158–164, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188, 2018.

[21] Tarek Elgamal. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312, 2018.

[22] Gareth George, Fatih Bakir, Rich Wolski, and Chandra Krintz. Nanolambda: Implementing functions as a service at all resource scales for the internet of things. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 220–231, 2020.

[23] Chunglae Cho, Seungjae Shin, Hongseok Jeon, and Seunghyun Yoon. Qos-aware workload distribution in hierarchical edge clouds: A reinforcement learning approach. *IEEE Access*, 8:193297–193313, 2020.

[24] István Pelle, János Czentye, János Dóka, András Kern, Balázs P. Gerő, and Balázs Sonkoly. Operating latency sensitive applications on public serverless edge cloud platforms. *IEEE Internet of Things Journal*, 8(10):7954–7972, 2021.

[25] Michael Zhang, Chandra Krintz, and Rich Wolski. Edge-adaptable serverless acceleration for machine learning internet of things applications. *Software: Practice and Experience*, 51, 12 2020.

[26] Bin Cheng, Gürkan Solmaz, Flavio Cirillo, Ernö Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa. Fogflow: Easy programming of iot services over cloud and edges for smart cities. *IEEE Internet of Things Journal*, 5(2):696–707, 2018.

[27] Bin Cheng, Jonathan Fuerst, Gurkan Solmaz, and Takuya Sanada. Fog function: Serverless fog computing for data intensive iot services. In *2019 IEEE International Conference on Services Computing (SCC)*, pages 28–35, 2019.

[28] David Jensen. *Beginning Azure IoT Edge Computing: Extending the Cloud to the Intelligent Edge*. Apress, 1st edition, 2019.

[29] Michele Sciabarrà. *Learning Apache OpenWhisk*. O'Reilly Media, Inc., 2019.

[30] Agus Kurniawan. *Learning AWS IoT*. Packt Publishing, 2018.

[31] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[32] Luciano Ramalho. *Fluent Python*. O'Reilly Media, Inc., 2015.

[33] shorewall tcdevices. wondershaper unix tool, 2012.

[34] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1, 1995.

[35] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[36] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics Data Analysis*, 38(4):367–378, 2002. Nonlinear Methods and Data Mining.

[37] S Madeh Piryonesi and Tamer El-Diraby. Data analytics in asset management: Cost-effective prediction of the pavement condition. *Journal of Infrastructure Systems*, 26, 01 2020.

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[39] Lutz Prechelt. An empirical comparison of c, c++, java, perl, python, rexx and tcl. *IEEE Computer*, 33(10):23–29, 2000.

Appendices

Appendix A

deployment configuration

A.1 Configuration grammar

The following scheme depicts the grammar for the configuration .yaml file. This grammar also includes the default values used for empty fields. The YAML scheme is divided into 4 sections: First of all, the general section, where the user should specify the AWS environment, region, the architecture of the edge device (arm or x86) and other AWS resources. Second of all, a dictionary of lambda functions and their configuration. The key of this dictionary is the lambda function's name and the value the configuration. Third and fourth of all, the optional modifications for the configuration for the scheduler and finalizer. A default configuration is already set for the scheduler and finalizer, but the developer could modify this default configuration using these fields.

For each lambda function, a number of configurations can be set:

- **no_lambda** can be set if the lambda function should be deployed as a component, not as a lambda function. Setting this to true means that the component can only be executed on the edge and cannot be scheduled.
- **component name** should be set to as an identification of the lambda for edge deployment.
- **handler** should be a path to the code of the function to be executed.
- **package type** indicates in what format the lambda should be uploaded, via zip file or docker image. Currently, for edge execution, only zip is supported. However, lambda functions running in the cloud can be deployed as a docker image.
- **package_metadata, local_event_topics and mqtt_event_topics** do not have to be specified. The automatic deployment script will automatically fill in these values according to the rest of the configuration. However, they can be set if the developer wishes to add extra endpoints to the edge functions which circumvent the scheduler.
- In **destinations** the developer should specify what lambda function the results should be sent to. The developer could also specify what protocols to use, but this should not be necessary. The scheduler will be able to automatically pick the best protocol for the job according to the message size given the default protocol

configuration.

- The **cloud** and **edge** fields depict the lambda configuration for the cloud and on the edge respectively. All fields and default values should be a one-to-one representation of the AWS lambda and AWS Greengrass configurations.
- Via the **requirements** field, a developer can give a custom test file containing external python libraries.
- The **dependencies** field allows the developer to attach custom Greengrass dependencies to the lambda component.
- The **placement** field allows the developer to specify if the lambda function should be scheduled, or should only be run on either the edge or in the cloud.

```
general:
  aws:
    thing_group_arn: str
    s3_bucket: str
    sqs_arn: str
    architecture: str
    region: str
    runtime: str
    lambda_role: str
    ecr_uri: Optional[str]
  scheduling:
    type: Enum[random,edge_only,cloud_only,deadline,cost]
    value: Optional[float]
lambdas: Dict[
    str,
        no_lambda: bool = False
        component_name: str = None
        handler: str
        package_type: Enum[zip,image] = zip
        package_metadata: Dict = {}
        local_event_topics: List[str] = []
        mqtt_event_topics: List[str] = []
        destinations: List[
            lambda_name: str
            protocols:
                cloud_cloud: Enum[sqs,lambda_invoke] = lambda_invoke
                local_cloud: Enum[s3,iot_mqtt,scheduled] = scheduled
                local_local: Enum[local_pub/sub] = local_pub/sub
            ]
        cloud:
            ephemeral_storage: int = 512
            memory: int = 128
            timeout: int = 3
            environment: Dict[str, str] = {}
            secret_environment: Optional[str]
        edge:
```

```
            timeout: int = 3
            max_idle_time: int = 60
            max_instance_count: int = 100
            max_queue_size: int = 1000
            warm_start: bool = True
            environment: Dict[str, str] = {}
        requirements: Optional[str]
        dependencies: Dict[str, Dict[str, str]] = {}
        placement: Enum[edge_only,cloud_only,scheduled] = scheduled
    ] = []
scheduler: Optional[Dict] # override scheduler configuration
finalizer: Optional[Dict] # override finalizer configuration
```

A.2 Example configuration

The following is the configuration we used for one of our experiments. More specifically, the evaluation of the scheduler with latency optimization under cost constraint with a constraint value of $0.007. Certain values have been replaced with example values to maintain the security of the Papertronics deployment

```
general:
  aws:
    architecture: arm64
    ecr_uri: 112233445515.dkr.ecr.eu-west-1.amazonaws.com
    lambda_role: arn:aws:iam::112233445515:role/lambda_role
    region: eu-west-1
    runtime: python3.8
    s3_bucket: experiment-images
    sqs_arn: arn:aws:sqs:eu-west-2:112233445515:main_lambda_queue
    thing_group_arn: arn:aws:iot:eu-west-2:112233445515:
        thinggroup/beer-o-meter_stations
  scheduling:
    type: cost
    value: 0.007
lambdas:
  alcohol_lambda:
    cloud:
      timeout: 20
    component_name: com.alcohol.lambda
    destinations:
    - lambda_name: collector_lambda
    edge:
      max_idle_time: 30
      max_instance_count: 3
      max_queue_size: 100
      warm_start: false
    handler: case_lambdas.alcohol_lambda.handle
```

```
    requirements: requirements_processing.txt

  collector_lambda:
    placement: cloud_only
    cloud:
      timeout: 60
    destinations:
    - lambda_name: finalizer_lambda
    handler: collector_lambda.handle
    requirements: requirements_collector.txt

  glucose_lambda:
    cloud:
      ephemeral_storage: 512
      memory: 128
      timeout: 20
    component_name: com.glucose.lambda
    destinations:
    - lambda_name: collector_lambda
    edge:
      max_idle_time: 30
      max_instance_count: 3
      max_queue_size: 100
      warm_start: false
    handler: case_lambdas.glucose_lambda.handle
    requirements: requirements_processing.txt

  image_processing_lambda:
    cloud:
      environment:
        SLOW: false
      ephemeral_storage: 666
      memory: 256
      timeout: 900
    component_name: com.imageprocessing.lambda
    destinations:
    - lambda_name: ph_lambda
    - lambda_name: alcohol_lambda
    - lambda_name: glucose_lambda
    edge:
      max_idle_time: 120
      max_instance_count: 3
      max_queue_size: 1000
      warm_start: false
    handler: image_processing_lambda.handle
    requirements: requirements_processing.txt

  ph_lambda:
```

```
    cloud:
      timeout: 20
    component_name: com.ph.lambda
    destinations:
    - lambda_name: collector_lambda
    edge:
      max_idle_time: 30
      max_instance_count: 3
      max_queue_size: 100
      warm_start: false
    handler: case_lambdas.ph_lambda.handle
    requirements: requirements_processing.txt

  station_lambda:
    placement: edge_only
    component_name: com.station.lambda
    destinations:
    - lambda_name: image_processing_lambda
    edge:
      max_idle_time: 240
      max_instance_count: 1
      max_queue_size: 1000
      warm_start: true
    handler: station_lambda.handle
    requirements: requirements_station.txt
```

A.3 Finalizer and scheduler configuration

This section describes the default configuration of the finalizer and the scheduler used for the experiments. The configuration grammar for these components is identical to the grammar of individual lambda functions in section A.1.

```
scheduler:
  allowed_operations:
  - '*'
  cloud:
    ephemeral_storage: 512
    memory: 128
    timeout: 900
  component_name: com.component.scheduler
  dependencies:
    aws.greengrass.TokenExchangeService:
      dependencyType: HARD
      versionRequirement: 2.0.3
  edge:
    environment:
      EXCLUDE_LAMBDAS: station_lambda, finalizer_lambda
```

```
      MODEL_BUCKET: papertronicsmodels
      RETRAIN_COUNT: 50
    max_idle_time: 60
    max_instance_count: 1
    max_queue_size: 1100
    warm_start: true
  handler: serverless_scheduler/scheduler_component.py
  local_event_topics:
  - out/station_lambda
  - out/ph_lambda
  - out/image_processing_lambda
  - out/glucose_lambda
  - out/alcohol_lambda
  mqtt_event_topics:
  - in/+
  - scheduler/+
  no_lambda: true
  requirements: requirements_scheduler.txt
```

```
finalizer:
    placement: cloud_only
    cloud:
      environment:
        EXCLUDE_LAMBDAS: station_lambda, finalizer_lambda, collector_lambda
        MODEL_BUCKET: papertronicsmodels
      ephemeral_storage: 1024
      memory: 512
      secret_environment: data/secret.json
      timeout: 600
    component_name: null
    destinations:
    - lambda_name: https://sqs.eu-west-2.amazonaws.com/
    119612254815/results_queue
      protocols:
        cloud_cloud: sqs
        local_cloud: null
        local_local: null
    handler: finalizer_lambda.handle
    package_metadata:
      DockerContext: .
      DockerTag: finalizer_lambda
      Dockerfile: aws/Dockerfile
    package_type: Image
    requirements: requirements_finalizer.txt
```

Appendix B

Machine learning methods performances

B.1 Residuals vs fit plot

B.1.1 Original pipeline

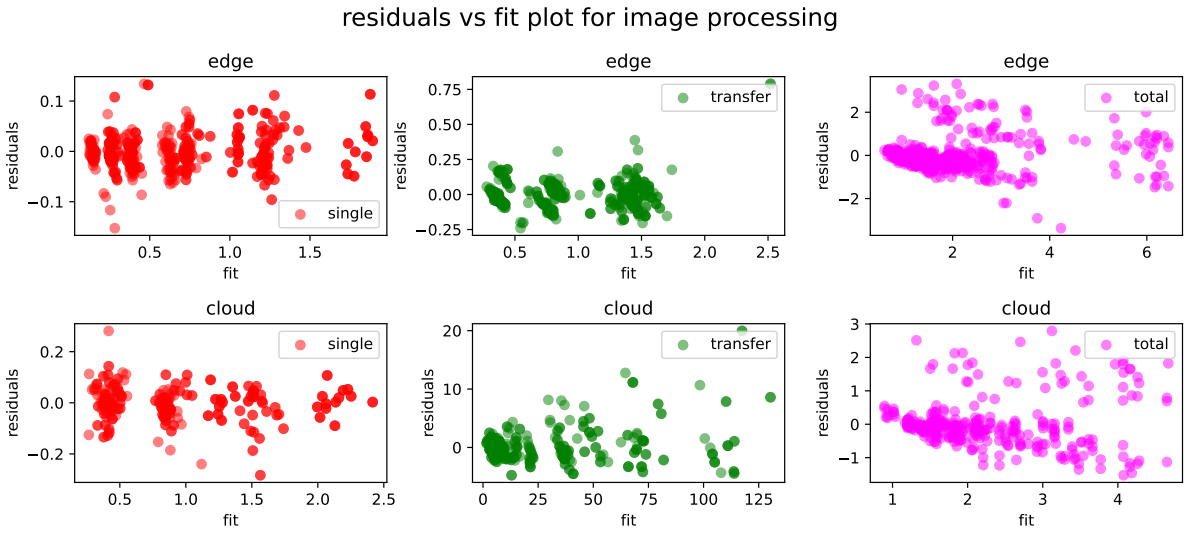


Figure B.1: Residuals vs fit plot for image processing lambda

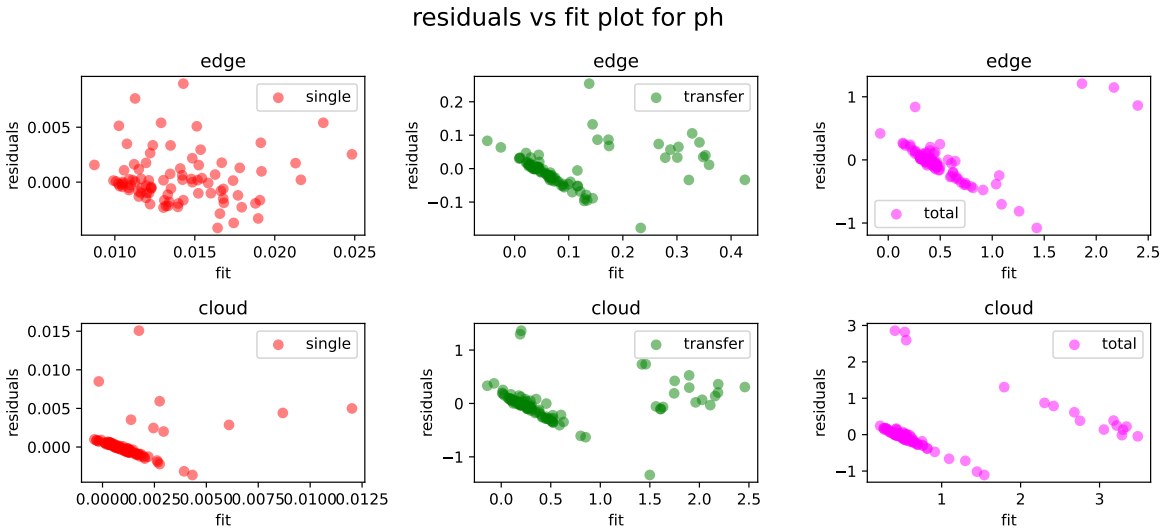


Figure B.2: Residuals vs fit plot for ph lambda

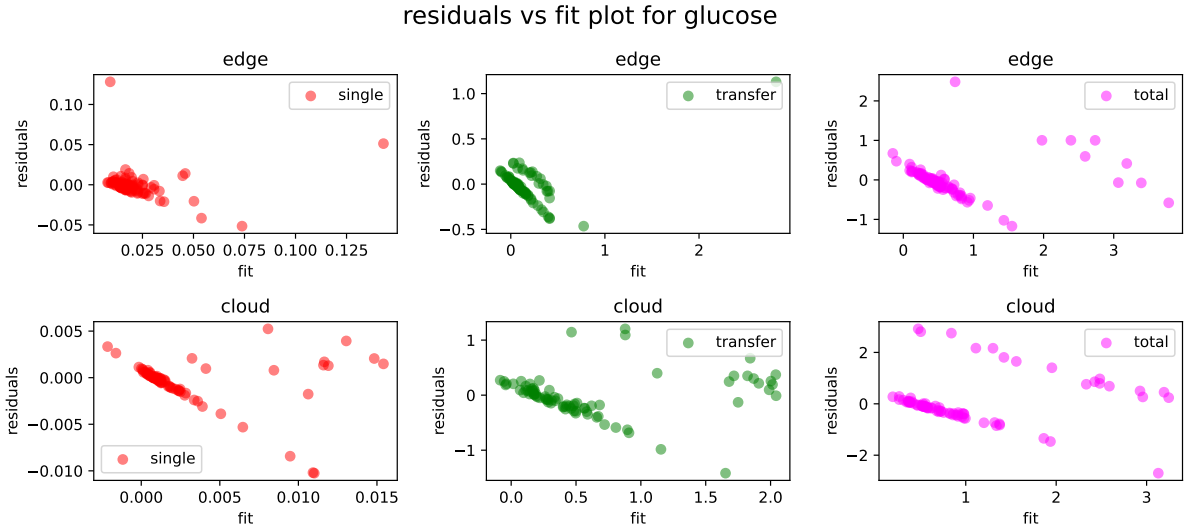


Figure B.3: Residuals vs fit plot for glucose lambda

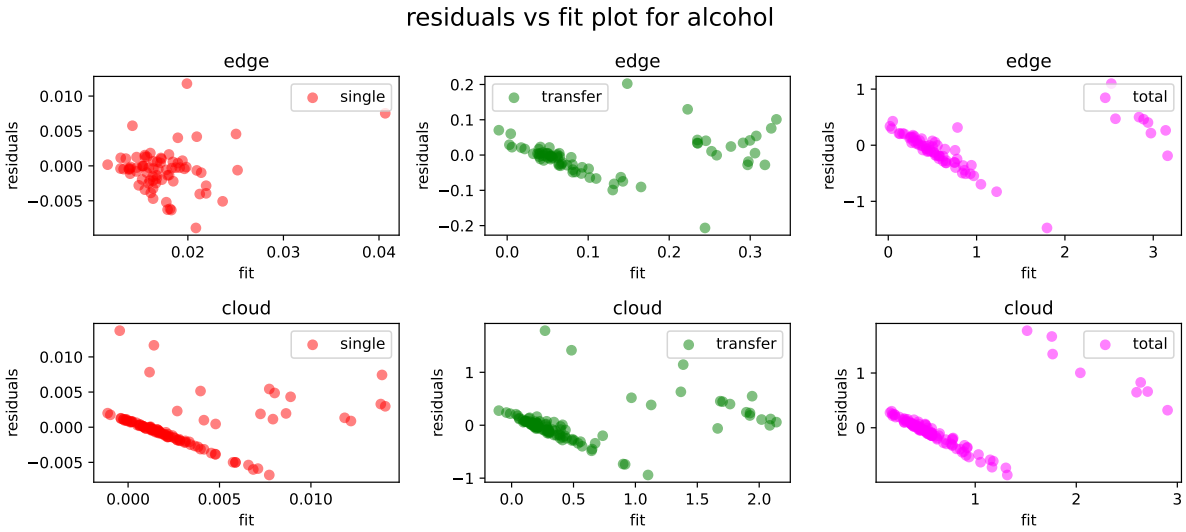


Figure B.4: Residuals vs fit plot for alcohol lambda

B.1.2 Resource intensive pipeline

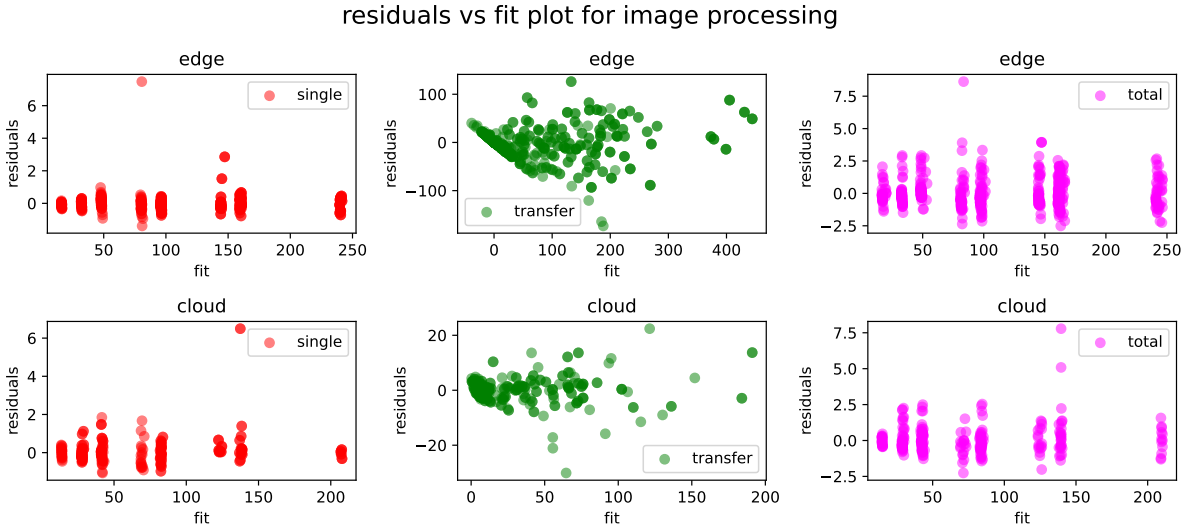


Figure B.5: Residuals vs fit plot for image processing lambda for a resource intensive pipeline

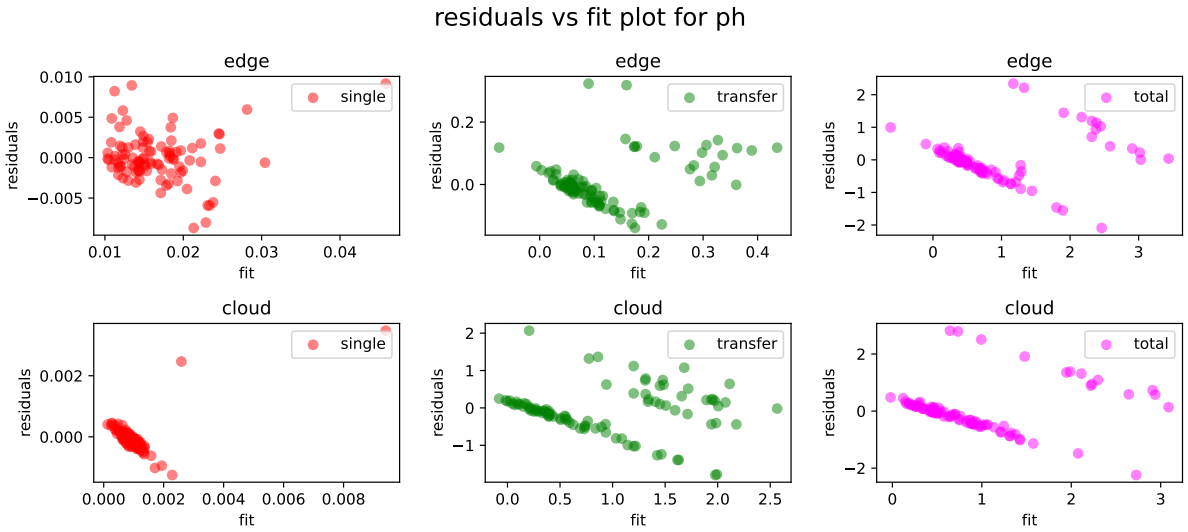


Figure B.6: Residuals vs fit plot for ph lambda for a resource intensive pipeline

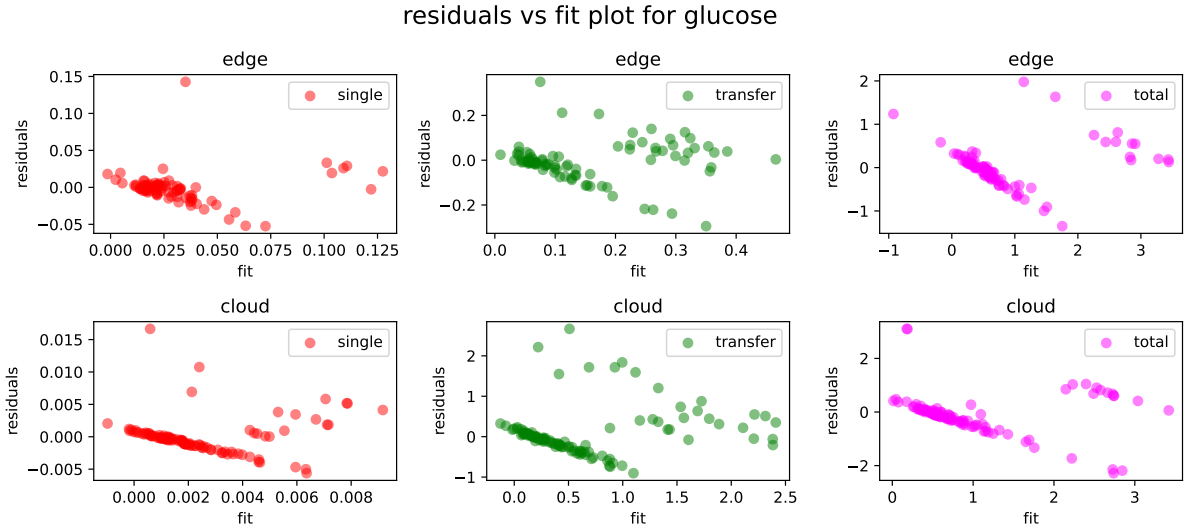


Figure B.7: Residuals vs fit plot for glucose lambda for a resource intensive pipeline

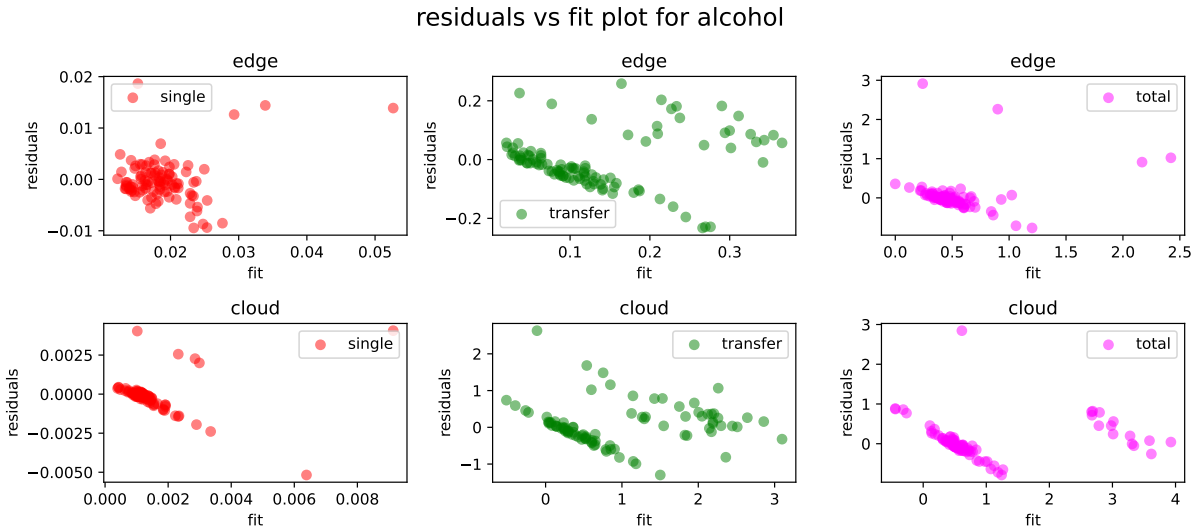


Figure B.8: Residuals vs fit plot for alcohol lambda for a resource intensive pipeline

Appendix C

Pipeline data visualized

C.1 Original pipeline

C.1.1 resources and message size vs duration

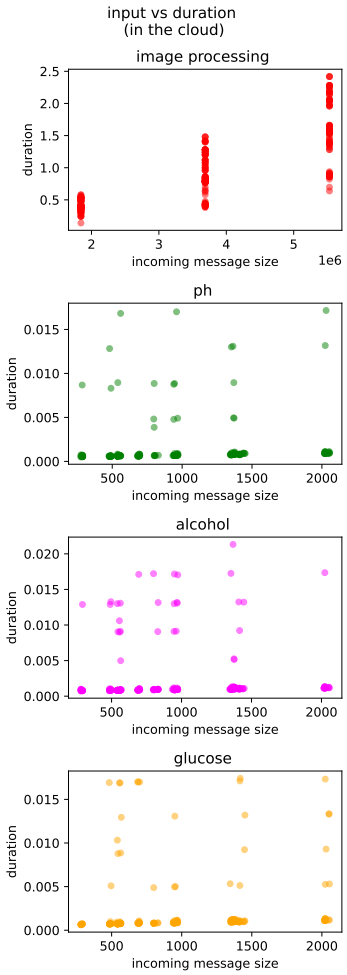


Figure C.1: incoming message size versus the duration of individual lambda functions in the cloud.

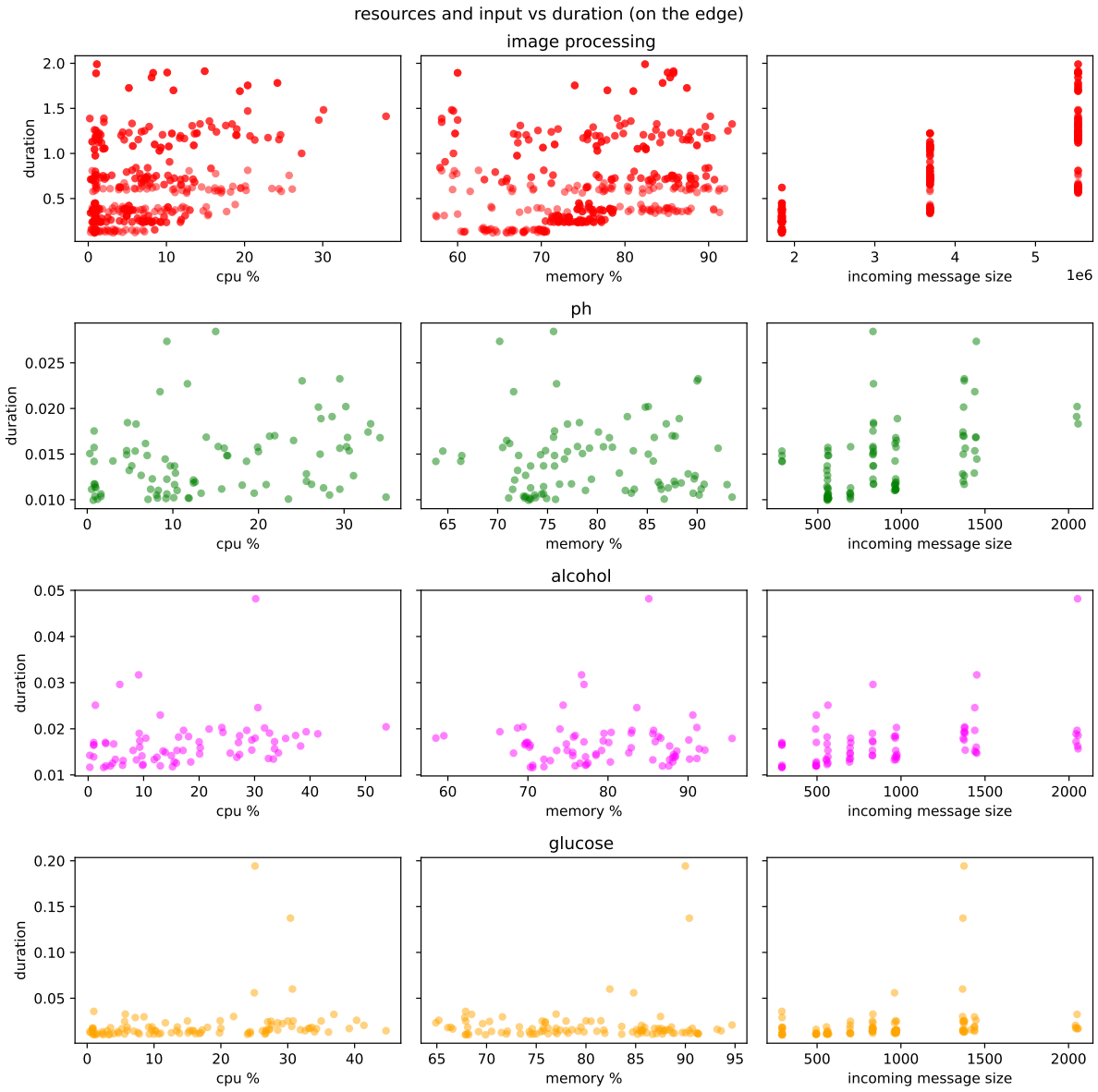


Figure C.2: Available resources and incoming message size versus the duration of individual lambda functions on the edge.

C.1.2 NETWORK BANDWIDTH, AVAILABLE RESOURCES AND MESSAGE SIZE VS TRANSFER TIMES

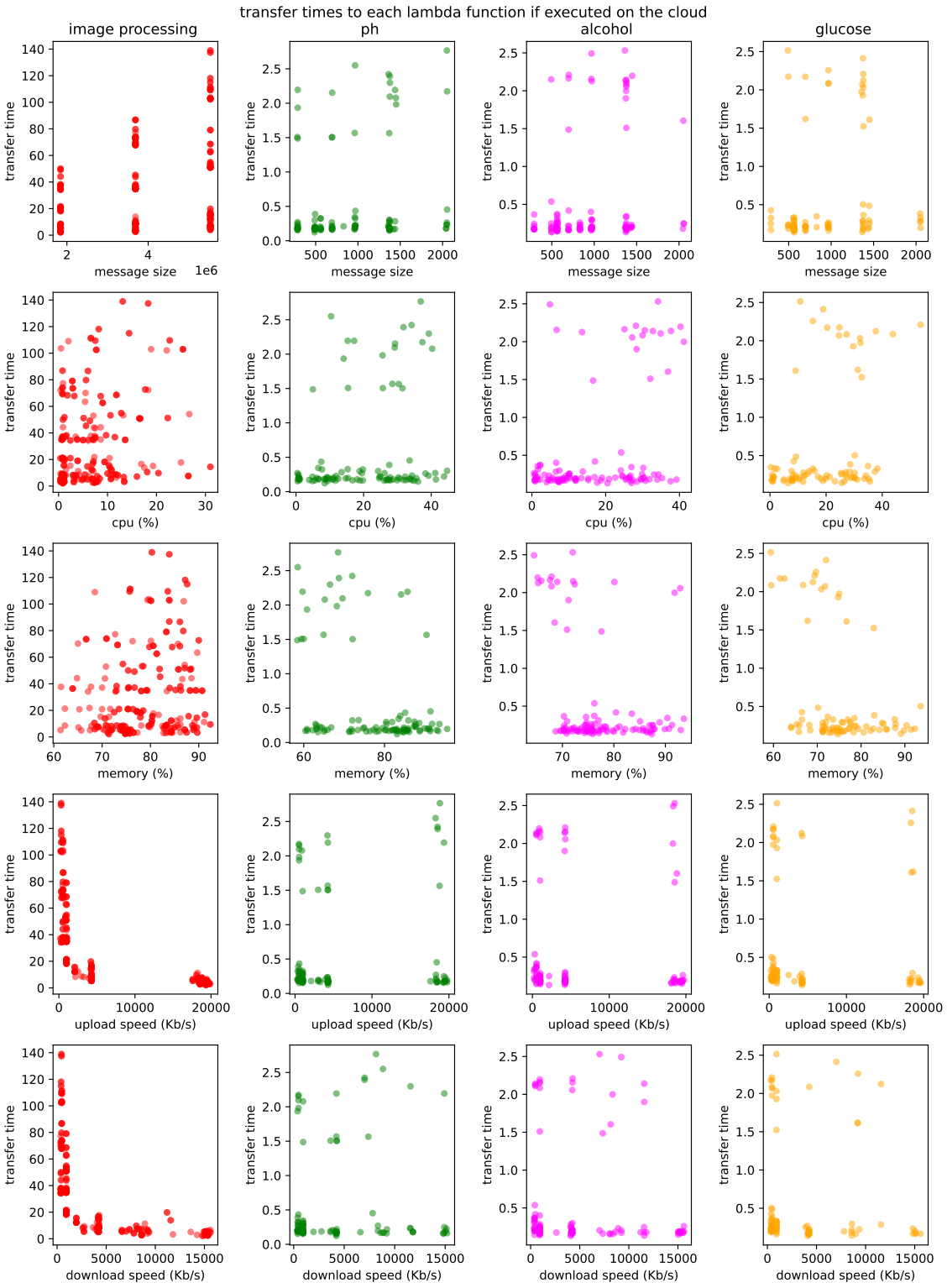


Figure C.3: Available resources and incoming message size versus the transfer time to lambda functions in the cloud.

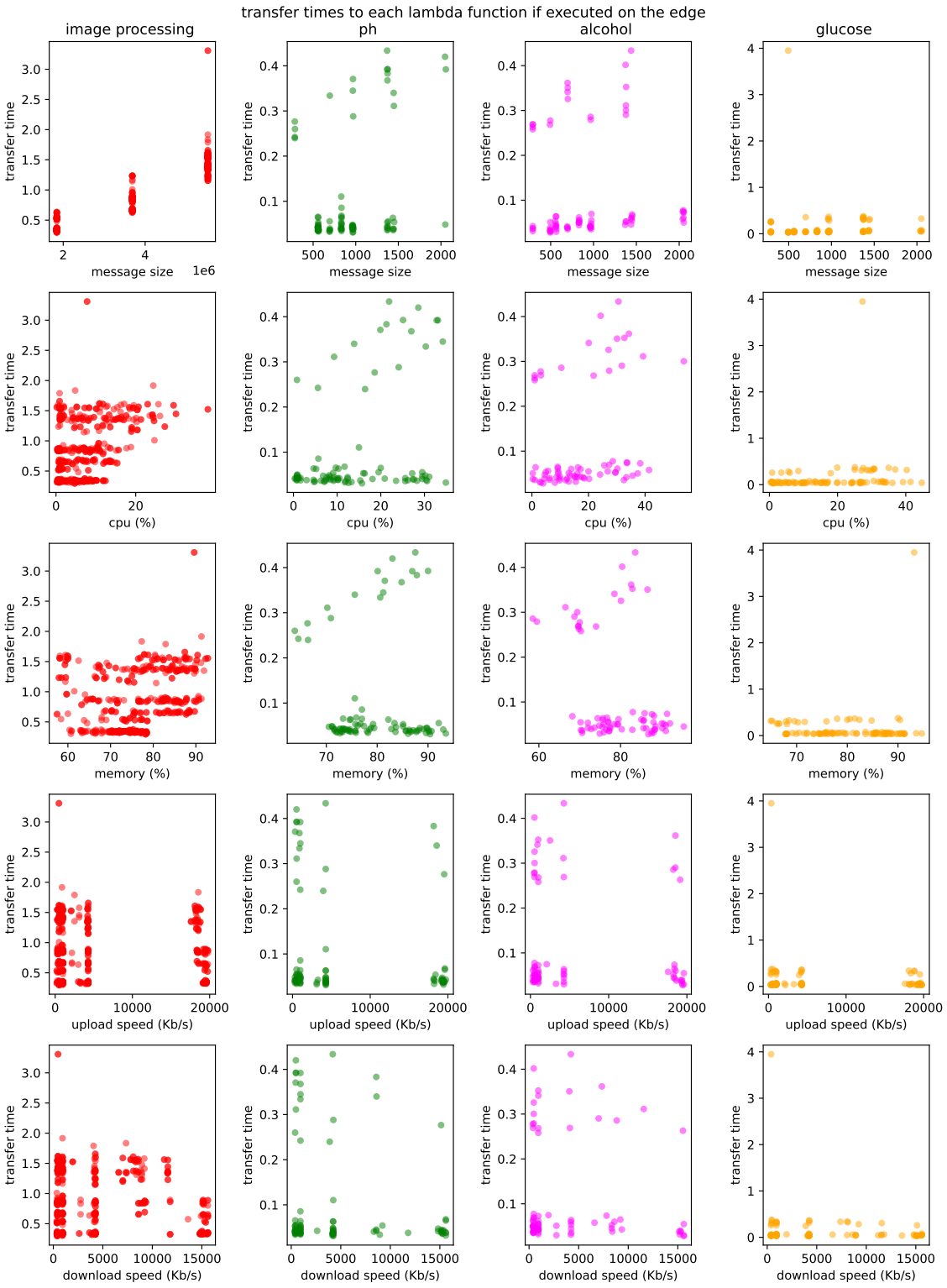


Figure C.4: Available resources and incoming message size versus the transfer time to lambda functions on the edge.

C.1.3 INPUT FEATURES VS TOTAL PIPELINE DURATION

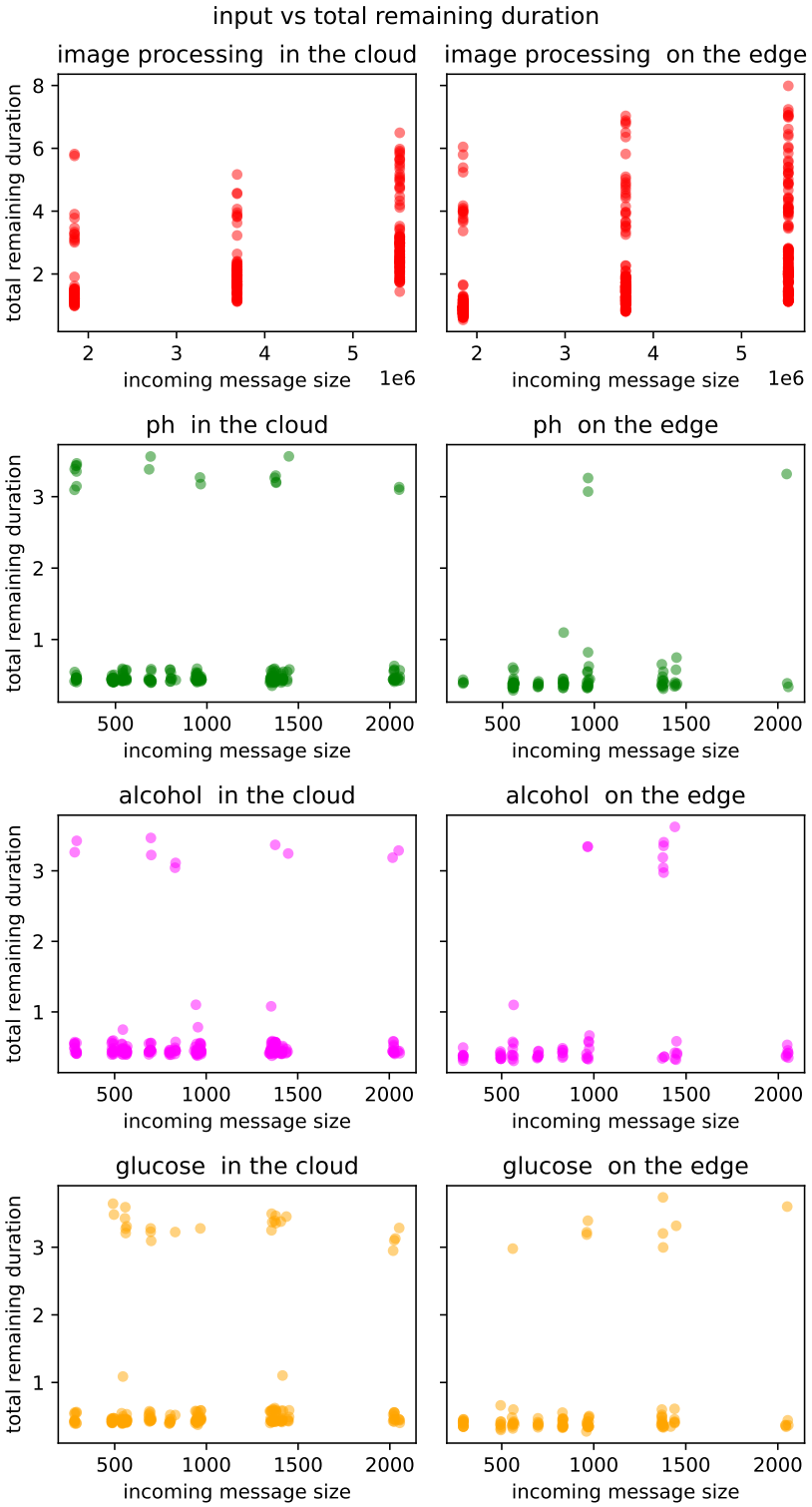


Figure C.5: incoming message size versus total remaining duration per function

C.2 Resource intensive pipeline

C.2.1 resources and message size vs duration

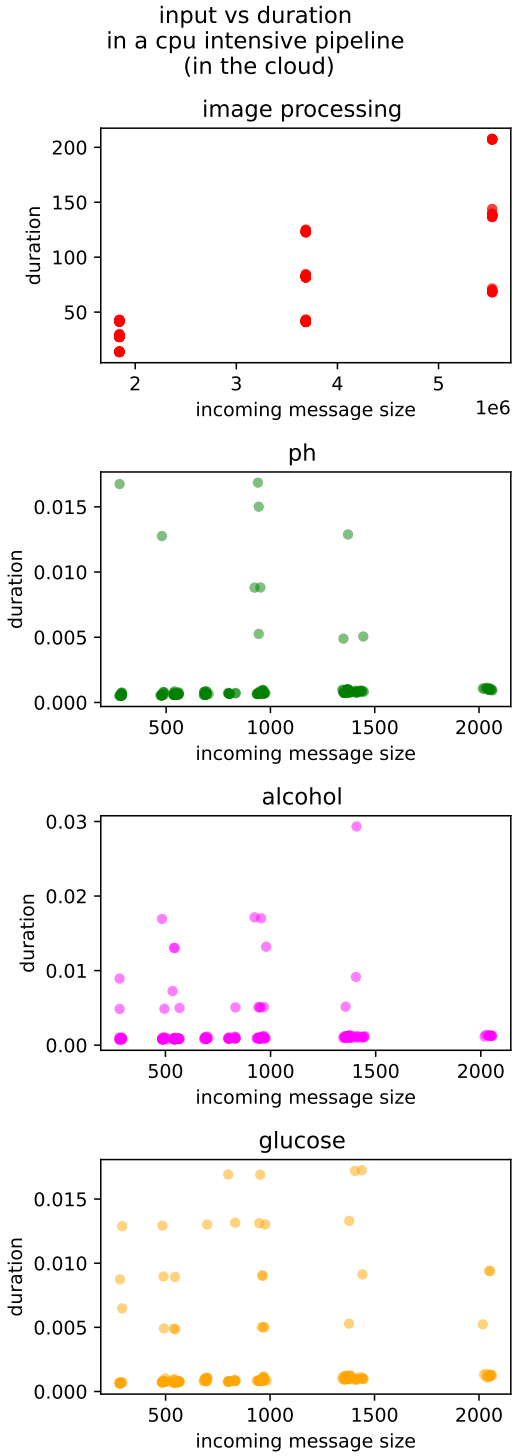


Figure C.6: incoming message size versus the duration of individual lambda functions in the cloud for a resource intensive pipeline.

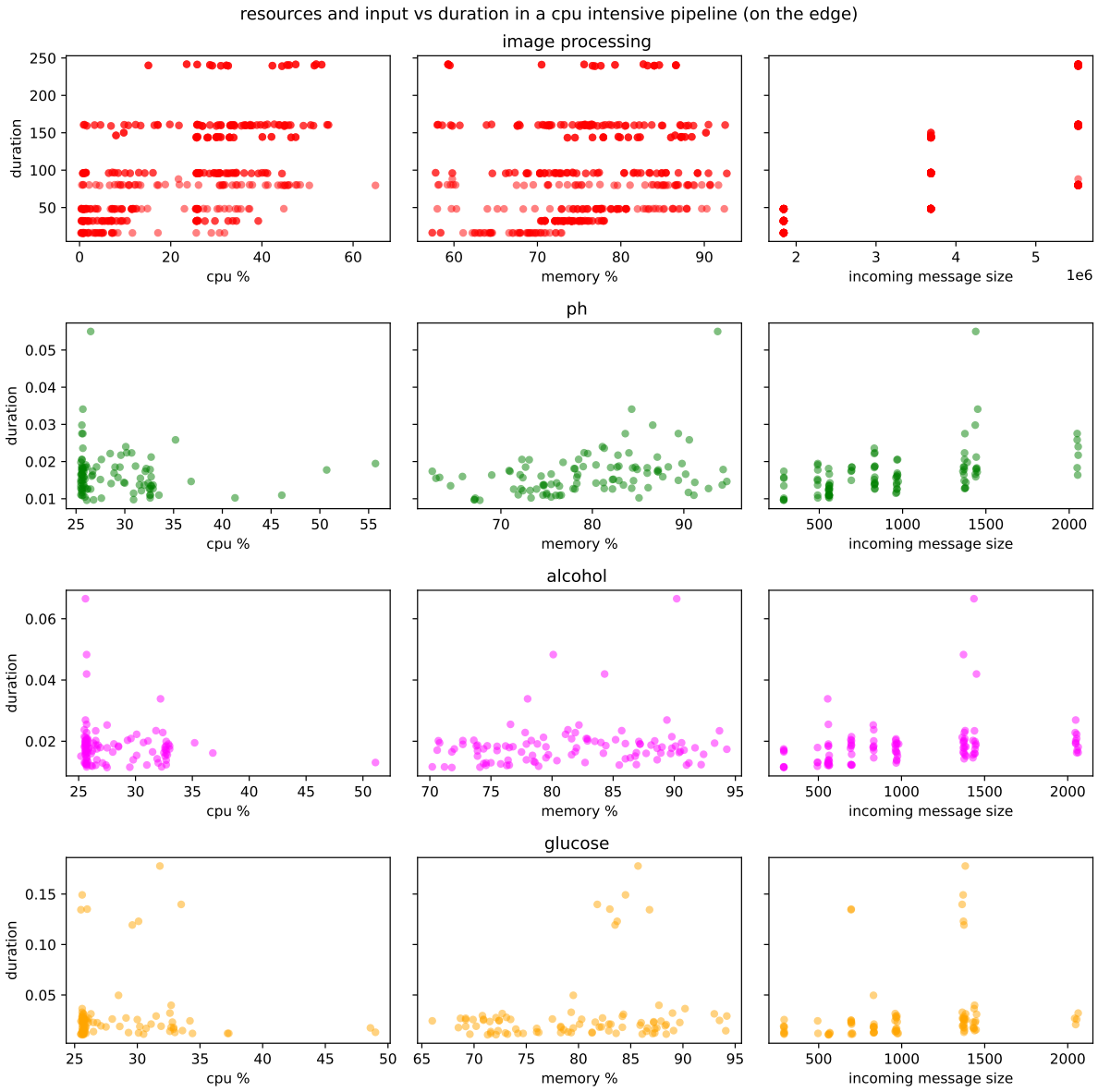


Figure C.7: Available resources and incoming message size versus the duration of individual lambda functions on the edge for a resource intensive pipeline.

C.2.2 NETWORK BANDWIDTH AND MESSAGE SIZE VS TRANSFER TIMES

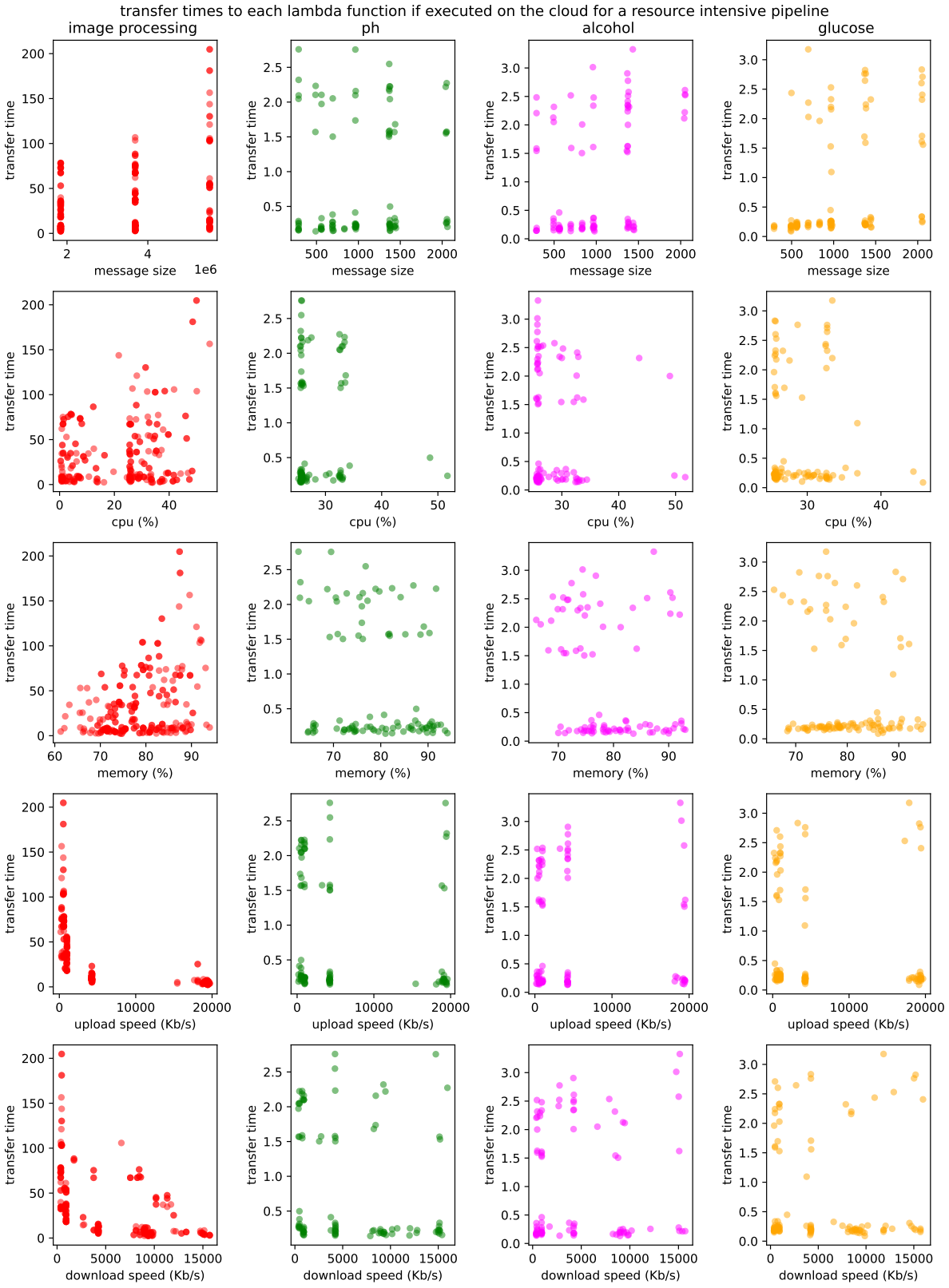


Figure C.8: Available resources and incoming message size versus the transfer time to lambda functions in the cloud for a resource intensive pipeline.

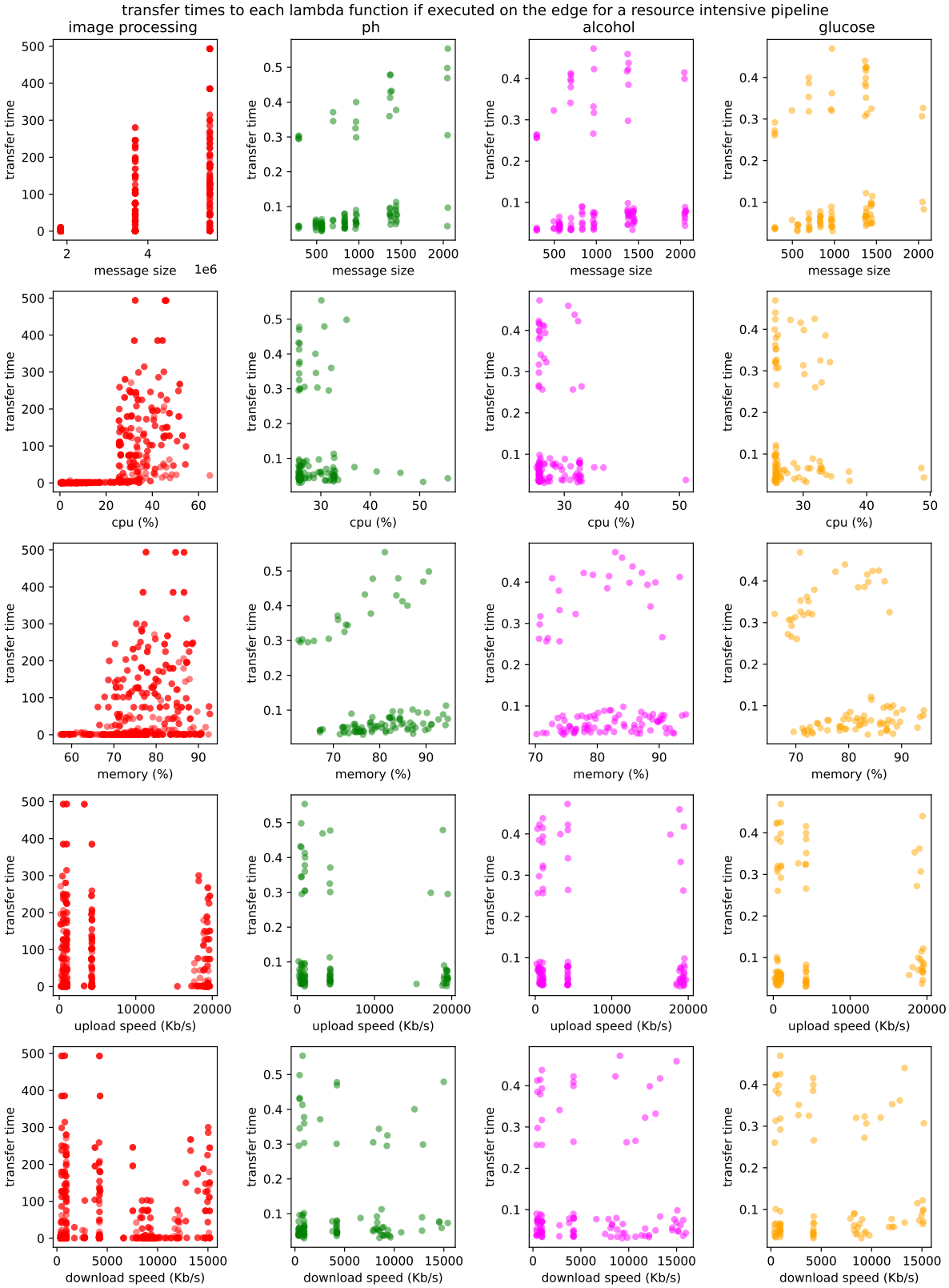


Figure C.9: Available resources and incoming message size versus the transfer time to lambda functions on the edge for a resource intensive pipeline.

C.2.3 INPUT FEATURES VS TOTAL PIPELINE DURATION

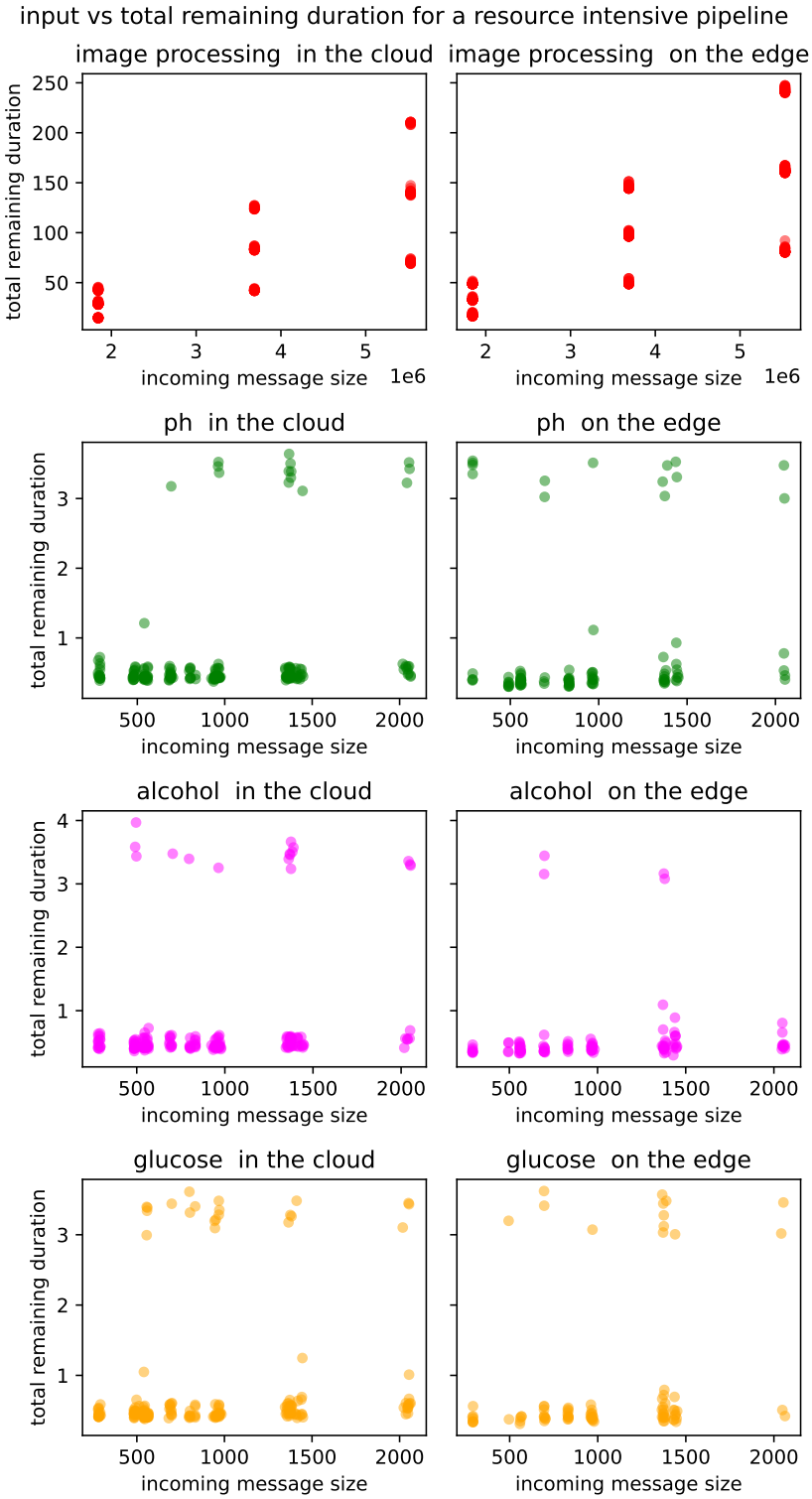


Figure C.10: incoming message size versus total remaining duration per function for a resource intensive pipeline