Dynamic Allocation of Serverless Functions in IoT Environments

Duarte Pinto

Dissertação
Dynamic Allocation of Serverless Functions in IoT Environments

Duarte Pinto

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. André Restivo
External Examiner: Prof. Pedro Ribeiro
Supervisor: Prof. Hugo Sereno Ferreira

July 23, 2018
Abstract

IoT is coming to people’s houses and is entering everybody’s life. The market has grown significantly in the last few years and is expected to reach a gigantic amount of 50 billion devices by 2020. As this number grows, there is an increasing necessity of maturing the technology and how we use it.

Parallel to IoT, serverless architectural design is also an increasing trend in the field of software engineering, with vendor solutions as AWS Lambda gaining traction and wide-spreading. Also known as "Function as a Service" or FaaS, this architectural design tries to decrease the demand and costs of having an always-running server by replacing it with short-term, on demand, power, that is managed by a 3rd-party entity like Amazon or Google.

Given this two increasing trends, combining IoT with a serverless architectural design can be positive as we are removing part of the overhead that comes with relying on a server for every action. This approach falls in the category of Fog Computing by trying to be a middle-point between Edge Computing and Cloud Computing, leveraging local computational capabilities that are closer to the end-user.

What is being aimed is to create and implement a decentralized architecture for IoT that uses both of the architectural designs (Serverless and Fog Computing) in order to reduce the costs, increase responsiveness, augment data-ownership, and improve the developers experience on designing IoT systems.

When a device requests for the execution of a serverless function, the proposed solution, which is placed between the device and the serverless function, will decided base on previous metrics of execution if the serverless function should be executed locally, in the fog layer of a local network of IoT devices, or if it should be executed remotely, in one of the many cloud servers. The solution makes this decision with the aim of improving response times, by taking advantage of the joint processing power of the local network of IoT devices when it is more beneficial and suitable to do so.
Resumo

IoT está a chegar à casa de toda a gente e a entrar na vida de todos. O mercado cresceu significativamente nos últimos anos e espera-se que alcance uma gigantesca quantidade de 50 bilhões de dispositivos até 2020. Ao mesmo tempo que este número cresce, cresce também a necessidade de amadurecer a tecnologia e como a usamos.

Paralelamente ao IoT, o design arquitectónico serverless também tem ganhado popularidade no ramo da engenharia de software, com soluções como o AWS Lambda que ganhou tração no mercado. Também conhecido como "Function as a Service" ou FaaS, este conceito arquitectónico tenta diminuir a carga e os custos de ter um servidor sempre a correr, substituindo-o por poder de processamento de curta duração e on demand que é gerido por uma entidade externa como Amazon ou Google.

Dado estas duas tendências crescentes, combinar o IoT com uma arquitetura serverless pode ser positivo dado que estamos a remover parte da sobrecarga que vem com depender de servidor para cada ação. Esta abordagem cai na categoria de Fog Computing, que tenta ser um ponto intermediário entre Edge Computação e Cloud computing, aproveitando o poder de processamento que existe no local e que está mais perto do usuário final.

O que se quer fazer é criar e implementar uma arquitectura descentralizada para IoT que usa ambos os os conceitos arquitectónicos (Serverless e Fog Computing) com o objectivo de reduzir os custos, aumentar a capacidade de resposta, aumentar o controlo sobre os dados e melhorar o processo de desenvolvimento de projetos de sistemas IoT.

Quando um dispositivo pede a execução de uma função serverless, a solução proposta, que está presente entre o dispositivo e a função, vai decidir, com base em métricas de execuções anteriores, se a função deve ser executada localmente, na camada fog da rede local de dispositivos IoT, ou se deve ser executada remotamente, num dos vários servidores. A solução toma esta decisão com o objectivo de melhorar o tempo de resposta, tirando partido do poder de processamento conjunto da rede local de dispositivos IoT, quando tal for mais benéfico e apropriado.
Acknowledgements

I would like to start by thanking my supervisor Hugo Sereno Ferreira for all the guidance and shared knowledge given during the realization of this dissertation.

I am also very grateful to João Pedro Dias, my co-supervisor, for always being available to help me and to guide me through this journey, for being my first line of support, and as well for always suggesting me how to improve my work.

I would also like to thank my friends for the immense knowledge they shared with me in these last five years, for helping me study for exams, for all the night outs doing projects all together, and also for making me like programming even more, guaranteeing that I will always love my job.

Lastly, I would like to express my immense gratitude to my family for the constant support, patience and for always trusting me and allowing me to make my own decisions.

Duarte Manuel Ribeiro Pinto
# Contents

1 Introduction
   1.1 Context ........................................... 1
   1.2 Problem definition ................................ 2
   1.3 Motivation and goals ................................ 2
   1.4 Structure ........................................... 3

2 State of the Art ........................................... 5
   2.1 Introduction ......................................... 5
   2.2 Internet of Things ................................... 5
      2.2.1 IoT at home ..................................... 6
      2.2.2 Industry 4.0 ..................................... 8
   2.3 Serverless ........................................... 8
      2.3.1 Definition ........................................ 8
      2.3.2 Architecture of a serverless platform ........... 10
      2.3.3 Existing Host and code execution cloud solutions .. 12
      2.3.4 Frameworks ....................................... 15
      2.3.5 Serverless On-Premises vs Cloud-based ............ 16
      2.3.6 Positive Aspects and Drawbacks .................... 16
   2.4 Fog Computing ....................................... 17
      2.4.1 Definition ........................................ 17
      2.4.2 Characteristics .................................... 17
      2.4.3 Why Fog Computing in IoT? ......................... 18
      2.4.4 Existing Solutions ................................ 18
   2.5 OpenFaaS ............................................ 20
      2.5.1 What is? ........................................... 20
      2.5.2 Components ....................................... 21
      2.5.3 Kubernetes vs Docker Swarm ....................... 22
      2.5.4 Function Metrics .................................. 22
   2.6 Exploration vs Exploitation and Multi-Armed Bandit .... 22
      2.6.1 Definition ......................................... 23
      2.6.2 $\varepsilon$-Greedy ................................ 23
      2.6.3 Upper Confidence Bounds ........................... 24
   2.7 Conclusions .......................................... 25

3 Problem Statement ......................................... 27
   3.1 What is the problem? ................................ 27
   3.2 Proposal ............................................. 27
   3.3 Assumptions .......................................... 28
CONTENTS

4 High Level Overview
  4.1 Solution description ................................................. 29
  4.2 Expected results and flow. Use cases ................................. 30
    4.2.1 Forward function execution to the cloud .................... 30
    4.2.2 Forward function execution to the local network .......... 31
    4.2.3 Fallback to the local network ............................... 31
    4.2.4 Manual forward. Bypass the weighting process ............ 32
  4.3 Weighting the runtime environments ................................ 32
  4.4 Code components .................................................. 33
    4.4.1 Packages .................................................. 33

5 Technical Implementation
  5.1 Implementation stack, components and interaction. ................ 37
    5.1.1 Stack .................................................. 37
    5.1.2 Interaction of the different components .................... 39
  5.2 Proxy .......................................................... 39
    5.2.1 Received Input ........................................... 39
    5.2.2 Fault-tolerance and Fallback to a local runtime environment .... 40
  5.3 Function weighting ................................................ 41
    5.3.1 Greedy algorithm ......................................... 42
    5.3.2 Multi-Armed Bandit: Upper Confidence Bounds ............... 42
  5.4 Metrics: Storage and retrieval .................................... 43
    5.4.1 MongoDB for logging and metric storage .................... 44
    5.4.2 Connection to the database ................................ 44
  5.5 Sample Functions .................................................. 45

6 Experimentation and evaluation
  6.1 Environment ...................................................... 47
    6.1.1 Connection latency ......................................... 47
  6.2 First Experiment: With internet connection ....................... 47
    6.2.1 Expected results ........................................... 47
    6.2.2 Results .................................................. 48
  6.3 Second Experiment: Without internet connection to the servers ... 49
    6.3.1 Expected results ........................................... 49
    6.3.2 Results .................................................. 49
  6.4 Third Experiment: Turning off Internet connection in a series of requests ... 50
    6.4.1 Expected results ........................................... 50
    6.4.2 Results .................................................. 51
  6.5 Fourth Experiment: Adapting to lag ................................ 51
    6.5.1 Expect results ............................................. 52
    6.5.2 Results .................................................. 52
  6.6 Points of Failure .................................................. 54
    6.6.1 Failure to recognize when the local network is overloaded with work .... 54
    6.6.2 Time limit and boundaries for local vs remote execution .... 54
    6.6.3 Slowness to adapt ........................................... 54
  6.7 Summary .......................................................... 54

viii
## CONTENTS

7 Conclusions and Future Work ............................................. 55
   7.1 Contributions .......................................................... 56
   7.2 Main Difficulties ....................................................... 57
   7.3 Future Work ............................................................. 57

References ................................................................................. 59

A Additional Content ............................................................. 63
   A.1 Comparison between different serverless platforms and the languages
       that they support .......................................................... 63
   A.2 Full sequence diagram of a function request through proxy .......... 63
   A.3 Third Experience’s Results Complete .................................. 63
   A.4 Fourth experience .......................................................... 65

B JSON Schemas of the proxy’s internal functions ......................... 69
   B.1 JSON Schema of the request to the proxy function .................. 69
   B.2 JSON Schema of the request to the weight_scale function ........ 70
List of Figures

2.1 The IoT market will be massive [Luc16] ........................................ 6
2.2 Flow diagram of Amazon Echo, Amazon's AI powered smart speaker, turning on the kitchen lights ................................................................. 7
2.3 How companies are finding opportunity through the Industrial Internet of Things [Tec15] ........................................................................ 9
2.4 Serverless in the scheme of developer control [BCC+17] ................. 9
2.5 example of a serverless architecture in a standard online shop application. this contrast with the standard architecture (client > online shop server > database) [?] 10
2.6 Increase of the popularity of the term `serverless' in Google Trends [kn:e] .... 11
2.7 Increasing shift from DevOps to serverless computing to drive the overall Function-as-a-Service market "The Function-as-a-Service (FaaS) market size is estimated to grow from USD 1.88 billion in 2016 to USD 7.72 billion by 2021, at a Compound Annual Growth Rate (CAGR) of 32.7% owing to growing number of users adopting FaaS due to their easy deployment and execution ability. However, application portability issues on cloud environments acts as a major restraint in the overall growth of the market." [kn:g][Mar] ........................................ 11
2.8 Basic example of the architecture of a Serverless platform [BCC+17] .... 12
2.9 Architecture of IBM OpenWhisk/IBM Cloud Functions [BCC+17] ........ 15
2.10 Model of how Fog Computing paradigm fits and interacts with other layers(Edge and Cloud) in a network [kn:d] ................................................. 17
2.11 How and where EdgeX Foundry fits in the network [kn:a] ................. 20
2.12 Overview of OpenFaaS [kn:f] ............................................................ 21
2.13 When the expected reward has a Gaussian distribution. \( \sigma(a_i) \) is the standard deviation and \( c\sigma(a_i) \) is the upper confidence bound. The constant \( c \) is a adjustable hyperparameter. [Wen18] [Sil] ...................................................... 25
4.1 High level overview of project's architecture ........................................ 29
4.2 Request for the execution of a demanding function to be executed. The proxy will forward the request to the cloud because due to the high processing power of the cloud server, the function will be executed more quickly. The nearest server was choosen because of latency. ......................................................... 30
4.3 Request for the execution of a simple, light function to be executed. The proxy will forward the request to be run locally, as there is no benefit in executing the function on the cloud. .......................................................... 31
4.4 The request for the function execution to be in the cloud could not be satified (e.g., no internet connection). The proxy will then forward the request for it to be executed in the local network. ............................................. 32
4.6 Component diagram of the project ....................................................... 34
# LIST OF FIGURES

4.5 High level sequence the whole process. This diagram sums the decision-making of the project when trying to answer the request for the function execution . . . . 36

5.1 Technology stack where this project fits into. . . . . . . . . . . . . . . . . . 38
5.2 Low level overview of the implemented solution. . . . . . . . . . . . . . . . . 39
5.3 Multiple instances of the same function accessing the mongoDB database. In the last instance there is one active connection and two shown with dotted lines which are about to be closed due to inactivity [Ell]. . . . . . . . . . . . . . . . . . . . . 45

6.1 Chart illustrating the results of the third experience. . . . . . . . . . . . . . . . 51
6.2 Requests total time throughout the various iterations of the fourth experiment . . 52
6.3 This figure illustrates the cumulative average duration of the request throughout the various iterations of the fourth experiment. E.g., the value in iteration 50 (2.085604 seconds) is the average duration of the first 50 requests. . . . . . . . . 53

A.1 Full sequence diagram of a function request through proxy . . . . . . . . . . . . 67
List of Tables

2.1 Which languages are supported in each runtime version for Microsoft Azure Functions [Mic] ........................................... 13

6.1 Results of 1st experiment. ..................................................... 48

A.1 Comparison between different serverless platforms and the languages that they support .................................................. 66
Abbreviations

IoT   Internet of Things
I/O   Input-output
PaaS  Platform-as-a-Service
SaaS  Software-as-a-Service
GGC   Greengrass Core
Chapter 1

Introduction

This chapter serves to introduce the context of this project, what is the problem that it is trying to solve, as well as to what is being aimed with this project. Section 1.1 describes how this project fits the overall state of the field. Section 1.2 presents what is the main problem that is being tried to solve, as well as some of the reasons of why it might exist. Then, Section 1.3 enumerates the main reasons pushing this project, as well as what it seeks to deliver. Finally, Section 1.4 gives a brief overview of how the rest of the document is structured, and what should be expected from each chapter.

1.1 Context

Continuing the current trend, mobile data usage is expected to keep increasing exponentially, part of it thanks to mobile video streaming and IoT. The estimation is that the number of data that was generated by mobile devices during the year of 2017 exceeded the $6 \times 10^9$ Gb per month. Together with the traffic generated by laptops and peer-to-peer communications, overall traffic demand might reach $11 \times 10^9$ Gb per month\cite{DGD+14} \cite{BFG17}. To compute such a big amount of data, cloud computing would appear to be the obvious solution but there are cases where the latency that comes with transmitting data back and forth might be undesired. In certain situations it is also not feasible to expect a constant and reliable internet connection to an always on server, either because it might not be economically wise or because it might not be infrastructurally possible. In order to solve the need for low latency, as well as to improve fault tolerance, by not relying on an always on centralized server, serverless architectures and fog computing aim to reduce the dependency on the cloud by making more use of the local resources of a device and improving communication between local devices, only leaving the data intensive tasks to the cloud\cite{BFG17}. With the increasing trend in serverless solutions, such as AWS Lambda, it is opportunistic to implement this concepts in IoT.
1.2 Problem definition

Although IoT has been around for a few years already, the same cannot be said about services that provide cloud solutions and cloud infrastructure for rent. Likewise, when Serverless and Fog Computing solutions first appeared their usefulness and benefits for the IoT ecosystem was obvious and developers began to mix them together in order to get the most out of these new trend.

Despite its success and the promising future for the mix of this concepts, the area is still fairly new and few solutions can take advantage of the processing power in the cloud and in the local network of IoT devices in an efficient way without compromising speed. It is already possible to have a network of IoT devices working together to execute a series of serverless functions, but not all serverless functions are suitable to run on low-end devices. To choose where each serverless functions should be executed (locally or in the cloud) is a manual task and the end result is that developers choose to have all serverless functions running in the cloud as it is easier to manage and less risky. Nonetheless, there is a lot of potential processing power dormant in each local network that could be used to improve response times, improve fault-tolerance, and to slash costs of hosting cloud processing infrastructure.

1.3 Motivation and goals

The aim of this project is to come up with an architecture and a proof of concept on how to build a serverless platform for IoT that makes use of Fog Computing.

There are studies that show that in data-intensive scenarios, a serverless solution can outperform a cloud-based offloading solution in terms of throughput and latency [BFG17]. The flexibility of serverless really fits into the IoT ecosystem and, together with Fog Computing, it could be the start of a new way of building commercial and industrial solutions. It is important to further the research in the implementation of serverless platforms IoT with the intent of maturing the the area and to analyze if it is worth it to take advantage of the joint processing power that exists in a network of IoT devices.

There is still a lack of solutions that advantage of this power efficiently and there is a lot to gain such as improving response times, improving fault-tolerance, as there is less dependency in the cloud servers, diminishing the cost of hosting servers and cloud processing power, as some of the serverless functions will be executed locally instead of in the cloud, and also in terms of security and data ownership, due to the fact that part of the data will be processed locally instead of being sent to a remote location. Progress in this field can lead to a more efficient and less centralized future for IoT.

The aim is to create a software layer that will receive a request for the execution of a serverless function. This layer is capable of analyzing previous metrics for each serverless function and deciding if it should be executed either locally, taking advantage of the dormant processing power of the local netwot of IoT devices, or remotely, on one of the many high-performing cloud servers.
1.4 Structure

Apart from the Introduction, this dissertation contains more 5 chapters. In chapter 2, it is described the state of the art. In chapter 3 it is presented the problem statement, describing the main point of this dissertation and what is the problem that is being aborded. Chapter 4 gives an high level overview of how the problem stated before was solved and the expected outcome and in chapter 5 it will be given a detailed description of the technical implementation and other technical aspects regarding the functional prototype. To finalize, chapter 6 will describe the testing mechanism to assure validation of the expected use cases and some analysis of the results and chapter 7, Conclusions and Future Work, will describe what was concluded in this project and possible future work.
Introduction
Chapter 2

State of the Art

In this chapter, it is described the state of art to show the relevance of the introduced concepts (Internet of Things, Serverless, and Fog Computing), existing projects that enable the development in these areas, and how other companies are building solutions in this field.

2.1 Introduction

In this chapter it will be presented an analysis to the current state of IoT and a brief look into important concepts necessary for this dissertation.

It will begin by contextualizing IoT by explaining what it is and where it is in the current world and how it can grow. And overview of what is a serverless architecture and Fog Computing as well as the advantages of applying these concepts to IoT.

2.2 Internet of Things

The *Internet of Things* is a term given to the network of ever increasing number of mundane objects with embedded systems, that allows them to interact with each other or with someone remotely. The term was first introduced in 1999 by Kevin Ashton [AB05] and since then the number of connected devices has grown exponentially and is expected that by 2020 the number reaches 50 billion [Cis], a spending of around $500 bilion[Tec15], and an install base of around 30.7 billion devices [Luc16].

The initial aim was that by making normal *things* smart and connecting them all to the Internet, a whole new set of possibilities would be created, like allowing the user to remotely turn on and off the *thing*, view its status and give other commands. However, by doing this, we are also building a world where billions of devices all gather data and that data could be used to achieve something that wouldn’t be possible if all these devices weren’t connected to each other and to the Internet. Thus, a new trend is rising, and the aim is now to not only make the *things* smart enough to receive commands, but also to communicate with the environment and other *things*, and to make decisions based on the input from various sources. This creates a more smart and self-regulated
environment that depends less on the input of a physical entity and more on the input of other things. This removes unnecessary steps and frees the user of dealing with mundane tasks.

For an object to be considered a thing it must have the following properties[Car17][DMC12]:

- Have a physical embodiment.
- Be equipped with a minimal set of communicative features.
- Be associated with both one name and one address.
- To have the possibility of sensing physical phenomena or to trigger an action by interacting with the environment.

2.2.1 IoT at home

IoT has begun spreading to people’s house changing the way people live. Smart thermostats that turn on and off when they receive voice commands, lights that change according to what the user is watching on the television, turning on the heating 10 minutes before the user arrives home, voice commands to have music/videos playing in different rooms. Home automation has invaded people’s houses and is a big chunk of the IoT market.

Tech giants, such as Google and Amazon, have all entered the IoT and home automation market looking for the next big thing and trying to get a slice of all the data that the things generate and collect. Their main focus are smart speakers, speakers with built-in microphones that act together with super AI assistants design by the companies in order to act as a voice driven bridge
between the other things and the user, as well as a hub for all the things that belong in the home ecosystem.

### 2.2.1.1 IoT-enabling platforms

**Raspberry PI**

The Raspberry Pi is an extremely low-cost, single-board computer that has become a reference for IoT. Since it’s launch, the Raspberry Pi has become the best selling British computer\textsuperscript{[Gu]}\textsuperscript{a}. The reason for its success relies on it’s low-price/high-performance and flexibility, allowing the buyer to use it for lot of purposes, like setting up a living room computer, building an arcade box, creating a cloud server or even having its own weather station. The variety of I/O connectors, specially the 8 I/O pins, make such modifications easy, the low power consumption makes it affordable not only in the short term, but also in the long term, the easiness to program to and control via cloud allows for easy set-up and management\textsuperscript{[Ma]}\textsuperscript{g}. All this has allowed the Raspberry PI to become a stepping stone in the history of IoT, opening many doors for either domestic and commercial users.

**Onion’s Omega**

Omega, most specifically Omega2, is a $5 Linux developer board with built-in WiFi, made for IoT.
Its aim is to combine the form factor and power efficiency of Arduino together with the flexibility of a Raspberry PI. It is a very capable device for the price, offering, at the size of 28.2mm x 42mm, specifications like a 580MHz CPU, 128MB memory, 32MB of storage plus a MicroSD Slot, USB 2.0, 100Mbps Ethernet and b/g/n Wi-Fi, 3.3V of power and a power consumption of 0.6W.

Omega 2 started out and is still, at the moment, a Kickstarter campaign.

2.2.2 Industry 4.0

"It (IoT) spans industries representing 62 percent of gross domestic product (GDP) among G20 nations (...) including manufacturing, mining, agriculture, oil and gas, and utilities" - Accenture [Tec15] [Eco]

IoT’s potential payoff in the industry is considerably big [Tec15] and might result in a new era for industry. One of the key advantages of IoT in the industry is the operational efficiency. Productivity could be increased by far as 30% with the introduction of flexible production techniques and automation. One of the areas where this happens is the predictive maintenance of assets, where there are savings in scheduled repairs, in maintenance costs, and in the elimination of breakdowns. There are companies where this can be verified, such as Thames Water, the largest water provider and wastewater services of the UK. Thames Water is using analytics and real-time data, that come from sensors, to anticipate failures and decrease response time.

Operational efficiency is not the only benefit of IoT in the industry. Some companies, like General Electric and Michelin, are coming up with ways of using IoT to offer more value to the client, see Figure 2.3.

2.3 Serverless

2.3.1 Definition

Defining the term serverless can be difficult, as the term is both misleading and its definition overlaps other concepts, such as Platform-as-a-Service (PaaS) and Software-as-a-Service(SaaS). Serverless stands in between this two concepts, where the developer loses some control over the cloud infrastructure but maintains control over the application code, as explained in Figure 2.4 [BCC+17].

"The term ‘Serverless’ is confusing since with such applications there are both server hardware and server processes running somewhere, but the difference to normal approaches is that the organization building and supporting a ‘Serverless’ application is not looking after the hardware or the processes - they are outsourcing this to a vendor." - [Rob16]

There are two main trends in serverless:

• **Backend-as-a-Service (BaaS)** - Refers to applications in which significant parts of the server-side logic are left to 3rd party entities [Rob16]. These 3rd party entities often deal with things like authentication or credit card payments.
State of the Art

Figure 2.3: How companies are finding opportunity through the Industrial Internet of Things [Tec15]

Figure 2.4: Serverless in the scheme of developer control [BCC+17]
State of the Art

Figure 2.5: example of a serverless architecture in a standard online shop application. this contrast with the standard architecture (client > online shop server > database) [?]

- **Function-as-a-Service (FaaS)** - In this type of applications, the server-side logic is still written and controlled by the developers, but they run in stateless containers that are triggered by events, ephemeral and are fully managed by the 3rd party entity, see Figure 2.5.[Rob16]

The second trend, FaaS, is the most recent one among the two and is the over which we will focus on in this project.

As it can be seen in Figure 2.6, since its boom, serverless popularity has increased a lot and it’s expected to continue increasing, accompanying the growth of IoT (Figure 2.1). The future estimated growth of Serverless can be seen in Figure 2.7. This growth is due to the existence of already present mature cloud solutions, availability of employees with strong technical skills in the area, increasing technologically advanced services and solutions, and the widespread of use among the growing industries of the market [Mar].

2.3.2 Architecture of a serverless platform

As mentioned before, the term serverless doesn’t mean that there are no servers, just that the managing and maintenance of those servers are not part of the developers functions. As the demand on the server changes, the serverless platform will increase or decrease the capacity number of servers accordingly, without the developer's action being necessary. The cost of hosting a serverless service is usually calculated on a cost per request plan[BCC+17].
**State of the Art**

**Figure 2.6:** Increase of the popularity of the term ‘serverless’ in Google Trends [kne]

**Figure 2.7:** Increasing shift from DevOps to serverless computing to drive the overall Function-as-a-Service market. "The Function-as-a-Service (FaaS) market size is estimated to grow from USD 1.88 billion in 2016 to USD 7.72 billion by 2021, at a Compound Annual Growth Rate (CAGR) of 32.7% over five years. However, application portability issues on cloud environments acts as a major restraint in the overall growth of the market." [kn:g][Mar]
Like what is seen in Figure 2.8, the main functionality relies on the ability of processing events on demand, from a set of static functions. The serverless platform must allow management of more than one set of functions, the reception of HTTP requests or other kind of similar request, associate the request with one of the functions, create an instance of the function or find an existing one, send the event to the function instance, wait for a response from the function instance, reply to the HTTP request with the response from the function instance, and then shutdown the function instance if it is no longer needed. All this has to be done while gathering logs of the whole process\cite{BCC:17}.

The main challenge for the serverless platform is to provide all of this while keeping in check with requirements such as cost, fault tolerance and scalability. Challenges that a serverless platform has to overcome in order to be competitive\cite{BCC:17}:

- Quickness to start a function and process the event.
- Queuing of events and proper adaptation. Based the arrival of events and on the current state of the queue, the scheduling of function’s execution has to be adjusted, and manage to stop or deallocate resources from standby functions.
- Carefully consider how to scale and how to manage failures in a cloud environment.

### 2.3.3 Existing Host and code execution cloud solutions

**Amazon Lambda**  Amazon’s AWS Lambda was the first platform to be defined as serverless and defined lots of factors of the market, such as price, programming model, deployment, resource limits, security, and monitoring \cite{BCC:17}. As of 2017, AWS Lambda offers support for:

- Node.js (JavaScript);
- Python;
State of the Art

Table 2.1: Which languages are supported in each runtime version for Microsoft Azure Functions [Mic]

<table>
<thead>
<tr>
<th>Language</th>
<th>1.x</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>GA</td>
<td>Preview</td>
</tr>
<tr>
<td>JavaScript</td>
<td>GA</td>
<td>Preview</td>
</tr>
<tr>
<td>F#</td>
<td>GA</td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>Experimental</td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td>Experimental</td>
<td></td>
</tr>
<tr>
<td>TypeScript</td>
<td>Experimental</td>
<td></td>
</tr>
<tr>
<td>Batch (.cmd, .bat)</td>
<td>Experimental</td>
<td></td>
</tr>
<tr>
<td>Bash</td>
<td>Experimental</td>
<td></td>
</tr>
<tr>
<td>PowerShell</td>
<td>Experimental</td>
<td></td>
</tr>
</tbody>
</table>

- Java (compatible with Java 8);
- C# (.NET Core).

Due to being the first one to release, it is the more mature solution and the de facto platform to host a serverless service. It is also one of the few that has already made it out of beta phase.

**Google Cloud Functions**  At the moment, Google Cloud Functions is still very limited as it only accepts functions written in Javascript, in a standard Node.js runtime environment, and can only respond to events from Google Cloud services or to HTTP calls [BCC+17]. Despite only offering basic FaaS functionalities, it is expected that the range of features and available languages increases in the future. At the moment, it is still in Beta.

**Microsoft Azure Functions**  Microsoft Azure Functions provides an open-source serverless service whose runtime code is available on Github, under the MIT License. The functions provided by the developer run on HTTP webhooks in integration with Azure services [BCC+17].

Microsoft offers 3 different levels of support for languages in Azure Functions [Mic]:

- **Generally available (GA)** - Fully supported and approved for production use.
- **Preview** - Not yet supported but is expected to reach GA status in the future.
- **Experimental** - Not supported and might be abandoned in the future; no guarantee of eventual preview or GA status.

At the moment, there are two versions available for the runtime of Azure functions, runtime 1.x and runtime 2.x. The 1.x runtime is GA. It’s the only runtime that is approved for production applications. The 2.x runtime is currently in preview, so the languages it supports are in preview. The list that contains which languages are supported in each runtime can be found in Table 2.1 on page 13.
State of the Art

OpenLambda OpenLambda is a new open-source serverless platform, available on Github under Apache License, developed at the University of Wisconsin, Madison. Despite the promising solution, the developers still list some issues to be solved, like [BCC+17][HSH+]:

- A slow startup time for heterogenous language runtimes and across a load balanced pool of servers;
- Deployment of large amounts of code;
- Support statefulness on top of stateless functions;
- Using of databases;
- Legacy decomposition;
- Cost debugging.

IBM OpenWhisk/IBM Cloud Functions One of the key aspects of IBM Cloud Functions is the ability to create a chain of serverless functions to serve as composite functions. It is based on Apache OpenWhisk, which is available on Github under the Apache License, and supports:

- Node.js;
- Java;
- Swift;
- Python;
- Arbitrary binaries embedded in a Docker container.

Apache OpenWhisk is still undergoing incubation at The Apache Software Foundation (ASF), what means that “While incubation status is not necessarily a reflection of the completeness or stability of the code, it does indicate that the project has yet to be fully endorsed by the ASF”[Apa]. Despite this, this IBM Cloud Functions already has important components that handle security, logging, and monitoring. This can be observed when comparing the architecture of OpenWhisk in Figure 2.9 with a more basic architecture in Figure 2.8.

Conclusions Overall, the most stable and complete solution is AWS Lambda. Being the first in market gave AWS more time to mature and to focus on supporting the most famous languages. Microsoft Azure Functions comes in second, offering stable support for Javascript and C#, but support for Python is still experimental, and it doesn’t support java. OpenWhisk (and IBM implementation of it) is the best open-source solution and might be the best solution for implementing a on-premises serverless platform. Despite being still in incubation, it is in the right path to become a dependable solution. Google Cloud Functions might be the most limited of all, only offering
support for JavaScript, but given the tight connection of Google Cloud services with the Android ecosystem and the fact that a Node.js is such a powerhouse for backend, it might actually be the best solution for some use cases.

The Table A.1 gives a brief overview of the language support for the mentioned hosting solutions.

2.3.4 Frameworks

Serverless frameworks come to aid the management of serverless functions in the cloud hosting services. While it is easy to get started with a serverless platform and to create the first functions, in the long term, the process of deploying and rollorting services becomes a mainly manual job whose time increases with the number of functions that the developer has in its platform. These frameworks solve that issue by making the process much more automatic, even sometimes providing their own unique language for building functions, with a familiar syntax, and allowing automatic deployment to more than one hosting platform.

**The Serverless Framework** One of the first and most famous frameworks, serverless started out as a framework for AWS Lambda only but has since then expanded to the Google Cloud Platform, Microsoft Azure, and IBM OpenWhisk. The Serverless.yml file contains all the necessary information about the hosting platform, runtime environment, and the definition of the functions, events that trigger them, and any other resources that might be used by the functions. The Listing 2.1 includes a Hello World Service.yml file.

```yaml
1 service: hello-world
2
```
State of the Art

```yaml
provider:
  name: aws
  runtime: nodejs6.10

functions:
  helloWorld:
    handler: handler.helloWorld
    events:
      - http:
        path: hello-world
        method: get
        cors: true
```

Listing 2.1: Example of a basic Service.yml file

"The Serverless Framework is a core component of The Coca-Cola Company’s initiative to reduce IT operational costs and deploy services faster." - Patrick Brandt Solutions Architect at The Coca-Cola Company [kn:h]

The Serverless Framework is one of the most used frameworks among developers looking to get into serverless [kn:i], in part because it facilitates both Multi-cloud and Multi-language applications.

2.3.5 Serverless On-Premises vs Cloud-based

Serverless is not only present as a service for rent on a cloud system hosted by a third-party entity. Although the pay-per-use code hosting model is the most direct approach to serverless, the same concept has been extrapolated to local solutions. In on-premises serverless, the same idea of encapsulating the code execution in an ephemeral and stateless container has been duplicated and applied to devices in the local network, like IoT devices. There are cases where there is no benefit in running the code in the cloud apart from the inability for the device to do it itself and this allows for code to be executed amongst different local machines without the need of paying for cloud processing power, without having to rely on a constant internet connection, and without having the delay associated with a request for the outside of the network.

2.3.6 Positive Aspects and Drawbacks

An application built to rely on a serverless architecture has many advantages. The code for a specific functionality is small and stateless, and the functions share the runtime environment, usually a pool of containers. Because of this, deployment of a pool of containers, called workers, to a machine and the execution of a function into any one of them is a process both fast and cost efficient. The fact that scaling becomes a fully automatic process is also a big advantage and saves the company the cost and time of dealing with a fully implemented server. Lastly, the payment model of a serverless architecture where the customer only pays for the time when the resources
are being used is much better, allowing companies to save a lot on their servers. The payment is measured by the 100ms instead of the more traditional hourly rate for a virtual machine [BFG17].

Despite this, because serverless platforms are still in an early stage, choosing one, especially closed-source ones, means having to stick to that ecosystem, as they force developers to use only services that their platform provides. This can might change in a foreseeable future, as open-source platforms grow and leave beta stage [BCC+17].

2.4 Fog Computing

2.4.1 Definition

Fog Computing is a virtual resource paradigm, located in between the Cloud layer (traditional cloud or data centers) and the Edge layer (smart end devices) in order to provide computing power, storage, and networking services. Although conceptually located in between the two layers, in practice, this platform is located at the edge of the network [BMZA]. "This paradigm supports vertically-isolated, latency-sensitive applications by providing ubiquitous, scalable, layered, federated, and distributed computing, storage, and network connectivity" [IFB+17]. In Figure 2.10 shows an example of how Fog Computing might fit in a cloud based ecosystem whose aim is to serve smart end devices.

2.4.2 Characteristics

The following Characteristics are commonly found in a Fog Computing ecosystem [IFB+17]:

- **Contextual location awareness, and low latency** - Fog Computing has advantage over Cloud computing because it can answer to requests from the Edge at a higher speed, because it is located nearer to the edges points.
State of the Art

- **Geographical distribution** - To meet its desired value, Fog Computing relies on a much bigger and widely distributed network of deployments. In contrast to Cloud Computing, which is built with a unique and distant central unit in mind, Fog Computing is based upon its smaller distance to the end points.

- **Large-scale sensor networks** - To monitor the environment.

- **Very large number of nodes** - Consequence of the wide geographical distribution, a bigger number of units is necessary to serve the network.

- **Support for mobility** - Such a network has to be able to serve mobile devices that keep disconnecting and reconnecting with the multiple points of the Fog layer.

- **Real-time interactions** - Fog applications rely on real-time interactions and not on batch processing.

- **Predominance of wireless access** - Despite also being used in a wired ecosystem, the proximity Fog Computing has with IoT demands that it supports wireless access.

- **Heterogeneity** - Has to support a large variety of end points.

- **Interoperability and federation** - To provide certain features and to take advantage of the multiple aspects each unit provides, the units must be able to interoperate.

- **Support for real-time analytics and interplay with the Cloud** - The aim is not to overrule the cloud. Fog only provides low latency and context awareness through localization, cloud still serves as a central globally available point. Although real-time analytics are usually part of fog’s tasks, historical and batch analysis is carried out in the cloud.

### 2.4.3 Why Fog Computing in IoT?

Fog Computing is appropriate for IoT if, in any of the many uses of IoT, the generated data usually meets requirements (which are usually present in any IoT based solution) such as being **delay sensitive**, produced in a **large amount**, **trust-sensitive**, of **intermittent** and **costly** (not only in monetarily, but also timely) transmission, and the whole project needs to be in an **energy efficient** ecosystem [KS].

Looking at the characteristics of Fog Computing listed in 2.4.2, the requirements listed above can be met implementing a Fog layer in the network. The fact that Fog Computing can meet this common requirements of IoT has lead to a surge of many highly featured Fog and Edge solutions like what can be seen in the Chapter 2.4.4.

### 2.4.4 Existing Solutions

**AWS Greengrass** AWS Greengrass is a software built on AWS Lambda and AWS IoT and it is designed for offline operations on the edge of the network. It aims to ease the implementation
State of the Art

of local processing, messaging and data caching across all the things present in the local network. It enables devices to run AWS Lambda functions, data syncing across devices, and secure and intermittent communication.

For this to work, AWS Greengrass comprises two different parts:

- **Greengrass Core (GGC)** - GGC’s function is to manage communication amongst the different things, handling the security of all the operations, and to run the local Lambda functions while interacting with the cloud at the same time. GGC was designed to run on either an ARM or x86 architecture with at least 1 GHz and 128MB of memory.

- **IoT Device SDK** - It is the SDK used for building applications that will eventually run on the devices whose intent is to communicate with the device that will host Greengrass Core. The connection between these two devices is a lan connection and the application’s purpose is to receive data from sensors, obtain and store state information, and MQTT topics subscription.

One of the advantages of Greengrass is the fact it is intelligent enough to only relying on the cloud when there is the necessity for such, for example for the analysis of a big amount of information (search query), and only distributing tasks to other nodes in the local network if it makes sense.

**Azure IoT Edge**  
Microsoft’s Azure IoT Edge is Microsoft’s answer to AWS’ Greengrass, with intent of "extending cloud intelligence to edge devices". It’s an extesion to an already existing product, Azure IoT Gateway SDK, with the inclusion of more features. It’s primary objective is to connect existing devices, not only to the cloud, but to each other, making them smarter in the sense that they not only communicate with the cloud for everything, just relaying information, but choose with who to talk according to the task. At the moment, it has not yet been released being in Preview state and the code is publicly available at Github.

**EdgeX Foundry**  
This project from the Linux Foundation aims to create a common open framework for IoT edge computing development allowing for better interoperability and for a healthy ecosystem in IoT. Its goals include:

- Making EdgeX as the go-to platform for developing for IoT Edge Computing, thus unifying IoT.

- Provide means and encourage providers like AWS and Azure build their ecosystems around the EdgeX platform.

- Certification of EdgeX components with the aim of ensuring compatibility and interoperability.

- Have the necessary tools available for creating EdgeX IoT solutions that will adapt to the ever-changing needs of businesses.
Ensure collaboration with open source projects, standards groups, and make industry alliances that will ensure interoperability and consistency across the field of IoT.

Figure 2.11 shows in which layers of the network EdgeX Foundry acts. It was built with the industrial ecosystem in mind and leverages cloud-native principals, like platform independence and loosely-coupled microservices), but is still architected to meet the needs of IoT edge, like enabling both IP and non-IP communication protocols and management of distributed nodes.

2.5 OpenFaaS

2.5.1 What is?

OpenFaaS is a framework that facilitates the process of creating serverless functions. It acts on the Fog layer, creating a network of devices where the serverless functions can run on. Every function will be turned into a Docker container which will then be turned into a serverless function that can be replicated as demand increases. OpenFaaS will distribute the workload amongst the various devices that belong to the network.

This framework makes it extremely easy to create serverless functions as it handles the deployment, workload balancing, configuration, forwarding of the HTTP requests from the entrypoint to the respective function, as well as simplifying the creation and development of the serverless functions by providing templates where the developer only has to fill in with the function’s source-code. It also provides support for metrics, through the use of Prometheus.
2.5.2 Components

As it can be seen in the Figure 2.12, OpenFaaS works on top of existing technologies and is deeply connected and dependent of the Docker ecosystem.

- **Docker** - Crucial dependency of the OpenFaaS framework. Docker allows for the portability and containerization of the functions, which is what allows for OpenFaaS to deploy the same function across different devices.

- **Docker Swarm or Kubernetes** - OpenFaaS relies on either Docker Swarm or Kubernetes to handle container orchestration. This helps in running the multiple containers across different machines, the scaling up or down or containers, adding and removing containers and load distribution amongst machines. The choice between Kubernetes or Docker Swarm is left to the developer to make, as both of them have their ups and downs.

- **Prometheus** - is a monitoring solution to augment the monitoring and alerting of metrics. It provides a UI interface as well as an API for the developer to keep track of the serverless functions.

On top of these tools, OpenFaaS has implemented two services, the **Function Watchdog** and the **API Gateway**. There is an additional component, that is only present on the developing side, which is the **CL**.
The **Function Watchdog** is an entrypoint that allows HTTP requests to be forwarded to the target process (serverless function) via STDIN. The send the response back to the caller, the serverless function as to write to the STDOUT. The Function Watchdog is responsible for this process of handling the messages between the function’s STDOUT and STDIN, and the caller of the request.

The **API Gateway** is a RESTful micro-service that provides an external route into the functions and is responsible for collecting Cloud Native metrics through Prometheus. Not only this, but it’s also the API Gateway that handles the scaling of functions, by altering the number of replica counts through the Docker Swarm of Kubernetes API. It also has a UI through the browser for simple creation and invocation of the functions.

The **CLI** is a command line interface to aid the developer in the process of creating and deploying a serverless function. It has support for multiple languages and automatically creates a serverless function from templates for Node.js, Python, Go and many others. It acts as a RESTful client for the API Gateway. Any Docker container can be a serverless function with OpenFaaS.

### 2.5.3 Kubernetes vs Docker Swarm

The choice between Kubernetes and Docker Swarm will have a big impact on a large scale. Because these tools are responsible for the orchestration of the containers across the multiple machines that belong to the OpenFaaS network of devices, the choice between one of them will impact both performance and usability.

### 2.5.4 Function Metrics

OpenFaaS provides a series of metrics for each of the serverless functions. This metrics are available on Prometheus, either through the UI or through the API. This are the metrics that are dispensed:

- **gateway_functions_seconds** - Function time taken
- **gateway_functions_seconds_sum** - Sum of the time taken in all of the function’s execution.
- **gateway_functions_seconds_count** - Number of invocations of the function
- **gateway_function_invocation_total** - High invocation total
- **gateway_service_count** - Docker service replicas.

### 2.6 Exploration vs Exploitation and Multi-Armed Bandit

Exploration vs Exploitation is a common decision making dilemma, both in real life and in the computer world. Choosing between a known good option or taking the risk of trying an unknown option in the hope of finding a best result is a choice that we try to balance in the hope of minimizing the total regret(total opportunity loss[Sil]) we face.
State of the Art

If we had access to all the information about the universe in question, we could either brute-force or use other smart approaches to achieve the best results. In this situation, the problem comes from only having incomplete information. In order to make the best overall decisions, we need to simultaneously gather enough about the system and keep the total regret at a minimum. Exploitation will choose the best known option in order to avoid any regret. Exploration will take the risk of choosing one of the less explored options with the purpose of gathering more information about the universe in question, reducing short-term success for long-term success. A good strategy will use both options, exploration and exploitation, to achieve the best results.

The Multi-Armed Bandit is a known problem that exemplifies the Exploration vs Exploitation dilemma. The problem places us with multiple slot machines, each with a different reward probability. Given the setting, the objective is to find the best strategy to achieve the highest long-term reward [Wen18].

2.6.1 Definition

The Bernoulli multi-armed bandit problem can be defined as a tuple of \( \langle A, R \rangle \) in which [Wen18]:

- There are \( K \) machines with a reward probability each, \( \{ \theta_1, \theta_2, ..., \theta_K \} \)
- For each step \( t \) an action \( a \) is taken, resulting in a reward \( r \).
- \( A \) refers to a set of action where each action is linked to a slot machine. The value of the action is the expected reward, \( Q(a) = \mathbb{E}[r|a] = \theta \).
- \( R \) is a reward function, which in this case, the reward is viewed in a stochastic fashion.

The goal is to maximize the total reward, \( \sum_{t=1}^{T} r_t \), or in other words, minimize the regret of not taking the optimal action in every step.

The optimal reward probability \( \theta^* \) of the optimal action \( a^* \) is:

\[
\theta^* = Q(a^*) = \max_{a \in A} Q(a) = \max_{1 \leq i \leq K} \theta_i
\]

2.6.2 \( \varepsilon \)-Greedy

This algorithm will choose the best known action most of the times but it will also explore randomly from time to time. The value of an action is given by [Wen18]:

\[
\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{\tau=1}^{t} r_\tau \mathbb{1}[a_\tau = a]
\]

\( \mathbb{1} \) is a binary indicator function and \( N_t(a) \) represents the total number of times that a given action as been selected, \( N_t(a) = \sum_{\tau=1}^{t} \mathbb{1}[a_\tau = a] \).

In this algorithm, we take the best known action most of the times, \( \hat{a}_t^* = \arg\max_{a \in A} \hat{Q}(a) \), or , with a probability of \( \varepsilon \), we take a random action. The best known action will be taken with a probability of \( 1 - \varepsilon \).
2.6.3 Upper Confidence Bounds

Random exploration might not be the best option because it might lead to trying an action that was previously concluded as bad. One way of avoiding is to give priority to options with a high degree of uncertainty, actions for which there isn’t a confident value estimation yet.

Upper Confidence Bounds (UCB) will translate this potential into a value, the upper confidence bound of the reward value, $\hat{U}_t(a)$ [Wen18]. The true will be below $Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$. $\hat{U}_t(a)$ will vary with the number of trials and a larger sample of trials will result in a smaller $\hat{U}_t(a)$.

With the UCB algorithm the next action to take in the will be:

$$a_t^{\text{UCB}} = \arg\max_{a \in A} \hat{Q}(a) + \hat{U}(a)$$

To estimate the upper confidence bound, if prior knowledge of how the distribution looks like can be discarded, then it is possible to apply "Hoeffding’s Inequality", a theorem that can be applied to any bounded distribution [Sil].

Theorem 1 (Hoeffding’s Inequality) Let $X_1, ..., X_t$ be i.i.d. random variables in $[0,1]$, and let $X_t = \frac{1}{t} \sum_{\tau=1}^{t} X_\tau$ be the sample mean. Then

$$P[\mathbb{E}[X] > X_t + u] \leq e^{-2u^2}$$

Applying Hoeffding’s Inequality to the rewards of the bandit will result in:

$$U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}$$

2.6.3.1 UCB1

To ensure that the optimal action is taken as $t \to \infty$, $p$ can be reduced with each trial [Sil].

In UCB1 algorithm replaces $p = t^{-4}$ resulting in:

$$U_t(a) = \sqrt{\frac{2\log t}{N_t(a)}}$$

The resulting algorithm, UCB1, is as follows

$$a_t = \arg\max_{a \in A} Q(a) + U(a) = \arg\max_{a \in A} Q(a) + \sqrt{\frac{2\log t}{N_t(a)}}$$

2.6.3.2 Bayesian UCB

The previous method, UCB1, does not make any assumptions about the reward distribution $R$, only relying on Hoeffding’s Inequality to make an estimation. Knowing the distribution would allow or better estimates.
In the Bayesian UCB it is assumed that the reward distribution is Gaussian, \( R_a(r) = N(r; \mu_a, \sigma_a^2) \). The action that will give the best result is the action that maximises standard deviation of \( Q(a) \) [Sil]:

![Figure 2.13](image-url)

Figure 2.13: When the expected reward has a Gaussian distribution. \( \sigma(a_i) \) is the standard deviation and \( c\sigma(a_i) \) is the upper confidence bound. The constant \( c \) is a adjustable hyperparameter. [Wen18] [Sil]

\[
a_t = \arg \max_{a \in A} \mu_a + \frac{c \sigma_a}{\sqrt{N(a)}}
\]

where the constant \( c \) is an adjustable hyperparameter.

Bayesian UCB relies on a considerable amount of prior knowledge of the rewards and for it to be accurate [Sil]. Otherwise, it would be too straightforward.

### 2.7 Conclusions

IoT is an ever-growing trend with no end in sight. But despite the size of IoT, the available solutions are still scarce and unstable, with AWS leading the market, not only offering the most mature solutions but also as the vendor that gets to say where the market will go next. To contrary this, there have been many efforts from the open-source community with the aim of stopping fragmentation and of increasing inter-operability among different technologies so that developers have an easier time when developing and also so that there is no vendor solution gaining a monopoly in the market. At the moment, most of these open-source solutions are still in a beta-like state, but they already offer a compelling number of features which makes them a starting point good enough for simpler and non specific solutions.

OpenFaaS is a very innovative tool that is built on top of mature open-source solutions and delivering easy to build FaaS. When applied to IoT, OpenFaaS brings together the concepts of Serverless, IoT, and Fog Computing, being of much interest for the aim of this dissertation.
The Exploration vs Exploitation problem could be adapted for many cases in informatics and to great extent fits this dissertation. The algorithms presented in this section could be applied to improve the accuracy of the expected outcome.
Chapter 3

Problem Statement

3.1 What is the problem?

As stated before in previous sections, not only is expected for the number of IoT devices to grow immensely, both commercially and industrially, but there are already many solutions that allow for serverless functions to be executed remotely. Due to the nature of serverless functions, some of them could perfectly be executed locally, using the joint processing power of the multiple IoT devices. The hardship comes with using this power efficiently, having multiple serverless functions, and knowing where to execute each one, locally or remotely. It is not feasible for each developer to manually analyze performance across the different runtime environments and make a decision where the function should be executed. This is impending the adoption of these concepts in IoT, despite the interest and potential that exists in this evergrowing area. Not only there is a lack of systems making use of serverless on-premises, the majority of the developers in IoT opt for using the cloud for each and every need, disregarding the power that exists locally.

3.2 Proposal

Like what was presented above, there is a lack of practical know-how knowledge available despite there being lots of incentives for it. There are lots of things, but it is not easy to start developing a serverless IoT solution.

Given this, the aim of this project is to create an architecture for serverless IoT platform and to build a proof of concept using existing open-source tools when possible and avoiding proprietary solutions. The platform should:

- Have a serverless cloud solution capable of answering HTTP requests from the things.

- Make use of the local processing power of the multiple IoT devices to create a serverless virtual processing unit on the Fog Layer to answer local requests.
Problem Statement

- Have multiple IoT devices with different functions capable of interacting with both the Cloud and Fog layer to execute different functions.

3.3 Assumptions

Despite the interest and potential of IoT in the industry, this project won’t try to answer the needs of an industrial environment. Instead, it will focus in a smaller and simpler space, like a smart space where there is a less amount of constant traffic of information. A smart space provides a huge amount of different things with different requirements and with different purposes, like getting information out the environment and acting accordingly, answering to user input, acting according to a different set of rules, answering based on cloud-exclusive information, and communication between different things.

The system will also admit that it is working in a safe environment, without the risk of an attack from an unauthorized entity, like a cyber-attack. Despite the need for an increasing amount of security in IoT, it is not the purpose of this project to evaluate or to build a fully safe platform.

Even though this mix of concepts is recent, there are mature solutions that take care of the load management, containerization, replication and clustering of serverless functions across multiple nodes in a network. Because these solutions are applicable to a local network, it is not of this project aim to focus on satisfying those requirements.
Chapter 4

High Level Overview

In this chapter it will be presented a more in technical overview of the general aspects regarding all the project’s components and how they interact with each other.

4.1 Solution description

This project tries to fix the problem stated in Chapter 3 by introducing a proxy between the entity requesting the function execution and the serverless function. This proxy will analyze each function’s past history, by looking at the time taken in past requests and make the decision of which runtime environment\(^1\) should the request be forwarded to, see Figure 4.1. The proxy should be able to decide between the local network of devices and one of the many available servers.

\[\text{Figure 4.1: High level overview of project’s architecture}\]

\(^1\)Runtime environment is the system where the serverless function will be executed, i.e., local network or one of the servers available in the cloud
High Level Overview

In order to improve fault-tolerance, in case of no Internet connection or if one the servers is not available, if the request to the server fails, the proxy should fallback to the local network. This way, even if the request to execute the function is forwarded to the server and fails, the function will still be executed locally.

The proxy is situated inside the local network of IoT devices and will forward the request for a specific function to a gateway which forwards the function execution to one of the IoT devices capable of executing the function. The load management, containerization, replication, and clustering of the serverless functions is not handled by the proxy, but it still has to be aware of the serverless functions installed in the local network or any of the other runtime environments.

### 4.2 Expected results and flow. Use cases

The following examples explain the expected results and decision-making of the proposed solution. The decisions taken by the proxy are based only on previous metrics of the time taken for the runtime environment to execute the function (including network latency).

#### 4.2.1 Forward function execution to the cloud

The use case in Figure 4.2 exemplifies a situation where the requested serverless function is hardware intensive, therefore taking a lot of time to execute locally. Due to the high processing power of the cloud servers, it is beneficial to forward the execution request to one of the cloud servers, even when considering the connection latency. From the multiple available servers, it will opt for the one that is physically nearest (lower latency).

![Diagram of proxy-forwarding a request to the cloud](Figure 4.2: Request for the execution of a demanding function to be executed. The proxy will forward the request to the cloud because due to the high processing power of the cloud server, the function will be executed more quickly. The nearest server was chosen because of latency.)
4.2.2 Forward function execution to the local network

Contrary to the previous case, the Figure 4.3 portrays a scenario where the requested serverless function is very light, being more beneficial to execute the function locally and avoid network latency. Despite the difference in power between the two environments, the previous metrics show that the local environment is capable of satisfying the request more quickly.

![Diagram](image)

Figure 4.3: Request for the execution of a simple, light function to be executed. The proxy will forward the request to be run locally, as there is no benefit in executing the function on the cloud.

4.2.3 Fallback to the local network

The Figure 4.4 depicts a scenario where the proxy first tries to forward the request to one of the cloud servers (because it is more beneficial) but fails in doing so. The proxy then decides to forward the request to the local network, successfully completing the request. There are certain situations where it is more favorable for the function to be executed on the cloud but it could still be executed locally. Because it is not possible to always guarantee a working connection, in these cases, if the connection fails the proxy will fallback to execute function locally, assuring fail redundancy and the reliability of the system.
High Level Overview

Figure 4.4: The request for the function execution to be in the cloud could not be satisfied (e.g., no internet connection). The proxy will then forward the request for it to be executed in the local network.

4.2.4 Manual forward. Bypass the weighting process

It should also be possible for the developer to bypass the weighting process (the evaluation of the different runtime environments) and manually choose where to forward the request. This option should be possible either in the setup process of the function or as an argument of the request for the function execution.

Overall sequence

In a very summarized way, the proxy when receiving a request will first decide, from a list of various runtime environments (the list must include the local network of devices and one or more cloud servers), where to forward the request to execute a serverless function. It will decide based on previous metrics of the time taken for the function to execute in the different runtime environments, and will aim to choose the one with less time taken. It is also possible for the runtime environment to be manually configured in the request options or when setting up the proxy. If it decides to forward the request to the local network, it will just wait for it to execute. If it decides to execute the function in one of the cloud servers, it will make a request to the cloud server for it to execute the serverless function and if this request fails (e.g. because there is no connection to the server), it will then try to execute the function in the local network of devices. After having the response from the function execution, the proxy will answer with the response. A sequence diagram of this process summarized can be seen in 4.5.

4.3 Weighting the runtime environments

In order to make the decision of which runtime environment to forward the function to, there has to be some weighting process that will weight each of the runtime environments and compare
them. This process will gather information about the different runtime environments and then make an accurate estimation of which one is the best choice (less time taken). This is similar to the Exploration vs Exploitation problem presented in 2.6. Therefore, the following algorithms were implemented to handle the weighting process:

- **Greedy** - Has no exploration, simply assigns the weight as the mean average time taken.
- **UCB1** - Uses Hoeffding’s Inequality to balance between exploration and exploitation, but the cumulative regret will still be considerable.
- **Bayesian UCB** - Analyzes the reward distribution to make a very accurate prediction of the weight but requires previous knowledge about the environments.

The developer or device can choose which of the weighting algorithms will be used for that request, in the options, but the default algorithm is UCB1. UCB1 was selected as default because despite Bayesian UCB being better, it requires previous knowledge about the environment, as stated in 2.6.3.2.

### 4.4 Code components

As it can be seen in Figure 4.6, the proposed solution is constituted of two main components, the **proxy** and the **sample_functions**. Each of these components is a package in itself.

#### 4.4.1 Packages

- **proxy** - The main package of the project responsible for all the logic. Deals with the reception of the request for the execution of a function, with the weighting of the environments in which the functions can run (locally or in the cloud in one of the multiple servers), and with the storage and retrieval of all metrics of previous function executions.

- **sample_functions** - The package that contains the serverless functions whose execution is going to be requested to the proxy package. This package is purely a sample with the purpose of simulating and analyzing resource demanding serverless functions (either light or heavy) and could be replaced by any other set of functions. The functions inside this package can be executed on either the local environment or on one of the servers (remote environments).

#### 4.4.1.1 proxy

All the different functions inside the proxy package communicate through HTTP and expect to receive the content as application/json.

- **proxy** - The main function through which all requests go through first. After receiving the list of weights associated to the execution of the requested function in each environment, it
High Level Overview

will choose the environment with the least weight and forward the execution of the serverless function to that runtime environment.

• **weight_scale** - This function will analyze all the collected metrics of the requested function and assign a weight to each runtime environment. It allows more than one algorithm for weight estimation.

• **get_duration** - Retrieves the list of all the collected metrics of a function.

• **insert_duration** - Store the time taken for a function to execute.

• **get_overall_stats** - Function that will return the summarized records of all the collected metrics for each function in each environment. Useful for analysis and evaluation of the results.

![Component diagram of the project](image)

Figure 4.6: Component diagram of the project

4.4.1.2 sample_functions

The serverless functions in this package have the purpose of simulating real serverless functions with different purposes and execution times. The functions are aware of the runtime environment they are being executed on and it is possible for them to answer differently according to this. Here, the different time taken is simulated using a *wait* and using different values for different runtime environments.

• **func_light** - This function answers instantly, there should be no difference between executing the function locally or in the cloud, other than connection latency.

• **func_heavy** - In this function there is a *wait* of 2 seconds if it is executed locally or a *wait* of 1 second if it is executed on the cloud. There should be no difference in the time taken across different cloud servers other than connection latency.
High Level Overview

- **func_super_heavy** - similar to *func_heavy* but here the difference in time is bigger. There is a *wait* of 4 seconds if the function is executed locally or a *wait* of 2 seconds if the function is executed in the cloud.

- **func_obese_heavy** - This is a function that, due to its nature, it can only be executed in the cloud and its execution has been flagged as cloud-only. The proxy will not even try to run the function locally, it will always forward the request to the cloud. Because of this, there is no fallback to run locally.
Figure 4.5: High level sequence the whole process. This diagram sums the decision-making of the project when trying to answer the request for the function execution.
Chapter 5

Technical Implementation

This chapter is dedicated to the presentation of details of a lower level, related to the context and technical implementation of the solution proposed in the previous chapter.

It is going to be presented the technical solutions behind this project and how they were built to fit the desired results, technical detail of how some of the core parts of the system were built, technical decisions that were taken and why they were taken, detailed information about the data that can be given to the system and how that will effect the end-result, adaptations to the weighting algorithms in order to fit the overall solution, and the solution used for data storage that would fit the non-functional requirements of the proposed solution.

5.1 Implementation stack, components and interaction.

5.1.1 Stack

This project was built mostly on top of the existing framework OpenFaaS (see 2.5), because OpenFaaS is not only built on top of mature technological solutions, like Kubernetes and Docker, but also because it is an open-source solution that handles all the technical aspects of creating a serverless function that is to run in across multiple IoT devices in the same local network. The Figure 5.1 has the different components that are part of the solution stack. It is very similar to OpenFaaS stack (2.12), only adding a layer of implementation between OpenFaaS and the serverless functions.
Each component of the proxy is itself a serverless function that is clustered across all the IoT devices, providing reliability, scalability, replicability, and fail redundancy.

The components upon which this project relies handle all the load management, containerization, replication and clustering of the serverless functions at the Fog layer.

- **Docker** - Handles the containerization of the serverless functions.

- **Docker Swarm** - This container orchestration platform handles the clustering, availability, load management, and fault-tolerance across the different serverless functions. **Docker Swarm** was chosen instead of Kubernetes because, despite Kubernetes being considered the most mature solution, given the added complexity of setting up Kubernetes it would not be beneficial for the purpose of this dissertation.

- **OpenFaaS** - Handles replication, setting up the serverless functions and also provides a simple process with templated examples for creating the serverless functions, supporting multiple languages.
5.1.2 Interaction of the different components

The Figure 5.2 gives a more detailed view of technical implementation and how the different components are architected to answer the proposed solution.

As said before, the proxy is constituted of multiple serverless functions, each with its own responsibility, and it’s integrated into OpenFaaS’ local network of IoT devices, together the sample_functions. Before the executing the serverless functions, all requests must be forwarded through the proxy.

For storing the serverless function’s metrics, it is used a mongoDB database, horizontally distributed across the local network of devices.

Outside of the local network of devices, on the cloud, the servers contain an exact copy of the serverless functions, whose execution could be requested remotely.

The Figure A.1 has the detailed sequence diagram of the request process of a serverless function and shows the chronological sequence of how the forwarding operation is processed and when each component of the proxy is used.

5.2 Proxy

5.2.1 Received Input

The proxy expects to receive a HTTP request and the content in the type of application/json. The content has to contain the func, which is the function whose execution is being requested,
and might contain other arguments such as **data** and **options**. **data** is the body of the request to the desired function. **options** is an object that contains options regarding how the proxy should proceed when analyzing the request. In Listing 1 there is a summarized description of how the request can be built.

```json
paths:
  /proxy
post:
  request:
    content:
      application/json:
        schema:
          func:
            type: string
          data:
            type: any
          options:
            type: object
            properties:
              forceCloud:
                type: boolean
              forceLocal:
                type: boolean
              weightAlgorithm:
                type: string
        example:
          func: 'func_heavy'
          data:
            value: true
          options:
            forceCloud: true
            forceLocal: false
            weightAlgorithm: 'bayesian_ucb'
```

Listing 1: Proxy post request. This follows the format of OpenAPI/Swagger Specification (https://swagger.io/specification/). Full JSON schema in Appendix B.1

### 5.2.2 Fault-tolerance and Fallback to a local runtime environment

To improve fault-tolerance, functions that can be executed locally will do so if the attempt to execute them on one of the cloud runtime environments fails to do so. Listing 2 details how the technical implementation of this process was made. A function will only fallback to local execution if it was not configured to be only executed in the cloud (e.g. a function that is not stateless would lead to consistency problems if it was executed locally)
Technical Implementation

```javascript
const request = require('request);

let url = server.url + requestedFunction.path;
let data = req.data;

request({url: url, form: data}, (error, response, body) => {
  if(error || response.statusCode !== 200){
    if(requestedFunction.cloudOnly){
      // CLOUD EXECUTION FAILED AND NO FALBACK POSSIBLE
      ...
    }
    // CLOUD EXECUTION FAILED. FALBACK TO LOCAL NETWORK
    makeLocalRequest(requestedFunction, data);
    return;
  }
  // CLOUD EXECUTION WAS SUCCESSFUL
  ...
});
```

Listing 2: Code example describing the technical implementation of the fallback to the local runtime environment process.

5.3 Function weighting

There is a function responsible for the weighting of the different runtime environments, `weight_scale`. The objective is to gather all the metrics of each environment, namely the duration of execution, and associate each environment with a weight (the cost of the executing the function in that environment). Also, the lower the weight, the better the runtime environment is as an option for executing the function.

The following algorithms/methods were implemented:

- Greedy.
- UCB1
- Bayesian UCB

The implemented algorithms only make decisions based on one metric, the duration of the function execution, and will calculate the weight (cost) associated with executing the requested function in each environment. To calculate the weight, the algorithms will only take into account metrics respecting the requested function. For example, if the requested function is `func_heavy`, the algorithm will not take into account metrics regarding other functions such as `func_light` or `func_super_heavy`.

Each of these algorithms will receive a set of arrays. Each array has all the records of the duration of the execution of a function in the requested environment, see Listing 3. In any of the
algorithms, the array of durations and the calculated weight refers to only one of the functions (as specified in the request)

```json
{
    "londonServer": [1.05690459, 1.20049831, 1.09016582, 1.13609471],
    "frankfurtServer": [1.55398254, 1.31049831, 1.00091233, 1.25388327],
    "local": [2.05690459, 2.20049831, 2.09016582, 2.13609471]
}
```

Listing 3: Set of arrays fed to the weighting algorithms. Each position of the array represents the duration of the execution of the function `func_heavy` in the respective environment

### 5.3.1 Greedy algorithm

This method is very simple but inefficient, as there is no exploration of the different options. It will purely choose the option with the least average duration.

\[
weight(a) = Q(a) = \frac{1}{N_t(a)} \sum_{\tau=1}^{t} r_\tau 1[a_\tau = a]
\]

where \(1\) is a binary indicator function and \(N_t(a)\) is the number of times that the action (environment) \(a\) was taken. \(r_\tau\) is the reward associated to the function execution. In this algorithm, the reward will be the duration of request:

\[
r_\tau = duration_\tau
\]

### 5.3.2 Multi-Armed Bandit: Upper Confidence Bounds

The dilemma of exploring less known runtime environments or exploiting the known better environment is similar to the Multi-Armed Bandit problem, see Section 2.6. Because of this, it was
decided to employ two of the existing algorithms, UCB1 and Bayesian UCB, to efficiently solve this problem with the least amount of regret.

Both the algorithms try to maximize the reward, which is the opposite of minimizing the cost. To adapt the objective to how the algorithms operate, small changes were made.

The reward associate to step \((\tau)\) will be as:

\[
 r_\tau = \frac{1}{\text{duration}_\tau}
\]

This way, the smaller the duration, the bigger the reward, and maximizing the reward will result in the action with the biggest value and therefore the action with the smallest duration.

The \textit{proxy} function will still expect the smaller \textit{weight} to be associated to the best environment to choose. Because both algorithms return the value of the option (the biggest value \(\rightarrow\) best environment), some changes were made:

\[
 \text{weight}(a) = \frac{1}{\text{value}(a)}
\]

5.3.2.1 UCB1

In the UCB1 algorithm, the Hoeffding’s Inequality theorem is applied to ensure smart exploration. The value of the action (environment) will be:

\[
 \text{value}(a) = Q(a) + \sqrt{\frac{2 \log t}{N_t(a)}}
\]

where \(Q(a)\) is the average reward for action \(a\), see 5.3.1. The reward, \(r_\tau\), differs in this algorithm, see 5.3.2.

5.3.2.2 Bayesian UCB

The Bayesian UCB will allow for a better bound estimation, if the knowledge of about the reward distribution is accurate. The Bayesian approach considers the reward distribution as a Gaussian distribution where \(R_u(r) = N(r; \mu_u, \sigma_u^2)\).

The value of an action will be given by the following equation:

\[
 \text{value}(a) = \mu_u + \frac{c \sigma_u}{\sqrt{N_t(a)}}
\]

In this equation, \(\mu_u\) is the median reward, \(\sigma_u\) is the standard deviation, and \(c\) is a hyperparameter.

5.4 Metrics: Storage and retrieval

There was a necessity to arrange some storage facility due to the necessity of storing metrics about the function’s execution across the different environments. These metrics are essential for
weighting algorithms to make an accurate prediction of the best environment where to execute the function.

OpenFaaS provides some metrics through Prometeus but these where not enough and only referred to the execution of function in the *local* environment. Also, some of the metrics, like the duration, were not immediately available, resulting in some delay.

A better alternative for the storage of metrics is needed. The alternative needs to be both fast and reliable to avoid being a bottleneck.

### 5.4.1 MongoDB for logging and metric storage

Each row of stored data contains the following columns:

- **function** - name of the function.
- **duration** - duration (in seconds).
- **environment** - runtime environment where the function was executed.

MongoDB was selected to store the data due to its [Xpl17]:

- **Flexibility** - allowing the addition of new columns or fields without affecting existing rows
- **Scalability** - Because MongoDB scales horizontally, it is the perfect fit for an environment that is made of multiple low-performance devices.
- **Speed** - It’s high-performing for simple queries

### 5.4.2 Connection to the database

Both functions that access to the MongoDB database, `insert_duration` and `get_duration`, operate in the same way. The implementation of these functions was based on the implementation of a MongoDB function by Alexellis (creator of OpenFaaS) \(^1\). Instead of being a normal *node* function, like the other functions in this project, it is a *node8-express* function, which allows for more control over HTTP requests and to keep a connection to the database alive even after a request is over.

The connection pooling has an important role as it allows for multiple requests to be made from just one connection to the database. Establishing a connection to the database for each request would be inefficient and introduce an overhead to the whole process. This way, for each instance of `insert_duration` function or the `get_duration` function, the first request will establish the connection to the database and all the subsequent calls will detect that a connection to the database already exists and will go straight to the connection pool, see Figure 5.3.

\(^1\)https://github.com/alexellis/mongodb-function
Figure 5.3: Multiple instances of the same function accessing the mongoDB database. In the last instance there is one active connection and two shown with dotted lines which are about to be closed due to inactivity [Ell].

5.5 Sample Functions

As specified before in 4.4.1.2, it were developed four sample functions with the purpose of simulating real serverless functions with different hardware demands and execution times. The functions are aware of the runtime environment they are being executed on, by having access to the configuration files.

All the functions will expect to receive a state as an argument and will answer whether the light is on or off (see Listing 4).

\[
\text{if } \text{state} = \text{true} \text{ then}
\]
\[
\text{result} \leftarrow \text{"The light is ON"}
\]
\[
\text{else}
\]
\[
\text{result} \leftarrow \text{"The light is OFF"}
\]
\[
\text{end if}
\]
\[
\text{return result}
\]

Listing 4: Pseudo-code representing what sample functions do

They will also answer with the nodeInfo (Hostname of the machine executing the request), the swarm (Runtime environment’s name defined in the configurations), and with message which
Technical Implementation

refers to the name of the function (light, heavy, super heavy, obese heavy), e.g. the `func_heavy`’s message will be "I was able to achieve this result using HEAVY calculations".

The functions have small differences to simulate different scenarios. `func_light` runs exactly the same in every runtime environment. `func_heavy` and `func_super_heavy` will wait 2 and 4 seconds in the local runtime environment and 1 and 2 seconds in a cloud runtime environment, respectively, before answering the request. `func_obese_heavy` was configured to only be executed on the cloud. These delays were introduced to represent real execution overhead and to simulate the required conditions in the use cases stated in 4.2.
Chapter 6

Experimentation and evaluation

6.1 Environment

In this serie of experiments, the setup was configured with 3 runtimes environment. 1 local virtual machine running all the functions with Lubuntu 16.04, 2 cores and 1 Gb of RAM. There are also 2 cloud servers, one in London and other in Frankfurt, both hosted in a EC2 instance in AWS. They are both t2.micro, running Ubuntu 16.04, with 1 core and 1 Gb of RAM.

6.1.1 Connection latency

Both servers were pinged before the test to verify the connection latency. The experiment is being made in Porto, Portugal. Here are the results:

- London - 71.153ms
- Frankfurt - 52.297ms

6.2 First Experiment: With internet connection

For this experiment both cloud servers were up and running and it were perfomed 99 iterations of requests. In each iteration was requested for every single one of the serverless functions in sample_functions (see 4.4.1.2 and 5.5) to be executed. The system has no knowledge about the environment. The aim here is to identify that it is accomplishing the mentioned use cases 4.2.1, 4.2.2, and 4.2.4

6.2.1 Expected results

It is expected that the proxy will forward requests for func_light to be executed locally, for func_heavy requests to be executed either locally or in the cloud. It will depende on the impact of the connection latency, but generally, the connection latency should be less than 1 second

1https://aws.amazon.com/pt/ec2/instance-types/
Experimentation and evaluation

Table 6.1: Results of 1st experiment.

<table>
<thead>
<tr>
<th>Local</th>
<th>avg. time (s)</th>
<th>count</th>
<th>avg. time (s)</th>
<th>count</th>
<th>avg. time (s)</th>
<th>count</th>
<th>avg. time (s)</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>func_light</td>
<td>0.085602</td>
<td>94</td>
<td>2.111840</td>
<td>20</td>
<td>4.100240</td>
<td>25</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>func_heavy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>func_super_heavy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>func_obese_heavy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>London’s server</td>
<td>3.423012</td>
<td>3</td>
<td>5.741277</td>
<td>10</td>
<td>6.169448</td>
<td>20</td>
<td>3.384546</td>
<td>99</td>
</tr>
<tr>
<td>Frankfurt’s server</td>
<td>0.336398</td>
<td>2</td>
<td>1.253591</td>
<td>69</td>
<td>2.265615</td>
<td>54</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

(difference in time that takes for the function to execute locally and remotely), which means that the expected result is for proxy to choose to forward to one of the cloud servers. \textit{func\_super\_heavy} is expected to be executed in the cloud most of the times, due to the big difference in time taken, and the function \textit{func\_obese\_heavy} should always be executed in the London server because it is configured that way. Apart from \textit{func\_obese\_heavy}, some exploration is expected for each of the different environments and not only exploitation of the runtime environment that the system considers as the best option. Because of the latency verified in 6.1.1, when choosing between one of the servers, it should choose the Frankfurt server, because it is the one with less latency (despite being physically further).

### 6.2.2 Results

The obtained results, see Table 6.1, match the expected results. For \textit{func\_light}, the time taken was so small (less than 1/10 of a second) that proxy immediately converged in the best option. The results for the execution of the function \textit{func\_light} translate the results expected for use case 4.2.2.

For \textit{func\_heavy} and \textit{func\_super\_heavy}, it kept a ratio of exploration vs exploitation of 3/7 and 5/6, respectively, but always choosing the fastest of the cloud servers. The exploration rate also increases with the duration of the execution, meaning that the proxy will look for better options the longer it takes for a function to execute. The results for these two functions correspond to the ones expected in use case 4.2.1.

Also, as expected, \textit{func\_obese\_heavy} had a 100% accuracy, thus matching the expected outcome stated in use case 4.2.4.

### Regarding the unexpected time taken for London’s server

Despite being physically nearer than Frankfurt’s server, the London server took much more time to answer the request than expected and had bigger latency. The issue where one of EC2 machines would start to slow down and take longer to answer requests had already been noticed when developing. In this experiment it ended up being beneficial for the test because the time taken for the function to execute contrasted more between the servers.
6.3 Second Experiment: Without internet connection to the servers

For this experiment both servers were turned off and the internet connection was cut, leaving the system only operational locally. The system still keeps all the knowledge acquired in the previous experiment. The aim here is to identify that it is accomplishing the use case 4.2.3.

6.3.1 Expected results

In this experiment, it is expected for the weighting algorithm to suggest executing the serverless function in one of the cloud servers, because it will lead to a faster execution. Because there is no internet connection, it is expected for the proxy to try to execute the function remotely, fail, and then to fallback to the local runtime environment. In the end, the function should be executed in the local runtime environment leading to the request being answered successfully.

6.3.2 Results

First the function `weight_scale` is queried to know which of the runtime environments the proxy is going to choose (because the proxy chooses the runtime environment with less weight, knowing the weights allow us to know which option the proxy is going to take). Because there is more information about the system, the weight algorithm used was the Bayesian UCB. As it can be seen in the Listing 5, the runtime environment that is going to be choosen is the Frankfurt’s server.

```json
{
  "status": "success",
  "londonServer": 3.5747403999957266,
  "frankfurtServer": 1.1756422708191938,
  "local": 2.090245031544607
}
```

Listing 5

Even though the choosen runtime environment was Frankfurt’s server, because there was no internet connection it had to fallback to execute the function locally in order to complete the request successfully, as seen in Listing 6. The observed results match the ones expected and also the proxy proceeded as stated in use case 4.2.3.
Experimentation and evaluation

6.3.1 Expected results

In this experiment, it is expected for the weighting algorithm to suggest executing the serverless function in one of the cloud servers, because it will lead to a faster execution. Because of this, the mean total time of the request should be smaller in the first 50 requests. After the 50th request, because the internet connection was cut off, the proxy will try to execute the function remotely, fail, then fallback to execute the function locally, resulting in a larger mean total time.

Listing 6: The request was executed locally, as indicated by the key swarm, which is the swarm (runtime environment) where the function was executed. local, is the name given to the local network of devices, as configured when setting up the proxy.

```
{  
  "nodeInfo": "61c20a65b48e ",  
  "swarm": "local",  
  "message": "I was able to achieve this result using HEAVY calculations",  
  "status": "The light is ON"  
}
```

6.4 Third Experiment: Turning off Internet connection in a series of requests

During this experiment, the main purpose is to run a cycle of requests and then turn off the Internet access in the middle of the cycle to observe how this will affect response times. It will be run 99 iterations of requests and in each iteration it will be requested the execution of func_heavy. After request number 50, the Internet connection will be cut off, leaving the system only operational locally. The system will keep all the knowledge gathered in the previous experiments. The aim here is to identify that it is accomplishing the use case 4.2.3 and how results vary throughout.

6.4.1 Expected results

In this experiment, it is expected for the weighting algorithm to suggest executing the serverless function in one of the cloud servers, because it will lead to a faster execution. Because of this, the mean total time of the request should be smaller in the first 50 requests. After the 50th request, because the internet connection was cut off, the proxy will try to execute the function remotely, fail, then fallback to execute the function locally, resulting in a larger mean total time.
6.4.2 Results

The obtained results, illustrated in Figure 6.1, match the expected results. The mean total time of the request when there was Internet connection was $1,71466602$ seconds, and, at iteration number 50, it jumped to $4,253785939$ seconds when the Internet connection was cut off. Despite having no internet connection, the system was still able to complete the request, just with an added delay. The added delay was due to the fact that it had to try to execute the function remotely and also because of the increased time it takes to execute the function locally (2 seconds). The complete list of results gathered from this experiment can be found in A.3.

6.5 Fourth Experiment: Adapting to lag

In this experiment, it is going to be executed a series of requests and after reaching to a while, the Internet connection is going to be purposely slowed to see how the system reacts in situations of lag and slow connection. It will be run 249 iterations of requests and in each iteration it will be requested the execution of func_heavy. After request number 50, the Internet connection will be slowed down (28 kbps UP, 14 kbps DOWN) and the system will continue to be asked to execute the functions. The system will have none of the knowledge gathered in the previous experiments. The aim here is to identify that it is accomplishing the use cases 4.2.1 and 4.2.2, and also that it is capable of adapting to changes in the network.
6.5.1 Expect results

In this experiment, it is expected that the system goes through three phases. In the first phase, in the first 50 iterations, while the connection to the server is working as expected, the system is supposed to gather information about the environment and to converge to the best option (one of the cloud servers).

In the second phase, the Internet connection is slowed down and the execution of function remotely should take longer than the execution of the function locally. In this phase, the system is supposed to still converge to one of the cloud servers but gradually diminishing the frequency in which it chooses the cloud servers as the best option.

After this phase, the system will enter a third phase where the results gathered after the introduction of network lag outweigh the results gathered in the first phase. Here, the system should start to converge to the local network as the best option.

6.5.2 Results

![Figure 6.2: Requests total time throughout the various iterations of the fourth experiment](image)

The gathered results can be observed in Figure 6.2. As expected, the results for the first 50 iterations are the expected results in a standard situation. After the introduction of network lag, we start to observe spikes in the total time it takes for the function to be executed. This spikes refer to the execution of the function remotely. In the first 30/40 requests after the introduction of network lag (iterations 50 to 90), the frequency of requests that are executed remotely is still high. The
frequency starts to diminish from that point on and at around iteration 175 the system starts to choose the local network more frequently than the cloud servers.

Figure 6.3: This figure illustrates the cumulative average duration of the request throughout the various iterations of the fourth experiment. E.g., the value in iteration 50 (2.085604 seconds) is the average duration of the first 50 requests.

Figure 6.3 shows a different perspective of the results, showing the cumulative average duration of the requests throughout the experiment. It can be seen here that around iteration 175 the system changed course and started to converge to the local network as the best option. This marks the point where the system finally adapted to the changes introduced.

The various phases can be seen more easily here, in Figure 6.3. The first phase can be seen from iteration 0 to 50, the second phase from iteration 50 to 175, and the third phase from iteration 175 to 250.

Nevertheless, it took around 125 iterations (from iteration 50 to iteration 175) for the system to adapt. After 50 iterations where the system gathered information that became invalid, it took the system 250% more iterations to adapt to the new conditions. These results show that the system, although capable of adapting, will take a considerable amount of time to adapt to new conditions.
6.6 Points of Failure

6.6.1 Failure to recognize when the local network is overloaded with work

Despite the fact that some of the layers upon which this dissertation was built on top of already handle load management across the network, no balancing of the work is made across the multiple networks. The solution expects for the knowledge to be consistent, and the used algorithms do not account for the fact that the local network might be overloaded, which could easily happen if there are not that many IoT devices in the network.

6.6.2 Time limit and boundaries for local vs remote execution

Using time to analyze where to deploy a function might be a raw approach in some cases, producing theoretically better but impractical results. If a function takes a long time to execute both remotely and locally, it might be more practical to execute it remotely, even if locally it produces faster results. IoT devices usually have low-performance and specifications and after a certain limit, it might not be beneficial to have a device executing a function for such a long time and holding such a large percentage of the processing power of the local network. It is hard to define where that line is and to have a fixed upper bound.

6.6.3 Slowness to adapt

As seen in 6.5, the system is capable of adapting but will take some time to do it. The bigger the amount of time the system has stayed in the previous conditions, the longer it will take to adapt to the new conditions. This could be fixed by limiting the number of previous metrics used to calculate the weight, e.g. only using the last 50 records.

6.7 Summary

The experimentation and results presented in this chapter go in accordance to those expected and satisfy the proposed use cases in 4.2. The developed solution is capable of analyzing the knowledge it has over the ecosystem and will make a decision that will lead to a faster execution time and at the same time explore different options that might lead to better results. Additionally, the developed solution is also capable of detecting failures in the remote execution of the serverless function and solve that problem, executing the function locally and answering the request successfully.

In sum, the proposed solution proved to be capable of answering the demanded use cases and the used approach was fruitful. Nonetheless, there are limitations and some questions that this approach cannot answer, as stated in this chapter. Despite the fact that the proposed solution already reaches a level that is very beneficial for most of the practical applications, further development must be made to reach a more compelling solution.
Chapter 7

Conclusions and Future Work

Several remarks could be made from the analysis of the state of the art of IoT, serverless, fog computing, and exploration vs exploitation, which determined the course and approach to the problem.

The serverless and fog computing overview states the problems and benefits that come from these two rising trends and how to fit the current paradigms in software development. It was given more attention to the aspects that fit the IoT ecosystem and this analysis showed how serverless and fog computing compliment the IoT ecosystem, allowing for easier integration of services to the IoT devices and also enhancing devices to take advantage of the joint potential processing power.

Because the decision between two or more runtime environments is not easy, specially if there is no knowledge about where each serverless function should be executed on for quicker results, the exploration vs exploitation overview was fundamental. Exploring different options but also exploiting the current knowledge of the system is a dilemma in itself and this overview helped understand how similar problems have been approached in different occasions.

Finally, the summary about the OpenFaaS framework helped understand a key point in this project. The OpenFaaS tool fit perfectly in the context of this project, covering the main areas of IoT, serverless, and fog computing, allowing for the union of these three areas using a simple framework that covered all the technical hardships. It allowed us to immediately build on top of the framework and let us to only focus on the aspects that were important to developing the proposed solution, quickly progressing towards the final goal. Because of this, it was important to have a clear understanding of OpenFaaS’ functionality, usage, and limits.

Taking into consideration some assumptions and abstracting from other layers that interacted with this solution, the added layer that we introduced in the process can make relevant decisions regarding where each serverless function should be executed, improving overall efficiency of the system and providing the developers with the opportunity of using local processing power instead of having to fully rely on the cloud processing power.
Conclusions and Future Work

The exploration vs exploitation techniques applied were important because they rapidly assign each runtime environment with a weight which will be used to make the best choice. As stated before, mixing more than one technique would be more beneficial in the long term to achieve less total regret but the chosen techniques already give an insightful result that enables us to draw conclusions.

A functional prototype was made, which can serve as a proof-of-concept and as baseline for further development. There had to be made some simplifications and assumptions which place the developed prototype far from its potential. A lot more work and improvement could be made in these fields to obtain better results and real life functionality. In order to keep record of the previous execution metrics, a horizontal storage was added to the local network of devices. Because the provided metric storage was not enough for the desired purpose, it required for the developed solution to have its own storage method.

During development of the solution, many conclusions were taken, regarding not only the immediate focus of this project, but also about surrounding topics. In the beginning it seemed obvious to also develop the solution as a group of serverless functions, taking the advantage of the pros of serverless and also tightly connecting the developing solution with the targeted serverless functions. But as the project grew, the architecture and interaction between the multiple functions that constituted the project resembled more and more a monolithic application, which resulted in the complexity of a serverless architecture without some benefits of a serverless architecture. Ultimately, the choice between a serverless application might not be that simple and even when the choice seems obvious, in the long term that might not be the case.

Nevertheless, the serveless paradigm, specially when applied together with fog computing, is still very new and the possibilities are enormous. Developing this project gave immense prospect about how this field can deeply impact IoT, both commercially and industrially, bringing data processing nearer to the end-user, augmenting responsiveness, stability, fault-tolerance, security and efficiency.

It was also insightful to observe that despite the evergrowing potential and financial prospects, the leading technologies that serve as baseline for development in the serverless field are mostly open-source, assuring transparency, security, and confidence in the built solutions.

7.1 Contributions

Within the fields surrounding this dissertation, there is lot of uncharted territory and unknown aspects. The choice between a serverless architecture and a monolithic one is still not clear in all cases and adding these concepts and infusing it with IoT and Fog Computing is a very new area. Because of this, the principal contributions of this dissertation were:

- Inovative approach to the mix of IoT, fog computing and serverless. There is not much work in this mix of fields and this approach is both inovative and unseen.
Conclusions and Future Work

• Enabling serverless both locally and remotely. The developer creating the serverless functions no longer has to actively choose where to deploy the functions to, it is possible to automate that process and still keep total regret at a minimum.

• The ability to run serverless functions locally even if the connection to the server fails and improvement of fault-tolerance in systems.

• Gathering of existing knowledge, tools, and platforms suitable for developing solutions in the areas of IoT, serverless and fog computing.

• Development of a functional prototype built on top of widely used and mature solutions that can serve as inspiration for future and better solutions.

All the code and work made during this dissertation is openly available at https://github.com/444Duarte/serverless-iot.

All the contributions of this dissertation will be summarized in the paper Dynamic Allocation of Serverless Functions in IoT Environments, to be submitted to the 16th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing [PDF18].

7.2 Main Difficulties

During this dissertation, some problems occurred that caused some distress and troubled the overall solution.

• This fusion of different areas like IoT, fog computing and serverless is new and there is no clear path or approach. The uncertainty in what path to follow led to some mistakes and to a slower pace sometimes.

• Not many, but some of the tools used to develop the functional prototype are also recent and poorly documented which made it harder to build the functional prototype.

• Adapting the exploration vs exploitation algorithms to our problem. The algorithms aim to maximize and the purpose was to minimize the cost.

• There were not enough resources to develop a proper local network of IoT devices and to create a truly authentic ecosystem that would better simulate a real life environment.

7.3 Future Work

Despite the efforts made, there is still many improvements that could be made to the developed work:

• Stateful - Introducing statefulness and consistency across serverless different runtime environments is definitely a challenge but the end result would be of utmost usefulness and importance.
Conclusions and Future Work

- Analyzis of other metrics other than time (e.g. energy consumption, CPU cycles, memory usage, network usage) - The time taken is not the only metric that is important when choosing where to execute a serverless function. Analyzing the impact of other metrics would also be applicable in other different contexts and is something to take into consideration. A solution mixing different points of view that analyzed different metrics to choose the overall best result is also a possibility that would be of great use.

- Static analysis to verify complexity of the function - A static analysis of the function could largely improve the exploration vs exploitation problem, allowing the system to start with some knowledge about the complexity of the function, diminishing the total regret.

- Integrate metrics with Prometheus - This is a practical quality of life improvement and only regards the technical implementation but would simplify the analysis process.

- Replace the functional prototype with a viable real life solution that could augment serverless development - A viable real solution would certainly be appreciated would deeply impact the field and advance both research in these areas but also the development of real applications that took advantage of these concepts and technologies.
References


[Apa] Apache Software Foundation. Apache OpenWhisk is a serverless, open source cloud platform.


REFERENCES


[kn:d] Fog computing’ could be more important than the cloud.


[kn:g] Prepare for serverless technology with these five tips.


REFERENCES


REFERENCES
Appendix A

Additional Content

A.1 Comparison between different serverless platforms and the languages that they support

The table can be found as Table A.1

A.2 Full sequence diagram of a function request through proxy

The diagram can be found as Figure A.1

A.3 Third Experience’s Results Complete

<table>
<thead>
<tr>
<th>Request total time</th>
<th>Mean total time with Internet connection</th>
<th>Mean total time without Internet connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1.707959</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>2 1.60958</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>3 1.644058</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>4 1.62977</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>5 1.799518</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>6 1.636532</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>7 1.700875</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>8 1.638327</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>9 1.552673</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>10 1.690422</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>11 1.547895</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>12 1.619954</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>13 1.622696</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>14 1.620614</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>15 1.814774</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>16 1.639204</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>17 1.695879</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>18 1.610265</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td>19 1.812435</td>
<td>1.71466602</td>
<td>4.253785939</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>20</td>
<td>1.644257</td>
<td>1.71466602</td>
</tr>
<tr>
<td>21</td>
<td>1.618132</td>
<td>1.71466602</td>
</tr>
<tr>
<td>22</td>
<td>1.628804</td>
<td>1.71466602</td>
</tr>
<tr>
<td>23</td>
<td>1.68762</td>
<td>1.71466602</td>
</tr>
<tr>
<td>24</td>
<td>1.601617</td>
<td>1.71466602</td>
</tr>
<tr>
<td>25</td>
<td>1.727228</td>
<td>1.71466602</td>
</tr>
<tr>
<td>26</td>
<td>1.704222</td>
<td>1.71466602</td>
</tr>
<tr>
<td>27</td>
<td>1.709243</td>
<td>1.71466602</td>
</tr>
<tr>
<td>28</td>
<td>1.737457</td>
<td>1.71466602</td>
</tr>
<tr>
<td>29</td>
<td>1.817899</td>
<td>1.71466602</td>
</tr>
<tr>
<td>30</td>
<td>1.736112</td>
<td>1.71466602</td>
</tr>
<tr>
<td>31</td>
<td>1.628152</td>
<td>1.71466602</td>
</tr>
<tr>
<td>32</td>
<td>1.701154</td>
<td>1.71466602</td>
</tr>
<tr>
<td>33</td>
<td>1.749505</td>
<td>1.71466602</td>
</tr>
<tr>
<td>34</td>
<td>1.678419</td>
<td>1.71466602</td>
</tr>
<tr>
<td>35</td>
<td>1.709836</td>
<td>1.71466602</td>
</tr>
<tr>
<td>36</td>
<td>1.664774</td>
<td>1.71466602</td>
</tr>
<tr>
<td>37</td>
<td>1.736111</td>
<td>1.71466602</td>
</tr>
<tr>
<td>38</td>
<td>1.731214</td>
<td>1.71466602</td>
</tr>
<tr>
<td>39</td>
<td>1.635211</td>
<td>1.71466602</td>
</tr>
<tr>
<td>40</td>
<td>1.668489</td>
<td>1.71466602</td>
</tr>
<tr>
<td>41</td>
<td>1.584524</td>
<td>1.71466602</td>
</tr>
<tr>
<td>42</td>
<td>1.629688</td>
<td>1.71466602</td>
</tr>
<tr>
<td>43</td>
<td>1.708243</td>
<td>1.71466602</td>
</tr>
<tr>
<td>44</td>
<td>1.624885</td>
<td>1.71466602</td>
</tr>
<tr>
<td>45</td>
<td>1.647711</td>
<td>1.71466602</td>
</tr>
<tr>
<td>46</td>
<td>1.723096</td>
<td>1.71466602</td>
</tr>
<tr>
<td>47</td>
<td>1.721528</td>
<td>1.71466602</td>
</tr>
<tr>
<td>48</td>
<td>1.716368</td>
<td>1.71466602</td>
</tr>
<tr>
<td>49</td>
<td>1.705236</td>
<td>1.71466602</td>
</tr>
<tr>
<td>50</td>
<td>3.563136</td>
<td>1.71466602</td>
</tr>
<tr>
<td>51</td>
<td>3.509016</td>
<td>1.71466602</td>
</tr>
<tr>
<td>52</td>
<td>3.65066</td>
<td>1.71466602</td>
</tr>
<tr>
<td>53</td>
<td>3.57584</td>
<td>1.71466602</td>
</tr>
<tr>
<td>54</td>
<td>3.526258</td>
<td>1.71466602</td>
</tr>
<tr>
<td>55</td>
<td>5.514042</td>
<td>1.71466602</td>
</tr>
<tr>
<td>56</td>
<td>3.455722</td>
<td>1.71466602</td>
</tr>
<tr>
<td>57</td>
<td>5.581585</td>
<td>1.71466602</td>
</tr>
<tr>
<td>58</td>
<td>3.59371</td>
<td>1.71466602</td>
</tr>
<tr>
<td>59</td>
<td>3.624373</td>
<td>1.71466602</td>
</tr>
<tr>
<td>60</td>
<td>5.552203</td>
<td>1.71466602</td>
</tr>
<tr>
<td>61</td>
<td>3.491006</td>
<td>1.71466602</td>
</tr>
<tr>
<td>62</td>
<td>5.469227</td>
<td>1.71466602</td>
</tr>
<tr>
<td>63</td>
<td>3.551192</td>
<td>1.71466602</td>
</tr>
<tr>
<td>64</td>
<td>5.653822</td>
<td>1.71466602</td>
</tr>
<tr>
<td>65</td>
<td>3.542799</td>
<td>1.71466602</td>
</tr>
<tr>
<td>66</td>
<td>3.538493</td>
<td>1.71466602</td>
</tr>
<tr>
<td>67</td>
<td>5.587518</td>
<td>1.71466602</td>
</tr>
<tr>
<td>68</td>
<td>3.592968</td>
<td>1.71466602</td>
</tr>
</tbody>
</table>
A.4 Fourth experience
<table>
<thead>
<tr>
<th></th>
<th>AWS Lambda</th>
<th>Google Cloud Functions</th>
<th>Microsoft Azure Functions</th>
<th>IBM OpenWhisk</th>
<th>OpenWhisk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batches</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binaries</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Javascript</td>
<td>X</td>
<td>X</td>
<td>X (GA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C#</td>
<td>X</td>
<td>X (GA)</td>
<td>X (Preview)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F#</td>
<td>X (GA)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>X</td>
<td>X (Preview)</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>X</td>
<td>X (Experimental)</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Powershell</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Batch (.cmd, .bat)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bash</td>
<td>X (GA)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Batch (.cmd, .bat)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Powershell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: Comparison between different serverless platforms and the languages they support.
Figure A.1: Full sequence diagram of a function request through proxy
Additional Content
Appendix B

JSON Schemas of the proxys internal functions

B.1 JSON Schema of the request to the proxy function

```
{
    "$id": "http://example.com/example.json",
    "type": "object",
    "definitions": {},
    "$schema": "http://json-schema.org/draft-07/schema#",
    "properties": {
        "func": {
            "$id": "/properties/func",
            "type": "string",
            "title": "The Func Schema",
            "description": "This parameter is used to detail the name of the function that is being requested",
            "default": "",
            "examples": [
                "func_light", "func_heavy", "func_super_heavy", "func_obese_heavy"
            ]
        },
        "data": {
            "$id": "/properties/data",
            "type": "object",
            "description": "Parameter that is going to be forwarded as the body of the request to the desired function"
        },
        "options": {
            "$id": "/properties/options",
            "type": "object",
            "description": "Additional options of the request",
            "properties": {
                "forceCloud": {
                    "$id": "/properties/options/properties/forceCloud",
                    "type": "boolean"
                }
            }
        }
    }
}
```
B.2 JSON Schema of the request to the weight_scale function

```json
{
   "$id": "http://example.com/example.json",
   "type": "object",
   "definitions": {},
   "$schema": "http://json-schema.org/draft-07/schema#",
   "properties": {
      "func": {
         "$id": "/properties/func",
      }
   }
}
```
JSON Schemas of the proxys internal functions

"type": "string",
"title": "The Func Schema ",
"default": "",
"examples": [ 
    "func_heavy"
]
},
"query": { 
    "$id": "/properties/query",
    "type": "string",
    "title": "The Query Schema ",
    "default": "",
    "examples": [ 
        "ucb1"
    ]
} }